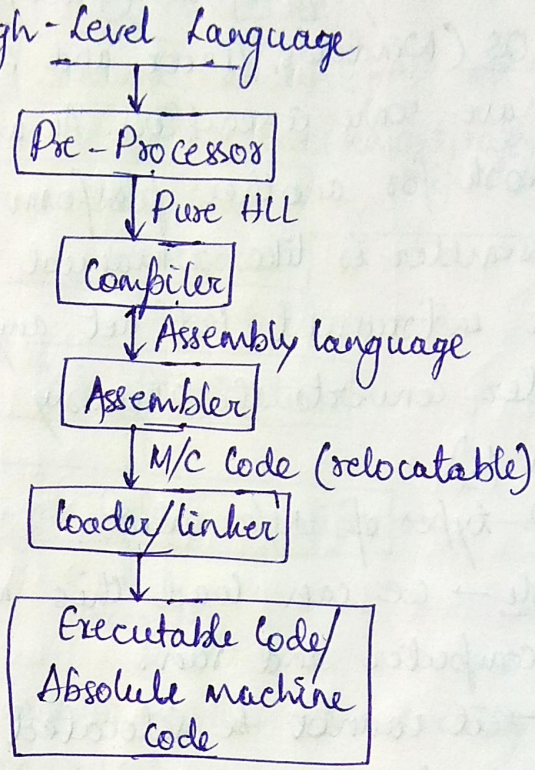
Compiler Design

The main purpose of compiler is to convert high-level language to low level language.

Stages of program written in high-level language converted to low-level language.



(i) High-Level language → it contains files such as #include <>, #define, which is removed by the Pre-Processor by including the file (file inclusion).

(ii) Pre-processor → it substitutes the file in the source program written by us by file inclusion.

NOTE: #define is a macro expansion function which is a small function for which we really don't call the actual function. Main function calling is overhead so we call the small functions.

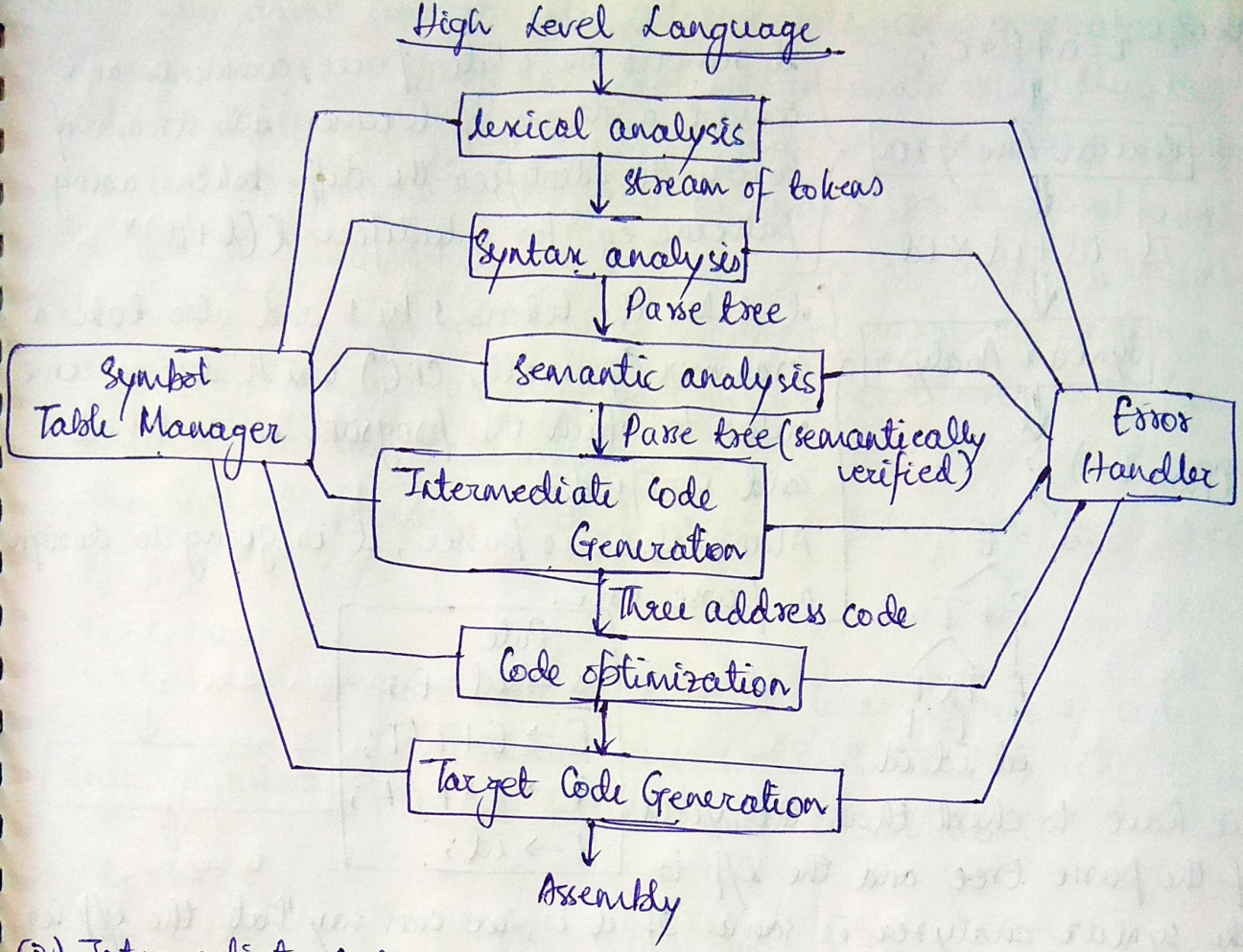
(iii) Compiler → Once the program reaches the compiler, it will be in pure form which means it will not contain any # lines. The # lines are also called pre-processor directives, ie, they direct

the pre-processor about what to do.
The compiler takes the HLL and converts it into assembly lang.
Assembly lang. is not entirely 0s and 1s and not entirely HLL.
It is an intermediate.

(iv) Assembler → for every platform (intel, amd, motorola) and on top of it, the OS (Windows, Linux and 64bit/32bit), we are going to have some assemblers. An assembler for 1 platform will not work for another platform.
Therefore, assembler is like a manual casing which we can operate one instrument but not any other instrument.
The assembler converts the assembly language into machine code (0s and 1s).
There are 2 types of M/C codes:-
(i) Relocatable → we can load this M/C code at any pt. in the computer and run.
(ii) Absolute → it cannot be relocated during execution of computer program and is loaded into computer fixed storage location at each use.

(v) Loader/Linker → It will take the relocatable M/C code and convert it into absolute code.

(8) Phases of compiler:-
(i) Lexical analysis → In this phase, the lexical analyzer will take the program, read it and convert the program into tokens.
(ii) Syntax analysis → the stream of tokens is given to the syntax analyzer (parser) which is going to convert it into a parse tree.
(iii) Semantic Analysis → The parse tree is given to semantic analyzer which verifies whether it is valid or not.



(iv) Intermediate Code generator → The semantically verified parse tree is given to ICG which generates 3 address code.

(v) Code Optimizer → The 3 add. code is given to the code optimizer. The main purpose of code optimizer is to reduce the size of the program or to reduce the no. of lines in 3 add. code

(vi) Target Code Generator → The reduced/optimized code is given to target code Generator which is finally going to generate the assembly code.

All the phases of the entire compiler software module is going to take support of the Symbol Table Manager. and the phases will also take help of Error handler.

Example to show the working of phases:-

x = a + b * c; // Assume as the source program.

Now,

$x = a + b * c;$

⇓

Lexical Analyzer

⇓

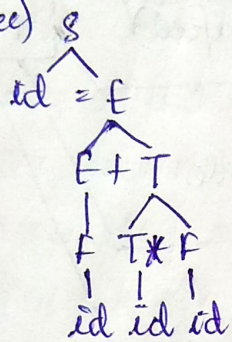
$id = id + id * id$

⇓

Syntax Analyzer

⇓

(Parse Tree)



it removes the white spaces, comments and convert a stream of lexemes into stream of tokens. It identifies the diff. tokens using patterns. eg. for identifier $l(l+d)^*$

it takes the tokens 1 by 1 and also takes a grammer (generally CFG) which defines some rules to which the program is matched and verified.
Also called as parser, it is going to design a parse tree.

Rule
$S \rightarrow id = E;$
$E \rightarrow E+T / T;$
$T \rightarrow T*F / F;$
$F \rightarrow id;$

We have to check that the yield of the parse tree and the i/p to the syntax analyzer is same. If it is, we can say that the i/p is according to the format we provided.

Parse Tree

⇓

Semantic Analyzer

⇓

(Parse Tree Verified semantically)

⇓

Such as left hand side has to be a variable name, here, it cannot be a digit or ~~an id~~ function name or an array name. The variable name should be compatible with the type of right hand side. These kind of type checkings are done.

Intermediate Code Generator

⇓

$t_1 = b * c;$

$t_2 = a + t_1;$

$x = t_2;$

it generates intermediate codes. There are many intermediate code types but the most popular is 3 add. code, in which there are only 3 add. at max. in every line, hence the name.

NOTE: Why do we convert into intermediate code generator?

→ If we convert a program into intermidiate code, then the intermediate code can be converted into m/c language in the last 2 phases which are depended on the platform. So, till the ICG, whatever m/c we run the compiler on, it is same for all, which means if we want to design a new compiler for a new platform, we need not start from scratch. We can take all phases till the ICS and change the last 2 phases, so that new compiler can be formed.

$t_1 = b * c$
$t_2 = t_1 + a;$
$x = t_2;$

⇓

Code Optimizer

⇓

$t_1 = b * c;$
$t_2 = a + t_1;$

⇓

Target Code Generator

⇓

mul R_1, R_2	Suppose,
add R_0, R_2	$a \rightarrow R_0$
mov R_2, x	$b \rightarrow R_1$
	$c \rightarrow R_2$

we reduce the no. of lines by use of code optimizer. Here, we dont need 3 lines to do the job, only 2 lines could do it.

the o/p of code optimizer is given to TCG. The main purpose of TCG is to write the code which assembler can understand

NOTE: Most of the jobs are done by the syntax analyzer although we have so many phases. Therefore, syntax analyzer is the heart of the compiler.

Till the phase ICG, it is also called as front-end, the remaining part is called back-end (CO and TCG). Theoritically, the entire front end is implemented together and then the back end.

For implementing the compiler, there are various tools available. For lexical analyzer, there is a tool called lex. For syntax analyzer, we have Yacc (Yet Another Compiler Compiler).

Yacc is the standard parser generator for the Unix operating system. An open source program, yacc generates code for the parser in the C programming language.
LANCE is implementation of front-end of the compiler which will give intermediate code and then we can implement backend on our own, depending on the platform.

Introduction to Lexical Analyzer and Grammar :-

(i) This is the only phase which reads the text, character by character. After this phase, in place of every lexeme, we will see tokens.

(ii) The main responsibilities of lexical analyzer are :-
(a) it will eliminate the comments, white spaces
(b) only at this phase, since we are reading char by char we know what is the line no. of each lexeme.
Therefore, syntax errors can be identified at which line it is occuring by lexical analyzer.
If there are any errors, it is going to show up.

eg-(i) int | max | (| x | , | y |) | { |
~~int~~ | int | x | , | y | ; |
| /* find max of x and y */ | → removed
{ |
| return | (| x | > | y | ? | x | : | y |) | ; |
} |
} |

No. of tokens = 27

(ii) | printf | (| " %d Hai " | , | & | x |) | ; | No. of tokens = 8

NOTE : The entire string const. is a single token.

Syntax analyzer / Parser :-

To the syntax analyzer, we have to give the grammar first and then the i/p. The i/p will then be checked according

to the grammar.

Grammar → it is represented as $G = (V, T, P, S)$ where,

V = variables,
T = Terminals,
P = productions,
S = start symbol.

Suppose for grammar,

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

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

$\therefore V = \{E\}$
$T = \{+, *, id\}$

Generate $id + id * id$.

$\therefore$ **Left-Most Derivation**

(i)
$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

(ii)
$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow id + \cancel{Ed} * E$
$\Rightarrow id + id * \cancel{id}$
$\Rightarrow id + id * id$

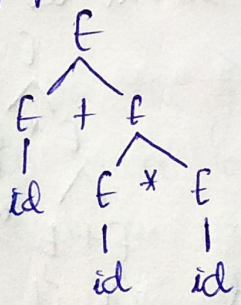
Right-Most Derivation

(i)
$E \Rightarrow E + E$
$\Rightarrow E + E * E$
$\Rightarrow E + E * id$
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

(ii)
$E \Rightarrow E * E$
$\Rightarrow E * id$
$\Rightarrow E + E * id$
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

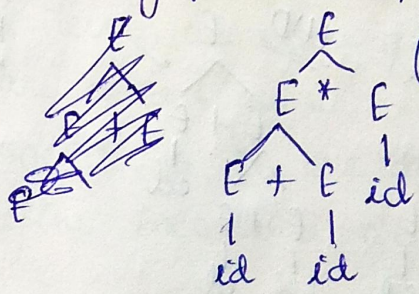
(iii) Using parse tree

$(2+(3*4))$
$= 14$
right



and

~~(iii) Using parse tree,~~



$((2+3)*4)$
$= 20$
wrong

NOTE: If we get more than 1 RMD, LMD or parse tree, we can say that the grammar is ambiguous. Ambiguous, here means that, if we get 2 parse trees, the parser will get confused about which to generate or which is right.

Parsers do not allow ambiguous grammar, so we have to convert it into unambiguous grammar.

There are no methods to find out ambiguity in the grammar and so we have to try hit and trial way to do so. Moreover, there are no algorithm to convert ambiguous grammar to unambiguous grammar.

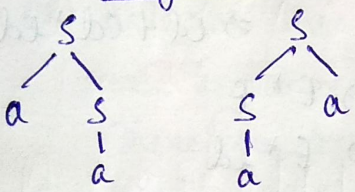
So, in order to find out ambiguity in grammar, we make use of parse trees. We take the grammar and a ~~tal~~ string and see if there are more than one parse tree for the string possible.

For eg (i) $S \to aS / Sa / a$

$w = aa$

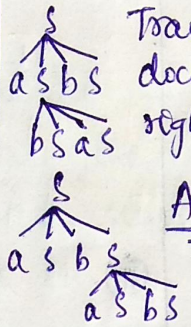
$S \to aS \to aa$ $\quad$ $S \to Sa \to aa$

Ambiguous



(ii) $S \to aSbS / bSaS / \epsilon$

$w = abab$

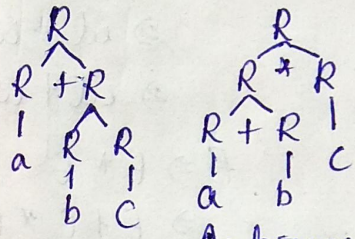


Travel top to down, left to right.

Ambiguous

$R \to R+R / RR / R^* / a / b / c$ represents all regular expressions.

$w = a+b*c$



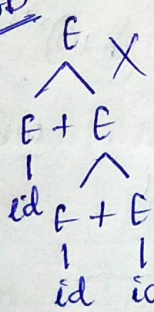
Ambiguous

Ambiguous grammars and making them unambiguous:-

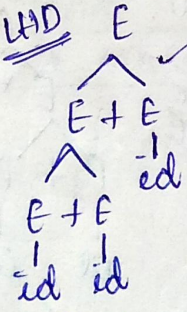
for ex. $E \to E+E / E*E / id$

$str1 = id + id + id$

RHD



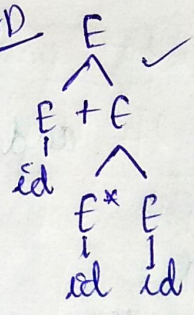
LHD



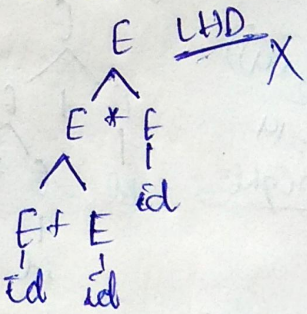
∴ 2 parse trees (Ambiguous)

$str1 = id + id * id$

RHD



LHD



∴ 2 parse trees (Ambiguous)

NOTE: For same precedence operators, we have to see the associativity of the operator. As for here, str1 has all '+' operator which is left associative.

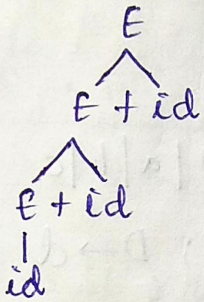
For different precedence operators, we have to locate the

higher precedence operator further away from the root than the lower precedence operator. So that, the higher precedence operator is evaluated first.

Therefore, we have to take care of associativity and precedence.

Now, for same precedence operator, we have to restrict the grammar to grow only in 1 direction depending on the associativity of the operator.

∴ str1, $E \rightarrow E + id \mid id$; We have here, left ~~most~~ recursion.



In left recursion, the left most symbol in RHS = LHS.

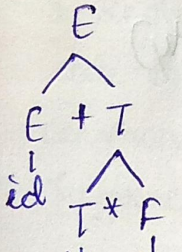
∴ If the grammar is left recursive, than the operator will be left associative.

And now, for the precedences, we define the grammar to be of different levels.

∴ str2, $E \rightarrow E + T \mid T$; Here, we define left recursive because every operator here is left associative and we are defining two levels in such a way that once we reach the second level, we never generate '+'.

$T \rightarrow T * F \mid F$

$F \rightarrow id$



Now, the grammars are unambiguous.

Therefore, in order to define associativity, we are defining recursion and in order to define precedence we are defining levels.

str3 = $(2^{(3^2)})$, This is right associative, therefore, the tree should grow rightwards using right recursion.

∴ $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow G \uparrow F \mid G$

$G \rightarrow id$

Here, $T = \{+, *, \uparrow\}$

highest pre. = ↑

lowest " = +

$\boxed{+ < * < \uparrow}$

eg. If we are having boolean expressions,

bExp → bExp or bExp | bExp and bExp | not bExp | True | False

Here, 'not' has the highest precedence than 'and' and then 'or'. 'or' and 'and' are left associative and there is no associativity for 'not' as it is unary operator.

∴ E → E or F | F ; F → F and G | G ; G → Not G | True | False

For regular expression,

R → R+R | RR | R* | a | b | c

Grammar, E → E+T | T ; T → TF | F ; F → F* | a | b | c

eg. (i) A → A$ B | B ; B → B# C | C ; C → C@D | D ; D → d

<u>Associativity</u> → $ ·> $; # ·> # ; @ ·> @

<u>Precedence</u> → $ <· # <· @

eg. (ii) E → E*F | F+E | F ; F → F-F | id

<u>Associativity</u> → * ·> * ; + <· + ; - (no associativity)

<u>Precedence</u> → + ≐ * ; - (highest precedence)

∴ + ≐ * <· -

Therefore, it is an unambiguous grammar.

Elimination of left recursion and left factoring the grammar:

Recursion

Left recursion

A → Aα / β

A, A, A

A α, β, A()

A α, {

A α, A()

β, α

(βα*) }

Right Recursion

A → αA / β

A()

{

α

A();

} α*β

<u>NOTE</u>: Left recursion is going to fall into infinite loop as it is going to call itself 1st and then do some work. While in right recursion, some work is done and then the func. calls itself.

NOTE: There are 2 types of parser, out of which the top-down parser does not work with left recursion. So, we have to eliminate left recursion without changing the language. ie, here, $\beta\alpha^*$.

$\therefore$ $A \rightarrow \beta\alpha^*$, to be generated,

$$\boxed{A \rightarrow \beta A' \; ; \; A' \rightarrow \epsilon / \alpha A'} \Longleftrightarrow \boxed{A \rightarrow A\alpha / \beta}$$

Right recursive Grammar equi. Left recursive

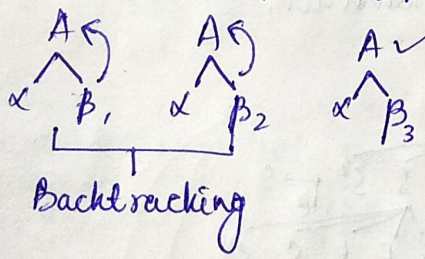
eg. (i) $\underset{A}{E} \rightarrow \underset{A}{E}\underset{\alpha}{+T} / \underset{\beta}{T}$ $\therefore E \rightarrow TE' \; ; \; E' \rightarrow \epsilon / +TE'$	(ii) $\underset{A}{S} \rightarrow \underset{A}{S}\underset{\alpha}{0S1S} / \underset{\beta}{01}$ $\therefore S \rightarrow 01S' \; ; \; S' \rightarrow \epsilon / 0S1SS'$
(iii) $S \rightarrow (L) / x \; ; \; \underset{A}{L} \rightarrow \underset{A}{L}\underset{\alpha}{,S} / \underset{\beta}{S}$ $\therefore L \rightarrow SL' \; ; \; L' = \epsilon / ,SL'$	(iv) $A \rightarrow A\alpha_1 / A\alpha_2 / A\alpha_3 \ldots\ldots / \beta_1 / \beta_2 / \ldots$ $\therefore A \rightarrow \beta_1 A' / \beta_2 A' / \beta_3 A' / \ldots\ldots$ $A' \rightarrow \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \ldots\ldots / \epsilon$

Deterministic and non-deterministic Grammar:

Non-Deterministic Grammar → In this case, we have many options on a single symbol. One or more productions in the right hand side is having something common in the prefixes. Due to this common prefixes, there is occurence of backtracking.

Suppose, $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$ and we have to derive '$\alpha\beta_3$'.

$\therefore$



NOTE: This backtracking occurs also due to, we are making decision of which prod. to choose without seeing much of the input.

Therefore, here we are making the decision here, only by seeing 'α'. If we would make the decision by seeing 'β_3' then there would be no problem. So, we need to postpone the decision making process until we see β_3. It is also called as left ~~tracking~~ factoring procedures or eliminating non-determinism.

Left factoring simply includes the common prefixes taken as common and write the remaining part separately.

For here, $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$, —(i)

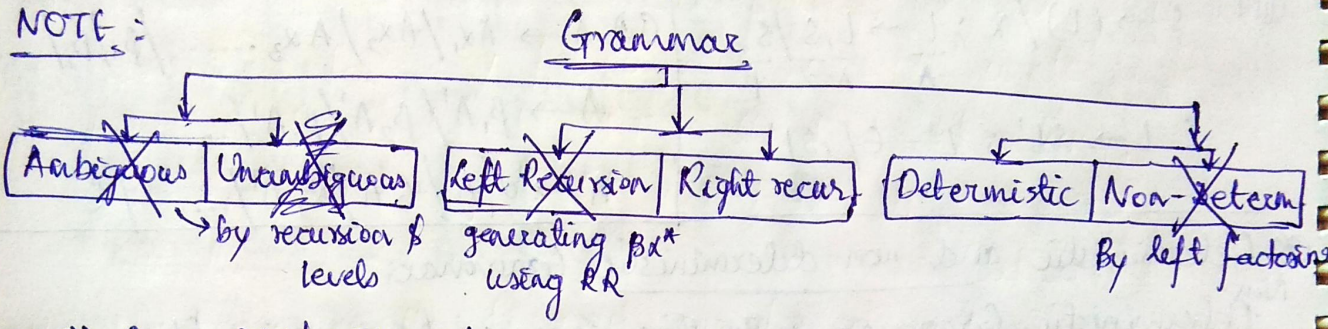
$A \to \alpha A'$; $A' \to \beta_1 | \beta_2 | \beta_3$ —(ii)

So, here we have postponed the decision making to the next part

So, here (i) is non-deterministic grammar for which we have proposed left factoring method to convert it to deterministic grammar.(ii)

The problem with non-deterministic grammar is that most of the top down parsers will fail and will have to backtrack.

NOTE :



eg.(i) $S \to iEtS | iEtSeS | a$; $E \to b$

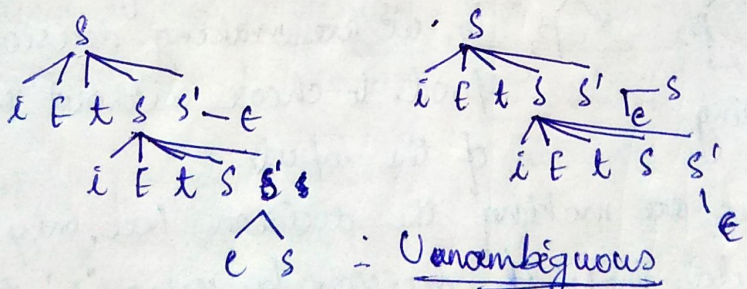
Converting, $S \to iEtSS' | a$

$S' \to \epsilon | eS$

$E \to b$, deterministic grammar.

Now checking for ambiguity in grammar,

eg. $w = iEtiEtSeS$



: Unambiguous

Therefore, eliminating non-determinism or applying left factoring does not eliminate ambiguity. Ambiguity will always be there if you convert ambiguous grammar into deterministic grammar.

But the reason for ambiguity here is not non-determinism. It is due to the no. of 'if' statements in the word is more than 'else'. So, we have to decide to which 'if' is the 'else' associated to. General rule we follow is, always 'else' has to be associated to the closest if.

So, ambiguity is due to the inherent properties of the language

<u>NOTE</u>: In non-deterministic removal, the common prefixes should be present for the same variable.

eg. $S \to \underline{a}Sab \mid \underline{a}Sbb \mid \underline{a}abS$; $A \to \underline{a}S$; $B \to \underline{a}B$

(Valid) (Invalid)

eg. Converting non-deterministic grammar to deterministic grammar.

(i) $S \to \underline{aS}bS \mid \underline{aS}aSb \mid abb \mid b$

~~$S \to aSS' \mid abb \mid b$~~

~~$S' \to SbS \mid aSb$~~

~~$S \to aS'' \mid b$; $S'' \to SS' \mid abb$~~

$S \to aS' \mid b$; $S' \to SSbS \mid SbSb \mid bb$

$S \to aS' \mid b$; $S' \to SS'' \mid bb$

$S'' \to SbS \mid bSb$

(ii) $S \to \underline{bS}Saas \mid \underline{bS}Sasb \mid \underline{bS}b \mid a$

$S \to bSS' \mid a$

$S' \to Saas \mid Sasb \mid b$

$S' \to SaS'' \mid b$; $S \to bSS' \mid a$

$S'' \to aS \mid Sb$

<u>NOTE</u>: The final derivation will be same for all diff. approaches.

Parsers:-

They are nothing but syntax analyzers. At the syntax analysis phase, we use the software, parsers.

Parsers

- Top down (TDP)
  - TDP with full Backtracking → Brute force method
  - TDP without Backtracking → Recursive Descent
    - Non-Recursive Decent (LL(1))
- Bottom up (BUP)
  - Also known as SR parsers or shift reduce parsers.

If we use a stack, generally shift is pushing and reduce is popping, hence the name shift-reduce parsers.

Bottom-Up Parser

- Operator precedence parsers can parse (ambiguous grammar)
- LR Parsers
  - LR(0)
  - SLR(1)
  - LALR(1)
  - CLR(1)

NOTE (i) LR parsers scans the string from left to right, hence 'L' and we use the reverse of right-most derivation, hence 'R'.

(ii) In case of TDP without backtracking, we have recursive descent as for every production, we will write recursive function. To construct TDP without backtracking,

(i) The grammar should not have left recursion.

(ii) The " " " " non-determinism.

(iii) No parser in the diagram can parse ambiguous grammar except operator precedence parser.

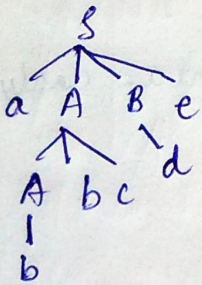
Difference b/w Top-Down and bottom-up parsers:-

eg. $S \rightarrow aABe$; $A \rightarrow Abc \mid b$; $B \rightarrow d$

$w = abbcde$

Top-down

(i)



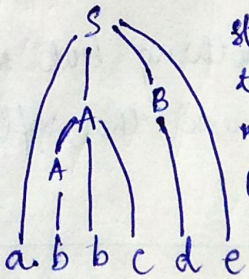
starting from start symbol to generate. And we make decision at every pt., the next production to be used for the variable.

(ii) It is very deterministic in approach.

(iii) It follows left-most recursion.

Bottom-up

(i)



start from the terminal and move towards the root. Here, we make decision about when seeing a terminal should we reduce it or not, when to reduce it.

(ii) It follows right-most recursion

ie, $S \Rightarrow aABe$
$\Rightarrow aAbcBe$
$\Rightarrow abbcBe$
$\Rightarrow abbcde$

forward process

Therefore, it will use left-most derivation.

ie, $S \Rightarrow aAABe$
$\Rightarrow aAAde$
$\Rightarrow aAbcde$
$\Rightarrow abbcde$

reverse process.

Therefore, it will use right-most derivation in the reverse way. as we start from the string.

LL(1) parsers:-

The 1st 'L' indicates, we are scanning the i/p from left to right. The 2nd 'L' means we are using Left-most derivation. And the '1' means no. of look-aheads, ie, the no. of symbols we want to see before we make a decision.

Generally in LL(1) we will ~~see~~ only one symbol, ie, why lookahead is 1.

The different components of LL(1) parsers:-

(i) LL(1) parser → the parsing algorithm which controls the action of the parser.

(ii) LL(1) parsing table → parsing table, here is, a data struc. which we will construct using the given grammar.

(iii) Stack → a data structure used for the procedure of parsing. The bottom of the stack always has a dollar ($) sign.

(iv) i/p buffer → inside it, we will have the i/p. The end of the i/p will always have $. (dollar).

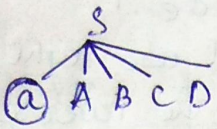
NOTE: The dollars ($) are used to indicate the stop.

To construct the LL(1) parser, we need to construct the LL(1) parsing table, with the given grammar.

Before constructing the LL(1) parsing table, we should know 2 functions, ie, First() and Follow().

(i) <u>First()</u> – it is nothing but if we start with any symbol and generate all the strings from it.

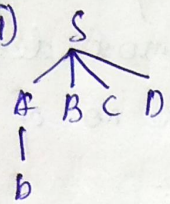
for ex. (i) $S \rightarrow aABCD$; $A \rightarrow b$; $B \rightarrow c$; $C \rightarrow d$; $D \rightarrow e$



In all the strings that we get from S is 'a'. Therefore, First(S) = 'a'.

First(A) = b ; First(B) = c . We always have to take the terminal in the First() function call.

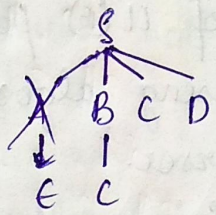
eg. (ii)



∴ First(S) = 'b'

<u>NOTE</u> : In case the 1st derives 'ϵ' (Epsilon), then the First() vanishes and the function First is passed to the next value in the string.

eg (iii) $A \rightarrow b/\epsilon$



∴ First(S) = 'c'

Therefore, if we have a variable or a string and we try deriving from it, all the strings, ~~the~~ whatever comes in the beginning, ie, the 1st terminal or symbol is called First().

(ii) <u>Follow()</u> – It means the terminal which ~~it~~ could follow a variable in the process of derivation.

for ex. S is always the start followed by $, that is, the i/p is always being followed by a $

If we have to find the follow of a variable, we have to find out the terminals following that variable in the '1st' of derivation (First())

eg. $S \rightarrow ABCD$, follow of B?

∴ B is followed by 'd'

NOTE: The follow of start symbol, here S, is going to contain $ as well as if we see 'S' in the right hand side of any prod. we have to consider that too.

∴ for $S \to ABCD$; $S - \{\$\}$

$A \to b/\epsilon$; $A - \{c\}$

$B \to c$; $B - \{d\}$

$C \to d$; $C - \{e\}$

$D \to e$; $D - \{\$\}$

NOTE: If there is nothing following a symbol, then there will be $ in the end. But it is not always so.
Whatever follows the RHS, will follow the rightmost symbol of the RHS.

Examples on how to find first and follow in LL(1):-

~~eg (i) $S \to ABCDE$; $A \to a/\epsilon$; $B \to b/\epsilon$; $C \to c$; $D \to d/\epsilon$; $E \to e/\epsilon$~~

eg (i)

	First	Follow
$S \to ABCDE$	{a, b, c}	{$}
$A \to a/\epsilon$	{a, ε}	{b, c}
$B \to b/\epsilon$	{b, ε}	{c} from 'S'
$C \to c$	{c}	{d, e, $}
$D \to d/\epsilon$	{d, ε}	{e, $}
$E \to e/\epsilon$	{e, ε}	{$}

(ii)

	First	Follow
$S \to Bb/Cd$	{a, b, c, d}	{$}
$B \to aB/\epsilon$	{a, ε}	{b}
$C \to cC/\epsilon$	{c, ε}	{d}

(iii)

	First	Follow
$E \to TE'$	{id, (}	{$,)}
$E' \to +TE'/\epsilon$	{+, ε}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to *FT'/\epsilon$	{*, ε}	{+, $,)}
$F \to id/(E)$	{id, (}	{*, +, $,)}

(iv)

	First	Follow
$S \to ACB/CbB/Ba$	{d, g, h, b, a, ε}	{$}
$A \to da/BC$	{d, g, h, ε}	{h, g, $}
$B \to g/\epsilon$	{g, ε}	{a, h, $, g}
$C \to h/\epsilon$	{h, ε}	{g, $, b, h}

(v)

	First	Follow
$S \to aABb$	{a}	{$}
$A \to c/\epsilon$	{c, ε}	{d, b}
$B \to d/\epsilon$	{d, ε}	{b}

If there are no variable in the ~~left~~ right, then the follow will contain $ nothing except for start symbol.

NOTE: For First, leftmost variable on the RHS. And for follow, search for the variable, for which the follow is being called, on the RHS.

(vi)

$S \to aBDh$	$\{a\}$	$\{\$\}$
$B \to cC$	$\{c\}$	$\{g, h, f\}$
$C \to bC/\epsilon$	$\{b, \epsilon\}$	$\{g, h, f\}$
$D \to EF$	$\{g, f, \epsilon\}$	$\{f, h\}$
$E \to g/\epsilon$	$\{g, \epsilon\}$	$\{f, h\}$
$F \to f/\epsilon$	$\{f, \epsilon\}$	$\{h\}$

NOTE: If a variable is not in the right, then its follows will be nothing.

Construction of LL(1) parsing table:

The main purpose of top-down parser is, whenever we have 2 alternatives for a variable, we have to be sure about which production should we use. Another use of top-down parser, is to find out the first generated by a symbol.

for eg.

	First	Follow
$E \to TE'$	$\{id, \epsilon\}$	$\{\$,)\}$
$E' \to +TE'/\epsilon$	$\{+, \epsilon\}$	$\{\$,)\}$
$T \to FT'$	$\{id, (\}$	$\{+,), \$\}$
$T' \to *FT'/\epsilon$	$\{*, \epsilon\}$	$\{+,), \$\}$
$F \to id/(E)$	$\{id, (\}$	$\{*, +,), \$\}$

NOTE: We are writing \$ in the below table as there is chance of \$ being in string, as the i/p is always going to end with \$.

	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: Whenever we have ϵ production or we have ϵ in the first of any variable then we are going to place the production even on the left-hand side, i.e, follows of that variable.

Therefore, if the first of ~~left-hand~~ right-hand side is a terminal, directly place it there. Else if the 1st of ~~left~~ right hand side is ϵ

then place it even in the follow of left hand side.

Now, with the help of LL(1) parsing table, we construct the parse tree.

eg (ii) $S \to (S)/\epsilon$

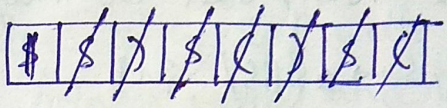
	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

NOTE: In many cases we get multiple entries in a single cell of the parse table, in such cases the grammar is not feasible.

Suppose, we want to generate: (( )) $

∴ The bottom of stack is $ and top of it is, start symbol (S).

pointer



∴ S

(S)

When there is a match b/w top of stack and i/p, so whatever we wanted to generate is generated so, pop off from the stack and increment the pointer.

'S' on open brace → (S), 'S' on close brace → ϵ

S

(S)

(S)

ϵ

NOTE: The LL(1) parsing is always going to start with a $ in the bottom of stack and $ on the end of i/p. When there is a match $ = $, we say it is successful matching, else we consult the parse table.

eg (iii) $S \to AaAb / BbBa$; $A \to \epsilon$; $B \to \epsilon$

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

Here, in every cell we have got only 1 prod. Therefore, the grammar is LL(1) and it could be constructed.

NOTE: Grammar that are not valid for LL(1) are:-

(i) Left recursive ; (ii) Non-deterministic.

But there are grammars which is neither (i) nor (ii) and still not valid for LL(1) parsing, ie, because they have got more than 1 entry. eg. $A \to \alpha_1 / \alpha_2 / \alpha_3 / \ldots / \alpha_n$

Here, if the first of α_1 and α_2 are same for eg., then they have

to be placed in the same cell. This is the reason why we may not get LL(1) passing table for all the grammars. For $\alpha_1, \alpha_2, \alpha_3, \ldots \alpha_n$ to be valid for LL(1) passing, they have to be mutually exclusive, ie, their first should not be same.

Another case is, when we have first of variable and follow of it same. eg. $A \to \alpha / \epsilon$, then we have to place them in the same row, then we may get non-LL(1) grammars.

eg. (i) $S \to aSbS / bSaS / \epsilon$ $\{a\}$ $\{b\}$ $\{a, b, \$\}$

Invalid for LL(1).

(ii) $S \to aABb$ $\{a\}$
$A \to c / \epsilon$ $\{c\}\{d, b\}$
$B \to d / \epsilon$ $\{d\}\{b\}$

Valid for LL(1)

(iii) $S \to A / a$ $\{a\}, \{a\}$
$A \to a$ $\{a\}$

Invalid for LL(1)

(iv) $S \to aB / \epsilon$ $\{a\}\{\$\}$
$B \to bC / \epsilon$ $\{b\}\{\$\}$
$C \to cS / \epsilon$ $\{c\}\{\$\}$

Valid for LL(1)

(v) $S \to AB$
$A \to a / \epsilon$ $\{a\}\{b, \$\}$
$B \to b / \epsilon$ $\{b\}\{\$\}$

Valid for LL(1)

(vi) $S \to aSA / \epsilon$ $\{a\}\{c, \$\}$
$A \to c / \epsilon$ $\{c\}\{\$, c\}$

Invalid for LL(1)

(vii) $S \to A$
$A \to Bb / Cd$ $\{a, b\}\{c, d\}$
$B \to aB / \epsilon$ $\{a\}\{b\}$
$C \to cC / \epsilon$ $\{c\}\{d\}$

Valid for LL(1)

(viii) $S \to aAa / \epsilon$ $\{a\}\{\$, a\}$
$A \to abS / \epsilon$ $\{a\}\{a\}$

Invalid for LL(1)

(ix) $S \to iEtSS' / a$
$\{i\}\{a\}$
$S' \to eS / \epsilon$ $\{e\}\{\epsilon\}$
$E \to b$

Invalid for LL(1)

Recursive Descent parser:

It is one of top-down parsers. The name recursive descent parser came from, for every name variable we will write a function.

for eg. $E \to iE'$ —— (i)

$E' \to +iE' / \epsilon$ —— (ii)

(i) E() {

```
1. if (l == 'i') { // l = lookahead, global variable
2.     match ('i');
3.     E'();
4. }
}
```

NOTE: 'l' is always going to call the next character.

Writing the functions is Recursive Descent parser.

(ii) E'() {

```
1. if (l == '+') {
2.     match ('+');
3.     match ('i');
4.     E'();
   }
5. else
6.     return;
7. }
```

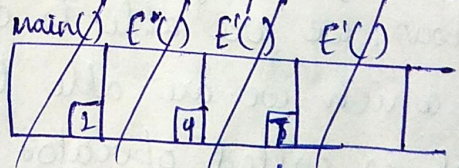
match () function is going to match the characters of the i/p with the look-ahead pointer value and then increment the lookahead if it match successfully.

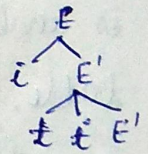
```
match (char t) {
1. if (l == t)
2.     l = getchar();
3. else
4.     printf ("Error");
}
```

```
main () { // it calls everything
1. E();
2. if (l == '$')
3.     printf ("parsing successful");
}
```

NOTE: Whatever the prod. we might have, we have to write one func. for each variable. If there are more than 1 prod. for a variable, then we will write if-else statement or switch case statements.

eg. i + i $
↑ ↑ ↑





∴ O/p = parsing successful

NOTE: We have written the function for every variable and recusion stack is called ~~for~~ provided by O.S. It is a top-down parser as it generated from the top and reached the bottom generating the entire string.

Operator grammar :-

Grammar used to define mathematical operation is operator grammar with some restrictions on the grammar.

eg. $E \to E+E \,/\, E*E \,/\, id$

But $\cancel{E} \to EAE \,/\, id$, $A \to + \,/\, *$] is not operator grammar as we have 2 adjacent variables which is not allowed in such operator grammar case.

eg-(ii) $S \to SAS \,/\, a$, $A \to bSb \,/\, b$] This is not op. grammar

$\Downarrow$ converting

$S \to SbSbS \,/\, SbS \,/\, a$, $A \to bSb \,/\, b$] This is op. grammar.

NOTE : Operator precedence parser is the only parser which can parse some of the ambiguous grammar. It can parse by constructing the operator relation table.

	id	+	*	$
id	-	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	- ← Completely successful

NOTE : id and id is never compared.

Operator Relation Table

NOTE : Even though the grammar is ambiguous, we are able to construct the operator ~~precedence~~ relation table, in which we are able to define the precedence rules. ie, why it is called operator precedence parser.

As we can see that in the grammar, it doesn't talk about precedence but the table talks about precedence.

Acc. to the precedence table, the op. precedence parser will parse the i/p.

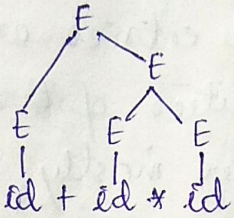
eg. (i) id + id * id $

$	id	+	id	*	id
	1	5	2	4	3

E E E
id * id * id

∴ Highest precedence gets popped 1st.

NOTE: It is same as infix to postfix algorithm, ie, whenever we get less than or equal to the top of stack, we push the i/p into stack and vice versa. (Rule)

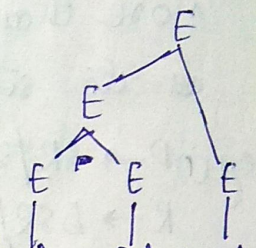


It is a kind of bottom-up parsing. As we started from the terminals and went up to the root.

(ii) id + id + id $

$	id	+	id	+	id
	1	3	2	5	4

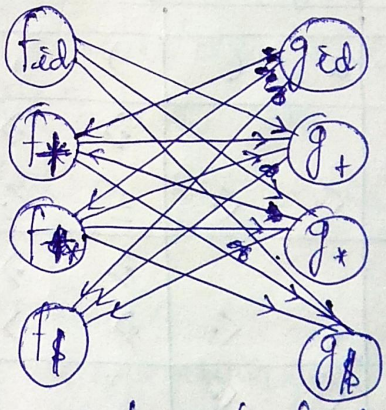
∴



NOTE: The disadvantage of the operator precedence table is that it has $O(n^2)$ no. of entries, where 'n' is the no. of operators.

So, in order to decrease the size of the table, we go for operator function table.

f \ g	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	-



NOTE: To construct the function table, we have to find out the length of the longest path for every node, starting from that node.

∴ $f_{id} \to g_* \to f_+ \to g_+ \to f_\$$

$g_{id} \to f_* \to g_* \to f_+ \to g_+ \to f_\$$

∴ Table,

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

NOTE: The function table is same as the operator precedence table. But the size of func. table is less as compared to relation table., ie,

func. table size = 2n ; relation table size = n^2

One disadvantage of func. table is that, even there is blank entries in the relation table, there are no blank entries in func. table. The blank entries are errors. Therefore, error detecting capabilities of relation table is more than function table. But we mostly use func. table as its size is small.

eg.(i) $P \to SR/S \to ①$ $\quad P \to SbP/SBS/S$ (SR)

$R \to bSR/bS \to ②$ $\quad S \to WbS/W$

$S \to WbS/W$ $\quad W \to L*W/L$

$W \to L*W/L$ $\quad L \to id$

$L \to id$

[Operator Grammar]

! Not op. grammar

operator Relation table

	id	*	b	$
id	-	·>	·>	·>
*	<·	<·	·>	·>
b	<·	<·	<·	·>
$	<·	<·	<·	-

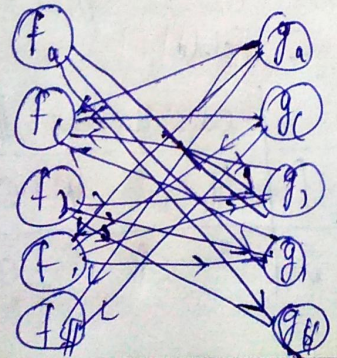
eg.(ii)

	a	(	)	,	$
a			·>	·>	·>
(	<·	<·	≐	<·	
)			·>	·>	·>
,	<·	<·	·>	·>	
$	<·	<·			

⇒

	a	(	)	,	$
f	2	0	2	2	0
g	3	3	0	1	0

Convert into relation function table.



Note: Some operators are never compared in the relation table but in func. table, it is always compared.

We have to maintain seperate DS for the empty cells and accordingly make comparisons.

⊛ LR passing, ~~items~~ and LR(0) items, LR(0) passing table :-

LR passing is a type of bottom-up parsing. They are of 4 types:

(i) LR(0) (ii) SLR(1) (iii) LALR(1) (iv) CLR(1)

Simple LR — Look-ahead LR — Canonical LR

NOTE :- least powerful is LR(0) and most powerful is CLR(1).

The LR parser is an algorithm and this algo. required :

(i) Input stack.

(ii) LR passing table

(iii) i/p buffer.

It is same as the bottom-up parser LL(1), every thing is same except for the parsing table. Therefore, only the construction of the parsing table is going to vary.

Here for the construction of the passing table for various types

LR(0), SLR(1) → LR(0) items

LALR(1), CLR(1) → LR(1) items

are needed

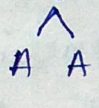
Now, like first() and follow, here we have 'closer' and 'goto'

Item is any production with a dot (.) on the RHS.

eg. $S \to .AA$, which means that on RHS we haven't seen anything.

$S \to A.A$, means we have seen A.

$S \to AA.$, means we have seen everything.



This indicates that the RHS of the a variable is seen and then reduced, completely.

Whenever we see a . dot at left of a variable we are going to write all productions with . dot in the beginning.

eg.

$S' \to S$ ← Augmented grammar

$S \to AA$

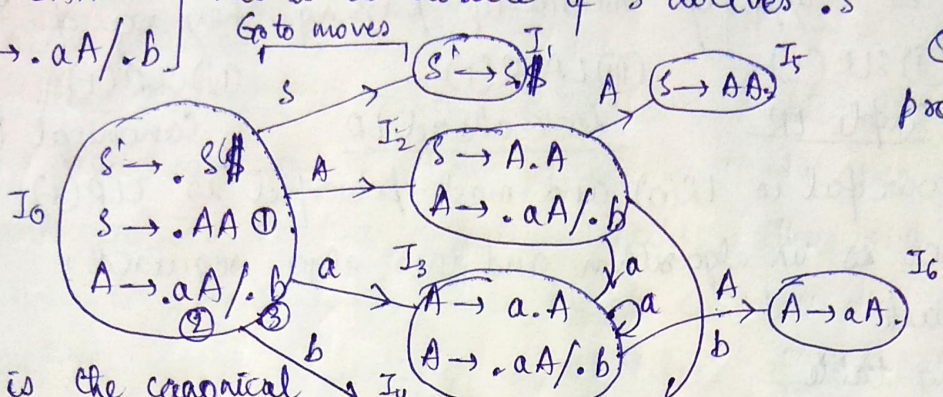
$A \to aA/b$ ← actual grammar

An extra prod. is added to the grammar, therefore, augmented grammar.

$S' \to .S$
$S \to .AA$
$A \to .aA / .b$] This is the closure of S' derives $.S$

Goto moves

Now,



① , ② , ③ production no.

This is the canonical collection of LR(0) items. [final item as . dot is to right end]

Construction of passing table :-

no. of states = 7 = no. of rows.

	a	b	$	A	S
	action			Goto	
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		

$r_1 = S \to AA$; $r_2 = A \to aA$; $r_3 = A \to b$ (final items)

NOTE :- Whenever we get final production item of the augmented grammar var. we have to understand that it is special prod. added which we have to accept. It also means that we have seen everything of the start symbol.

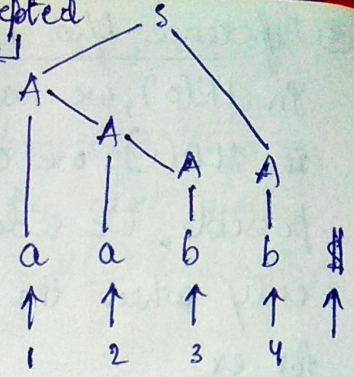
NOTE :- In the construction of LR(0) passing table, whenever any state is having final item in that particular row, we are going to write reduce move completely. The shift moves (S_3, S_4) is nothing but whenever a state travels to another state on a terminal, that corresponds to a shift move. And if a state goes to some other state on a variable, it corresponds to Goto move.

Q. Verify 'aabb' is generated by the above grammar.

→ aabb $, always the i/p is followed by $.
↑↑↑↑

0	a	b	a	b	b	a	A	a	A	b	A	a	b	a	A	a	S	1
	7	8	3	4	1	2	5	6	9	10	13	14	11	12	15	16		

accepted



NOTE: Shift means to push the element on top of the stack and then shift the state no. on top of the stack.
Whenever we shift an element into the stack, we increment the o/p pointer.
Whenever we see a reduce function, we do not reduce the current symbol instead we reduce the previous symbol.

How do we reduce :-

In order to reduce, we see the no. of symbols present on the RHS of a production and then multiply it by 2 and therefore, the resultant is the no. of symbols to be popped of from the stack and the LHS of the production is pushed into the stack and also the state no. ~~and~~ relevant of the corresponding ~~RHS~~ ~~and~~ in the stack.

NOTE: We call it LR(0) as we do not care what the next symbol is or what the look-ahead is, we simply the current symbol, ie, the symbol preceding the symbol on which the look-ahead points to.

Disadvantage of LR(0) :-

(i) Suppose,

$$\underline{a\ a\ \overset{A}{\overset{|}{b}}\ b} \rightarrow aaAb$$, according to LR(0) reduction.

Therefore, LR(0) says that don't look at the next symbol, for whatever the next symbol is, you can reduce it always. This is wrong, as we might get some errors which might be detected later.

(ii) The blank addresses in the table corresponds to errors.

NOTE: LR(0) says that, whenever you have final item in the state you have to put the entire reduce moves in the entire row.

Difference b/w LR(0) and SLR(1) :-

In LR(0), we have blind reduction of the next symbol. while in SLR(1), we do not blindly reduce the next symbol whenever possible. We reduce the current symbol to some variable, X only when the next symbol is also in the follow of X. ~~other~~

for ex

$$\begin{array}{ccccc} & & A & & \\ a & a & b & b & \$ \\ \uparrow & \uparrow & \uparrow & \uparrow & \end{array}$$

aaAb

NOTE: if the next symbol is not in follow of A, then we cannot reduce it further to the start state. Therefore, SLR(1) is a bit more powerful than LR(0), as it is taking wise decision in terms of, if it is not a symbol in the follow of 'A' (here). Therefore, reduce moves are ~~followed~~ done only when LHS is being followed.

$S' \to S$
$S \to AA$
$A \to aA / b$

SLR(1)

	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		

For 4, Checking for reduce moves

State 4 : $A \to b$.

Since state 4 is having the final item, don't place r_3 in the entire row, just place the follow of LHS.

∴ follow of A : {a, b, $}

For 5,

State 5 : $S \to AA$.

We don't place in r_1, we place r_1 in the LHS follow, ie,

follow of S : {$}

For 6,

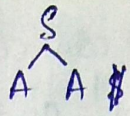
State 6 : $A \to aA$.

follow of A : {a, b, $}

NOTE: In terms of shift moves and Goto moves, both LR(0) and SLR(1) are same. The difference is only in case of reduce move placement, ie, in LR(0) reduce moves are placed in the entire row while in SLR(1) we see whether the next symbol that we get is in the follow of LHS or not. or simply place it in the follow of LHS.

for eg. for state 5,

$S \to AA$



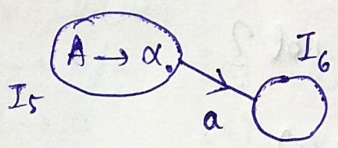
The entry says that you can reduce AA to S, provided that the next symbol is $. And we always see that the next symbol is always in the follow of LHS. Hence, called SLR(1) which for '1' is the no. of lookaheads is 1.

Therefore, SLR(1) is more powerful as it has less no. of reduce moves and if there is any error, SLR(1) will detect them faster while in LR(0), since all the entries are full, detection of errors is slower.

<u>How do we implement the errors:-</u>

In place of blank entries, we have pointers to functions. So, whenever we hit that entry, we can call the function and that function will make the move.

<u>NOTE</u> : Every grammar is not LR(0), due to (i) Shift-reduce conflict. for eg.

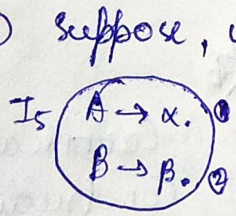


	action	
	a	b
5	S_6/r_1	r_1

due to the final item, we have to put the reduce move in the entire row, for LR(0).

Shift-reduce conflict

(ii) Suppose, we have



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

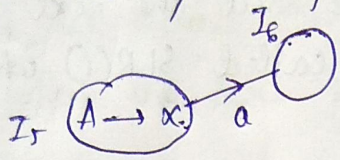
we have r_1 for the ① st production and r_2 for the ② nd production.

Therefore, if we have more than 1 final items, the grammar cannot be LR(0) because of r-r, ie, reduce-reduce conflict. We cannot have 2 entries in a cell.

For SLR(1), the limiting points for SLR(1) is (i) Shift reduce conflict and (ii) Reduce-reduce conflict same as for LR(0).

(i) <u>Shift-reduce conflict</u>: This can occur only when we have shift move for a symbol and reduce move for the same symbol, by the follow of LHS.

eg...



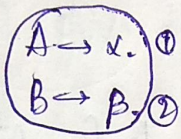
follow of $A \to \{a, \ldots\}$

$\therefore$ <u>SR conflict</u>

(ii) <u>Reduce-reduce conflict</u>: This can occur only when there is any intersection b/w follow of the 2 variables on the LHS.

Here, if there is conflict b/w A and B.

eg.



$A \to \{a, c, d\}$
$B \to \{a, e, f\}$] $\{a\}$

$\therefore$ <u>rr conflict</u>

<u>Follow of A $\cap$ Follow of B $\neq \phi$</u>

Q. Given grammar,

$S \to dA / aB$
$A \to bA / c$
$B \to bB / c$

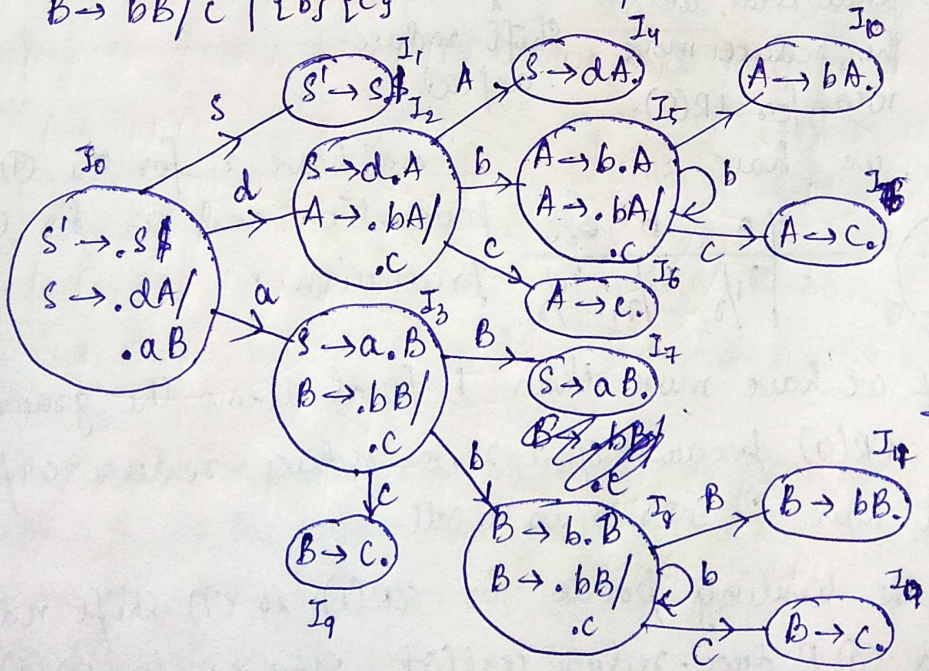
Find out whether it is
(i) LL(1) or not?
(ii) LR(0) or not?
(iii) SLR(1) or not?

→ (i)

	First
$S \to dA / aB$	$\{d\}\{a\}$
$A \to bA / c$	$\{b\}\{c\}$
$B \to bB / c$	$\{b\}\{c\}$

$\therefore$ Valid for LL(1)

(ii)



<u>Canonical Construction of LR(0)</u>

NOTE: We need not construct the parse trees in order to find the SR conflict and RR conflict. We can construct the canonical diagram for LR(0) item and in the construction, find out the conflicts.

For conflict, reduce move is a must. So, whenever we get a final item, we have to stop and find out whether there is a conflict or not.

$\therefore$ $I_4, I_6, I_7, I_9, I_{10}, I_{11}$ does not have shift operation ~~and~~ hence no SR conflict and also there is only 1 final item these cases therefore, no R-R conflict.

Hence, the grammar is LR(0).

Moreover, the grammar is SLR(1) also as SLR(1) is a more concentrated form of LR(0), which means, it has less no. of reduce moves as compared to LR(0).

Q.2 $S \to A/a$; $A \to a$

(i) LL(1)?
(ii) LR(0)?
(iii) SLR(1)?

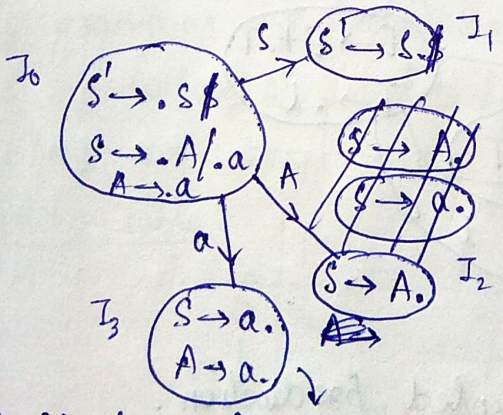
→ (i) For LL(1),

$S \to A/a$ $\{a\}, \{a\}$

$A \to a$ $\{a\}$

Not valid for LL(1)

(ii) For LR(0), canonical construction,



$\therefore$ There are 2 reduce moves for 'a' as o/p. Hence, the grammar is not LR(0).

~~We have to~~

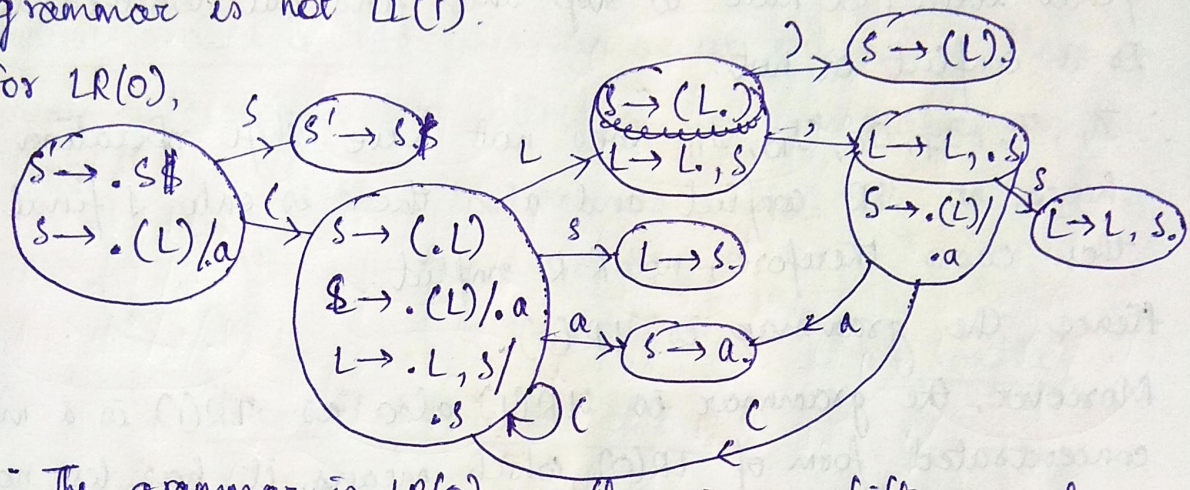
(iii) We have to put the same entry under \$, \$ ie, follow of 'S' and 'A' for I_3 hence the grammar is not SLR(1).

Q.3. $S \to (L)/a$ (i) LL(1)?
$L \to L, S/S$ (ii) LR(0)?
(iii) SLR(1)?

→ (i) $S \to (L)/a$ $\{(, a\}$
$L \to L, S/S$ $\{$

Any left recursive grammar cannot be LL(1). Hence, the grammar is not LL(1).

(ii) For LR(0),



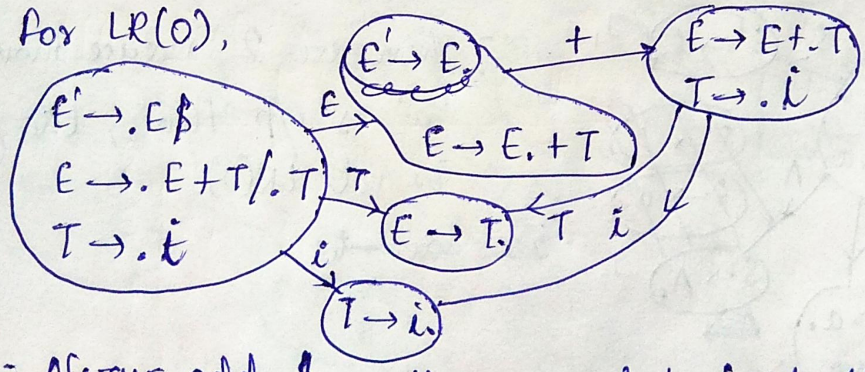
∴ The grammar is LR(0) as there is no shift or reduce moves in the final item.

(iii) And hence, it is also SLR(1).

Q.4 $E \to E+T/T$ (i) LL(1)?
$T \to i$ (ii) LR(0)?
(iii) SLR(1)?

→ (i) ∵ There is recursion in the grammar (left recursion), hence it is not LL(1).

(ii) For LR(0),



NOTE: Always add $ in the augmented production.

∴ Grammar is LR(0). Hence, it is also SLR(1).

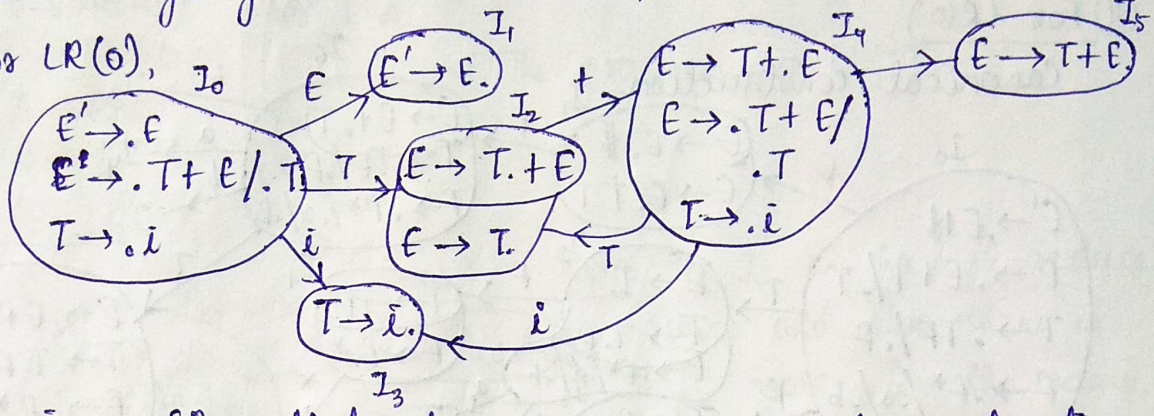
Q.5 $E \to T + E$ ① / T ② (i) LL(1)?
$T \to i$ ③ (ii) LR(0)?
(iii) SLR(1)?

→ (i) For LL(1),

$E \to T+E / T$ {i} {i}

∵ Both are going into the same cell, hence it is not LL(1).

(ii) for LR(0),



There is a SR conflict at I_2 in the canonical construction. Therefore, the grammar is not LR(0). ~~And hence not SLR(1)~~.

Passing Table :-

	i	+	$	E	T
0.	S_3			1	2
1.			accept		
2.	r_2	S_4 / r_2	r_2		
3.	r_3	r_3	r_3		
4.	S_3			5	2
5.	r_1	r_1	r_1		

(iii) The grammar is SLR(1) as

Follow of $T = \{+, \$\}$; Follow $(E) = \{\$\}$

∴ Non-conflicting SR and R-R, hence SLR(1).

Passing Table :-

	i	+	$	E	T
0	S_3			1	2
1			accept		
2		S_4	r_2		
3		r_3	r_3		
4	S_3			5	2
5			r_1		

Q.6 $E \to E+T \,/\, T$ ①②
$T \to TF \,/\, F$ ③④
$F \to F^* \,/\, a \,/\, b$ ⑤⑥⑦

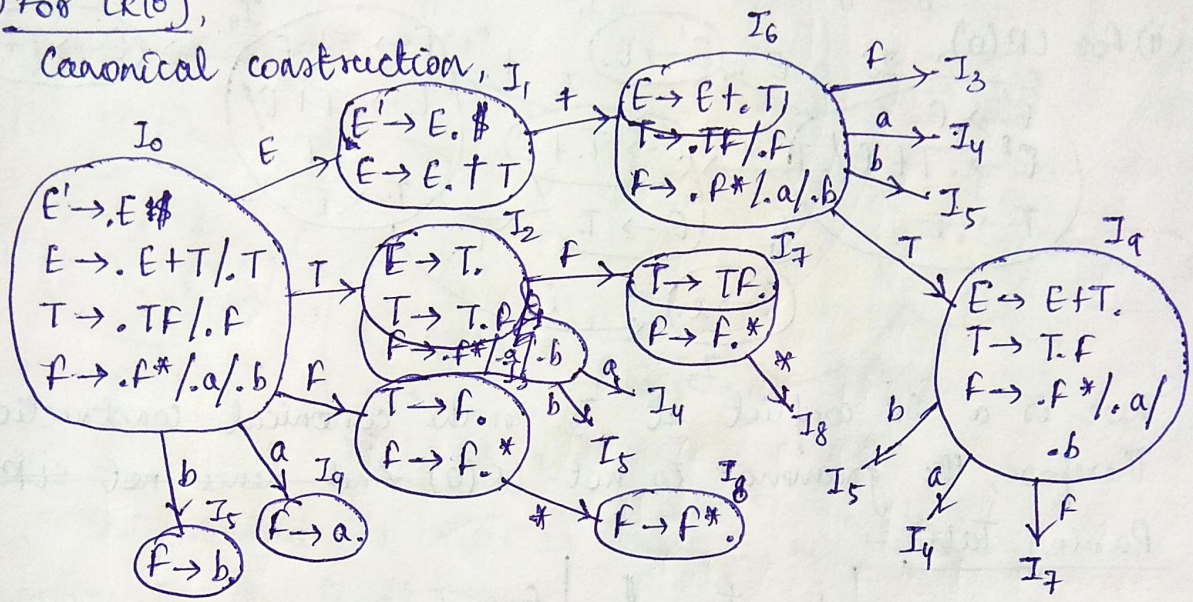
(i) LL(1)? ✗
(ii) LR(0)? ✗
(iii) SLR(1)?

→ (i) <u>For LL(1),</u>

Since, the grammar given is left recursive, it is not LL(1).

(ii) <u>For LR(0),</u>

Canonical construction,



∵ There is a reduce move at I_2 as well as there is a shift move for variables symbols 'a' and 'b', there is SR conflict resulting in invalidity for LR(0).

(iii) <u>For SLR(1),</u>

We are going to construct the parsing table only for the conflicting states and not the entire canonical construction for SLR(1) checking.

∴

	a	b	+	*	$\$$
2	S_4	S_5	r_2		r_2
3	r_4	r_4	r_4	S_8	r_4
7	r_3	r_3	r_3	S_8	r_3
9	S_4	S_5	r_1		r_1

Follow of E : {+, $}
Follow of T : {+, $, a, b}
Follow of F : {*, +, $, a, b}

∴ The grammar is SLR(1).

Therefore, even when the grammar is not LL(1) and LR(0), still it could be SLR(1) by the removal of the conflicts.

Q.7. $S \to AaAb/BbBa$ ①
$A \to \epsilon$ ②
$B \to \epsilon$ ③ ④

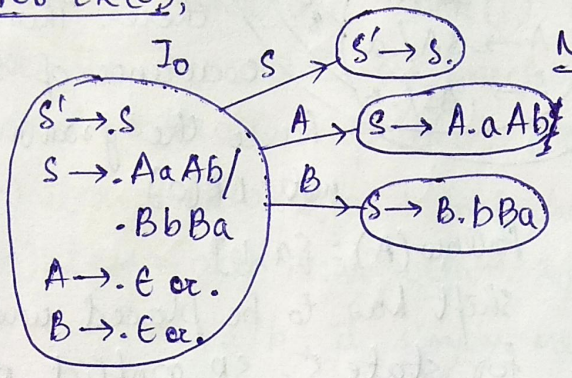
(i) LR(1) ? ✓
(ii) LR(0) ? ✗
(iii) SLR(1) ? ✗

→ (i) for LL(1),

$S \to AaAb/BbBa$ {a} {b}
$A \to \epsilon$ {ε}
$B \to \epsilon$ {ε}

∴ The grammar is LL(1).

(ii) for LR(0),



NOTE: In I_0 we have 2 reduce moves and also there is shift moves for the same. Therefore, the grammar is not LR(0).

NOTE: For anything like $A \to .\epsilon$ or $B \to .\epsilon$, we can write it as $A \to .$ or $B \to .$

(iii) for SLR(1),

	a	b	$
0	r_3/r_4	r_3/r_4	

∵ There is R-R conflict in 'a' and 'b' for follow of 'A' and follow of 'B' for the state 0 gives the same symbol o/p.

Therefore, the grammar is also not SLR(1).

Q.8. $S \to AS/b$
$A \to SA/a$

(i) LL(1) ? ✗
(ii) LR(0) ? ✗
(iii) SLR(1) ?

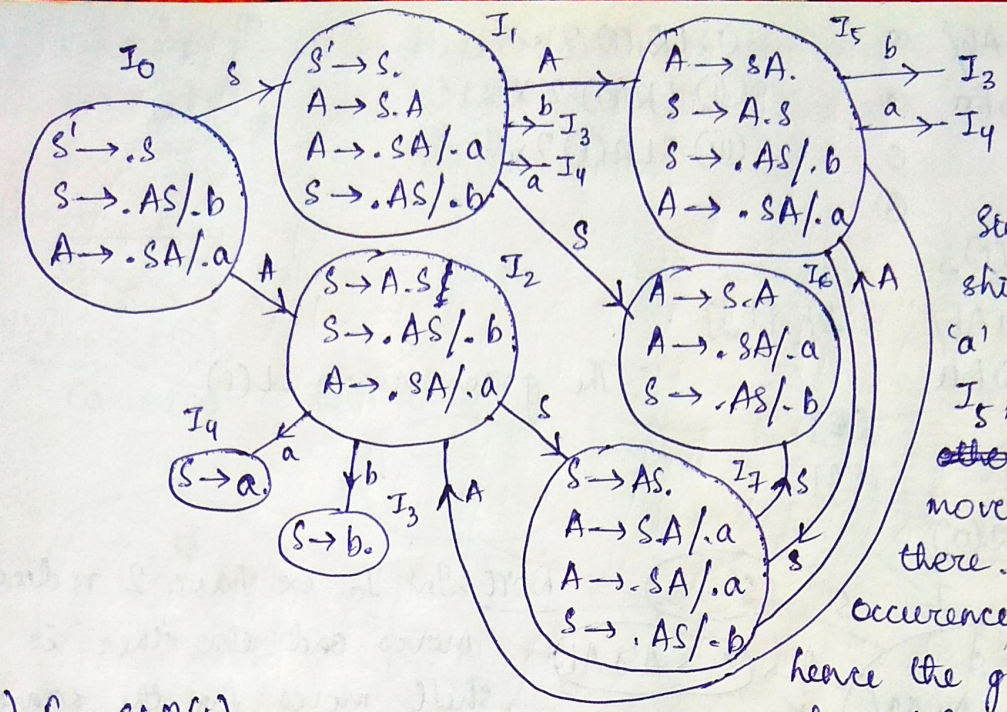
→ (i) for LL(1),

	First	Follow
$S \to AS/b$	{a, b}	{b}
$A \to SA/a$		

∵ The production S has to be placed 2 cells, the grammar is not LL(1).

(ii) for LR(0),

Canonical Construction :



Since, there are shift moves for 'a' and 'b' in I_5, I_7 and ~~many~~ ~~other~~ so, reduce moves are also there. This leads to occurence of SR conflict hence the grammar is not LR(0).

(iii) for SLR(1),

	a	b	$
5	r_1/~~s_4~~	r_1/s_1	

Follow(A) = {a, b}
shift has to be placed under 'b', for state 5. SR conflict occurs.

∴ The grammar is not SLR(1) also.

Moreover, since the grammar is ambiguous, no ~~grammar~~ parser can parse this grammar.

Q.9 S → Aa/bAC/dC/bda
A → d

→ (i) for LL(1),

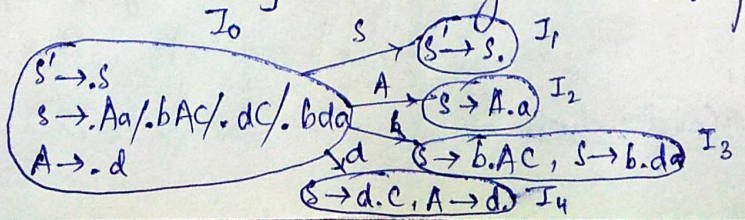
S → Aa/ {d}
bAC/ {b}
dC/ {d}
bda {b}

∴ 2 productions are going into the same cell, LL(1) is not valid.

(ii) For LR(0),

S → Aa/bAC/dC/bda
A → d

] Constructing canonical form



Since, in I_4 state, there is a reduce move for 'd' and shift move for 'c', SR conflict occurs. Hence, the grammar is not LR(0).

(ii) for SLR(1),

	a	b	c	d	$
I_4	r_3		s/r_3		

∴ SR conflict, hence not SLR(1) also.

CLR(1) and LALR(1) parsers:-

In order to construct LALR(1) and CLR(1), we are not going to use the canonical construction of LR(0) items, we are using canonical collection of LR(1) item.

LR(1) item = LR(0) item + lookahead,

LR(0) item = any item with . dot on LHS.

eg. $S \to .aA, a/b$, it simply means that not to place the reduce move in the entire row or in the follow of LHS, just place them in the lookaheads 'a' and 'b'. Therefore, the lookaheads are going to be useful, in order to determine where we will be placing the final item.

NOTE: For the lookahead,

Suppose, $A \to \alpha.B\beta, a/b$ is the given production. —(i)

$B \to .\gamma, c/d$, —(ii) The variable 'B' production is added because the 'B' in prod.(i) after the . dot.

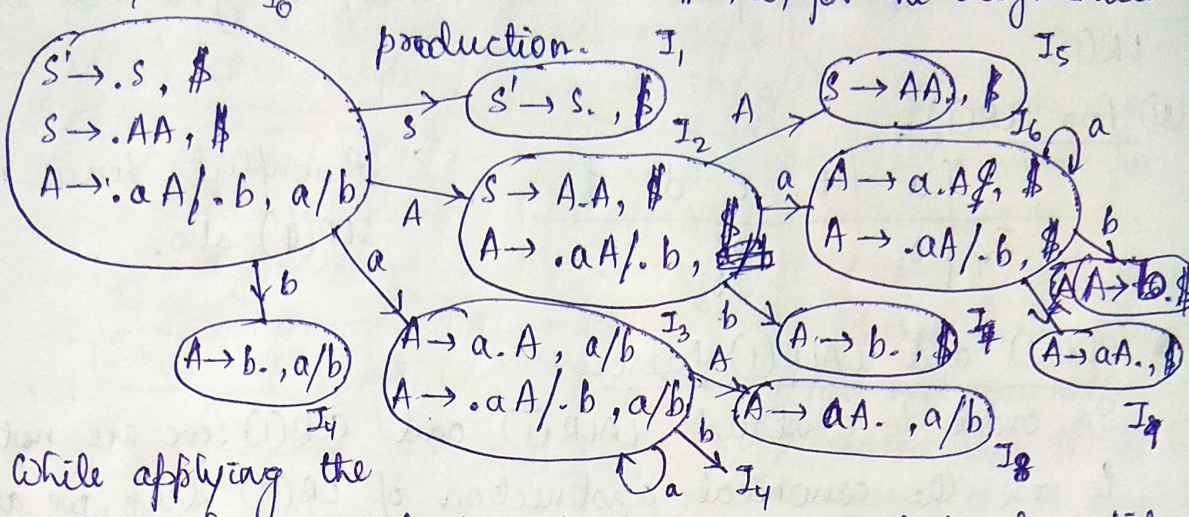
lookahead added

∴ For the lookahead, we find the first of remaining symbols and then find the first of that symbol. In case, there is no β, then add the 2 terminals a/b directly as the lookahead.

Here, $\beta \to CBA/d$ first(β) : {c, d}

eg. $S \to AA$
$A \to aA/b$

NOTE: For the initial condition always start with the $, ie, for the augmented production.



NOTE: While applying the transition of the . dot, do not change the lookahead, while applying the closure, lookahead might change.

The no. of canonical connection for LR(1) > no. of canonical connection for LR(0) items.

In CLR(1), the no. of states = no. of states in the canonical conn. The Goto part and shift part is same as the previous. The diff. is going to be in the placement of final items, which here we are going to place in the look-ahead symbols, and not in the entire row or in the follow of LHS.

for. eg. here for state I_4, we are going to place them in 'a' and 'b'.

Therefore, here the no. of reduce moves are less than LR(0) and also SLR(1). Hence, reducing the no. of conflicts. Moreover, due to the reduction of reduce moves the blank spaces are going to increase. So, error detecting capabilities are also going to increase, of the parser.

We are increasing the no. of states because of having lookaheads and also decreasing the no. of reduce moves because of having the look-aheads.

CLR(1) parsing table :-

$S \to AA$ ①
$A \to aA$ ②
$A \to b$ ③

	a	b	$	S	A
0	S_3	S_4			2
1.					
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		

NOTE: The diff. in LR(0), SLR(1). and CLR(1) is in the placement of reduce moves.
Here, we got more no. of states as there a states with same LR(0) items but different look-aheads.
here, $\boxed{I_3, I_6}$, $\boxed{I_4, I_7}$, $\boxed{I_8, I_9}$
$\Downarrow$ I_{36} $\Downarrow$ I_{47} $\Downarrow$ I_{89}

NOTE: If we merge the states with same LR(0) items and diff. look-aheads, we could decrease the size of the CLR(1) parsing table.
For eg where there is 3, 6 we are going to replace them with 36.
By merging the states, we make the lookaheads irrelevant.
If we merge the states we will again get 7 states as in case of SLR(1). or LR(0).

CLR(1) $\geq$ SLR(1) = LR(0), for no. of states.
LALR(1) = SLR(1) = LR(0), for no. of states.

The merging of states results in the construction of LALR(1) item.

LALR(1) parsing table :-

	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
2	S_{63}	S_{74}			5
36	S_{36}	S_{47}			98
47	r_3	r_3			
5			r_1		
63	S_{63}	S_{74}			89
74			r_3		
89	r_2	r_2			
98			r_2		

$\Longrightarrow$

	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
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		

NOTE: We need to perform union operation for the value of cells in merge items.

Conflicts and examples of CLR(1) and LALR(1) :-

	SR Conflict	RR Conflict
LR(1) items	$A \to \alpha . a\beta , c/d$ $B \to \gamma . , a/\$$	$A \to \alpha . , a$ $B \to \beta . , a$
	Shift move for symbol 'a' at 'A' and reduce move for 'a' at 'B' leading to shift-reduce conflict.	Reduce move for 'a' for both 'A' and 'B' in the same state, leading to reduce-reduce conflict.

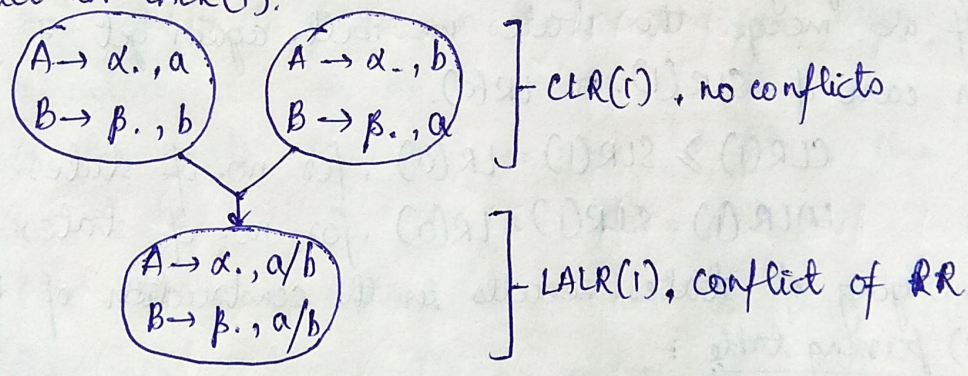
NOTE :

(i) Therefore, SR conflict and RR conflict for LR(1) will even depend on the look-ahead.

(ii) If the above conditions are satisfied, the grammar is proved to be not CLR(1). Moreover it will also not be LALR(1) as even if we reduce the no. of states in CLR(1) the conflicts (SR/RR) is going to remain.

(iii) If a grammar is CLR(1), it may not be LALR(1) always, as though we do not have RR conflicts in CLR(1), we could get RR conflict in LALR(1).

eg.



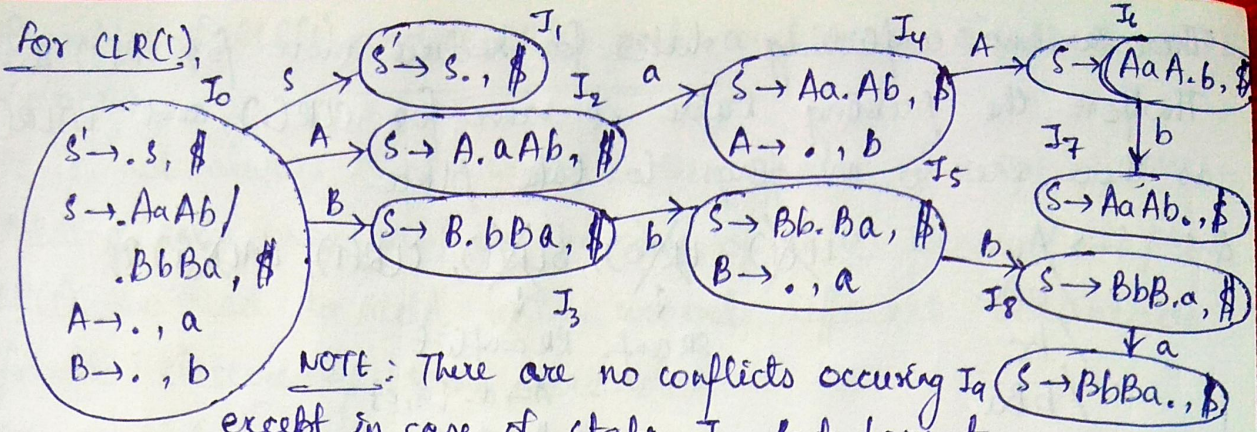
(a) If there are no SR conflicts in CLR(1), then there are no conflicts in LALR(1). If there are no RR conflicts in CLR(1), then there may or may not be in ~~RR~~ LALR(1).

Q.1 $S \to AaAb / BbBa$
$A \to \epsilon$
$B \to \epsilon$

~~LL(1)~~, ~~LR(0)~~, ~~SLR(1)~~, ~~CLR(1)~~, ~~LALR(1)~~ ?

follow(A) = {a, b}
follow(B) = {a, b}

→ For CLR(1),



NOTE: There are no conflicts occuring except in case of state I_0. But here to we have 2 reduce moves with different lookaheads. Hence, CLR(1) is satisfied without any conflict.

For LALR(1),

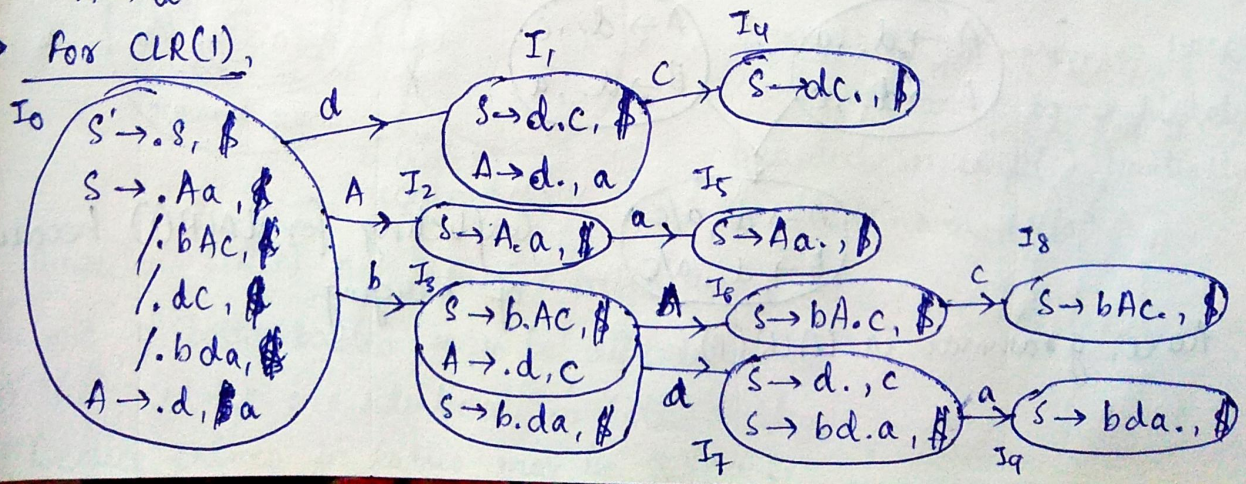
If we see the canonical connection, we do not have any state containing final states with same ~~lookaheads~~ LR(0) items and different lookaheads. Therefore, LALR(1) is satisfied without any conflict. There cannot be any merging here. We need not ~~cannot~~ consider LALR(1) at all as parsing table for CLR(1) is for LALR(1).

NOTE: For this grammar, the no. of states for all valid parsers are going to be the same.

Q.2 S → Aa/
bAc/
dc/
bda
A → d

~~LL(1)~~, ~~LR(0)~~, ~~SLR(1)~~, CLR(1), LALR(1)

→ For CLR(1),



There are no conflicting states for reduce move for LR(1) item. Therefore, the parsing table is same for CLR(1) and LALR(1). as there are no mergings to take place.

Q-3. $S \to Aa$
$/ bAc$
$/ Bc$
$/ bBa$

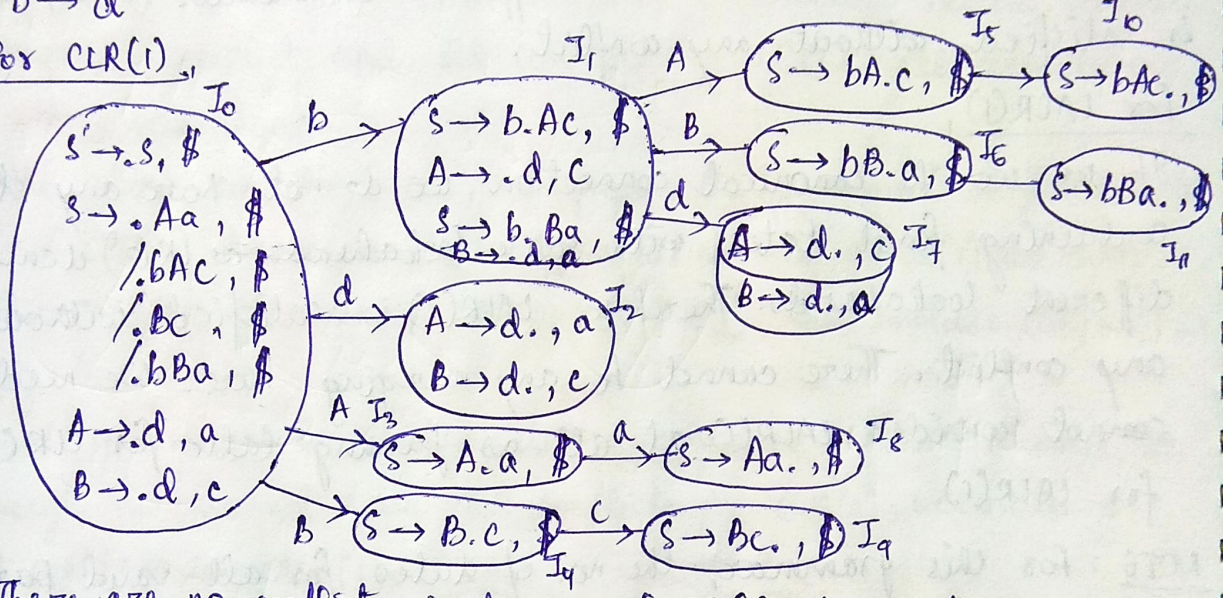
$A \to d$
$B \to d$

LL~~(1)~~, LR~~(0)~~, SLR~~(1)~~, CLR(1), LALR~~(1)~~ ?

LR(0) ↓ SR conf.

SLR(1) ↓ RR conflict
$A \to d. \{a, c\}$
$B \to d. \{a, c\}$

→ For CLR(1),

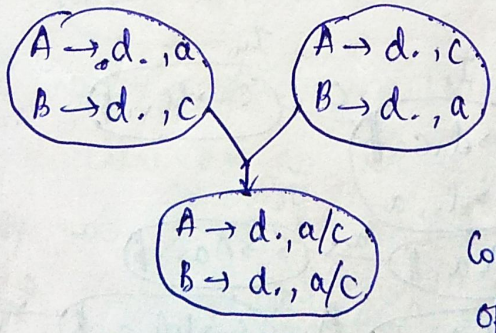


∴ There are no conflicts in terms of LR(1) items. Hence, grammar is CLR(1).

For LALR(1),

We have 2 states with same LR(0) items but dif. lookaheads ie, I_2 and I_7

∴



Conflicting for LALR(1) because of merging.

Hence, grammar is LALR(1).

Examples of CLR(1) and LALR(1) and comparison of all the parsers:

It is not always that we add the closure of a symbol once. Adding once only happens in case of LR(0) items. But for LR(1), we need to add until we get different lookaheads for the closure of same variable.

eg. $S \to A$
$A \to AB / \epsilon$
$B \to aB / b$

Closure $(S \to .A, \$)$

$S \to .A, \$$ — (i)
$A \to .AB/., \$$ — (ii) $\Rightarrow$
$A \to .AB/., a/b$ — (iii)

$S \to .A, \$$
$A \to .AB/, \$/a/b$
$.c, \$/a, b$

We added (ii) because of (i) and (iii) because of (ii)

eg. (i) $E \to E+T$
$/T$
$T \to T*F$
$/F$
$F \to i$

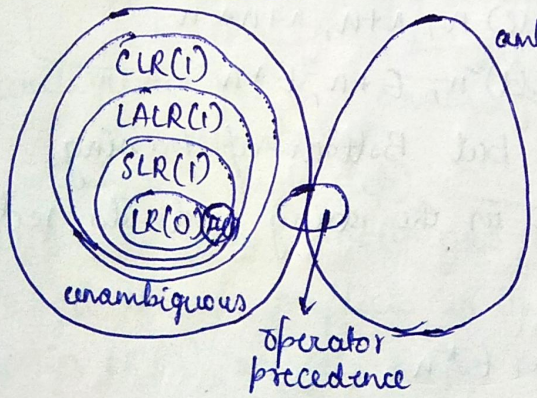
$E' \to .E, \$$
$E \to .E+T/.T, \$$
$E \to .E+T/.T, +$
~~$T \to .T*F/.F, +/\$$~~
$T \to .T*F/.F, *$

$\Rightarrow$

$E' \to .E, \$$
$E \to .E+T/.T, \$/+$
$T \to .T*F/.F, +/*/\$$

→ On completing the table, we get LR(1) satisfied without any conflicting states.

Parser Comparison:



If a grammar is LR(0), it is definitely SLR(1), LALR(1) and CLR(1). and so on.

Every LL(1) grammar is LALR(1) grammar as it is completely included in LALR(1), partially in SLR(1) and LR(0).

Q. Consider SLR(1) and LALR(1) table for a CFG,

(a) Goto of both tables may be different.
(b) Shift entries are identical in both tables.
(c) Reduce entries in tables may be different.

(d). errors entries in tables may be different.

NOTE: The no. of states in SLR(1) and LALR(1) is same.

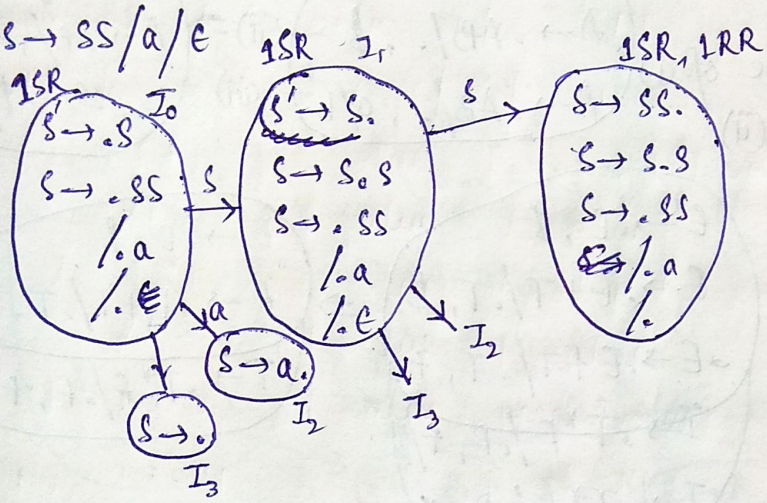
Q. $S \to CC$
$C \to cC / d$

(a) LL(1)
(b) SLR(1) but not LL(1)
(c) LALR(1) but not SLR(1)
(d) CLR(1) but not LALR(1)

Q. Find the number of SR and RR conflicts in dfa with LR(0) items.

$S \to SS / a / \epsilon$

→ 1SR



∴ 3SR conflicts, 1RR conflict.

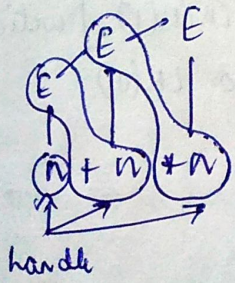
Q. $E \to E+n / E*n / n$

For the string $n+n*n$, the handles in right sentential form of reduction are

(a) $n, E+n, E+n*n$
(b) $n, E+n, E+E*n$
(c) $n, n+n, n+n*n$
(d) $n, E+n, E*n$

→ right sequential is nothing but Bottom-up parsing.
handle is the part we find in the string and its reduction to what in the LHS.

∴



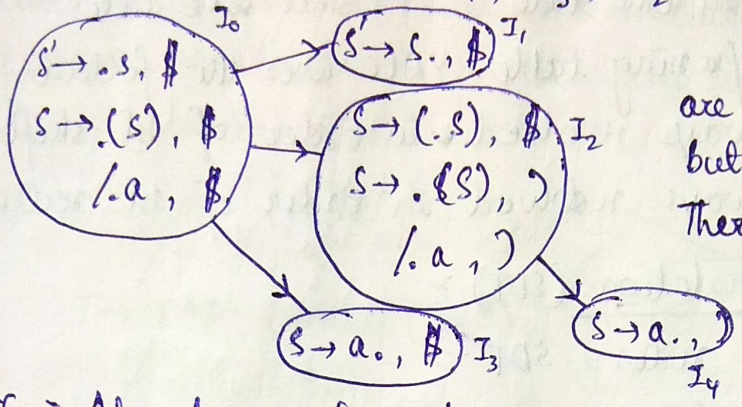
∴ $n, E+n, E*n$

$E \Rightarrow E*n$
$\Rightarrow E+n*n$
$\Rightarrow n+n*n$

Q. $S \to (S) / a$, SLR(1), LR(1), LALR(1)
n_1 n_2 n_3

(a) $n_1 < n_2 < n_3$
(b) $n_1 = n_3 < n_2$
(c) $n_1 = n_2 = n_3$
(d) $n_1 \geq n_3 \geq n_2$

→



Both the states I_3 and I_4 are having same LR(0) items but different lookaheads. Therefore, no. of states in CLR(1) will be greater.
$\therefore n_1 = n_3 < n_2$

NOTE : Advantages of ambiguous grammar over unambiguous grammar :

(i) The no. of productions are less, hence grammar is shorter and easier to represent.
(ii) It is natural to read and simpler to understand.
(iii) There is flexibility in the grammar as the associativity and precedence of the terminals are not fixed. Therefore, later at some point of time, we can specify about the associativity and precedence.

eg.

$E \to E+E$	$E \to E+T / T$
$/E*E$	$T \to T*F / F$
$/id$	$F \to id$

→ E ↺ | id → to generate 'id' (only in a single step)

ambiguous gr.

→ E ↺ | T ↺ | F ↺ | id → to generate 'id' (series of steps required)

unambiguous grs

Why do we not use ambiguous grammar for parsing table?

→ (1) We get lot of conflicts in the parsing tables. For the above eg. in LR(0), we get many conflicts in the construction of the parsing table. But if we could reduce these conflicts, then we can go for ambiguous grammar. We can use this grammar in which there will be less no. of production, less no. of states and the parsing table will be smaller but we have to resolve

the conflicts.

Conflicts are like

$E \to E.+E$
$E \to E+E.$
⋮

at some state

S/R conflict

id + id.+ id

$E \to E.+E$
$E \to E*E.$

at some state

S/R conflict

id * id.+ id

NOTE: YACC uses LALR(1) parsing table. YACC uses the following rules:

(i) SR conflicts are always resolved in order of 1st shift.

(ii) RR conflicts are always resolved in order of 1st reduce.

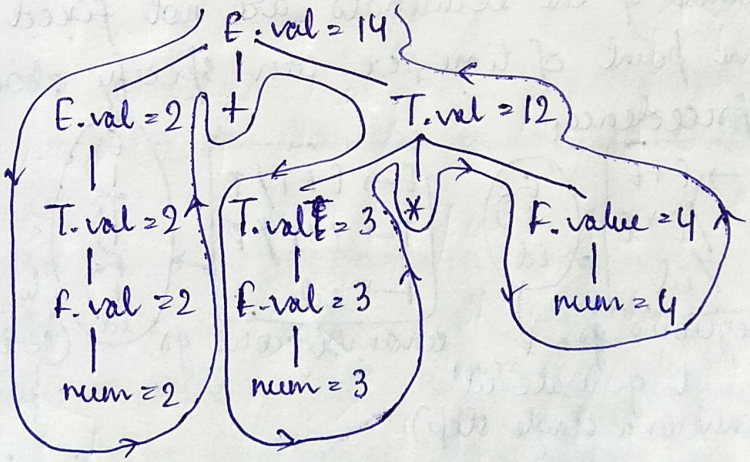
Syntax Directed Translation (SDT):

Grammar + Semantic rules = SDT

SDT for evaluation of expression,

$E \to E+T$	{E.value = E.value + T.value}	Semantic rules
/T	{E.value = T.value}	
$T \to T*F$	{T.value = T.value * F.value}	
/F	{T.value = F.value}	
$F \to num$	{F.value = num.value}	

eg. (i) 2+3*4



NOTE:

(i) Whenever there is reduction, take the production and perform the action. Attribute here is, 'val'.

(ii) In SDTs, every variable is going to get an attribute or more than one attribute or sometimes it might get even 0 attributes.

(iii) At the time parsing is being done, many other actions can be performed by the help of SDTs by the parsers. (semantic actions).

In the compiler design, we are not going to just use plane grammars. Along with the grammars we give some meaningful rules so that syntax analysis, code generation and storing of some values in the symbol table, intermediate code generation or expression evaluation can be done in parallel when the parsing is being done.

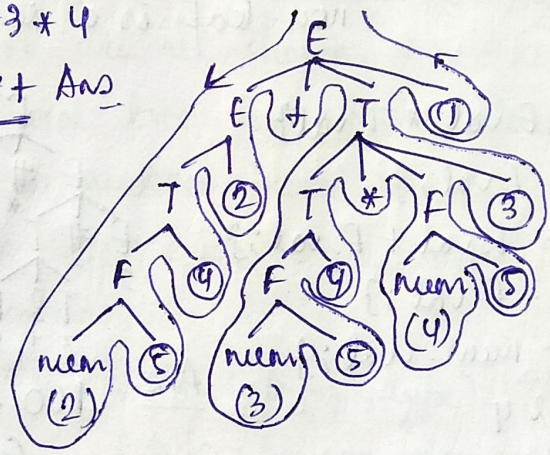
eg (ii)

$E \to E+T$	{printf("+");}	①	Semantic Actions
$/T$	{ }	②	
$T \to T*F$	{printf("*");}	③	
$/F$	{ }	④	
$F \to num$	{printf(num.lvalue)}	⑤	

The above helps to find out postfix expression for an infix exp.

$w = 2+3*4$

o/p = 234*+ Ans



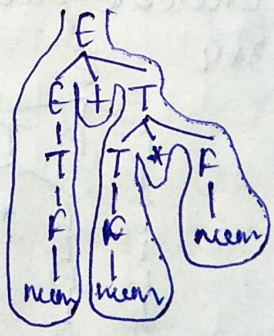
Top-down approach

NOTE: We are going to traverse the tree top-down, left to right and whenever we are going to approach semantic action, we are going to perform the action.

Bottom-up approach

The parser will try to reduce the terminals to variables and whenever they reduce, they perform the action.

o/p = 234*+ Ans

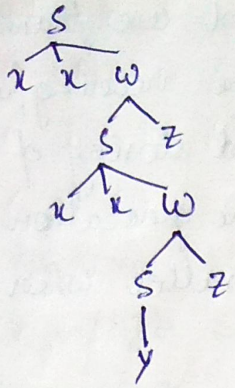


eg (iii)

$S \to xxw$	{printf(1);}
$/y$	{printf(2);}
$w \to Sz$	{printf(3);}

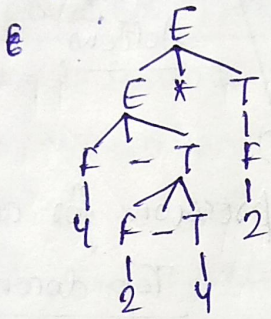
String: xxxyzz

Find the output for the above string.

→ 

O/p = 23131 Ans

eg (iv) $E \to E * T$ {E.val = E.val * T.val;}
/T {E.val = T.val;}
$T \to F - T$ {T.val = F.val − T.val;}
/F {T.val = F.val;}
$F \to 2$ {F.val = 2;}
/4 {F.val = 4;}

→ 

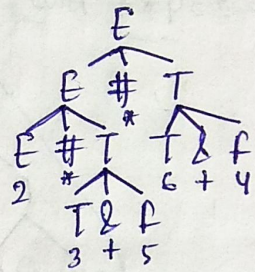
word = 4−2−4*2

Ans : 12 , ie, ((4−(2−4)) * 2)

NOTE : For bottom-up approach, every non-leaf is a reduction.

eg. (v) $E \to E \# T$ {E.val = E.val * T.val;}
/T {E.val = T.val;}
$T \to T \& F$ {T.val = T.val + F.val;}
/F {T.val = F.val;}
$F \to$ num {F.val = num.lval;}

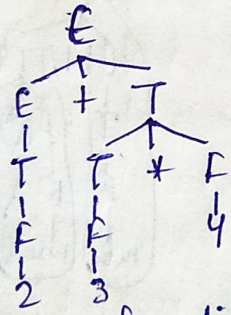
word = 2#3&5#6&4

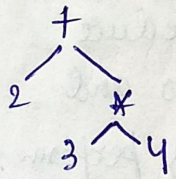


Ans : 160 , ie, (2*(3+5)) * (6+4)

NOTE : Concrete parse Tree is Abstract parse Tree :-

w = 2+3*4





If we only use the terminals to represent a parse tree, it is abstract parse tree hiding the unnecessary part.

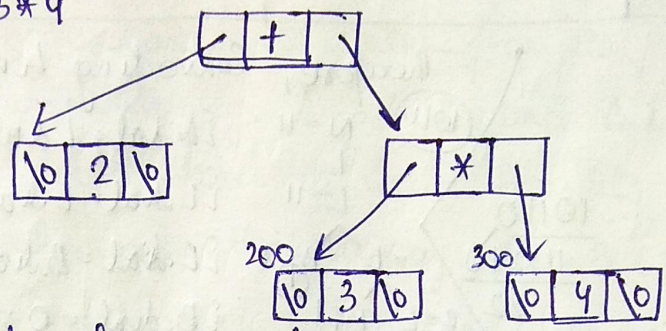
The detailed construction of the parse tree, representing everything in it, is concrete parse tree.

Therefore, if we carry out SDT on any ~~parse~~ input string then we are supposed to get the abstract parse tree.

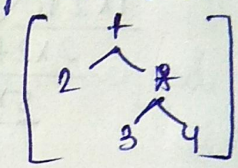
Std to build syntax tree :-

$E \to E+T$ { E.nptr = mknode (E.nptr, '+', T.nptr); }
/ T { E.nptr = T.nptr; }
$T \to T*F$ { T.nptr = mknode (T.nptr, *, F.nptr); }
/ F { T.nptr = F.nptr; }
$F \to id$ { F.nptr = mknode (null, id name, null); }

word = 2+3*4



This is DS of parse tree for the abstract parse tree.



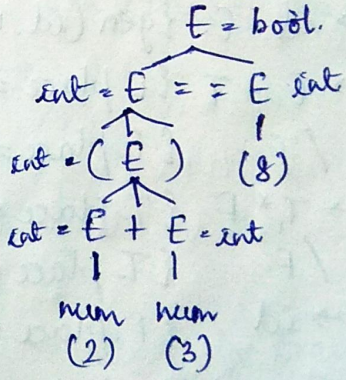
Abstract syntax tree is a parse tree in which the variables are not shown.

NOTE : Any parse tree for which values are written for the variables is anoted parse tree.

eg. $E \to E_1 + E_2$ {if((E1.type == E2.type) && (E1.type = int)) then E.type = int else error;}
/ $E_1 == E_2$ {if((E1.type == E2.type) && (E1.type = int/boolean)) then E.type = boolean else error;}
/ (E_1) { E.type = E1.type; }
/ num { E.type = int ; }
/ True { E.type = bool ; }
/ False { E.type = bool. ; }

word : (2+3) == 8
∴ O/P = True (boolean)

Parse Tree



eg. Grammar to generate all possible binary nos.

$N \to L$
$L \to LB$
/ B
$B \to 0/1$

→ **Count 1's**

{N.C = L.C;}
{L.C = L.C + B.C;}
{L.C = B.C;}
{B.C = 0;}
{B.C = 1;}

Count 0's

{N.C = L.C;}
{L.C = L.C + B.C;}
{L.C = B.C;}
{B.C = 1;}
{B.C = 0;}

Count no. of bits

{N.C = L.C;}
{L.C = L.C + B.C;}
{L.C = B.C;}
{B.C = 1;}
{B.C = 1;}

NOTE : $(11)_2 \rightarrow (3)_{10}$

$(101)_2 \rightarrow (5)_{10}$

Now,

1011

$1 \times 2 + 0$

$2 \times 2 + 1$

$5 \times 2 + 1$

11

$\therefore 10110$

$11 \times 2 + 0$

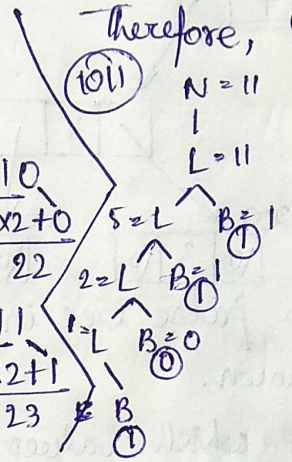
22

10111

$11 \times 2 + 1$

23

Therefore,



Converting binary to decimal,

{N.dval = L.dval;}
{L.dval = L.dval * 2 + B.dval;}
{L.dval = B.dval;}
{B.dval = 0;}
{B.dval = 1;}

eg. $(11.01)_2$

$.01 = \frac{1}{2^2} = \frac{1}{4} = 0.25$

$.11 = \frac{3}{2^2} = \frac{3}{4} = 0.75$

$N \rightarrow L_1 . L_2$ {N.dval = L_1.dval + $\frac{L_2.dval}{2^{L_2}}$}

$L \rightarrow L_1 B$ {L.c = L_1.c + B.c ; L.dval = L_1.dval + B.dval;}

/ B {L.c = B.c ; L.dval = B.dval;}

$B \rightarrow 0$ {B.c = 1 ; B.dval = 0;}

/ 1 {B.c = 1 ; B.dval = 1;}

SDT to generate 3 address code :-

$S \rightarrow id = E$ {gen(id.name = E.place);}

$E \rightarrow E_1 + T$ {E.place = newTemp(); gen(E.place = E_1.place + T.place);}

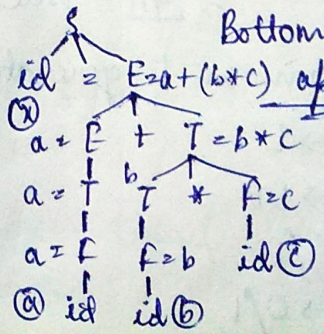
/ T {E.place = T.place;}

$T \rightarrow T_1 * F$ {T.place = newTemp(); gen(T.place = T_1.place * F.place);}

/ F {T.place = F.place;}

$F \rightarrow id$ {F.place = id.name;}

eg. $x = a + b * c$



Bottom-up approach

NOTE : newTemp helps to generate temporary var. used for the temporary evaluations and generation of resultant value for further evaluation.

The original expression containing 4 variables is dropped down to 3 expressions with the first 2 containing 3 variables and the last one with 2 variables. Therefore, each line in the 3 address code contains maximum of 3 addresses.

$$\boxed{\begin{aligned} t_1 &= b * c; \\ t_2 &= a + t_1; \\ x &= t_2 \end{aligned}}$$

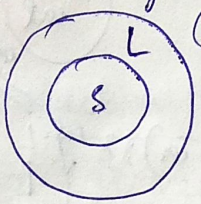
NOTE: If a node takes its value from its parent or sibling, it is called inherited attribute. eg. $A \rightarrow BCD$; and $B.i = C.i$, $D.i = A.i$

If the parent takes its value from its children, it is called synthesised attribute. eg. $A.i = B.i$;

Depending on the type of attributes, SDTs can be classified into 2 types:-

S-attribute SDT

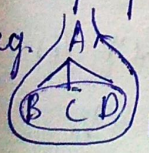
(i) uses only synthesised attributes. eg. $A \rightarrow BCD$ $(A.s = B.s;)$



(ii) Semantic actions are placed at right end of the production.
eg. $A \rightarrow BC\{\}$
also called as postfix SDT.

(iii) Attributes are evaluated during bottom-up parsing.

eg.



By the time reaching A, all values B, C, D have been evaluated.

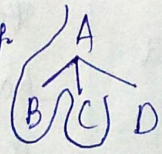
L-attribute SDT

(i) uses both inherited and synthesised attributes. Each inherited attribute is restricted to inherit either from parent or left sibling only.
eg. $A \rightarrow XYZ$ $\{Y.s = A.s;\ Y.s = X.s,\ Y.s \cancel{=} Z.s;\}$

(ii) Semantic actions are placed anywhere on RHS. eg. $A \rightarrow \{\}BC$
$/ D\{\}E$
$/ FG\{\}$

(iii) Attributes are evaluated by traversing parse tree depth first, left to right.

eg.



Here, by the time we reach 'C', all values to left of 'C' and above it, have been evaluated.

eg. (i) $A \to LM$ $\{L.i = f(A.i);\ M.i = f(L.s);\ A.s = f(M.s);\}$

$A \to QR$ $\{R.i = f(A.i);\ Q.i = f(R.i);\ A.s = f(Q.s);\}$

(a) S-attri. (b) L-attr. (c) both ~~(d)~~ none

→ $A \to LM$ (L-attribute SDT) ; ($L.i = f(A.i)$ → not S-attributed SDT)

$A \to QR$ (not-L attributed SDT, because of $Q.i = f(R.i)$;)

(ii) $A \to BC$ $\{B.s = A.s;\}$ inherited attribute

(a) S-attri. ~~(b)~~ L-attri. (c) both (d) none

⊛ SDT to add. type information into symbol table – both S attributed and L attributed :-

Suppose we have

int x, y, z ;

x	int
y	int
z	int

Then we have to store the type of the variable into the symbol table.

Below SDT is L-attributed,

Now,

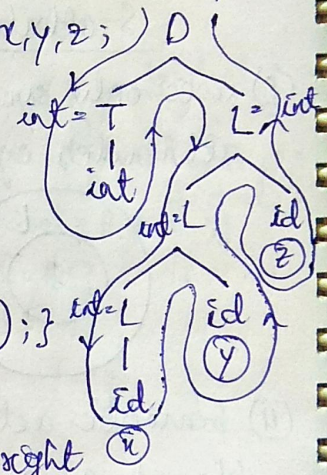
$D \to TL$ $\{L.in = T.type\}$

$T \to$ int $\{T.type = int\}$

/char $\{T.type = char\}$

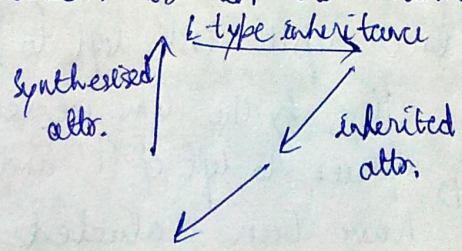
$L \to L, id$ $\{L_1.in = L.in,\ addtype(id.name, L_1.in);\}$

/id $\{addtype(id.name, L.in);\}$



NOTE: Start tracing out from ~~left~~ top to down, left to right and whenever you come across any inherited attribute, evaluate it, first time you visit it. And whenever you come across synthesised attribute, evaluate it, last time you visit it.

From the above graph, we see that the order of movement of information is in the manner,



Therefore, information has to start at some point and it has to flow up in order for it to be S-attributed SDT.

Now, in order to make the grammar S-attributed, we have to change the grammar itself.

$D \rightarrow D_1$, id {addType (id.name, D_1.type); D.type = D_1.type;}

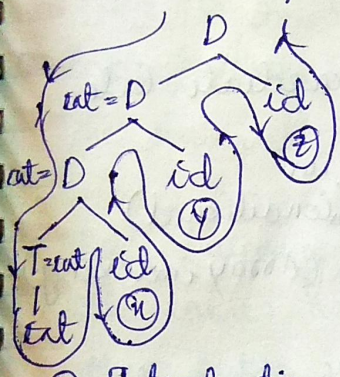
/ T_1 id {addType (id.name, T.type), D.type = T.type;}

$T \rightarrow$ int { T.type = int;}

/ char { T.type = char;}

Here, the grammar is S-attributed SDT.

NOTE: In order to make any grammar synthesised, make sure that the grammar constructed starts from the lowest level and moves upwards level by level, as it is the condition that satisfies S-attributed SDT.



Traversal is top to down and left to right.

int	x
int	y
int	z

Introduction to Intermediate Code:

Having Intermediate code is important to build new compilers for other machines. ICG is used so that till some phases, all the phases put together will generate the intermediate code and now given that intermediate code, the backend can take that code and generate the machine code.

Intermediate code means given a code, we have to represent it in some other form. There are many forms of intermediate code available.

Mainly, 2 forms that are popular are:

(i) Linear form (ii) Tree form.

Intermediate Code

eg. $(a+b)*(a+b+c)$

- Linear form
  - Postfix
  - Three Address Code
- Tree form
  - Syntax Tree
  - DAG

Postfix

$ab+ab+c+*$

Three Address Code

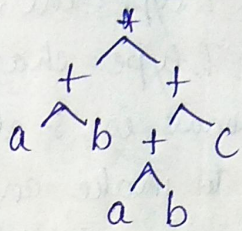
$t_1 = a+b$

$t_2 = a+b$

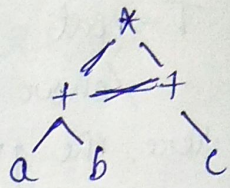
$t_3 = t_2+c$

$t_4 = t_1 * t_3$

Syntax Tree



DAG



NOTE: Three Address Code is Globally accepted and popularly used. So, if we want to convert into intermediate Code, we have the above 4 options.

∴ We mainly use the 3 address Code for Intermediate Code Generation. Types of 3 address code :-

(i) $x = y \text{ op } z$ (binary oper.)

(ii) $x = \text{op } z$ (unary oper.)

(iii) $x = y$ (assignment)

(iv) if x (rel op) y goto L

(v) goto L (unconditional)

(vi) $A[i] = x$ (Array indexing)
$y = A[i]$

(vii) $x = *p$ (pointer indexing and addressing)
$y = \&y$

Whatever we have in high-level lang., we ultimately have to convert them to one of the 7 types of 3 address code.

Various representation of 3 address code:-

eg. $(a+b)*(c+d)+(a+b+c)$

1. $t_1 = a+b$
2. $t_2 = -t_1$
3. $t_3 = c+d$
4. $t_4 = t_2 * t_3$
5. $t_5 = a+b$
6. $t_6 = t_5 + c$
7. $t_7 = t_4 + t_6$

Quadruples

	opr.	op1	op2	res.
1.	+	a	b	t_1
2.	−	t_1	−	t_2
3.	+	c	d	t_3
4.	*	t_2	t_3	t_4
5.	+	a	b	t_5
6.	+	t_5	c	t_6
7.	+	t_4	t_6	t_7

Triple

	opr.	op.1	op.2
1.	+	a	b
2.	−	(1)	
3.	+	c	d
4.	*	(2)	(3)
5.	+	a	b
6.	+	(5)	c
7.	+	(4)	(6)

Indirect triples

(i)	(1)
(ii)	(2)
(iii)	(3)
(iv)	(4)
(v)	(5)
(vi)	(6)
(vii)	(7)

Every line is a pointer to an instr.

	Quadruples	Triples	Indirect triples
Advantage:	statement can be moved around.	Space is not wasted.	statement can be moved.
Disadvant.:	too much space is wasted	Statement cannot be moved.	Two memory access.

NOTE:> 2 memory access is as 1 memory access to get the address of the statement. The 2nd memory access is to get the statement itself.

Statements cannot be moved in triples, as once the instructions have been no. and their no. have been used in some other instruction, if we change the instruction ordering then the instructions have to be changed.

* Back patching and conversions of to 3 address code:

The statement, if (a<b) then t=1 else t=0] is a 4 address code and hence has to be converted to 3 address code.

Back patching means leaving the Lables as empty and filling them later is called Back patching.

∴ (i) : if a<b goto (i+3)
(i+1) : t=0
(i+2) : goto (i+4)
(i+3) : t=1
(i+4)

if (a<b) then t=1
else t=0

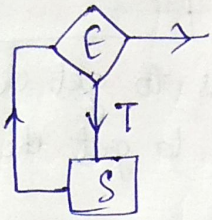
eg.(i) $\boxed{\underset{t_1}{a<b} \text{ and } \underset{t_2}{c<d} \text{ or } \underset{t_3}{e<f}}$

100. if (a<b) goto 103
101. $t_1=0$
102. goto 104
103. $t_1=1$
104. if (c<d) goto 107
105. $t_2=0$
106. goto 108
107. $t_2=1$
108. if (e<f) goto 111
109. $t_3=0$
110. goto 112
111. $t_3=1$
112. $t_4=t_1$ and t_2
113. $t_5=t_4$ or t_3

NOTE: (i) While loop is not present in 3 address code. But while loop is present in many high-level language.

∴ Converting to 3 address code:

While E do S,




```
L: if (E==0) goto L1
   S
   Goto L
L1:
```

or

```
L: if (E) goto L1
   goto last
L1: S
   goto L
last:
```

But, identifying the while loop in 3 address code is very difficult, as we will not be able to understand that whether it is a pure if statement or a looping structure. When we convert it to intermediate 3 add. code, there will be no block at all to differentiate the looping structure and it will look like plane text as there is nothing as while and curly braces in 3 add. code.

Code optimization will help us to identify such structure.

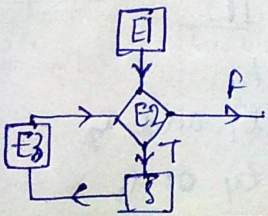
eg. (1) while (a<b) do
x = y + z;

```
L: if a<b goto L1
   goto last
L1: t = y+z
   x = t;
   goto L
last:
```

(ii) For loop is also not present in 3 address code which is present in high level language.

∴ Converting to 3 address code:

for (E1; E2; E3)



eg. for (i=0; i<10; i++)
a = b+c;

```
   i=0;
L: if (i<10) goto L1
   goto last
L1: t1 = b+c
   a = t1
   t = i+1
   i = t
   goto L
last:
```

Here, too, the identification of looping structure is difficult. This problem is resolved again in code optimization.

(iii) Switch statements are also high level language code not present in 3 address code.

∴ Converting to 3 address code:

```
switch (i+j) {
    case (i) : a = b+c;
               break;
    case (ii) : p = q+r;
               break;
    default : x = y+z;
               break;
}
```

```
t = i+j
goto test
L1: t1 = b+c;
    a = t1;
    goto last
L2: t2 = q+r;
    p = t2;
    goto last
L3: t3 = y+z
    x = t3
    goto last
test: if (t==1) goto L1
      if (t==2) goto L2
      goto L3
last: ~~~
```

In switch, we do not have looping structure. Therefore, execution and evaluation of 3 add. code is very easy in this case.

(iv) 2 dimensional array to 3-address code:-

We store the 2 dimensional array as 1-dimensional array in the memory using Row Major Order or Column Major Order. We usually use the RMO for storage representation.

Suppose, we have an array A : 10×20

$\boxed{A : i \times j}$

$x = A[y, z]$

$RMO = (((y * j) + z) * size) + \text{base address}$

3 add. code by RMO:

$t_1 = y * 20$

$t_2 = t_1 + z$

$t_3 = t_2 * 4$ // size (4)

$t_4 =$ base address of A

$x = t_4 [t_3]$ // take the base, add the offset

A[y, z]

$$\begin{bmatrix} 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \end{bmatrix}_{4\times4}$$

$A[2,3] = (((2 \times 4) + 3) * 2) + 100$

$= (11 \times 2) + 100$

$= 122$

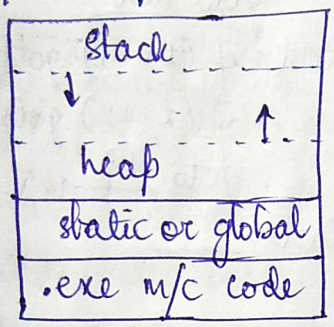
RMO: 00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33

NOTE: Whatever be the dimension of the original array, it has to be converted into 1 dimensional array for storage in the memory.

Runtime Environments :-

It means that, when we run the program, what is the support that we need from the operating system and how the program is loaded.

When we run a program, it is called process and the process is managed by the operating system, ie, called operating system process management.



Memory storage space.

For the stack, the storage grows downward and for heap the storage grows upwards. We have not specified contiguous mem. locations to stack and heap as they are used for dynamic memory allocation and we do not know which of the 2 is going to occupy more space. A program can use lot of heap space and lot of stack space but we do not know which space the program requires.

If the entire space is over, we get memory or segmentation error and error will be printed according to the O.S. The space depends on the OS. And stack, heap, etc are the runtime environments which are provided by OS to the process these days. OS, presently, provide us with 3 types of storage technique

(i) Static (ii) heap (iii) stack.

But during evolution, earlier people used either of the 3 techniques.

Storage Allocation Strategies :-

1. Static : (i) Allocation is done at compile time.

(ii) Binding do not change at runtime, ie, a variable's address once fixed, it cannot move out of that add-

(iii) One activation record per procedure, ie, at compile time we cannot decide how many times the function has to be called. Therefore, if we have 10 function, we allocate 10 activation records and then every func. will be called only once at any time.

Disadvantage: (i) Recursion is not supported, we can call the func. only once.

(ii) Size of data objects must be known at compile time.

(iii) Data structures cannot be created dynamically. So, there is no support of heap. The space is occupied by the program initially.

Advantage: If we use static memory, the variable stays alive throughout the program.

2. Stack: (i) Whenever a new activation begins, activation record is pushed on to the stack and whenever activation ends activation record is popped off.

(ii) Local variables are bound to fresh storage, ie, the local variable will be created as many times as we call the function and each time it will be created in a different stack space.

Disadvantage: Local variables cannot be retained once the activation ends.

3. Heap: Allocation and deallocation can be done in any order. which means when the program is running, we can call malloc and allocate it and we can call free and deallocate it. We can do it in any order.

Disadvantage: Heap management is overhead, as when we

free some space in the storage memory, it leads to generation of a hole and because of the hole, the management is going to be difficult. This is handled by memory mgmt. of OS. eg. In CS, we ~~use~~ follow First Fit because it is faster.

Summary :-

Activations can have,

(i) permanent lifetime in case of static allocation.

(ii) nested lifetime in case of stack allocation. (push/pop)

(iii) Arbitrary lifetime in case of heap allocation.

Introduction to Code Optimization :-

It is the phase after ICG. The main purpose of this phase is reducing the no. of lines of the program. Most of time running a program is spent on blocks containing loops (iterative code). Therefore, reducing the no. of lines in the loop will reduce the overall time of the program. And hence, most of the time is spent on reducing the redundancy inside the loops or decreasing the no. of lines in the loop.

Optimization of are 2 types :-

(i) Machine Independent - we need not worry about the type of machine, the compiler is working on, it can be performed on every machine.

(ii) Machine dependent - depending on the features of the m/c, we can apply the optimizations.

Types of Machine independent optimization :-

(i) loop optimizations : (a) code motions (or) frequency reduction.

(b) loop unrolling

(c) loop jamming

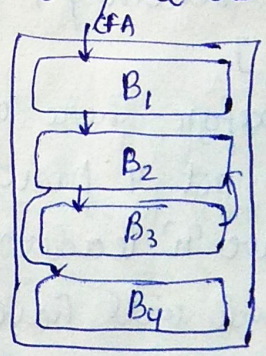
(ii) folding:- constant propagation.
(iii) Redundancy elimination.
(iv) strength reduction.

Types of Machine dependent optimizations:
(i) Register allocation.
(ii) use of addressing modes.
(iii) peephole optimization: (a) Redundant load/store.
(b) flow of control optimizations.
(c) strength reduction.
(d) use of m/c idioms.

Machine Independent optimizations:-

(i) Loop optimizations:
(a) To apply loop optimizations, we must 1st detect the loops.
(b) For detecting loops, we use control flow analysis (CFA) using Program flow graphs (PFG).
(c) To find PFG, we need to find the basic blocks.
A Basic Block is a sequence of 3-address statements where control enters at the begining and leaves only at the end without any jumps or halts (no goto,s).

For eg. suppose we have got 4 basic blocks.
After finding the PFG, we have to do CFA, ie, how the control is entering each block and how it is leaving to some other block.
And if we find any cycle for the control flow in the graph, these cycles will indicate loops.
And once we find the loops, we can apply loop optimization.



Program Flow Graph

(a) Finding the Basic Blocks:-
In order to find the basic blocks, we need to find the leaders in the program. then a basic block will start from one leader

to the next leader but not including the next leader.

eg. If line no. 1 is a leader and the next leader is line no. 10 than a block is from line no. 1-9.

* Identifying leaders in a basic block :- (Algorithm)

(i) 1st statement is a leader.

(ii) Statement that is target of conditional or unconditional (jump or goto) statement is a leader.

for eg. if (...) goto 200, then line no. 200 is a leader.

or

goto 300 ← unconditional

(iii) Statement that follows immediately a conditional or unconditional statement is a leader.

eg. of loop optimizations :-

```
fact(x){//factorial of x
  int f=1;
  for (i=2; i<=x; i++)
    f=f*i;
  return f;
}
```

← High Level Language

3 address code :-

```
* 1. f=1;                 B1
  2. i=2;
* 3. if (i>x) goto 9      B2
* 4. t1=f*i;
  5. f=t1;
  6. t2=i+1;              B3
  7. i=t2
* 8. goto (3);            B4
* 9. goto calling program
```

Therefore, there are 4 leaders here. We got 4 basic blocks. Incase, we have 'n' leaders, we will get 'n' basic blocks, as every basic block will have to start with a leader. Therefore, the no. of basic blocks will depend upon the no. of leaders.

Now, the program flow graph,

Therefore, we find out a cycle b/w B_2 and B_3. This cycle is a loop. The statements in B_2 and B_3 are the statements present in the loop. And once loops are found, we can apply the loop optimization.

Optimization
↓
loops
↓
CFA(PFG)
↓
Basic Blocks
↓
leaders

By loop optimizations,

(i) we want to decrease the no. of lines in loop.

(ii) we want to reduce the no. of loop, ie, eliminating a loop completely or decrease the no. of times a loop is executed.

Types of loop optimizations :-

(a) frequency reduction : moving the code from high frequency region to low frequency region, also called code motion.

eg.
```
i=0;
while (i<5000) {
    A = sin(x)/cos(x) * i;
    i++;
}
```
⟹ Converting
```
t = sin(x)/cos(x);
while (i<5000) {
    A = t * i;
    i++;
}
```

Here, the value of $\frac{\sin x}{\cos x}$ won't change as it is constantly repeating on the value of 'x' and is present in high frequency region repeating uselessly. Therefore, we can move this code from the high frequency region to a low frequency region, compute it there once and put back the result.

Hence, we have reduced the no. of divisions to be performed.

(b) Loop unrolling : it means we could reduce the no. of times, we enter into the loop or reduce the no. of times, we do a comparison.

eg.
```
while (i<10) {
    x[i] = 0;
    i++;
}
```
⟹ Converting
```
while (i<10) {
    x[i] = 0;
    i++;
    x[i] = 0;
    i++;
}
```

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

Here, instead of initializing 1 with 0, we could have taken 2 no. at once for initializing them to 1.

(i) <u>Loop jamming</u>: it means combining 2 loops together or decreasing the no. of loops.

eg.
```
for (i=0; i<10; i++){
    for (j=0; j<10; j++)
        x[i,j] = 0;
}
for (i=0; i<10; i++)
    x[i,i] = 0;
```
$\xrightarrow{\text{Converting}}$
```
for (i=0; i<10; i++) {
    for (j=0; j<10; j++){
        x[i,j] = 0;
        x[i,i] = 0;
    }
}
```

Instead of writing the 2nd loop, we could have simply combined the 2 loops together and mentioned their functionalities without changing their meanings. Therefore, no. of loops are eliminated here.

(ii) <u>Folding</u>: Replacing an expression that can be computed at compile time by its values.

eg. $2+3+C+B = 5+C+B$

We try to do it till the compile time only and not leave it for the runtime to reduce the possibility of the evaluation to fall into a loop and be repeated many times.

(iii) <u>Redundancy elimination</u>: It is same as DAG, ie, if an expression is already evaluated and next time we call for the same expression, then DAG is going to return us the particular pointer for the expression which has already been evaluated. eg. $A = B+C$

$D = 2+B+3+C$

$D = \underbrace{(2+3)}_{5,\text{ by folding}}+A$ [$\because A = B+C$, already evaluated]

(iv) <u>Strength reduction</u>: replacing a costly operation by a cheaper one.

eg. $B = A * 2$ Multiplying by 2 is nothing but left shifting that no. by 1. which can be performed much faster than multiplying

$B = A \ll 1$

∴ Left shifting is easy.

<u>Algebraic Simplification</u>: eliminating trivial statements.

eg. $A = A + 0$
$x = x * 1$] As the result won't affect the no. itself and it will remain the same.

(v) <u>Machine-dependent optimizations</u>:

Depending on the architecture and the platform, we perform the optimizations. by the facilities provided by the platform.

(i) <u>Register allocation</u>: depends on the no. of registers present and depending on that we can do the allocations. Therefore, purely dependent on the architecture of the machine.

2 types:- (a) local allocation (b) global allocation.

(ii) use of addressing modes.

(iii) <u>Peephole optimization</u>:

(a) Redundant store and load elimination.

eg. $x = y + z$ — MOV y, R_0
— add z, R_0
— MOV R_0, x
— (1.)

$a = b + c$
$d = a + e$

MOV b, R_0
add c, R_0
$\boxed{\text{MOV } R_0, a \quad \text{MOV } a, R_0}$
add e, R_0
MOV R_0, d — (2)

For 1. only 3 codes are generated. But for 2., each statement has 3 codes to be run and total of 6 codes are run independently. When the 2 codes are combined then there is a redundancy occuring which is a load store problem. (loading and storing of a, here), which has to be eliminated.

This is optimization.

(b) flow of control optimization :- sometimes control flow in a redundant manner.

Avoid jumps on jumps

eg. L1 : jmp ~~L2~~ L4

L2 : jmp L3

L3 : jmp L4

unnecessary jumps, directly jump to L4 to avoid such jumps.

eliminate dead code

eg. # define x 0

if (x) {

X ↕ Dead Code

}

as 0 is a const. and it is never going to change, so instead of compiling the code, we could have simply left it.

(c) <u>Strength reduction</u> : already discussed.

(d) Use of machine idioms

eg $i = i+1$ | Mov R_0, i; add R_0, 1; Mov i, R_0 } inc i

If our processor has such instruction set, we need not perform the mov and add operations to perform i+1 and do 3 instr. Instead do inc i and perform only 1 instruction.