Compiler Design

Urvash Shende
20051627

Compiler Design

What is a compiler?

→ It is a program which converts the source code into target code.

→ It converts high level language to machine understandable language

Source Code (High level language)

↓

Compiles

↓

Target Code (Lower level language)

Input → System → output

→ The compiler executes the code line by line.

→ It generates error report for the whole program, after the rectification of program, target code is generated then the input is given to the target code to generate output.

→ Interpreter :- It takes the source code and the error code simultaiously and generate the output by input output mapping.

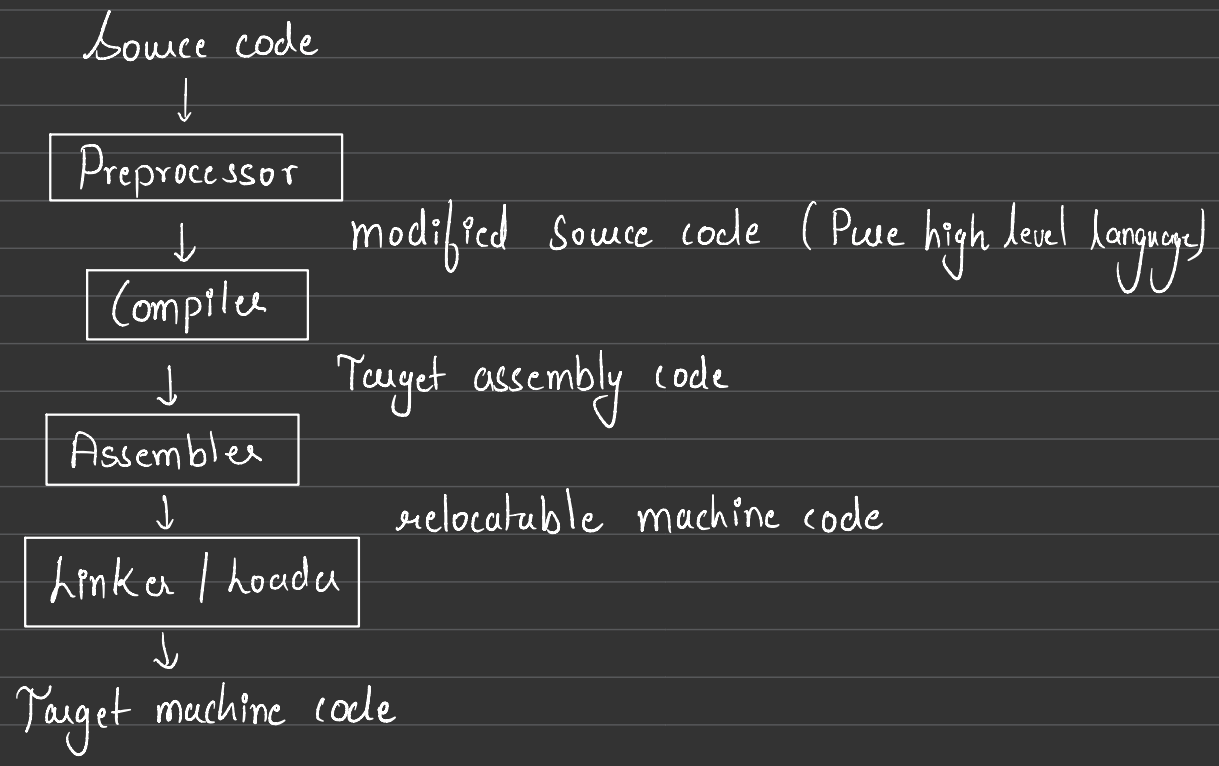
i/p → Source code →

Interpretor → o/p

→ Compiler is faster than interpreter but interpreter gives better error report than compiler.

• Compiler Language processing System

→



→ The preprocessor converts the source code into modified source code (pure high level language) by excluding the macros and excluding the header files.

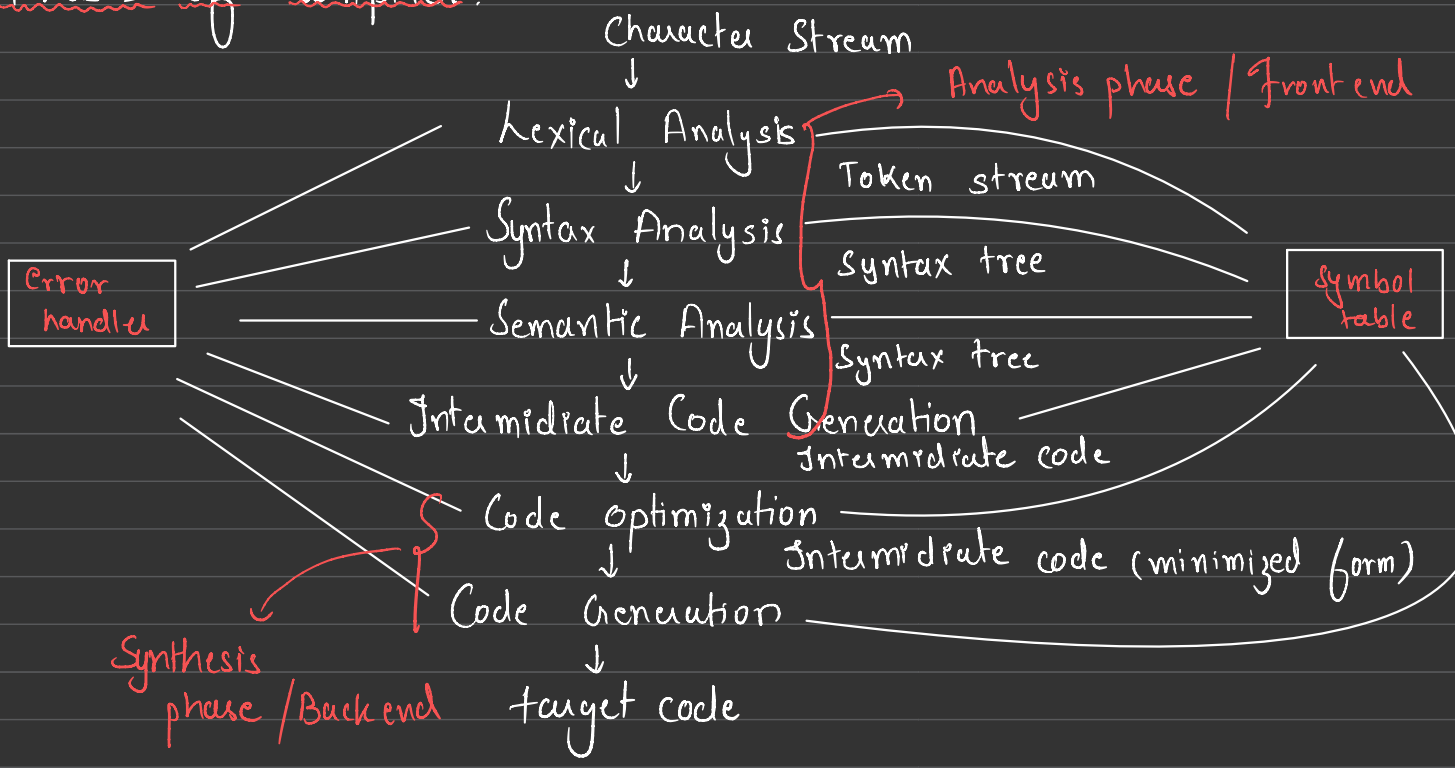
→ The assembler converts this code to relocatable machine code. (The code which doesn't have a memory address)

→ The linker links different object files by putting memory locations to the relocatable machine code.

→ The loader combines all this together and store it in the memory and the target code is generated.

Phases of Compiler:

⇒



Lexical analysis phase:

→ This phase converts the input character stream to lexemes are maped into token.

→ The token is in the form < token_name, attribute_value >

token_name: name of the lexem/symbol

attribute_value: location of that symbol in the symbol table

→ Symbol table is a space where when character are inputed and variables are stored.

Syntax analysis phase:

→ This phase associates with the grammatical representation and structure to the tokens and frame or construct the syntax tree (parse tree)

•) Semantic analysis:

→ The main task of this phase is type checking. It find the matching operands for each operator.

Eg: In array if instead of int value, floating values are used as index then the semantic analysis phase will report an error.

⇒ Type casting

```
int a = 10;
float b = 10.5;
```

•) Intermediate code generation

→ It converts the resultant syntax tree in the semantic phase to an intermediate code by using 3-address coding.

•) Code optimization

→ It converts the output of intermittate code generation phase to optmised code ie by removing unnecessary temp variable.

•) Code Generation → It generates the final target code from the intermidiate code.

Q) position = initial + rate * 60

→ List and explain the different phases of compiler with an output where position and rate are of floating type variable.

for the given expression.

11/1/23

End Lect - 2

Solution :

Lexical analysis

Symbol table

1	Position
2	initial
3	rate

→ Input for the Syntax Analysis

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

↓ Position ↓ initial ↓ rate

Variables will be stored in the symbol table

precedence = ^ > *, / > +, - , right to left

associativity = *, /, +, - → left to right

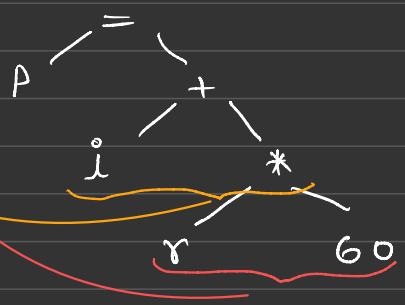
^ → right to left

Leaf node signifies expressions.

from bottom to top evaluation.

∴ Syntax tree → Tokens are the input for this.

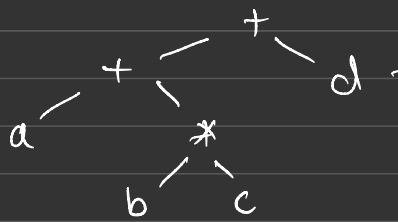
position = initial + rate * 60



Construction starts from * as it has the higher precedence

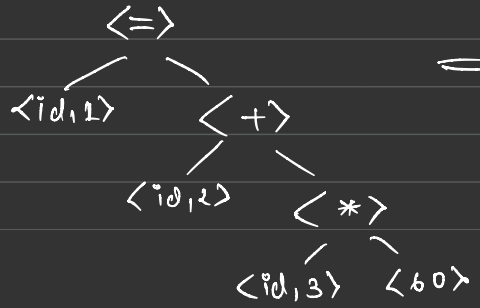
Example :

a + b * c + d



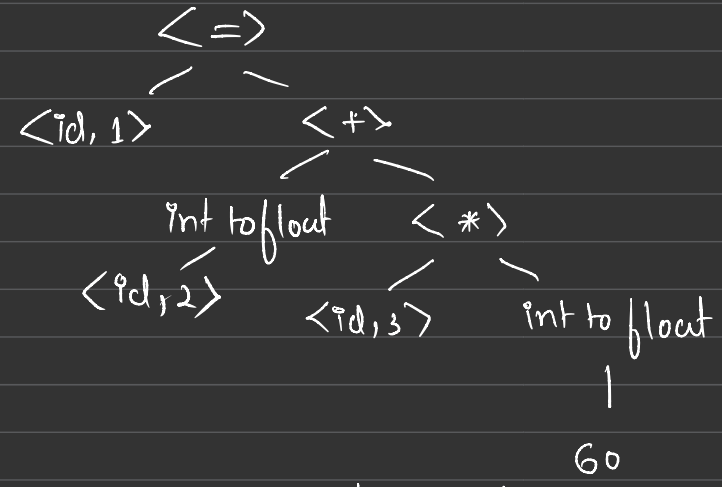
→ as d is on the right side it will get assigned in the right

Syntax tree (Output for Syntax Analysis)



⟹ Input for semantic Analysis

Input for Semantic analysis.



Input to intermidiate code generation

Intermidiate code output.

Temp variable → T_1 = Int to float (60)
T_2 = id3 * T_1
T_3 = id2 + T_2
id1 = T_3

•) Code optimization

→ t_1 = id3 * int to float (60)
id1 = id2 + t_1

→ Code Generation

```
LOAD   R1  id3
MUL    R1 , R1 , #60
LOAD   R2 , id2
ADD    R2 , R2 , R1
Store  id1 , R2
```

Q) Show the output of each phase for the expression

s = a + b * c + 20

Here variable a, b, c are of floating type and s is of integer type

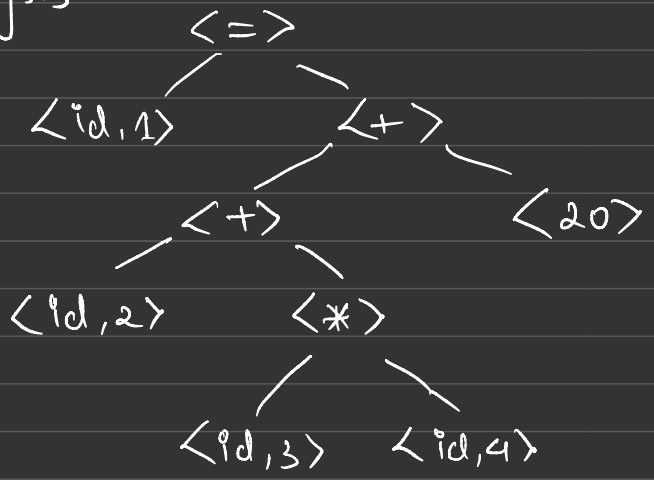
Lexical analysis

Symbol table

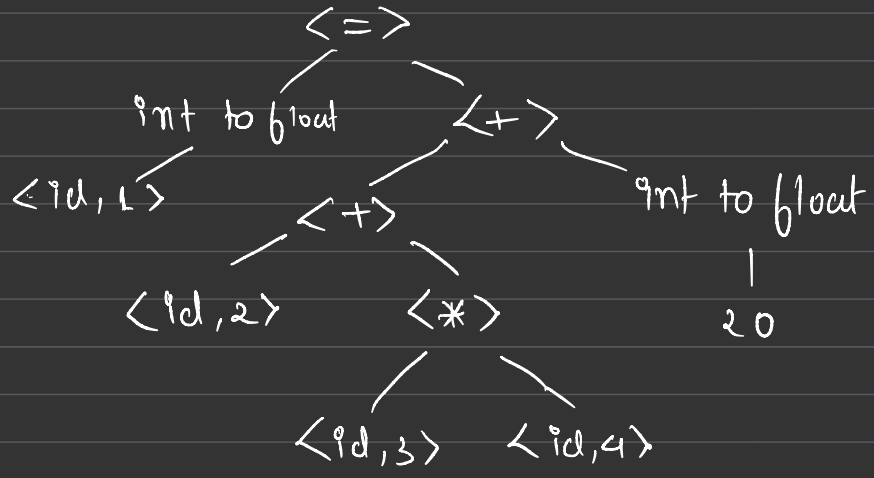
1	s
2	a
3	b
4	c

<id,1> <=> <id,2> + <id,3> <*> <id,4> <+> <20>

Syntax analysis



Semantic analysis



Intermidiate code generation.

$T_1 = \langle id,3 \rangle * \langle id,4 \rangle$
$T_2 = \langle id,2 \rangle + T_1$
$T_3 = \text{int to float } (20)$
$T_4 = T_2 + T_3$
$T_5 = \text{int to float } \langle id,1 \rangle$
$T_5 = T_4$

Code optimization

only 1 operation can happen on the right side in optimization.

$\Rightarrow$

$T_1 = id,3 * id4$
$T_2 = id2 + T_1$
$T_3 = T_2 + inttofloat(20)$
$T_4 = int\ to\ float\ (id2)$
$T_4 = T_3$

Code Generation

$\Rightarrow$

```
LOAD   R1   id3
LOAD   R2   id4
MUL    R1   R1  R2
LOAD   R3   id2
Add    R3   R3 , R1
Add    R3   R3 #20
Store  id1  R3
```

Q1) S = X/Y - Z + 20.0 , X, Y = float , Z, S = int

→ lexical Analysis

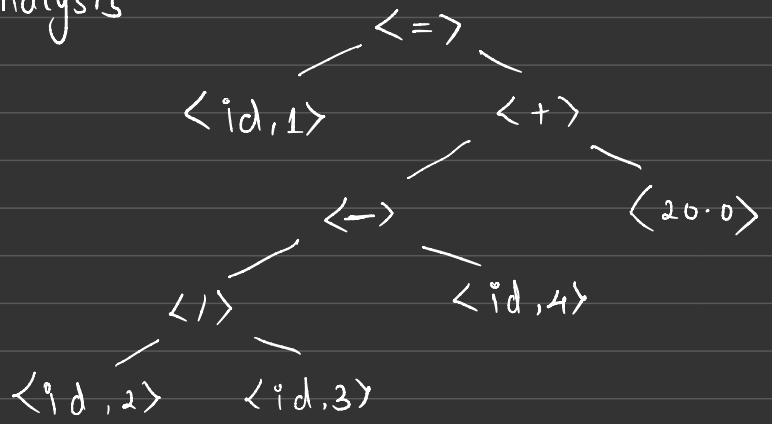
→ <id, 1> <=> <id, 2> </> <id, 3> <-> <id, 4> <+> <20.0> (Tokens created)

1	S
2	X
3	Y
4	Z

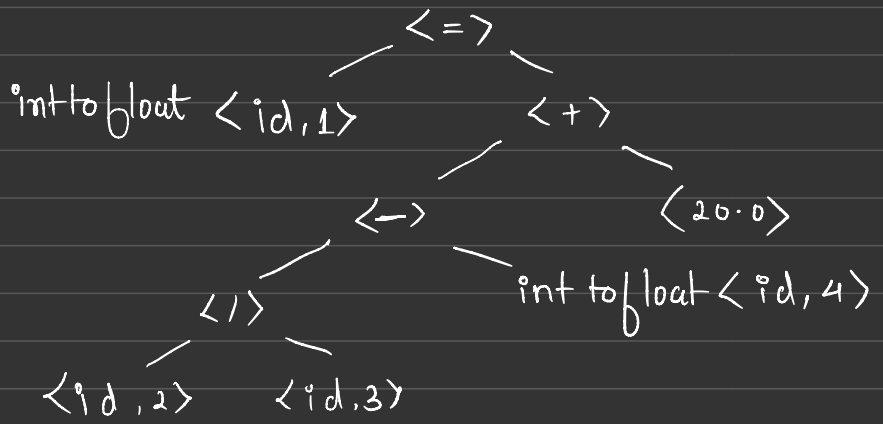
Symbol table → id 1, id 2, id 3, id 4

→ Syntax Analysis

→



→ Semantic Analysis



→ Intermidiate Code Generation.

$T_1 = \langle id,2 \rangle \; / \; \langle id,3 \rangle$
$T_2 = int\ to\ float\ \langle id,4 \rangle$
$T_3 = T_1 - T_2$
$T_4 = T_3 + 20.0$
$T_5 = int\ to\ float\ \langle id,1 \rangle$
$T_5 = T_4$

→ Code Optimization

$T_1 = id2 \; / \; id3$
$T_2 = T_1 - int\ to\ float\ (id,4)$
$T_3 = T_2 + 20.0$
$T_4 = int\ to\ float\ (id,1)$
$T_4 = T_3$

→ Code Generation

```
Load  R1  id2
Load  R2  id3
DIV   R1  R1  R2
Load  R3  id4
Sub   R1  R1  R3
Add   R1  R1  #20
Store id1 R1
```

Q2) a = b * c / d + e - 60.0 , b, c, d → float a, e → int

1) lexical Analysis

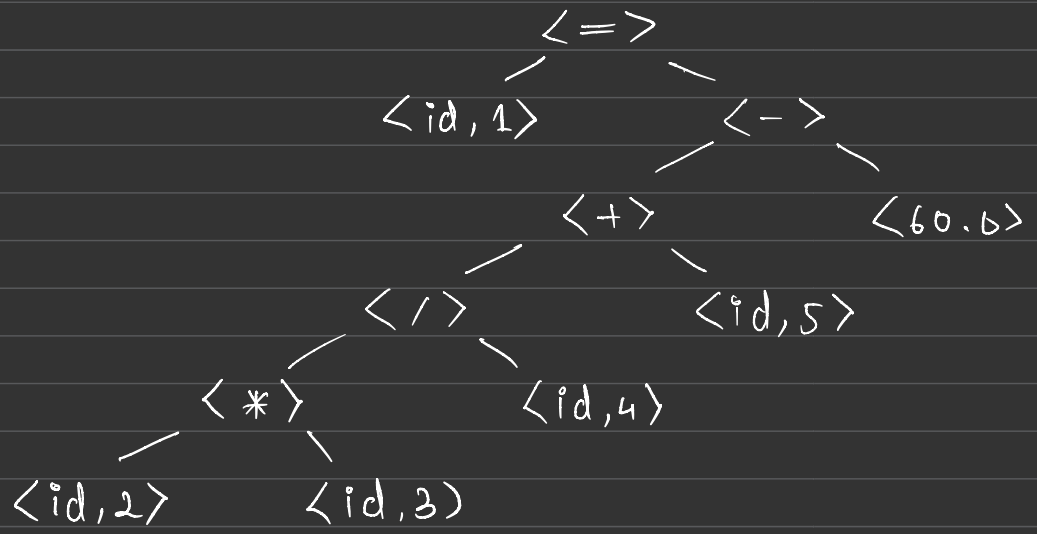
Symbol table

1	a
2	b
3	c
4	d
5	e

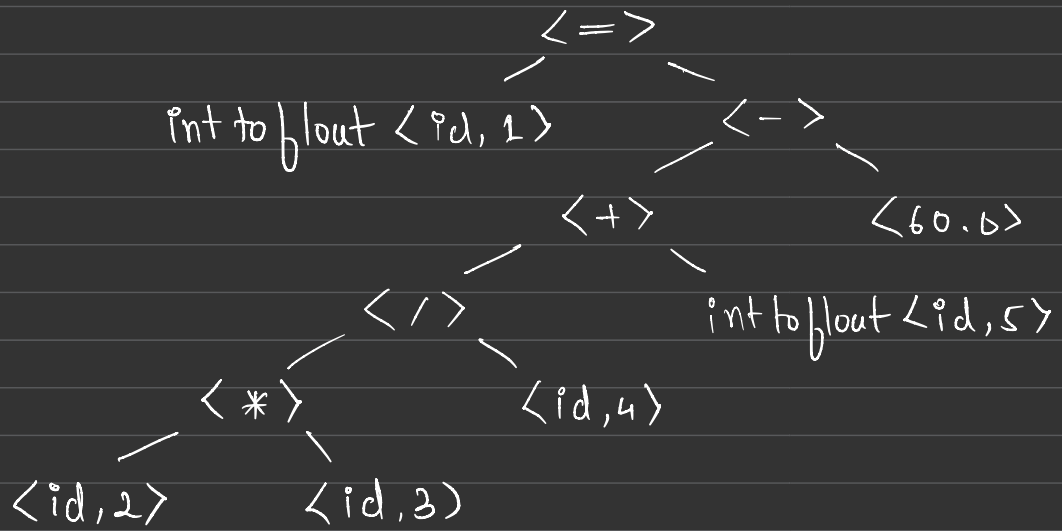
→ id 1 → int
→ id 2
→ id 3
→ id 4 } float
→ id 5 → int

→ <id, 1> <=> <id, 2> <*> <id, 3> </> <id, 4> <+> <id, 5> <-> <60.0>

2) Syntax Analysis



3) Semantic Analysis



12/1/2023

End Lect-3

•) Intermidiate Code Generation

$T_1 = \langle id,2 \rangle * \langle id,3 \rangle$
$T_2 = T_1 / \langle id,4 \rangle$
$T_3 = \text{inttofloat} \langle id,5 \rangle$
$T_4 = T_2 + T_3$
$T_5 = T_4 - 60.0$
$T_6 = \text{inttofloat} \langle id,1 \rangle$
$T_6 = T_5$

•) Code Optimization

$T_1 = \langle id,2 \rangle * \langle id,3 \rangle$
$T_2 = T_1 / \langle id,4 \rangle$
$T_3 = T_2 + \text{inttofloat} \langle id,5 \rangle$
$T_4 = T_3 - 60.0$
$T_5 = \text{inttofloat} \langle id,1 \rangle$
$T_5 = T_4$

•) Code Generation

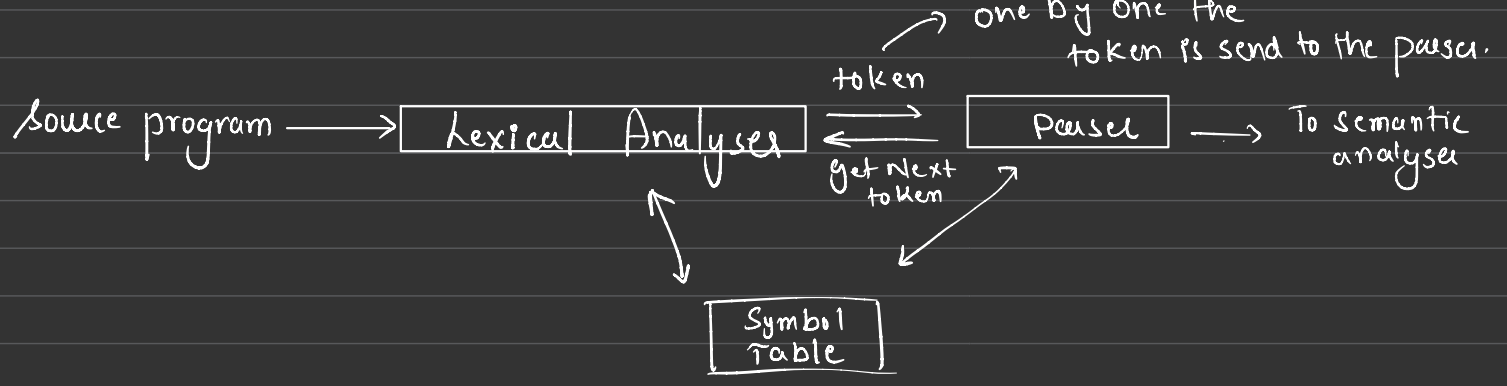
```
Load  R1  id2
Load  R2  id3
MUL   R1  R1 R2
Load  R3  id4
DIV   R3  R1 R3
Load  R4  id5
Add   R4  R3 R4
SUB   R4  R4 #60
Store id1 R4
```

Chapter – 2 Lexical Analysis Phase / Analyzer.

→ The main task of this phase is to read the input character of source program then group them into lexemes and produce the output as a sequence of tokens for each lexemes.

→ Now this stream of tokens are send to the parser for syntax analysis

→ The lexical Analyser interact with the symbol table. when it found that the lexeme is an identifier then it enters the information of the identifier inside the symbol table.



Interaction between LA and parser.

Example

$a = a++;$

$\therefore \langle a \rangle \langle = \rangle \langle a \rangle \langle + \underset{\text{wait for next +}}{\downarrow} + \rangle \langle ; \rangle$

the get next token takes the next char from input and produce the token

Lexical analyser vs parsing

→ 1) It simplifies the design
2) Compiler efficiency is improved
3) Compiler portability is enhanced

} (Parser)

•) Token, patterns and lexemes.

Token	Sample lexemes
if	if
else	else
Comparision	<=, >= , !=
id	Score (Name of identifier), D2, a
number	60, 3.14, 6.023979
Literals	" ", ; , , , :

PrintF ("Total = %d \n", score);
PrintF → id; "Total = %d \n" → Literal; score → id; ( ) ; → Literals

→ Lexemes of a particular pattern is called token

→ pattern is the description of the form that the lexemes of the token follows.

•) lexemes → It is a sequence of characters in the source program that matches with the pattern for a token.

→ Keywords of an operator → int a = 10;
int → keyword; a → identifier; = → operator; 10 → Number; ; → literal.

•> Find the total number of tokens (Compulsory 1 mark Questions)

printf (" Enter number ") ;

① ② ③ ④ ⑤

→ int f (int x , int y)
{
printf (" %d " , x >= y ? x : y) ; } → 24
}

→ int max (x, y)
int x, y ;
/* find max of x & y */ → Ignores comment lines & blank spaces.
{ return (x > y ? x : y) ;
} → 25

main ( ) {
int a = 10, c = 0 ;
int b = 20 ; → 32
c = a + b ;
printf (" sum = %d " , c) ;
}

Switch (input value) {
case 1 : b = c * d ; break ; — 26
default : b = b ++ ; break ;
}

Switch (input value) {

case 1 : b = c x d ; break ;
case 2 : x = y * z ; break ; — 37
default : b = b ++ ; break ;

}

•) lexical errors

→ These are the errors detected during lexical analysis phase

1) exceeding the length of identifiers or numeric constants. (eg int a = 40.0)
2) Appearance of illegal characters (eg int +a)
3) Unmatched strings (eg char a[3] = table)

•) Recovery (Error Recovery)

1) Panic mode recovery
2) Input buffering

1) PMR → In this method successive char from the input are removed one at a time until a designated set of synchronizing token is found.

→ Synchronize tokens are delemetres (; , { })

```
int a = 10;
int b = 20;
printf("%d", a);
printf("%d %d, a, b);
```

example

will stop here & ignore a, b till we reach ;

The advantages is that it is easy to implement and it gauntees that it will not go into infinite loop.

The disadvantage is that a considerable amount of input characters are skiped without checking it for additional errors.

2) Input buffering

Lexical analyser reads the source code line by line and from left to right. So this characters are stored in a buffer

Using input buffering as the method to check how this buffer is working and stores the input characters.

There are pointers used to store the elements in the buffer
1) Begin pointer
2) forword pointer.

example : int i, j;
i = i + 1;
j = j + 1;
↓ bp

Buffer

↑
bp

Two methods are used in input buffering

1) one buffer scheme → One buffer is used to store input char.
2) two buffer scheme → two buffers are used

problem is if the length of the lexemes is very very long then it crosses the buffer boundary.

So the input characters get overridden in the buffer

When the 1st / current buffer is filled the inputs are stored in the second buffer.

To identify the boundary of the buffer it uses a string eof (end of buffer) OS / sentinals.

→ Contex free grammer
→ left most deri
→ right most deri

Chapter-2: Syntax Analysis

Grammer: Rules and regulation to write automata

$G = \{ V, T, P, S \}$

V: Variable, T: Terminal, P: Production, S: Start symbol

A grammer is said to be context free when $A \rightarrow B$

A: Variable

B: Any combination of variable or terminals.

$(V+T)^*$

$S \rightarrow aA \mid a$

$A \rightarrow aA \mid \epsilon$

} Grammer

Let aaaa be a part of grammer.

$S \rightarrow aA$
$\rightarrow aaA$
$\rightarrow aaaA$
$\rightarrow aaaaA$
$\rightarrow aaaa\epsilon$

} LMD

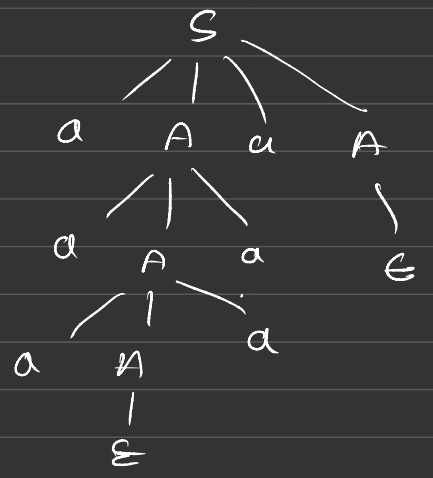
Parse tree Representation.

Q) $S \to aAaA \mid a$
$A \to aAa \mid \epsilon$

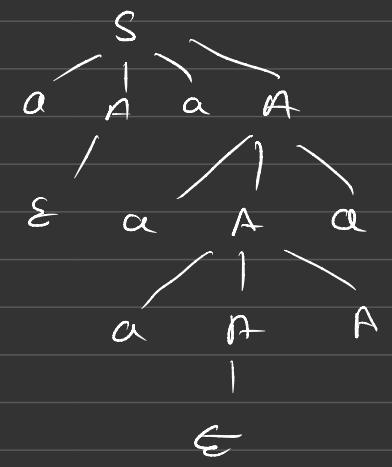
Construct aaaaaa from given grammer.

$S \to aAaA$
$\to aaAaaA$
$\to aaaAaaaA$
$\to aaa\epsilon aaaA$
$\to aaa\epsilon aaa\epsilon$

LMD
highlighted will only change



$S \to aAaA$
$\to aAaaAa$
$\to aAaaAaaa$
$\to aA\epsilon aaAaaa$
$\to a\epsilon aaa\epsilon aaa$
$\to aaaaaa$



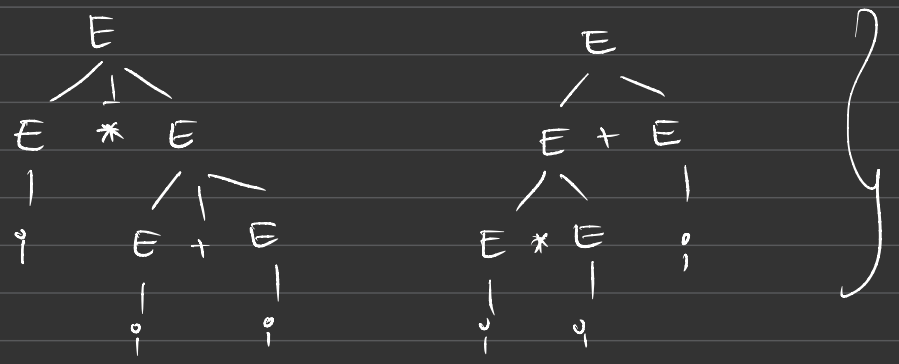
• Ambigious Grammer.

A grammer is Ambigious if a single string of a grammer or language can be represented by more than one parse tree.

$E \to E+E \mid E*E \mid i$

$i * i + i$ AMD → $E*E$
$E*E+E$
$E*E+i$
$E*i+i$
$i*i+i$

LMD $E \to E*E$
$i*E$
$i*E+E$
$\to i*i+E$
$\to i*i+i$



Ambiguous.

Q $S \rightarrow aS \mid Sa \mid a$ $\quad w = aaaa$

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

$w = abab$

CMD

$S \rightarrow aS$

$S \rightarrow aaS$

$S \rightarrow aaaS$

$S \rightarrow aaaa$

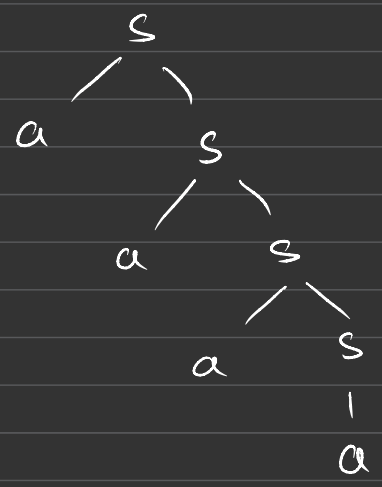
RMD

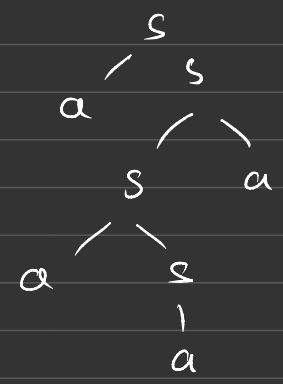
$S \rightarrow aS$

$S \rightarrow aSa$

$S \rightarrow aaSa$

$S \rightarrow aaaa$



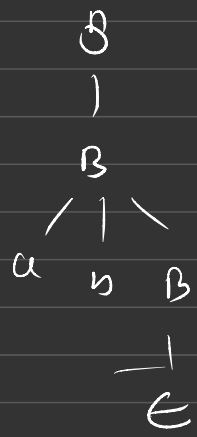


24/01/23

Q) $S \to A/B$
$A \to aAb/ab$
$B \to abB/\epsilon$

$S \to A$
$\to aAb$
$\to aaAbb$
String $\to$ aaabbb

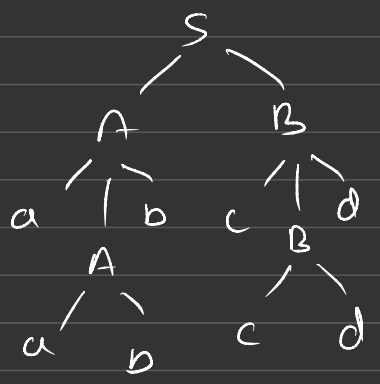
ab

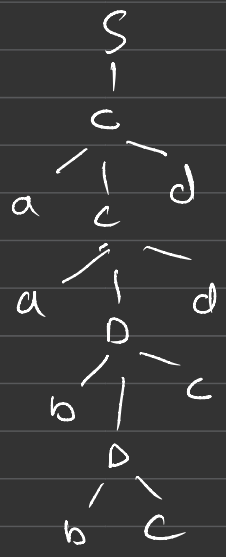


am

Q) $S \to AB/C$
$A \to aAb/ab$
$B \to cBd/cd$
$C \to aCd/aDd$
$D \to bDc/bc$

aabbccdd





ambiguous.

★ Imp

• Left recursive grammer

A grammer is said to be left recursive if all the productions are in the form

left recursive means.

$A \rightarrow A\alpha / \beta$ (calling itself at extreme left)

where

$\alpha, \beta = (V+T)^*$

Right recursive grammer.

A grammer is said to be right recursive if all the production are in the form

Right recursive means

$A \rightarrow \alpha A / \beta$

where, $\alpha, \beta = (V+T)^*$

Here the non terminal is calling itself from the LHS so it may lead to fall in ∞ loop. so most of the parser discourage the use of left recursive grammer.

• Elimination of left recursive grammer.

$A \rightarrow A\alpha / \beta$

This is substituted as

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

↙ variable

$S \rightarrow (L) / a$ → Terminal
$L \rightarrow L, S / S$
↓ α ↘ β

⇓

$L \rightarrow SL'$
$L' \rightarrow ,SL' / \epsilon$ } $S \rightarrow (L) / a$

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

$E \to TE'$
$E' \to +TE' / \in$

$T \to FT'$
$T' \to *FT' / \in$

Q) $S \to Aa / b$
$A \to Ac / Sd / \in$

$\Rightarrow S \to Aca / sda / a / b$
$S \to sda / Aca / a / b$

If $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 \cdots / \beta_1 / \beta_2 / \beta_3$

then
$A \to \beta_1 A' / \beta_2 A' / \beta_3 A' \cdots$
$A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \cdots \in$

Q) $S \to Aa / b$
$A \to Ac / Sd / \in$

$\Rightarrow S \to Aa / b$
$A \to \underline{Ac} / \underline{Aad} / \underline{bd} / \in$
$\alpha_1 \quad \alpha_2 \quad \beta_1 \quad \beta_2$

$A \to bdA' / A'$
$A' \to cA' / adA' / \in$

Q) $A \to Ba / Aa / c$
$B \to Bb / Ab / d$

$A \to A\underline{a} / \underline{Ba} / \underline{c}$
$\alpha_1 \quad \beta_1 \quad \beta_2$

$A \to BaA' / cA'$
$A' \to aA' / \in$

$B \to B\underline{b} / A\underline{b} / \underline{d}$
$\alpha \quad \beta \quad \beta$

$B \to B\underline{b} / Ba\underline{A'b} / \underline{cA'} / \underline{d}$
$\alpha_1 \quad \alpha_2 \quad \beta_1 \quad \beta_2$

$A \to BaA' / cA'$
$A' \to aA' / \in$
$B \to cA'bB' / dB' \quad B' \to bB' / aA'bB' / \in$

a) $A \to B / a / CBD$
$B \to c / b$
$C \to A / c$
$D \to d$

Q) $X \to XSb / Sa / b$
$S \to Sb / Xa / a$

$X \to \underset{\alpha_1}{XSb} / \underset{\beta_1}{Sa} / \underset{\beta_2}{b}$

$X \to Sax' / bx'$
$x' \to Sbx' / \epsilon$

$X \to Sax' / bx'$
$x' \to Sbx' / \epsilon$

$S \to bx'as' / as'$
$s' \to bs' / ax'as' / \epsilon$

Substitude production of X in S prodn.

$S \to \underset{\alpha_1}{Sb} / \underset{\alpha_2}{Sax'a} / \underset{\beta_1}{bx'a} / \underset{\beta_2}{a}$

$S \to A$
$A \to Ad / Ac / aB / ac$
$B \to bBc / b$

Q) $S \to A / a / BAC$
$A \to B / b$
$B \to S / C$
$C \to d$

$\Rightarrow$ $S \to B / b / a / BAC$
$A \to B / b$
$B \to B / b / a / BAC / C$
$C \to d$

$B \to B / BAC / b / a / C$

$\to$ $S \to B / b / a / BAC$
$A \to B / b$
$B \to bB' / aB' / cB'$
$B' \to ACB' / \epsilon$
$C \to d$

Left factoring

→ It is a method which is used to convert a non deterministic grammar into a deterministic grammer.

→ The grammer is said to be non deterministic if multiple productions are present for a single variable which contain same prefix and the right hand side of the production.

→ To avoid the backtracking, left factoring should be used.

$S \to iE+S \mid iE+SeS \mid a$
$E \to b$

$\Rightarrow$

$S \to iE+SS' \mid a$
$S' \to \epsilon \mid eS \mid$
$E \to b$

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

$\Downarrow$

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

$S \to iEtS\,S' \mid a$
$S' \to \epsilon \mid es$
$E \to b$

$S \to abc \mid abd \mid ae \mid b$

$S \to aS' \mid b$
$S' \to bc \mid bd \mid e$
$S' \to bS'' \mid e$
$S'' \to c \mid d$

ans

$S \to aS' \mid b$
$S' \to bS'' \mid e$
$S'' \to C \mid d$

Q) $A \to aAB \mid aBc \mid aAc$

$A \to aA'$
$A' \to AB \mid Bc \mid Ac$
$A' \to AA'' \mid Bc$
$A'' \to B \mid c$

Ans →

$A \to aA'$
$A' \to AA'' \mid Bc$
$A'' \to B \mid c$

Q) $S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \to bSS' \mid a$
$S' \to SaaS \mid SaSb \mid b$
$S' \to SaS'' \mid b$
$S'' \to aS \mid Sb$

Q) $S \rightarrow a \mid ab \mid abc \mid abcd$

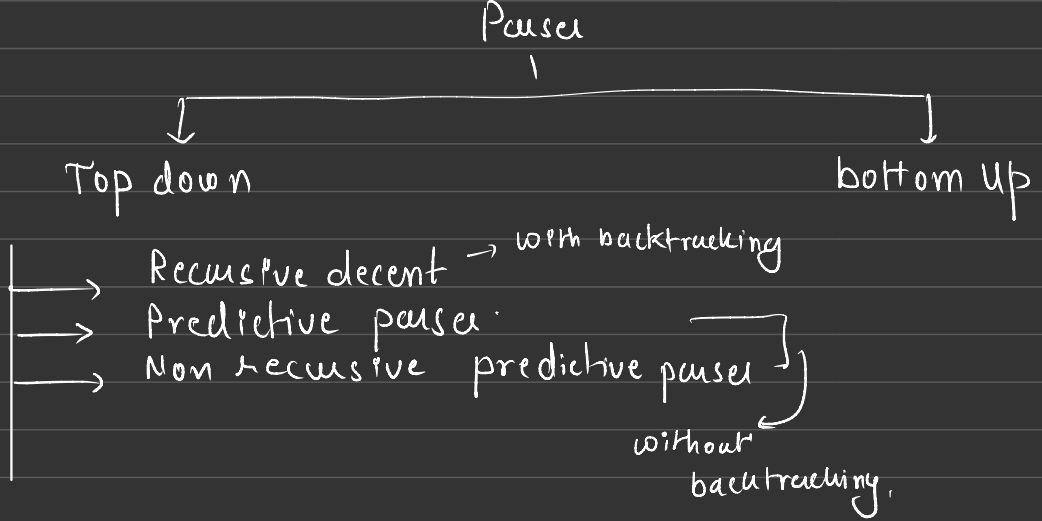
$S \rightarrow aS'$
$S' \rightarrow b \mid bc \mid bcd \mid \epsilon$
$S' \rightarrow bS'' \mid \epsilon$
$S'' \rightarrow c \mid cd \mid \epsilon$
$S'' \rightarrow cS''' \mid \epsilon$
$S''' \rightarrow \epsilon \mid d$

Q) $S \rightarrow aAd \mid aB$
$A \rightarrow a \mid ab$
$B \rightarrow ccd \mid ddc$

$S \rightarrow aS'$
$S' \rightarrow Ad \mid B$

$A \rightarrow a \mid ab$
$A \rightarrow aA'$
$A' \rightarrow \epsilon \mid b$
$B \rightarrow ccd \mid ddc$

•) Pausea



left to right scanning

•) LL(1) → At a time we are scanning one symbol.
→ LMD → Top down parser.

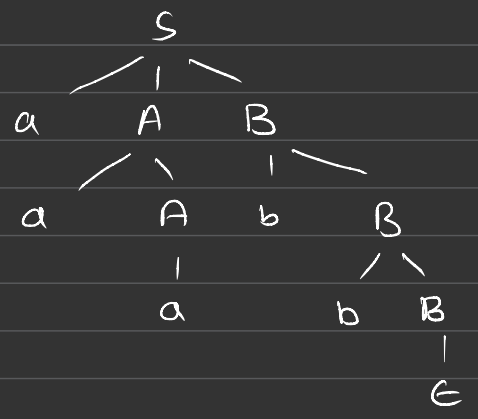
1) To construct a LL(1) grammer first eliminate the left recursion and non determinism. (Left factoring)

2) Then we have to construct a parse table.

As we are scanning the symbols from left to right

$S \rightarrow aAB \mid \epsilon$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid \epsilon$

string: aaabb



$S \rightarrow aAB$
$\rightarrow aaAB$
$aaaB$
$aaabB$
$aaabbB$
$aaabb$

Parser is used to verify the input string according to the grammer.

Then top down parser → in this parser the parse tree is generated from root to the children

So the derivation used in top down parser is left most derivation.

•) FIRST (x) ← variable

1) If x is a terminal then include the terminal in the FIRST(X)

First (x) = {x}

2) If x is variable and $X \to x_1, x_2, x_3, \ldots$

→ if $x_j \Rightarrow \epsilon \quad \forall j = 1 \ldots n$.

then FIRST(x) includes ϵ

→ if x_i derives some terminal then they are included in FIRST(x)

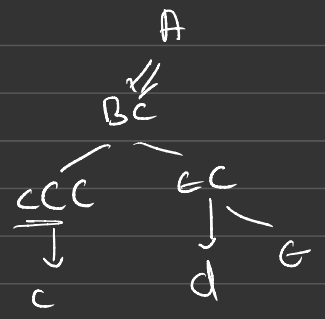
3) if $X \to \epsilon$

then FIRST (x) = ϵ

Example: Find the FIRST (A)

$A \to BC$
$B \to cC \mid \epsilon$
$C \to d \mid \epsilon$

FIRST(A) = {c, d, ϵ}



Example :-

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

FIRST(E) = {(, id)
(E') = {+, ϵ)
(T) = {(, id}
(T') = {*, ϵ}
(F) = {(, id}

(*FT') = {*}
(T'F) = {*, (, id}

•> FIRST is for both terminals and non terminal but FOLLOW is only for Non terminal & ϵ is not present in follow.

•> FOLLOW(x)

1> $ (end of the string) is included in follow(A) if A is Starting Symbol.

2> if $A \to \alpha B \beta$

FOLLOW(B) include the symbols present in First(β) except ϵ.

3> if $A \to \alpha B \beta$, where $\beta \overset{*}{\Rightarrow} \epsilon$ (→ continous series of derivation.)

or $A \to \alpha B$ then FOLLOW(B) includes all symbols present in FOLLOW(A).

$A \to a B c C A$

↓

First (cCA)

↓

c

Example $A \to aBCA \mid b$

$C \to a \mid \epsilon$

$A \to aBCA$

↓

First(CA)

aA ϵA → {a, b}

↓

a

Follow (E) = { $,) }

Follow (E') = { $,) }

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

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

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

Q1
S → ABCDE
A → a/ϵ
B → b/ϵ
C → c
D → d/ϵ
E → e/ϵ

Q2)
S → Bb / cd
B → aB / ϵ
C → cC / ϵ

Q3) S → ACB / CbB / Ba
A → da / BBC
B → g / ϵ
C → h / ϵ

Q4) S → aBDh
B → cC
C → bC / ϵ
D → EF
E → g / λ
F → f / λ

Q5)
S → aABb
A → C / ϵ
B → d / ϵ

Q1)
S → ABCDE
A → a/ϵ
B → b/ϵ
C → c
D → d/ϵ
E → e/ϵ

First	Follow
{a, b, c}	{$}
{a, ϵ} ✓	{b, c}
{b, ϵ} ✓	{c}
{c} ✓	{d, e, $}
{d, ϵ} ✓	{e, $}
{e, ϵ} ✓	{$}

S → ABCDE

F(aBCDE) F(ϵBCDE)

a

F(bCDE) F(ϵCDE)

b

F(cDE)

c

S → ABCDE

F(BCDE)

F(bCDE) F(ϵCDE)

b

F(CDE)

c

Follow of B

Q2)

$S \to Bb \mid cd$
$B \to aB \mid \epsilon$
$C \to cC \mid \epsilon$

First	Follow
{a, b, c, d}	{$}
{a, ε}	{b}
{c, ε}	{d}

Q3)

$S \to ACB \mid CbB \mid Ba$
$A \to da \mid BBC$
$B \to g \mid \epsilon$
$C \to h \mid \epsilon$

First	Follow
{d, g, h, ε, b, a}	{$}
{d, g, h, ε}	{h, g, $}
{g, ε}	{$, a, g, h}
{h, ε}	{g, $, b, h}

Q4) $S \to aBDh$
$B \to cC$
$C \to bC \mid \epsilon$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

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

→ In the parse table the number of rows is equal to no of non terminals + 1

& no of columns = no of terminals + 1.

Q)

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

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

If multiple entry in a single cell then the grammer is not LL(1)

FIRST(E) = {(, id)
(E') = {+, ε)
(T) = {(, id}
(T') = {*, ε}
(F) = {(, id}

Follow(E) = {$,)}
Follow(E') = {$,)}
Follow(T) = {+, $,)}
Follow(T') = {+, $,)}
Follow(F) = {*, +, $,)}.

		First	Follow
Q)	$S \to iE+SS' / a$	$\{i, a\}$	$\{\$, e\}$
	$S' \to eS / \epsilon$	$\{e, \epsilon\}$	$\{\$, e\}$
	$E \to b$	$\{b\}$	$\{b\}$

	i	+	a	e	b	$
S	$S \to iF+SS'$		$S \to a$			
S'				$S' \to eS$ $S' \to \epsilon$		$S' \to \epsilon$
E					$E \to b$	

Q1) $S \to aABb$
$A \to c / \epsilon$
$B \to d / \epsilon$

Q2) $S \to AB$
$A \to a / \epsilon$
$B \to b / \epsilon$

Q3) $S \to aSA / \epsilon$
$A \to c / \epsilon$

•) Bottom up Parser.

→ $E \to E+T / T$
$T \to T*F / F$
$F \to (E) / id$

id + id * id.

→ Right most derivation

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

Shift reduce parser.

→ The parser is generated from the leaf to root.

→ The SRP is one of the bottom up parsing approach which uses the LR grammer.
↓
left to right scanning & R is for reverse for rightmost parsing

→ The bottomup parsing is nothing but multiple reduction takes place that reduces the substring to the head of the production when the substring matches with the body of the production.

→ If the string w is given & we are able to find out the starting symbol by successive reductions then we can say that the string is accepted.

1) Check whether the given grammer is LL(1) or not

a) $S \to (L) \mid a$
$L \to SL'$
$L' \to ,SL' \mid \epsilon$

b) $S \to i(tSS_1 \mid a$
$S_1 \to eS \mid \epsilon$
$C \to b$

c) $S \to aSa \mid bS \mid c$

d) $S \to aSbS \mid bSaS \mid \epsilon$

e) $S \to aB \mid \epsilon$
$B \to bC \mid \epsilon$
$C \to cS \mid \epsilon$

g) $S \to A \mid a$
$A \to a$

b)

	First	Follow
S	{i, a}	{e, $}
S_1	{e, ϵ}	{e, $}
C	{b}	{t}

	a	i	t	b	e	$
S	S→a	S→iCtSS$_1$				
S_1					$S_1 \to \epsilon$ $S_1 \to eS$	$S_1 \to \epsilon$
C				C→b		

a)

	First	Follow
S	{(, a}	{$,), ,}
L	{(, a}	{)}
L'	{,, ϵ}	{)}

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

c)

First	Follow
{a, b, c}	{$, a}

d) {a, b, ϵ} | {$, a, b}

e)

{a, ϵ}	{$}
{b, ϵ}	{$}
{c, ϵ}	{$}

g)

{a}	
{a}	

•) Fast Method

⇒ First find all the first of the grammer which has no null production

$$S \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4 \quad \text{(like this)}$$

If intersection of all the first production is ϕ then the grammer is LL1

Example

$S \to aAb$ $\quad F(S) = a$
$A \to bC$ $\quad F(A) = b$ $\quad = \phi$
$C \to c$ $\quad F(C) = c$

$S \to aA = a$
$A \to abB = a \quad \therefore \quad a = \phi = a \quad \therefore$ Not LL1
$B \to b = b$

Q)

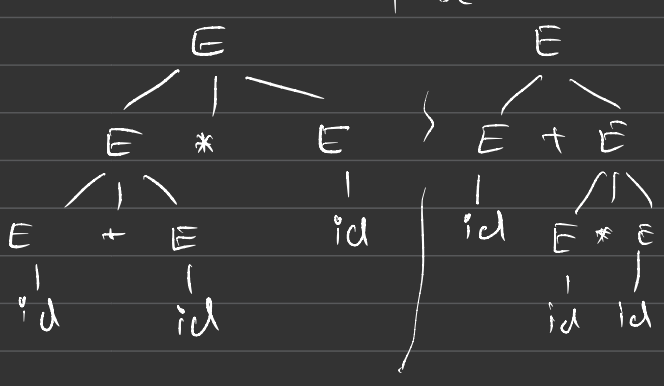
	First	Follow
$S \to$	{ (, a }	{ $,), , }
$L \to$	{ (, a }	{) }
$L' \to$	{ , , ∈ }	{) }

$S \to (\cap a = \phi$
$L \to (\cap a = \phi$
$L' \to , \cap) = \phi$

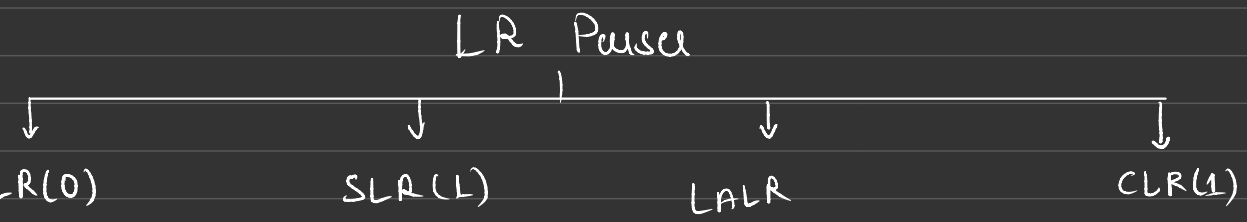
Q) $E \to E+E \mid E*E \mid id$

$id + id * id$

multiple parse

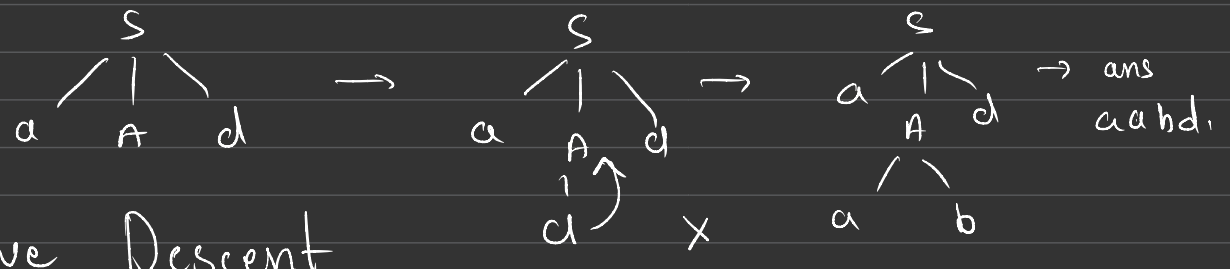


Shift Reduce Parser



Q) S → aAd / aB
A → a / ab
B → ccd / ddc

I/P → aabd (continous checking and if error occurs then it will backtrack to the Non terminal where we are facing issue.



Recursive Descent

Q) E → iE'
E' → +eE / ∈

Inp → i i+i

```
E()
{ // l is a lookahead
  if (l == 'i') {
      match ('i')
      E'()
  }
```

```
E'()
{
  if (l == '+') {
      match ('+')
      match ('e')
      E()
  }
  else {
      return
  }
```

inp → iii+i $ → By default there is a $ symbol

```
match (char t) {
        if (l==t) {
            l = getchar();
        }
        else{
            printf("Error")
        }
    }
main () {
    E();
    if (l == '$') {
        print ( Parsing successful )
    }
}
```

for each variable we have write a function.

```
main() {
    E();
    if (l=='$') {
        print
```

→ end of s

aaab$

Q) E → i E' / i a E'
E' → +eE' / ∈

```
E() {
    if (l == 'i') {
        match ('i');
        if (l == 'a') {
            match('a');
            S()
        }
        else {
            E'();
        }
    }
```

• Shift Reduce parser. (Bottom up parser Type)

→ The parsing is done from leaf to root.

• Stack : Action are performed here

Input Buffer: Storing the input string.

→ Rules:

→ There are two actions we need to perform in this parser

1) Shift() → Parser Shifts zero or more input symbols onto the stack until the handle is on top of the stack.

Handle is the RHS of the production ($E \rightarrow \underbrace{iE}_{\text{handle}}!$)

2) Reduce() → Parser reduce or replace the handle on top of the stack to the left side of the production ie RHS of production is poped & LHS is pushed.

3) Accept() → Shift & reduce will be repeated until it has detected an error or until the stack includes start symbol s and input buffer is empty. ie it contains $.

$S \rightarrow CC$
$C \rightarrow cC$
$C \rightarrow d$

Stack	Input buffer	Action.
$	ccdd$	shift
$c	cdd$	Shift
$cc	dd$	Shift
$ccd	d$	Reduce (by C→d)
$ccC	d$	Reduce (by C→cC)
$cC	d$	Reduce (by C→cC)
$C	d$	Shift
$Cd	$	reduce (by C→d)
$CC	$	reduce (by S→CC)
$S	$	Accepted

example:-

$S \rightarrow S+S$
$S \rightarrow S*S$
$S \rightarrow id$

ip → id + id * id.

Stack	Input Buffer	Action
$	id + id * id $	shift
$ id	+ id * id $	reduce S → id
$ S	+ id * id $	shift
$ S +	id * id $	shift
$ S + id	* id $	reduce S → id
$ S + S	* id $	reduce S → S+S
$ S	* id $	shift
$ S *	id $	shift
$ S * id	$	reduce S → id
$ S * S	$	reduce S → S*S
$ S	$	Accept.

Q) $A \rightarrow 5A5$
$A \rightarrow 7A7$
$A \rightarrow 9$

ip → 75957

Stack	Input Buffer	Action
$	75957 $	Shift
$ 7	5957 $	Shift
$ 75	957 $	Shift
$ 759	57 $	reduce A → 9
$ 75A	57$	shift
$ 75A5	7 $	reduce A → 5A5
$ 7A	7 $	shift
$ 7A7	$	reduce A → 7A7
$ A	$	Accepted.

Q) $T \rightarrow T+T \mid T-T \mid id$

ip id + id - id

Stack	input Buffer	Action
$	id + id - id $	shift
$id	+ id - id $	reduce $T \rightarrow id$
$T	+ id - id $	shift
$T +	id - id $	shift
$T + id	- id $	reduce $T \rightarrow id$
$T + T	- id $	reduce $T \rightarrow T+T$
$T	- id $	shift
$T -	id $	shift
$T - id	$	reduce $T \rightarrow id$
$T - T	$	reduce $T \rightarrow T-T$
$T	$	Accepted

Q) $E \rightarrow E+E \mid E*E \mid (E) \mid c$

i/p $\rightarrow$ c * (c + c)

Stack	input buffer	Action
$	c * (c + c) $	shift
$c	* (c + c) $	reduce $E \rightarrow c$
$E	* (c + c) $	shift
$E *	(c + c) $	shift
$E * (	c + c) $	shift
$E * (c	+ c) $	reduce $E \rightarrow c$
$E * (E	+ c) $	shift
$E * (E +	c) $	shift
$E * (E + c	) $	reduce $E \rightarrow c$
$E * (E + E	) $	reduce $E \rightarrow E+E$
$E * (E	) $	shift
$E * (E)	$	reduce $E \rightarrow (E)$
$E * E	$	reduce $E \rightarrow E*E$
$E	$	accepted

• LR parser (Bottom up)

Bottom up parser is more powerful than Top down parser.

⇒ It's of 4 type.

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

Left to right scan.

$$LR(0) < SLR(1) < LALR(1) < CLR(1)$$

1) LR(0) parser.

⇒ L stands for Left to right scanning, R stands for rightmost derivation in reverse & 0 stand for zero lookahead.

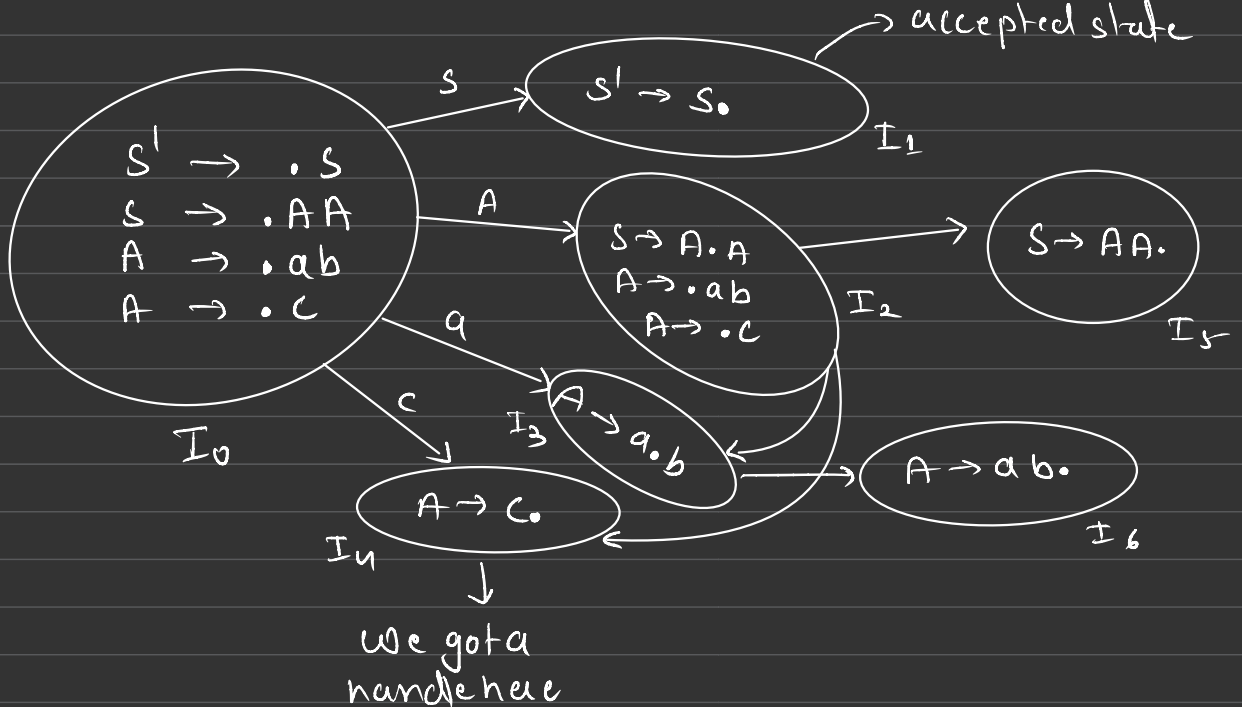
⇒ In order to construct the LR(0) parser we use cannonical collection of LR(0) items.

$S \rightarrow AA$
$A \rightarrow ab/c$

$\Rightarrow$

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow ab/c$

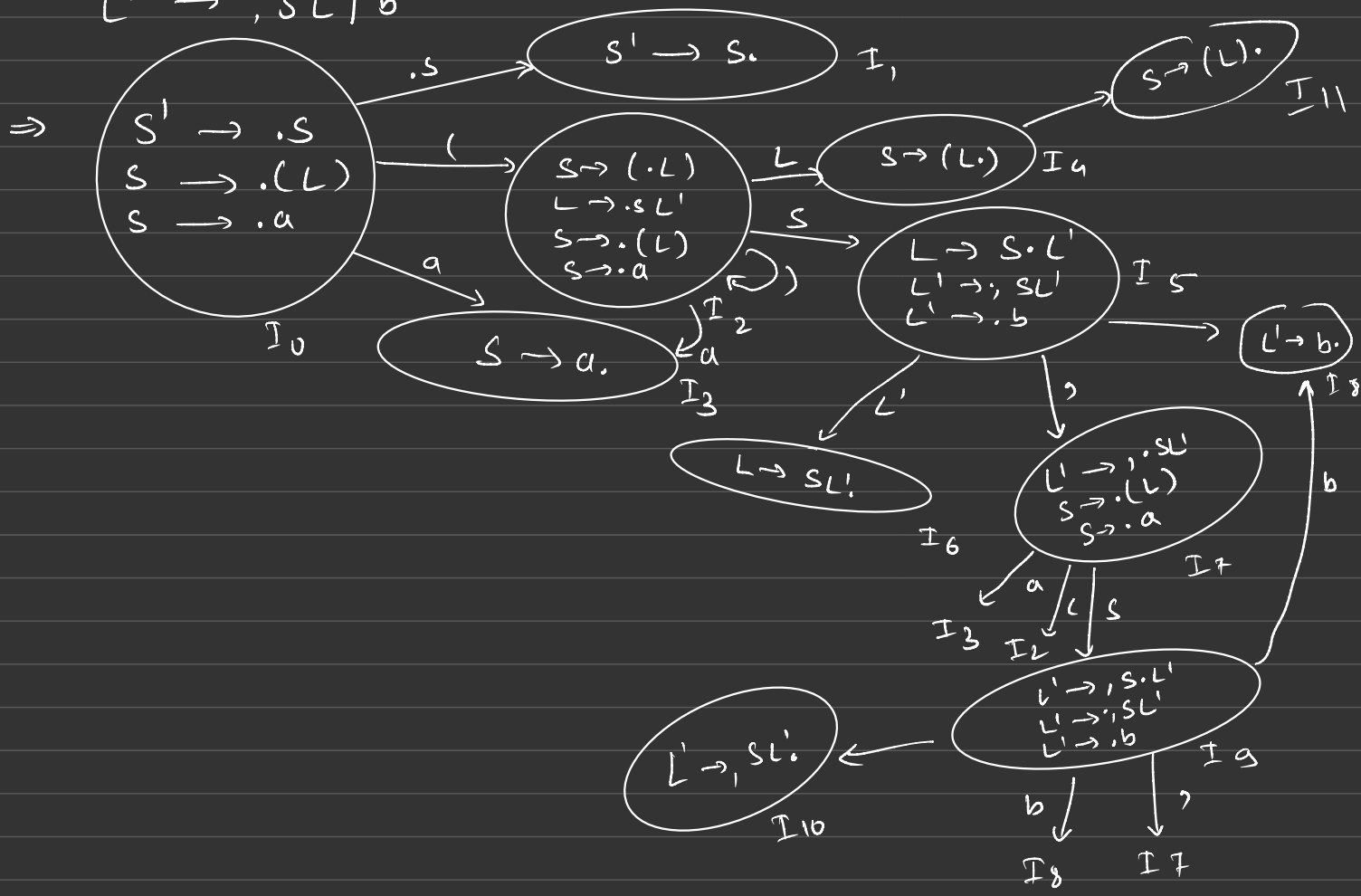
Example



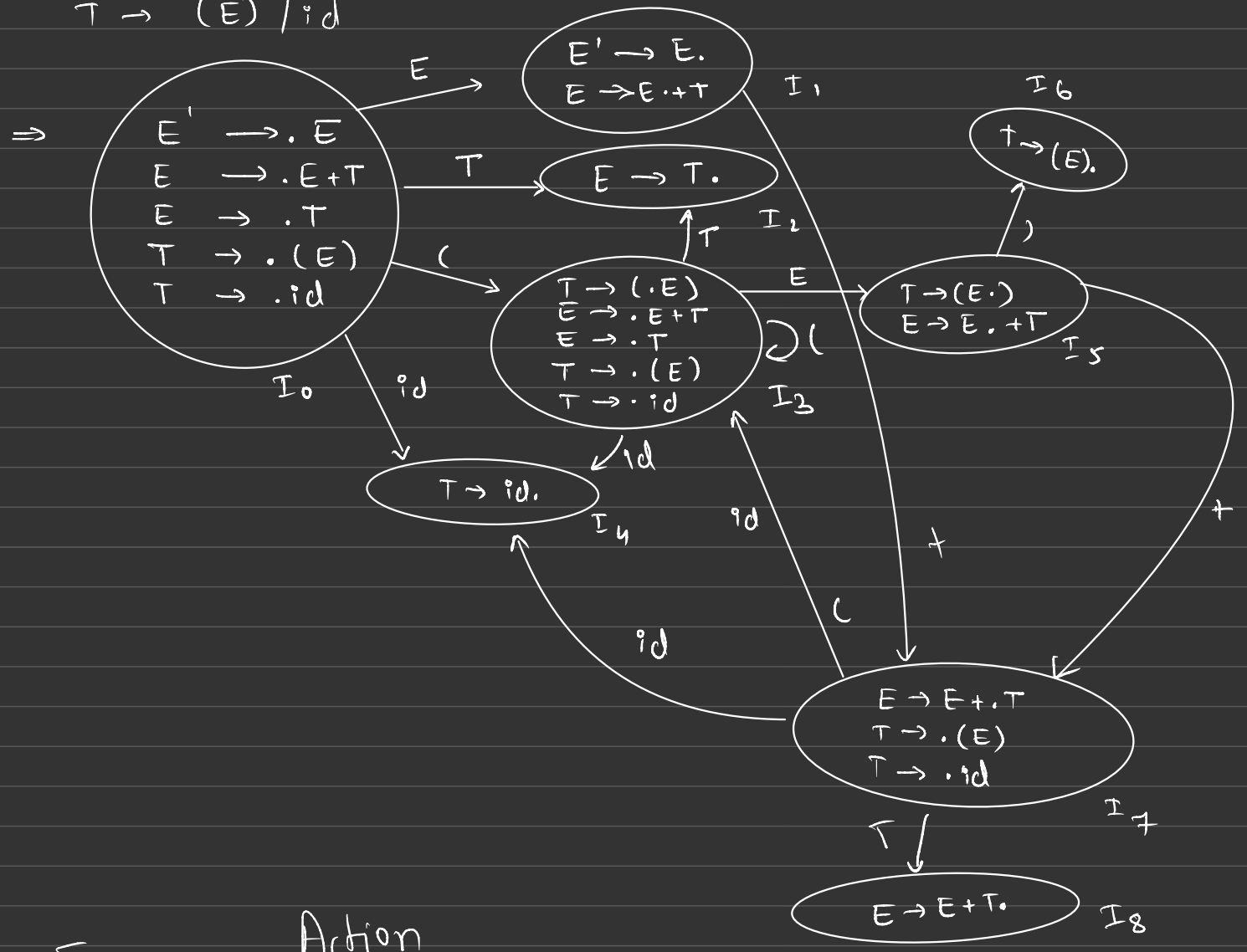
Q) $S \to (L) \mid a$

$L \to SL'$

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



(1) $E \to E+T \mid T$
$T \to (E) \mid id$



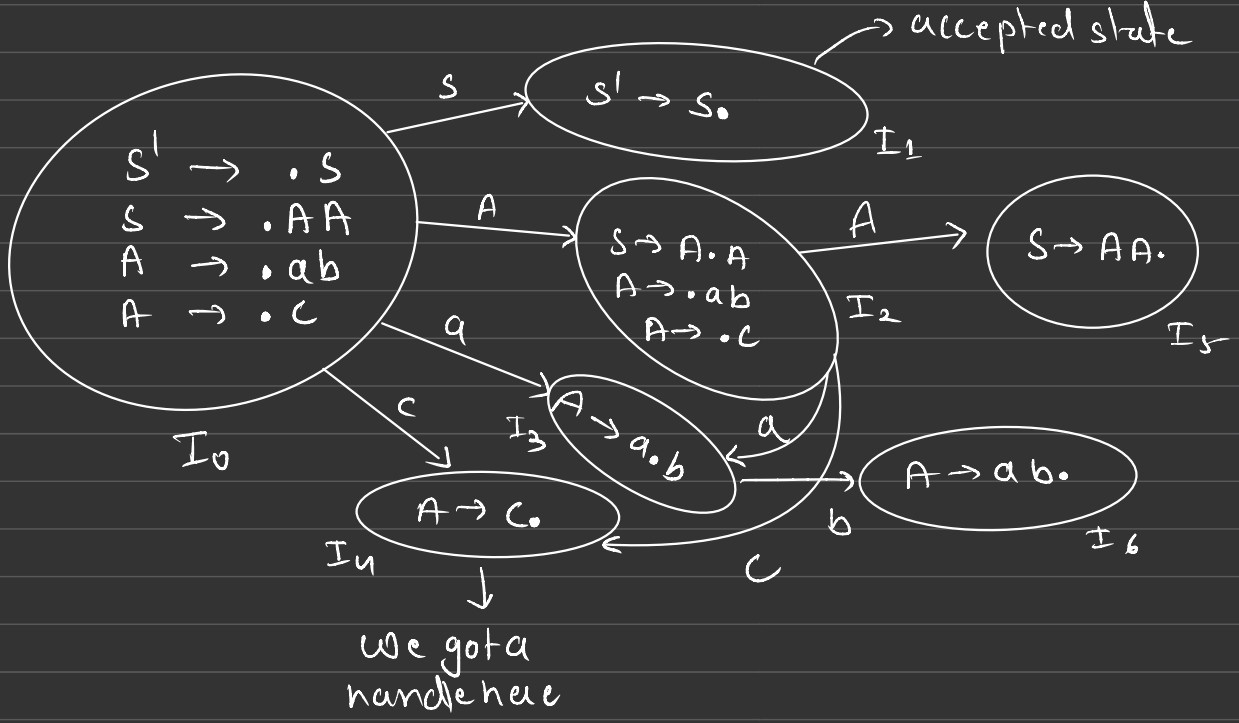
I	Action					Go to	
	+	(	)	id	$	E	T
0		S_3		S_4		1	2
1	S_7				Ac		
2	R_2	R_2	R_2	R_2	R_2		
3		S_3		S_4		6	2
4	R_4	R_4	R_4	R_4	R_4		
5	S_7		S_6				
6	R_3	R_3	R_3	R_3	R_3		
7		S_3		S_4			8
8	R_1	R_1	R_1	R_1	R_1		

$S \to AA$
$A \to ab/c$

$\Rightarrow$

$S' \to S$
$S \to AA$
$A \to ab/c$

Example



I	Action a	b	c	$	Go to S	A
0	Shift 3		Shift 4		1	2
1				Accept		
2	Shift 3		shift 4			5
3		S_6				
4	R_3	R_3	R_3	R_3		
5	R_1	R_1	R_1	R_1		
6	R_2	R_2	R_2	R_2		

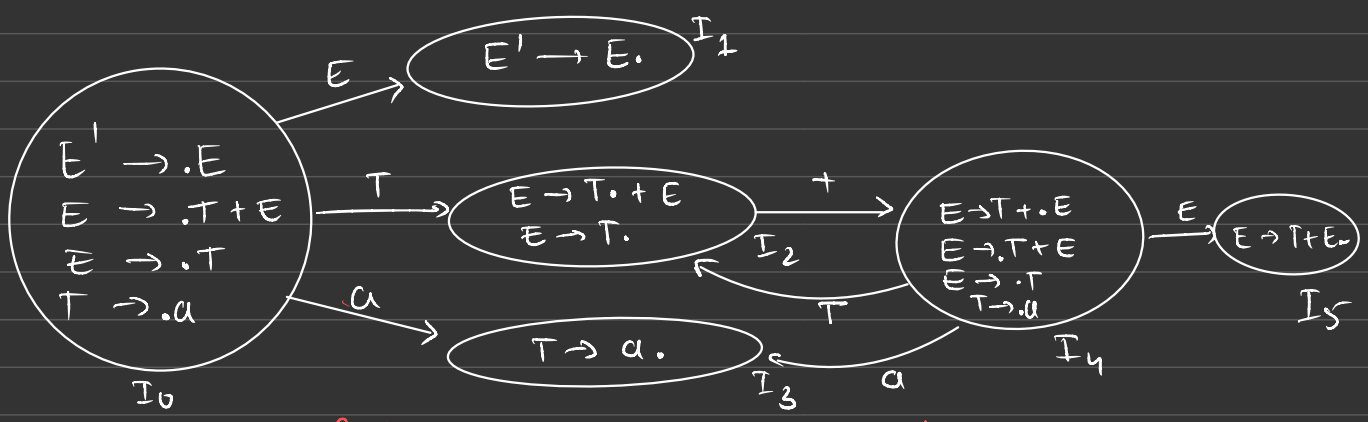
R → Reduce
S → shift

R_3 is not for state 3 but for production 3

$S \to AA$ (1)
$A \to ab$ (2)
$A \to c$ (3)

Q) $E \to T+E \mid T$
$T \to a$

find if grammer is LR(0) or Not



	Action			Goto	
	+	a	$	E	T
0		S_3		1	2
1			Accept		
2	S_4/R_2	R_2	R_2		
3	R_3	R_3	R_3		
4		S_3		5	2
5	R_1	R_1	R_1		

Not LR(0) grammer.

SR conflict hence

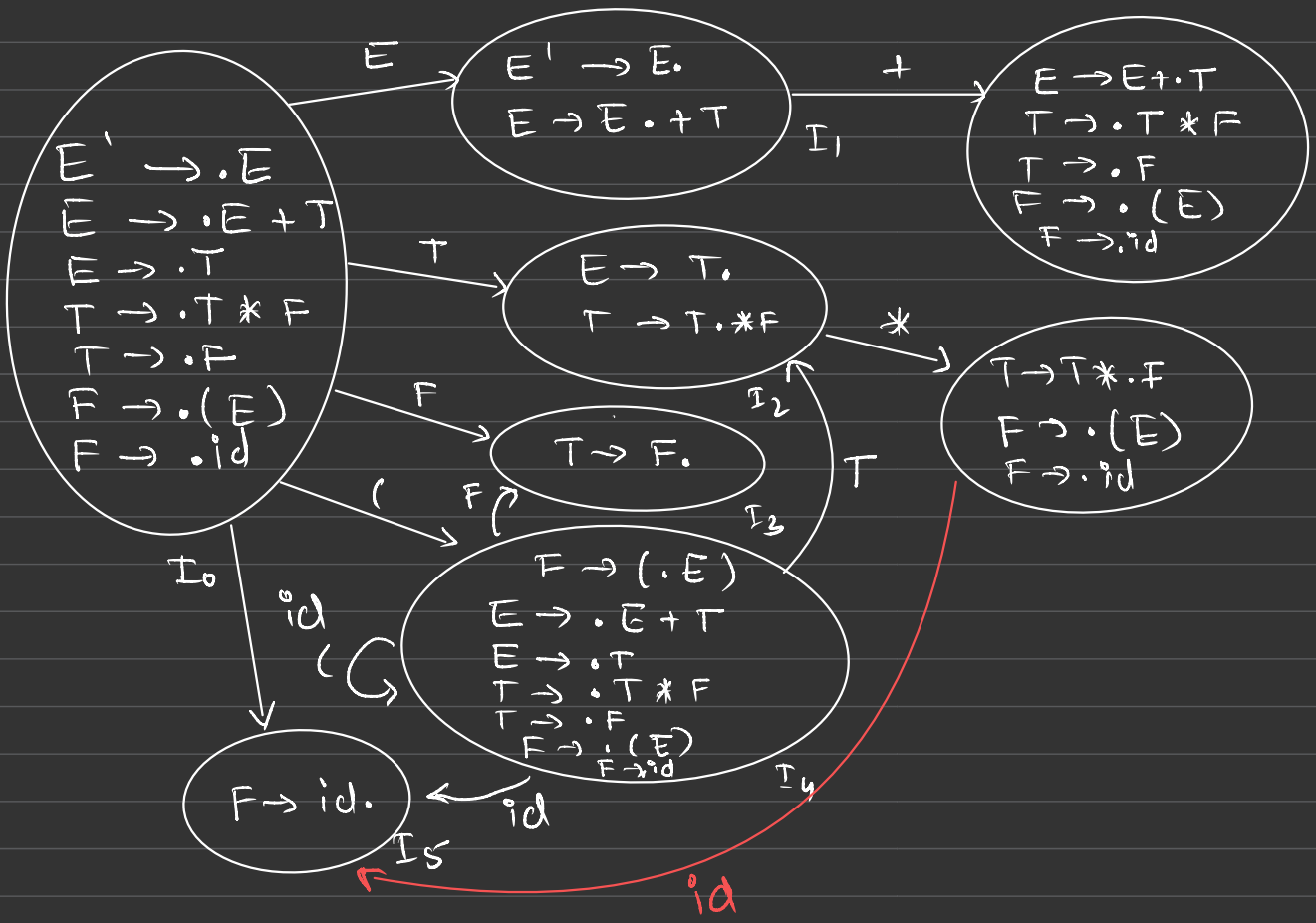
• SLR(1) Parser. (Diagram same hoga as LR(0) parser)

→ Find the follow of the left hand side of the handle.

	Action			Goto	
	+	a	$	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		

Check if LR(0) or not. If not check for SLR(1)

→ $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$



Q1) $S \to xAy \mid xBy \mid xAz$
$A \to qS \mid q$
$B \to q$

Q2) $S \to 0S0 \mid 1S1 \mid 10$.

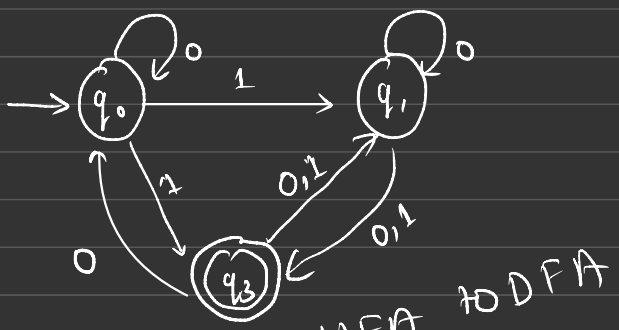
Q3) $S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$.

•) Revision:

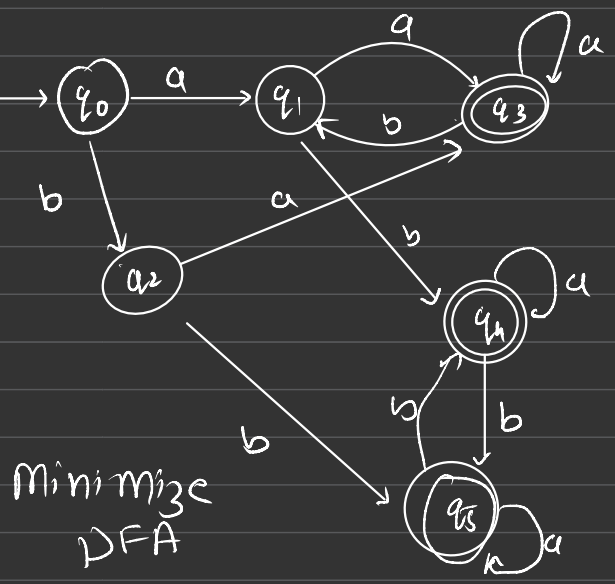
Q1) $S \to aAc / aAb / aBc$
$A \to d$
$B \to e$

Q2) $E \to E+E / E*E / a$

Q3)



Q4)



Sol

$S \to aA'$
$A' \to Ac / Ab / Bc$
$A' \to AA'' / Bc$
$A'' \to c / b$
$A \to d$
$B \to e$

Q5) $S \to (L)$
$L \to L, S / \epsilon$

Sol ⇒ (2) $\underline{E} \to \underline{E}(+E) / \underline{E}*E / a$

$\Rightarrow E \to aE'$
$E' \to +EE' / *EE' / \epsilon$

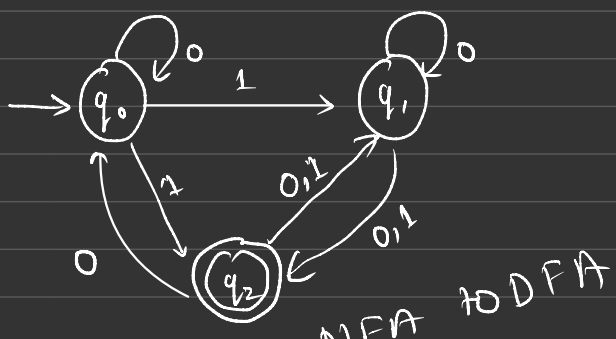
$E \to aE'$
$E' \to +EE' / *EE' / \epsilon$

$S \to S\alpha_1 / S\alpha_2 / \dots / \beta$

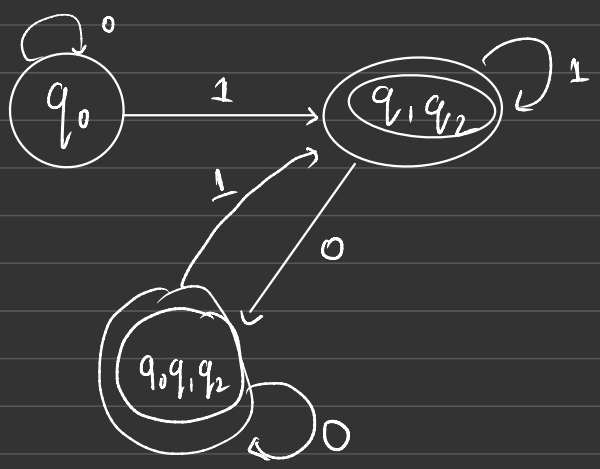
$S \to \beta S'$
$S' \to \alpha E' / \epsilon$

Q3) Sol → Transition table

States	0	1
→ q_0	q_0	$\{q_1 q_2\}$
* $q_1 q_2$	$q_0 q_1 q_2$	$q_1 q_2$
* $q_0 q_1 q_2$	$q_0 q_1 q_2$	$q_1 q_2$



NFA to DFA



Q4)

Transition table

State	0	1
q_0	q_1	q_2
q_1	q_3	q_4]
q_2	q_5	q_5]
q_3	q_3	q_1
q_4	q_4	q_5
q_5	q_5	q_4

Same grp hence

$\pi_0 = \{q_0\ q_1\ q_2\}\ \{q_3\ q_4\ q_5\}$

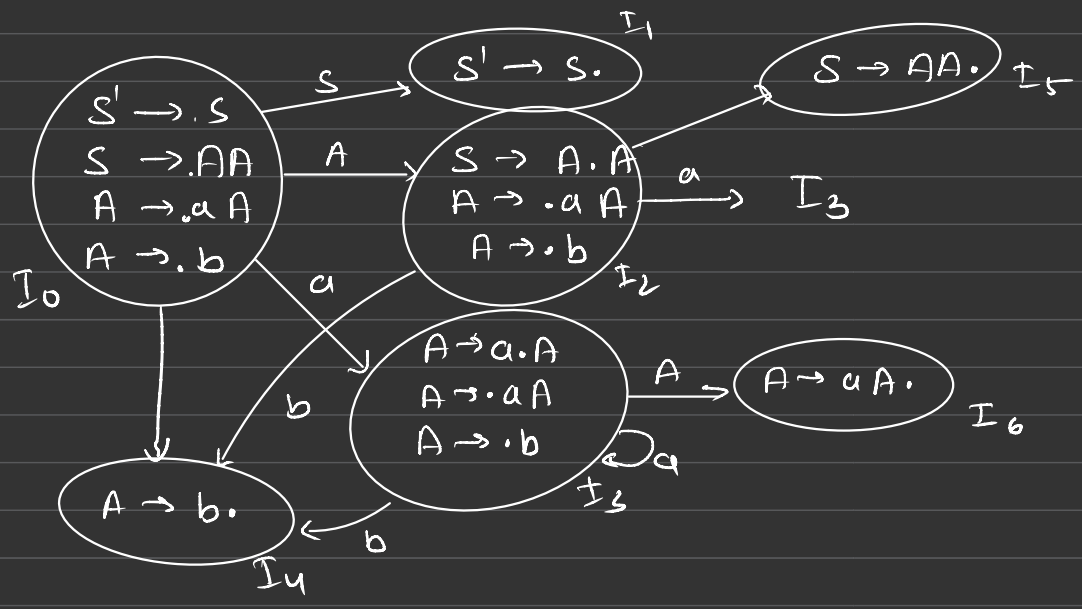
q_1 ⊥ Non final $\quad$ q_2 ↓ final state

$\pi_1 = \{q_0\}_{q_{21}}\ \{q_1, q_2\}_{q_{12}}$ → Same grp $\{q_3\}_{q_{23}}\ \{q_4\ q_5\}_{q_{24}}$ Check $\{\text{Same grp}\}$

$\pi_2 = \{q_0\}\ \{$

Q) $S \to AA$, $A \to aA$, $A \to b$

$S' \to S$
r_1 $S \to AA$
r_2 $A \to aA$
r_3 $A \to b$

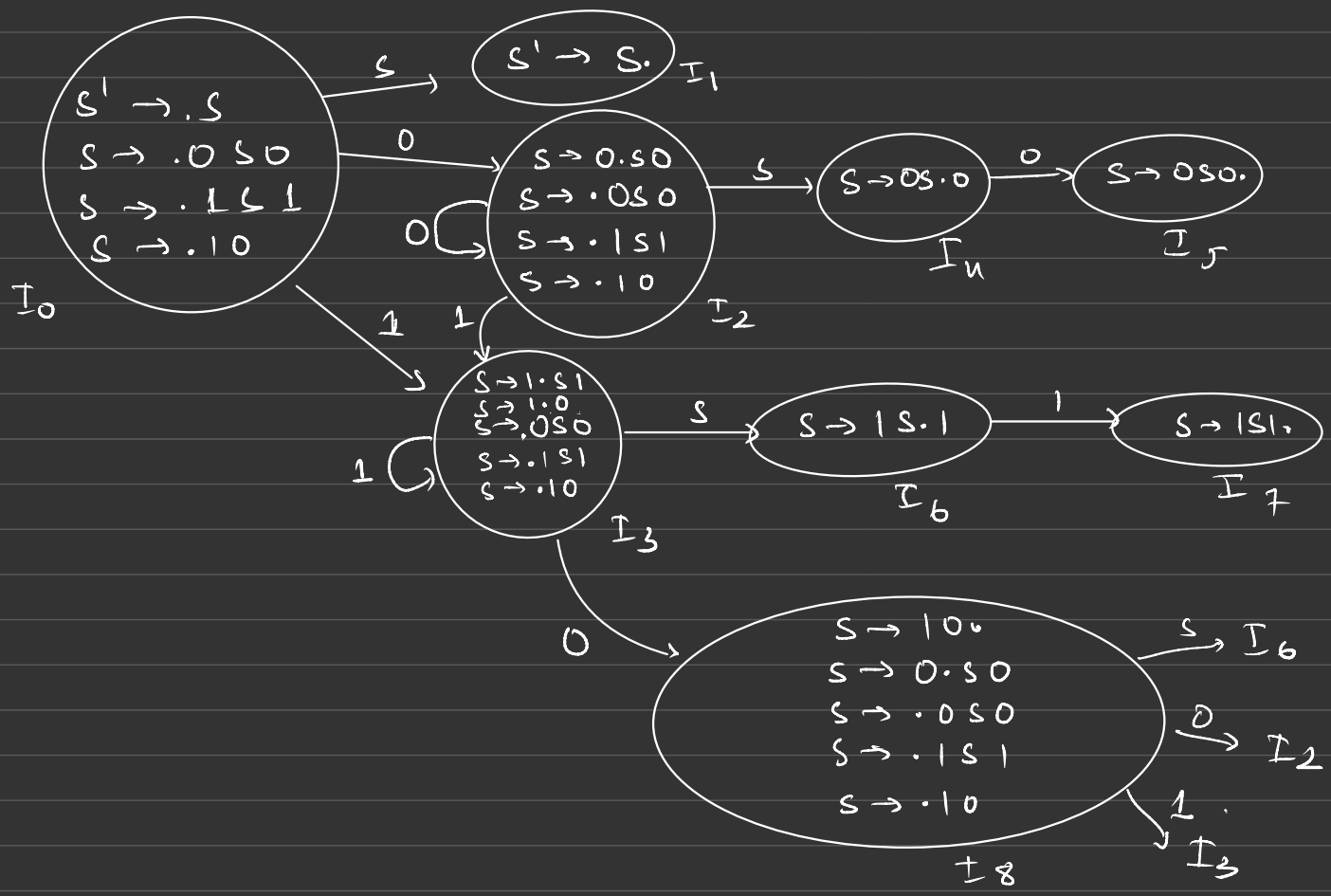


	action			goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1	Accept	Accept	Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

Q) Check

$S \rightarrow 0S0 \mid 1S1 \mid 10$

Is not SLR 1



Q) $S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

⇒ What is LR(1) parser — LR(0) has no look ahead symbol.

•> LR(1) Parser.

⇒ LR(0) + look-ahead

$A \to .XYZ$, $ ← look ahead symbol

Closure of (i) → any symbol (symbol

⇒ All items present in (i) are included in the closure of (i)

If a production in this form.

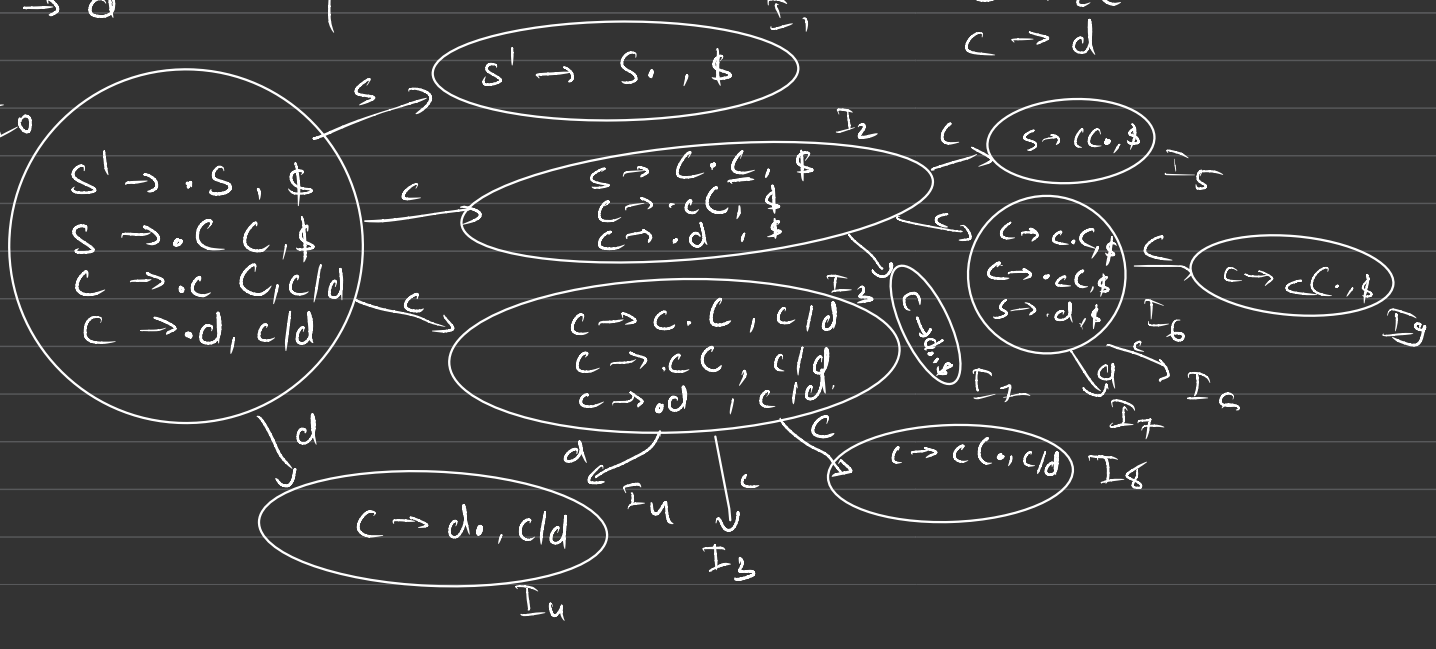
$A \to \alpha.B\beta$, $ ← look ahead symbol is in closure of (i) include all the production of B.

$B \to .\gamma$, (First of (β, $) rest part after B)

Q)

$S \to CC$
$C \to cC$
$C \to d$

augmented grammer ⇒ $S' \to S$
$S \to CC$
$C \to cC$
$C \to d$



	Action			Go to	
	c	d	$	S	C
0	S_3	S_4		1	2
1	------		Accepted		
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		

•) LALR

→ In LALR(1) the two states present in CLR(1) are combined if they are varying only in their lookahead symbol.

∴ from previous CLR

We have to merge states as they are similar with different LA

$I_3 + I_6 = I_{36}$

$C \to c.C$, $\$ / c / d$
$C \to .cC$, $\$ / c / d$
$C \to .d$, $\$ / c / d$

$I_4 + I_7 = I_{47}$

$C \to d.$, $c/d/\$$

$I_8 + I_9 = I_{89}$

$C \to cC.$, $\$ / c / d$

	Action			Go to	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1	------		Accepted		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			8
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

LALR

Q) $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$.

$I_0 \to$

$E' \to .E, \$$
$E \to .E+T, \$$
$E \to .T, \$$
$E \to .E+T, +$
$E \to .T, +$
$T \to .T*F, \$$
$T \to .F, \$$
$T \to .T*F, *$
$T \to .F, *$
$T \to .T*F, +$
$T \to .F, +$
$F \to .(E), \$$
$F \to .id, \$$
$F \to .(E), +$
$F \to .id, +$
$F \to .(E), *$
$F \to .id, *$

I_0

Q) $S \to AaAb \;(1) \;/\; BbBa \;(2)$ (LR(0), SLR(1), CLR(1))

(3) $A \to \epsilon$

(4) $B \to \epsilon$

Sol→

I_0:
- $S' \to S$
- $S \to AaAb$
- $S \to BbBa$
- $A \to \bullet$
- $B \to \bullet$

As A & B are in a reduced state we will have a RR conflict in I_0 itself therefore not LR(0)

for SLR(1) we will find the follow of the left hand element for production
∴ for $A \to \bullet$ & $B \to \bullet$ we will have
Follow(A) = a, b & Follow(B) = a, b
∴ We will again have RR conflict so not SLR(1).

For CLR(1) we will first find the lookahead for the above grammer.

I_0:
- $S' \to .S, \$$
- $S \to .AaAb, \$$
- $S \to .BbBa, \$$
- $A \to ., a$
- $B \to ., b$

No conflict here

On S → I_1: $S' \to S., \$$

On A → I_2: $S \to A.aAb, \$$

On B → I_3: $S \to B.bBa, \$$

I_2 on a → I_4: $S \to Aa.Ab, \$$; $A \to ., b$

I_4 on A → I_5: $S \to AaA.b, \$$

I_5 on b → I_6: $S \to AaAb., \$$

I_3 on b → I_7: $S \to Bb.Ba, \$$; $B \to ., a$

I_7 on B → I_8: $S \to BbB.a, \$$

I_8 on a → I_9: $S \to BbBa., \$$

	Action			Go to		
	a	b	$	S	A	B
I_0 →	R_3	R_4		1	2	3
I_1 →			Accepted			
I_2 →	S_4					
I_3 →		S_7				
I_4 →		R_3			5	
I_5 →		S_6				
I_6 →			R_1			
I_7 →	R_4				8	
I_8 →		S_9				
I_9 →			R_2			

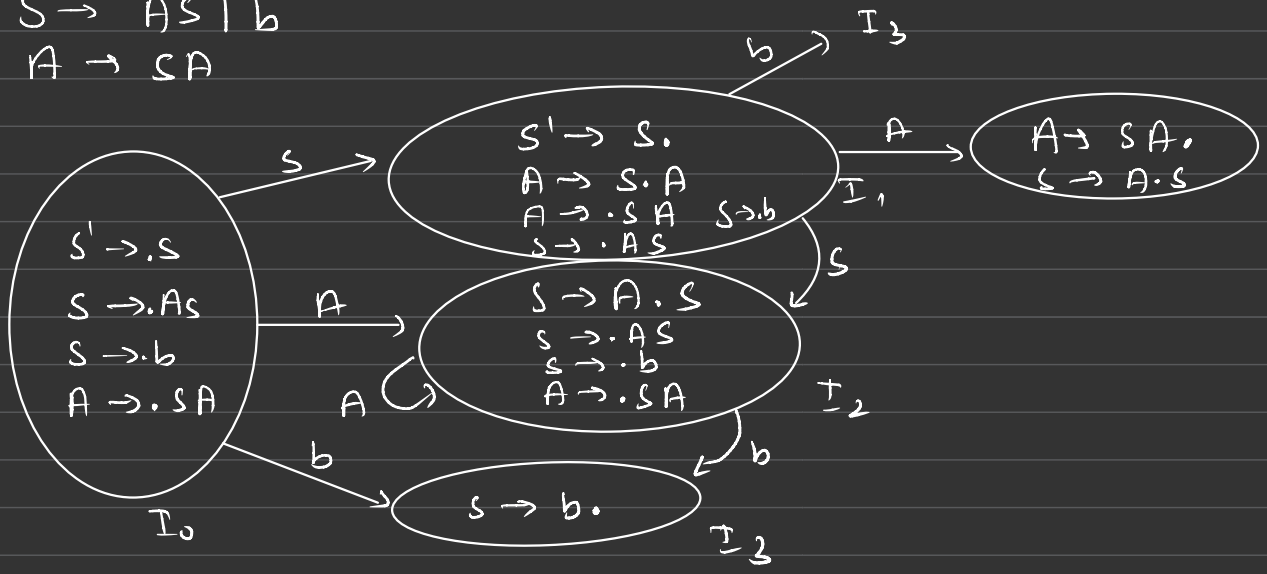
Q) $S \to As \mid b$
$A \to SA$

Q) $S \to xAy \mid xBy \mid xAz$
$A \to qS \mid q$
$B \to q$

Q) $S \to Au \mid bAc \mid dc \mid bda$
$A \to d$

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

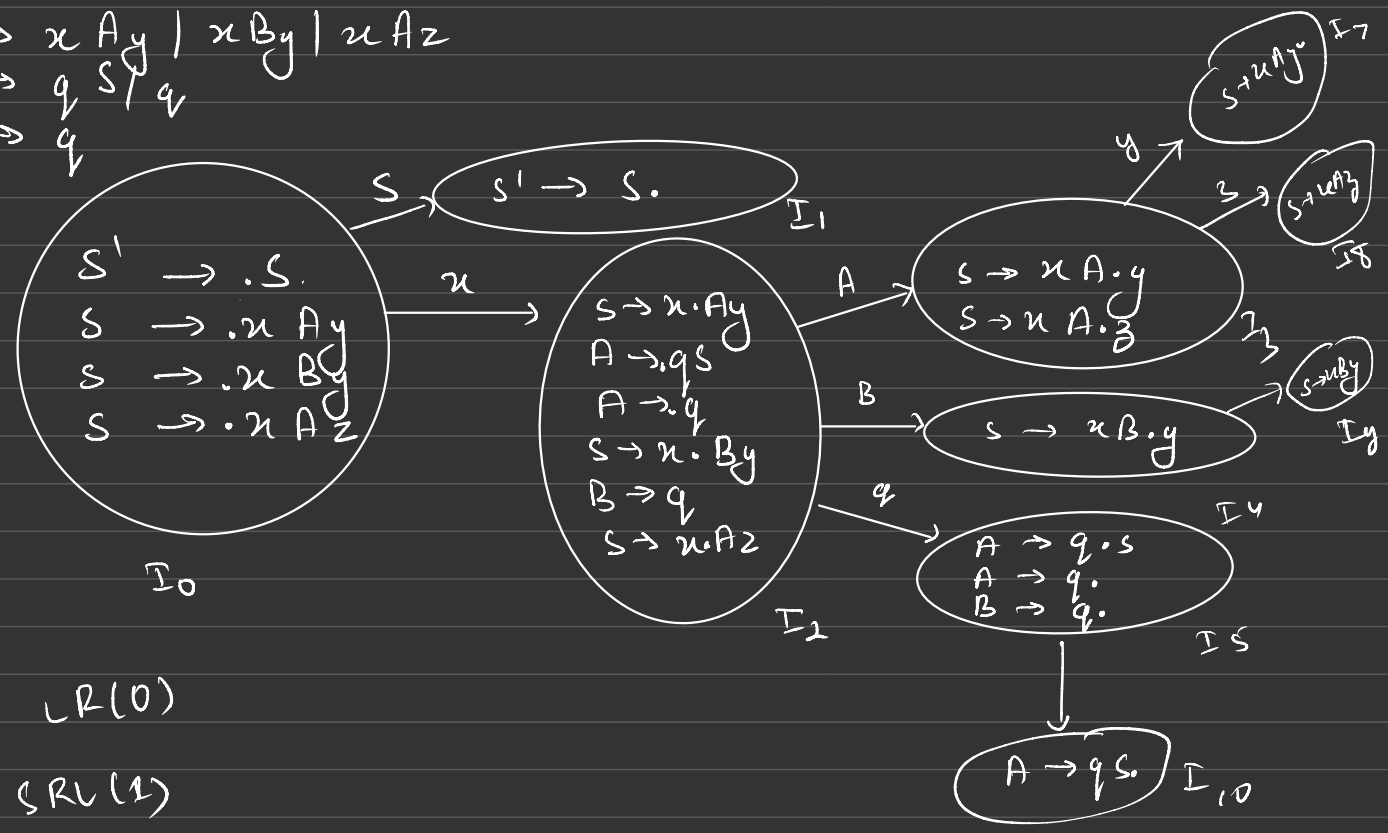
Q1) $S \to AS \mid b$
$A \to SA$



LL(1)

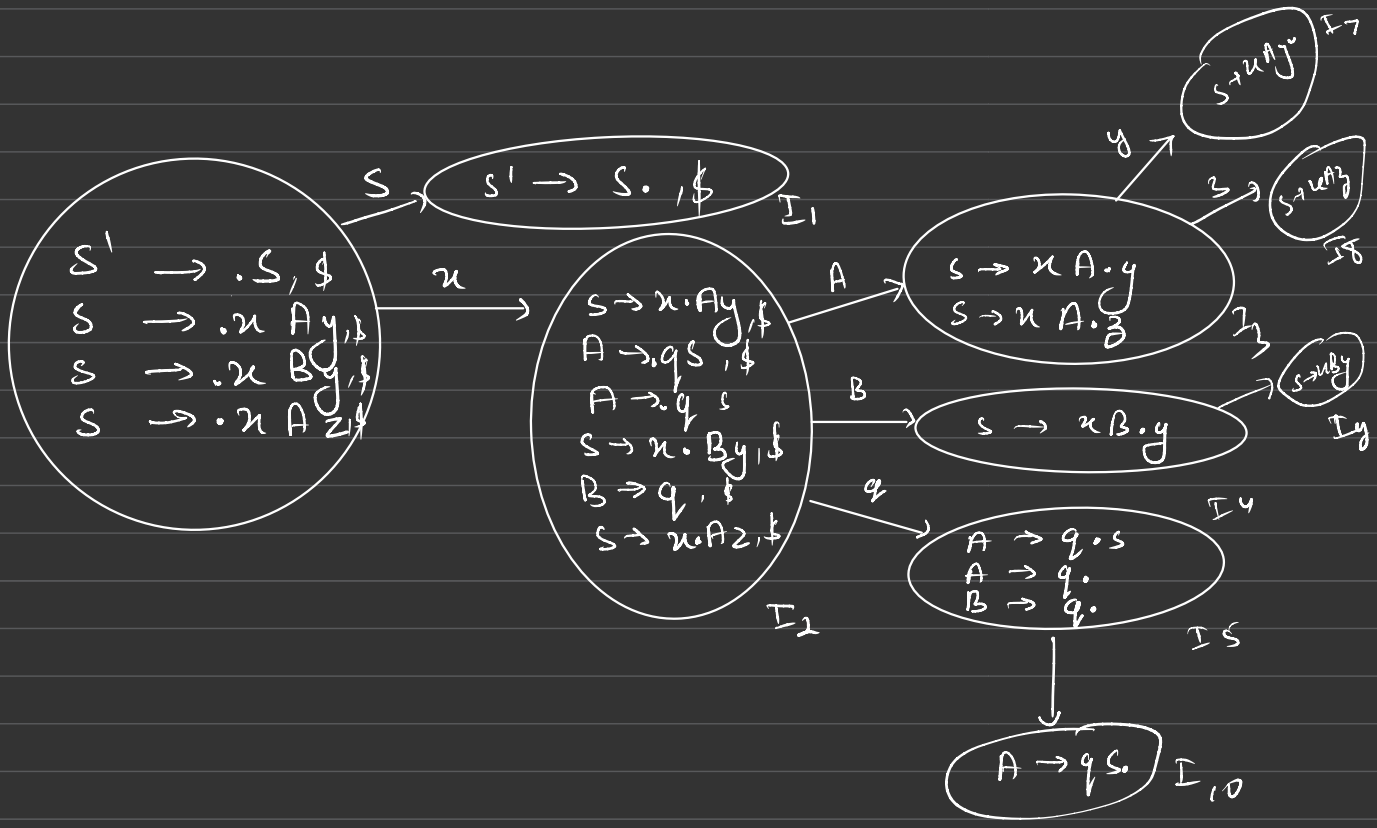
Q) $S \to xAy \mid xBy \mid xAz$
$A \to qS \mid q$
$B \to q$

Ans →



LR(0)

SLR(1)



• Chapter - 3 (Semantic Analysis)

→ The procedure used is syntax directed translation.

→ In syntax directed translation we are adding semantic rules and semantic actions to each productions of the grammer.

→ It is of two types

1) Sytax Directed Definition. (SDD) } semantic rules.
2) Sytax Directed Translation. (SDT) } semantic action.

Difference Between SDD & SDT

SDD

1) In SDD attributes and semantic rules are associated with the grammer.

ex $E \rightarrow E+T$

Rule $E.code = E.code + T.code$

→ $E.code = E.code / T.code / '+'$

Infix – postfix semantic rule

Here code is the attribute which is associated with the grammer symbol.

2) SDD is more Readable than SDT.

3) It is used in case of specification.

SDT

1) The sematic actions are associated with the production of the grammer.

ex $E \rightarrow E+T$ { print '+' }

Always will be inside the curly braces

2) SDT is more efficient than SDD

3) It is used in implementation.

•) Syntax Directed Definition :-

⇒ In SDD we are constructing a parse tree which is called Anotated parse tree. (Also called decorated parse tree)

→ Anotated parse tree

⇒ It is a tree in which each node defines the value of the attribute example E.x is unique & E.y is unique

→ Attribute associated with the non terminals.

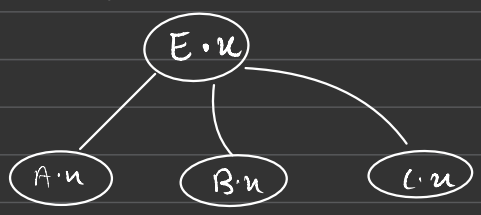
⇒ There are two types of attribute

1) Synthesized
2) Inherited

Synthesized

1) Here the value of the attribute in the node n is evaluated in terms of its childrens or itself

ex :-

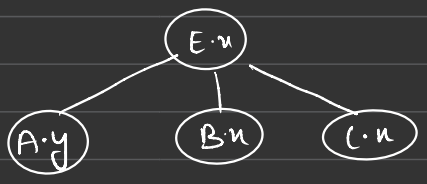


The value of attribute E.x is given by either A.x / B.x / C.x / E.x itself.

Inherited

1) If the value of the attribute is evaluated in terms of its parent or its siblings or itself

ex



A.y = A.y | B.x | C.x | E.x

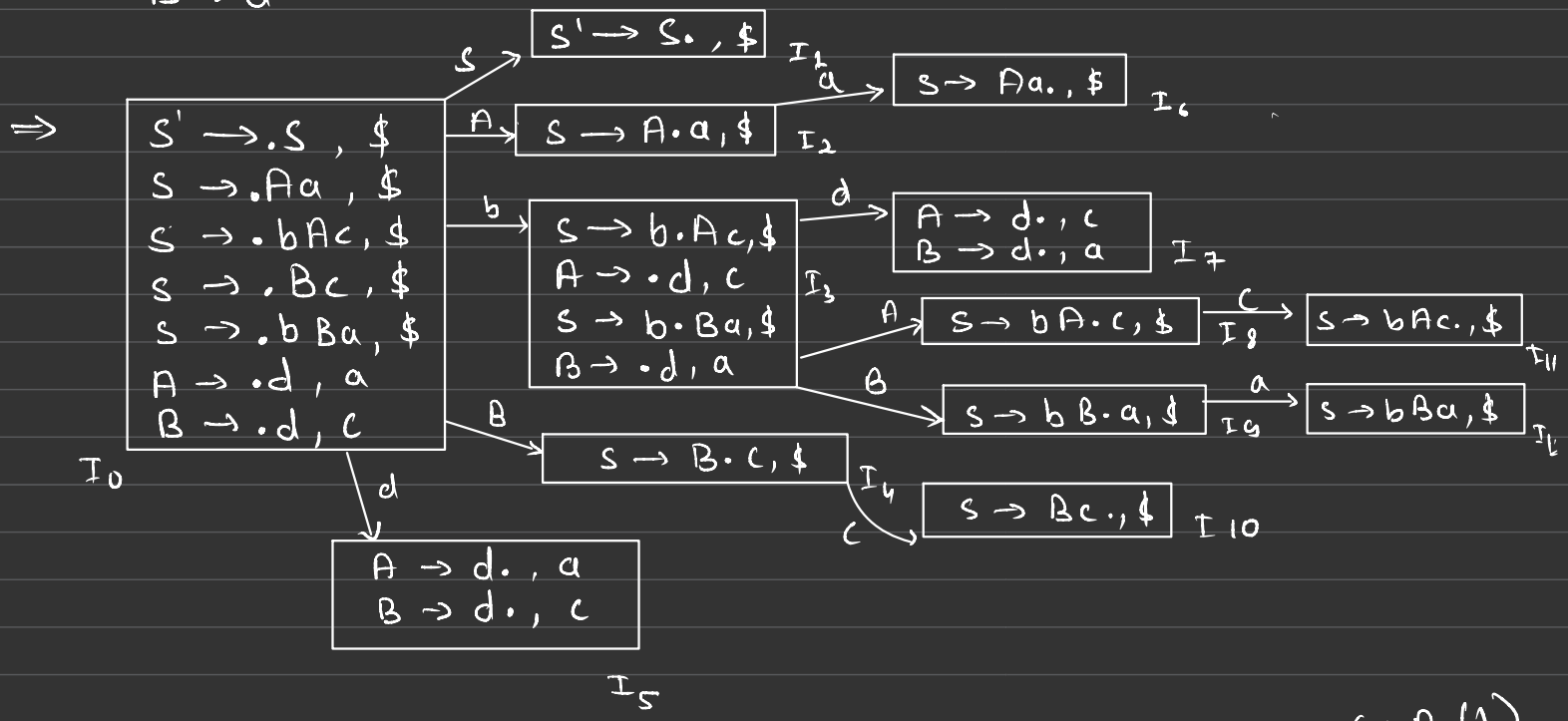
⇒ Attributed associated with only terminals ie synthesized attributes. they cannot be inherited.

Q) Check whether G if it is CLR(1) or not also check for LALR(1)

1) $S \to \overset{1}{A}a \mid \overset{2}{b}Ac \mid \overset{3}{B}c \mid \overset{4}{b}Ba$

$A \to d$ c

$B \to d$ a



CLR (1)

	Action					Goto		
	a	b	c	d	$	S	A	B
I_0		S_3		S_5		1	2	4
I_1					Ac.			
I_2	S_6							
I_3				S_7			8	9
I_4			S_{10}					
I_5	R_5		R_6					
I_6					R_1			
I_7	R_6		R_5					
I_8			S_{11}					
I_9	S_{12}							
I_{10}					R_3			
I_{11}					R_2			
I_{12}					R_4			

	Action				Goto		
	a	b	c	d	$	S	A	B
I_0		S_3		S_{57}		1	2	4
I_1					Ac.			
I_2	S_6							
I_3				S_{57}			8	9
I_4			S_{10}					
I_{57}	R_5/R_6		R_5/R_6					
I_6					R_1			
I_8			S_{11}					
I_9	S_{12}							
I_{10}					R_3			
I_{11}					R_2			
I_{12}					R_4			

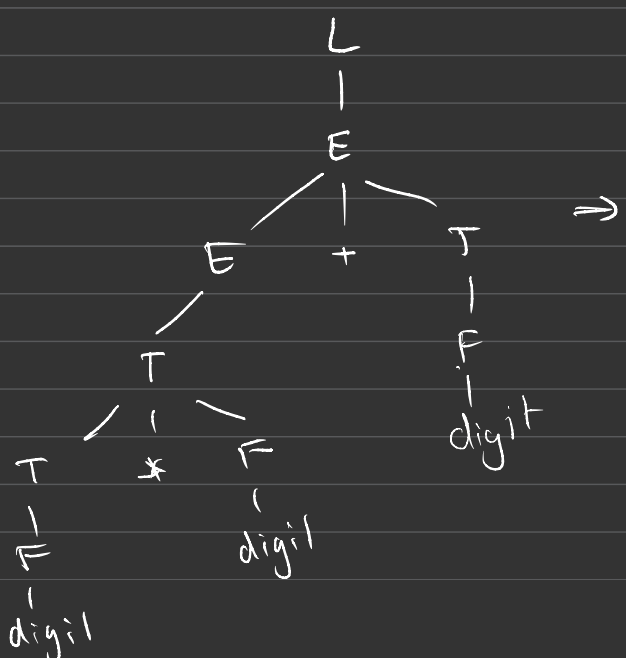
(Q)

Grammer		Semantic Rule
$L \to En$	$\longrightarrow$	$L.val = E.val$
$E \to E_1 + T$	$\longrightarrow$	$E.val = E_1.val + T.val$
$E \to T$	$\longrightarrow$	$E.val = T.val$
$T \to T_1 * F$	$\longrightarrow$	$T.val = T_1.val * F.val$
$T \to F$	$\longrightarrow$	$T.val = F.val$
$F \to (E)$	$\longrightarrow$	$F.val = E.val$
$F \to digit$	$\longrightarrow$	$F.val = digit.lexval$

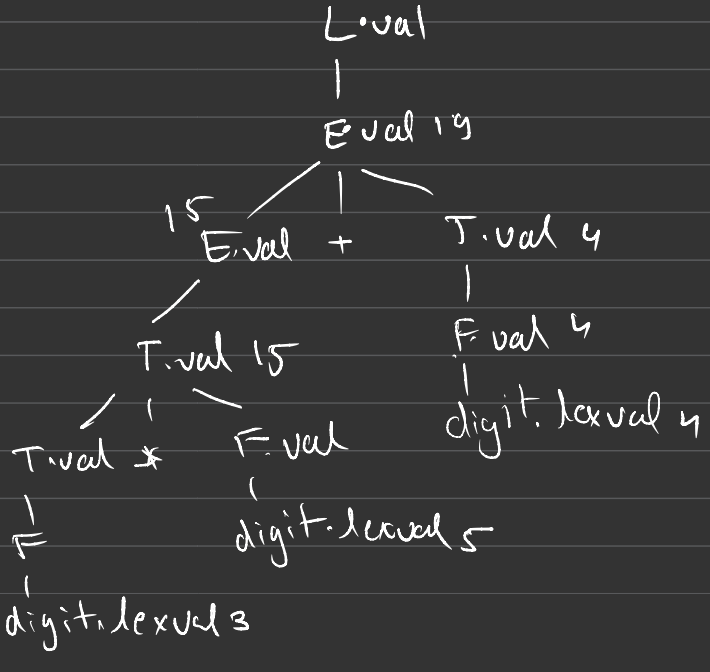
Input string = 3 * 5 + 4 n
↓
end of string.

Construct a annotated parsetree.

⇒



⇒



→ if all the attributes are synthesised then the order of evaluation will be bottom up.

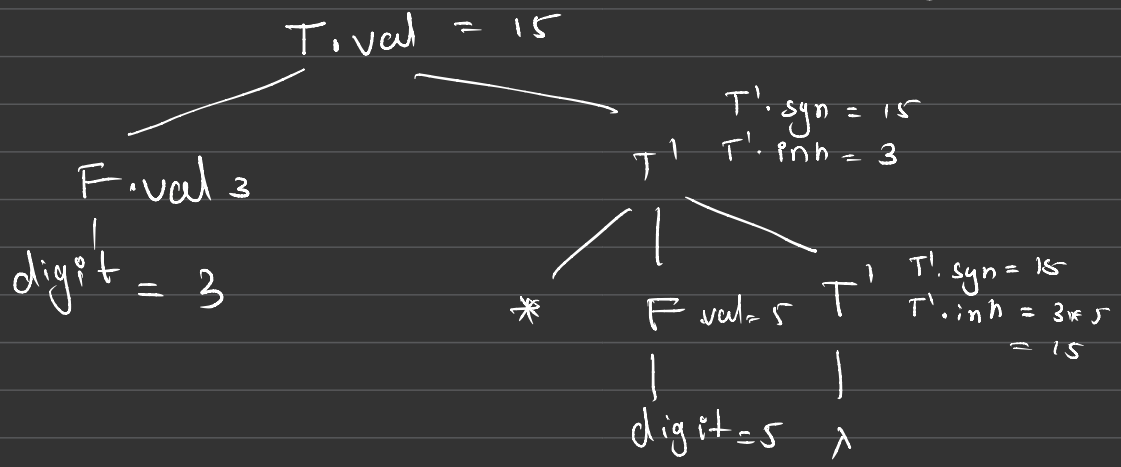
→ if both synthesised & inherited attribute are present then there is no single order of evaluation.

Q)

Grammer		Rules
$T \to FT'$	$\longrightarrow$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \to *FT_1'$	$\longrightarrow$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
$T' \to \lambda$	$\longrightarrow$	$T'.syn = T'.inh$
$F \to digit$	$\longrightarrow$	$F.val = digit.lexval$

Input string = 3 * 5 n
↓
eos

Tree



•) Dependency Graph:-

⇒ It is used to find out the order of evaluation of the attribute value.

→ In the Dependency Graph for each attribute of the symbol there is a node present.

→ The edge in the dependency graph is constructed if the semantic rule is

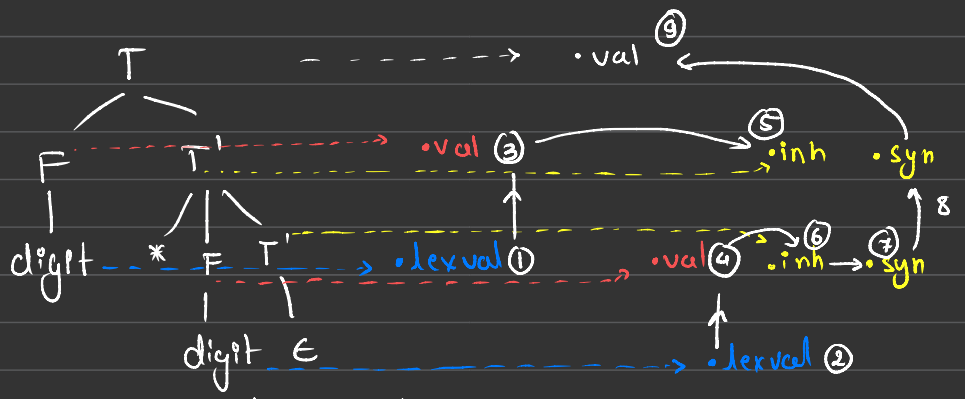
$$x.a = y.b$$

then graph will be ⇒ $y.b \rightarrow x.a$

Example

$T \rightarrow FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.val$
	$T'.syn = T_1'.syn$
$T' \rightarrow \lambda$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$

Step 1 construct a parse tree



•) Topological Sort. order of evaluation : 1 2 3 4 5 6 7 8 9
2 1 4 3 5 6 7 8 9

1) Start the node having indegree zero (To find the order of evaluation).

2) After considering the node delete the node and its outgoing edges and consider the next node having indegree zero.

3) This process continues until all the nodes are traversed.

Note: The order of evaluation is only possible if graph is acyclic & directed.

•) S-attributed definition & L-attributed definition.

⇒ S-attributed

→ In this definition all the attributes are synthesised attribute evaluation is bottom up order of evaluation.

→ It uses post order traversal.

→ It is used in bottom up parser or LR parser.

⇒ L-attributed

→ In this definition either

a) All the attributes are synthesised. } It is used in top down
b) Inherited with some restrictions. } parser.

⇒ R-attributed

→ All attribute are synthesised.

Restrictions:

⇒ If a production $A \rightarrow X_1 X_2 \cdots X_n$

X_i can be evaluated by

1) Either inherited or synthesised attribute of A
2) Either synthesised or inherited attribute of $X_1, X_2 \cdots X_{i-1}$
3) Either synthesised or inherited attribute of X_i itself.

Example

$T \rightarrow FT'$ $\quad T'.inh = F.val$

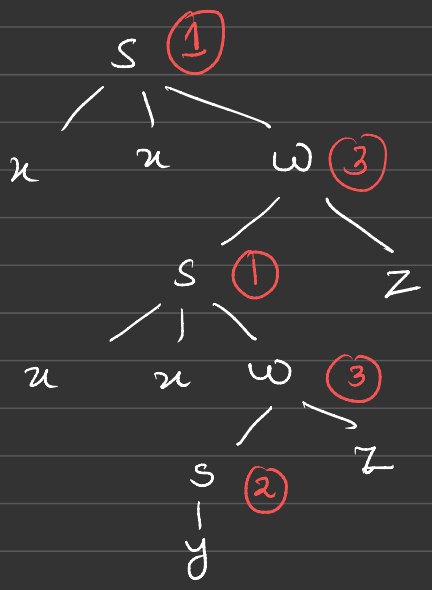
$T' \rightarrow *FT_1'$ $\quad T_1'.inh = T'.inh * F.val$

•) Find out the syntax directed

⇒
$S \rightarrow xxw$ { printf("1") }
$S \rightarrow y$ { printf("2") }
$w \rightarrow Sz$ { printf("3") }

input → xxxxyzz

⇒



⇒ 2 3 1 3 1

Question will be of 5 marks and 1 marks.

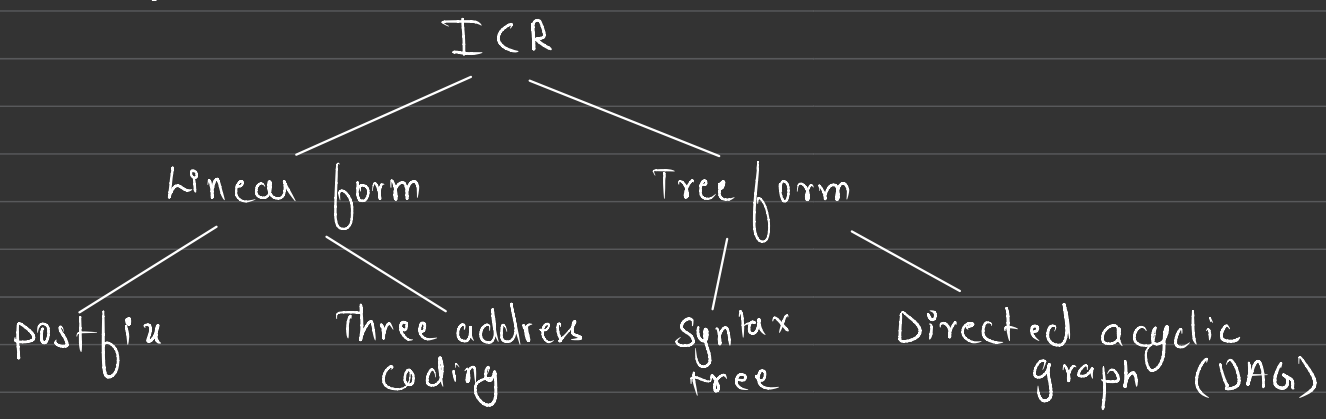
Note: If there are m different way to convert the source code to intermidiate code & n diffent way to convert intermidiate code to target code then m x n no of compilers are possible.

•) Intermidiate Code Generation (Chapter 4)

•) Intermidiate Code representation.

→ There are two ways of representation.

a) Linear form
b) Tree form



•) Directed Acyclic Graph (DAG):

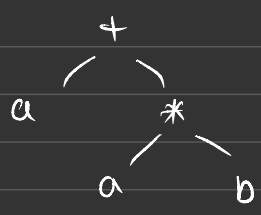
→ The construction of DAGs is similar with the construction of syntax tree but in case of syntax each node has a single parent

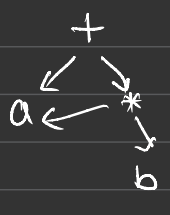
⇒ In DAG a node N can have multiple parents if there is a common sub expression in the expression.

→ SDT will be same for DAG.

Example

Syntax tree a + a * b DAG



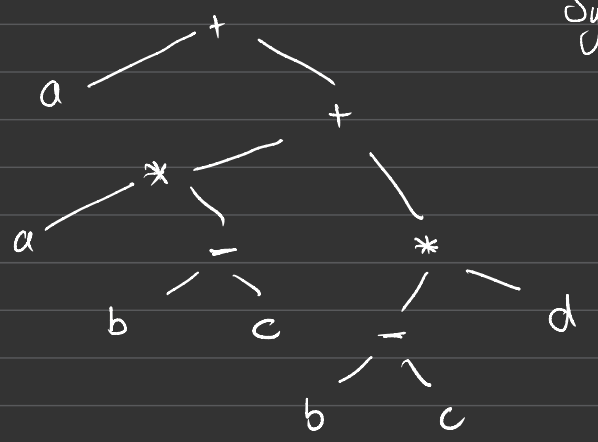


here a has multiple parents.

Example :-

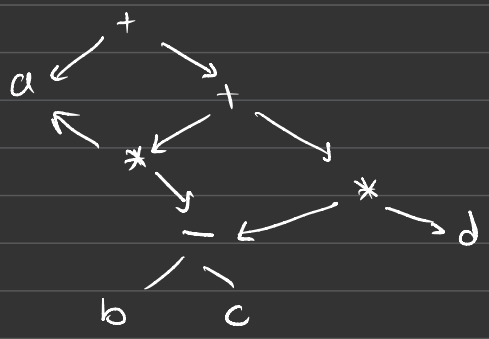
⇒ a + a * (b-c) + (b-c) * d

⇒



Syntax tree.

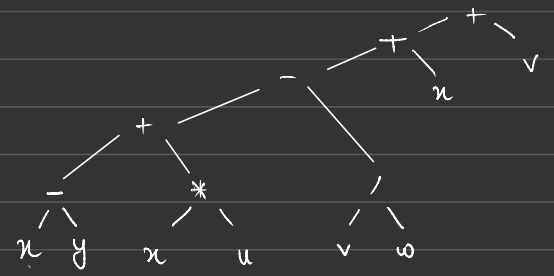
⇒



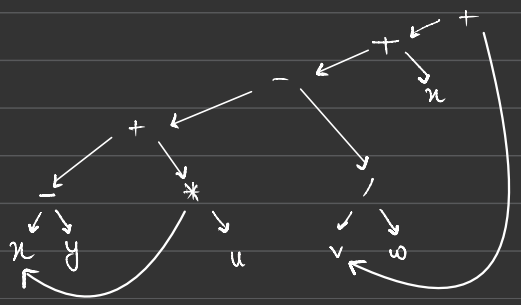
DAG.

$z = x - y + x * u - v / w + u + v$

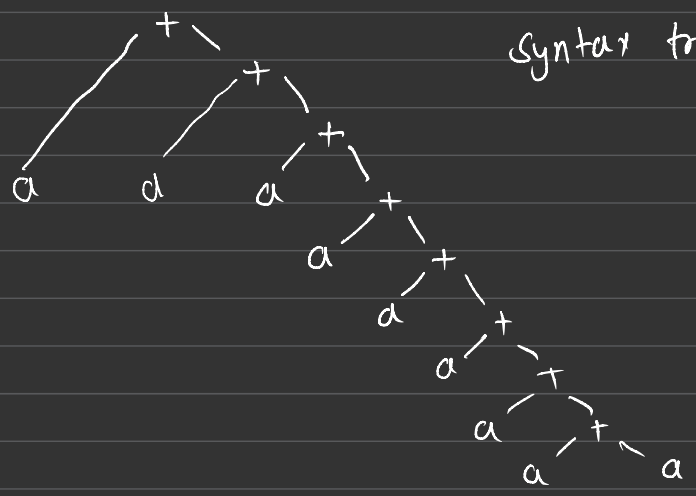
Syntax tree



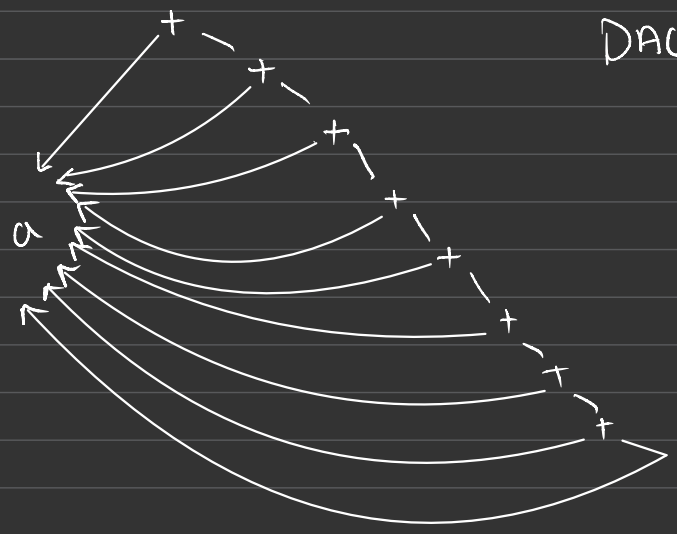
DAG



a) $a + a + (a + a + a + (a + a + a + a))$

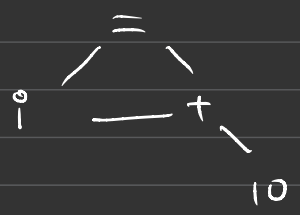


Syntax tree



DAG.

b) $i = i + 10$



Value of

1	id	i	
2	Numb	10	
3	+	1	2
4	=	1	3

→ Three address code

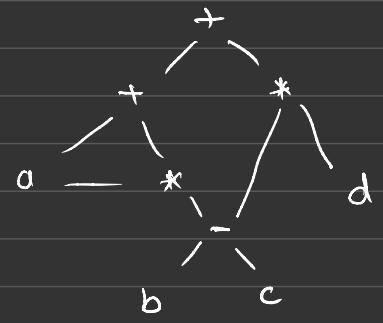
→ There is atmost 1 operator at the right hand side of the assignment

$y = x$
$y = x * z$
} examples.

Example : $x + y * z$

$T_1 = y * z$
$T_2 = x + T_1$

Example



⇒ $t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

⇒ The three address code can be represented by 3 different form

→ qudraple
→ triple
→ Indirect triple.

⇒ In quadruple the records are in this form:-

form < op, arg 1, arg2, result >

op → operator
arg 1, arg2 → arguments

→ Triple

form < op, arg 1, arg2 >

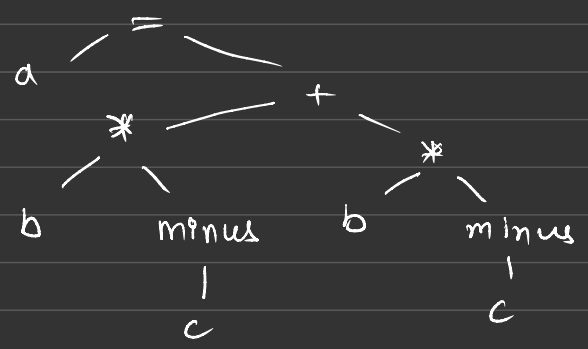
Indirect Triples

In this form we are listing the triples using pointers.

Example :- Find out the quadraple, triple, indirect triple representain of the given expression.

$$a = b * -c + b * -c$$

↓ unitary operator



$t_1 = \text{minus } c \;/ -c$
$t_2 = b * t_1$
$t_3 = t_2$
$t_4 = t_2 + t_3$
$a = t_4$

Quadruple

0	−	c		t_1
1	*	b	t_1	t_2
2>	=	t_2		t_3
3>	+	t_2	t_3	t_4
4	=	t_4		a

•> Triple Representation

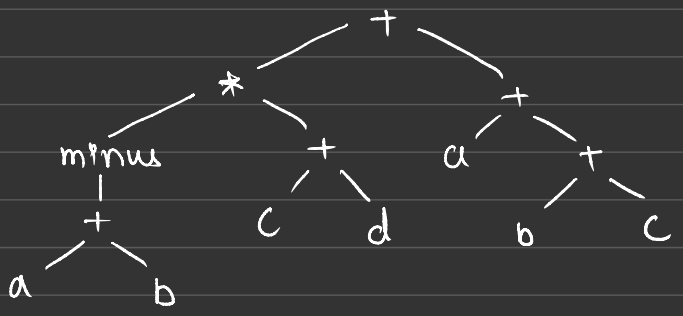
0	-	c	
1	*	b	(0) → address of t1
2	=	(1)	
3	+	(1)	(2)
4	=	(3)	

•> Indirect Triple

→ address location.

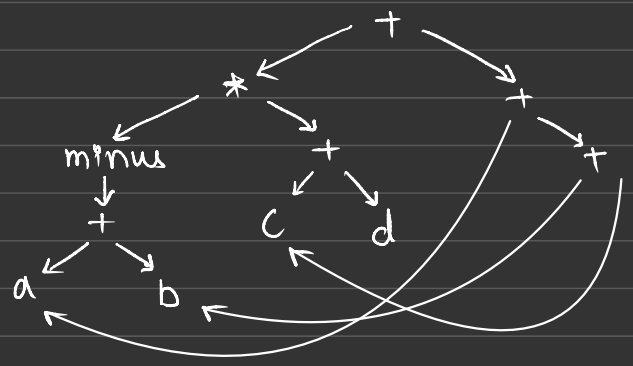
36	(0)	0	-	c	
37	(1)	1	*	b	(0
38	(2)	2	=	(1)	
39	(3)	3	+	(1)	(2)
40	(4)	4	=	(3)	

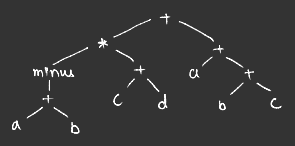
Q) $-(a+b) * (c+d) + (a+b+c)$



Syntax tree.

DAG





$\Rightarrow$

$t_1 = a + b$
$t_2 = \text{minus } t_1$
$t_3 = c + d$
$t_4 = t_2 * t_3$
$t_5 = t_1 + c$
$t_6 = t_4 + t_5$

Quadruple

0	+	a	b	t_1
1	−	t_1		t_2
2	+	c	d	t_3
3	*	t_2	t_3	t_4
4	+	t_1	c	t_5
5	+	t_4	t_5	t_6

Triple Representation

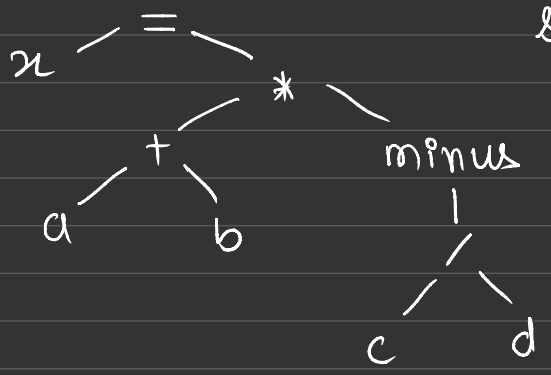
0	+	a	b
1	−	(0)	
2	+	c	d
3	*	(1)	(2)
4	+	(0)	c
5	+	(3)	(4)

Indirect Triple

35	0
36	1
37	2
38	3
39	4
40	5

+	a	b
−	(0)	
+	c	d
*	(1)	(2)
+	(0)	c
+	(3)	(4)

Q) $x = (a+b) * -c/d$



Syntax tree.

$t_1 = a+b$
$t_2 = -c$
$t_3 = t_1 * t_2$
$t_4 = t_3 / d$
$x = t_4$

Quadruple

0	+	a	b	t_1
1	−	c		
2	*	t_1	t_2	t_3
3	/	t_3	d	t_4
4	=	t_4		x

Triple

0	+	a	b
1	−	c	
2	*	(0)	(1)
3	/	(2)	d
4	=	(3)	

Indirect Triple

25	(0)
26	(1)
27	(2)
28	(3)
29	(4)

+	a	b
−	c	
*	(0)	(1)
/	(2)	d
=	(3)	

• Generation of 3 address code using postfix expression.

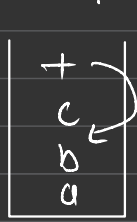
Example

$a + (b+c) / (d+e)$ $\qquad$ $abc+de+/+$

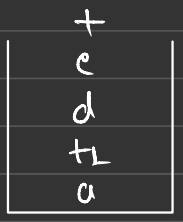
Symbol	Stack	expression
a	(	a
+	(+	a
(	(+(	a
b	(+(	ab
+	(+(+	ab
c	(+(+	abc
)	(+	abc+
/	(+/	abc+
(	(+(	
d	(+/(	abc+d
+	(+/(+	abc+d.
e	(+/(+	abc+de
)	(+/	abc+de+
)		abc+de+/+

Intermidiate code generation using postfix

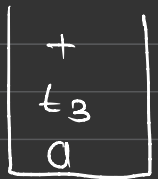
push expression inside the stack.



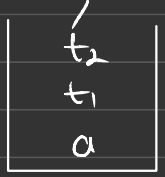
$t_1 = b + c$ $\Rightarrow$



$\Rightarrow$ $t_2 = d + e$



$t_4 = a + t_3$ $\leftarrow$



$t_3 = t_1 / t_2$

$\therefore$ Code = $t_1 = b + c$

$t_2 = d + e$

$t_3 = t_1 / t_2$

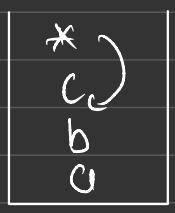
$t_4 = a + t_3$

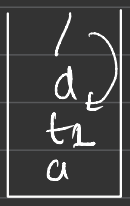
Q) a - b * c / d + e * f

⇒

Symbol	Stack	Expression
a	(	a
-	(-	a
b	(-	a
*	(- *	a
c	(- *	a b c
/	(- * /	a b c *
d	(- /	a b c * d
+	(+	a b c * d / -
e	(+	a b c * d / - e
*	(+ *	a b c * d / - e
f	(+ *	a b c * d / - e f
		a b c * d / - e f * +

intermidiate code generation

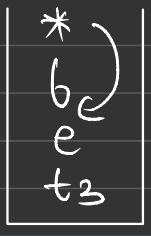
⇒ 

$t_1 = b * c$ ⇒ 

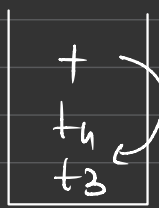
$t_2 = t_1 / d$

$t_3 = a + t_2$

←



$= t_4 = c * f$



⇒ $t_5 = t_3 + t_4$

Code = $t_1 = b * c$
$t_2 = t_1 / d$
$t_3 = a + t_2$
$t_4 = c * f$
$t_5 = t_3 + t_4$

Q) if (x > 0)
x = 0
else
x = 1

⇒ 1) if (x > 0) goto (3)
2) x = 1;
3) goto (5);
4) x = 0;
5) exit

Q) c = 0
do {
if (a < b) then
x++;
else
x--;
c++;
}
while (c < 5)

1) c = 0
2) if (a < b) goto (6)
3) $t_1 = x - 1$
4) $x = t_1$
5) goto (8)
6) $t_2 = x + 1$
7) $x = t_2$
8) $t_3 = c + 1$
9) $c = t_3$
10) if (c < 5) goto (2)
11) end

Q) for loop

• for loop

⇒
```
for( i=0; i<10; i++)
    x = y+z;
```

```
i=0
L1 : if (i<10) goto (L2)
     goto (L3)

L2 :  t1 = y+z
      x = t1
      t2 = i+1
      i = t2
      goto (L1)

L3 : exit
```

• switch case

```
switch (i+j) {
    case 1:  x = y+z;
             break;

    case 2:  a = b+c
             break;

    default: m = n+0;
             break.
```

⇒
```
t1 = i+j
goto (L)

L: if (t1 == 1) goto (L1)
   if (t1 == 2) goto (L2)
   goto (L3)
```

★ 406047 ★

```
L1 :  t2 = y+2
      x = t2
      goto (L4)

L2 :  t3 = b+c
      a = t3
      goto (L4)

L3 :  t4 = n+0
      m = t4
      goto (L4)

L4 :  exit.
```

Q) a[i] = 10 → size of int.

⇒ t1 = add(a) } array
t2 = i x 2 bytes
t3 = t1 + t2
t4[t3] = 10

Q) a[i][j] = 10 formula = (y x n + i) * w

Q) for (i=0; i<5; i++)

b = a[i] + b[i]

```
i=0
if (i<5) goto (L1)
goto (L2)
L1 :  t1 = add(a)
      t2 = i x size of int
      t3 = t1 + t2
      t4 = add(b)
      t5 = t4 + t2
      t6 = t7[t3] + t8[t5]
      b = t6
L2 :  exit
```

Q) if (a > b)
    x = a + b
else
    x = a - b

→ 1) if (a > b) goto (4)
2) x = a - b
3) goto (5)
4) x = a + b
5) exit

•) Final Chapter (Code Optimization)

→ Optimization type

1) Intra block Optimization
2) Intra procedual
3) Inter procedual

Example of Intra block Optimization.

⇒ a := b * -c + b * -c

⇒
$$\left.\begin{array}{l} t_1 = -c \\ t_2 = b * t_1 \\ t_3 = -c \\ t_4 = b * t_3 \\ t_5 = t_2 + t_4 \\ a = t_5 \end{array}\right\} \rightarrow \left.\begin{array}{l} t_1 = -c \\ t_2 = b * t_1 \\ t_4 = b * t_1 \\ t_5 = t_2 + t_4 \\ a = t_5 \end{array}\right\} \left.\begin{array}{l} t_1 = -c \\ t_2 = b * t_1 \\ t_5 = t_2 + t_2 \\ a = t_5 \end{array}\right\} \text{optimized}$$