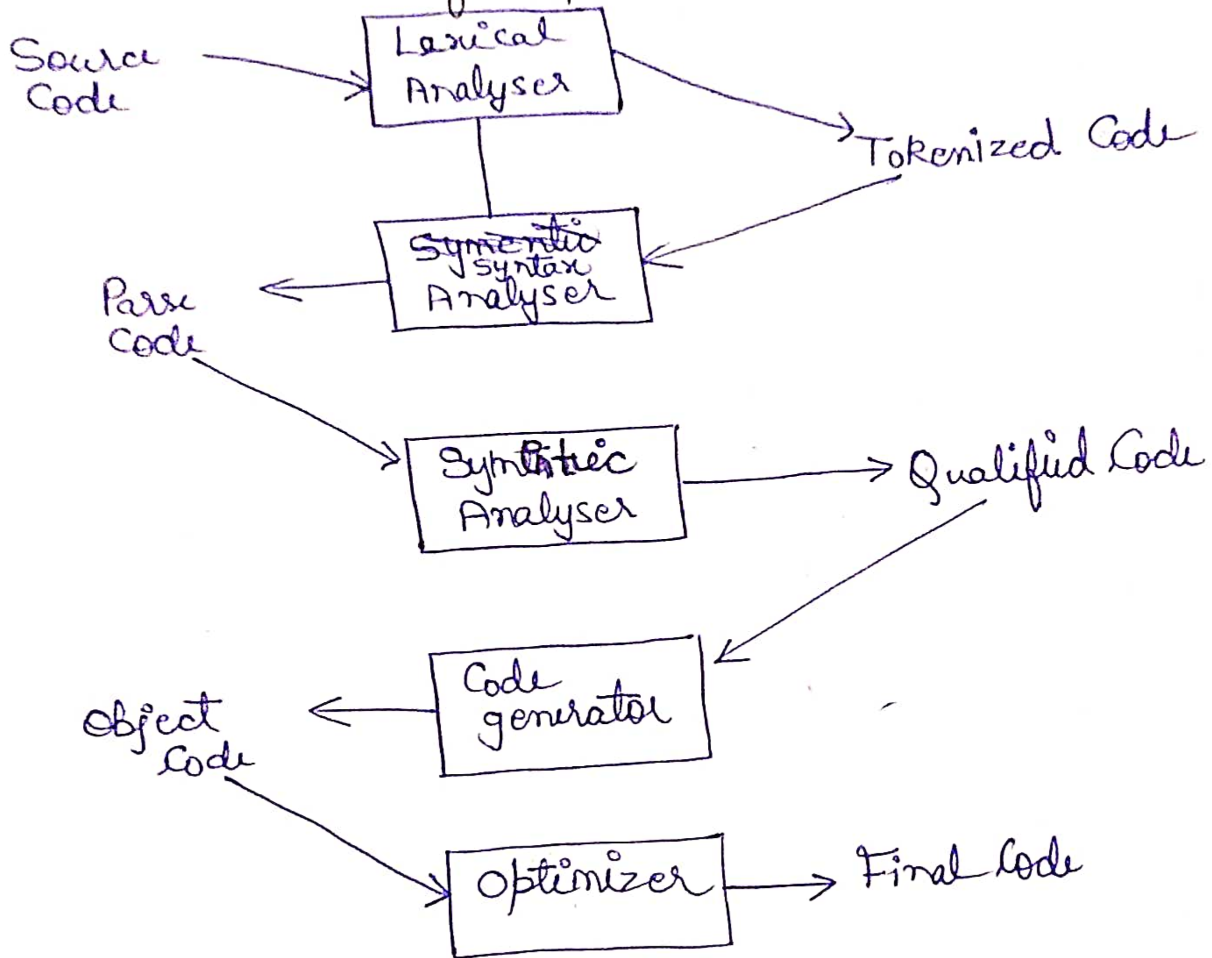
(*) Compiler Design ⇒ A Compiler are the mechanism which is used to translate and Compile the programming source code into machine readable code and it also help to the programmer to find out the error & bugs in the program

(*) The Basic Structure of Compiler :-

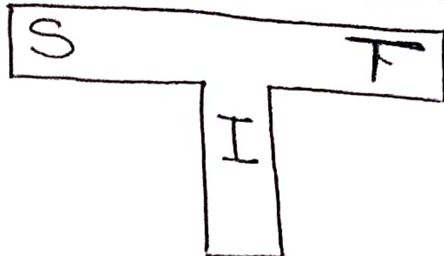


A compiler can be characterised by three languages are as follow :-

(a) Source Language
(b) Implement Language
(c) Target Language

The T-diagram show a compiler $S C_I T$

where, S denotes to Source
I denotes to Implementation
T denotes to Target



(*) Boot Stripping ⇒ It is used to produce a self hosting compiler. Self hosting compiler is a type of compiler that can compile its own source code.
Boot Strup compiler is used to compile the compiler and then you can use this ~~compiled~~ to compile everything else as well as future versions of itself

(*) Finite state machine :-
Finite Automata is a state machine that takes a strings of symbols as I/P changes its state accordingly. Finite automata is a recognizer of regular expression when regular expression string is fed into finite automata it changes its state for each litral. The mathematical model of finite automata consist of :-
(a) Finite Set of States (Q)
(b) Finite Set of I/P Symbols (Σ)
(c) One start State (q_0)
(d) Final State (q_f)
e) Transition Function (δ)
The transition function (δ) mapes the finite set of states (Q) or finite set of I/P symbols (Σ)

$$Q \times \Sigma \rightarrow Q$$

*) Finite Automata Construction :-
Let L(r) be a regular language recognise by some finite automata (FA).
) State ⇒ States of FA are represent by circles state name are written inside the circle

(*) Start State ⇒ The state from which the automata starts is known as start state. Start state has an arrow pointed toward its.

(*) Intermediate State ⇒ All intermediate state have atleast two arrows. One pointing to another pointing out from them.

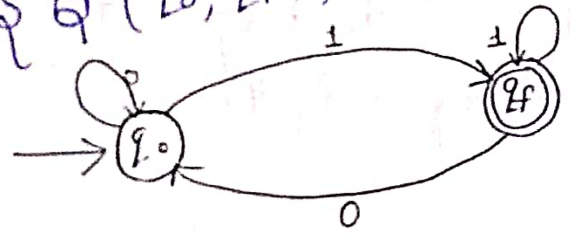
(*) Final State ⇒ If the input string is successfully passed the automata is expected to be in this state final state. The final state is represented by double circle it may have any odd no. of arrows pointing to it and even no. of arrows pointing out from it. The no. of odd arrows are one greater than even. odd = even + 1

(*) Transition ⇒ The transition from one state to another state happens when a desired symbol in the input in found upon transition automata can either move to the next state or stay in the same state. Moment from one state to another is shown as directed arrow.

When the arrow points the destination state. If automata stay on the same state an arrow pointed from a state to itself is drawn.

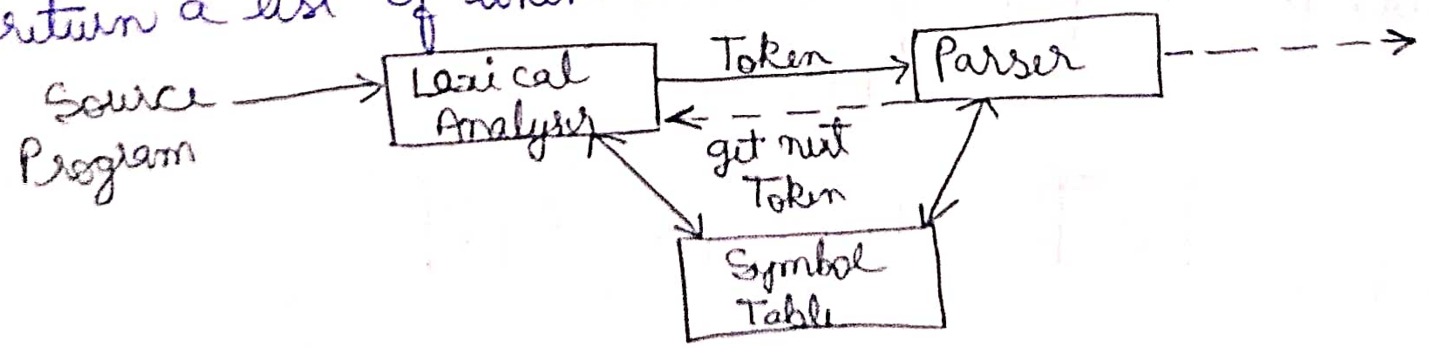
Ex ⇒ We assume finite automata expect any three digit binary number value ending 1.

FA = { Q (q₀, qf), Σ(0, 1), q₀, qf, δ }

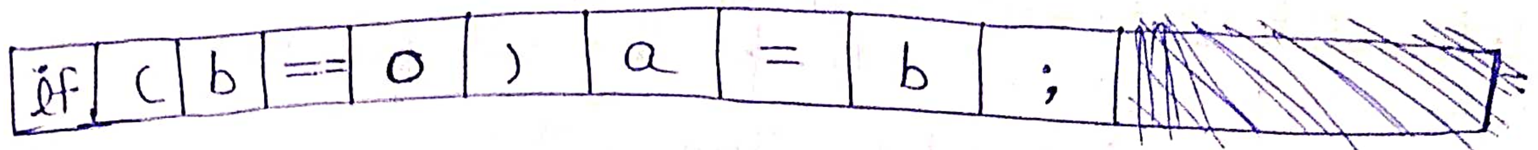


(*) Lexical analyser :-

It reads the source program character by character to produce token normally a lexical analyser does not return a list of token at one short. It return a token



if (b == 0) a = b; (Lexical analyser process)
↓
LA (reduce white space)
↓



- Transform multicharacter input string to Token string.
- Reduce length of program representation (remove space)

*) Token :-
- A token is a pair consisting of token name and optional attribute value.
- Token name is an abstract symbol representing a kind of lexical unit.

Ex⇒ id for identifier.
- The token name are the I/P symbol that the parser processes.
- Each token represent a logically cohesive sequences of character such as identifier, ~~operation~~ operator, Key words.

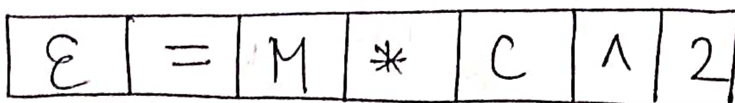
) Pattern:-
It is a description of the form that the lexemes ~~me~~ of token may take.

exemes:-
lexeme is a sequence of token in the program that matches pattern for a token and is identified by the lexical analyser as a instance.

) Regular Expression ⇒ Regular expression are widely use to specify pattern.

) Attribute for a token ⇒ The token names & associate attribute Value for the following Statement

E = M*C^2



* <id point to symbol table entry for E>
* <assign - op>
* <id point to symbol table entry for M>
* <mult - op>
* <id point to symbol table entry for C>
* <Exp - op>
* <number, Integer value 2>.

(*) How to describe Token :-
* Use regular expression to describe programming language expression
* A regular expression is defined inductively
* a ordinary char stand itself.
* ε empty string.
* R|S either R or S (alternation) where R|S = Rε
* RS R followed by S (concatenation)
* R* concatenation of R0 or more time (KC).

(*) Operations on languages :-

(a) Concatenation ⇒ $L_1 L_2 = \{S_1 S_2 / S_1 \in L_1 \text{ and } S_2 \in L_2\}$

(b) Union ⇒ $L_1 \cup L_2 = \{S / S \in L_1 \text{ or } S \in L_2\}$ $S_1 = \{2,3,4\}$, $S_2 = \{2,3,5\}$, $S_1 \cup S_2 = \{2,3,4,5\}$

(c) Exponentiation ⇒ $L_0 = \{\}, L^1 = L, L^2 = L \cdot L$

(d) Kleen closure ⇒ L^* = zero or more character of L.

(e) Positive Closure ⇒ $LL^* | L^+$ = one or more character of L.

Example of Regular expression in Lexical Analysis :-

a) Let L be the set of alphabet consisting of the set upper case and lower case let us as

$$L = \{A - - - - Z, a - - - - z\}$$

b) Let D the set of alpha consisting 10 decimal digit

$$D = \{0 - - - - - - 9\}$$

(*) New languages

a) LUD is the set of letters and digits.

b) LD is the set of string consisting of a letter followed

- <id point to symbol table entry for E>
- <assign - op>
- <id point to symbol table entry for M>
- <mult - op>
- <id point to symbol table entry for C>
- <Exp - op>
- <number, integer value 2>.

*) How to describe Token :-
* Use regular expression to describe programming language expression
* A regular expression is defined inductively
* a ordinary char stand itself.
* ε empty string.
* R|S either R or S (alternation) where R|S = Rε
* RS R followed by S (concatenation)
* R* concatenation of R0 or more time (KC).

*) Operations on languages :-

a) Concatenation ⇒ $L_1L_2 = \{S_1S_2 / S_1 \in L_1 \text{ and } S_2 \in L_2\}$

b) Union ⇒ $L_1 \cup L_2 = \{S / S \in L_1 \text{ or } S \in L_2\}$ $S_1 = \{2,3,4\}$
$S_2 = \{2,3,5\}$,
$S_1 \cup S_2 = \{2,3,4,5\}$.

c) Exponantiation ⇒ $L_0 = \{\}, L^1 = L, L^2 = L \cdot L$

d) Kleen closure ⇒ L^* = zero or more character of L.

e) Positive Closure ⇒ $LL^* | L^+$ = one or more character of L.

Example of Regular expression in Lexical Analyser :-

Example of Regular expression in Lexical Analyser :- a) Let L be the set of alphabet consisting of the set upper case and lower case let us as

$$L = \{A ----- Z, a- ---- z\}$$

b) Let D the set of ~~alpha~~ consisting 10 decimal digit

$$D = \{0 ------- 9\}$$

*) New languages

a) LUD is the set of letters and digits.

b) LD is the set of string consisting of a letter followed

be digits.

(c) L^* is the set of all string of letters including ε of empty string.

(d) $L(L \cup D)^*$ is the set of all strings of letter and digits beginning with a letter.

(e) D^* is the set of all string of one or more digits.

(*) Regular Expression :-

R.E is used to describe token of a programing language

. A R.E is a build up of simpler Regular expression.

Each R.E denotes a language.

A language denotes by a regular expression is called regular se

(*) Regular Definition :-

To write Regular expression for some languages can be difficult because their R.E can be queite complex in those cases we may ~~May~~ use in R.E.

We can give name of R.E and we can use of symbol of define other R.E.

) Optimization of DFA :-

To optimize the DFA we have need to follow the various step. These are as follow :-

Remove all the state that are unreachable ~~that~~ from the initial state by any set of transition of DFA.

) Draw the transition table for all pair of states.

Split the transition table into two tables T_1 and T_2.

T_1 contain all final states.

T_2 contain all non-final states.

Find the similar rows from T_1 such that

$$\delta(q, a) = p$$

$$\delta(r, a) = p$$

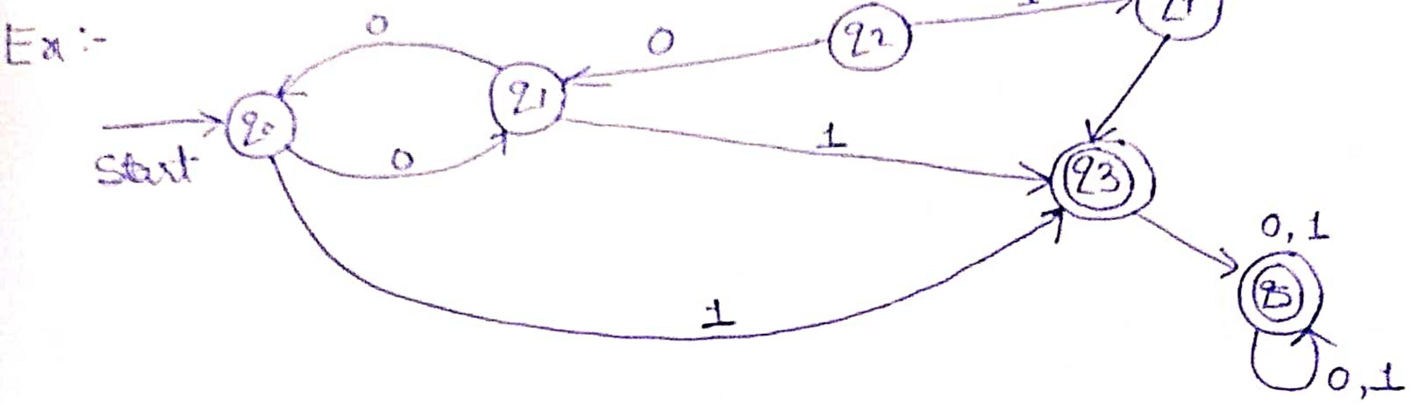
that means find the two rows which have same value and (4) remove one of them.

5) Repeat step (3) untill there is no similar rows are available in the transition table T1.

6) Repeat step (3) and step (4) for table T2 also.

7) Now combine the reduce T1 and T2 table. The combine transition table is the transition table of minimized DFA.

Ex :-



Ans ⇒ In the given DFA q2 are q4 are the unreachable state so remove them.

(•) Draw the transition table for the rest state.

State	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_5	q_5
* q_5	q_5	q_5

(•) Now divide rows of transition table into 2 and sets one. Set contain those rows which start from non-final state.

State	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3

(•) Other set contains ...

State	0	1
* q_3	q_5	q_5
* q_5	q_5	q_5

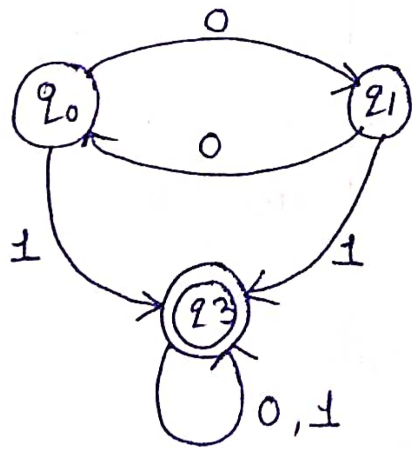
(*) Set 1 has no similar rows so set 1 will be the same.
(•) In set 2 rows 1 and row 2 are similar since q_3 & q_5 transit to same state.
So, skip q_5 and then replace q_5 by q_3 in the rest.

State	0	1
* q_3	q_3	q_3

Now combine set 1 and set 2

State	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_3	q_3

Now its the minimize DFA.



Formal Grammar

Grammer is nothing but the set refers or simply is define by 4 terms (V_n, Σ, P, S)

- $V_n \Rightarrow$ No of non terminal
- $\Sigma \Rightarrow$ terminal
- $Q \Rightarrow$ Production rule
- $S \Rightarrow$ Start State

* Reverse Substitution is not permitted.
For eg $\Rightarrow$ If $S \rightarrow AB$ is a production then WR can replace S by AB but we can not replace AB by S

* No reversion operation is permitted for eg, if $S \rightarrow AB$ is a production, it is not necessary $AB \rightarrow S$ is a production.

(*) Notation Use :-
We use following Notations.
(•) All capital letters like A,B,C---Z are used as variables.
(•) All small letters like a,b,c -- z are used as terminals.
(•) x,y,z denotes strings of terminals
(•) α, β, γ denotes the elements
$(V_n \cup \Sigma)^*$
That is denote string of terminals or variables or both including null string.
(•) Any symbol to the pointer zero is equivalent to null.
(•) If A is any set, then A^* denote the set of all string over A.

$$A^* \longrightarrow \{ \varepsilon \}$$

where ε is the empty string.

Q⇒ Try to recognise language in given grammer.

$G = [\{S\}, \{a,b\}, P, \{S\}]$

where $P = \left\{ \begin{matrix} S \rightarrow aSb \\ S \rightarrow ab \end{matrix} \right\}$

Ans⇒ Since $S \rightarrow aSb$
$S \rightarrow ab$ is a rule that indicate

If this rule can be recorsively applied

$S \rightarrow aSb$

$S \rightarrow aaSbb$ [∵ $S \rightarrow aSb$]

$S \rightarrow aaabbb$ [∵ $S \rightarrow ab$]

We can have any number of a's and equal no. of b's hence, we can guess the language $L = \{ a^n b^n \; n \geq 1 \}$

Imp
Q= Recognise the ~~contain~~ context free language from the given glammer.

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

(*) BNF Notation :- (Backus Naur Form)

It is used to write a formal representation of a context free grammer. It is also used to describe the syntax of a programming language.

BNF Notation is basically just a varient of context Free Grammer.

In BNF, production have the forms

left side → definition

where left side belongs to $(V_n U V_t)^+$

Definion belongs to $(V_n U \text{ final states}(F_t))^*$

In BNF left side contain one non-terminal. We can define the saveral production are separated by a | (vertical bar symbol)

Ex:- There is a production for any grammer are as follows.

S → aSa

S → bSb

S → c

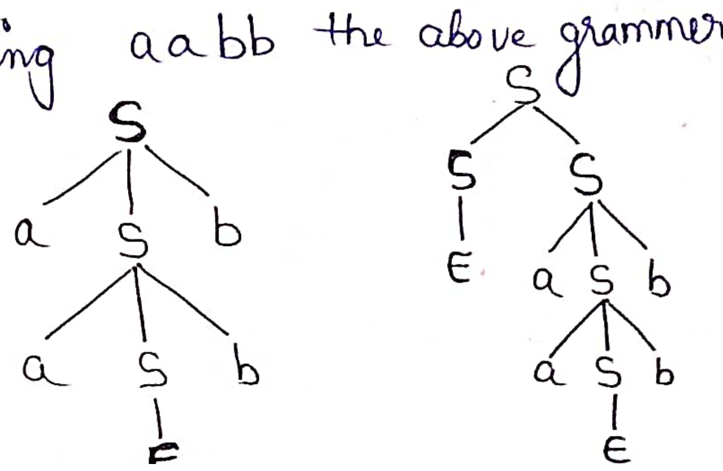
Ans ⇒ S → aSa | bSb | c

(*) Ambiguity :-

A grammer is said to be ambiguous if there exist more than one left most derivation or more then one right most derivation or more then one parse tree for the given input string. If the grammer is not ambiguous then it is called unambiguous.

Ex ⇒ S = aSb | SS

S ⇒ E for the string aabb the above grammer generate two parse tree



If the grammer has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity be re-writing the whole grammer without ambiguity.

(*) YACC :-

It stand for yet another compiler compiler

(a) YACC provides a tools to produce a parser for a given grammer.

(b) YACC is a program design to compile a LALR (one Grammer)

(c) It is used to produce the source of the syntactic analysis of the language produced by LALR (one grammer)

(d) The i/p of YACC is the rule or grammer and the output is a C program.

These are some points about YACC

I/P :- ACFG - File.Y

O/P :- A parser y.tab.c (yacc) [∵ tab stands for table]

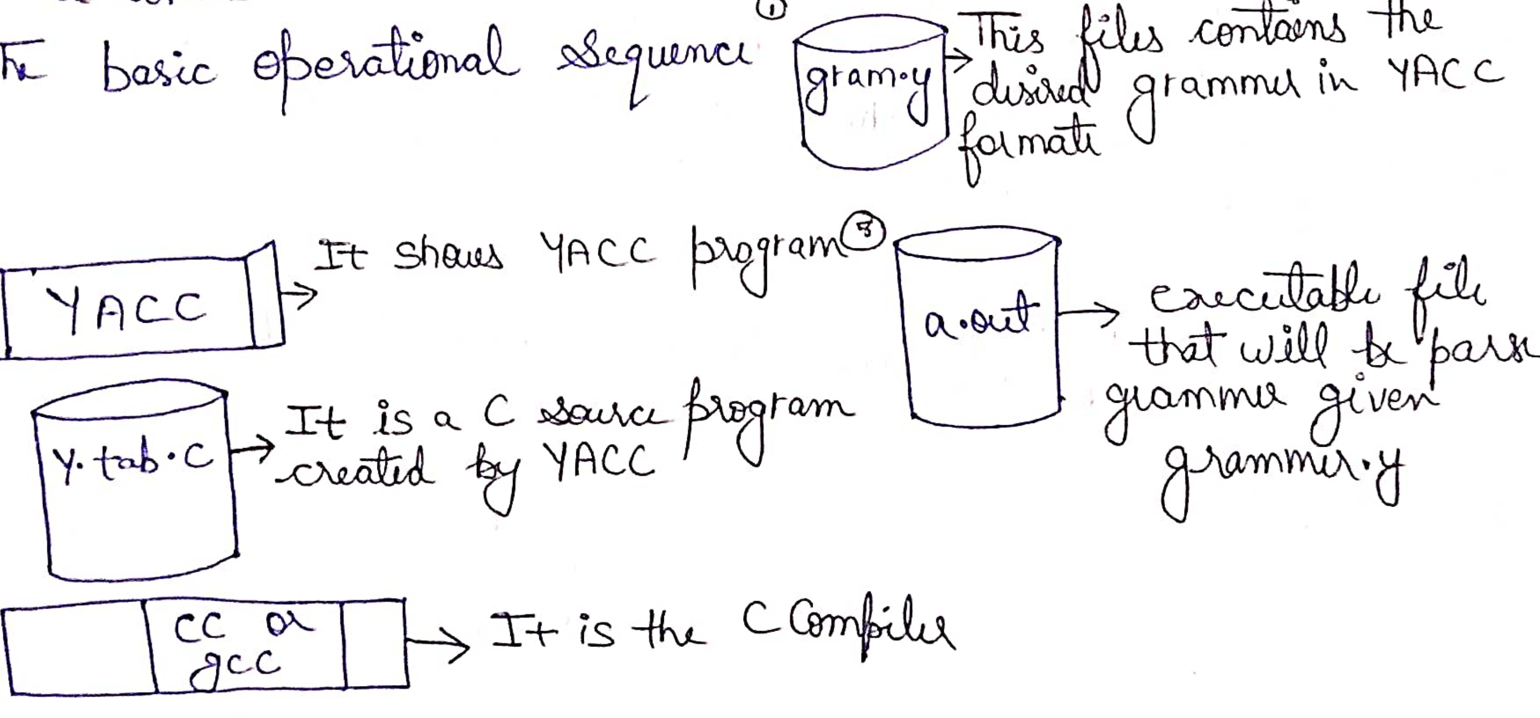
a) The O/P file "File.output" contains the parsing tables

b) The "file.tab.h" contains declaration.

c) The parser called YY parser.

d) The parser except to use the function called YYLEX ( ) to get a tokens.

The basic operational sequence



Write a program ... lexical analyser in phase.

~~#include <stdio.h>~~

algorithm → 1) Enter the String

2)

3) Identify the Keyword & expression by use of loop and if-else.

4) Print the result.

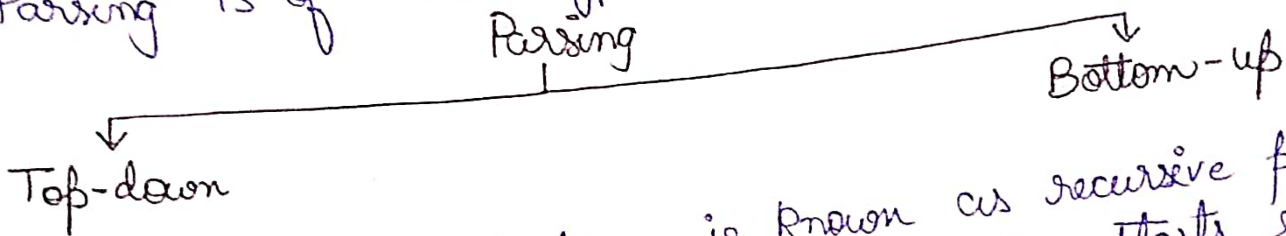
```
#include <stdio.h>
# include <conio.h>
# include< string.h>
Void main()
{
  char St[30];
  long int l;
  clrscr();
  Printf(" Enter the Sentence\n");
  Scanf ("%d", & St );
   l = Strln(St);
   long int i ;
  For (i= 0 ; i< l ; i++)
  {
   if (St[ i] == 'w' && St [i+1] == 'n' && St[i+2]==
          'i' && St [i+3 ] == 'l' && St [i+4]=='e')

  { Printf ("Keyword ");
  }
  else if(St [i]  = 'f' && St[i+1]== 'o' && St[i+2
                = = 'r')

   { Printf (" Keyword");
   }
  elseif(St [i] = = 'd' && St[i+1] =='o')
```

```
{
Printf ("Keyword");
}
else if (St[i]=='g' && St[i+1]=='o'
&& St[i+2]=='t' && St[i+3]=='o')
{
Printf ("Keyword");
}
else if (St[i]==')')
{
Printf ("expression");
}
}
getch();
}
```

Unit = 2

Parsing

(*) Parser ⇒ Parser is a compiler that is used to break the data into smaller coming from lexical analyser into phase. A parser take input in the sequence of tokens and produce output in the form of parse tree.
Parsing is of two types :-



(a) Top-down ⇒ The Top-down is known as recursive parsing
* In the top-down parsing start from the starts symbol and transform it into the input symbol. It is recursive parsing
(b) Bottom-up ⇒ It also known as shift-reduce parsing.
It is used to construct a parse tree for an i/p string.
In the Bottom-up parsing the parsing start with i/p symbol and construct the parse tree upto the start symbol by reaching out the right most derivation of string in reverse.
Bottom-up parsing is classified into the various parsing these are as follows :-

i) Shift Reduce parsing
ii) Operator Precedence Parsing
iii) Table Driven LR Parsing
a) LR (one)
b) SLR (1)
c) CLR (1)
d) LALR (1)

) Shift Reduce parsing is a process of reducing a string to the start symbol of a grammar. Shift Reduce parsing uses a stack to hold the string

A string ⟶ the Starting Symbol reduce to

It perform the two action shift and reduce that is why it is known as Shift Reduce parsing.

- At the shift action, the current symbol in the input string is pushed to a stack.
- At the reduction the symbol will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production

(*) Grammer :-

S → S+S

S → S*S

S → id

i/p string "id+id+id".

Check the i/p string accept the production rule of G.

Stack	I/P String	Action
$	id+id+id$	Shift id
$id	+id+id$	Reduce S→id
$S	+id+id$	Shift +
$S+	id+id$	Shift id
$S+id	+id$	Reduce S→id
$S+S	+id$	Reduce S→S+S
$S	+id$	Shift +
$S+	id$	Shift id
$S+id	$	Reduce S→id
$S+S	$	Reduce S→S+S
$S	$	Accept

Q1 Consider a grammer $E \to 2E2$
$E \to 3E3$
$E \to 4$

i/p string → "32423" Check the i/p string is fit or accept by the grammer or not.

Ans⇒

Stack	I/P String	action
$	32423 $	Shift 3
$3	2423 $	Shift 2
$32	423 $	Shift 4
$324	23 $	Reduce $E \to 4$
$32E	23 $	Shift 2
$32E2	3 $	Reduce $E \to 2E2$
$3E	3 $	Shift 3
$3E3	$	Reduce $E \to 3E3$
$E	$	Accept

Q=2 Consider a grammer $S \to S+S$
$S \to S-S$, $S \to a$

i/p string " a1 − (a2 + a3)

Ans⇒

Stack	I/P String	action
$	a1−(a2+a3) $	Shift a1
$a1	−(a2+a3) $	Reduce $S \to a$
$S	−(a2+a3) $	Shift −
$S−	(a2+a3) $	Shift (
$S−(	a2+a3) $	Shift a2
$S−(a2	+a3) $	Reduce $S \to a2$
$S−(S	+a3) $	Shift +
$S−(S+	a3) $	~~Reduce S→S~~ Shift a3

→

Stack	Input	Action
$S-(S+ a3	)$	Reduce S→ a3
$S-(S+S	)$	Shift)
$S-(S+S)	$	Reduce
$S-S	$	Reduce S→ S-S
$S	$	Accept

(*) There are two Categories of Shift Reduce parsar

(a) Operator Precedence Parsar ⇒ Operator Precedence parsar grammar is a kind of shift reducing parsing method. It is applied to a small class of operator grammar.

Operator precedence parsing grammar has two properties.

(i) No RHS of any production has a ∈

(ii) No Two Non-terminal adjacent.

→ Steps to Solve Operator Precedence Parsing

(i) Check the operator precedence grammer is accepted or not.

(ii) Create operator precedance relation table.

(iii) Parse the given string.

(iv) Generate Parse tree.

Operator Precedence parsing can only established b/w the terminal of grammar. It egnore the non terminal.

There are three operator precedence relation :-

- a > b (means that terminal a has the higher precedence than terminal b).
- a < b (means that terminal 'a' has the lower precedence than terminal 'b'.
- a = b (means terminal 'a' and 'b' both have same precedence.

Priority of operator :-

id, a, b, c → High

$ → low

+ > + (L·S·H, R·S·L)

* > * (L.S.H , R.S.L)

id ≠ id

$ A $ — Accept

* high

\+ less then *

Q- * Consider a grammar

T → id
T → T + T
T → T * T

input string - "id+id*id"

Check the string accept by grammar

Ans ⇒ (R.H.S across the top, L.H.S down the side)

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	X	>
$	<	<	<	A ← Accept

↓ Operator Parsing Presedence table
operator Precedence relation table

Stack	Relation	input String	Action
$	<	id+id*id$	Shift id
$id	>	+id*id$	Reduce T→id
$T	<	+id*id$	Shift +
$T+	<	id*id$	Shift id
$T+id	>	*id$	Reduce T→id
$T+T	<	*id$	Shift *
$T+T*	<	id$	Shift id
$T+T*id	>	$	Reduce T→id
$T+T*T	>	$	Reduce T→T*

$T+T	>	$	Reduce T→T+T
$T	>	$	Accept

(*) Predictive Parsar :-

It is a top-down parsing :-

1) An efficient non--back tracking from off top-down parser called a predictive parsar.

2) LL(1) grammar from which predictive parsar can be constructed automatically.

To construct a predictive parsar we must know :-

(a) Input symbol 'a'.

(b) Non-terminal 'A' to be expanded

(c) Alternatives of production $A \rightarrow a_1 | a_2 | ----- | a_n$

(d) That derives a string beginning with 'a'.

(e) Proper alternatives must be detectable by looking at only first symbol it derives.

(*) Transition Diagram for Predictive parsar :-

To construct the transition diagram of a predictive parsar from a grammar

(a) Eliminate LR from the grammar.

(b) Then Left Factor the grammar.

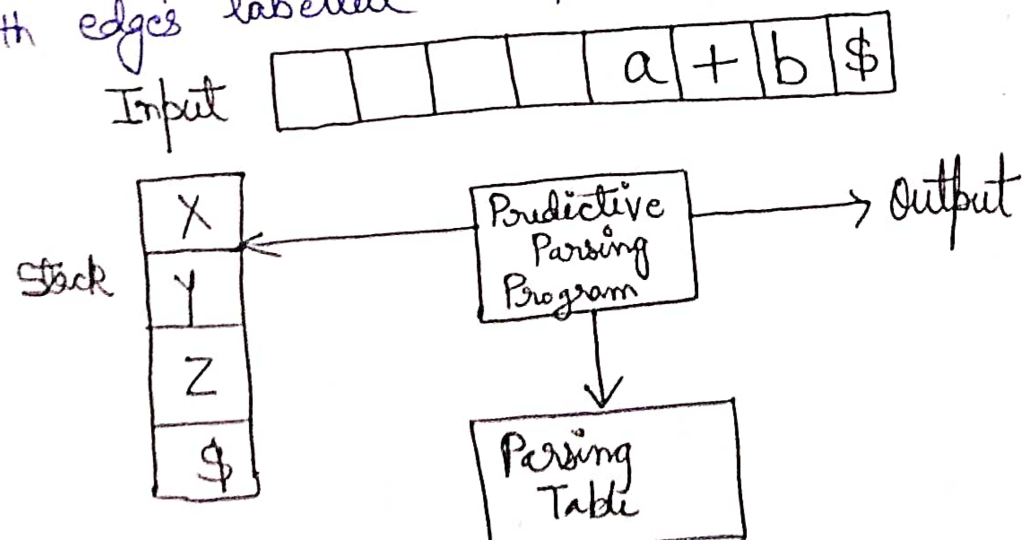
(c) For each non-terminal 'A' do the following steps:-

(i) Create an initial and final state.

(ii) For each production.

(iii) Create a path from initial to final state

(iv) With edges labelled $x_1, x_2 ----- x_n$



It is possible to build non-recursive predictive parser ...
parsar maintaining stack-explicit
recursive - implicit.

Note ⇒ Key problem determining production to be applied for a non-terminal.

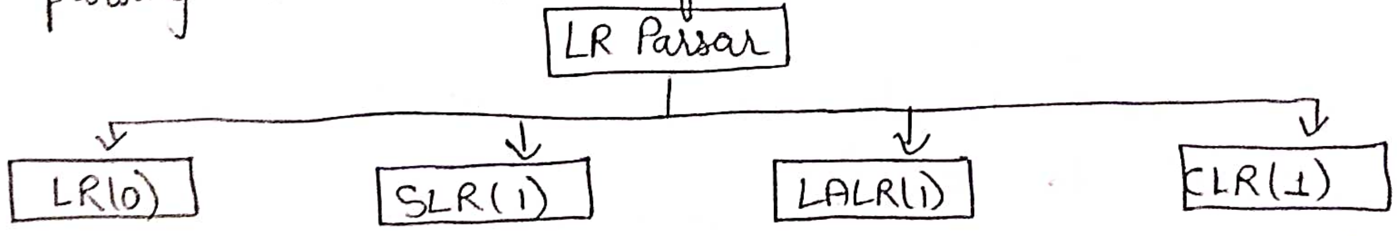
Predictive parsar has some important point :-

(a) Input ⇒ Input contains string to parsar followed by dollar($).
(b) Stack ⇒ Stack has the sequence of grammar symbol with $
(c) Parsing table ⇒ Parsing table contain two dimension array followed by M[X, a]
(d) Output ⇒ It is used to show the output.
(e) $ ⇒ Dollar is a right end marker to indicate end of string.
X → non-terminal
a → terminal
(f) + ⇒ operator.

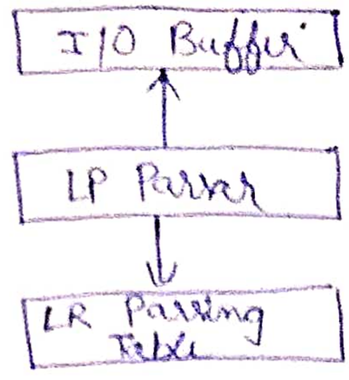
⊛ LR Parsar :-

LR parsing is one type of bottom up parsing it is uses to parse the large class of grammar. In the LR parsing "L" stands for left to right scanning of the I/P "R" stands for constructing a right most derivation in reverse. "K" is the no. of input symbols of the look ahead use to make no. of parsing decision.

LR parsing is divided into four parts



LR algorithm requires stack, i/P output and parsing table in all type of LR parsing i/P-o/P and stack are same but parsing table is different.



I/O buffer is used to indicate end of i/p and it contain string to be parsed followed by a dollar ($) symbol.
A stack is used to contain a sequence of grammar symbol with dollar at the bottom of the stack.
Parsing table is a two dimentional array. It contain action part and go to part.

LR(1) Parsing ∃ In the LR(1) there are various steps in LR(1) parsing.

Step 1 := For the given i/p string write the CFG.
Step 2 := Check the ambiguity of the grammar.
Step 3 := At the augment production in the given gramar.
Step 4 := Create cononical collection of LR(0) items.
Step 5 := Draw a data flow diagram.
Step 6 := ~~Contr~~ Construct a LR(1) parsing table

In the augmented grammar 'G' will be generated if we add one more production in the grammar G. It helps the parser identify when the stop the parsing and announce the acceptance the i/p.

Given Grammar	Augment Grammar G'
$S \rightarrow AA$	$S' \rightarrow S$
$A \rightarrow aA$	$S \rightarrow AA$
$A \rightarrow b$	$A \rightarrow aA$
	$A \rightarrow B$

Canonical Collection of LR(0):-

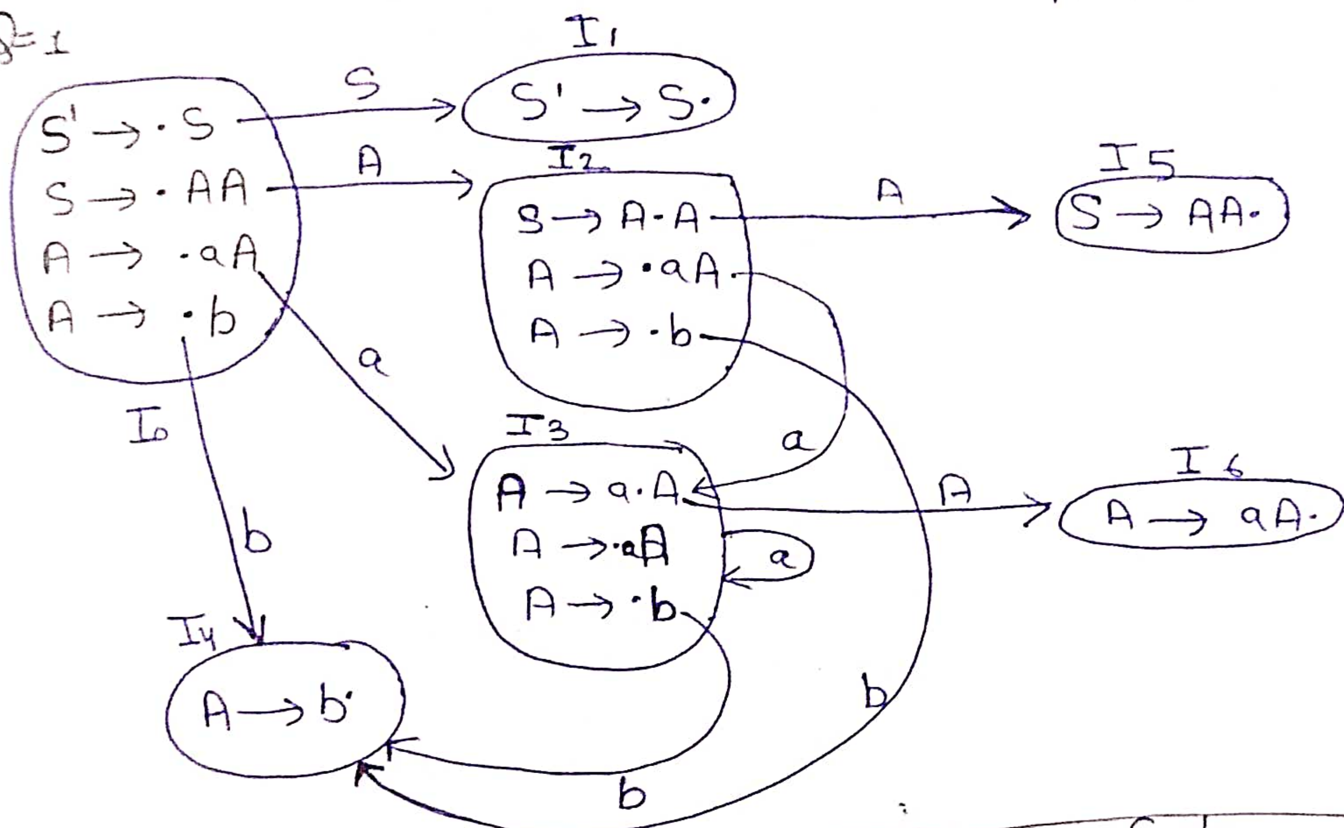
A LR(0) is a production Grammar (G) with '.' sign at the some position of Right side of the production.
LR(0) item is useful to indicates that how much of the i/p has been scanned up to a given point in the process of parsing.
In the LR(0) we replace the reduce node in the entire row.

Example:- Given grammar

S → AA
A → aA
A → b

Augmented grammar with dot(.) symbol

S' → .S
S → .AA
A → .aA
A → .b



Stack	Action a	Action b	Action $	Goto S	Goto A
I_0	S3	S4	~~Acc~~	1	2
I_1			ACC		
I_2	S3	S4			5
I_3	S3	S4			6
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

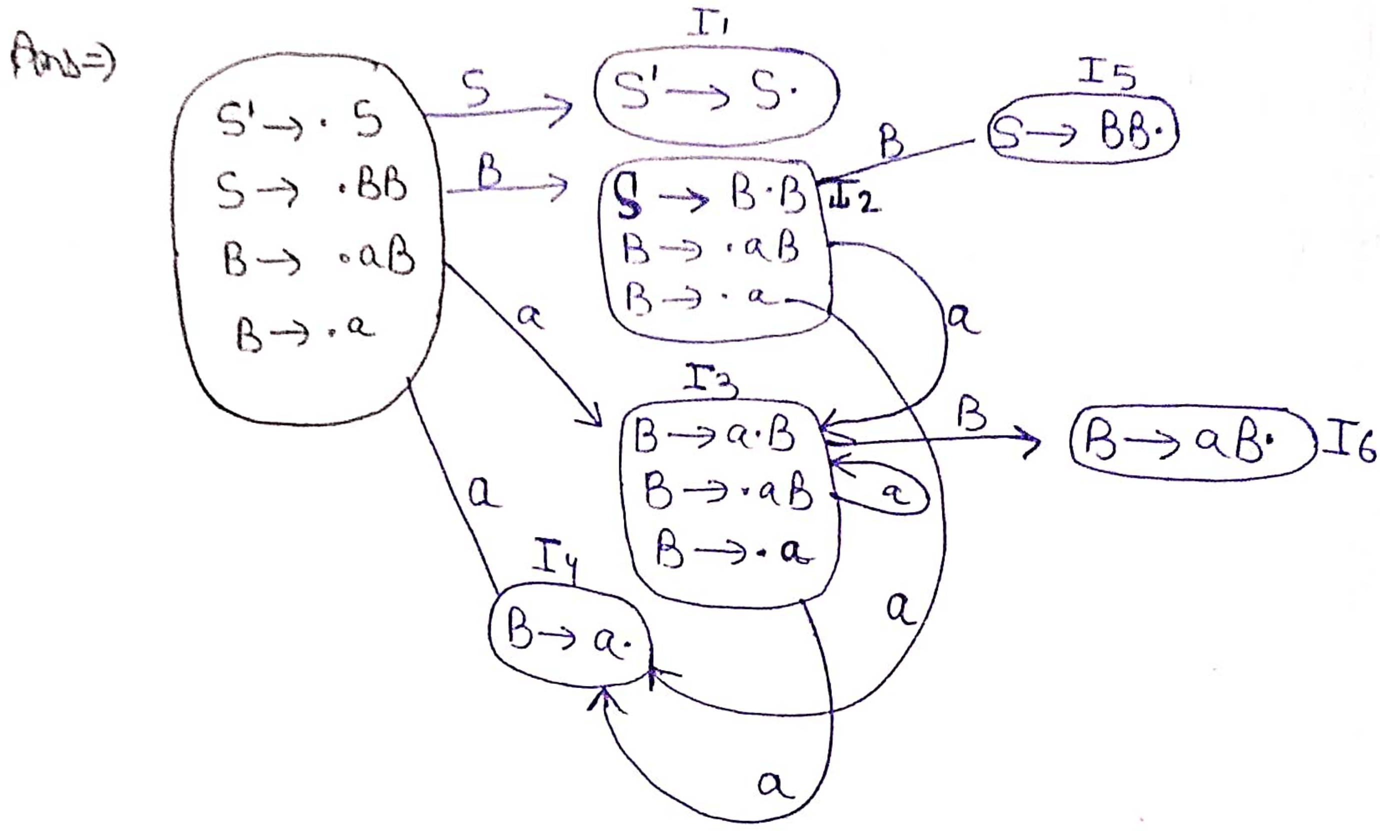
Q=2 $S' \rightarrow \cdot S$

$S \rightarrow \cdot BB$

$B \rightarrow \cdot aB$

$B \rightarrow \cdot a$

Ans=)



(*) Key nodes for making LR(0) canonical collection of tables :-

- If a state is going to some other state on a terminal then its correspondance to a shift move.
- If a state is going to some other state on a variable then its correspondance to go to move.
- If a state contain the final item in the particular row then we write reduce node complitily.

(*) SLR Parsing :-

The SLR Parsing is a type of parsing. SLR stands for simple Left to right parser. It works on complite set of LR(1) and it also generate large cables and large no. of state. Its working is slow in construction

SLR :-

It refers to simple LR parsing it is same as LR(0) parsing. The only difference is in the table. To construct SLR(1) parsing table we use canonical collection of LR(0) item.

In the SLR(1) parsing we place the reduce moves only in the follow of Left Hand Side (LHS).

(*) Construction of SLR Parsing :-

Set of LR(0) item will be the states of 'action' and 'go to'.

(*) Table of the SLR Parsar :-

A collection of set of LR(0) item (the canonical LR(0) collection is the basis for constructing SLR Parser

(*) Augumented Grammer :-

G' is G with a production rule E' → E where E' is the new starting symbol

E → E+T
E → T
T → T*f
T → f
F → id

E' → E
E → ·E + T
E → ·T
T → ·T×F
T → ·F
F → ·id

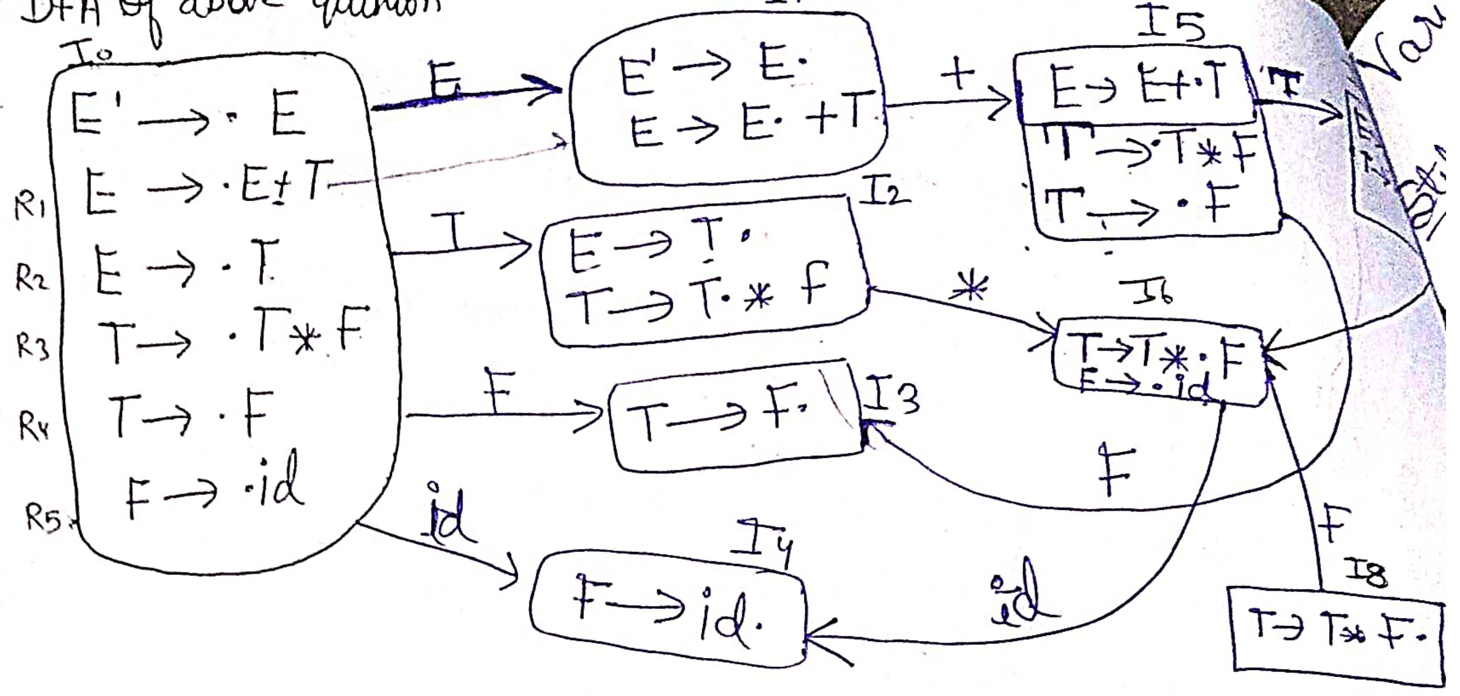
Augmented grammar with dot production

Some closure operation in SLR parsar :-

(a) Language (L) is the set of LR(0) ~~parsar~~ item for grammer (G) then closure (1) is the set of LR(0) item construct from (1) by 2 rules.

Rule 1 ⇒ Initially LR(0) item in L is added to closure 1.

Rule 2 ⇒ If non-terminal is in closure 1 and again non-terminal production rule of ~~graf~~ grammer then non-terminal will be in the closure language



Stack	id	+	*	$	E	T	F
I0	S4				1	2	3
I1		S5		Accept			
I2		R2	S6				
I3	R4	R4	R4				
I4	R5	R5	R5				
I5	S4					7	3
I6	S4						8
I7	R1	S6	R1				
I8	R3	R3	R3				

(In the original, "Action" is written above the id, +, * and $ columns, and "Goto" above E, T and F.)

(*) CLR Parsing :-

CLR refers to canonical look Ahead. CLR parsing used to canonical collection of LR(1). LR(1) items to build the CLR(1) parsing table. CLR(1) parsing product the more no. of states as compared to the SLR(1) parsing. In the CRL(1) We replace the reduce node only in the look ahead symbols.

Various steps involved in the CLR parsing :-

Step1 ⇒ For given input string write a CFG.

Step2 ⇒ Check the ambiguity of the grammer.

Step3 ⇒ Add augment production in the given grammer.

Step4 ⇒ Create canonical collection of LR(0) items.

Step5 ⇒ Draw DFA.

Step6 ⇒ Construct a CLR parsing table.

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

The look Ahead is used to determine that where we place the final item.

The look Ahead always add end symbol for the argument production.

*) CLR :-

Grammer

S → AA

A → aA

A → b

Augment Grammer with System

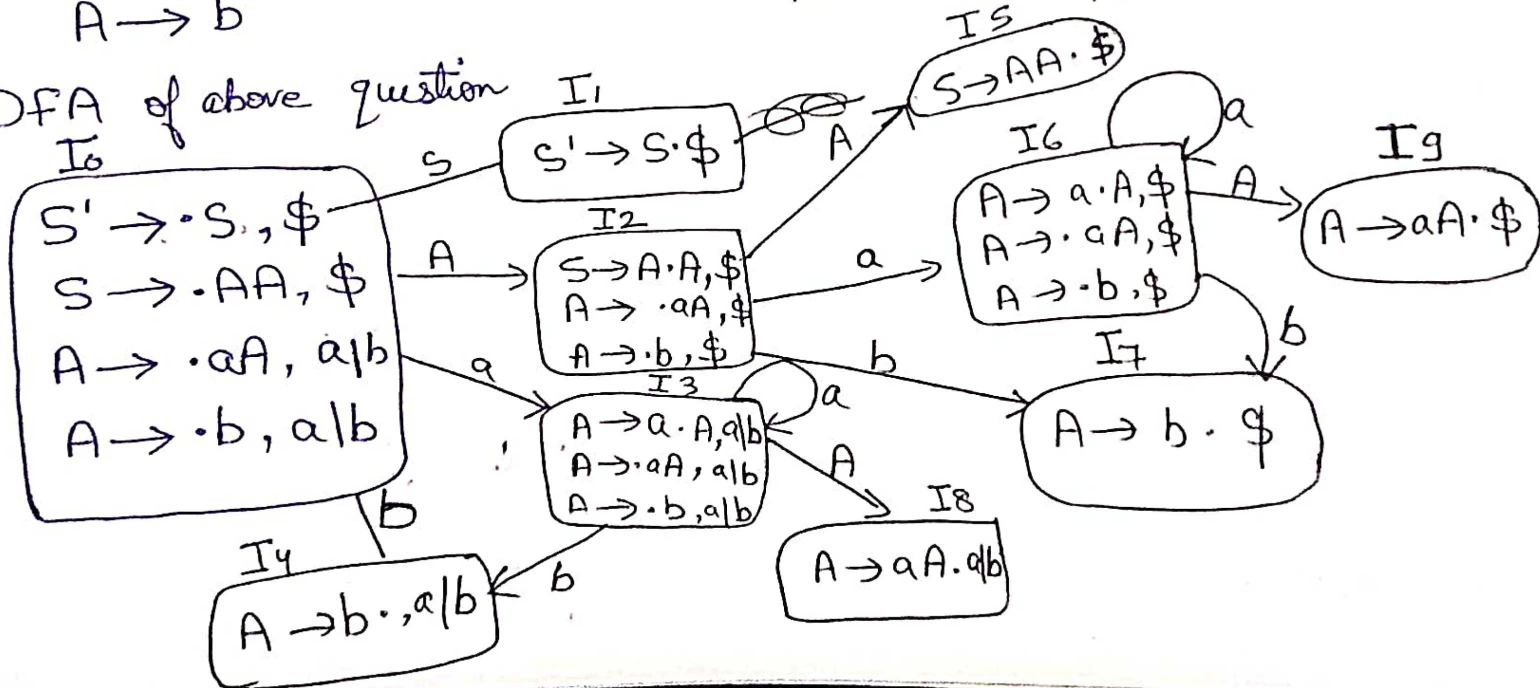
S' → .S, $

S → .AA, $

A → .aA, a/b

A → .b, a/b

DFA of above question



State	Action a	Action b	Action $	Goto S	Goto A
I_0	S3	S4		1	2
I_1			Accept		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	R3	R3			
I_5			R1		9
I_6	S6	S7			
I_7			R3		
I_8	R2	R2			
I_9			R2		

(*) LALR Parsing :-

LALR Parsing , LALR stands for "LOOK Ahead Left to right" Parser, in this LALR Parsing we have need to follow two steps.

Step 1 ⇒ Design LR(1) Parser :-

(a) Augmented grammar

(b) Calculated of first set

(c) Transition Diagram

(d) LR(1) Parsing table

Step 2 ⇒ Design LALR Parsar :-

(a) Find states having same production and merge both the states.

(b) Transition diagram.

(c) LALR Parsing Table.

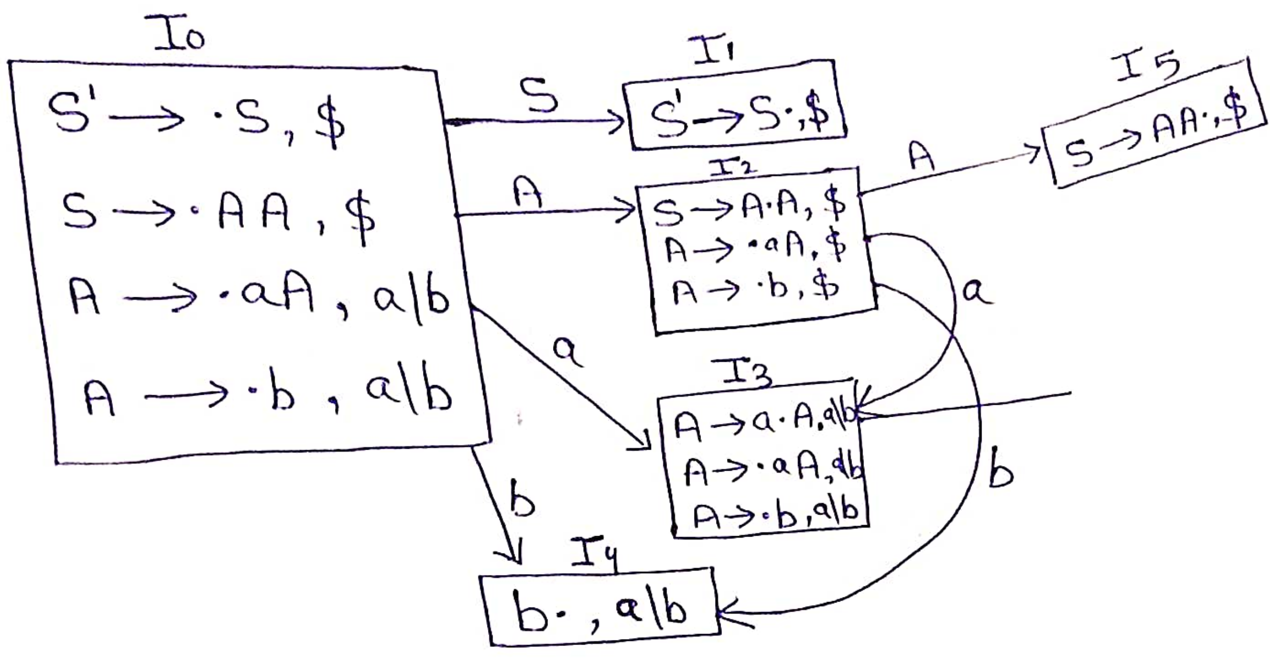
(H)

1 $S \to AA$
$A \to aA$
$A \to b$

$S' \to \cdot S, \$$
$S \to \cdot AA, \$$
$A \to \cdot aA, a|b$
$A \to \cdot b, a|b$

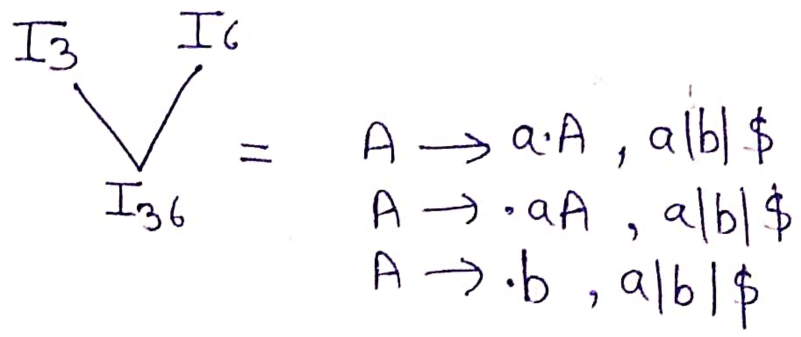
by CLR ?

Ans ⇒

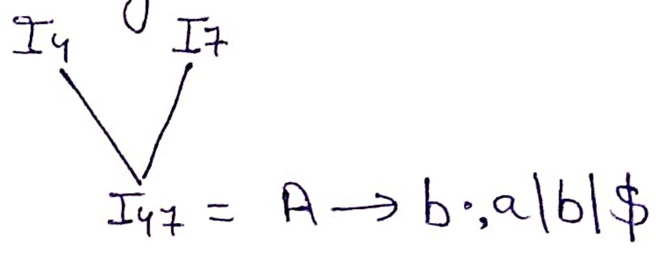


1) The above table does not contain multiple entries this given grammer is LR(1).

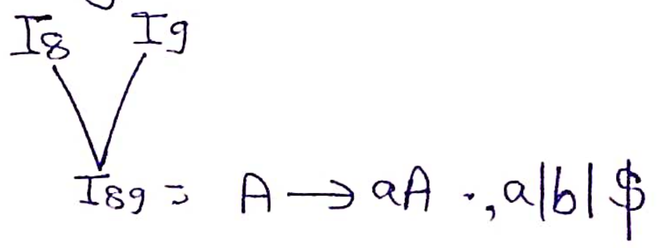
2) Merge state I_3 and state I_6 because their production is same and look ahead doesn't matter.



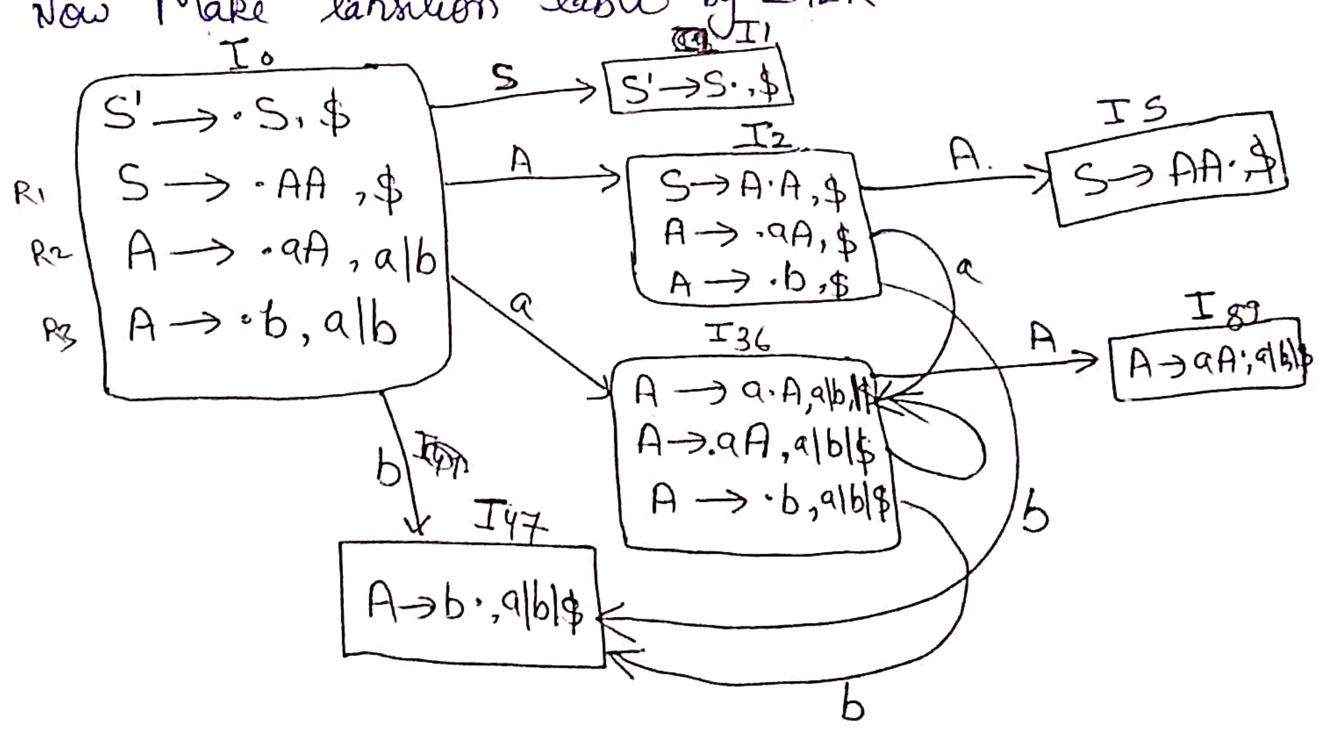
following the same merge I_4 and I_7



following same merge state I_8 and I_9



Now Make Transition table by LALR



State	Action			Goto	
	a	b	$	S	A
I_0	S36	S47		1	2
I_1			Accept		
I_2	S36	S47			5
I_{36}	S36	S47			89
I_{47}	R3	R3	R3		
I_5			R1		
I_{89}	R2	R2	R2		

The given Grammar is LALR(1) because in table entries can not conflict

Q=2 $S \rightarrow BB$
$B \rightarrow aB$
$B \rightarrow b$

Ambiguous Grammar:-

A CFG is said to be ambiguous is if there exist more than one derivation tree for the given i/p string i.e; more then one left most tree and right most tree.

Definition of ambiguous grammer:-

$G = (V_n, T, P, S)$

is said to be CFG ambiguous if and only if there exist string in that more then one parse tree where V_n stand for non-terminal, T for Terminal, P stand for Production rule and S stand for starting symbol

There are two types of derivation:-

(a) Left Most derivation (b) Right Most derivation

Ex ⇒ Describe the string 'aba' for Left Most derivation using a CFG given by

$S \to XYX$
$X \to a$
$Y \to b$

L.M.D

$S \Rightarrow XYX$
$S \Rightarrow aYX$ [$X \to a$]
$S \Rightarrow abX$ [$Y \to b$]
$S \Rightarrow aba$ [$X \to a$]

R.M.D

$S \Rightarrow XYX$
$S \Rightarrow XYa$ [$X \to a$]
$S \Rightarrow Xba$ [$Y \to b$]
$S \Rightarrow aba$ [$X \to a$]

Q=1 Describe the string 'aabbabba' for LMD and RMD using a grammer given by

$S \to aB \mid bA$
$A \to a \mid aS \mid bAA$
$B \to b \mid bS \mid aBB$

LMD

$S \to aB$

$S \to aaBB$ [$B \to aBB$]

$S \to aabB$ [$B \to b$]

$S \to aabbS$ [$B \to bS$]

$S \to aabbaB$ [$S \to aB$]

$S \to aabbabS$ [$B \to bS$]

$S \to aabbabbA$ [$S \to bA$]

$S \to aabbabba$ [$A \to a$]

RMD

$S \to aB$

$S \to aaBB$ [$B \to aBB$]

$S \to aaBbS$ [$B \to bS$]

$S \to aaBbbA$ [$S \to bA$]

$S \to aaBbba$ [$A \to a$]

$S \to aabSbba$ [$B \to bS$]

$S \to aabbAbba$ [$S \to bA$]

$S \to aabbabba$ [$A \to a$]

(*) Syntax Directed Translation:-

In the SDT along with the grammar we associated some informal notations and these notations are called Symantic

Grammar + Semantic Rule = SDT

In the Syntax directed Translation every non-terminal can get one or more then one attributes or some times zero attribute depending on the type of the attribute. The value of these attribute is evaluated by the semantic rule associated with the production rule.

In Symantic Rule attribute is VAL and an attribute may hold anything like a String, a number, a memory location and a complex record.

In Syntax directed translation, whenever a construct encountered in the programming language then it is translated according to the Symantic rule define in that particular programming language

Production	Semantic Rule
$E \Rightarrow E + T$	E.VAL := E.VAL + T.VAL
$E \rightarrow T$	E.VAL := T.VAL
$T \rightarrow T * F$	T.VAL := T.VAL * F.VAL
$T \rightarrow F$	T.VAL := F.VAL
$F \rightarrow (F)$	F.VAL := F.VAL
$F \rightarrow num$	F.VAL := num.lexVAL

- E.VAL is one of the attribute of E
- num.lexVAL is the attribute greater return by the lexical analyser.

(*) Syntax Derected Translation Scheme:-

It is a CFG.

- The SDTS is used to evaluate the order of symentic rule
- In translation scheme. Symentic rule are embeded within the right side production.
- The position at which an action is to be ~~exdited~~ executed is shown by enclose b/w bracket.

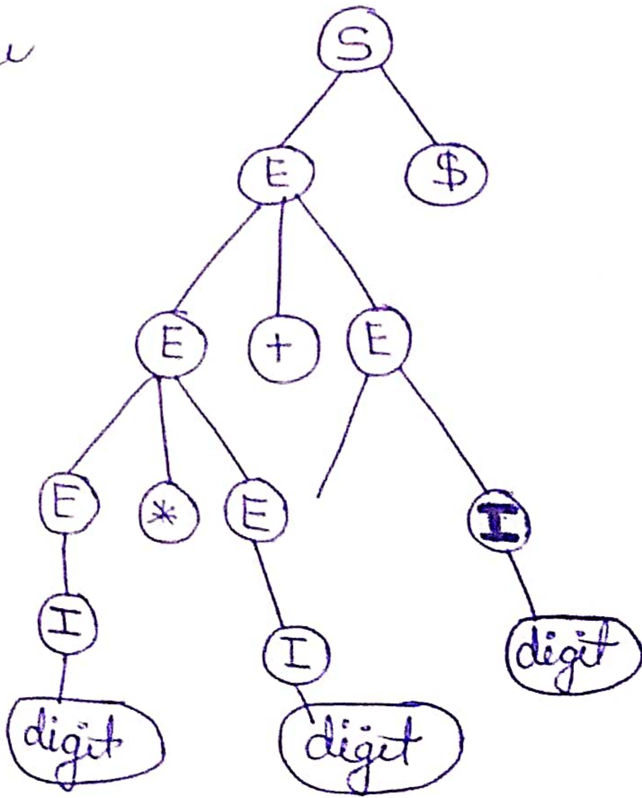
(*) How to implement syntax directed translation:- ~~is emplem~~

Is implemented by constructing a parse tree and performing the action left to right and depth of first order.

Ex:-

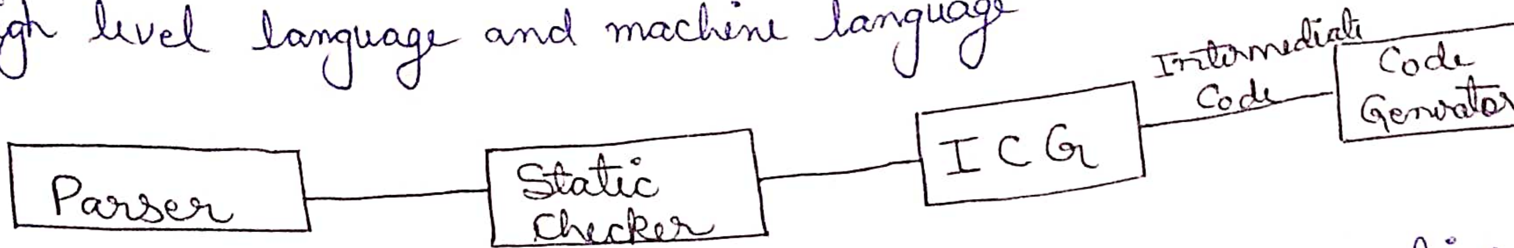
Production	Semantic Rule
S → E $	{ print E.VAL }
E → E+E	{ E.VAL := E.VAL + E.VAL }
E → E*E	{ E.VAL := E.VAL * E.VAL }
E → E	{E.VAL := E.VAL }
E → I	{ E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVA
I → digit	{ I.VAL := LEXVAL }

Parse tree



Intermediate Code :-

Intermediate code is used to translate the source code into machine code. And intermediate code lies b/w the high level language and machine language



(*) Code Generator :-

- If the compiler directly translate source code to machine code without generating then full native compiler is required for each new machine.
- The intermediate code need keeps the analysis portion same for all the compiler i.e; why does not mean a full compiler for every unique machine.
- Intermediate code Generator recived i/p from its syntactic phase and symentic phase. It takes i/p in the form of an anonated syntax tree
- Using the intermediate code, the second phase of the compiler syntax phase is change accordingly to the target machine.

(*) There are two ways to represent intermediate Code :-

1) High level intermediate code ⇒ In High level intermediate code can be represent as source code. To enhance the performance of source code, we can easily apply code modification but to optimise the target machine.

2) Low level intermediate Code ⇒ is close to the target machine. which makes it suitable for register and memory allocation. It is used for machine dependent optimization.

(*) Post fix notation :-

1) It is the useful for of intermediate code in the given language expression.

2) Post fix notation is also called as suffix notation and reverse polish.

3) It is a linear representation of a syntax ~~tor~~ tree.

4) In the Post fix notation any expression can be written without paranthesis

5) In PfN operator follow operands.

Q=1 ((A − (B+C)) * D) ↑ (E+F)

Ans⇒

Symbol	Stack	Postfix

Q=2 (A+B)+(C*D)

Ans⇒

Symbol	Stack	Postfix
(	(	
A		A
+	(+	A
B	(+	AB
)	~~(+)~~	AB+
+	+	AB+
(	+(	AB+
C	+(	AB+C
*	+(*	AB+C
D	+(*	AB+CD
)	+~~(*)~~	AB+CD*
—	+	**AB+CD*+**

Q=3 (e+f) * (g)

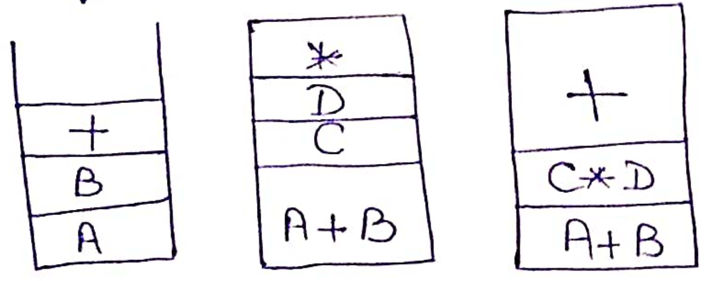
Ans⇒

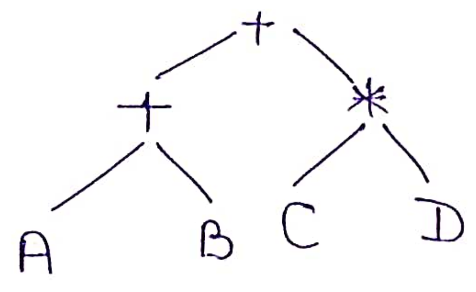
Symbol	Stack	Postfix
(	(	
e	(	e
+	(+	e
f	(+	ef

)	~~(+)~~	ef+
*	*	ef+
(	*(	ef+
g	*(	ef+g
)	*~~( )~~	ef+g
—	*	**ef+g*** ans.

Syntax tree of (A+B)+(C*D)

postfix: AB+CD*+





Q=1 ((A-(B+C))*D) ↑ (E+F)

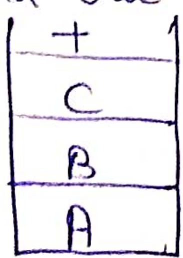
Ans=)

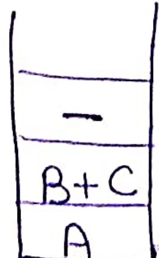
Symbol	Stack	Postfix
(	(	
(	((	
A	((	A
—	((-	A

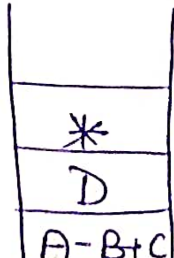
Symbol	Stack	Postfix
(	((−(	A
B	((−(	AB
+	((−(+	AB
C	((−(+	ABC
)	((−~~(+)~~	ABC+
)	(~~(−)~~	ABC+−
*	(*	ABC+−
D	(*	ABC+−D
)	~~(*)~~	ABC+−D*
↑	↑	ABC+−D*
(	↑(	ABC+−D*
E	↑(	ABC+−D*E
+	↑(+	ABC+−D*E
F	↑(+	ABC+−D*EF
)	↑~~(+)~~	ABC+−D*EF+
—	↑	ABC+−D*EF+↑

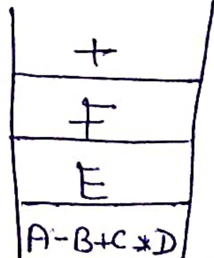
Postfix → ABC+−D*EF+↑

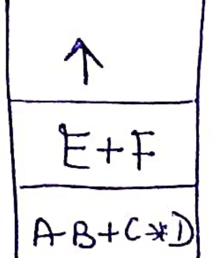
Expression tree



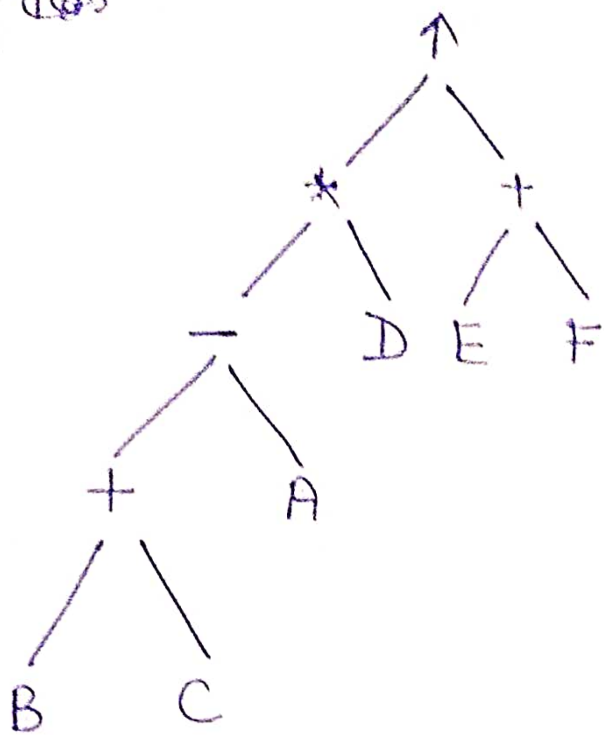








Pos



(*) Three address Code :-

It is an intermediate code. It is used by optimizing compiler.

In three address code the given expression is broken down into several separate instructions. These instruction can easily translate into assembly language.

Imp

Note ⇒ Each three address Code instruction have atmost three operand. It is the combination of assignment and a binary operator.

The example of three address code is

a:=(−c*b)+(− c*d)

Symbol	Stack	Postfix
a (	(	a
−	(−	
c	(−	c
*	(− *	c
b	(− *	b

)	(-*)	b-*
+	+	b-*
(	+(	b-*
-	+(-	b-*
c	+(-	b-*c
*	+(-*	b-*c
d	+(-*	b-*cd
)	+(-*)	b-*cd-*
—	+	b-*cd-*+

$a := (-c * b) + (-c * d)$

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

] three address Code

t is used as register in the target program

Q=2 $b := (e + f) - (g + h)$

Ans⇒

$t_1 := e$
$t_2 := f + e$
$t_3 := g$
$t_4 := h + g$
$t_5 := t_2 - t_4$
$b := t_5$

Three address Code Representation:-

It is represented into two forms

(1) Quadruple

(2) Triple

1 ⇒ The Quadruple are the form of the three address code to representation the Quadruple have four field to implement the three address Code.

The field of Quadruple contains the name of the operator, the first source operand, the second source operand and the result respectively and result denotes with the result destination.

Operator
Source 1
Source 2
Destination

Ex:- $a := -c * d + a$

$t_1 := -c$

$t_2 := d + a$

$t_3 = t_1 * t_2$

$a := t_3$

Operator	Source 1	Source 2	Destination
) minus	c	—	t_1
+	d	a	t_2
*	t_1	t_2	t_3
:=	t_3	—	a

Q=2 $a := -e * f + g$

Ans⇒ $t_1 := -e$

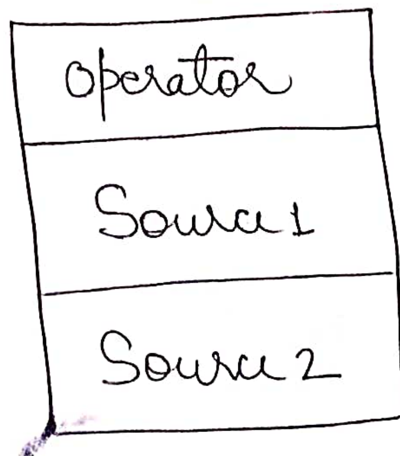
$t_2 := f + g$

$t_3 := t_1 * t_2$

$a := t_3$

Operator	Source 1	Source 2	Destination
−	e	—	t_1
+	f	g	t_2
*	t_1	t_2	t_3
:=	t_3	—	a

(2) Triple ⇒ The triples are the part of three address code, in this triple have three field to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.
In triples the result of respective sub-expression are denoted by the position of expression. Triple is equivalent to DAG while representing expression



$a := -b * c + d$

$\Rightarrow$

$t_1 := -b$ (0)

$t_2 := c + d$ (1)

$t_3 := t_1 * t_2$ (2)

$a := t_3$ (3)

Operator	Source 1	Source 2
U minus	b	—
+	c	d
*	0	1
:=	2	—

(*) Translation of assignment Statement :-

The translation of assignment statement are the part of Syntax directed translation in this the assignment statement is mainly deals with expressions. The expression can be type of real integer, array and records.

(*) Some Special Terms of translation of assignment statement :-

(1) The P returns the entry for id.name in the symbol table.

(2) The Emit is a function is used for included the three address code to the output file otherwise it will report an error.

(3) The new temp( ) is a function use to generate new temporary Variable.

(4) E.place hold the value of E.

Production Rule	Semantic action
S → id = E	{ P = look-up (id-name); if P ≠ nil then Emit (P = E.Place) else error; }
E → E1 + E2	{ E.Place = newtemp(); Emit (E.Place = E1.Place '+' E2 Place)}
E → E1 * E2	{ E.Place = new temp ( ); Emit (E.place = E1.place '*' E2.place)}
E → (E1)	{ E.place = E1.place }
E → id	{ P = look-up (id-name); if P ≠ nil then Emit (P = E.Place) else error ;}

This is the translation scheme of given grammer

S → id = E
E → E1 + E2
E → E1 * E2
E → (E)
E → id

Boolean Expression :-

The boolean expression have two primary purpose functions. They are used for computing the logical values. They are also used as conditional expression if–then–else.

Consider a grammer E → E OR E
E → E AND E
E → E NOT E
E → (E)
E → id1 relop id2
E → TRUE
E → FALSE

- The relop is denoted by greater then (>) and less then (<)
  relop ⟺ <, >, <, >.
- The AND and OR are left associated NOT has higher precedence then AND and OR.

(1) E → E OR E production rule and their semantic action is { E.place = new temp();
Emit (E.place ':=' E.place 'OR' E.place }

(2) E → E AND E production rule and their semantic action is { E.place = new temp();
Emit (E.place ':=' E.place 'AND' E.place }

(3) E → NOT E and their semantic action is
{ E.place = new temp();
Emit(E.place ':=' 'NOT' E.place }

(4) E → NOR E and their semantic action is
{ E.place = new temp();

(5) E → E and their semantic action is
{E.place = E1.place}

(6) E → id1 relop id2 production rule and their semantic action is
{ E.place = newtemp();
Emit('If' id1.place relop. id2
place 'goto'
next start +3);
Emit(E.place:='0')
Emit('goto' next start +2)
Emit('E.place':=' '1')
}

(7) E → TRUE production rule and their semantic action is
{E.place':=' new temp ();
Emit(E.place ':=' '1')
}

(8) E → FALSE production rule and their semantic action is
{E.place':=' new temp ();
Emit (E.place':=''0')
}

There are two important points for boolean expression :-

(1) The emit function is used to generate the three address code and the new temp() function is used to generate the temporary variable.

(2) The E → id relop id contains the next-state and it given the index of next three address code statement in the output sequence.

(*) Statement that alter the flow of control :-

The goto statement alter the flow of control if we implement goto statement then we need to define LABEL for a statement. A production can be added for this purpose

S → LABEL : S
LABEL → id

In this production semantic action is attached to record the label and its value in the symbol table

(*) Following Grammar used to incorporate :-

S → If E then S
S → If E then S else S
S → while E do S
S → begin L end
S → A
L → L ; S
L → S

Here S is a statement, L is a statement list.
A is an assignment statement
E → E is a boolean valued expression

(*) Translation Scheme for statement that alters flow of control:-

- We introduce the marker non terminal 'M' as in case of grammar for boolean expression.
- This M is put before statement in both if-then else in case of while do. We need to put 'M' before E. As we need to come back to it after executing S.
- In case of if then else if we evaluate E to be there, first S will be executed.
- After this we should ensure of second S the code after the if then else will be executed then we place another non-terminal marker N after first S.

case 0_2 : S_2

(*) Grammer follow the scheme:-

S → if E then M S

S → if E then MS else MS

S → while M E do MS

S → begin L end

S → A

L → L ; MS

ML → S

M → E

N → E

(*) Postfix translation:-

In a production A → α, the translation rule of A, CODE consist of the concatination of the CODE translation of the non terminal is α in the same order as the non terminal appear in α.

(*) Imp Note ⇒ Production to be factored to achive postfix form.

(*) Postfix translation of while Statement :-

S → while M_1 E do M_2 S1

Can be factored as

S → S_1

C → W E do

W → While

⇒ A suitable transition scheme would be

Production rule	Semantic action
W → while	W, QUAD = NEXT QUAD
C → W E do	C WEdo
S → C S_1	BACKPATCH (S1, NEXT: (QUAD) S.NEXT = (FALSE.

Chitra

Date ___/___/___

1) Array Reference in arthmatic Expression :-

Elements of array can be accessed quickly if the elements are stored in a block of consecutive location. array can be one dimentional and two dimentional.

For 1D array :-

A: array [low ---- high] of the ith element is at :

Base + (i - low) * width → i * width + (base - low * width)

In multidimentional array the two Dimentional array based on rows and coloumn major :-

Raw :- a[1,1], a[1,2], a[1,3], a[2,1], a[2,2], a[2,3]

Coloumn :- a[1,1], a[2,1], a[1,2], a[2,2], a[1,3], a[2,3]

The formula of 2D array or multidimentional array

Base + ((i1 - low1) * (high2 - low2 + 1) + i2 - low2) * width

(*) Translation Scheme for array elements :-
Limit(array, j) return nj = high j - low j + 1 place:
The temporary Variables offset : offset from the base, null if not an array reference.

(*) Procedures Calls :-
Procedure is an important and frequently use programming contruct for a compiler. It is used to generate good code for procedure calls and returns.

(*) Calling Sequence :-
The translation for a call includes a sequence of actions taken on entry and exit from each procedure.

(*) Actions take place in calling sequence:-

(1) When a procedure call occurs then space is allocated for activation record.
(2) Evaluate the argument of the called procedure.
(3) Establish the environment pointers to enable called process to access data in enclosing blocks.
(4) Save the state of the calling procedure so that it can be resume execution after call.
(5) Also save the return address.
return address ⇒ It is the address of the location to which the called routine must transfer after it is finished.
(6) Finally generate a jump to the beginning of the code for the called procedure.

(*) Let us consider a grammar for a simple procedure call statement.

S → Call id (E list)
E list → Elist , E
E list → E

(*) A suitable transition scheme for procedure:-

Production rule	Semantic action
S → Call id (E list)	For each item p on QUEUE do GEN (param p) GEN (

"Failure comes only when we forget our ideals and objectives and principles." - Jawaharlal Nehru

Elist → Elist, E   append E.PLACE to the end of QUEUE.

Elist → E   initialize QUEUE to contain only E.PLACE

QUEUE is used to store the list of parameters in the procedure call.

Production

D → integer id   Enter (id. PALACE integer) D.ATTR = integer

D → real .id   ENTER (id. Place . real) D.ATTR = real

DA → D1.id   Enter (id. PLACE D1 ATTR) D.ATTR = D1.ATTR

The translation scheme for declared production

* Enter is used to make entry into symbol table and 'attr' is used to trace the datatype

(*) Case statement :-

Switch case statement is available in a variety of lang the syntax of case statement is as follows

Switch
begin
Case V1 : S1
Case V2 : S2

;

Case V_{N-1} , S_{n-1}
default : S_n
end —

(*) Translation Scheme for this below:-
Code to evaluate $E \rightarrow T$
goto TEST
L_1: Code for S_1
goto NEXT
L_2: Code for S_2
goto NEXT
Soon
L_{n-1}: Code for S_{n-1}
go to NEXT
L_n : Code for S_n
go to NEXT
TEST : if $T = V_1$ go to L_1
if $T = V_2$ go to L_2
,
,
if $T = V_{n-1}$ go to L_{n-1}

When switch keyword is seeing there new temporary T and a new LABELS [TEST & NEXT] are generated. when a CASE keyword occurs, then for each CASE keyword a new LABEL L, Li is creates and entered in to symbol table.
The value Di for each case constant and pointer. This symbol table entry are placed on stack.

Unit = 4

(*) Symbol Table :-
It is an important data structure used in a compiler. Symbol table is used to store the information about the occurence of various entities such as object classes variables name, interface, function name. It is used by both the analysis and the synthesis phases.
The symbol table used for the following purpose :-
1) It is used to store the name of all entities in a structured form at one place.
2) It is used to verify if a variable has been declared.
3) It is used to determine the scope of a name.
4) It is used to implement type of checking by verifying assignments and expression in the source code are sementically correct.
A symbol table can either be linear or hash table. Using the following formate it maintains the entry for each name.
< Symbol name, type, attribute >
Ex ⇒ Static int salary
Then, it stores an entry in the following formate
< Salary, int, static >
The clause attribute contains the entry related to name.
(*) Implementation of Symbol Table :-
The symbol table can be implemented in the unordered list if the compiler used to handle the small amount of data.
A symbol table can be implemented in one of the following techniques :-
(a) Linear
(b) Hash table
(c) Binary Search Tree.

Symbol table ...

(ii) Operations in symbol Table :-
The symbol table provides the following operations:-

a) Insert ():-
- Insert ( ) operation is more frequently used in the analysis phase when the token are identify and names are stored.
- Insert ( ) operation is used to insert the operation in the symbol table like the unique name occuring in the source code.
- In the source code the attribute for a symbol is the information associated with that symbol.
- The information contain the state, value, type and scope about the symbol.
- The insert operation function takes the symbol and its value in the form of argument

Ex ⇒ int x;
Should be processed by the compiler as insert (X, int)

b) Look up operation():-
In the symbol Table look up( ) operation is used to search a name It is used to determine :-
- The existance of symbol in the table.
- The declaration of the symbol before it is used.
- Check whether the name is used in the scope.
- Initialization of the symbol
- Checking whether the name is declared multiple time.

The basic formation of look up( ) operation is as follows:-
look up (Symbol)

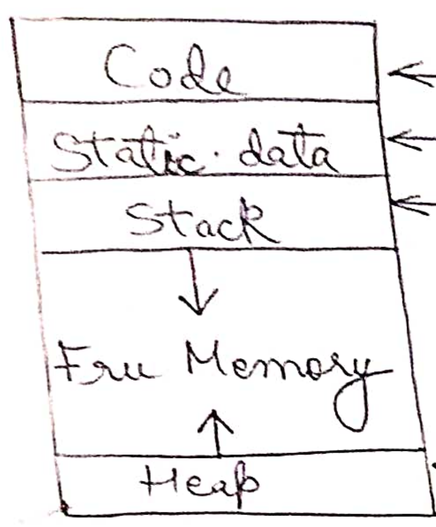
Note ⇒ This formate is varies according to the programming language.

(*) Storage Organisation :-

When the target programme executs then it run in it own logical address space in which the value of each programme has a location.

The logical address space is shared among the compiler, OS and target machine for management and organisation. The OS is used to map the logical address into physical address which is usually spread throughout the memory.

(*) Sub-division of Run-time Memory :-



← Memory location for code are determine at compile time.

← Location of Static Data can also be determine at compile time.

← Data object allocate at run-time (activation record)

← Other dynamically located data objects at run time.

- Run-time storage comes into blocks where a byte is use to show the smallest unit of addresable memory using the four bytes a machineword can form object of multibyte is stored in consecutive byte and gives the first byte address.
- Run-time storage can be sub-divided to hold the different components of an executing program.

(a) Generate Executable Code
(b) Static data objects
(c) Dynamic data objects like Heap
(d) Automatic data object like stack

(*) Imp Activation Record :-

- Control Stack is a run-time stack which is used to keep track of the live procedure activation. It is used to find out the procedure whose execution have not been completed

- When it is called (activation starts) then the procedure name will be push to be on the stack and when it return (activation ends) then it will poped.
- Activation record is use to manage the information needed by a single execution of procedure.
- An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

Return Value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

(a) Actual Paramiter ⇒ It is used by calling procedure to supply paramiters to the called procedure.
(b) Control Link ⇒ It is a point to activation record to the caller.
(c) Access link ⇒ It is used to refer non-local data held in the other activation record. ~~save machine sta~~
(d) Save Machine Status ⇒ It hold the information about status of machine before the procedure is called.
(e) Local Data ⇒ It holds the data i.e; local to the execution of the procedure.
(f) Temporaries ⇒ It store the value that arises in the evalution of an expression.

Storage Allocation :-
There are different ways to allocate memory are as fo...
(i) Static Storage allocation
(ii) Stack Storage allocation
(iii) Heap storage allocation

(i) Static Storage allocation :-
(a) In Static allocation names are bound to storage location.
(b) If memory is created at compile time then the memory will be created in static area only once.
(c) Static allocation support the dynamic data-structure that means memory is created only at compile time and allocate after programme completion.
(d) The drawback of SSA is the size of portion data objects should be known at compile time.
(e) Another drawback is restriction of the recursion procedure

(ii) Stack Storage Allocation :-
(a) In static storage allocation storage is organised as a stack.
(b) An activation record pushed into stack when activation begins and it is poped when activation is end.
(c) Activation record contains the local data so they they are bound to fresh storage in each activation record. The value of local data is deleted, when the activation record is end.
(d) It work on the basis of LIFO and this allocation support the recursion process

(iii) Heap Storage Allocation :-
a) Heap allocation is the most flexible allocation Scheme.

(b) Allocation & deallocation of memory can be done at any time and at any place depending upon the user requirement.

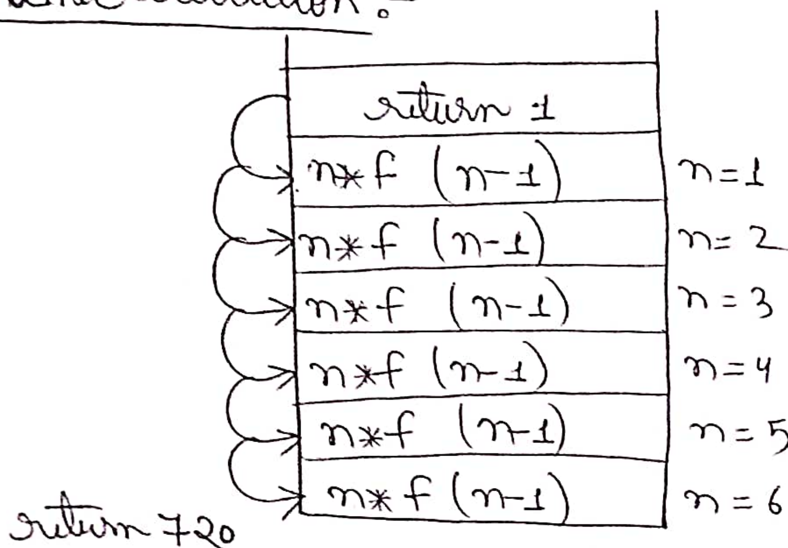
(c) Heap allocation is used to allocate memory to the variables dynamically and when the variable are no more used then claim it back.

(d) Heap storage allocation also support the recursion process.

```
fact (int n)
{ if (n< =1)
      return 1;
  else
      return (n* fact (n-1));
}
```

fact (6) = 720

(*) Dynamic allocation :-



(*) Lexical error :-

During the Lexical Analysis phase this type of error can be detected.

Lexical Error is a sequence of characters that does not match the pattern of any token. Lexical face error is found during the execution of the program.

exical phase error can be as follows:-
Spelling Error
Exceeding length of identifier or numeric constants.
Appearance of illegal ~~chara~~ characters.
) To remove the character that should be present.
) To replace the character within incorrect ~~word~~ manner.
Transposition of two characters.

) Syntax Error:-

During the syntax Analysis phase. This type of error appears,
Syntax error is found during the execution of the program.
Syntax error can be:-
) Error in structure
) Missing operators
When an invalid calculation enters into calculator then a syntax error can also occur. This can be caused by entering several decimal points in one no. or by opening brackets without closing them.

Ex ⇒ using "=" when "==" is needed

```
if (number=200)
Count << "number is equal to 20");
  else
Count << "number is not equal to 200");
```

*) Symantic error:-

During the semantic analyzer phase. These type of error can be detected at compile time. Most of the compile time error are related scope or variable.
Ex ⇒ Undeclared or Multiple declared identifiers. This type of Mismatch is another compile time error.

(*) Some common error can be detected :-

1) Incompatible type of operands
2) Undeclared Variable
3) Not Matching of actual argument with formal argument

Ex (1) ⇒
```
int i;
void f (int m)
{
    m=t;
}
```

Ex (2) ⇒ Time incompatibility
```
int a = hello;
```

Ex (3) ⇒ Error in expressions
```
String = "----";
int a = 5 - S;
```

Unit-5

(*) Code Generator :-

Code Generator is use to produce the target code for three address statement. It uses register to store the operands of the three address code statement.

(*) Register and address descriptor :-

. Contain the track is currently in each register. The register descriptors show that all the register are initially empty.

. An address descriptor is use to store the location where current value of the name can be found at run time.

(*) Algorithm of Code Generation :-

There are few given steps of Code Generation Algo.

a) Invoke a function Getreg to findout the location (L) where the result of Computation b op C should be stored.

b) Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y'. If the value of y is not already in L then generate the instruction Mov y', L to place a copy of y in L.

c) Generate the instruction OP Z', L where Z' is used to show the current location of z. If z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update the descriptor and remove x from all the descriptor.

d) If the current value of Y or Z have no next of use or not live on exit from block of the register, then the register descriptor to indicate that after execution of X := Y OP Z those register will no longer contain Y or Z

Q=1. Draw DAG -

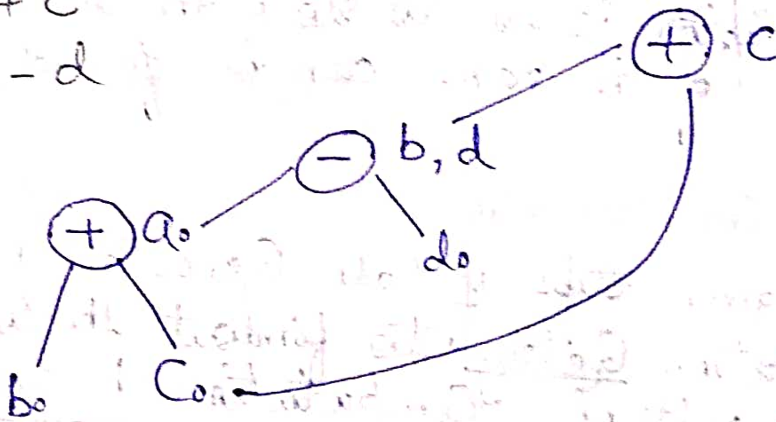
a = b + c

b = a - d

c = b + c

d = a - d

Ans ⇒



Q=2 a = b + c

b = b - c

c = c + d

e = b + c

Ans ⇒