Compiler Design.

D 205

Compiler → used to find error in programs

converts
- → high-level language
- → Assembly / middle-level language
- → low-level language

High → low level language is not a single step process

I Introduction
  Lexical Analysis

II. Syntax Analysis

III. SDT
  Intermediate Code Representation

IV Code generation

V Code Optimization

Course objectives.

- To understand the various phases in the design of a compiler.
- To ~~design~~ understand the design of top-down and bottom-up parsers
- To understand syntax directed translation schemes
- To introduce lex and yacc tools
- To learn and develop algorithms, to generate code for a target machine

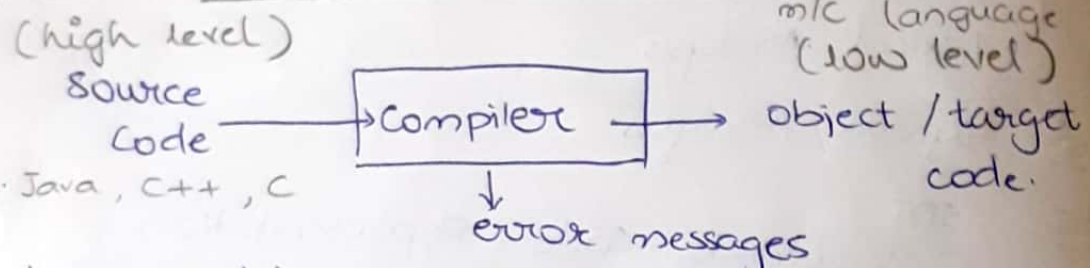
Course Outcomes:-

- Ability to design, develop and implement a compiler for any language
- Able to use, lex and yacc tools for developing a scanner & a parser

- Able to design and implement LL and LR parsers
- Able to design algorithms to perform code optimization in order to improve the performance of a program, in terms of space & time complexity
- Ability to design algorithms to generate machine code

(2m) **Language translators**

Compiler.



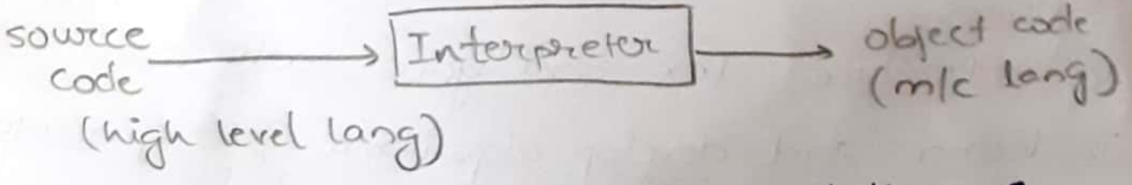
compiler contains a software programs that should satisfy the following properties.

Properties :-

- → bug free
- → proper machine code
- → portable
- → consists an optimized code
- → execution fastly

- It converts the source code into an intermediate code (set of instruction)
- debugging is difficult
- execution is fast

Interpreter :



- An interpreter transforms | translating a source code to object code line by line
- Execution is slow but debugging is easy
- Languages ex:- Perl
  Python
  Ruby
  Matlab

Compiler :- A compiler is a program that translates or converts a program written in high level language into equivalent program in low level language. And while doing so, it also produces error messages

(2M)

Properties :- (which are used in building a compiler)

- The program itself should be bug free
- It should translate proper machine code
- Execution must be fast
- should be ~~pr~~ portable
- should contain consistency in terms of optimized code
- It should provide error messages with proper line numbers

Interpreter :-

→ It is a programs that translates or converts a program written in high level language to a equivalent program written in machine level language but line by line conversion takes place.

→ It produces error messages after translating every line and if an error occurs, unless it is cleared, the translation / conversion of the next line stops

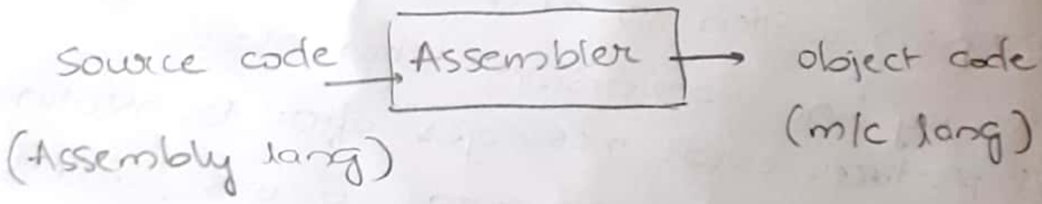
Differences b/w compiler & interpreter

Compiler	Interpreter
• Compiler converts the entire source code at a time	• ~~compile~~ Interpreter converts line by line
• Analysing the entire Source code at a time	• Analyzing line by line code is easy but

is difficult but the execution is very fast	execution is slow
• Debugging is hard to implement	• Debugging is easy
• Compiler can convert source code into intermediate form sometimes	• Intrepreter doesnt do it.
• Ex : C, C++, Java	• Ex :- Perl, Python, ruby, matlab.

Assembler :-

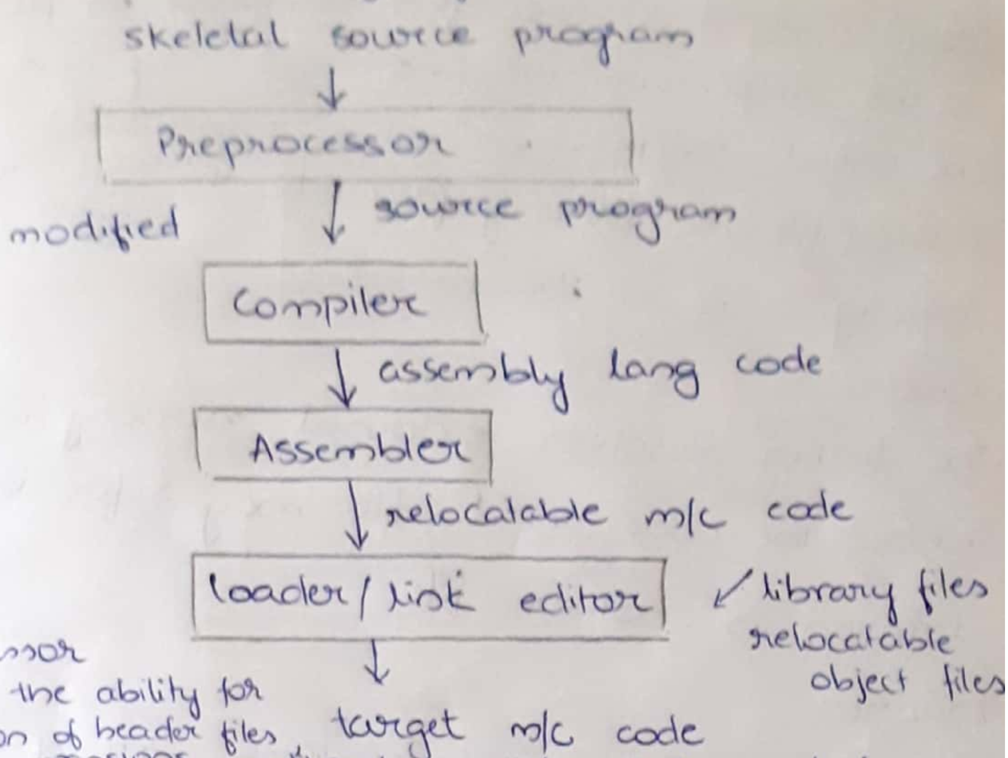
It is a software program which converts program written in assembly level language which contains symbolic instructions into equivalent machine language



- Language translators are software programs, that convert programs written in high level language / assembly language into equivalent machine language code

Language processing system :-

Language processing system :-



* i.e. preprocessor provides the ability for inclusion of header files, macro expansions, conditional compilation & line control.

Preprocessor :-

- A source program may be divided into modules stored in separate files, the task of collecting the source program is sometimes included to a separate program called as a preprocessor
- * It is a tool that produces input for compilers therefore it deals with micro processing augmentation, file inclusion, language extension etc.
- This modified source program is then fed to a compiler.

Compiler :-

- The compiler will produces an assembly lang program as its output because assembly language is easier to produce and easier to debug

Assembler :

The assembly lang is then processed by a program called an assembler that produces relocatable machine code as its output

- Large programs are often compiled in pieces so the relocatable machine code may have to be linked together with other relocatable object files & library files into the code that actually runs on the machines

Linker :-

- The linker resolves external memory addresses where the code in 1 file may refer to a location in another file.

Loader :-

- The loader then puts together, all of the executable object files into memory for execution, it also calculates the total size of program & create memory place / space for it.
- Registers are used for fast access.

Note :-

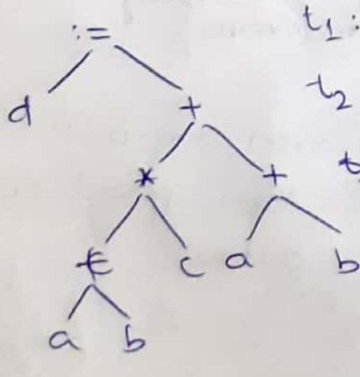
It initializes various registers to initiate execution

Structure of a compiler

(or)

Phases of compiler / compilation

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



$t_1 := a+b$

$t_2 := t_1 * c$

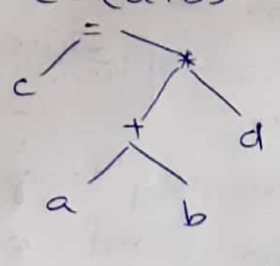
~~$t_3 = t_2 + t_3$~~

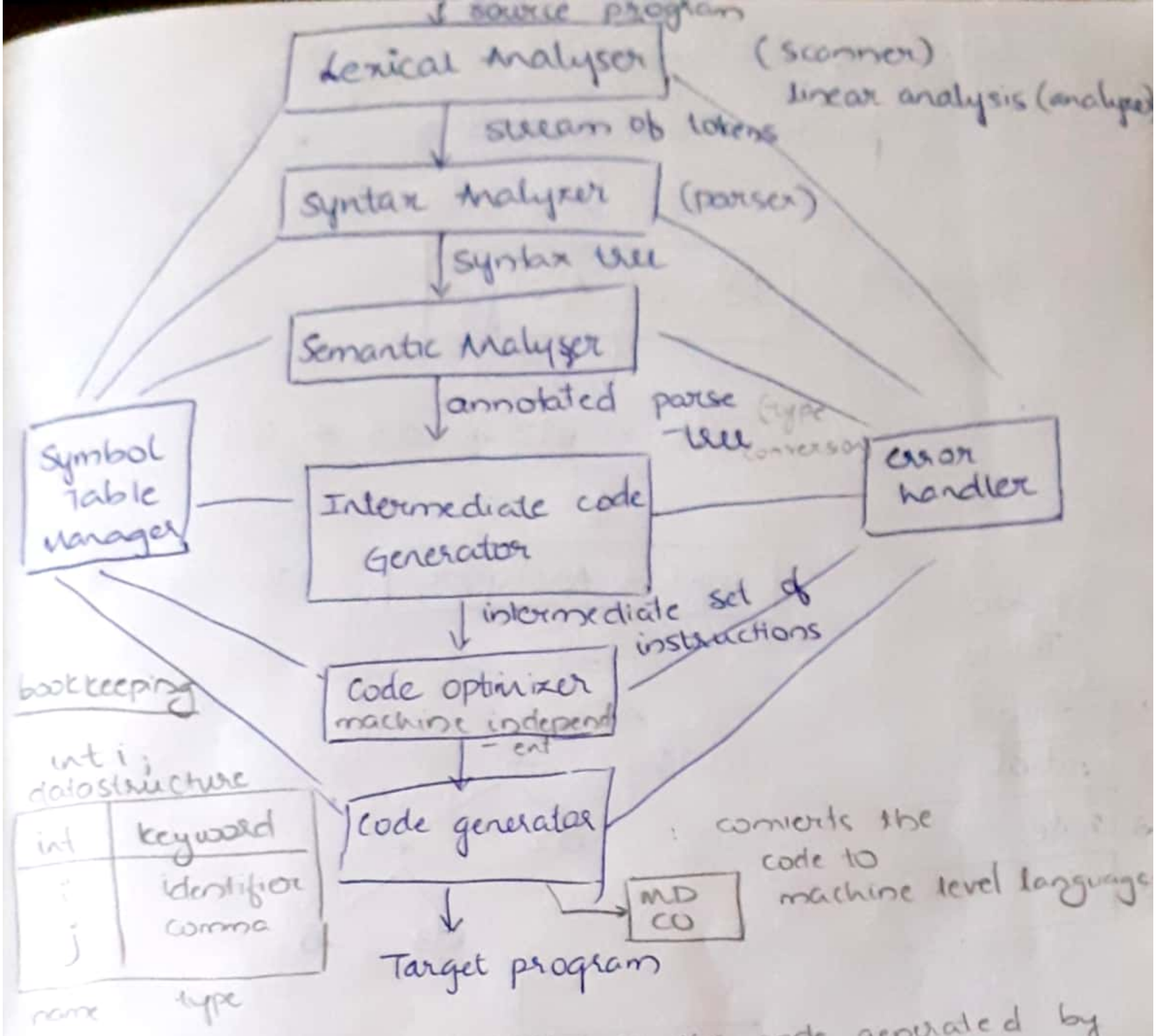
$t_3 = a+b$

$t_4 := t_2 + t_3$

$d := t_4$

2) $c = (a+b) * d$





int	keyword
i	identifier
j	comma

: converts the code to machine level language

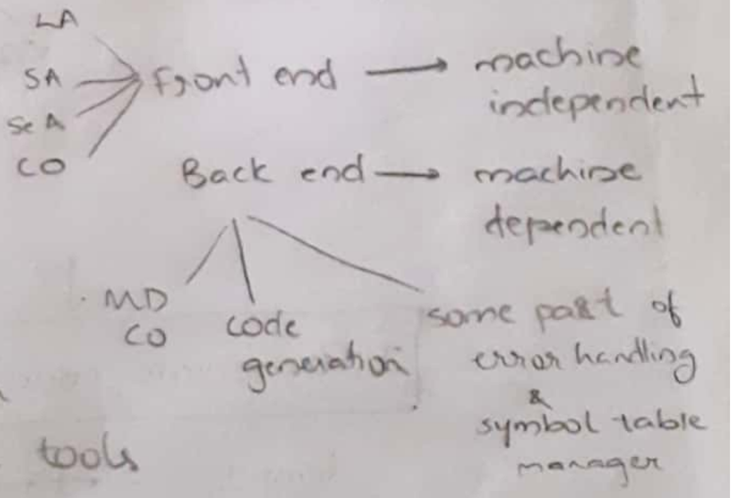
5.) Code optimization :- optimizes the code generated by the intermediate code generator.

$t_1 = a+b;$

$t_2 := t_1 * c;$

$d := t_1 + t_2$

MD machine dependent code optimization



1) LEX : It is a tool used by lexical analyzer to generate tools

Translation of statement :-

position := initial + rate * 60

↓

Scanner **Lexical Analyser**

↓

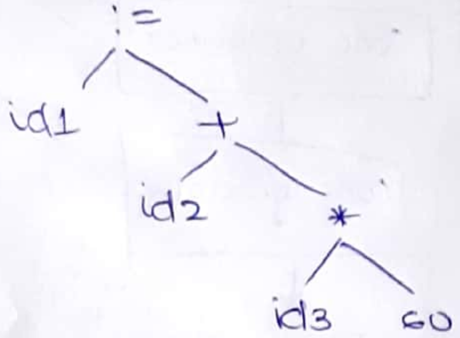
id1 := id2 + id3 * 60

↓

Syntax Analyser : Parser

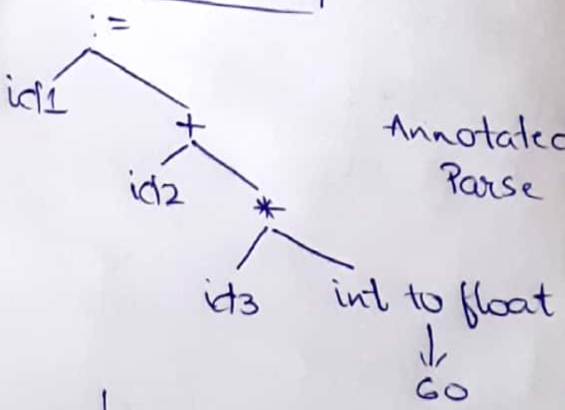
Symbol Table

1	position	
2	initial	
3	rate	
4		



↓

Semantic Analyser



Annotated Parse tree.

↓

Intermediate code Generator

temp1 := ~~id3 *~~ (int to float) 60

temp2 := id3 * temp1

temp3 := id2 + temp2

id1 := temp3

↓

Code Optimizer

temp1 := id3 * 60.0

id1 := id2 + temp1

↓

Code Generator

↓

float

LDF R_2 , id3 ⟶ load id3 into R_2

MULF R_2 , R_2 , # 60.0 → multiply R_2 & 60.0
and store in R_2

LDF R_1 , id2 ⟶ load id2 into R

ADDF R_1 , R_1 , R_2 → add R_1 , R_2 and
store in R_1

STF id1 , R_1 ⟶ store R_1 into id1

Lexical analyzer:-

- It is the 1st phase of compilation
- It is also called as scanner because it work as text scanner
- This phase scans the source code as stream of characters and converts into meaningful lexemes.
- Lexical analyzer represents these lexemes into the form of tokens as

<tokenname, attribute name>

- Tokens can be keywords, identifiers, punctuation symbols, operators etc.

Syntax analyser

- The next phase in compiler is syntax analyser
- It is also called parser.

- It takes the token produced by lexical analyzer as i/p and generates parse stream (syntax stream)
- In this phase token arrangements are checked against the source code grammar i.e, the parser checks the expression made by token are syntactically correct.

Semantic analyzer :-

- It checks whether the parse tree constructed follows rules of language

ex:- Assignment of value is b/w computable datatypes and adding string to an ~~a~~ integer

- It checks if the given statement is syntactically correct or not.
- Also the semantic analyzer keeps the track of identifiers, their types & expressions and whether the identifiers are declared before use or not.
- It produces an annotated parse tree as output.

Intermediate code generator :-

- After the semantic analysis, the compiler generates intermediate code of source code for target machine.
- It represents
- It is in b/w high level language & middle level language.

- This intermediate code should be generated in such a way that it makes it easier to translate into the machine code
- We have 3 different way for representing intermediate code

Code optimizer :-

- It is an optional phase of compilation & it does the optimization of the intermediate code
- This phase does the optimization by assuming as something that removes unnecessary code lines and arranges the sequence of statements in order to speed up the program execution ~~whether~~ without wasting

Code generator :-

- In this phase, the code generator takes the intermediate optimized representation of the intermediate code & maps it to target machine language.
- The code generator translates the intermediate code into a sequence of relocatable machine code.
- Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol table :-

- It is a datastructure maintained throughout all the phases of compiler.
- All the identifier names along with their types are stored here.
- The symbol table makes it easier for the compiler to quickly search the identifier

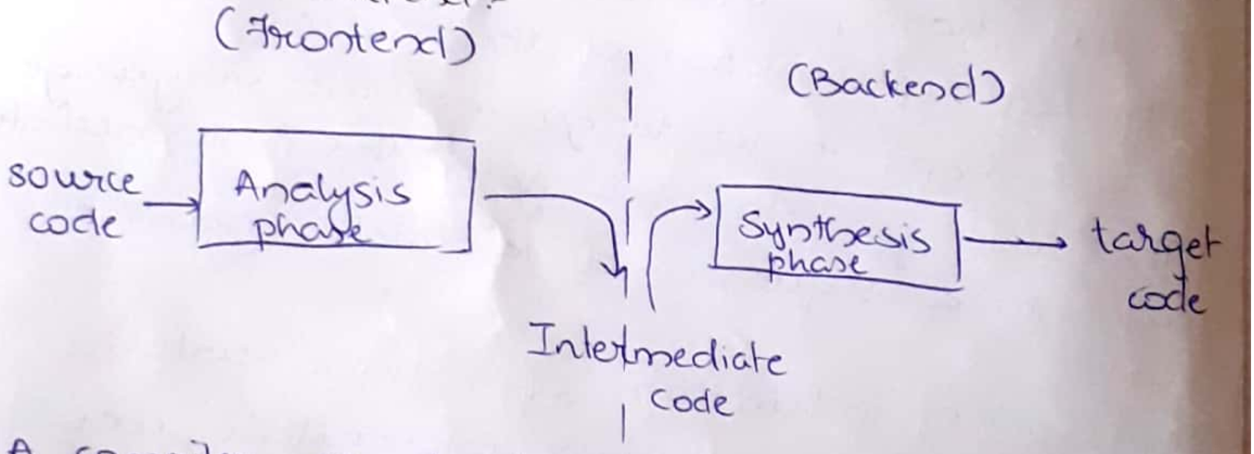
record & retrieve it.

- The symbol table is also used for scope management.

Error handler :-

- Error handler is invoked when an error in the source program is detected.
- It must warn the programmer by issuing a command and adjust the info being passed from phase to phase so that each phase can proceed.
- Both symbol table & error handler ~~are~~ routines interacts with all the phases of the compiler

Frontend & backend :-



A compiler can be broddly ~~is~~ divided into 2 categories based on the way they can be compiler

(i) Analysisa phase :- It is known as the front end of the compiler. It reads the source program & divides it into core parts and then checks for lexical, gramatical & syntax errors.

- The analysis phase generates the intermediate representation of source program which is fed as an input to synthesis phase
- It includes lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator & some part of code optimization.
- It also includes creation of symbol-table, error handler that goes along with each of these phases.

(ii) Synthesis phase :-

- It is known as back end of the compiler, it generates the target program with the help of intermediate code generation & symbol table

Cross compiler :-

A compiler that runs on platform A is capable of generating an executable code for platform B is called cross compiler.

Source to source compiler :-

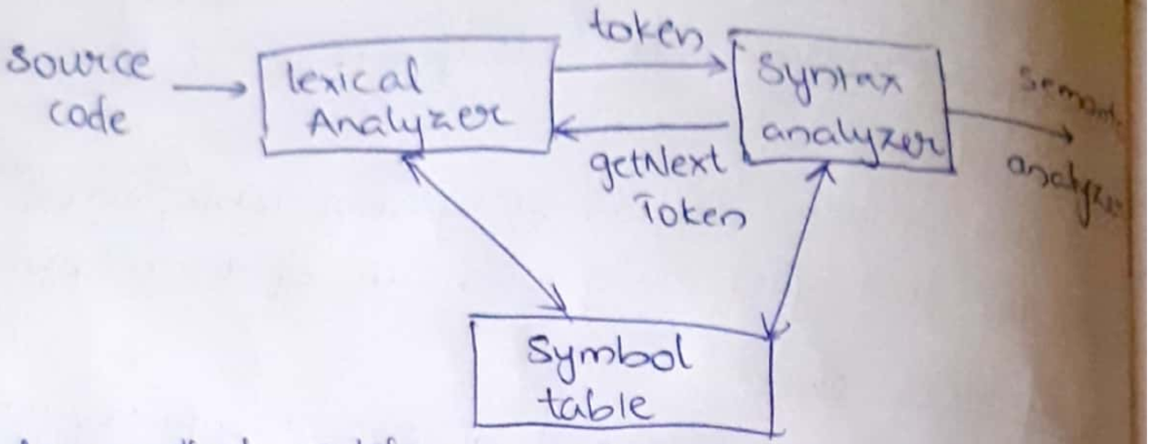
A compiler that takes the source code of 1 programming language & translating it into source code of another programming language is called source to source compiler.

Pass & phase :-

A pass refers to traversal of a compiler through an entire program.

Phase : A phase of compiler is a distinguishable stage, which takes input from previous stage, processes & yields output that can be used as input for the next stage. A parse can have more than 1 phase.

lexical Analysis:-



(A lexeme that matches a pattern is a token).

token :- a smallest character which cannot be further divided.

lexeme :- a set of characters that forms a pattern /meaning.

pattern :- A rule describing a lexeme

- The secondary task of lexical analysis phase,

→ 1 task is stripping out of comments & white spaces in the form of blank tab & new line characters

→ 2nd task is correlating or associating error messages from the compiler with the source program. i.e, for eg:- the lexical analyzer may keep the track of new line characters seen, so that the line no. can be associated with any error message

→ lexical analyzer is the cascade of 2 methods
  (i) Scanning :- for doing simple task
  (ii) lexical analysis :-

→ A lexical analyzer deals with the terms, token , lexeme, pattern with specific meanings

token:- A sequence of characters having collective meaning.

lexeme:- A sequence of characters in a source pgm that is matched with a pattern for a token.

pattern:- A pattern is a rule describing a set of lexemes that can represent a particular stmt in a source program

Token	Sample lexeme	Pattern
relop	<, <= , ==, < >, >, >=	< or <= or == or < > or > or >=
num	3.14, 0.142	any numeric const
literal	"core dumped"	any characters between " and ".
id	pi, d3	letter followed by letter (or) digit any no. of times letter (letter/digit)*

eg:- **Fortran :-**

$E = mc^2$

E = m * c * * 2

<id, pointer to symbol table entry for E >

<assignop , >

<id, pointer to symbol table entry for m>

<mult-op , >

<id, pointer to symbol table entry for c>

<exp-op, >

<num, integer value 2>

Eg: fi (a== f(x))

→ This syntax error is not dentified by the lexical analyzer but is done by the rest of the phases of compilation.

→ such type of syntax errors might be covered by following 4 techniques :

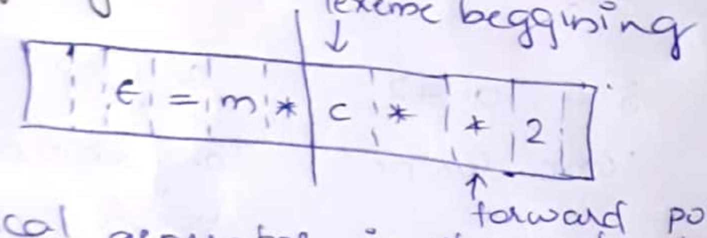
(i) transposing of 2 adjacent characters

(ii) inserting a missing character

(iii) Deleting an extra character.

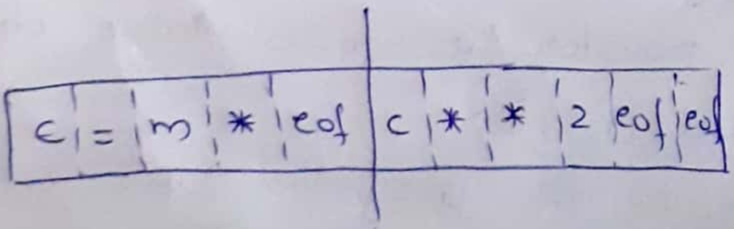
(iv) Replacing an incorrect character or string by the correct character/string.

Input buffering :-



- The lexical generator, in ~~the~~ order to scan the characters from i/p buffers maintains 2 pointers
  (i) lexeme beginning
  (ii) forward pointer
- Forward pointer is implemented until a lexeme is found
- End of the source program will be mentioned ~~to~~ by EOF marker.

Sentinals :-



- code that represents the moving of forward pointer

```
if forward at the end of the first half then begi
        reload second half ;
        forward := forward+1;
end
else if forward at the end of second half then
                                          begin
      reload first half ;
      move forward pointer to beginning of
                                  first half ;
 end
 else
        forward := forward + 1 ;
 ~~end~~.
```

- code for the sentinals :

```
forward := forward +1;
if forward↑ := eof then
     if forward at the end of first half then
                                          begin
          reload second half;
          forward := forward+1;
     end
     else if forward at the end of second half
                                   then begin
          reload first half;
          move forward pointer to beginning
                                  of first half
     end
     else
          lexicalanalyzer stops
end
```

Specification of tokens.

- Tokens can be described with the help of regular expression

Alphabet : The term alphabet or character class denotes any finite set of symbols.

Symbols can be letters & characters.

String :- A string over some alphabet is a finite sequence of symbols drawn from that alphabet.

- It is also called as sentence/word.
- length of the string is represented by |s|.
- Empty string is denoted by ϵ is a special string of length 0 (zero).

Prefix & suffix of the string :-

Eg :- CSE 3
prefix : CS
suffix : E3
substring : SE

- A string not containing ϵ and it isn't a same string, then it is called a proper prefix/suffix.
  eg :- CS, E3
- If substring is not continues, then it is subset.
  eg :- CSE
  EE → subset of CSE3

Language :-

It is a set of strings performed on alphabets if L & M are the languages, following operations can be performed.

Regular expression :- • Σ is symbol of alphabet (set of symbols)

• Regular expression is defined over an alphabet 'Σ' in the following way where Σ & the sets are defined as follows.

(1) ϕ is a R.E. and denotes an empty set

(2) ϵ is a regular expression and denotes $\{\epsilon\}$ i.e. a set containing an empty string

(3) If 'a' is a symbol in Σ then 'a' is a regular expression that denotes $\{a\}$ i.e, the set containing string 'a'.

(4) If r and s are regular expressions denoting the languages $L(r)$ and $L(s)$ then $(r)|(s)$ is a R.E. denoting $L(r) \cup L(S)$. similarly $r.s$ is a R.E., denoting $L(r).L(S)$. Similarly $(r)^*$ is a R.E. denoting $L(r)^*$ is a r.e

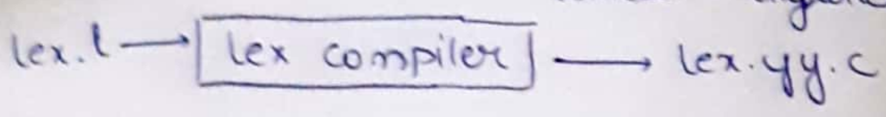
Axioms / properties of R.E :

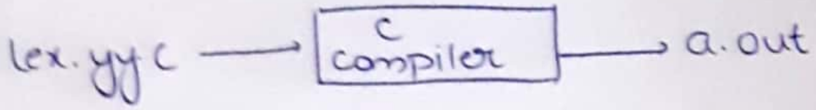
Axioms	description
1) $r\|s = s\|r$	$\|$ is commutative
2) $r(s\|t) = (r\|s)t$	$\|$ is associative
3) $rs = sr$	concatenation is commutative.
4) $r(st) = (rs)t$	concatenation is associative
5) $r(s/t) = rs\|rt$ $(s\|t)r = (sr)(tr)$	concatenation distributive over r
6) $\epsilon r = r$ $r\epsilon = r$	where ϵ is an a identity element for concatenation
7) $r^{**} = r^*$	* is idempotent

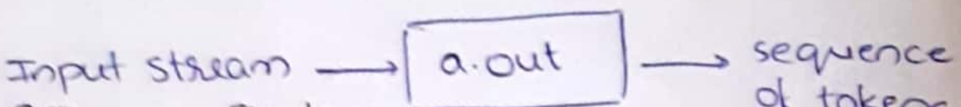
→ unary * is having highest precedence

→ concatenation – 2nd } associativity L to R

LEX tool :- It is an automated tool used by the lexical analyzer to generate the tokens







lex program syntax:

- declarations

    %.%.

- translation rules

    %. %.

    auxilliary procedures

    P_1 {action1}

    P_2 {action2}

    ⋮

    P_n {action n}

write a lex program
To identify the ~~keywords~~ identifiers

```
%{ #include <stdio.h>
%}
digit [0-9]
letter [A-Z a-z_]
%.%.
{letter}({letter/digit})*
{printf("%s is an identifier", yytext);}
%%
main()
{
    yylex();
}
```

o/p :- cse is an identifier.

Q. write a lex program to recognize a keyword

```
%{ #include <stdio.h>
%}
keyword  int| float | char | for | while / do
%%
{keyword} { printf ("%s is a keyword", yytext)
                                        ;}

%%
main ()
{
   yylex ();
}
```

Q write a lex program to recognize arithmetic operations, relational operators, logical & bitwise operators

```
%{# include <stdio.h>
%}
operators   *|+|-|/|<|<=|<>| > |>=/==|
              || / && | ! | \ | & | ^

%%
{operators} { printf ("%s is an operator",
                                  yytext) ; }

%%
main ()
{
   yylex ()
}
```

(or)

```
%{# include <stdio.h>
%}
 arithmetic  *|+|-|/
 relational  <|<=|<>|>|>=|==
  logical   |||&&|!
  bitwise   ||&|^
%%
{arithmetic} {printf("%s is an arithmetic operator",
                                      yytext);}
{relational}{ printf("%s is an relational operator",
                                       yytext);}
{logical}{ printf ("%s is a logical operator",
                                        yytext);}
{bitwise}{ printf(" %s is a bitwise operator",
                                         yytext);}
%%
main ()
{
  yylex ();
}
```

metacharacters :-

* 0-9 (0 or more)

\+ 1-9 (1 or more)

? 0/1

[ ] – character class

( ) – enclosing of RE in one format

. # . keyword is

`^`	to specify the beginning of a character string
`$`	to specify the end of a character string
`[^]`	to specify all string, not starting with (^T) a character.
`" "`	iterator, a string
`\`	esc character
`|`	or symbol (pipe)

yyerror() :- It is used to display the error msgs.

yyin():- It is variable used to store i/p source program

yyout():- It is used to store the o/p file.

yylen():- It is used to store length of the string

yywrap():- The function returns 2 values

If it returns zero(0). It continues the scanning

If it returns 1. It shows it reaches the end.

~~yyva~~

yylval():- This variable is used to store the next generated token.

Finite Automata:

$M = (Q, \Sigma, \delta, q_0, F)$

A finite automata can be represented by a 5 tuple notation

$M = (Q, \cancel{\epsilon}\,\Sigma, \delta, q_0, F)$

where,

Q = finite non-empty set of states

Σ = finite set of non-empty set of i/p. called as input alphabet

δ = is a transition fn which maps

$Q \times \Sigma \longrightarrow Q$

q_0 is the initial state

F is the final state where $F \subseteq Q$

General defination of FA :

An automata is defined as a system where energy materials and information are transformed and used for performing some function without direct participation of humans.

Non-deterministic finite automata

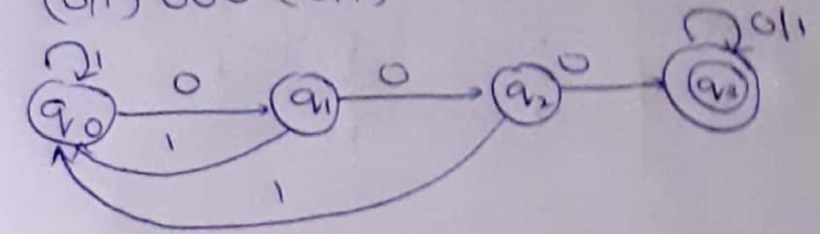
A DFA is which has ambiguity is called NFA.

A FA which allows ~~of~~ 0, 1 or more transitions from a state on the same i/p symbol is called NFA

Q. Draw a DFA to accept the strings of 0's & 1's having 3 consequtive 0's.

$(0/1)^* 000 (0/1)^*$

DFA :

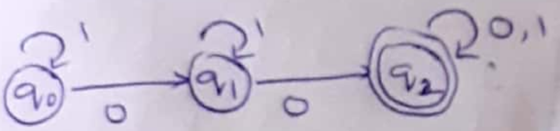


δ	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_3	q_0
→q_3		

Transition table

Q. Draw a DFA to accept the strings of 0's & 1's having atleast 2 zeroes.

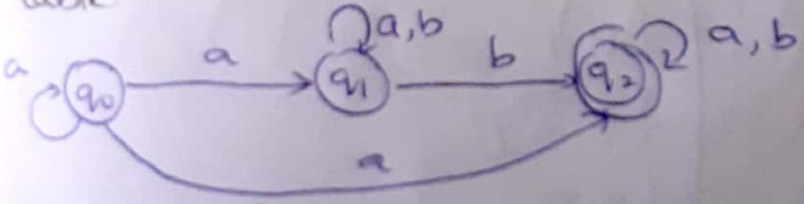
00 , $(0,1)^* 0 (0,1)^* 0 (0,1)^*$



1001100

δ	0	1
q_0	q_1	q_0
q_1	q_2	q_1
→q_2		

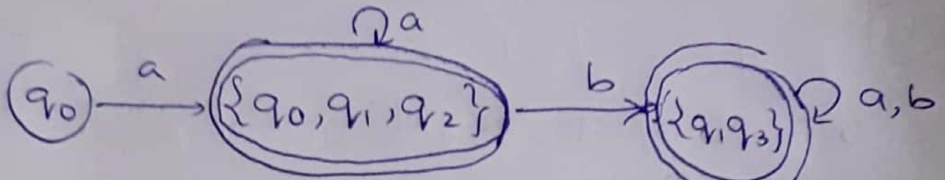
Q convert NFA to DFA and then draw transition table



δ	a	b
q_0	$\{q_0, q_1, q_2\}$	ϕ
q_1	q_1	$\{q_1, q_2\}$
q_2	q_2	q_2

δ	a	b
q_0	$\{q_0,q_1,q_2\}$	ϕ
q_1	q_1	$\{q_1,q_2\}$
q_2	q_2	q_2

δ	a	b
q_0	$\{q_0,q_1,q_2\}$	ϕ
$\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$,	$\{q_1,q_2\}$,
$\{q_1,q_2\}$	$\{q_1,q_2\}$	$\{q_1,q_2\}$



Q2) $M = (\{p,q,r,s\}, \{0,1\}, \delta, p, \{s\})$

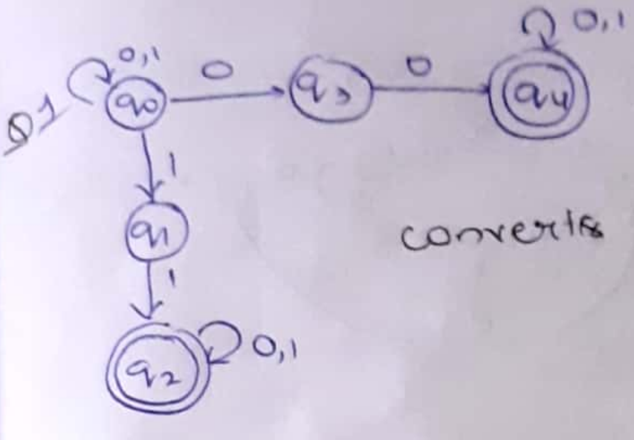
δ	0	1
p	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	ϕ
s	$\{s\}$	$\{s\}$

δ	0	1
p	$\{p,q\}$	$\{p\}$
$\{p,q\}$	$\{p,q,r\}$	$\{p,r\}$
$\{p,q,r\}$	$\{p,q,r,s\}$	$\{p,r\}$
$\{pr\}$	$\{p,q,s\}$	$\{p\}$
$\{p,q,r,s\}$	$\{p,q,r,s\}$	$\{p,r,s\}$
$\{p,q,s\}$	$\{p,q,s\}$	$\{p,r,s\}$

{p,$\overset{s}{q}$,r}	{p,q,s}	{p,s}
{p,s}	{p,q,s}	{p,s}

Draw DFS

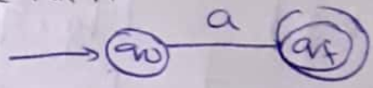
Q1



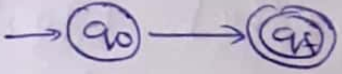
converts it to DFS.

R.E. to ε-NFA

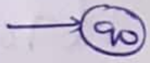
1.) $r = a$



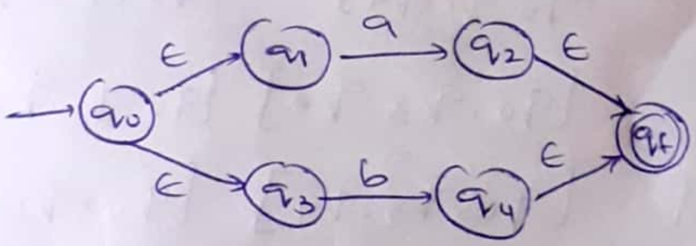
2.) $r = \phi$



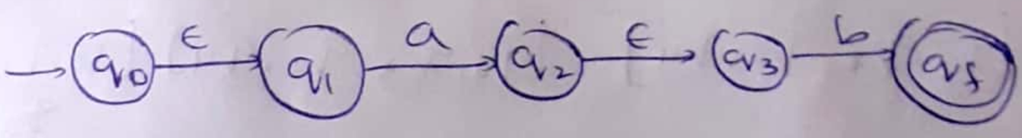
3.) $r = \epsilon$



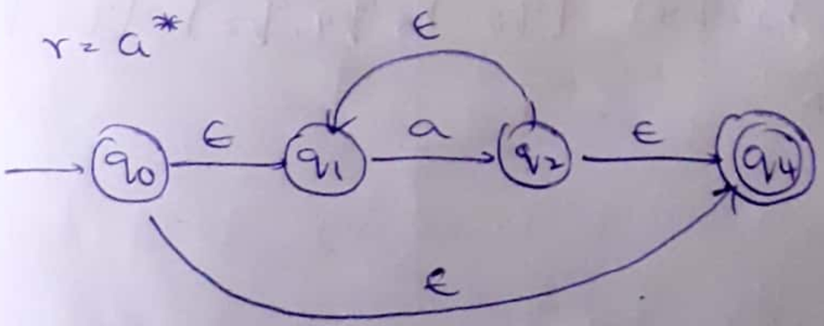
4.) $r = a/b$ (or) $(a+b)$



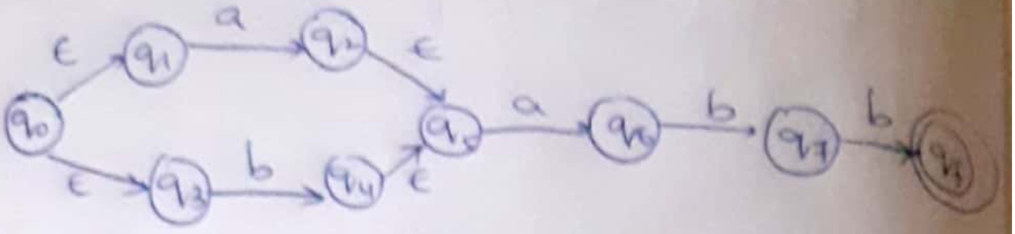
5.) $r = ab$



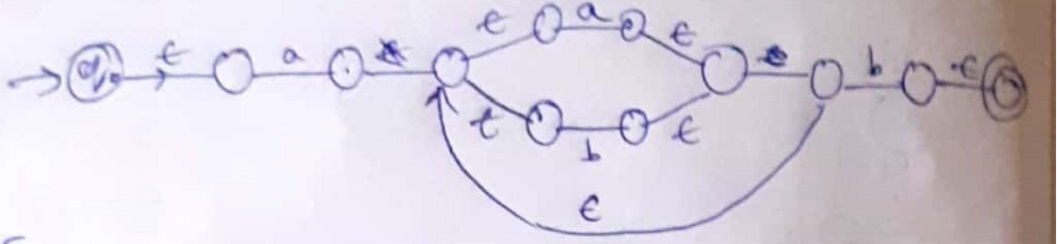
6.) $r = a^*$



Q Obtain ε NFA for the regular expression $(a|b)^*abb$



Q Obtain εNFA for the R.e $a(a+b)^*b$

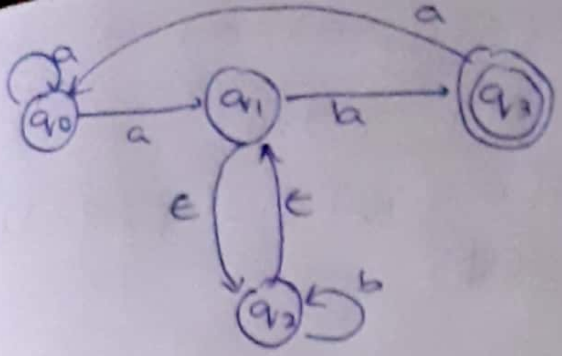


1)

δ	0	1
q_0	$[q_0, q_3]$	$[q_0, q_1]$
$[q_0, q_3]$	$[q_0, q_3, q_4]$	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_3]$	$[q_0, q_1, q_2]$
$[q_0, q_3 q_4]$	$[q_0; q_1, q_4]$	$[q_0, q_1, q_4]$
$[q_0, q_1, q_2]$	$[q_0, q_3, q_2]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_4]$	$[q_0, q_3, q_4]$	$[q_0, q_1, q_2, q_4]$
$[q_0, q_3, q_2]$	$[q_0, q_3, q_4, q_2]$	$[q_0, q_1, q_2]$
$[q_0, q_3, q_2, q_4]$	$[q_0, q_3, q_2, q_4]$	$[q_0, q_1, q_2,$ ~~q_3~~ $, q_4]$
$[q_0, q_3, q_2, q_4]$	$[q_0, q_3, q_4, q_2]$	$[q_0, q_1, q_4, q_2]$

ε-NFA to NFA :-

Q1



ϵ closure$(q_0) = \{q_0\}$

ϵ-closure$(q_1) = \{q_1, q_2\}$

ϵ-closure$(q_2) = \{q_1, q_2\}$

ϵ-closure$(q_3) = \{q_3\}$

$\delta(q_0, a) = \epsilon$-closure$(\delta(\delta(q_0, \epsilon), a))$

$= \epsilon$-closure$(\delta(q_0, a)\}$

$= \epsilon$-closure(q_0, q_1)

$= \{q_0, q_1, q_2\}$

$\delta(q_0, b) = \epsilon$-closure$(\delta(\delta(q_0, \epsilon), b))$

$= \epsilon$-closure$(\delta(q_0, b))$

$= \epsilon$closure(ϕ)

$= \phi$

$\delta(q_1, a) = \epsilon$closure$(\delta(q_1, \epsilon), a))$

$= \epsilon$-closure$(\delta(q_1, q_2), a))$

$= \epsilon$-closure$(\delta(q_1, a) \cup \delta(q_2, a))$

$= \epsilon$-closure$(\phi \cup \phi)$

$= \phi$

$\delta(q_1, b) = \epsilon$-closure$(\delta(q_1, \epsilon), b))$

$= \epsilon$-closure$(\delta(q_1, q_2), b))$

$= \epsilon$-closure$(\delta(q_1, b) \cup \delta(q_2, b))$

$= \epsilon$-closure(q_3, q_2)

$= (q_1, q_2, q_3)$

$\delta(q_2, a) = \epsilon\text{-closure}(\delta(q_2,\epsilon), a)$

$= \epsilon\text{-closure}(\delta(q_1, q_2), a)$

$= \epsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a))$

$= \epsilon\text{-closure}(\phi)$

$= \phi$

$\delta(q_2, b) = \epsilon\text{-closure}(\delta(q_2,\epsilon), b)$

$= \epsilon\text{-closure}(\delta(q_1, q_2), b)$

$= \epsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$

$= \epsilon\text{-closure}(q_3, q_2)$

$= \{q_1, q_2, q_3\}$

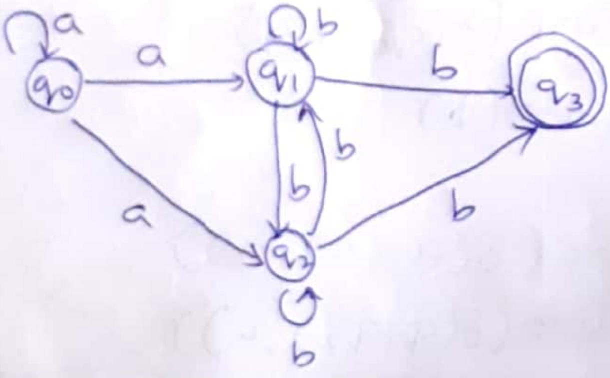
$\delta(q_3, a) = \epsilon\text{-closure}(\delta(q_3, \epsilon), a)$

$= \epsilon\text{-closure}(\delta(q_3, a))$

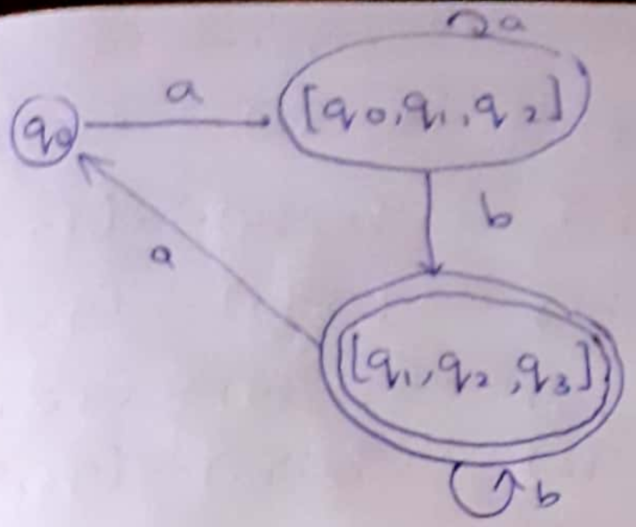
$= \epsilon\text{-closure}\{q_0\}$

$= \{q_0\}$

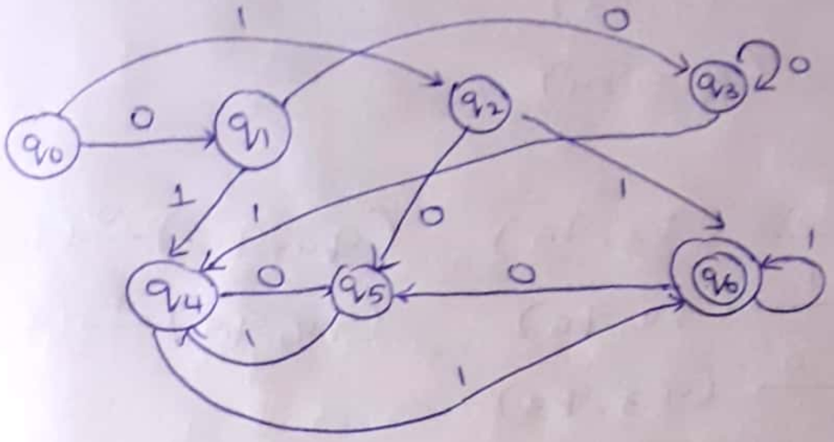
$\delta(q_3, b) = \phi$



δ	a	b
q_0	$\{q_0, q_1, q_2\}$	ϕ
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2, q_3\}$
$\{q_1, q_2, q_3\}$	$\{q_0\}$	$\{q_1, q_2, q_3\}$



Q2.



Transition table :-

δ	0	1
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_5	q_6
q_3	q_3	q_4
q_4	q_5	q_6
q_5	q_3	q_4
q_6	q_5	q_6

$(q_0, q_1) \xrightarrow{0} (q_1, q_3)$

$(q_0, q_1) \xrightarrow{1} (q_2, q_4)$

$(q_0, q_2) \xrightarrow{0} (q_1, q_5)$

$(q_0, q_2) \xrightarrow{1} (q_2, q_6)$

$(q_0, q_3) \xrightarrow{0} (q_1, q_3)$

$(q_0, q_3) \xrightarrow{1} (q_2, q_4)$

$(q_0, q_4) \xrightarrow{0} (q_1, q_5)$

$(q_0, q_4) \xrightarrow{1} (q_2, q_6)$

$(q_0, q_5) \xrightarrow{0} (q_1, q_3)$

$(q_0, q_5) \xrightarrow{1} (q_2, q_4)$

$(q_1, q_2) \xrightarrow{0} (q_3, q_5)$

$(q_1, q_2) \xrightarrow{1} (q_4, q_6)$

$(q_1, q_3) \xrightarrow{0} (q_3, q_3)$

$(q_1, q_3) \xrightarrow{1} (q_4, q_4)$

$(q_1, q_4) \xrightarrow{0} (q_3, q_5)$

$(q_1, q_4) \xrightarrow{1} (q_3, q_6)$

$(q_1, q_5) \xrightarrow{0} (q_3, q_3)$

$(q_1, q_5) \xrightarrow{1} (q_4, q_4)$

$(q_2, q_3) \xrightarrow{0} (q_5, q_3)$

$(q_2, q_3) \xrightarrow{1} (q_6, q_4)$

$(q_2, q_4) \xrightarrow{0} (q_5, q_5)$

$(q_2, q_4) \xrightarrow{1} (q_6, q_6)$

$(q_2, q_5) \xrightarrow{0} (q_3, q_3)$

$(q_2, q_5) \xrightarrow{1} (q_6, q_4)$

$(q_3, q_4) \xrightarrow{0} (q_3, q_5)$

$(q_3, q_4) \xrightarrow{1} (q_4, q_6)$

$(q_3, q_5) \xrightarrow{0} (q_3, q_3)$

$(q_3, q_5) \xrightarrow{1} (q_4, q_4)$

$(q_4, q_5) \xrightarrow{0} (q_5, q_2)$

$(q_4, q_5) \xrightarrow{1} (q_6, q_4)$

(q_0, q_1) (q_0, q_3)

(q_0, q_5) (q_1, q_3)

(q_2, q_4)

(q_3, q_5) (q_1, q_5)

(q_2, q_4) q_6

q_1	✓					
q_2	×	×				
q_3	✓	✓	×			
q_4	×	×	✓	×		
q_5	✓	✓	×	✓	×	
q_6	×	×	×	×	×	×
	q_0	q_1	q_2	q_3	q_4	q_5

Rules :-

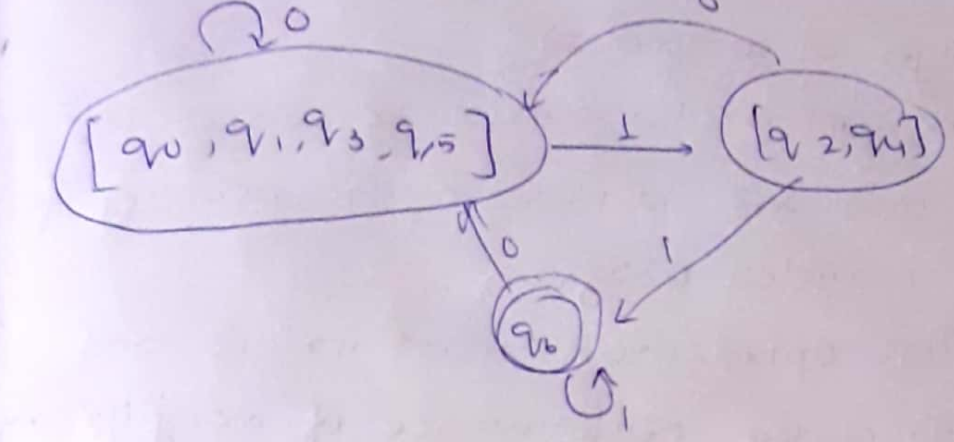
1. If one of the resultant states is FS then × final state

otherwise ✓

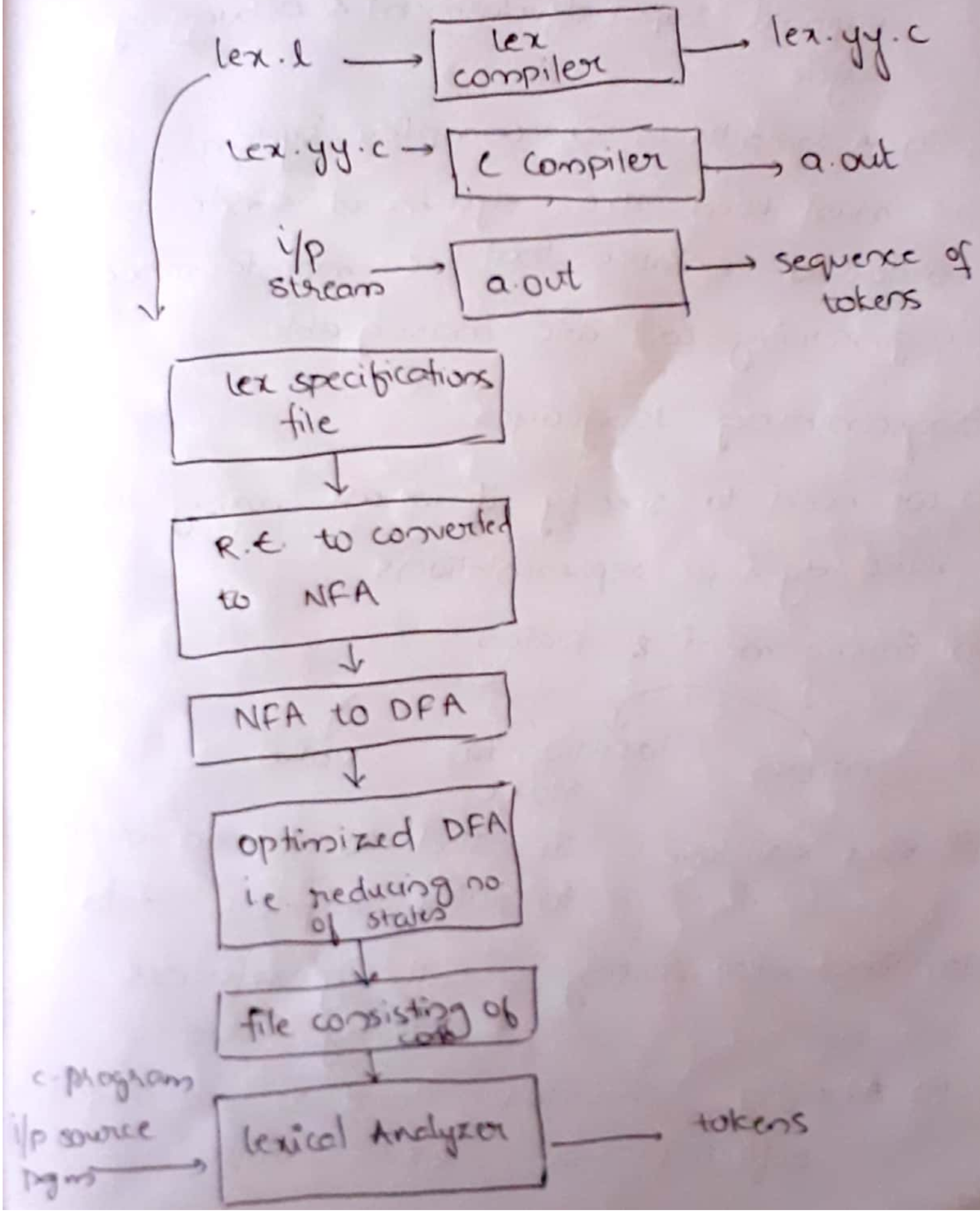
2. If both states are the same equivalent final state then ✓

write down all the combinations where is you put ✓ in the table

(q_0, q_1) (q_0, q_3) (q_0, q_5) —— (q_0, q_4)



Design of lexical Analyzer Generator :-



Science in building of a compiler :-

- Steps that have to be undertaken in order to design a compiler are :-

(1) The optimization must be correct ie, in order ~~the~~ to preserve the meaning of a compiler program

(2) The optimization must include and improve the performance of many i/p progra

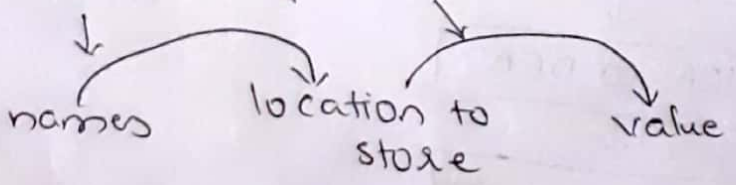
(3) The compilation time must be short to support rapid development & debugging cycle

(4) A compiler is a complex system, so, we must keep the system is simple in order to ensure that the maintainance & engineering cost are manageable.

Programming languages

(1) We need to specify if we are using the static / dynamic representations

(2) Environment & States



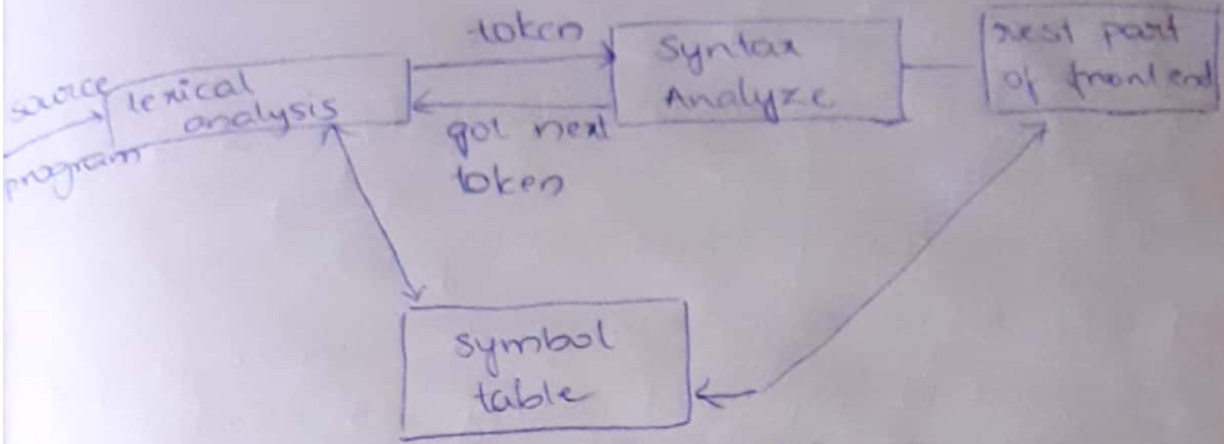
(3) Block structure :- It shd to be answerable to public & private data

(4) Parameter passing : call by reference, call by value

(5) Aliasing :- Changes done to one object is reflected on another object

Unit II.

Syntax Analysis :



There are 3 techniques of parsing :-

(1) universal parsing techniques – It is inefficient to solve parsing

(2) Top down parsing

(3) Bottom up parsing

~~Universing~~

universal parsing technique → cocke younger kasami Alg
→ Earley's Alg

Grammar ⟶ set of rules

language ⟶ set of strings

Notational conventions :- It is used by parsing techniques

(1) Terminals :-

→ Terminals can be represented by

early lowercase alphabets : a, b, c, d . . .

operator symbols : +, –, *, /, ^ . . .

Punctuational symbols :- (,), {, }, ; , ,

The digits – 0, 1, 2 . . . 9

Bold strings – **if** **else** etc

(2) Non terminals :-

These symbols are non - terminals
early uppercase alphabets - A, B, C . .
lower case italic strings - expr, stmt
The start symbol :- S

3) The late uppercase alphabets , suppose X, Y, Z are both terminals & non terminals.
These are called Grammar symbols

4) Late lower case alphabets, suppose x, y, z are used to represent string with terminals

5) Lower-case greek letters α, β, γ are used to represent a string with terminals and non - terminals (variables)

6.) $A \longrightarrow \alpha_1$, $A \longrightarrow \alpha_2$, $A \longrightarrow \alpha_3$ ... $A \longrightarrow \alpha_n$ are a set of productions . Left hand side non terminal is common and then it can be written as

$$A \longrightarrow \alpha_1 | \alpha_2 | \alpha_3 | \dots | \alpha_n$$

$\alpha_1, \alpha_2, \alpha_3 \dots \alpha_n$ are alternatives of A .

7.) unless otherwise stated the left side of first production is the start symbol.

Type(2) or contextfree Grammar (CFG) :

CFG is accepted by pushdown Automata

A Grammar $G = (V, T, P, S)$ is said to be type 2 or CFG if all the productions are of the form $A \longrightarrow \alpha$ where $\alpha \in (V \cup T)^*$ and A is a single non terminal

The symbol ϵ can appear on right hand side of any production.

language generated by this grammar is called as type-2 or context free grammar

$B \longrightarrow Aab/\epsilon$ ✓ $A\epsilon \longrightarrow Ba$ ✗

$A \longrightarrow a$ ✓

$Aa \longrightarrow B$ ✗

$B \longrightarrow \epsilon$ ✓

Elimination of ambiguity :-

$$E \longrightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid -E \mid (E) \mid id$$

The above production is ambiguous. we can dis-ambiguiate the above grammar by specifying associativity and precedences

(i) ( )

(ii) − (unary minus)

(iii) ↑

(iv) *, /

(v) +, −

we introduce a non terminal for each precedence level

⇒ element ⟶ (exp) /id

A subexpression that is essentially undivisible we call as an element. element is a single identifier or (exp)

↓

parenthesized expression

→ we introduce a set of primaries which are the elements with 0 or more occurances of the operations of highest precedence i.e. - unary minus

primary ⟶ - primary/element

→ Then we construct factors as a sequence of one or more primary connected by exponential operator

factor ⟶ primary ↑ factor | primary

⟶ Next we introduce term which are the sequence of one or more factors connected by multiplication & division operator.

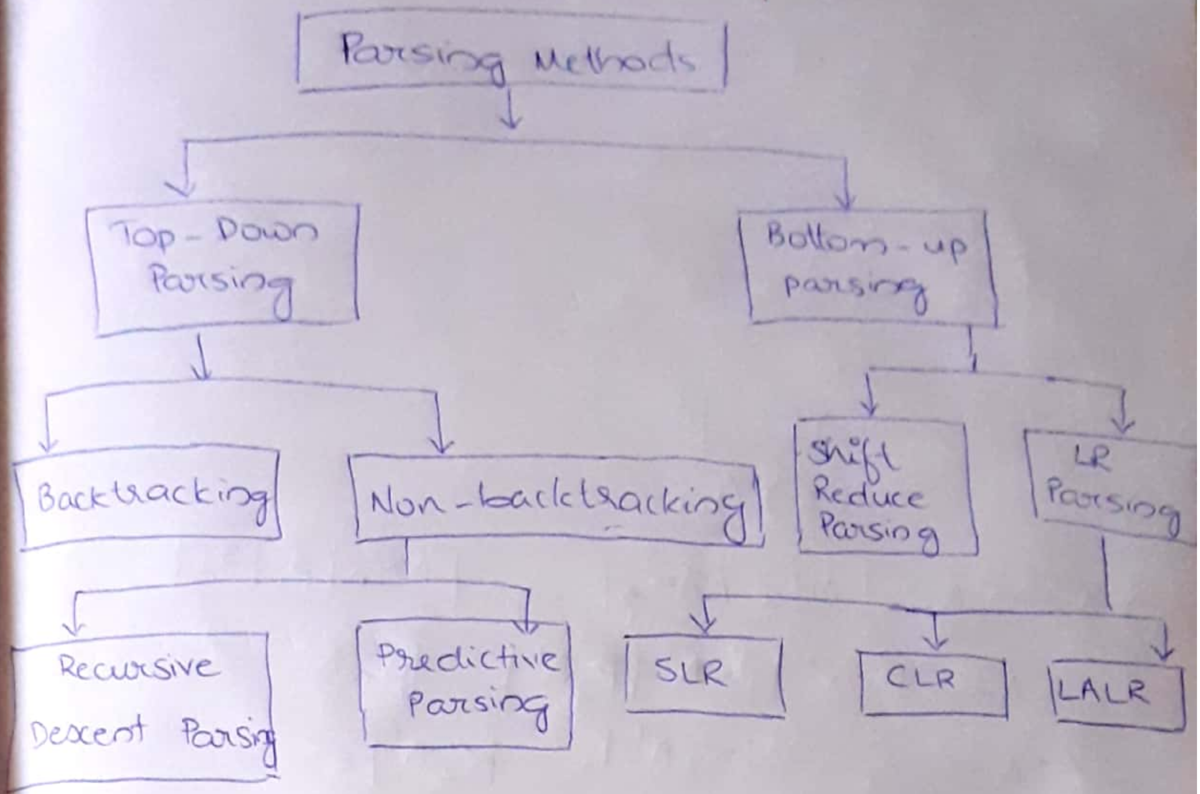
term ⟶ term * factor | term | factor | term

⟶ Next we introduce expression which are the sequences of one or more terms connected by + or - operators

exp ⟶ exp + term | exp - term | term

⟶ E ⟶ E + T | E - T | T
T ⟶ T × F | T | F | F
F ⟶ P ↑ F | P
P ⟶ - P | A
A ⟶ (E) | id

Classification of parsing techniques:



Left Recursion:-

- If a production is of form $A \longrightarrow A\alpha/\beta$ where A is a non-terminal present on lefthand side of the production, then the grammar is left recursive
- Top-Down parsing techniques cannot handle left recursive grammar. So, the transformation that eliminates left recursion is needed.
- The production $A \longrightarrow A\alpha/\beta$ could be replaced with

$A \longrightarrow \beta A'$ where

$A' \longrightarrow \alpha A'/\epsilon$

Examples :

i) $E \Rightarrow E+T/T$

$T \rightarrow T*F/F$

$F = (E)/id$

$E \longrightarrow E\overbrace{+T}^{\alpha}/T \qquad E \longrightarrow TE'$

$E' \longrightarrow +TE'/\epsilon$

$T \longrightarrow T\underbrace{*F}_{\alpha}/F$

$T \rightarrow FT'$

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

solution:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (\epsilon)/id$

2) $A \rightarrow A\alpha_1 | A\alpha_2 / A\alpha_3 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$

$A \rightarrow \beta_1 A' | \beta_2 A' | \beta_3 A' | \ldots | \beta_m A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots \alpha_2 A' | \epsilon$

consider an example,

1) $S \rightarrow Aa|b$

$A \rightarrow Ac|Sd|\epsilon.$

$A \rightarrow Ac|Aad|bd|\epsilon$

$A\alpha_1^{+} | A\alpha_2 | \beta_1 | \beta_2$

$A \rightarrow \beta_1 A' | \beta_2 A' \Rightarrow A \rightarrow bdA' | \epsilon A'$

$A' \rightarrow cA' | ~~Aa~~\ adA' | \epsilon$

Left factoring :-

→ If $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ are the 2 A productions, then and the input begins with a non-empty string derived from α, where we do not know whether to expand $\alpha\beta_1 | \alpha\beta_2$.

→ so we replace these productions by

$A \rightarrow \alpha A'$

where $A' \rightarrow \beta_1 | \beta_2$

Consider an example :

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

solution

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \epsilon \mid eS \Rightarrow S' \rightarrow eS \mid \epsilon$

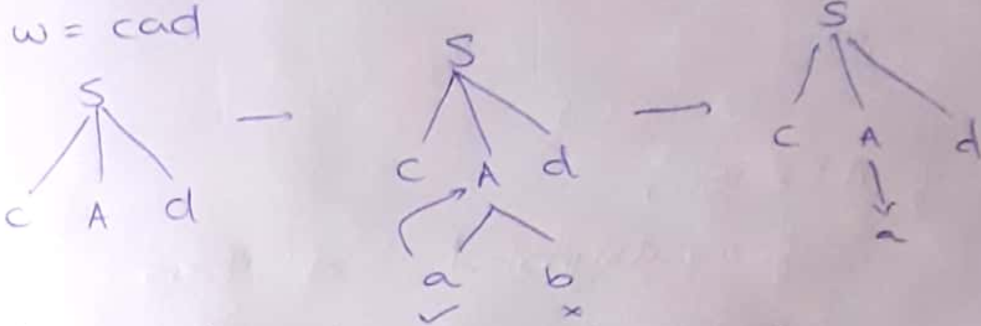
$E \rightarrow b$

Backtracking :-

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

$w = cad$



while parsing of an input string, if an alternative of a non-terminal does not matches, with the symbols of parsing then we need to go back to the non-terminal and choose the other alternative of that non-terminal in order to parse the i/p string. The technique is called as backtracking.

- choosing of alternative is disadvantage of backtracking
- backtracking itself
- left recursion

Disadvantages of top down parsing :-

- Backtracking
- left recursion

→ order of alternatives

→ report of failures

Recursive Procedures :-

(i) Recursive procedures used in backtracking technique of top down parsing.

(a) procedure S() (S → cad)

```
procedure S()
begin
        if input symbol = 'c' then
        begin
                ADVANCE()
        if A() then
                if inputsymbol = 'd' then
                begin
                        ADVANCE()
                return true;
                end;
        end;
    return false;
}
```

(b) Procedure A()

```
procedure A()
begin
    1. Save = inputpointer;
    if input symbol = 'a' then
    begin
            ADVANCE()
    if input symbol = 'b' then
    begin   ADVANCE()
```

```
        return true;
      end;
    end;
    input pointer = i save
      if inputsymbol = 'a' then
      begin
              ADVANCE ( )
              return true;
              end;
      return false;
      end;
```

Recursive descent parsing :-

Consider the example in order to represent non-backtracking recursive descent parsing

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow (E) \mid id$

After eliminating left recursion
New productions are :-

$E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow (E) \mid id$

Note :- Recursive descent parsing makes use of following control structures which tells us which alternative is the only one that would be possibly succeed if we are to find a statement

stmt → if condition then stmt else stmt
while condition do stmt
begin statement - list end

→ Recursive procedures for non terminals of above grammar used in recursive descent parsing

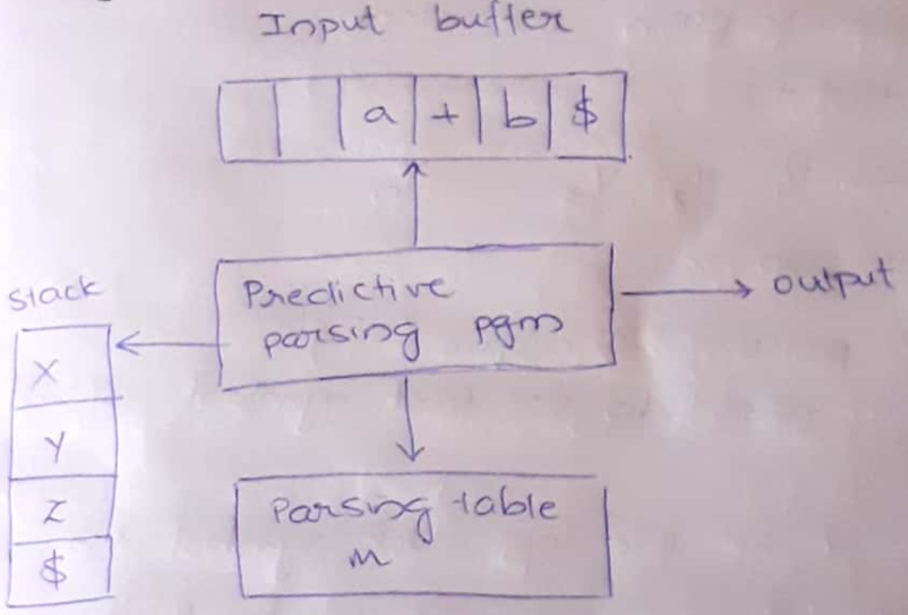
```
procedure E( ):-
begin
        T( ) ;
        Eprime ( )
end
procedure Eprime ( );
    if input symbol = '+' then
        begin
          ADVANCE( ) ;
          T( )
          EPRIME( ) ;
          end ;
procedure T( ) ;
  begin
      F( ) ;
      TPRIME( ) ;
  end
procedure TPRIME ( )
  if input symbol = '*' then
    begin
        ADVANCE( ) ;
        F( ) ;
        TPRIME( ) ;
    end ;
procedure F( )
    if inputsymbol = 'id' then
        ADVANCE ( ) ;
    elseif inputsymbol = '(' then
        begin
```

```
ADVANCE();
E();
if inputsymbol = ')' then
        ADVANCE();
else
        ERROR();
end;
else
  ERROR();        // print what type of error.
```

Non - recursive predictive parsing :-

The tabular implementation of recursive procedures used in recursive descent parsing is predictive parsing.

Block diagram representation of predictive parsing:



Predictive parsing technique of top - down parsing is a table given, which has an input buffer, a stack & a parsing table and an output string.

• The input buffer contains the string to parse, followed by $, where $ symbol is ~~the~~ used as right end marker to indicate, end of i/p string.

The stack contains the grammar symbols with $ at the bottom

- The parsing table is a 2D array indicating M[X, a] where, X is a non-terminal on the top of the stack and 'a' is a terminal or $ symbol
- The parsing is controlled by a program that behaves as follows, where, x is a symbol on the top of the stack and 'a' is a current i/p symbol then there are 3 possibilities.

<i> if $x = a = \$$

i/p string is parsed accepted

<ii> if $x = a \neq \$$

pop x from the stack

ip++;

<iii> $X \longrightarrow Y_1, Y_2, Y_3 \dots Y_n$

m pop x

Predictive parsing technique :-

```
Set i/p to point to the first symbol of w$;
repeat
        let x be the top stack symbol & a
          the symbol pointed to, by the i/p
        if x is terminal or $ then
              if x = a then
                  pop x from the stack and
                  advance p
              else
                  error()
```

```
else /*x is a non terminal */
    if $m(x,a) = x \longrightarrow y_1, y_2 \cdots y_x$ then
    begin pop x from the stack ;
        push $y_k, y_{k-1}, \cdots y_3, y_2 \cdots y_1$ onto the
        stack with y on the top ;
        output the prediction $x \longrightarrow y_1, y_2 \cdots y_k$
    end
        elsef
            error()

out x = $ /* stack is empty */
```

FIRST AND FOLLOW :

To construct predictive parsing table or to make entries to parsing table, the parser uses 2 functions FIRST AND FOLLOW

Rules to compute FIRST(x) where x is a grammar symbol :

1.> if x is a terminal

$\text{FIRST}(x) = \{x\}$

2.> if we have the prediction of the form $x \longrightarrow \epsilon$ then add ϵ to FIRST{x)

If $x \longrightarrow Y_1, Y_2, Y_3 \dots Y_k$

then $\text{FIRST}(x) = \text{FIRST}(Y_1)$

if $\text{FIRST}(Y_1)$ contains ϵ

then $\text{FIRST}(x) = \text{FIRST}(Y_1) - \{\epsilon\} \cup \text{FIRST}(Y_2)$

again if $\text{FIRST}(Y_2)$ also contains ϵ

then $\text{FIRST}(x) = \text{FIRST}(Y_1) \cup \text{FIRST}(Y_2) - \{\epsilon$

$\cup \text{FIRST}(Y_3)$

:

if again $\text{FIRST}(Y_{k-1})$ also contains ϵ

then $\text{FIRST}(x) = \text{FIRST}(Y_1) \cup \text{FIRST}(Y_2) \dots$

$\text{FIRST}(Y_{K-1}) - \{\epsilon\} \cup \text{FIRST}(Y_K)$

if FIRST(Y_K) also contains ϵ then add ϵ to to FIRST(X)

Rules to compute FOLLOW(A) where ∀ non terminals A.

1.) If S is a start symbol

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

where \$ represents the right end marker of i/p.

2.) If $A \longrightarrow \alpha B \beta$

then $\text{FOLLOW}(B) = \text{FIRST}(\beta)$

If FIRST(β) contains ϵ then

$\text{FOLLOW}(B) = \text{FIRST}(\beta) - \{\epsilon\} \cup \text{FOLLOW}(A)$

3.) If $A \longrightarrow \alpha B$

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

coz, there are no grammar symbols after B

4.) If $X \longrightarrow Y_1, Y_2, \cdots Y_K$ then.

$\text{FOLLOW}(Y_1) = \text{FIRST}(Y_2)$

If FIRST(Y_2) contains ϵ then

$\text{FOLLOW}(Y_1) = \text{FIRST}(Y_2) - \{\epsilon\} \cup \text{FIRST}(Y_3)$

⋮

If FIRST(Y_K) also contains ϵ then

$\text{FOLLOW}(Y_1) = \text{FIRST}(Y_2) \cup \text{FIRST}(Y_3) \cdots \cup \text{FIRST}(Y_K) - \{\epsilon\} \cup \text{FOLLOW}(X)$

Q for given grammar $E \longrightarrow E + T \mid T$

$T \longrightarrow T * F \mid F$

$F \longrightarrow (E) \mid id$

construct predictive parsing table using FIRST and FOLLOW functions and check parsing at the i/p string id + id * id

After elimination of left recursion.

$E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow (E) \mid id$

+, -, *, /, id, (,), $ are terminals.

$FIRST(F) = FIRST((E)) \cup FIRST(id)$

$= \{ (, id \}$

$FIRST(T) = FIRST(FT')$

$= FIRST(F)$

$= \{ (, id \}$

$FIRST(E) = FIRST(TE')$

$= FIRST(T)$

$= \{ (, id \}$

$FIRST(E') = FIRST(+TE') \cup FIRST(\epsilon)$

$= \{ +, \epsilon \}$

$FIRST(T') = FIRST(*FT') \cup FIRST(\epsilon)$

$= \{ *, \epsilon \}$

After computation of FIRST (*) for all grammar symbols *. If it doesnot contain ε then no need to compute FOLLOW (A) for all non terminals A

$FOLLOW(E) = \{ \$,) \}$

$FOLLOW(T) = FIRST(E')$

$= \{ +, \epsilon \}, - \{ \epsilon \} \cup FOLLOW(E)$

$= \{ +, \$,) \}$

$\text{FOLLOW}(E') = \text{FOLLOW}(E)$

$= \{ \$,) \}$

$\text{FOLLOW}(T') = \text{FOLLOW}(T)$

$= \{ +, \$,) \}$

$\text{FOLLOW}(F) = \text{FIRST}(T')$

$= \{ *, \epsilon \} - \{ \epsilon \} \cup \text{FOLLOW}(T)$

$= \{ *, +, \$,) \}$

$\text{FIRST}(E) = \{ (, id \}$

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

$\text{FIRST}(T) = \{ (, id \}$

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

$\text{FIRST}(F) = \{ (, id \}$

$\text{FOLLOW}(E) = \{ \$,) \}$

$\text{FOLLOW}(E') = \{ \$,) \}$

$\text{FOLLOW}(T) = \{ +, \$,) \}$

$\text{FOLLOW}(T') = \{ +, \$,) \}$

$\text{FOLLOW}(F) = \{ *, +, \$,) \}$

$E \longrightarrow TE'$
$E' \longrightarrow +TE' | \epsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT' | \epsilon$
$F \longrightarrow (E) | id$

Procedure to make entries into parsing table :

i> For each production $A \longrightarrow \alpha$ do the step 2 and 3

ii> For each terminal A present in first(α) add $A \longrightarrow \alpha$ to $M[A, a]$

iii> If ϵ is present in FIRST(α) then add $A \rightarrow \alpha$ to $M[a, b]$ so the terminal b present in FIRST(α) and also add $A \rightarrow \epsilon$ for the terminals present in FOLLOW(A). If ϵ is is FIRST(α) and \$ is in FOLLOW(A) then add $A \rightarrow \epsilon$ to $M[a, \$]$.

E' ~~E'→+TE'~~

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

NOTE :

(i) If a variable is at the right end of a production ∴ FOLLOW(E
= FOLLOW(left hand side).

(ii) If a variable comes to the right end by substituting its right hand side variables with ε, then FOLLOW(variable) = FOLLOW(left hand side).

Stack	Input	Action
$E	id+id* id $	E →TE'
$E'T	id+id* id $	T → FT'
$ E'T'F	id+id* id $	F → id
$ E'T'id	+id * id $	Pop id and move pointer to the next position.
$E'T'	+ id * id $	T'→ ε
$E'	+id * id $	E' → +TE'
$E'T+	+id * id $	pop + move next
$E'T	id*id $	T → FT'

$\$E'T'F$	$id*id\$$	$F \to id$
$\$E'T'id$	$id*id\$$	pop id, move next
$\$E'T'$	$*id\$$	$T' \to *FT'$
$\$E'T'F*$	$*id\$$	pop *, move next
$\$E'T'F$	$id\$$	$F \to id$
$\$E'T'id$	$id\$$	pop id, move next
$\$E'T'$	$\$$	$T' \to \epsilon$
$\$E'$	$\$$	$E' \to \epsilon$
$\$$	$\$$	string is accepted.

Q. Consider the grammar

$S \longrightarrow AaAb \mid BbBa$

$A \longrightarrow \epsilon$

$B \longrightarrow \epsilon$.

construct the predictive parsing table for it and parse the input string ba

Solution:-

$FIRST(A) = \{\epsilon\}$

$FIRST(B) = \{\epsilon\}$

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

$FOLLOW(A) = FIRST(aAb) \cup FIRST(b)$

$= \{a, b\}$

$FOLLOW(B) = FIRST(bBa) \cup FIRST(a)$

$= \{b, a\}$

$FOLLOW(S) = \{\$\}$

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

stack	input	Action
$S	ba$	S ⟶ ~~AaAb~~ BbBa
$aBbB	ba$	B ⟶ ε
$aBb	ba$	pop b , move the pointer next
$aB	a$	B ⟶ ε
$a	a$	pop a , move next
$	$	input string is accepted.

Consider a grammar

$S \rightarrow iEtSS_1 \mid a$

$S_1 \rightarrow cS \mid \epsilon$

$E \rightarrow b$

construct predictive parsing table for this grammar.

FIRST (S) ⟶ FIRST ($iEtSS_1$) ∪ FIRST(a)

$= \{i, a\}$

FIRST(S_1) ⟶ FIRST(cS) ∪ FIRST (ε)

$= \{c, \epsilon\}$

FIRST(E) = {b}

FOLLOW(S) = {$}

FOLLOW(S_1) = {$} ∪ FIRST($S_1$)

= ~~{ε, $ $~~

= {$, c, ε} − {ε} ∪ FOLLOW(S)

$FOLLOW(S) = \{\$, c\}$

$FOLLOW(S_1) = FOLLOW(S)$

$= \{c, \$\}$

$FOLLOW(E) = \{t\}$

Predictive parsing table :

	i	t	c	a	b	$
S	$S \rightarrow iEtSS_1$			$S \rightarrow a$		
S_1			$S_1 \rightarrow cS$ $S_1 \rightarrow \epsilon$			$S_1 \rightarrow \epsilon$
E					$E \rightarrow b$	

LL(1) Grammar :-

→ If the predictive parsing table doesnot have any multiple entries then the grammar is said to be LL(1) grammar

→ A grammar is LL(1) if the predictive parsing table constructed for that grammar does'nt contain multiple entries.

FIRST L stands for left to right scanning of i/p and second L stands for leftmost derivation, and 1 in the bracket indicates that the next i/p symbol is used to decide what is to be done next in parsing process.

① Construct the predictive parsing table for the below grammar & check whether it is LL(1) or not.

$S \rightarrow 1AB \mid \epsilon$

$A \rightarrow 1AC \mid 0C$

$B \rightarrow 0S$

$C \rightarrow 1$

Note :- Two condititions to check whether the grammar is LL(1) or not

1) for every pair of productions $A \rightarrow \alpha \mid \beta$ if

$FIRST(\alpha) \cap FIRST(\beta) = \phi$ then the grammar is LL(1)

2) If the grammar is not ϵ free then the additional requirement is. i.e, for every pair of productions

$A \longrightarrow \alpha | \beta$

If $FIRST(\beta)$ contains ϵ and $FIRST(\alpha)$ doesn't contains then $FIRST(\alpha) \cap FOLLOW(\beta) = \phi$

Eg:-

$S \rightarrow \overset{\alpha}{AaAb} | \overset{\beta}{BbBa}$ $\quad A \longrightarrow \epsilon \quad B \longrightarrow \epsilon$.

$FIRST(\alpha) \cap FIRST(\beta)$ $\quad FIRST(A) = \epsilon \quad FIRST(B) = \epsilon$

a

$FIRST(\alpha) = FIRST(AaAb)$

$= \epsilon - \{\epsilon\} \cup FIRST(aAb) = \{a\}$

$FIRST(\beta) = FIRST(BbBa)$

$= \epsilon - \{\epsilon\} \cup FIRST(bBa) = \{b\}$

$a \cap b = \phi$

It is LL(1)

Solution 1:

$S \longrightarrow 1AB | \epsilon$

$A \longrightarrow 1AC | 0C$

$B \longrightarrow 0S$

$C \longrightarrow 1$

$FIRST(S) = \{1, \epsilon\}$

$FIRST(A) = \{1, 0\}$

$FIRST(B) = \{0\}$

$FIRST(C) = \{1\}$

$FOLLOW(S) = \{\$\} \cup FOLLOW(B)$

$= \{\$\}$

$FOLLOW(A) = FIRST(B) \cup FIRST(C)$

$= \{0, 1\}$

$FOLLOW(B) = FOLLOW(S)$

$= \{\$\}$

$FOLLOW(C) = \{0, 1\}$

Scanned with CamScanner

Predictive parsing table

	0	1	ε	$
S		S → 1AB	S → ε	
A	A → 0C	A → 1AC		
B	B → 0S			
C		C → 1		

Thus the above grammar is LL(1) as it does not have multiple entries

Bottom - up parsing :-

Ex:- consider a grammar

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

Input string : abbcde

Check whether the i/p string is parsed or not

Solution :-

$\underline{a}bbcde$

matched with right side production of start symbol

$a\underline{Ab}cde$ $(A \rightarrow b)$

$aAde$ $(A \rightarrow Abc)$

$aA\underline{d}e$ $(B \rightarrow d)$

$aABe$

S

Reverse of rightmost derivation of starting symbol is bottom up parsing

i/p string :- abbcde

$S \rightarrow aAB\underline{e}$

matched with production on right side

$S \xrightarrow{rm} aA\underline{B}e$ $(B \rightarrow d)$

$S \xrightarrow{rm} aAde$ $(A \rightarrow Abc)$

$S \xrightarrow{rm} a\underline{A}bcde$ $(A \rightarrow b)$

$S \xrightarrow{rm} abbcde$

Handle :- A handle of a string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of a production represents one step along reverse of rightmost derivation. This derivation is called handle pruning.

Ex:- consider the grammar:

$E \rightarrow E+E \mid E*E \mid id$ show the handle of each right sentential form for the string id+id*id

Soln:- $E \rightarrow E+E$

$\rightarrow E+E*E$

$\rightarrow E+E*id$

$\rightarrow E+id*id$

$\rightarrow id+id*id$

Sentential form	Handle	Reducing production
id+id*id	id	$E \rightarrow id$
E+id*id	id	$E \rightarrow id$
E+E*id	id	$E \rightarrow id$
E+E*E	E*E	$E \rightarrow E*E$
E+E	E+E	$E \rightarrow E+E$
E		

Consider the following grammar $S \rightarrow (L) \mid a$ $L \rightarrow L,S \mid S$. Show that the handle of each right sentential form for the string (a,(a,a))

$S \rightarrow (L)$

$S \rightarrow (L,S)$ $L \rightarrow L,S$

$S \rightarrow (L,(L))$ $S \rightarrow (L)$

$S \rightarrow (L,(L,S))$ $L \rightarrow L,S$

$S \rightarrow (L,(L,a))$ $S \rightarrow a$

S → (L,(S,a))   L → S

S → (L,(a,a))   S → a

S → (S,(a,a))   L → S

S → (a,(a,a))   S → a

Sentential form	handle	Reducing production
(a,(a,a))	a	S → a
(S,(a,a))	S	L → S
(L,(a,a))	a	S → a
(L,(S,a))	S	L → S
(L,(L,a))	a	S → a
(L,(L,S))	{L,S}	L → L,S
(L,(L))	(L)	S → (L)
(L,S)	{·L,S	L → L,S
(L)	(L)	S → (L)
S		

Shift reduce parsing :- This technique is convinient way to implement bottom up parsing of an i/p string where it consists of a stack which is empty with $ symbol at bottom. An i/p buffer consisting of w$ where w is an i/p string. The parser operates by shifting 0 or more i/p symbols onto the stack until a handle β is on the top of the stack.

The 4 actions performed by shift reduce parser are :-

i> In shift action, the next i/p symbol is shifted on the top of the stack.

<ii> In reduce action, the parser knows the right end of the handle is at top of the stack. It must take or locate the left end of the handle within the stack and beside with what non-terminal to replace the handle.

<iii> In an accept action, the parser announces the successful completion parsing of i/p string

(iv) In an error action, the parser discovers the syntax error has occured and cause error recovery routine.

ex:- For the given grammar $S \rightarrow (L) | a$
$L \rightarrow L,S / S$

using shift reduce parsing parse the i/p string (a,(a,a))

<u>soln:-</u>

$S \rightarrow (L) | a$

$L \rightarrow L,S | S$

stack	Input buffer	Reduced production
$	(a,(a,a))$	shift (
$(	a,(a,a))$	Shift a
$(a	,(a,a))$	Reduce S → a
$(S	,(a,a))$	Reduce L → S
$(L	,(a,a))$	Shift ,
$(L,	(a,a))$	Shift (
$(L,(	a,a))$	Shift a
$(L,(a	,a))$	Reduce S → a
$(L,(S	,a))$	Reduce L → S
$(L,(L	,a))$	Shift ,
$(L,(L,	a))$	shift a

$(L,(L,a	))$	Reduce S → a
$(L,(L,S	))$	Reduce L → L,S
$(L,(L	))$	Shift)
$(L,(L)	)$	Reduce S → (L)
~~$(L,S~~	~~)$~~	~~Shift)~~
$(L,S	)$	Reduce L → L,S
$(L	)$	Shift)
$(L)	$	Reduce S → (L)
$S	$	

5. Parse i/p string id+id*id for the grammar.

E → E+T | T
T → T*F | F
F → (E) | id

using Shift reduce parsing

6. Parse i/p string int id, id; using shift reduce parsing for the grammar.

S → TL
T → int / float
L → L, id / id

Operator precedence grammar:-

Operator grammar:- The grammar which has no ε on the right side of the production and no 2 adjacent non terminals is called operator grammar.

→ Two adjacent non-terminals

ex:- E → EAE | (E) / -E / id

This is not an operator grammar

A → + | - | * | / | ↑

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id

This is an operator grammar.

operator precedence grammar makes use of 3 disjoint precedence relation b/w certain pair of ~~terms~~ terminals

i> $a <\cdot b$ " a yeilds precedence to b "

ii> $a \doteq b$ " a having same precedence to b "

iii> $a \cdot> b$ " a having precedence over b "

Ex: consider operator precedence relations for the right sentinal form id+id * id

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

using precedence table the input id+id * id is written as:

$\$ <\cdot id \cdot> + <\cdot id \cdot> * <\cdot id \cdot>$

$<\cdot id \cdot>$ — it is called as handle.

Steps undertaken by operator precedence grammar in order to identify handle.

- scan i/p string from left end until the first $\cdot>$ precedence is found then scan backwards over $\doteq$ until $<\cdot$ is found.
- The left of $\cdot>$ and right of $<\cdot$ is considered as handle

operator precendence relation based upon precedences and associativity

i> If operator θ_1 has higher precedence than operator θ_2 then, $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

ii> If operator θ_1 has the same precedence with operator θ_2 then,

$\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if operator is left associative (+ , −)

$\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if operator is right associative (*, /)

(iii) If θ is any operator, then make

$\theta <\cdot$ id

id $\cdot> \theta$ (because id has highest precedence over all)

$ $<\cdot \theta$ $ has the least precedence.

$\theta \cdot>$ $ θ has higher precedence

$\theta <\cdot$ ( ) $\cdot> \theta$

($<\cdot \theta$ $\theta \cdot>$)

apart from these, also make (≐), (<.(,).>)

id .> $ (<.id $<.(~~[illegible]~~

$ <.id id.>) { (.>$

Based upon the above rules construct the operator precendence relation for the above string

id*(id ↑ id) − id/id

+, −, *, /, ↑ id, (,), $

left associative → right associative

	+	−	*	/	↑	id	(	)	$
+	.>	.>	<.	<.	<.	<.	<.	.>	.>
−	.>	.>	<.	<.	<.	<.	<.	.>	.>
*	.>	.>	.>	.>	<.	<.	<.	.>	.>
/	.>	.>	.>	.>	<.	<.	<.	.>	.>
↑	.>	.>	.>	.>	<.	<.	<.	.>	.>
id	.>	.>	.>	.>	.>			.>	.>

	+	-	*	/	↑	id	(	)	$
(	<.	<.	<.	<.	<.	<.	<.	=	
)	.>	.>	.>	.>	.>			.>	.>
$	<.	<.	<.	<.	<.	<.	<.	<.	

we have to check left associative or right associativity.

Parsing of i/p string id + id * id

stack	input buffer
$	id+id * id$
$<. id	+id* id$
$ <.id.>	+ id * id$
$ <.F + <.	id * id $
$<.F+ <.id.>	* id $
$<F+ F *<.	id $
$ <.F+F * <.id.>	$
$ <.F+F*F.>	$

Advantages:-

- Operator precedence technique is easy to implement

Disadvantage:-

- only small class of grammar are supported
- It can't parse / assurance of acceptance of i/p string for desired language
- It will not support or handle operators which is having 2 different precedences.

Precedence function :-

we make use of 2 functions f and g, that for symbols A & B we can write

i> if f(a) < g(b) whenever a <· b

ii> if f(a) > g(b) whenever a ·> b

iii> f(a) = g(b) whenever a = b

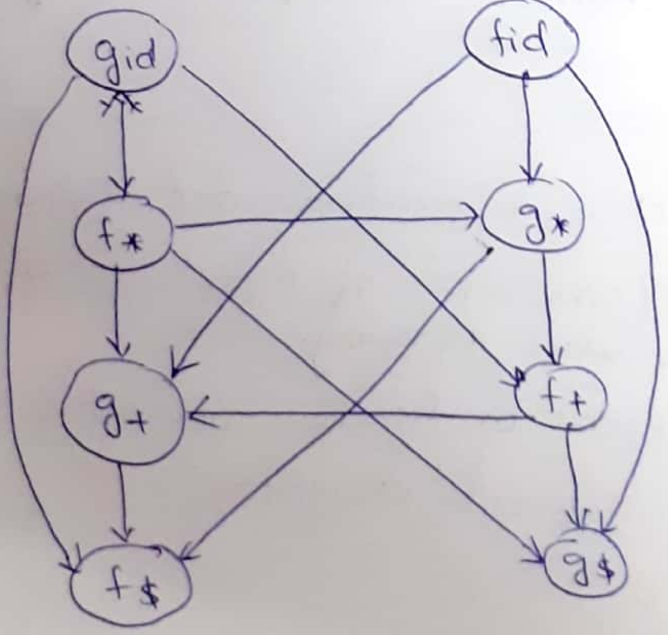
Consider an example which represents the precedence matrix for the terminals id, +, *, $

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

id, +, *, $ has no equal precedence so draw separate nodes using f and g

if f(a) ·> g(b) draw f(a) ⟶ g(b)

if ~~g~~ f(a) <· g(b) draw f(a) ⟵ g(b)



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

① Consider a precedence matrix for +, *, (,), id, $

	+	*	(	)	id	$
+	.>	<.	<.	.>	<.	>
*	.>	.>	<.	.>	<.	>
(	<.	<.	<.	=.	<.	
)	.>	.>	=.	.>		.>
id	.>	.>	~~.>~~	.>		.>
$	<.	<.	<.		<.	

For this matrix, draw precedence graph & also mention the numerical values for function f and g w.r.t all terminals.

LR Parsing :-

Precedence Functions :-

Compilers using operator precedence parsers need not store the take of precedence relations. In most cases, the table can be encoded by the two precedence functions f & g that map terminal symbols to integers. We attempt to select f & g so that for symbols a and b.

1> f(a) < g(b) whenever a < b
2> f(a) = g(b) whenever a = b
3> f(a) > g(b) whenever a > b

The precedence relation b/w a & b can be determined by a numerical comparison b/w

$f(a)$ and $g(b)$

Eg:- The precedence functions for the precedence table consisting of +, −, *, /, ↑, (,), id, $ is

	+	−	*	/	↑	(	)	id	$
f									0
g									0

Algorithm : Constructing precedence functions:-

Input :- An operator precedence matrix

Output :- Precedence functions representating the i/p matrix or an indication that none exists

Method :-

1) Create symbols fa & ~~gb~~ ga for each a or $ a → i.e terminal or $

2) partition the created symbols into as many groups as possible in such a way that if $a \doteq b$, then fa & gb are in the same group

3) Create a directed graph whose nodes are the groups in ②

For any a & b, if $a <\cdot b$, P

construct SLR parsing table for the following grammar.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Step 1 :- Augument the grammar & add a . operator at the beggining of every grammar.

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

(∴ Always begin with $S' \rightarrow .S$
↓
Start symbol)

Step 2: Find closure $(E' \rightarrow E)$: I_0

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

→ shift the . to the next place.

Step3 :- closure [goto(I_0, E)]

→ nonterminals / grammar symbols terminals

$E' \rightarrow E.$

$E \rightarrow E.+T$

I_1

Step4: closure [goto$(I_0, T]$)

$T \rightarrow T.*F$

~~$T \rightarrow F.$~~

$E \rightarrow T.$

I_2

step 5 :- closure [goto [I_0, F]]

$\left. T \rightarrow F \cdot \right\} I_3$

step6 :- closure [goto [I_0, (]]

closure { $F \rightarrow (\cdot E)$ }

$$\left.\begin{array}{l} E \rightarrow \cdot E+T \\ E \rightarrow \cdot T \\ T \rightarrow \cdot T * F \\ T \rightarrow \cdot F \\ F \rightarrow \cdot (E) \\ F \rightarrow \cdot id \end{array}\right\} I4.$$

step7 :- closure [goto (I_0, id]]

$\left. F \rightarrow id \cdot \right\} I_5$

Step 8:- perform closure [goto [I_i, nonterminal/ grammar symbol/ terminals]

(i) closure [goto [I_1 , +]] → I_6

closure{ $E \rightarrow E+\cdot T$ }   using T from I_0

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

(ii) closure[goto[I_2, *]] → I_7

closure { $T \rightarrow T * \cdot F$ }

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

(iii) closure [goto [I_4, E]]

$$\left.\begin{array}{l} F \rightarrow (E \cdot) \\ E \rightarrow E \cdot + T \end{array}\right\} I_8 \checkmark$$

(iv) closure [goto [I_4, T]] → already exists

$$\left.\begin{array}{l} E \rightarrow T\cdot \\ T \rightarrow T\cdot * F \end{array}\right\} I_2$$

(v) closure ~~of~~ $[\text{goto}(I_4, F]]$

$$T \rightarrow F\cdot \quad \} I_3$$

(vi) closure $(\text{goto}(I_4, ( ])$

$$\text{closure}\left\{ \begin{array}{l} F \rightarrow (\cdot E) \\ E \rightarrow \cdot E + T \\ E \rightarrow \cdot T \\ T \rightarrow \cdot T * F \\ T \rightarrow \cdot F \\ F \rightarrow \cdot (E) \\ F \rightarrow \cdot id \end{array}\right\} I_4$$

closure $(\text{goto}(I_4, id]] = I_5$

(vii) closure $(\text{goto}(I_6, T])$

$$\left.\begin{array}{l} E \rightarrow E + T\cdot \\ T \rightarrow T\cdot * F \end{array}\right\} I_9 \checkmark$$

(viii) closure $\text{goto}(I_6, F] = I_3$

closure $\text{goto}(I_6, ( ] = I_4$

closure $\text{goto}[I_6, id] = I_5$

(ix) closure $[\text{goto}(I_7, F]] = I_{10}$ ✓

$$T \rightarrow T * F\cdot$$

(x) closure $(\text{goto}(I_7, ( ]] = I_4$

closure $(\text{goto}(I_7, id]) = I_5$

(xi) closure $(\text{goto}[I_8,)]] = I_{11}$ ✓

$$F \rightarrow (E)\cdot$$

closure $(\text{goto}(I_8, +]) = I_6$

(xii) closure $[\text{goto}(I_9, *]] = I_7$

closure $(\text{goto}(I_9,$

closure[goto[I_0, E]] = I_1

closure[goto[I_0, T]] = I_2

closure[goto[I_0, F]] = I_3

closure[goto(I_0, (]] = I_4

closure[goto(I_0, id]] = I_5

closure[goto(I_1, +)] = I_6

closure[goto[I_2, *]) = I_7

closure[goto[I_4, E]) = I_8

closure[goto[I_4, T]) = I_2

closure[goto[I_4, F]) = I_3

closure[goto(I_4, ( )] = I_4 closure[goto[I_4, id]] = I_5

closure[goto(I_6, T]) = I_9

closure(goto[I_6, F]) = I_3

closure[goto[I_6, (]) = I_4

closure[goto(I_6, id)] = I_5

closure(goto(I_7, F)] = I_{10}

closure[goto(I_7, (]) = I_4

closure[goto(I_7, id]) = I_5

closure(goto(I_8,)]) = I_{11}

closure(goto(I_8, +)]) = I_6

closure(goto(I_9, *]) = I_7

Scanned with CamScanner

state	id +	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6	~~S_7~~			accepted			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

$E \rightarrow TE'$
$E' \rightarrow +TE' / \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' / \epsilon$
$F \rightarrow (E) | id$

FIRST (E) = { id, (}
FIRST(T) = { id, (}
FIRST(F) = { id, (}

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

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

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

$I_1, I_2, I_3, I_5, I_9, I_{10}, I_{11}$

→ Choose the states that contain a production that ends with a •

1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

- The states that contain the production that 'end with .

Steps.

1) In state 1, (I_1) the production that ends with . is $E' \longrightarrow E.$. Since it is an augment production make the entry as "accept" and with the state I_1 and the terminal \$

2) In state I_2, the production that ends with . is $E \longrightarrow T.$. So in the given grammar the production number is 2, so make the entry as r_2. In state I_2 w.r.t. all the terminals in FOLLOW(E)

$E \longrightarrow T$ is r_2 fill it in FOLLOW(E)

3) I_3 $T \longrightarrow F.$ is production r_4

Fill the table with FOLLOW(T)

4) I_5: $F \longrightarrow id.$ is r_6 fill it in table at FOLLOW(F)

6.) I_9: $E \longrightarrow E+T.$ is production r_1

7.) I_{10}: $T \longrightarrow T*F.$ is production r_3. fill it in FOLLOW(T)

8.) I_{11}: $F \longrightarrow (E).$ is production r_5

fill it with FOLLOW(F)

Scanned with CamScanner

Q. with reference to the above parsing table pass the ~~input~~ input string id+id*id $

state	i/p buffer	Action
\$0	id+id*id\$	Shift s_5
\$0id5	+id*id\$	Reduce r_6 $F \rightarrow id$
\$0F3	+id*id\$	Reduce r_4 $T \rightarrow F$
\$0T2	+id*id\$	Reduce r_2 $E \rightarrow T$
\$0E1	+id*id\$	Shift s_6
\$0E1+6	id*id\$	Shift s_5
\$0E1+6id5	*id\$	Reduce r_6 $F \rightarrow id$
\$0E1+6F3	*id\$	Reduce r_4 $T \rightarrow F$
\$0E1+6T9	*id\$	Shift s_7
\$0E1+6T9*7	id\$	Shift s_5
\$0E1+6T9*7id5	\$	Reduce r_6 $F \rightarrow id$
\$0E1+6T9*7F10 pop 6 elements	\$	Reduce r_3 $T \rightarrow \underbrace{T * F}_{3 \times 2 = 6}$
\$0E1+6T9	\$	Reduce r_1 $E \rightarrow E+T$
\$0E1	\$	Accepted.

Q. With respect to the above parsing table $id * id\$$:- parse the input string

State	i/p buffer	Action
$0	id*id$	Shift S_5
$0id5	*id$	Reduce r_6 $F \to id$
$0F3	*id$	Reduce r_4 $T \to F$
$0T2	*id$	Shift S_7
$0T2*7	id$	Shift S_5
$0T2*7 id5	$	reduce r_6 $F \to id$
$0T2*7F10	$	Reduce r_3 $T \to T*F$
$0T2	$	Reduce r_2 $E \to T$
$0E1	$	Accept.

Canonical LR Parsing technique :-

Q. Construct canonical L Parsing table for the given grammar.

$S \to CC$

$C \to aC / d$

LR(1)

$[A \to \alpha . B\beta, a]$

$[B \to \gamma, a]$

$[B \to .\gamma, b]$

↳ look ahead symbol

↑ FIRST(β, a)

Soln:-

$S' \to .S$

$S \to .CC$

$C \to .aC$

$C \to .d$

In CLR we should have the productions in the form $[A \to \alpha . B\beta, a]$

(1) closure $[S' \to .S, \$]$

Since 1st production is augmented grammar, look ahead symbol = \$.

compare $[A \to \alpha . B\beta, a]$

$A \to S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$, $a = \$$

~~clos~~ $[S \to .CC, \$]$

$$\mathrm{FIRST}(\beta a) = \mathrm{FIRST}(\epsilon \$)$$
$$= \$$$

$S' \to .S, \$$

closure$[S \to .CC, \$]$

$A \to \alpha . B\beta, a$

$A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$, $a = \$$

$[C \to .aC, a/d]$

$[C \to .d, a/d]$

$$\mathrm{FIRST}(\beta a) = \mathrm{FIRST}(C\$)$$
$$= \mathrm{FIRST}(C)$$
$$= \{a, d\}$$

Now write all the productions

$$\left.\begin{array}{l} S' \to .S, \ \$ \\ S \to .CC, \ \$ \\ C \to .aC, \ a/d \\ C \to .d, \ a/d \end{array}\right\} I_0$$

closure$[\mathrm{goto}[I_0, S]] = I_1$

closure$[\mathrm{goto}[I_0, C]] = I_2$

closure$[\mathrm{goto}[I_0, a]] = I_3$

closure$[\mathrm{goto}[I_0, d]] = I_4$

closure$[\mathrm{goto}[I_2, C]] = I_5$

closure$[\mathrm{goto}(I_2, a]] = I_6$

closure$[\mathrm{goto}[I_2, d]] = I_7$

closure$[\mathrm{goto}(I_3, C]] = I_8$

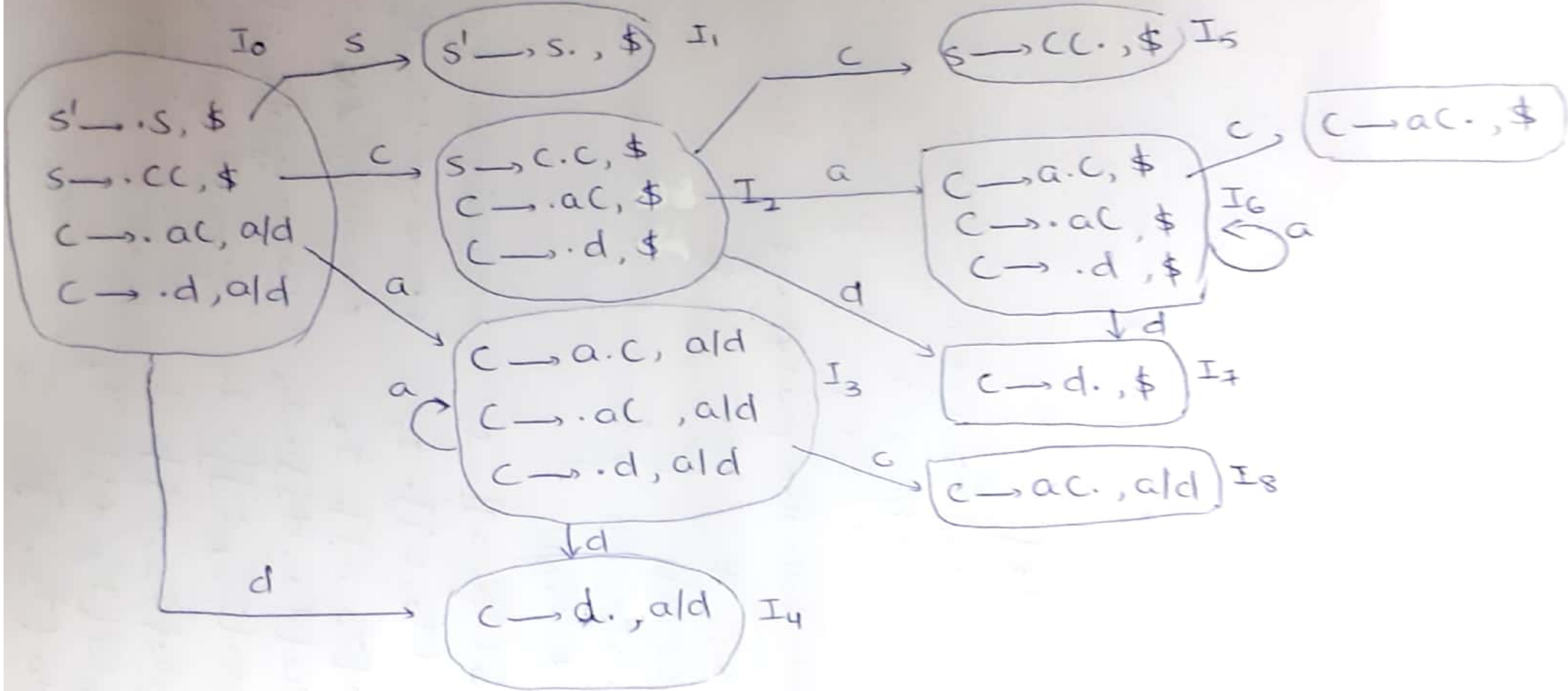
closure$[\mathrm{goto}[I_3, a]] = I$

closure$[\mathrm{goto}[I_3, d]] = I_4$

closure$[\mathrm{goto}(I_6, C]] = I_9$

closure$[\mathrm{goto}[I_6, a]] = I_6$

closure$[\mathrm{goto}[I_6, d]] = I_7$


I0
S' → ·S, $
S → ·CC, $
C → ·aC, a/d
C → ·d, a/d
S
I1
S' → S·, $
C
S → C·C, $
C → ·aC, $
C → ·d, $
I2
C
I5
S → CC·, $
a
C → a·C, $
C → ·aC, $
C → ·d, $
I6
a
C
C → aC·, $
d
d
C → d·, $
I7
a
C → a·C, a/d
C → ·aC, a/d
C → ·d, a/d
I3
a
C
C → aC·, a/d
I8
d
d
C → d·, a/d
I4

state	Action			Goto	
	a	d	$	S	C
0	s_3	s_4		1	2
1			accept		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
			r_2		

$I_1, I_4, I_5, I_7, I_8, I_9$

1. $S \to CC$
2. $C \to aC$
3. $C \to d$

For each of the states, using productions that end with a . Place the r_i (where i is production number) in the look ahead terminal.

Construct w.r.t the above constructed canonical parsing table, parse the input string aadd

Stack	i/p buffer	Action
$0	aadd$	shift S_3
$0a3	add$	shift S_3
$0a3a3	dd$	shift S_4
$0a3a3d4	d$	Reduce R_3 $C \to d$
$0a3a3C8	d$	Reduce r_2 $C \to aC$

ϕ0a3C8	d$	Reduce R_2 $C \to aC$
ϕ0C2	d$	Shift S7
ϕ0C2d7	$	Reduce R_3 $C \to d$
ϕ0C2C5	$	Reduce R_1 $S \to CC$
ϕ0S1	$	accepted.

LALR Parsing technique :-

I_0 : $S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a/d$
$C \to .d, a/d$

I_1 : $S' \to S., \$$

I_2 : $S \to C.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

$I_{36} =$ $C \to a.C, a/d/\$$
$C \to .aC, a/d/\$$
$C \to .d, a/d/\$$

$I_{47} =$ $C \to d.\ a/d/\$$

$I_{89} =$ $C \to aC.\ a/d/\$$

	a	d	$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

To generate a parse tree in LALR, we used
YACC ⟶ Yet Another Compiler Compiler

A grammar is said to be ambiguous, if we have ~~a~~ more than 1 parse tree
learn left parse tree & right parse tree.

Q. construct predictive parsing table for the grammar

$S \rightarrow (L)/a$

$L \rightarrow L,S/S$

and check if the input string (a,a) be parsed or not

Soln:-

$S \rightarrow (L)/a$

$L \rightarrow SL'$

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

$\text{FIRST}(S) = \text{FIRST}((L)) \cup \text{FIRST}(a)$

$= \{(,a\}$

$\text{FIRST}(L) = \text{FIRST}\{SL'\}$

$= \text{FIRST}\{S\} = \{(,a\}$

$\text{FIRST}(L') = \text{FIRST}\{,SL'\} \cup \text{FIRST}(\epsilon)$

$= \{,,\epsilon\}$

$\text{FOLLOW}(S) = \text{FOLLOW}(L') \cup \{\$\}$

$= \{\$,)\}$

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

$\text{FOLLOW}(L') = \{\text{FOLLOW}(L)\} \cup \text{FOLLOW}(L')$

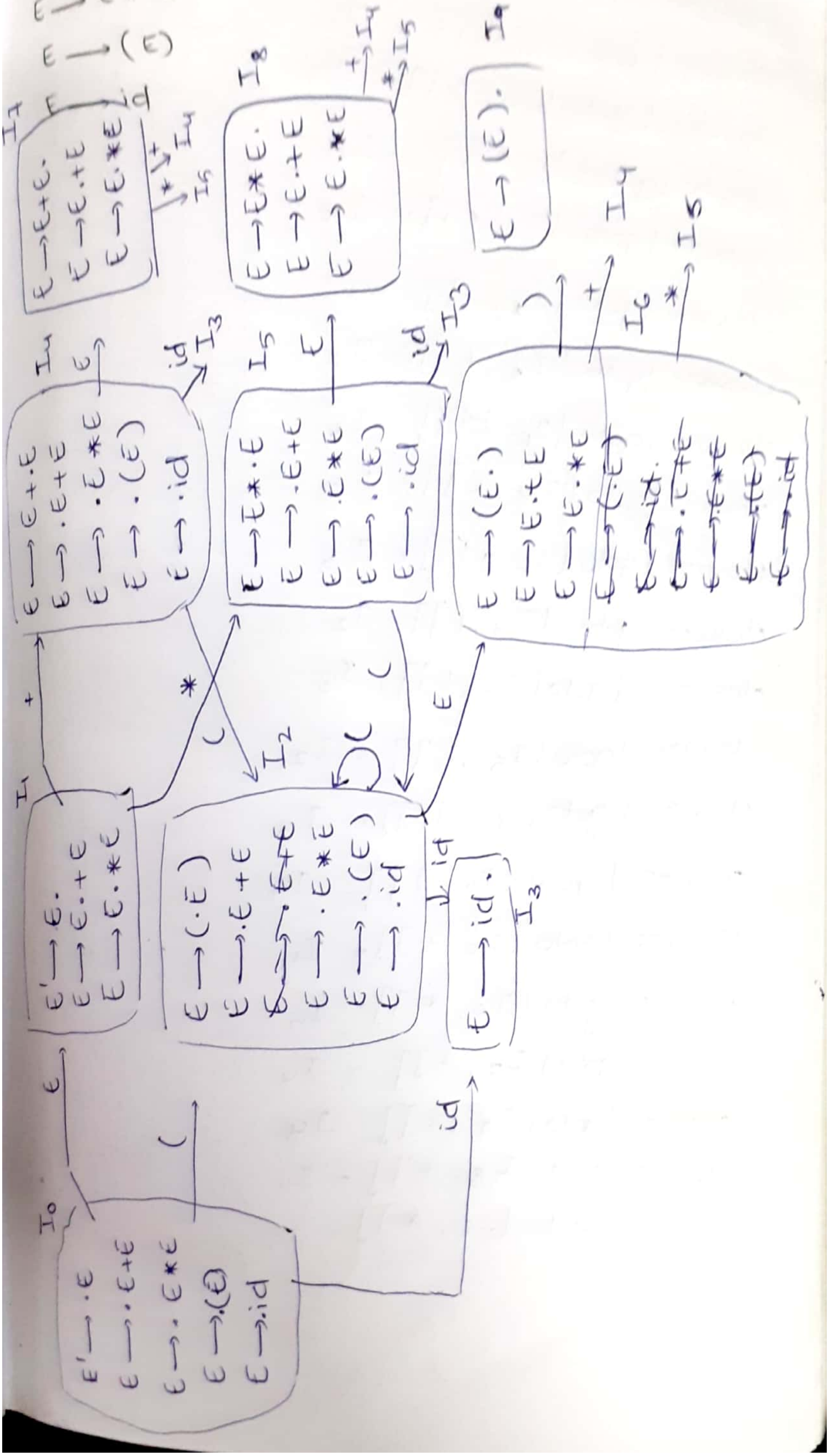
$= \{)\}$

	(	a	,	ε	$	)
S	S→(L)	S→a				
L	L→SL'	L→SL'				
L'			L'→,SL'	[illegible]		L→ε

Stack	Input	action
$S	(a,a)$	S→(L)
$)L(	(a,a)$	pop (and move the next pointer
$)L	a,a)$	L→SL'
$)L'S	a,a)$	S→a
$)L'a	a,a)$	pop a move the pointer
$)L'	,a)$	L'→,SL'
$)L'S,	,a)$	pop , move the pointer
$)L'S	a)$	S→a
$)L'a	a)$	pop a and move the next pointer
$)L'	)$	L'→ε
$)	)$	pop) move next pointer
$	$	accepted

Handling Ambiguous Grammar

E → E + E
E → E * E
E → (E)
E → id



closure (goto (I_0, E)) = I_1

closure (goto (I_0, ( )) = I_2

closure (goto (I_0, id)) = I_3

closure (goto (I_1, +)) = I_4

closure (goto (I_1, *)) = I_5

closure (goto (I_2, E)) = I_6

closure (goto (I_2, ( )) = I_2

closure (goto (I_2, id)) = I_3

closure (goto (I_4, E)) = I_7

closure (goto (I_4, id)) = I_3

closure (goto , (I_4, ( )) = I_2

closure (goto (I_5, E)) = I_8

closure (goto (I_5, ( )) = I_2

closure (goto (I_5, id)) = I_3

closure (goto (I_6,))) = I_9

closure (goto (I_6, +)) = I_4

closure (goto (I_6, *)) = I_5

closure (goto (I_7, +)) = I_4

closure (goto (I_7, *)) = I_9

closure (goto (I_8, +)) = I_4

closure (goto (I_8, *)) = I_5

	Action						goto
State	id	+	*	(	)	$	E
0	s_3			s_2			1
1		s_4	s_5			accept	
2	s_3			s_2			6
3		r_4	r_4			r_4	
4	s_3			s_2			7
5	s_3			s_2			8
6		s_4	s_5		s_9		
7		s_4 or r_1	s_5 or r_1			r_1	
8		s_4 or r_2	s_5 or r_2			r_2	
9		r_3	r_3			r_3	

① $E \longrightarrow E + E$

② $E \longrightarrow E * E$

③ $E \longrightarrow (E)$

④ $E \longrightarrow id$

FIRST $(E) = \{ (, id \}$

follow $(E) = \{\$, +, *, \cancel{\$})\}$

I_1, I_3, I_7, I_8, I_9

$I_7 : E \longrightarrow E+E.$

$I_8 : E \longrightarrow E*E.$

$I_9 : E \longrightarrow (E).$

$I_1 \rightarrow E' \rightarrow E.$

$I_3 \rightarrow E \rightarrow id.$

Taking an input string

	i/p	Action
\$0	id+id*id\$	s_3
\$0 id3	+id*id\$	r_4 $E \longrightarrow id$
\$0E1	+id*id\$	s_4
\$0E1+4	id*id\$	s_3
\$0E1+4id3	*id\$	Reduce r_4 $E \longrightarrow id$

$0E1+4E7	*id $	we choose Shift S_5
$0E1+4E7*5	id $	Shift S_3
$0E1+4E7*5id3	$	r_4 $E \rightarrow id$
$0E1+4E7*5E8	$	reduce r_1 $E \rightarrow E*E$
$0E1+4E7	$	r_1 $E \rightarrow E+E$
$0E1	$	accept

For id*id+id$

Stack	input	Action

using dangling-else Ambiguous Grammar -

S → iSeS | iS | a

stmt → if expression then stmt else stmt

stmt → if expression then stmt

stmt → for all other conditions

Construct SLR parsing table for the grammar ↑

S' → .S

S → .iSeS

S → .iS

S → .a

I0 :- closure[goto[I0, s']]

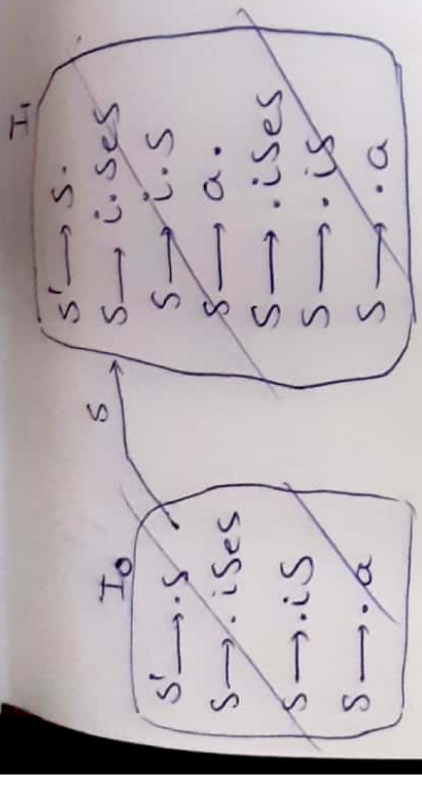
~~Action~~ ~~goto~~

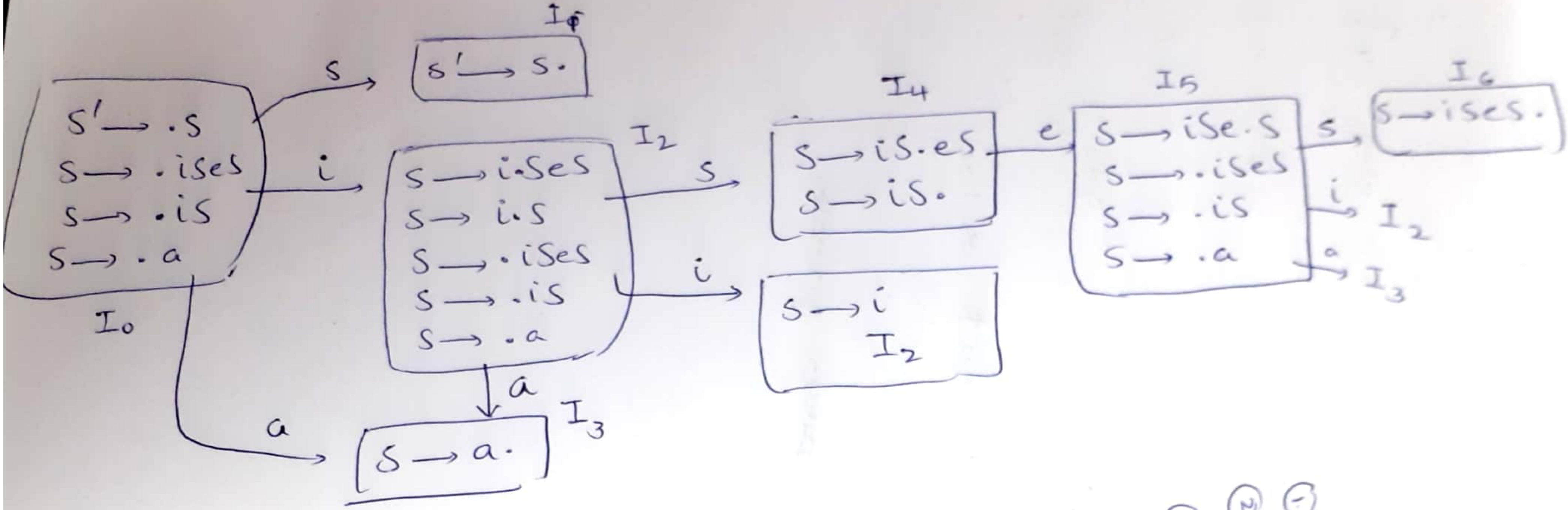
~~state id + * (~~

Parse ~~the~~ string iiaea$ as the input string:

Note :

YACC is a ~~LABL~~ LALR parser generator





① $S \to iSeS$

② $S \to iS$

③ $S \to a$

I_1, I_3, I_4, I_6

I_1 :- $S' \to S.$

I_3 :- $S \to a.$

Follow(S) =
{e, $}

I_4 :- $S \to iS.$

I_6 :-
$S \to iSeS.$

state	Action i	a	e	$	goto S
0	s_2	s_3			1
1				accept	
2	s_2	s_3			4
3			r_3	r_3	
4			s_5 or r_2	r_2	
5	s_2	s_3			6
6			r_1	r_1	

Input symbol:-

$0	iiaea$	shift S_2
$0i2	iaea$	Shift S_2
$0i2i2	aea$	Shift S_3
$0i2i2a3	ea$	Reduce r_3 S→a
$0i2i2S4	ea$	shift S_5
$0i2i2$4e5	a$	shift S_3
$0i2i2S4e5a3	$	Reduce r_3 S→a
$0i2i2S4e5S6	$	Reduce r_1 S→iSeS
$0i2S4	$	r_2 S→iS
$0S1	$	accept

YACC : ~~P~~ LALR parser generator.

translate.y → YACC compiler → y.tab.c

y.tab.c → c compiler → a.out

Input → a.out → output

Q. write a YACC compiler for desk calculator grammar for

$E \longrightarrow E+T / T$

$T \longrightarrow T*F / F$

$F \longrightarrow (E) / id$

```
%{
# include <ctype.h>
%}
% token DIGIT
%%
 line : expr '\n' {printf("%d\n", $1);}
        ;
 expr : expr'+' term {$$ = $1 + $3 ;}
        | term
        ;
 term : term '*' factor { $$ = $1 + $3 ; }
        | factor
        ;
 factor : '(' expr ')' { $$ = $2; }
        | DIGIT
        ;
 %%
 yylex()
 { int c;
   c=getchar();
   if(isdigit(c))
   { yylval=c-'0';
     return DIGIT;
   }
   return c;
 }
```

Error recovery in LR Parsers :-

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

	id	+	*	(	)	$	E
0	s_3	e_1	e_1	s_2	e_2	e_1	1
1	e_3	s_4	s_5	e_3	e_2	accept	
2	s_3	e_1	e_1	s_2	e_2	e_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	s_3	e_1	e_1	s_2	e_2	e_1	7
5	s_3	e_1	e_1	s_2	e_2	e_1	8
6	e_3	s_4	s_5	e_3	s_9	e_4	
7	r_1	r_1	s_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_3	r_3	r_3	r_3	r_3	r_3	

Step 1 : Fill all the empty entries with the reduction value.

e_1 : "Missing operand"
e_2 : "unbalanced right parenthesis"

$E \rightarrow \cdot \epsilon$

Stack	Input	Error function
$	+	Missing operand
$	*	Missing operand
$	$	Missing operand
$	)	Unbalanced Right Parenthesis

Step 2 : Fill error messages in respective operators

$E \rightarrow id$

e_3 : "Missing operator"

Stack	i/p	Error Action
$....id	id	Missing operator
$... id	(	Missing operator

Step 3: $E \rightarrow (\underset{\substack{\downarrow \\ id}}{E})$ e4 : "Missing right paranthesis"

stack	i/p	error Action
\$...(id	id	"Missing operator"
\$...(id	\$	"Missing right parenthesis"
\$...(id(	id	"Missing operator"

Unit III

Syntax Directed Translation :-

Attributes are assigned as a property for each grammar symbol in the production. These attributes are of 2 types :-

Synthesized attributes :-

If the attributes in the grammar symbols on the left side of the production are computed using the attributes in the grammar. Then, that attributes are called as synthesized attributes.

If the attributes of the grammar symbol depends on the right side of the production are computed ~~as~~ using the attributes of the grammar symbols on the left side, then, the attributes are called as inherited attributes.

Attribute grammar :- Its a syntax directed defination in which the function in semantic rule is written as expressions attached with attributes.

Production	Semantic Rules
$E \longrightarrow E + T$	$E.val \longrightarrow E_1.val + T.val$
$E \longrightarrow T$	$E.val \longrightarrow T.val$

$T \rightarrow T * F$	$T.val \rightarrow T_1.val * F.val$
$T \rightarrow F$	$T.val \rightarrow F.val$
$F \rightarrow (E)$	$F.val \rightarrow E.val$
$F \rightarrow$ digit	$F.val \rightarrow$ digit.lexval

Translating the syntactic construct in the source code using their attributes is called as directed

SDD : Syntax directed defination:-

In SDD, each grammar production $A \rightarrow \alpha$ is associated with semantic, a semantic rules of the form $b = f(c_1, c_2 \ldots c_n)$ where f is a function & b is either inherited attribute or synthesized attributed.

(i) S-attributed defination:- If SDD uses only synthesized attribute then it is called as S-attributed definition.

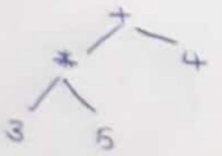
- If SDD uses, only inherited attributes, then it is called as L-attributed definations.

Consider an example:-

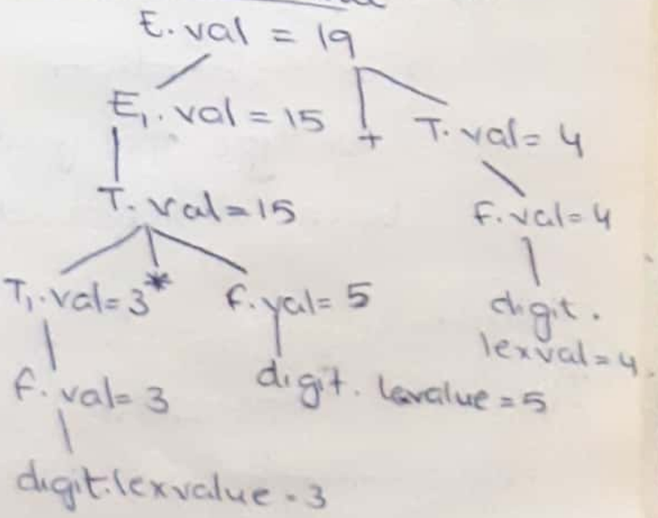
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$F \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow$ digit

then construct syntax tree, parse tree & annotated parse tree for the string $3*5+4$

Syntax tree:- based on precedence of operators



Annotated Parse Tree



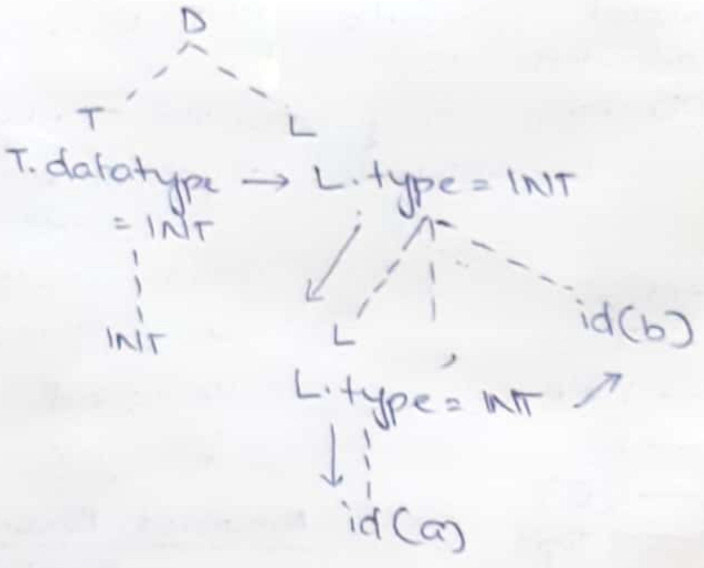
Annotated Parse tree :-

computing the parse tree, showing the values of the attributes at each node is called annotated parse tree or decorated parse tree.

Inherited attribute :

Production	Semantic Rules.
D → TL	L.type := T. datatype
T → INIT	T.datatype := INIT
T → FLOAT	T. datatype := FLOAT
T → CHAR	T. datatype := CHAR
L → L_1, id	L_1.type = L. type insert (id.entry, L. type
L → id	insert (id.entry, L type)

Annotated parse tree.
int a, b



where,

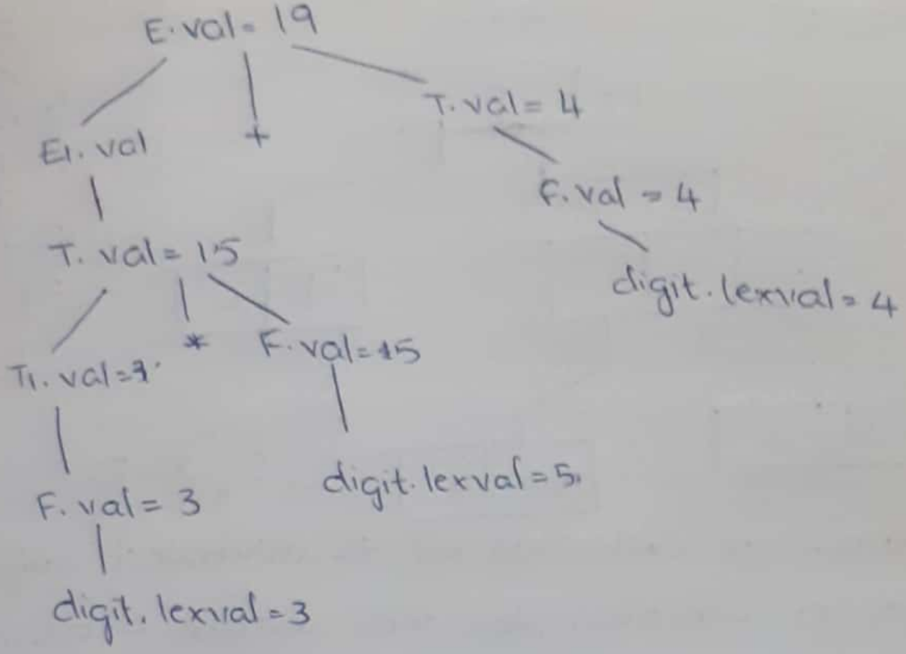
D is declaration

T is type specification

L is identifierlist

Dependency graph :-

If b is dependent on a, it is represented by a ⟶ b



construction of syntax Tree :-

3 functions :-

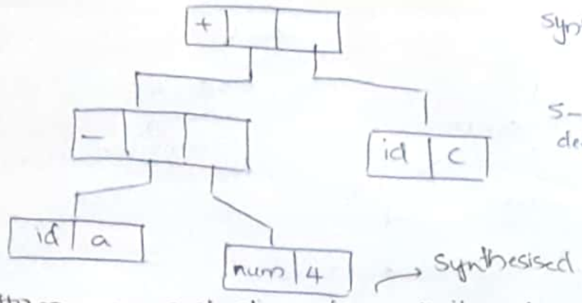
1) mknode (op, left, right)

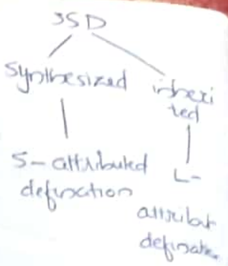
2) mkleaf (id, entry)

3) mkleaf (num, value)

Production	Semantic rule
$E \rightarrow E_1 + T$	$E.nptr = mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = mknode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow id$	$T.nptr = mkleaf('id', id.entry)$
$T \rightarrow num$	T.nptr = mkleaf('num', val~~ue~~)

Example :- a-4+c





Bottom-up evaluation of S-attributed defination:-

consider an example that reveals a grammar ~~Proof~~ which represents the implementation of desk calculator with an LR Parser

Production	Semantic Rule	codefragment
L ⟶ En	print(E.val)	print(val[top])
E ⟶ E₁+T	E.val ⟶ E₁.val+T.val	val[ntop] = val[top-2]+ val[top]
E ⟶ T	E.val ⟶ T.val	
T ⟶ T₁*F	T.val ⟶ T₁.val * F.val	val[ntop]= val[top-2] * val[top]
T ⟶ F	T.val ⟶ F.val	
F ⟶ (E)	F.val ⟶ E.val	val[ntop] = val[top-1]
F ⟶ digit	F.val ⟶ digit level	

using above code fragment translate a stmt 3+5+4n and show the translation scheme in detail

Input	State	val	Production rule
3*5+4n	–	–	
*5+4n	3	3	F ⟶ digit
*5+4n	F	3	T ⟶ F
*5+4n	T	3	
5+4n	T*	3 op	
+4n	T*5	3 op 5	F ⟶ digit
+4n	T*F	3op 5	T ⟶ T*F
+4n	T	3*5	E ⟶ T
4n	E+	15_	
n	E+4	15_4	F ⟶ digit
n	E+F	15_4	T ⟶ F
n	E+T	15_4	E ⟶ E+T
n	E	15+4	
n	E	19	
	En	19	L ⟶ En
	L	19	

—x—

Top down evaluation of attributed defination (OR) translation scheme for inherited attribute grammar

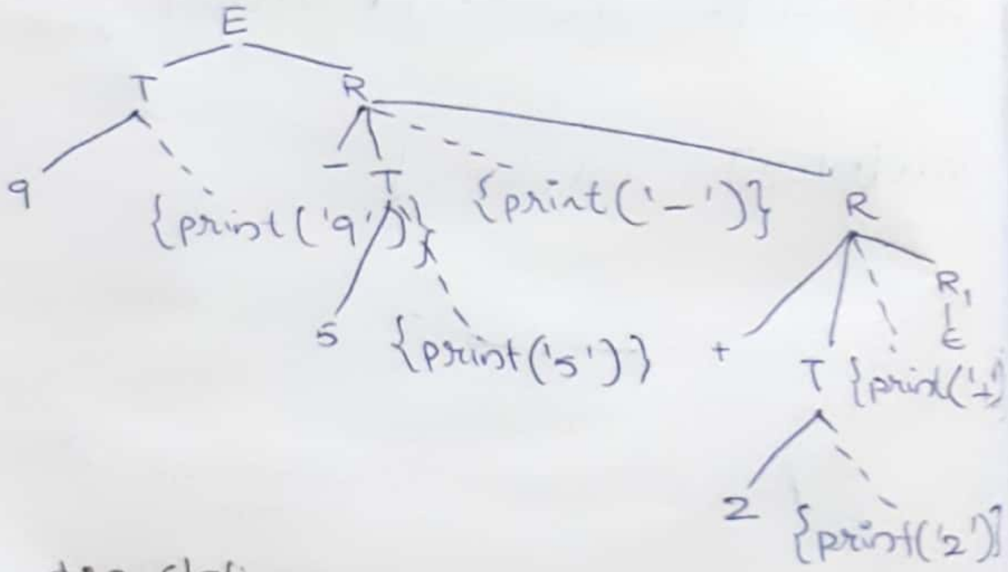
Consider an example which includes a translation scheme that maps infix expression with addition & subtraction into corresponding postfix expression.

Grammar

⊗ E → TR
R → addop T { print (addop. lexeme) } R_1/ε
T → num { print (num.val) }

Construct the parse tree for the i/p string for the i/p string 9-5+2 with each semantic action attach.

~~E → TR~~
~~R → addopT { print(addop_lexeme) } R_1/ε~~
~~T →~~



Top down translations :-

$E \rightarrow E_1 + T$ $\{E.val := E_1.val + T.val\}$

$E \rightarrow E_1 - T$ $\{E.val := E_1.val - T.val\}$

$E \rightarrow T$ $E.val \longrightarrow T.val$

$T \rightarrow (E)$ $T.val \longrightarrow E.val$

$T \rightarrow num$ $T.val \longrightarrow num.lexval$

$E \rightarrow T \ \{R.i := T.val\}$
$\quad R \ \{E.val := R.s\}$

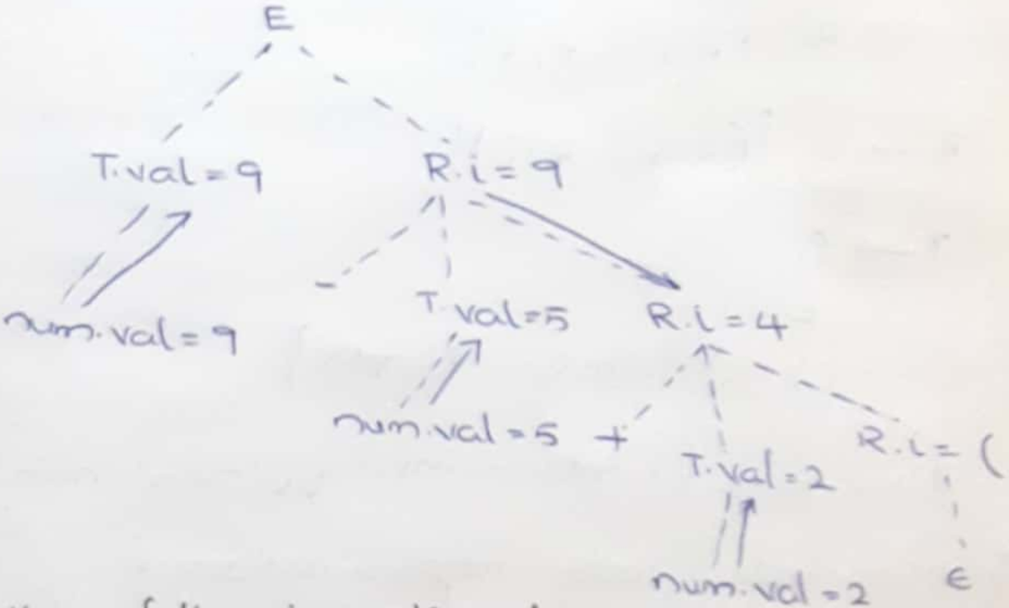
$R \rightarrow +$
$\quad T \ \{R_1.i := R.i + T.val\}$
$\quad R_1 \ \{R.s := R_1.s\}$

$R \rightarrow -$
$\quad T \ \{R_1.i := R.i + T.val\}$
$\quad R_1 \ \{R.s := R_1.s\}$

$R \rightarrow \epsilon \ \{R.s := R.i\}$

$T \rightarrow ($
$\quad E$
$\quad) \ \{T.val := E.val\}$

$T \rightarrow num \ \{T.val := num.val\}$



consider the following translation sheme to construct the syntax tree

$E \rightarrow E_1 + T / E_1$

$E \rightarrow TR / \epsilon R$

$R \rightarrow +TR / -TR / \epsilon$

~~Consider the following~~

$E \rightarrow E_1 + T$ { E.nptr = mknode(+, E_1.nptr + T.nptr)}

$E \rightarrow E_1 - T$ {E.nptr = mknode(−, E_1.nptr + T.nptr)}

$E \rightarrow T$ {E.nptr = T.nptr}

$T \rightarrow (E)$ {T.nptr = E.nptr}

$T \rightarrow$ num {T.nptr = mkleaf(num, numval)}

$E \rightarrow T$ {R.i := T.nptr}
  R {E.nptr := R.s}

$R \rightarrow +$
  T {R_1.i := mknode('+', R.i, T.nptr)}
  R_1 {R.s := R_1.s}

$R \rightarrow -$
  T {R_1.i := mknode('−', R.i, T.nptr)}
  R_1 {R.s := R_1.s}

$R \rightarrow \epsilon$ {(R.s := R.i)}

$T \rightarrow$ (
  E
) {T.nptr := E.nptr}

$T \rightarrow$ num {T.nptr := mkleaf('num', num.val)}

T → num {T.nptr := mkleaf('num', num.val)}

Intermediate code representation :-

Intermediate code can be represented in 3 ways :-

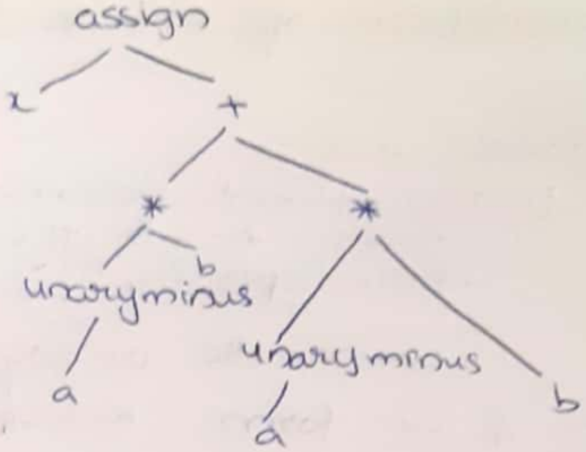
(i) Abstract syntax tree

(ii) postfix expression

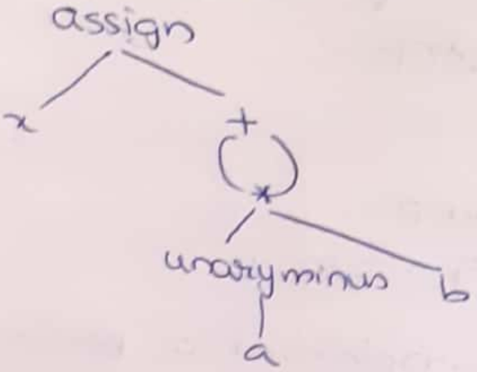
(iii) 3-address code

Consider an i/p string x := -a*b + -a*b

(1) Abstract syntax tree



There is repetation of -a*b to avoid this we use DAG (Directed acyclic graph)



(2) Postfix expression :-

(a+b) * c = ab+c *

ex: (3+4) *5

34 +5 *

(3) 3-address code :-

The reason why it is called 3-address code is, the statement usually contains 3 addresses, where 2 addresses are used to store the operands and one address to store the result

x = y * z

$\left\{\begin{array}{l} t_1 := y * z \\ x := t_1 \end{array}\right.$

3 addresses

1. one to store result

2. the rest to store operands

General representation of 3-address code :-

A := B op C

Types of 3-address code :-

1.) A := B op C
2.) A := op B
[illegible]

(1) Assignment statemen of the form A := B op C where operator can be binary arithemetic or logical operation

(2) Assignment is of the form A := op B where op is ~~of~~ a unary operator, logical negation or shift

(3) Copy statements of the form A := B

(4) Unconditional jump

goto L

(5) Conditional jump

if A relop B goto L

(6) Instruction to implement a procedure call.

param A and call P, n , where n is number of actual parameters

parm A_1
param A_2
⋮
param A_n
call P, n
return B

(7) Indexed assignments

A := B[i]
(or) A[i] := B

(8) Address & pointer assignments

A := &B
A := *B
*A := B

ex:- The 3 address code for $A = -b * (c + d)$

$t_1 :=$ uminus b

$t_2 = c + d$

$t_3 = t_1 * t_2$

$A := t_3$

ex2: $a := -b * c + -b * c$

$t_1 =$ uminus b

$t_2 = t_1 * c$

$t_3 := -b$

$t_4 = t_3 * c$

$t_5 : t_2 + t_4$

$a := t_5$

Three forms of representing 3 address code are :-

(i) Quadruples

(ii) Triples

(iii) Indirect triples

Quadruples :-

	op	arg1	arg2	result
(0)	uminus	b		t_1
(1)	*	t_1	c	t_2
(2)	uminus	b		t_3
(3)	*	t_3	c	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Triples :-

In triple representation of 3-address code we dont have result column, so the evaluation will be expressed where the results ~~evolution is expr~~ are stored in a particular position or location in the memory

2) Triples

	op	arg1	arg2
(0)	uminus	b	
(1)	*	(0)	c
(2)	uminus	b	
(3)	*	(2)	c
(4)	+	(1)	(3)
(5)	:=	a	(4)

A = B[i]

	op	arg	arg
(0)	=[ ]	B	i
(1)	:=	A	(0)

A[i] = B

	op	arg1	arg2
(0)	[ ]=	A	i
(1)	=	(0)	B

(3) Indirect Triples :-

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Triples

	op	arg1	arg2
(14)	uminus	b	
(15)	*	(14)	c
(16)	uminus	b	
(17)	*	(16)	c
(18)	+	(15)	(17)
(19)	:=	a	(18)

- It is another form of representation of 3-address code where it lists out pointers to triples such that an array statement is used in order to list out pointers to triples in the desired order

Example: write the 3 address code forms for quadruples, triples, indirect triples for

A := -B * (C+D)

t_1 := uminus b

t_2 := c + d

t_3 := $t_1 * t_2$

A := t_3

(1) Quadruples :

	op	arg1	arg2	result
(0)	uminus	b		t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3		A

(2) Triples :-

	op	arg1	arg2
(0)	uminus	b	
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	A	(2)

(3) Indirect Triples :-

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)

	op	arg1	arg2
(14)	uminus	b	
(15)	+	c	d
(16)	*	(14)	(15)
(17)	:=	A	(16)

conversion of popular programming language constructs in 3-Address code

(1) Assignment statements

$v_1 = (v_2 - v_3) * (v_2 + 2 * v_3)$

$t_1 := v_2 - v_3$

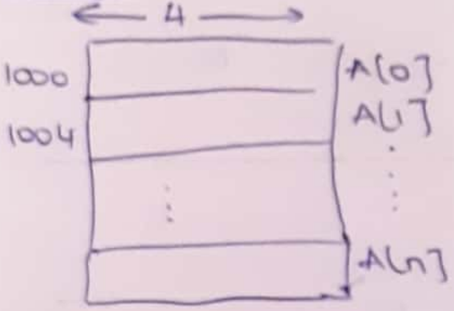
$t_2 := 2 * v_3$

$t_3 := v_2 + t_2$

$t_4 := t_1 * t_3$

$v_1 := t_4$

(2) Arrays :-



To find the location of A[i] the formula is

base + i*w
(w = width)

1008 = 1000 + 2*4

eg① int a[10];
Stmt : P= a[5];

Three Address Code

$t_0 := 5*4$

$t_1 := \&a$

$t_2 := t_1[t_0]$

$P := t_2$

eg② int a[2][3]

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 00 & 01 & 02 \\ 1 & 10 & 11 & 12 \end{array}$$

	Row Major Form		Column Major Form
1st row	a[0][0]	1st col	a[0][0]
	a[0][1]		a[1][0]
	a[0][2]	2nd col	a[0][1]
$	a[1][0]		a[1][1]
2nd row	a[1][1]	3rd col	a[0][2]
	a[1][2]		a[1][2]

datatype $a[d_1][d_2]$
$a[i_1][i_2]$

Row major form
$\left\{ \begin{array}{l} base + (i_1 * d_2 * w + i_2 * w) \\ \text{or} \\ base + (i_1 * d_2 + i_2)\, w \end{array} \right.$

column major form
$\left\{ \begin{array}{l} base + (i_1 * w + i_2 * w * d_1) \\ \text{(or)} \\ base + (i_1 + i_2 * d_1) * w \end{array} \right.$

eg:- int $a[\overset{d_1}{20}][\overset{d_2}{30}]$
$P = a[\underset{i_1}{5}][\underset{i_2}{9}]$

Row:- $base + (5 * 30 + 9) * 4$
$base + (159) * 4$
$base + 636$

int $a[\overset{d_1}{20}][\overset{d_2}{30}]$
Code
$P = a[5][9]$

Intermediate code form
$t_0 := 30 * 4$
$t_1 := t_0 * 5$
$t_2 := \&a$
$t_3 := 9 * 4$
$t_4 := t_1 + t_3$
$t_5 := t_2[t_4]$
$P := t_5$

eg:- int $a[20][30]$
$a[4][3] = h$

Row:- $base + (4 * 30 + \overset{3}{9}) * 4$

Intermediate code form
$t_0 := 30 * 4$
$t_1 := t_0 * 4$
$t_2 := \&a$
$t_3 := \overset{3}{\cancel{4}} * 4$
$t_4 := t_1 + t_3$

Pointers & Address Assignments :-

① P = &x ;
*P = 40;

t_0 := &x ;
P:= t_0
P[0] := 40;

} Intermediate code

② P = &x ;
y = *P ;

t_0 := &x
P := t_0
t_1 := P[0].
y := t_1

③ P = &a[5];
*P = 50;

t_0 := 5*4
t_1 := &a
t_2 := t_0 + t_1
P := t_2
P[0] := 50

Record Access :-

structures :-

struct Students
{
int stud_id;
int age ;
}*s_1 ;
⋮

s_1.age := 18;

$s_1 \longrightarrow$ age :=18 ;
↓
t_0 := &s_1 ;
t_1 := 4 .
t_0[t_1] := 18

} Intermediate code form

Intermediate code

t_0 := &s_1;
t_1 := 4 ;
t_0[t_1] :=18;

(or)

t_0 := &s_1 ;
t_0[4] := 18 ;

Flow of control statements & boolean expressions

(1) If-else statements

①

```
if (a>b)
      c=40;
  else
      c:=30;

  x=50;
```

```
if (a>b) goto label L0
 goto L1
 Label L0
    C:=40
    goto label L2
  Label L1
     c:=30
     goto label L2
  Label L2
     x:=50;
```

②

```
if(a>b) || (b>c)
      a=40;
  else
      a:=20;
```

Intermediate code form

```
if (a>b) || (b>c) goto L0
 goto L1
 Label L0
     a:=40
 Label L1
     a:=20
```

2. switch case statement :-

```
switch (ch)
{
  case 1:  a=b;
           c=d;
           break;
  case 2:  e:=f;
           g:=h;
           break;
  default : a=b;
            c=d;
            break;
}
p=q;
```

```
to := ch
goto label L3
Label L0
   a:=b
   c:=d
goto label L4
Label L1
   e:=f
   g:=h
   goto L4
Label L2
   a:=b
   c:=d
   goto L4
Label L3
if t0==1 goto L0
if t0==2 goto L1
goto L2
Label L4
  p:=q
```

Loop statements :-

```
while (i>0)
{
  val := val*i ;
   i:=i-1;
}
```

```
Label L0
   if(i>0) goto L1
   goto L2
Label L1
   t0 := val * i
   val := t0
   t1:=i-1
   i:=t1
 goto L0
Label L2
```

Procedure call (parameters)

1.) $p = sum(n_1, n_2)$

Intermediate code form :

calling sequence & return

```
param n1
param n2
call sum 8
```

↘ where 8 is size of n_1 + size of n_2

For returning

```
return t0;
P:= t0
```

ex :-
2)

```
sum (int n1 ,int n2)
{
  int s;
  s= n1+n2;
  return(s);
}
```

Intermediate code :

```
TAC : proc- begin sum
        t0:=n1+n2
        s := t
        returns
        goto L0
        Label L0
          proc- end sum
```

Unit - IV

Run time Environments

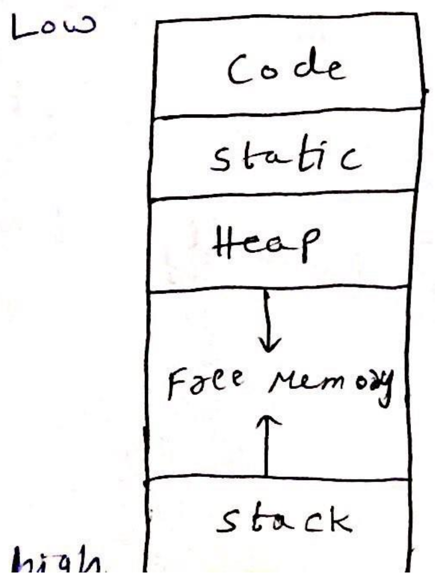
Storage Organization

→ The management and orgnization of this logical address space is shared between the Compiler, operating system and target Machine.

→ The operating system maps the logical addresses into physical addresses, which are usually spread throughout Memory.

→ The runtime representation of an object program in the logical address space consists of data and program areas.

Typical Subdivision of run-time Memory into code and data areas



→ Runtime storage comes in blocks of contigous bytes, where a byte is the smallest unit of addressable memory.

→ A byte is eight bits and four bytes form a machine word.

→ Multiple byte objects are stored in consecutive bytes and given the address of the first byte.

→ The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

→ Space left unused due to alignment considerations is refered to as padding.

→ The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area Code, (usually low end of memory)

→ Similary, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time and these data objects can be placed in another statically determined area called static.

→ One of the reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.

→ To maximize the utilization of space at run time, the other two areas, stack and heap are at the opposite ends of remainder of address space.

→ These areas are dynamic, their size can change as the program executes.

→ These areas grow towards each other as needed.

→ The stack is used to store data structures called activation records that get generated during procedure calls.

→ stack grows towards lower addresses, heap towards higher.

Storage Allocation ~~st~~ (static vs Dynamic)

→ The layout and Allocation of data to Memory locations in the runtime environment are key issues in storage management.

→ storage alloction decision is static, if it can be made by compiler looking only at text of program, not at what program does when it executes.

→ storge allocation decision is dynamic, if it can be decided only while the program is running.

→ stack storage - Names local to a procedure are allocated space on stack. The stack supports the normal call/return policy for procedures.

→ Heap storage : Data that may ~~outlive~~ outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage. The heap is area of virutal memory that allows ~~ob~~ objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

Storage Allocation ~~stra~~ strategies

→ There are three different type of storage allocation strategies based on the division of runtime storage. These are

(i) static Allocation - It is ~~fall~~ for all data objects at compiletime

(ii) stack Allocation - In the stack allocation a stack is used to manage runtime storage

(iii) Heap Allocation - In heap allocation the heap is used to manage dynamic memory allocation.

(i) Static Allocation

→ The size of data objets is known at compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static Allocation.

→ The binding of name with the amount of storage allocated donot change at run time.

→ In static Allocation the compiler can determine the amount of storage required by each data object. Therefore it becomes easy for a compiler to find the addresses of these data in the activation record.

→ At compile time compiler can fill the addresses at which the target code can find the data it operates on.

→ FORTRAN uses static Allocation strategy.

Limitations of static Allocation

→ The static allocation can be done only if the size of data object is known at compile time.

→ The data structures can not be created dynamically. The static allocation cannot manage the allocation of memory at runtime.

→ Recursive procedures are not supported by this type of allocation.

(ii) stack Allocation

→ Stack Allocation strategy is a strategy in which the storage is organized as stack. This stack is also called control stack.

→ As activation begins, the activation records are pushed onto the stack and on completion of this activation record the corresponding activation records are popped.

→ The locals are stored in each activation record. Hence locals are bound to corresponding activation record on each fresh activation.

→ The data structures can be created dynamically for stack allocation.

Limitations of stack allocation

→ The memory addressing can be done using pointers and index registers. Hence this type of allocation is slower than static Allocation.

Heap Allocation

→ If the values of non-local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its Last In First out nature. For retaining of such local variables Heap Allocation strategy is used.

→ The Heap allocation allocates the continous block of memory when required for storage of activation records (or) other data object.

→ This allocated memory can be deallocated when activation ends.

→ The deallocated (free) space can be further reused by heap manager.

→ The efficient heap Management can be done by

(i) creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.

(ii) Allocate the most suitable block of memory from the Linked List (i.e) use best fit technique for allocation of block.

Stack Allocation of space

→ ~~All~~ → Almost all compilers for Languages that use procedures, functions (or) methods as units of user-defined actions manage atleast part of their runtime ~~memory~~ as stack.

→ Each time a procedure is called, space for its local variables is pushed onto a stack, and when procedure terminates, that space is popped off the stack

Activation Trees

→ stack allocation would not be feasible if procedure calls (or) activations of procedures, did not nest in time.

Eg: quick sort (recursive)

```
int    a[11];

void   readArray(){ /* Read 9 integers into
                                        a[1]....a[9]*/
      int i;
      ....
      }

int   partition (int m, int n)
{
/* Picks a separator value v, and partitions
   a[m..n] so that a[m..p-1] are less than
   v, a[p]=v , and a[p+1..n] are equal to
   (or) greater than v. Returns p. */
   ....
  }
void quicksort( int m, int n)
 {
  int i;
   if(n>m)
   {
    i= partition (m,n);
    quicksort(m,i-1);
    quicksort(i+1,n);
   }
 }
```

```
main()
{
readArray();
a[0] = -9999;
a[10] = 9999;
quicksort(1,9);
}
```

→ Sequence of calls that might result from an execution of the program.

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter partition(1,3)
         - - - -
         - - - -
         leave partition(1,3)
         enter partition(5,9)
          - - - -
          - - -
          leave partition(5,9).
    leave quicksort(1,9)
leave main()
```

→ In this example, as it is true in general, procedure activations are nested in time.

→ If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end.

There are three common cases:

(i) The activation of q terminates normally. Then the control resumes just after the point of p at which the call to q was made.

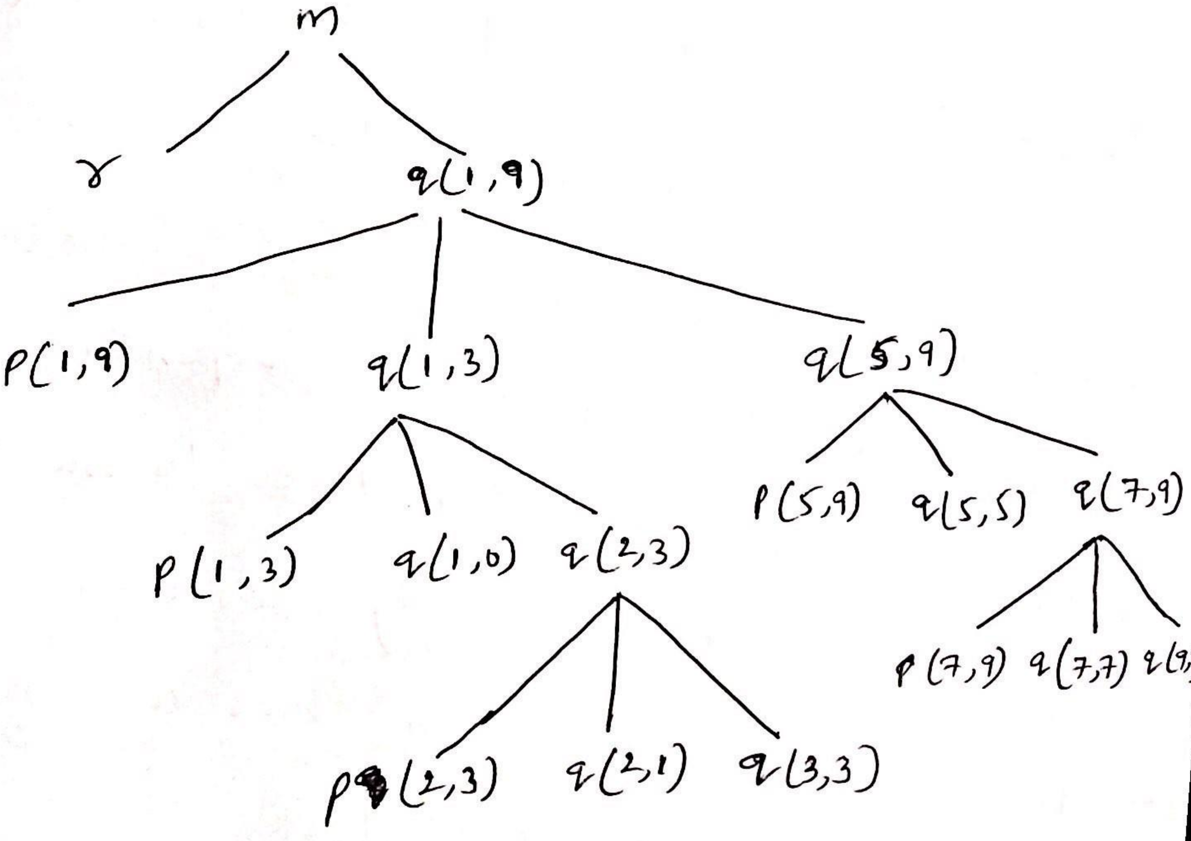
(ii) The activation of q, or some procedure q called, either directly or indirectly, aborts ie it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

(iii) The activation of q terminates because of an exception that q cannot handle. Procedure p may handle the exception, in which case the activation of q has terminated while the activation of p continues.

→ we represent the activations of procedures during the running of entire program by a tree, called Activation Tree.

→ Each node corresponds to one activation, the root is the activation of the "main" procedure that initiates execution of program.

→ At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.



Activation tree representing calls during an execution

→ The sequence of procedure calls correspond to a preorder traversal of the activation tree

→ The sequence of returns corresponds to a postorder traversal of the activation tree

Activation Records

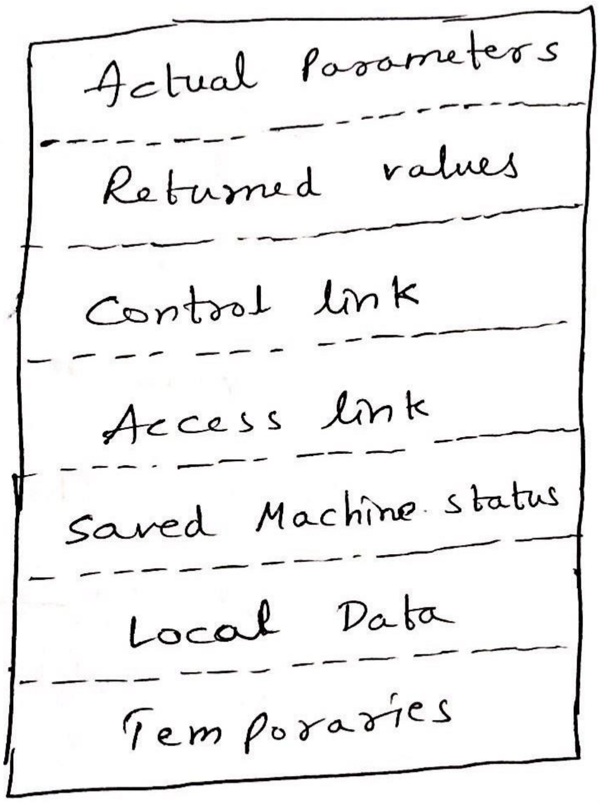
→ procedure calls and returns are usually managed by a runtime stack called the control stack.

→ Each live activation has an activation Record. on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.

→ The latter activation has its record at the top of the stack.

→ The Activation Record is a block of memory used for managing information needed by a single execution of a procedure

→ The contents of activation records vary with the language being implemented.



Actual Parameters
Returned values
Control link
Access link
Saved Machine status
Local Data
Temporaries

A general Activation Record

→ Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

→ Local data belonging to the procedure whose activation record this is.

→ A saved Machine status, with information about the state of the machine Just before the call to procedure. This information typically includes the return address and the contents of registers that were used by calling procedure and that must be restored when the return occurs.

→ An "access link" may be needed to locate data needed by the called procedure. This field refers to nonlocal data in another activation record. This field is also called static link field.

→ A "control link", pointing to the activation Record of the caller. It points to activation record of calling procedure. This link is also called dynamic link.

→ space for return value of the called function

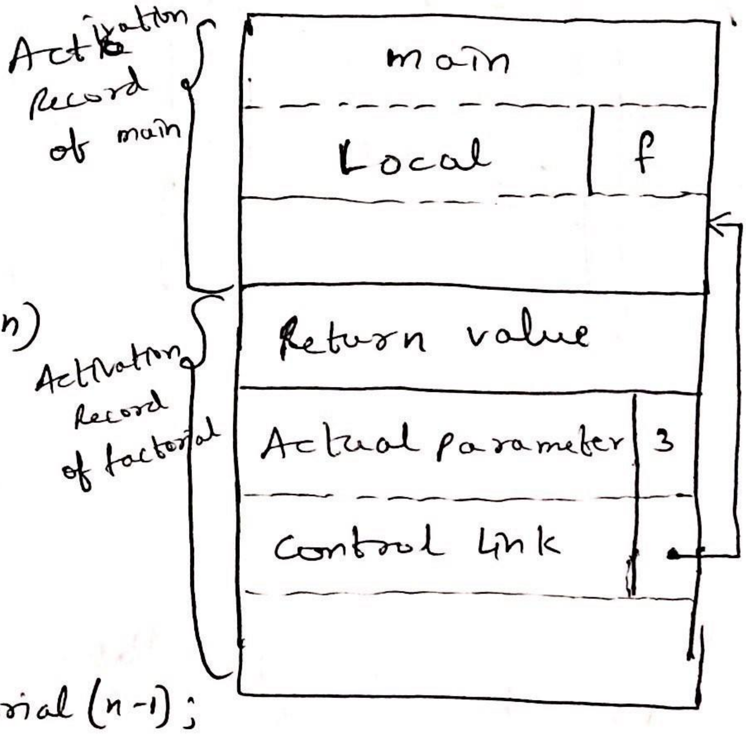
→ The actual parameters used by the calling procedure.

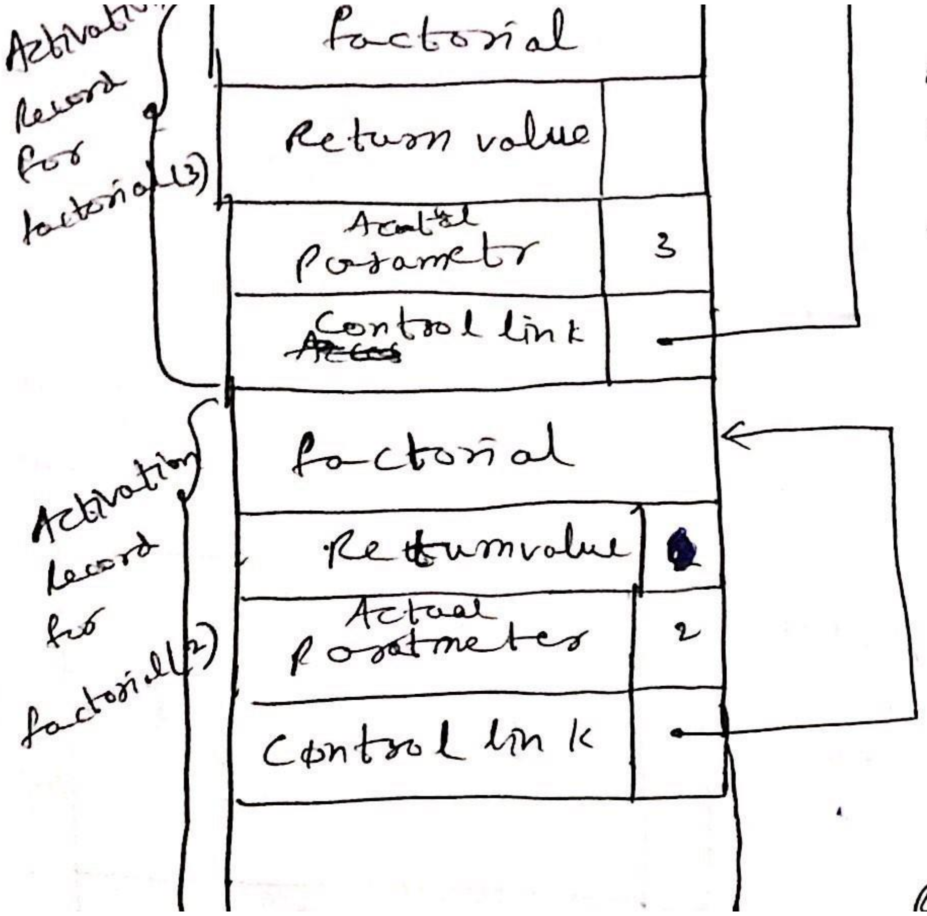
→ The size of each field of activation record is determined at the time when a procedure is called.

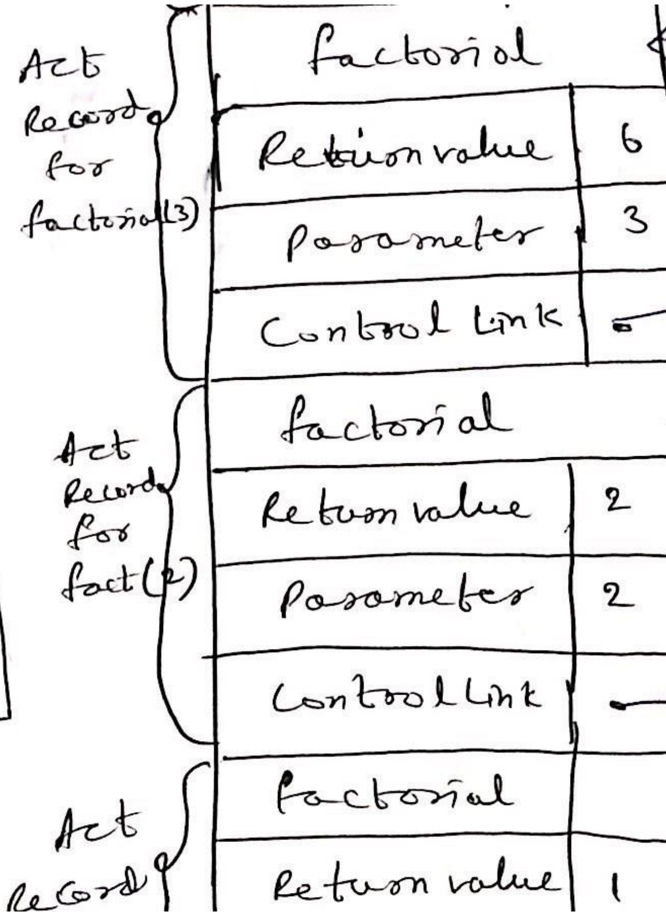
Eg:-

```
main()
{
int f;
f= factorial (3);
}
int factorial(int n)
{
if (n==1)
 return 1;
else
 return n* factorial (n-1);
}
```

First call Activation Record







Scanned with CamScanner

Eg:- Suppose a procedure A calls the procedure B (callee procedure). ~~begins after the array~~.

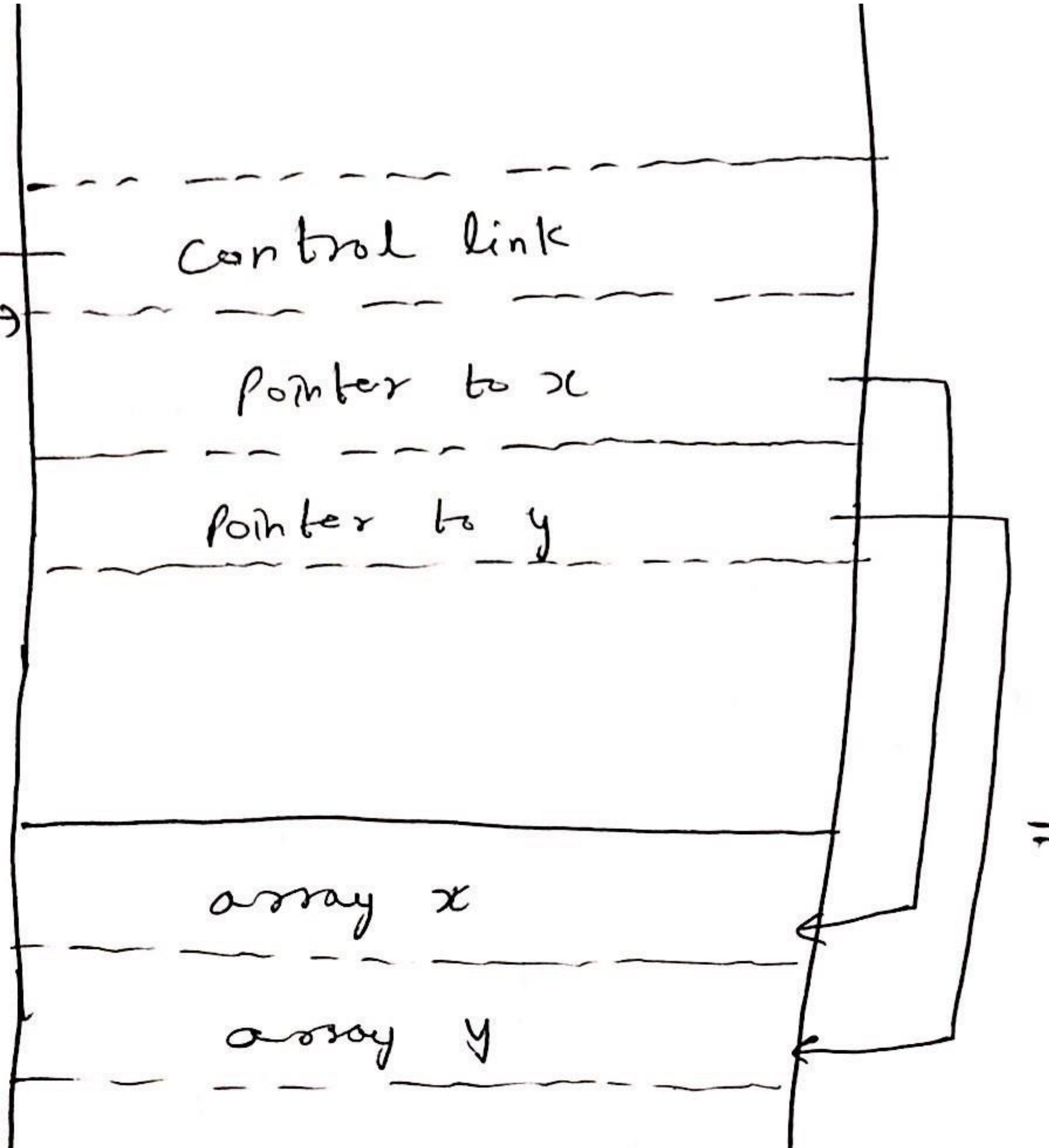
→ procedure A has two arrays x & y. The storage of these arrays is not the part of activation ~~record~~ record of A.

→ In the activation record of A only the pointers to the beginning of x and y are appearing we can obtain the relative addresses of these arrays at compile time.

→ The activation Record ~~of~~ for procedure B begins after the arrays of A. Suppose the procedure B has variable length arrays p and q. Then after the activation record of B the arrays for procedure B can be placed.

→ Two pointers can be maintained top and top-sp to keep track of these records.

→ The top points to actual top of the stack and top-sp points to end of some field in the activation record.


control link
pointer to x
pointer to y
array x
array y

Access to ~~the~~ Non-Local Data on the stack.

→ The storage Allocation can be done for two types of data variables.

(i) Local data

(ii) Non-Local data

→ The Local data can be handled using activation record whereas non local data can be handled using scope information

→ The block structured storage allocation can be done using static scope or lexical scope.

→ The Non block structured storage allocation can be done using dynamic Scope.

→ Local data

→ The Local data can be accessed with the help of activation record.

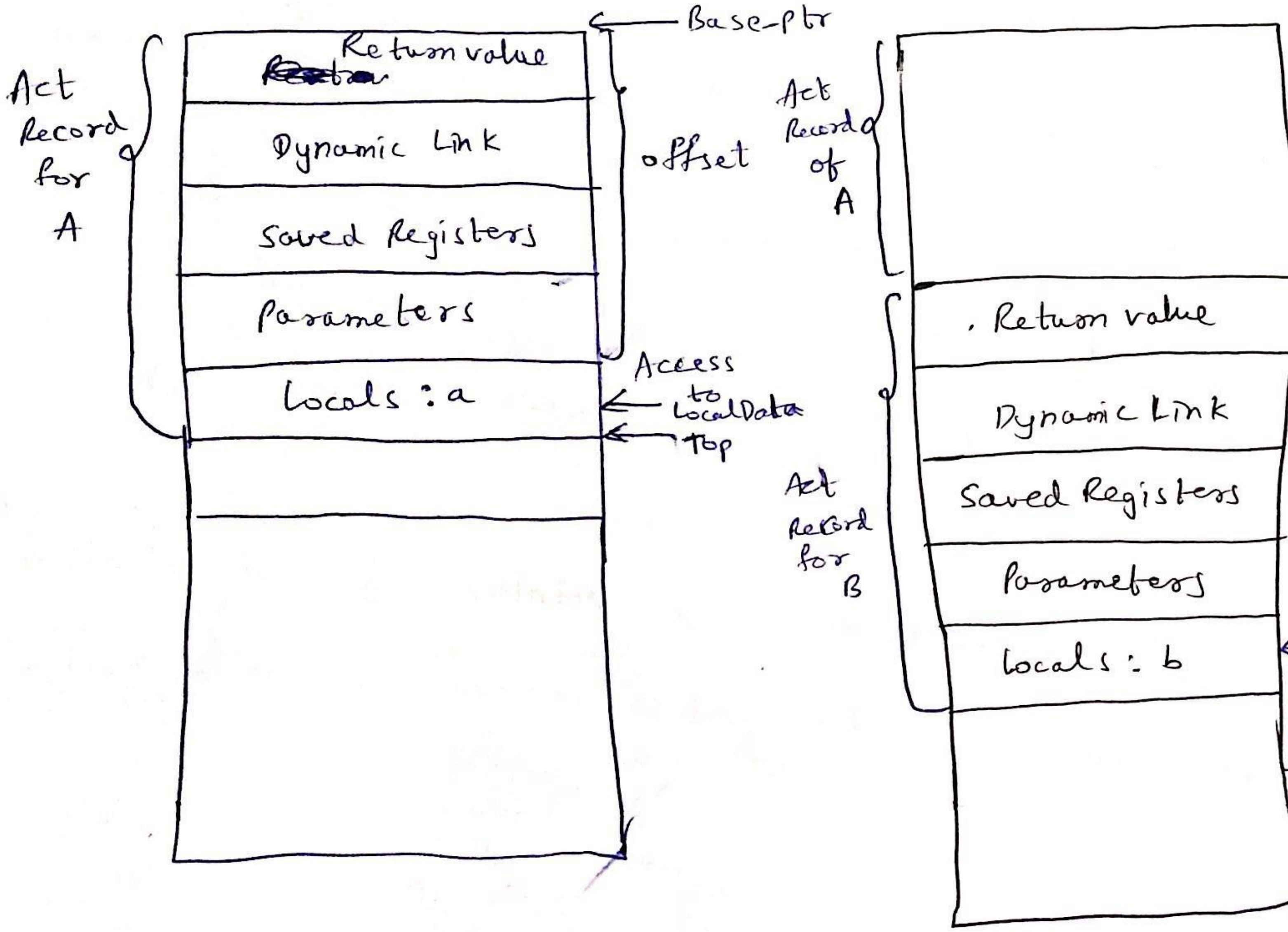
→ The offset relative to base pointer of an activation record points to local data variables within activation record.

Eg :-

```
Procedure A
    int a;
Procedure B
    int b;
body of B;
body of A;
```

The contents of Stack along with base pointer a

offset

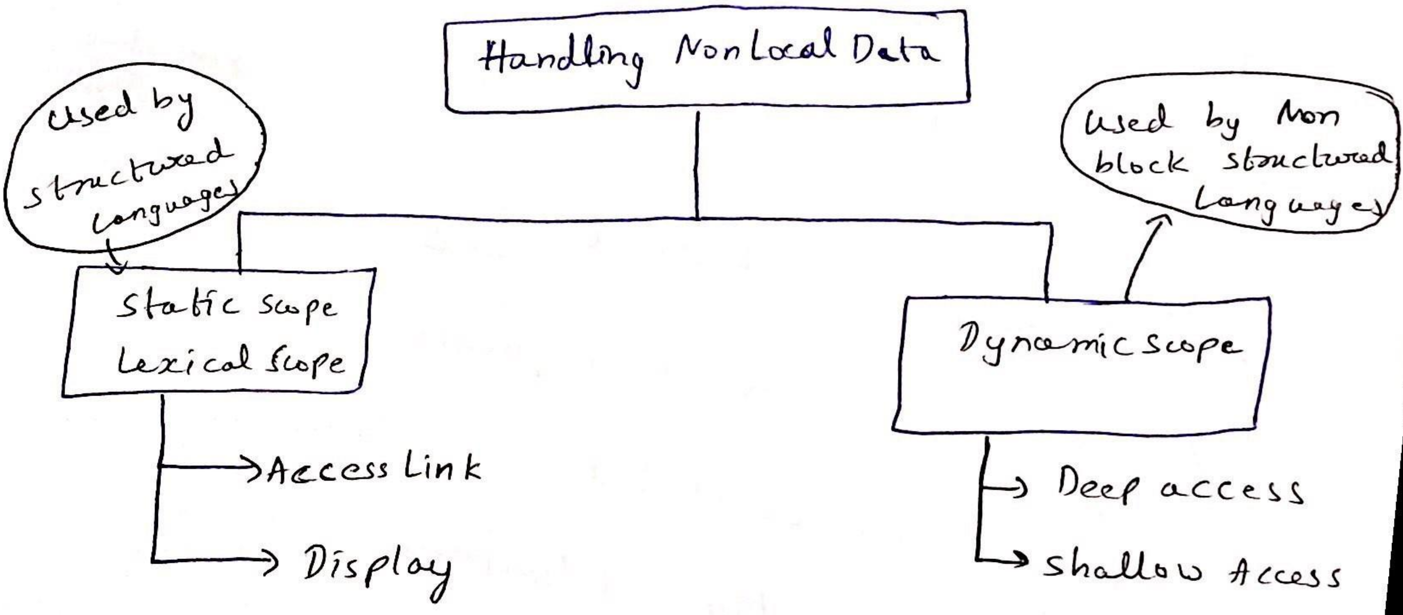


Access to Non Local Names

→ A procedure may sometimes refer to variables which are not local for it. Such variables are called as Non local variables.

→ For the Non-local Names there are two types of scope rules that can be defined

(i) static
(ii) dynamic



(i) Static Scope Rule

→ It is also called as lexical scope. In this type the scope is determined by excamining the Program text.

→ PASCAL, C and ADA use static scope rule.

→ These Languages are also called as Block structured Languages.

(ii) Dynamic Scope rule

→ For non block structured Languages this dynamic scope allocation rules are used.

→ The dynamic scope rule determines the scope of declaration of names at runtime by considering the current activation.

Eg:- LISP and SNOBOL use Dynamic Scope rule.

Static Scope (or) Lexical Scope

→ Block :- It is a sequence of statements containing the Local data declarations and enclosed with in the delimiters.

Eg:-
```
{
    Declaration statements;
    .....
}
```

→ The delimiters mark the beginning and end of the Block. The Blocks can be in nesting fashion that means block B_2 completely can be inside the Block B_1.

→ The scope of declaration in a Block structured Language is given by most closely nested loop (or) static rule.

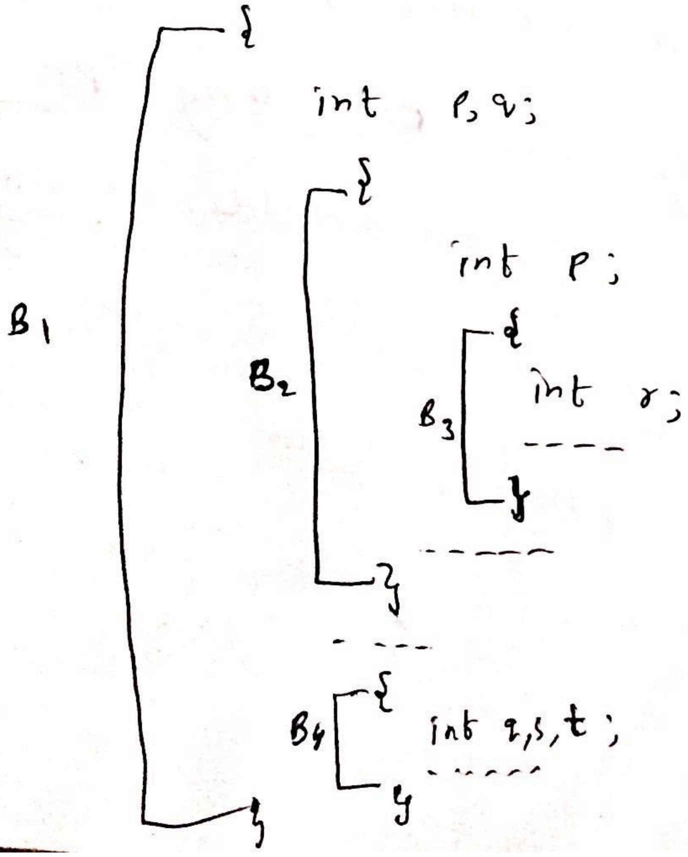
→ The declarations are visible at a program point are:

(i) The declarations that are made Locally in the procedure.

(ii) The names of all enclosing procedures

(iii) The declarations of names made immediately within such procedures.

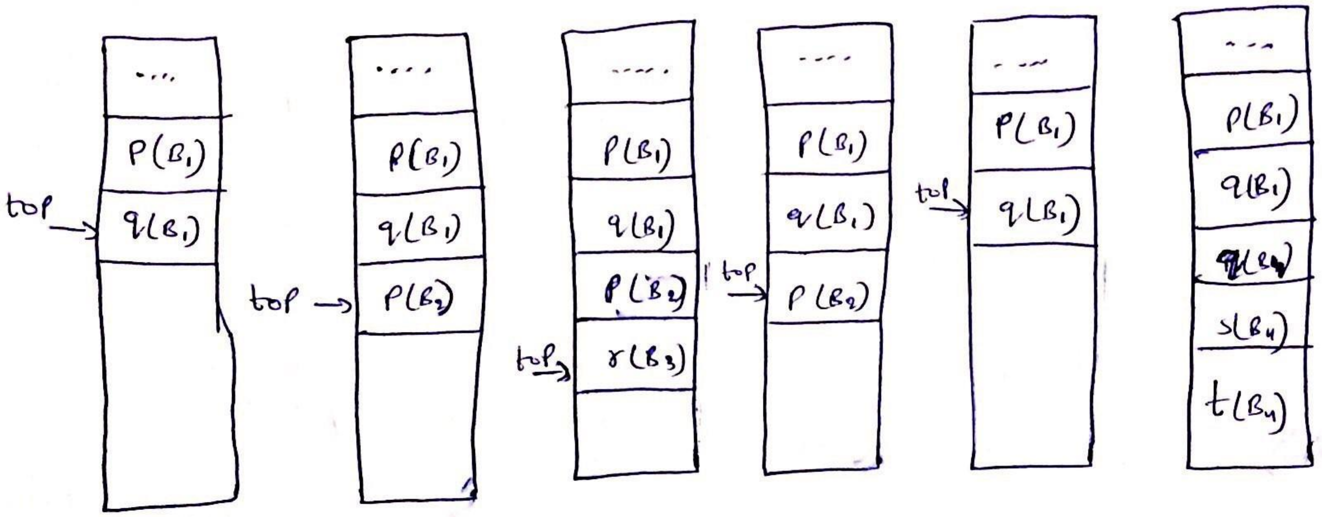
Eg:- Scope_test ()



```
{
   int p,q;
      {
         int p;
            {
               int r;
               ----
            }
         -----
      }
   ----
      { int q,s,t;
        ......
      }
}
```

→ The storage can be allocated for a complete procedure body at one time.

→ The storage for the names corresponding to particular block can be as shown below.



Lexical Scope for Nested procedure

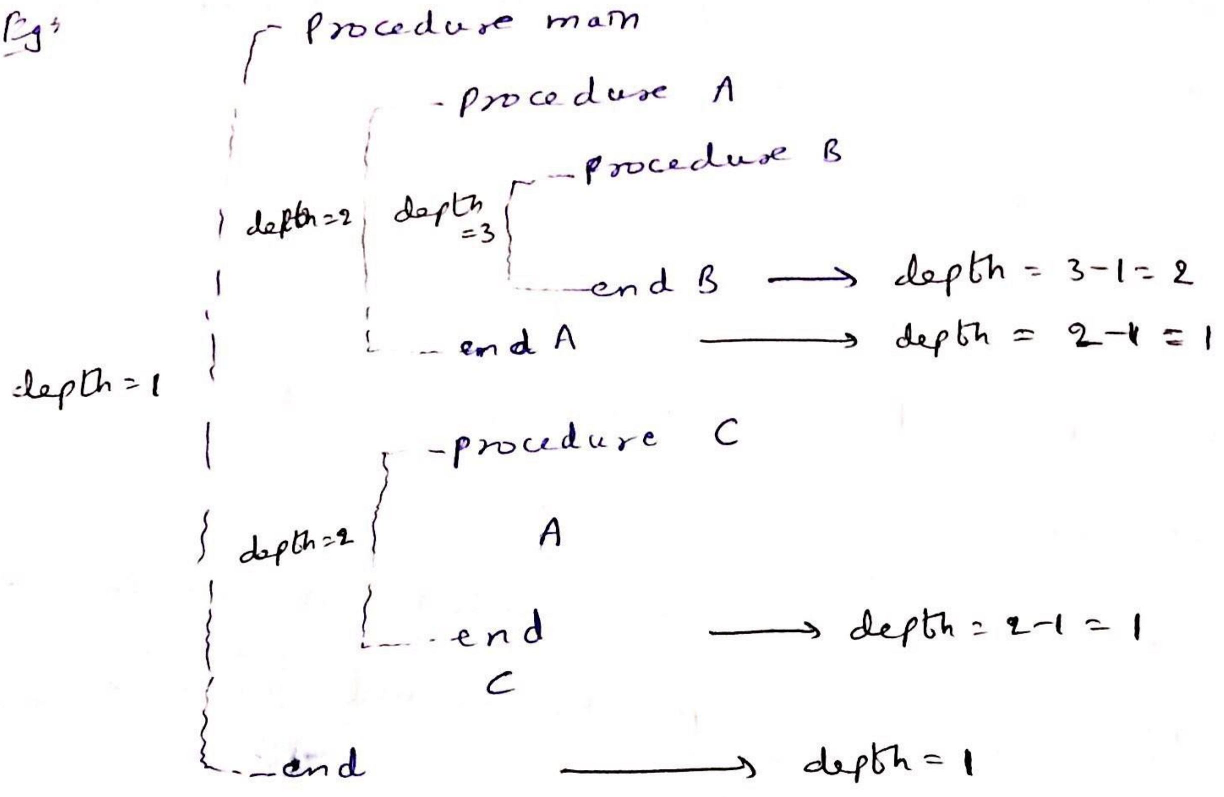
→ Nested procedure is a procedure that can be declared within another procedure.

→ A procedure Pi, can call any procedure that is its direct ancestor or older siblings of its direct ancestor.

→ The nested procedures can be as shown

Procedure main
Procedure P1
" P2
" P3
" P4

Eg:



→ Nesting depth - Nesting depth of a procedure is used to implement lexical scope. The Nesting depth can be calculated as follows:

(i) The nesting depth of main program is 1

(ii) Add 1 to depth each time when a new procedure begins

(iii) Subtract 1 from depth each time when you exit from a nesting procedure.

(iv) The variable declared in specific procedure is associated with nesting depth.

→ The lexical scope can be implemented using Access links and Displays

Access Link

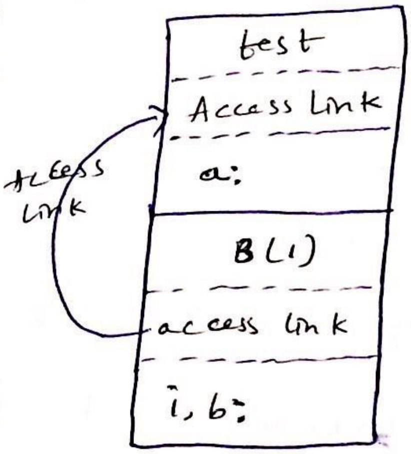
→ The implementation of lexical scope can be obtained by using pointer to each activation record.

→ These pointers are called Access links.

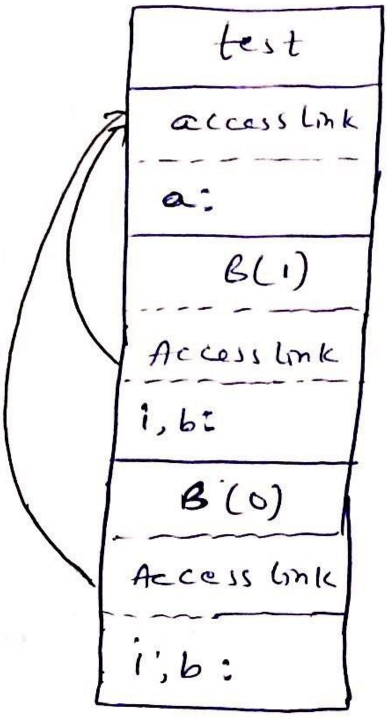
If a procedure p is nested within a procedure q then access link of p points to access link of most recent activation record of procedure q.

Eg:
```
program test;
      var a: int;
         procedure A;
            var d: int;
               begin a:=1, end;
         Procedure B (i: int);
            var b: int;
               procedure C;
                  var k: int;
                  begin A; end;
                  begin
```

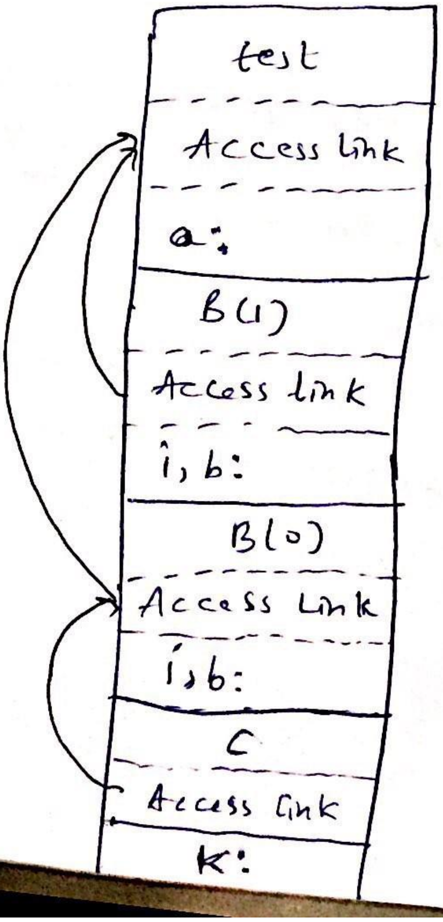
```
        if (i<>0) then B(i-1)
        else C;
        end
    begin B(1); end;
```



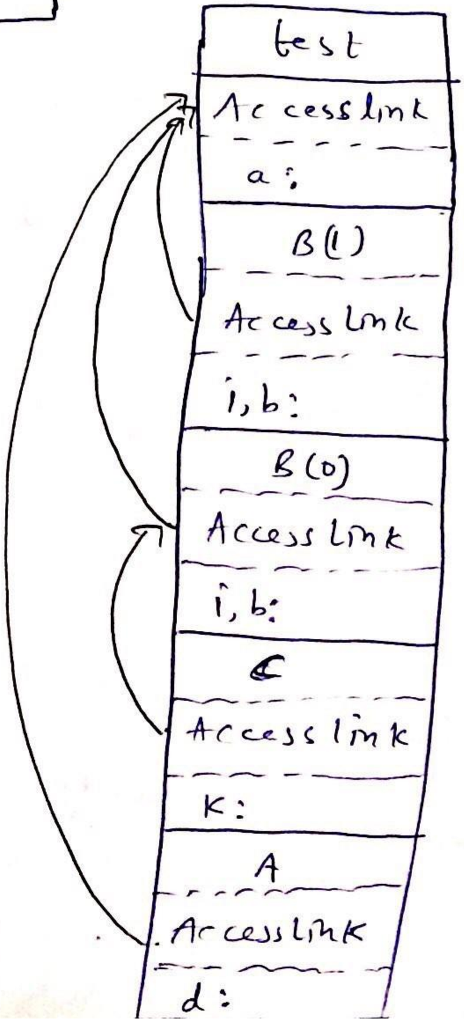
(a)



(b)



(c)



→ To set up the access links at compile time:

(i) if procedure A at depth n_A calls procedure B at depth n_B then

Case 1: if $n_A < n_B$, then B is enclosed in A and $n_A = n_B - 1$

Case 2: if $n_A \geq n_B$, then it is either a recursive call or calling a previously declared procedure.

(ii) The access link of activation record of procedure A points to activation record of procedure B where the procedure B has procedure A nested within it.

(iii) The activation record for B must be active at the time of pointing.

(iv) If there are several activation records for B then the most recent activation record ~~with~~ will be pointed.

→ Thus by traversing the access links, Non Locals can be correctly accessed.

A Basic Mark and sweep Collector

→ Mark-and-sweep garbage collection algorithms are straight forward, stop-the world algorithms that find all the unreachable objects, and put them on the List free space

→ Algorithm ~~marks~~ "visits" and "marks" all reachable objects in the first tracing step.

→ "Sweeps" the entire heap to free up unreachable objects.

Mark-and-sweep garbage collection.

Algorithm : Mark-and-sweep

Input: ~~The~~ A Root set of objects, A heap, and a freeList, called Free, with all unallocated chunks of the heap.

All chuncks of space are marked with boundary tags to indicate their free/used status and size.

ouput: A modified ~~Free~~ Free list after all the garbage has been removed.

METHOD:

→ The algorithm uses several simple data structures.

→ List Free holds objects known to be free.

→ A list called unscanned, holds objects that we have determined are reached, but whose successors we have not yet considered. i.e, we have not ~~scan~~ scanned these objects to see what other objects can be reached through them.

→ The unscanned list is empty initially.

→ ~~Addit~~ Additionally, each object includes a bit to indicate whether it has been reached (the reached bit).

→ Before the Algorithm begins, ~~[illegible]~~ all allocated objects have the reached bit '0'.

Marking Phase

```
set the reached bit to 1 and add to list unscanned each object referenced
by the root set;
while (unscanned $\neq \phi$) {
remove some object o from unscanned;
for (each object o' referenced in o) {
        if (o' is unreached; i.e it reached bit is 0) {
                set the reached bit of o' to 1;
                put o' in unscanned;
        }
    }
}
```

Sweeping phase

```
11)  Free = ∅
12)  for (each chunck of memory o in heap) {
13)      if (o is unreached, i.e its reached bit is 0 zero)
             add o to Free;
14)      else set reached bit of o to zero
15)  }
```

→ In Line (1) of Marking phase, we initialize unscanned list by placing there all the objects referenced by the root set.

→ The reached bit for all these object is also set to 1.

→ Lines (2) to (7) are in loop, in which we, in turn, examine each object o this is ever placed in Unscanned list.

→ The for-loop of lines (4) to (7) implements scanning of objects o.

→ we examine each object o' for which we find a reference within o.

→ If o' has already been reached (if reached bit is 1), then there is no need to do anything about o'; it has either been scanned previously, (or) its is ~~not~~ on unscanned list to be scanned later.

→ If o' was not reached already; then we need to set its reached bit to 1 in line (6) and add o' to the unscanned list in line (7).

→ Lines (11) to (14), the sweeping phase, reclaim the space of all the objects ~~that~~ remain unreached at the end of the marking phase.

→ Note that these will include any objects that were on the Freelist originally.

→ Because the set of unreached objects cannot be enumerated directly, the algorithm sweeps through entire heap.

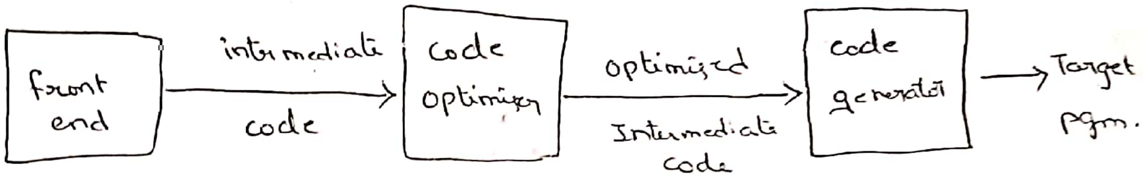
→ Line (13) Puts free and unreached objects on the Free List, one at a time.

→ Line (14) handles the reachble objects and we set their reached bit to zero (0), to maintain the proper predictions for the next execution of the garbag collection Algorithm.

unit- 8

Object code forms :-

The final phase in compiler design is code generator. The i/p is optimized intermediate code of the i/p source pgm & output ~~of~~ from CG (code generator) is a equivalent target pgm.



The o/p of the code generator is Target pgm or object code. It may be in different forms. Like

~~Lik~~ ① Absolute machine language :- This code can be placed in a fixed memory location and immediately executed.

Ex :- Example of compiler which produce target code in the Absolute machine language are "WATFIV" & PL/C.

② Relocatable machine code :- In this form, we can allow Subprogram to be compiled Separately. But we need to link together all such seperately compiled sub-programs by linking and we need to load them for execution, for this we need loader.

If the target machine can't able to do all this automatically, the compiler provide explicit relocation information to the loader to link the seperately compiled sub-prgms.

③ **Assembly code** or **Assembly language code** :-

If target code is in the form assembly-language code which contain symbolic instructions & use macro facilities. makes code generation easier.

~~If the o/p is Assembly language code~~

This type of o/p is applicable for target machine with small memory.

Machine - Dependent Code Optimization :-

The code target code generated by the code generator is optimized in order to improve speed & size reduce the amount of memory required to execute.

So, the code optimization is based on the machine in which code is going to execute.

It depends on the Target machine

1. Memory management
2. Instruction Selection
3. Register Allocation.
4. Evaluation order.

① Memory Management :-

In front end & code generator phase, the names in the three-code statements are mapped to addresses of data objects in run-time.

The name in the TAC refer to the symbol-table entry for the name.

The type in the declaration determines the width. means the amount of memory required to store. Based on that a relative address is determined ~~to~~ for the name.

It is done in two ways
1. static Allocation
2. Stack Allocation.

The information that is needed during the execution of a procedure is kept in a block of storage ~~structure~~ called " Activation Record ".

In static Allocation, a fixed memory is allocated for Activation Record at compile time.

(or).

The position of an activation record is fixed at compile time.

whereas in stack Allocation, the position of the activation record is not known until run time.

Instruction Selection :-

The nature of instruction set of the target machine determine the difficulty in selecting the instruction. Instruction speed & idioms are impo8t factors in the selection process.

Ex:- if we have exp a : = b+c. is converted into code like

```
Mov b, Ro.
ADD c, Ro.
mov Ro, a
```

If we convert statement-by-stament into code, it produce code which is poor & inefficient

Ex:- a: = b + c
d : = a + e.

```
mov b, Ro
ADD c, Ro.
mov Ro, a.
mov a, Ro.
ADD e, Ro.
mov Ro, d.
```

→ In this case, we are moving Ro value to 'a' and once again we are moving a value to Ro.

which is reduntant value, this is due to Statement-by-Statement code generation.

The quality of the code is determined by its speed & size. If no of instructions are less, then the size is reduced & speed is increased.

If target machine has rich instruction set then we can implement a operation in many ways.

If we use efficient instruction set, then the instruction cost also reduced.

Ex:- a : = a + 1 ;

```
mov a, R0.
ADD #1, R0.
mov R0, a.
```

If the Target machine has instruction like "INC". we can perform this a : = a+1 in 1 step like.

```
INC a, 1.
```

Register Allocation :-

"Registers" are one of the most important resource of the machine. Efficient utilisation of resource of registers produce efficient target code.

The operands of the instructions are stored either at Registers or memory.

Instruction with register operands are short & fast when compare with instruction with memory operands.

If the number of variables is higher than the no of registers available, then the code generator should identify which variables are stored in registers & which variables are stored n memory.

Register Allocation : It is the process of identifying the what set of variables need to be retained in registers.

Register Allignment :- The process of alliging a specific register to a specific variable.

Choice of Evaluation order :- The order in which the instructions are performed also can effect the efficiency of the target code.

Addressing Modes :-

Mov	move	source to destination.
ADD	ADD	Source to Destination.
SUB	Substract	Source from Destination.

MODE	FORM	Address	Added cost
Absolute	M	M	1
Register	R	R	0
indexed	c(R)	c + contents(R)	1
Indirect register	*R	contents (R)	0
Indirect indexed	*c(R)	contents (c+ contents(R))	1
Literal	#c	c	1

Ex:- ① MOV R0, M.

Store the contents of Register R0 into memory location M.

② MOV *4 (R0), M.

Store Contents (4 + contents (R0)) into memory location M.

③ MOV #5, R0.

Store Constant 5 into Register R0.

Instruction Costs :-

The cost of an instruction be one plus the cost associated with the source & destination address.

The Instruction cost corresponds to the length of the instruction.

The Addressing mode involve Registers have cost – 0
" " " memory locations of Literal " – 1

In most machines, for most instructions, the time taken to fetch an instruction from memory is more when compared the time taken for executing it.

Ex:- $a := b + c$

MOV	b, R0	-	1 +1+0 : 2
ADD	c, R0	-	1 +1+0 : 2
mov	R0, a	-	1 + 0+1 : 2
			6

② MOV b, a 1+1+1 3

ADD c, a : 1+1+1 : 3

cost : 6.

Assume that if R_0, R_1, R_2 contain a, b, c addressa, then

MOV $*R_1$, $*R_0$: 0+0+1 : 1

ADD $*R_2$, $*R_0$: 0+0+1 : 1

+

cost : 2.

② If R_1, R_2 contains the values of b & c respectively. And 'b' is not required further then

ADD R_2, R_1 : 0+0+1 : 1

MOV R_1, a : 0+1+1 : 2

+

cost : 3

For Generating good code, we need to utilise addressing capabilities efficictly, by which we can reduce instruction cost.

DAG Representation of a Basic Blocks :-

Directed Acyclic graph, is a data structure which is used for implementing transformations on basic blocks.

A ~~fo~~ dag for basic blocks is constructed with the following labels on nodes.

1. Leaves are labelled by unique identifiers, variable names are constants. It is subscript with 0,... to avoid confusion with labels denoting "current values" of the identifiers.
2. Interior nodes are labeld with operator symbol.
3. Nodes are labelled with given sequence of identifiers. It specify that this identifiers are holding that computed values.

Dag Construction :-

In this process, each TAC statement is processed. If it come across a stmt like x : = y op z. first it check, whether there is node for current value of 'y' is exist already, if so uses it. Otherwise it creates a new node and label it as 'y'.

The same is done for 'z'. Also.

we have two types of operators

Class 1 : ADD, SUB, MUL, DIV, UMINUS, ADDR-OF, ASSIGN, R-INDEX ASSIGN.

CLASS 2 : LT, GT, LE, GE, EQ, NE, L-INDEX ASSIGN PROC-BEGIN, PROC-END, RETURN, CALL, LBL, GOTO etc.

If op is one of class 1 operators.

Then find any interior node "p", with ~~of~~ child nodes as 'y' & 'z'. If we don't find such a node create a new node labelled as 'x' with child nodes 'y' & 'z'.

② If we find, then add the identifier 'x' to the list of identifiers attached to 'p'.

① If 'x' is attached to some other dag node previously, remove that link, Associate the current value of 'x' with 'p'.

if the TAC is of form x := y.

If 'x' is attached to some other dag node previously, remove that link, associate the current value of 'x' with y. Add the 'x' to the list of identifiers attached to 'y'.

Ex:-

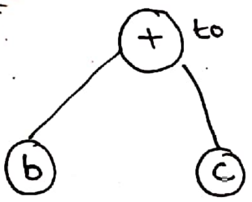
```
int main ( )

{
   a = (b+c) * d ;

   e = f * a ;

   f = (b+c) * e;

    g = (b+c)/d ;

}.
```

Intermediate IAC statements

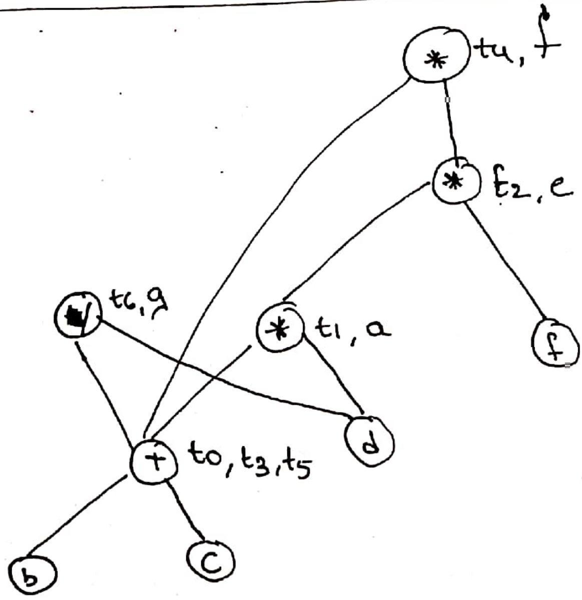
(1) proc-begin main
t_0 := b+c.
t_1 := t_0 * d.
a := t_1
t_2 := f * a.
e := t_2.
t_3 := ~~b+e~~. b+c
t_4 := t_3 * e.
f := t_4.
t_5 := b+c
t_6 := t_5/d.
g := t_6.
proc-End.

DAG :-

①



②



Optimization of Intermediat code by DAG :-

(or)

Reconstruction of Intermediat code by DAG :-

(or)

Application of DAG :-

→ Optimising transformations like common sub-exp elimination, dead store elimination, copy propagation etc occur during DAG construction.

whenever we are creating a new node, if we check whether previously a node is exist with same children & with same operator.

If not then only we are creating new node. This process allow us to detect common sub-expression & eliminate us from recomputing common-sub-expression.

Ex:- for Statement

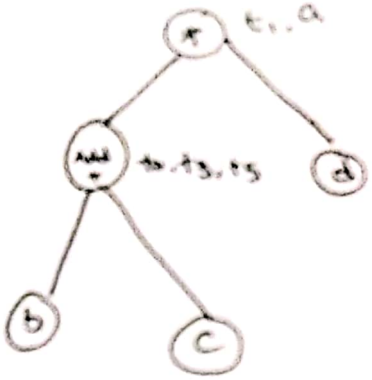
$t_3 := b + c.$

Already node 't_1' is exist, with same exp $b+c$, so we avoid re-computing '$b+c$', & simply add identifier t_3 is to. like that for $t_5 := b+c.$ also.

If ~~we~~ a node have multiple identifiers as labels we selet ~~one~~ any one among them to represent the value. This eliminates un-necessary assignment stmt of the form $a := b$

Like this we implement copy propagation.

Ex.-



we take only 'a' out of t1, a,
since 'a' is a user-defined variable.

we replace

$t_1 := t_0 * d;$

$a := t_1;$

A simple code generator :-

A code generator generate target code for a sequence of TAC statements.

→ Ex:- If there is a TAC statement like this

$x := y \text{ op } z.$ → $x := y + z$;

First it check the operands y, z are in registers. If they are in registers, it use them directly. If then find what is Target language operator for '+'. If the Target language is Assembly, then for '+' it find it is "ADD". Then it compute the statement and store the result in the register, until that register is needed for any other computation.

The code generator generate target code in different ways

① If Y is in Register - R_0 & Y is not used in any other statemts.

If Z is in Register - R_1 but Z is going to use in other statements.

Then it generate code this

ADD R_1, R_0 — R_1 - Z.
R_0 - Y.

$Z+Y$ = value in R_0.

[illegible] 'w' is in R_0

② If Y is in Register R0 and 'Y' is not used next.
Z is in memory location, Z is used next.

Then it generate code like this.

```
Add Z, R0
```

↓
Here Z is the memory location of Variable 'Z'.

③ If Y is in R0 & Y is not having any next use
Z is in memory location Z but it is used frequently
in the remaining code, then it creates code like this

```
Mov Z, R1 ;
ADD R1, R0 ;
```

It is fast to fetch Z value next time as it is stored in Register, instead of memory locations.

for doing this, it uses 2 Descriptors or 2 Data Structures

① **Register Descriptor** :- It keep track of what is currently in each Register.
(or)
It give information about which variables are currently held in a particular Register.

Initially RD show all Registers are empty. As code generator procedes, each register will hold zero or more variables at any given time.

If the statement is copy statement, then we use single register to hold more than one variable.

Ex:- x := y. a = b

If x is in R0. then R0 hold even 'y' also.

(or)

we can store both x & y variables in single register.

Register Name	Current variables
R0	a, b
R1	y
R2	z.
	⋮

→ a = b

→

Address Descriptor :- It keep track of the location or locations where the current value of the variable can be found at run time.

(or)

The location might be a register, memory location, stack location or some set of these.

Ex:-

Variable name	current location
y	R0
z	Memory location z.
x	R0, ML-x
⋮	⋮

Code Generation Algorithm :-

- I/P is TAC Statement O/P is Target language code.

Ex:- If i/p x := y op z.

Step :- 1 It first invoke getreg function.

This ~~ft~~ function return the location 'L' where the result of the computation x := y op z is stored.

→ 'L' may be a register or memory location.

Ex:- ① If y is in R0, & y is no longer going to use.

Then it return 'R0' as 'L'.

② If y is going to use in other statements, then it check the list of available registers. then send one of the Register as 'L'.

→ If case 1 fail we are going to case 2.

If case 2 fail means all registers are occupied with some variables.

③ Then it ~~send~~ select some available memory location m as send 'M' value as 'L'.

Step 2 :- Then it consult Address Descriptor, to current location of y. If it is in both R & m then it select R. If it is in only M then it perform the following stmt

Mov y, L.

step 3:- Consult the AD to get current location of z.

If it is in both R & M select R. and perform

op R, L.

↓

z is in Register R.

If it is memory location z then it perform like this

op z, L.

③ Then Mov x, L. Update the Address Descriptor of x as 'L'.

If 'L' is Register, update its RD with 'x'.

④ If y, z are in Register R₁, R₂ & they are no longer used in the block after this Statement.

free the registers R₁, R₂.

Ex:-

Register Allocation & Assignment Techniques :-

(Register Allocation is the process of identifing what variables need to be stored in the registers)

In one method, some set of registers are assigned to store base addresses, some set to store stack-pointers, some to store the result of arithmetic computations.

This method make compiler design easy, but strict use of method leads to wastage of register resources.

For efficient utilisation of registers some Register Allocation techniques are there.

1. Global Register Allocation :-

If Register allocation is performed at a basic block level is called " Local Register Allocation".

All the variables that are live at the end of the basic block which are present in registers are spilled or saved into the memory location so that the successor blocks can generate code correctly.

But for loops, in LRA, spilling should done frequently. So instead of LRA, In Global register Allocation - registers are allocated across the basic blocks.

In this, registers are allocated to frequently used variables throughout a loop.

In 'c' & Bliss language, by using "Register" storage class directey assign a register to a variable.

Usage counts :-

If 'x' is a variable present in register r. Then each reference to 'x' save one unit of cost.

To measure the frequency of usage of a variable we have a metric of one unit of cost being saved. for each use of this variable.

Ex:- If 'x' is live at the exit basic block of the loop & used outside. we save two units of cost. by avoiding store & load operations.

V. / Other

If 'x' is used at the entry of loop, first it must be loaded before getting into the loop. which require two units of cost.

So, for each exit block of loop at which 'x' is live on entry to some successor of B outside of the loop, it is stored at the cost of two units.

so, the advantage of allocating a register to 'x' within the loop 'L' is given by

$$\sum_{\text{Block } B \text{ in } L} (use(x, B)) + 2 * live(x, B).$$

use(x, B) is the number of times 'x' is used in B prior to any definition of x.

→ live(x, B) – 1, if it is live on exit from B and assigned a value in B.

→ live(x, B) – 0, if it is not live on exit from B.

Graph colouring :-

It is a systematic technique for allocating registers & managing register spills. In this method, ~~we use~~ two passes are used.

1. In the first pass, target machine instructions are selected with names used in the intermediate code became name of the registers a symbolic registers.

Once the instructions selected, second pass starts.

2. In the second pass, physical registers are assigned to the symbolic ones. and also a register-interference graph is constructed for each block.

In this nodes are symbolic registers, an edge present b/w two nodes, if one is live at a point where the other is defined.

Ex :-

1. a = b + c → Registers are a, b, c

2. d = d - b. Registers are d, b.

In Register-interference graph nodes are a, b, c, d. An edge is present b/w this nodes if one is live at a point where the other is defined.

At line '2' 'a' is live & 'd' is defined. So we connect node 'a' & node 'd' with an edge.

Here In this method the Register-interference graph is coloured with k-colors, where 'k' is the no of assignable register.

The graph is colored like that no two adjacent nodes have same color.

Here color means register. it implies that no two names are assigned with same physical register.

Code generation by DAG :-

1. Rearranging the order :-

i/p stmt : (a+b) - (e - (c+d)).

Then TAC statements are

t_1 : = a + b.

t_2 : = c + d.

t_3 : = e - t_2

t_4 : = t_1 - t_3.

code generation by DAG is done in three '3' ways:-

1. **Rearranging the order:-**

The order in which the TAC statements are executing will affect the cost of the Target code generated. If we rearrange, means if we change the order in which TAC statements execute we can optimise the Target code.

Ex:- $a+b+(e-(c+d))$.

① $t_1 := a+b$.
$t_2 := c+d$.
$t_3 := e-t_2$
$t_4 := t_3+t_1$

TAC

TAC

③ $t_2 := c+d$.
$t_3 := e-t_2$
$t_1 := a+b$.
$t_4 := t_3+t_1$

object-code

④
1. mov c, R0.
2. ADD d, R0.
3. Mov e, R1
4. SUB R0, R1.
5. mov a, R0
6. ADD b, R0.
7. ADD R1, R0.
8. mov R0, t4

② R0 & R1 are available Register, each statement is converted into object code. means line-by-line code is converted into Target obj. code.

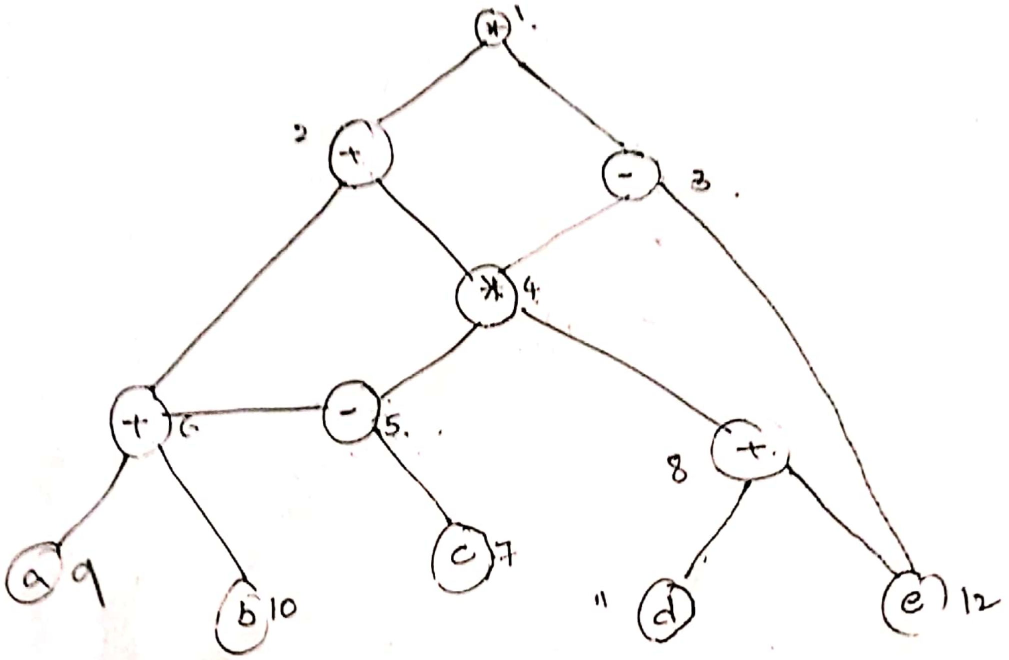
object form.

① Mov a, R0.
2 ADD b, R0.
3. mov c, R1
4. ADD d, R1.
5 Mov R0, t1 — $t_1 := a+b$
6 Mov e, R0. —
7 SUB R0, R1 — $e-(c+d)$.
8 Mov t1, R0.
9 ADD R0, R1 — R0 : a+b
R1 : e-(c+d)
10 mov R1, t4.

② Heuristic Ordering :-

In this ordering, the evaluation of a node immediately follow the evaluation of its leftmost argument.

Algorithm :-



unlisted :- 123

unlisted   list

initially - ①   1   ⊙

left-child of 1 is 2, move 2 to list   1 2

left-child of 2, is, 6, but it has one more parent '5' which is not in list.

So, we move to 3, its parent is '1' which is in list so, we move '3' to list.   1 2 3

'3' left child is 4, it has two parents 2 & 3 both are in lists so move '4 to list. 1 2 3 4

4 left child 5, move 5 also to list 1 2 3 4 5

5 " " '6' it has two parents '5' & '2' both are in list, so move '6' also to list 1 2 3 4 5 6

'6' left child is '9' but it is interior node, so we move to the right child of '4' which is 8. Its parent is '4' so we move '8' to list. 1 2 3 4 5 6 8.

8 left child is '11' which is interior node now we stop.

→ The list is 1 2 3 4 5 6 8. The order of evaluation is in reverse order of the list i.e, 8 6 5 4 3 2 1.

t_8 := d + e.

t_6 := a + b

t_5 := t_6 - c

t_4 := t_5 * t_8

t_3 := t_4 - e

t_2 := t_6 + t_4

t_1 := t_2 * t_3.

③ Labelling Algorithm :-

It has two parts

1. In first part, In bottom-up fashion label each node with an integer, with out store of the intermediate values. so few registry are required.

2. In the second part, tree traversal is done. The order is based on the computed node labels.

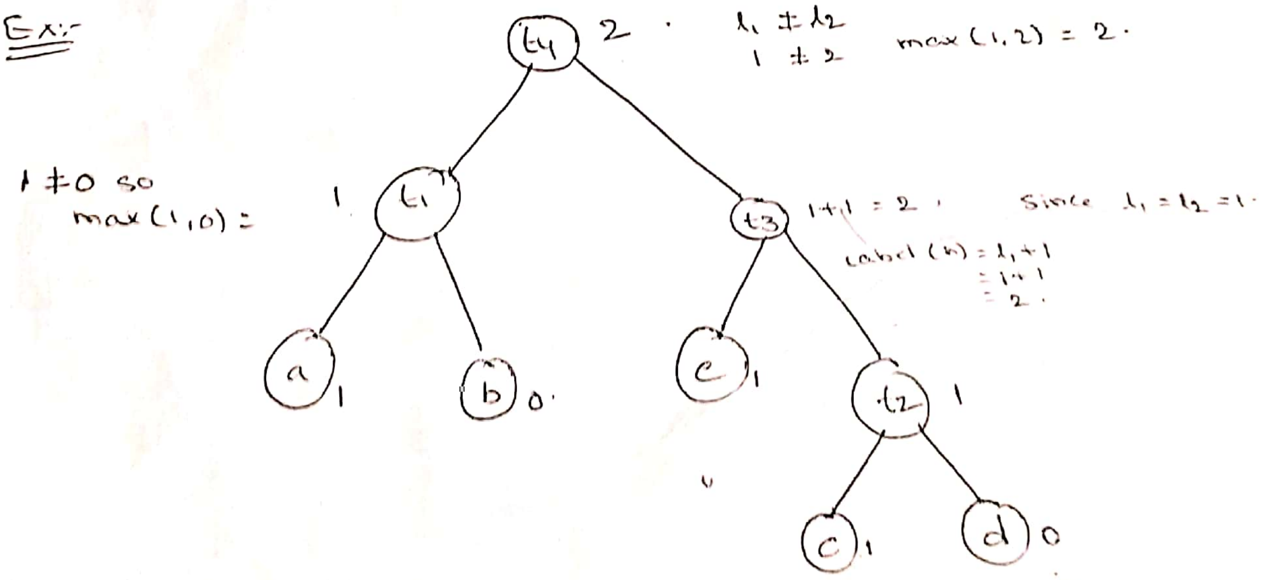
Labeling is done like that, a node is not visited untill all its children are labeled.

It is given by

$$\text{Label}(n) = \max(l_1, l_2) \quad \text{if } l_1 \neq l_2$$

$$= l_1 + 1 \quad \text{if } l_1 = l_2 .$$

Ex:-



If 'n' is leaf node and it is left leaf node of its parent,

Label (n) = 1 if it is right leaf node leaf (n) = 0 .

Peephole optimization :-

If we produce target code by converting TAC statement by statement, we get redundant instructions.

It can be optimized & improved by appling Transformations on the target pgm.

" Peephole optimization " is one of the simple & effective optimization technique which is done locally to improve target code.

[It is done on local on a small segment of code called peep hole & window.]

It is done directly after intermediate code generation to improve the intermediate representation instead after code generation.

The peep-hole code need to be continous code.

The optimization techniques that can be done on peep-hole are peep-hole optimization Techniques :-

→ First we find peep-hole instructions & replacing them with fast & short instructions.

1. Redundant - instruction elimination
2. flow - of - control optimization
3. Algebraic simplications
4. use of machine idioms.

① Redundant instruction elimination :-

(or)

Redundant load & store :-

Ex:- If we have two instructions like this

① mov Ro ~~Ro~~ a } mov Ro, a
② mov Ro, a } mov a, Ro.

we remove Instruction ② specifying that value of 'a' already in 'Ro' register.

② Removing unreachable code :-

Ex:- if age >18

\# define debug 0.

if (debug)
{
=
}

Intermediate code

If debug =1 Goto L1
Goto L2.
L1 : debugging info
L2 : - - - -

It is replaced like

if debug ≠ 1 Goto L2
print debug inf.
L2 : - - -

It remove jump over jumps.

By copy propogation it is converted as

if 0 ≠ 1 goto L2.
point debugging inf

L2 :

Use of machine Idioms :-

~~some~~ Target machines may have different hardware instructions to implement certain specific operations efficiently. If we use such instructions we can reduce execution time & cost significantly.

Ex:- Instructions are present for Auto increment, Auto decrement etc.,

```
a = a+1
MOV  a, R0                INC  i, 1
ADD  #1, R0               MOV  i, R0.
MOV  R0, a.
```

9/07/16 Absentees 2nd

1, 2, 4, 6, 10, 11, 13, 16, 18, 19, 20, 22, 23, 24, 25, 28, 31, 32, 33, 34, 36, 37, 38, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 52, 54, 55, 57,

LE2, LE3, 4, 5, 8, 6 517 524.

28/7/16 5th hour Absent list

5, 6, 11, 13, 24, 27, 28, 31, 35, 37, 39, 49, 50, 51, 53, 57, 58, 59
60, LE - 3, RE-524, 501, 517.

Code optimisation :- The process of improving the intermediate code in terms of speed & amount of memory required for execution is called "**optimisation**".

Optimisation Techniques :-

Constant folding :- In this, the constant expressions in the input source prgm are calculated and replaced by the equivalent values at the time of compilation.

Ex:-

Code	Normal Intermediate code	Optimized Intermediate code
$p = a[3]$	$t_0 := 3*4$ $t_1 := \&a;$ $t_2 := t_1[t_0]$ $p := t_2$	$t_0 := 12$ $t_1 := \&a$ $t_2 := t_1[t_0]$ $p := t_2$.
$a[3] = p$	$t_0 := 3*4$ $t_1 := \&a$ $t_1[t_0] := p$	$t_0 := 12$ $t_1 := \&a$ $t_1[t_0] := p$.

copy propagation & Dead store Elimination :-

Ex:-

If a = b. the use of variable 'b' instead of variable 'a'.

```
                                          t1   : = &a
P = a[3]          t0 : = 12               ~~t1 [12] &a~~
                  t1 : = &a               t2  : = t1 [12]
                  t2 : = t1 [t0]          P   : = t2.
                  P : = t2.

                                          t1 := &a
a[3] = 10 ;       t0  : = 12              t1 [12] : = 10.
                  t1  : = &a
                  t1 [t0] : = 10.
```

It is also called as copy propagation.

Dead code Elimination :- ~~Dead~~ code that is never executed by the prgm is called Dead code. Eliminating dead code is called Dead code Elimination.

This technique reduces the memory required for the pgm.

Ex:-

```
int vote()        void main()
{                 {
                    int age = 15, vote = 0;
                    if (age > 18)
                      vote = vote + 1;
                  }.
```

Intermediate code

```
proc-begin main
  age := 15
If 15 >= 18 goto L0
  goto L1
Label L0
  t0 := Vote +1
  Vote := t0.
Label L1
proc-end main.
```

IC after copy propogation

```
proc-begin main
  age := 19
goto L1
Label L1
proc-end main.
```

After dead code elimination

Algebraic Transformations :

Name of the Identity	Example
Additive Identity	x+0 = x
multiplicative Identity	x * 1 = x.
Multiplication with 0	x * 0 = 0.

IC statement	IC after Transformation	Name of the identity.
y: = x +0.	y := x	Additive Identity
y: = x * 1	y := x	multiplicative Identity.
y = x *0.	y :=0.	Multiplication with 0.

Strength Reduction Transformation :-

Addition operation takes fewer cycles than the multiplication operation.

Shift operation takes fewer cycles when compared to multiplication or division operation on most of the processors.

Strength Reduction Transformation identify & replace costly operations by less expensive one, which will have the same effect.

```
Ex:-    y := x * 2 ;          y := x + x.

        y := x * 32 ;         y := x << 5
```

Common Sub-expression Elimination :-

If any expression or a part of a expression i.e. Sub-expression is present more number of times in a pgm is called common sub-expression.

The process of identifying common sub-exp & eliminating their computation multiple times in the IC is known as

```
Ex:-   int main()
       {
         int a,b,c,s,avg;
          :
          s = a+b+c
          avg = a+b+c
                -----
                  2
       }
```

```
proc-begin main
  t0 : = a+b+c ,
  s : = t0 ;
  t1 : = a+b+c
  t2 : = t1/2
  avg : = t2
proc-end main .
```

Ex:-

```
int  sum(int n)
{
sum-n = (n * (n+1))/2 ;
sum-n1 = ((n * (n+1) * (2 * n+1))/6;
}.
```

```
proc-begin sum
Goto Label Lo.
Label Lo.
to  := n+1
t1  := n * to
t2  := t1/2
sum-n  := t2.
t3  := n+1
t4  := n * t3.
t5  := 2 * n
t6  := t5 +1
t7  := t4 * t6.
t8  := t7/6
sum-n1  := t8.
proc-end sum.
```

IC After common sub-expression elimination:-

Here common sub-exp is : n * (n+1).

```
proc-begin sum
    to  := n+1
    t1  := n * to.
  sum-n  := t1/2
    t2  := 2*n.
    t3  := t2 +1
    t4  := t1 * t3
  t5  := t4/6 j
    sum-n1  := t5 .
proc-end sum.
```

Constant Propagation

- In this technique the value of variable is replaced & computation of an exp is done at the compile time.

Ex
```
pi=3.14;
r=5;
Area=pi*r*r
```

Loop Optimizing Techniques:-

③ **Loop Unrolling** - No. of jumps & tests can be reduced by writing the code two times.

Ex Copying contents of one array to another -

```
int i=0;                      int i=0
while (i<10)                  while (i<10)
{                       vs    {
  b[i]=a[i]                     b[i]=a[i];
   i++;                         i++;
}                               b[i]=a[i];
                                i++;
                              }
```

Loop optimisation :-

In general, the statements in a loop get executed many times. if we optimize the code in the loop, then we get good performance improvement.

Loop optimization Techniques :-

① **Loop invariant code Motion** :- The statements within a loop whose values are computed, whose value do not change throughout the life of the loop are called "loop invariant statements". such statements are identified & moved outside of the loop. This Technique is called "Loop invariant code motion".

Ex :-

```
int func(int a, int b)
{
  int i, n1, n2;
  i = 0;
  n1 = a*b;
  n2 = a-b;
  while(i < (n1*n2))
  {
     i = i+1;
  }
  return (i);
}.
```

```
proc-begin func
 i := 0
 n1 := a*b
 n2 := a-b.
 Label .L0
 t0 := n1*n2.
 if i < t0 goto .L1
 goto .L2.
 Label .L1
   t1 := i+1
   i := t1
 Goto .L0.
 Label .L2
 return i.
 proc-end func.
```

Then the code after loop invariant code motion is.

```
proc- begin func
    i:= 0
    n1 : = a* b
    n2 : = a-b.
    t0  : = n1* n2.
    Label  .L0
       |
       |
       |
       |
       |
    proc- end func.
```

② strength Reduction on induction Variables :-

A Variable that changes by a fixed quantity on each of the iterations of a loop is called as " Induction Variable ".

```
int  ind ;
int  a[10] ;
int  func()
{
 while (ind <20)
 {
   a[ind] =10 ;
   ind = ind +1 ;
 }
}.
```

```
proc.-begin func
label .L0
if  ind <20 goto .L1
 goto  .L2
 Label  .L1.
  t0 : = ind * 4
  t1 : = &a
  t1 [t0] : = 10
  t2  :  =  ind +1
  ind : = t2
  goto  . L0
 proc-end func.
```

```
proc-begin func
 t0 : =ind *4
 Label . L0
 if  ind <20 goto.L1
   :
   t1 : = &a
   t1 [t0] : =10
   ind  : =ind+1
   t0  : = t0 +4.
   goto  .L0
 proc-end func.
```

a[5]

t0 = 5x4

= &a

a[5]

0 1 2 3 4

0+4 4+4 8+4 12+4 16+4

Basic Blocks:-

Alg: Partition into basic blocks

Input: A sequence of three address statements

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:-

1) We first determine the set of leaders, the first statements of basic blocks.

The rules we use ~~case~~ are the following:-

(i) The first statement is a leader

(ii) Any statement that is the target of a conditional or unconditional goto is a leader.

(iii) Any statement that immediately follows a goto or conditional goto statement is leader.

2) For each leader, its basic block consists of leaders and all statements up to but not including the next leader or end of. program.

Ex:-

```
int func1(int c)
{
  v3 = v1+v2;
 if. (c>100)
 {
  v4  = v1+v2;
  v1  = 0;
 }
  v5 = v1+v2;
}.
```

```
proc-begin func1
 v3  = v1+v2
if (c>100) .Goto .L0
    Goto .L1
Label .L0
 v4 : =v1+v2
  v1 : =0.
 label .L1
 v5 : = v1+v2
 Label .L2

 proc-end  func1.
```

proc begin func

[illegible]

12.

```
int func1 (int a, int b, int c)
{
  int x, y, z;
  x = 2 * a;
  y = 2 * a + 9 * b;
  if (a > 1)
  {
    x = 2 * a + 3 * b + 20;
    y = 2 * a + 4 * b + 40;
  }
  z = x * y;
  return z;
}.
```

```
1. proc begin func1              } B0
   t0  := 2 * a
   x   := t0
   t1  := 2 * a
   t2  := 5 * b
   t3  := t1 + t2
   y   := t3
If a>1 Goto Label L2
   Goto Label L1                 } B1
Label L2
   t4  := 2 * a                  } B2
   t5  := 3 * b
   t6  := t4 + t5
   t7  := t6 + 20
   x   := t7
   t8  := 2 * a
   t9  := 4 * b
   t10 := t8 + t9
   t11 := t10 + 40
   y   := t11
Label L1
   t12 := x * y;                 } B3
   z   := t12
return z
Goto Label L2
   Label L2                      } B4
proc end func1
```

Local optimization :- Performing transformation inside a basic block is called Local optimization.

B0.

```
proc-begin func1
  t0 := 2 * a
  x := t0
  t1 := 2 * a
  t2 := 5 * b
  t3 := t1 + t2
  y := t3
if a>1 Goto label .L0.
```

B0. optimised code.

```
proc-begin func1
x := 2 * a.
t2 := 5 * b.
t3 := x + t2 → This is eliminate by copy propagation
y := x + t2
if a>1 Goto label .L0.
```

Description

t0 := 2 * a is eliminated By applying copy propagation

B1

```
Goto label . L1
```

B1. optimised code.

```
Goto label :L1
```

B2

```
Label L0
t4 := 2 * a
t5 := 3 + b
t6 := t4 + t5
t7 := t6 + 20
x  := t7
t8 := 2 * a
t9 := 4 * b
t10 := t8 + t9
t11 := t10 + 40
y  := t11.
```

B2. optimised code.

```
Label .L0
t4 := 2 * a
t5 := 3 * b.
t6 := t4 + t5.
x  := t6 + 20.
t9 := 4 * b.
t10 := t4 + t9.
y  := t10 + 40.
```

Description

The statement x := t6 + 20 became ~~eliminated~~ is using copy propagation.

t8 := 2 * a is eliminated based on common-sub exp elimination

B3

```
Label L1
t12 := x * y;
z := t12
return z
Goto label .L2
```

B3 optimised code

```
Label .L1
z := x * y;
return z
Goto Label .L2
```

Description.

The statement t12 := x * y is eliminated by using copy propagation, then the z := x + y;

B4

```
Label .L2
proc-end func1
```

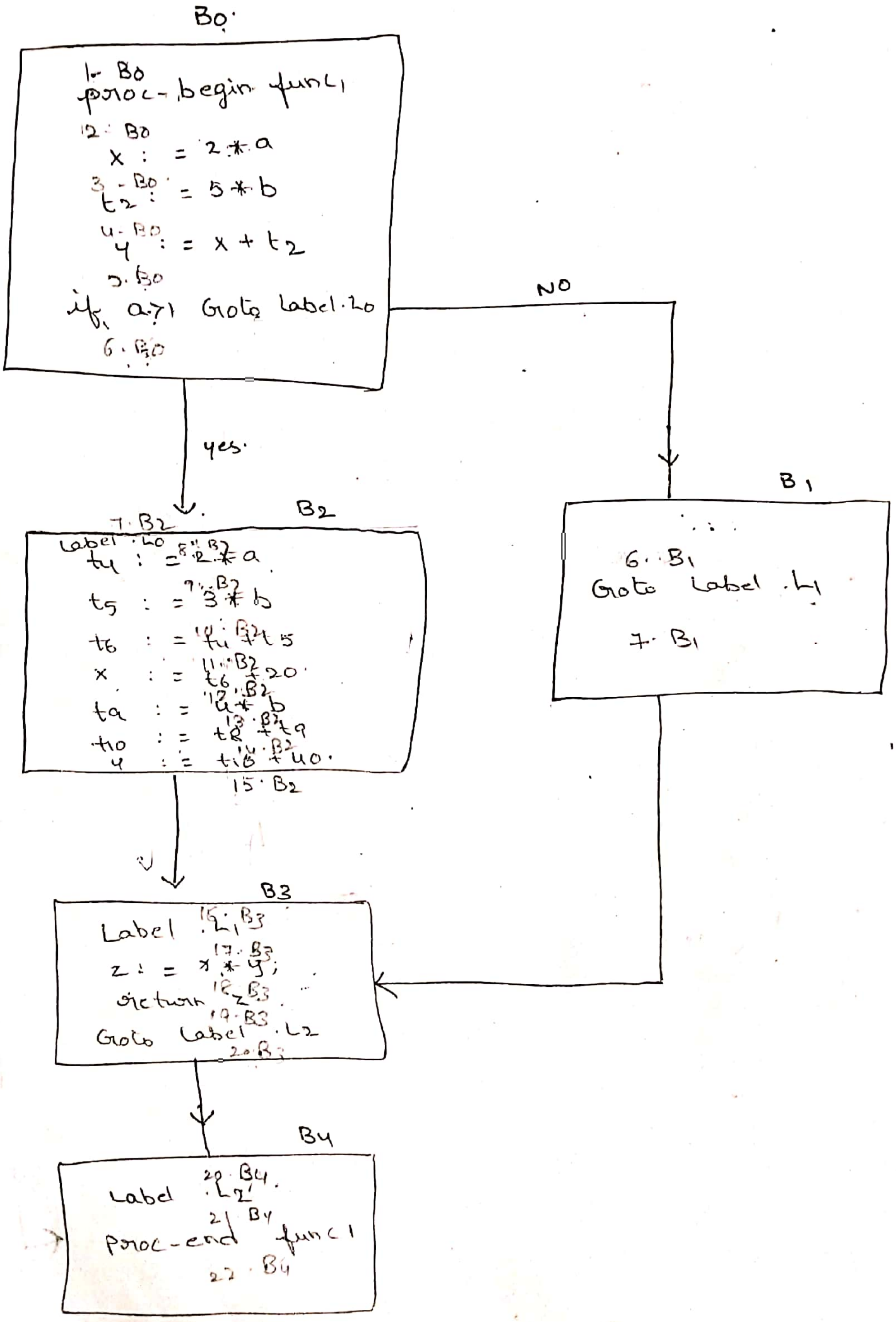
B4 optimised code

```
Label .L2
proc-end func1.
```

No optimization.

Flow- Graph :- The way of how the flow of control goes from one block to another block is represented in the form of a directed graph is called as "Flow- Graph".

Ex:-



→ ~~The initial node~~

→ The starting node in a flow graph is called "Initial node" or the node which contain the first TAC statement as a leader is called "initial node". In this example - B0 is initial node.

B_1 & B_2 are called "successors" to Block B0.

B0 is called "predecessor" to blocks B_1 & B_2.

Global optimization :-

performing transformations or optimisations among multiple blocks is called Global optimisation. The locally optimized blocks are i/p to the Global optimization.

Note :- means first we should perform local optimisation before going to global optimization.

Terms used in Global optimization :-

① point & path :- A point is a place of reference that can be found ~~be~~ at

1. before the first statement
2. After the last statement
3. or between two statements within a basic block.

Ex :- In B0 we have 6 points
B1 we have 2 points
B2 we have 9 "
B3 we have 5
B4 we have 3 ".

Path :- A path is a sequence of points in which the control can flow between two points.

Ex:- The path b/w two points $P_9 \cdot bo$ & $P_{12} \cdot B_2$ is the sequence of points : $P_5 - bo$, $P_6 - bo$,
~~$P_6 - bo$~~ $P_1 - bo$, $P_2 - bo$, $P_3 - bo$, $P_4 - bo$,
$P_9 \cdot b_2$ $P_{10} \cdot b_2$, $P_{11} \cdot b_2$, $P_{12} \cdot b_2$.

Ex:- The path b/w two points $P_0 \cdot B_0$ & $P_{20} \cdot B_3$ is given ~~in~~ as
one sequence of points ~~$P_0 B_0$~~ $P_1 \cdot bo$, $P_2 \cdot bo$, $P_3 \cdot bo$, $P_4 \cdot bo$, $P_5 \cdot bo$,
$P_6 \cdot bo$, $P_7 \cdot B_1$, $P_8 \cdot B_1$, $P_{18} \cdot B_3$, $P_{19} \cdot B_3$, $P_{20} \cdot B_3$.

(or)

Another sequence of points
$P_1 \cdot bo$, . . . $P_6 \cdot bo$, $P_9 \cdot B_2$, . . . $P_{17} \cdot B_2$, $P_{18} \cdot B_3$, $P_{19} \cdot B_3$
& $P_{20} \cdot B_3$.

Ex:- path b/w $P_{15} \cdot B_2$ & $8 \cdot B_1$.

There is no sequence of points that can take control from $P_{15} \cdot B_2$ to $P_8 \cdot B_1$.
So, we can say that there is no path b/w $P_8 \cdot B_1$ & $P_{15} \cdot B_2$.

Defining & usage of Variable :-

→ In our example ~~we~~ if we take TAC statement

x := 2 * a ;

Here, we are using the variable 'a'.

And we are defining the variable 'x'.

Data flow property :-

1. Available Expressions
2. Reaching Definition.
3. Liveness

The process of obtaining information about data flow properties by analysing the input data is called as "Data flow Analysis".

The data flow properties are computed by using some equations known as "data flow equations".

Available Expression :-

An Expression "a+b" is available at point 'p' if it satisfy the following conditions.

1. In a flow graph, every path from the initial node to point 'p' evaluates "a+b".

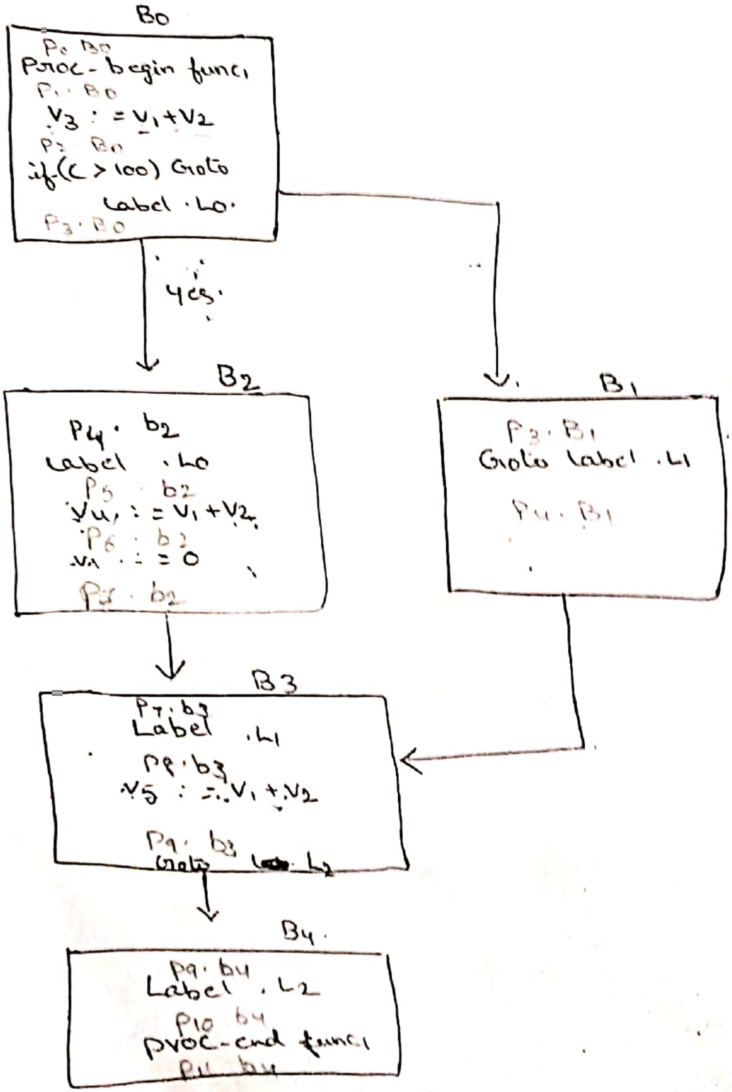
2. After last evaluation of 'a+b' & before reaching point 'p' in every path, there is no subsequent assignment to either 'a' or 'b'.

Then we say the expression 'a+b' is available expression at P.

Ex:-

```
int func1 (int c)
{
 v3  = v1 +v2 ;
if (c > 100)
{
 v4  = v1 +v2 ;
  v1  = 0 ;
}
 v5  = v1 +v2 ;
}.
```

Flow Graph :-



This property (AE) of data flow is computed during global optimization for eliminating re-evaluation of common-sub expressions across blocks.

Data-flow Analysis to compute AE :-

A block generates an expression 'a+b', if it evaluates 'a+b' expression & it does not make any assignment to either 'a' or 'b' variable in that block.

~~That~~ It is denoted as e-Gen[B], where B indicate the block number.

A block kills an expression 'a+b', if it assign a value to either 'a' or 'b' and it doesn't recompute the value of 'a+b' in that block.

It is denoted as e-kills[B], where 'B' represent the block number in which the expression is killed.

Data flow equations of AE, express the relationship between e-in[B] and e-out[B].

where.

e-In [B] - is the set of expressions that are available at the beginning of the block.

e-out [B] - is the set of expressions that are available at the end of the block.

Data flow equations of AE :-

e-out [B] = e-Gen [B] ∪ [e-In [B] - e-kill [B]]

e-In [B] = ∧ e-out [P] → for all precedenor P of the block

e-In [B_0] = φ, if B_0 is the initial block.

B_0.	e-Gen [B]	e-kill [B]	e-In [B].
proc-begin func1	{v_1+v_2}	φ	φ.
$v_3 := v_1+v_2$			
if c>100 goto L0.			

e-Gen [B] := {v_1+v_2} - This block B_0 is evaluating v_1+v_2 expression and also there is no assignment to either v_1 or v_2 in this block B_0.

e-In $[B_0]$ = ϕ, Since B_0 is the initial block.

e-Kill $[B_0]$ = ϕ, In this block there is no assignment to either v_1 or v_2 variables.

$\therefore$ e-Out $[B_0]$ = e-Gen $[B_0]$ $\cup$ [e-In $[B_0]$ - eKill $[B_0]$]

$= \{v_1 + v_2\} \cup [\phi - \phi]$

$= \{v_1 + v_2\}$.

B_1	e-Gen$[B_1]$	e-Kill$[B_1]$	~~e-Kill~~ e-In $[B_1]$.
Goto L_1	ϕ [since this block not evaluating any expression]	ϕ. [Since, it is assigning a value to any variable v_1 or v_2].	$\{v_1 + v_2\}$

e-In$[B_1]$ = e-Out $[B_0]$ $\rightarrow$ B_0 is the only predecessor to B_1 block.

$= \{v_1 + v_2\}$.

$\therefore$ e-Out $[B_1]$ = e-Gen $[B_1]$ $\cup$ [e-In$[B_1]$ - eKill $[B_1]$]

$= \phi \cup [\{v_1 + v_2\} - \phi]$

$= \{v_1 + v_2\}$.

B_2	e-Gen[B_2]	e-Kill [B_2]	e-In [B_2].
Label. L0 $V_4 := V_1 + V_2$ $V_1 := 0$	ϕ [It is assigning value to variable V_1 in the block].	$\{V_1+V_2\}$ → It is assigning value to V_1 & and recomputing the exp in the block].	$\{V_1+V_2\}$.

e-In [B_2] = e-Out [B_0] - Since 'B_0' is the only precedor to B_2.

= $\{V_1 + V_2\}$.

e-out [B_2] = e-Gen [B_2] $\cup$ [e-In [B_2] - e-Kill [B_2]]

= $\phi \cup [\{V_1+V_2\} - \{V_1+V_2\}]$

= $\phi \cup [0]$

= ϕ.

B_3	e-Gen[B_3]	e-Kill [B_3]	e-In [B_3].
Label. L1 $V_5 := V_1 + V_2$	$\{V_1+V_2\}$ It is [This block is evaluating V_1+V_2 & also it doesn't assign value to V_1 & V_2].	ϕ [It is not assigning & doesn't recomputing].	ϕ

e-In [B_3] = e-out [B_1] $\wedge$ e-Out[B_2] - for B_3, B_1 & B_2 both are precedurors

= $\{V_1+V_2\} \wedge \phi$.

= ϕ.

c-out [B3] = e-Gen [B3] $\cup$ [c-In [B3] - e-kill [B3]]

= $\{v_1+v_2\} \cup [\phi - \phi]$

= $\{v_1+v_2\}$.

B4	e-Gen [B4]	e-kill [B4]	e-In [B4].
Label . L2 proc-end func1	ϕ	ϕ.	$\{v_1+v_2\}$.

e-In [B4] = e-out [B3] → only B3 is predecessor to B4

= $\{v_1+v_2\}$

e-out [B4] = e-Gen [B4] $\cup$ [e-In [B4] - e-kill [B4]]

= $\phi \cup [\{v_1+v_2\} - \phi]$

= $\{v_1+v_2\}$.

Block	e-Gen [B]	e-kill [B]	e-In [B]	e-out [B].
B0	$\{v_1+v_2\}$	ϕ	ϕ	$\{v_1+v_2\}$
B1	ϕ	ϕ	$\{v_1+v_2\}$	$\{v_1+v_2\}$
B2	ϕ	$\{v_1+v_2\}$	$\{v_1+v_2\}$	~~$\{v_1+v_2\}$~~ ϕ
B3	$\{v_1+v_2\}$	ϕ	ϕ	$\{v_1+v_2\}$.
B4.	d	ϕ	$\{v_1+v_2\}$	$\{v_1+v_2\}$.

Note :- Data flow property Available expression in global optimization is used to perform Common - Sub - Expression elimination.

Global Common - Sub expression Elimination using AE :-

Algorithm :-

Identity the statements 'q' that contain an expression of the form x : = y op z which satisfy the following conditions

1. e - In [B] should contain that expression x : = y op z.
   B is the block which contain expression x : = y op z.
2. There is no assignment to either y or z variable before 'q' in the Block.

The statements 'q' which satisfy this conditions are added to set 'M'.

For each statement 'q' in M do the following to eliminate common sub exp.

1. Identify all the statements evaluating 'y+z' that reach q and add them to set A.
2. Create a new temporary variable 'tn'
3. for every statement w : = y op z in set A,
   tn : = y op z.
   w : = tn.
4. Replace 'q' by w : = tn.

In our Example.

In Block [B_1] : e-In [B_1] = $\{v_1+v_2\}$ And also

There is no assignment to v_1 & v_2 before this stmt.

In Block [B_2] = e-In[B_2] = $\{v_1+v_2\}$ And also

There is no assignment to v_1 & v_2 before this stmt.

In Block [B_4] = e-In [B_4] = $\{v_1+v_2\}$ And also

There is No assignment to v_1 & v_2 before this stmt.

∴ Set m = {B_1, B_2, B_4} → B_1 & B_4 Not evaluating v_1+v_2, so

Now, if we take B_2. Bo is the only block

Set A = {B_2}.
↓
quad (5)

① statement that is evaluating $\{v_1+v_2\}$ that reach quad (5) in Block B_2 is quad (1) in Bo.

② ~~so~~ creating tempolary variable to as

$t_0 := v_1 + v_2$;

$v_3 := t_0$.

③ Replace the quad (5) in B_2 as $v_4 := t_0$.

Bo. After opti.

proc-begin func
$t_0 := v_1 + v_2$
$v_3 := t_0$
if c > 100 goto .L0.

B_2. After optimi

Label .L0
$v_4 := t_0$
$v_1 := 0$.

Live Variable Analysis

Input Source

```
int func (int a, int b)
{ int i,j,K;
  i= 45;
  j=a+b;
  /* 'i' is replaced by 45 */
  if ( (a+i) >100) {
      K = a+j;
  }
  else
  { K= b+j;
  }
  return(K);
}
```

TAC after local Optimization

```
B0 | (0) proc_begin func
   | (1) i:=45
   | (2) j=a+b
   | (3) t1:=a+45
   | (4) if t1>100 goto L0
B1 | (5) goto L1
B2 | (6) label L0
   | (7) K:=a+j
   | (8) goto L2
B3 | (9) Label L1
   | (10) K:=b+j
B4 | (11) Label L2
   | (12) return K
   | (13) goto L3
B5 | (14) Label L3
   | (15) proc-end func.
```

→ A variable v is said to be <u>live</u> at a point p, if it is used in some path in the flow graph starting p.

→ The variable is considered <u>dead</u>, if it is not live.

→ The global dead code elimination is done using live-variable analysis.

- The data flow analysis done for the data flow property liveness is called "<u>live-Variable Analysis</u>".

→ We use the term live_DEFS[B] to represent the set of variables whose definition precedes any use in the block B.

→ We use the term live_USES[B] to represent the set of variables whose use precedes any definition within the block B.

- The data flow equations for live variable analysis express the relationship between the live_IN[B] & live_OUT[B]

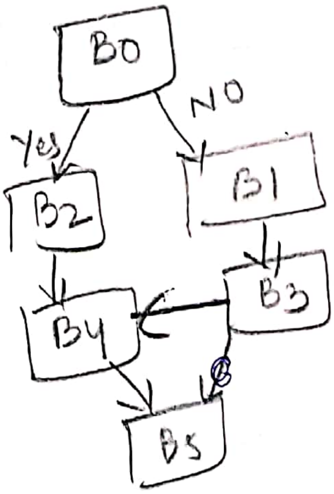
live_IN[B] is the set of all the variables that are live before reaching the beginning of the block 'B'.

live_OUT[B] is the set of variables that are live at the end of block 'B'.

Data flow equations for live variable analysis are

$$live\_IN[B] = live\_USES[B] \cup (live\_OUT[B] - live\_DEFS[B])$$

$live\_OUT[B] = \cup\ live\_IN[S]$ ..., where S is the set of successors of the block B.



The calculations of live-DEFS, live-USES, live-IN & live-OUT

	TAC after local opt	live DEFS & live-USES	live-out & live-IN
B0	(0) proc_begin func (1) i:=45 (2) j: =a+b (3) t_1:= a+45 (4) if t_1>100 goto L0	live-DEFS (B0]= {i,j,t_1} live-USES(B0]={a,b}	live-OUT(B0)={a,b,j} live-IN(B1)= {a,b}
B1	(5) goto L_1	live-USES[B1]={} live-DEFS[B1]={}	live-OUT(B1)={b,j} live-IN(B1)= {b,j}
B2	(6) label L0 (7) K:=a+j (8) goto L2	live-DEFS[B2]={K} live-USES[B2]={a,j}	live-OUT(B2)= {K} live-IN(B2)= {a,j}
B3	(9) label L_1 (10) K: = b+j	live-DEFS [B3]= {K} live-USES [B3]= {b,j}	live-OUT(B3)= {K} live-IN(B3)= {b,j}
B4	(11) label L2 (12) return K (13) goto L3	live- DEFS[B4]={} live- USES[B4]={K}	live-OUT(B4)= {} live-IN(B4)= {K}
B5	(14) label L3 (15) proc-end func	live-DEFS[B5]={} live- USES [B5]={}	live-OUT(B5)= { } live-IN(B5)= { }

live-OUT(B5) = $\{\}$

live-IN(B5) = $\{\} \cup \{\{\} - \{\}\}$
$= \{\}$

live-OUT(B4) = live-IN[B5] = $\{\}$

live-IN[B4] = $\{k\} \cup \{\{\} - \{\}\} = \{k\}$

live-OUT[B3] = live-IN(B4) = $\{k\}$

live-IN(B3) = $\{b, j\} \cup \{\{k\} - \{k\}\}$
$= \{b, j\}$.

live-OUT[B2] = live-IN(B4) = ~~{b,j}~~. $\{k\}$

live-IN(B2) = $\{a, j\} \cup \{\{k\} - \{k\}\}$
$= \{a, j\}$.

live-OUT(B1) = live-IN(B3) = $\{b, j\}$

live-IN[B1] = $\{\} \cup \{\{b, j\} - \{\}\}$
$= \{b, j\}$.

live-OUT[B0] = live-IN(B1] ∪ live-IN{B2] = ~~{b,j}~~. $\{a, b, j\}$

live-IN [B0] = $\{a, b\} \cup \{\{a, b, j\} - \{i, j, t,\}\}$
$= \{a, b\} \cup \{a, b\} = \{a, b\}$

Dead Code Elimination using Live Variable Analysis Information

Consider the block B0 & calculate the live-OUT information for each of the quad.

B0
(0) proc_begin func
(1) $i := 45$
(2) $j := a+b$
(3) $t_1 := a+45$
(4) if $t_1 > 100$ goto L0.

live-OUT(B0) = {a,b,j}

NOTE: The live-OUT of the last quad in the block is the same as the live-OUT of the block.

Calculating the live-OUT information for each quad.

quad	live-OUT	live-IN
(4) if $t_1 > 100$ goto L0	{a,b,j}	{t_1,a,b,j}
	live-DEFS[q_4] = { } live-USES[q_4] = {t_1}	
(3) $t_1 := a+45$	{t_1,a,b,j}	{a,b,j}
	live-DEFS[q_3] = {t_1} live-USES[q_3] = {a}	
(2) $j := a+b$	{a,b,j}	{a,b}
	live-DEFS[q_2] = {j} live-USES[q_2] = {a,b}	
(1) $i := 45$	{a,b}	{a,b}
	live-DEFS[q_1] = {i} live-USES[q_1] = { }	

→ live-out information at each quad can be used in the global dead code elimination.

Statmt #	Quad	live-OUT
0	proc-begin func	{a, b}
1	i := 45	{a, b}
2	j := a+b	{a, b, j}
3	t_1 := a+45	{a, b, j, t_1}
4	if t_1 > 100 goto L_0	{a, b, j}

Algorithm formalizes the idea of global dead code elimination using the liveout information

```
for each quad 'q' of the form x := y op z in the block
{
    if (live-out [q] does not contain x)      ~~{/* x is dead */~~
    {  /* x is dead */
       eliminate the quad q
    }
}
```

→ Here in quad #1, the variable 'i' is not a member of live_OUT. i,e 'i' is dead
∴ quad #1 can be eliminated.

∴ TAC after global dead code elimination is.

(0) proc-begin func
(1) j := a+b
(2) t_1 := a+45
(3) if t_1 > 100 goto L_0

Reaching Definition

- We use the RD property to perform optimizations in loop.

Detection of loop - The ~~de~~ loops in programs are detected during the data flow analysis by using a concept called as "domination" in a flow graph

→ Here we examine the concepts & algorithms to:

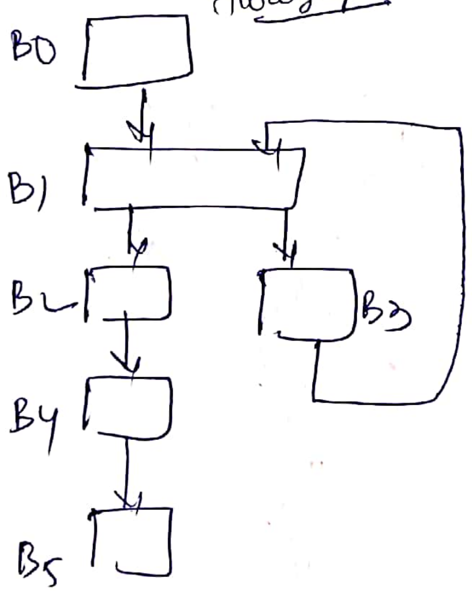
ⓐ Detect the presence of a loop, given the intermediate code.

ⓑ Identify the basic blocks in the intermediate code that constitute a loop.

→ A node 'd' of a flow graph dominates node 'n', if every path from the initial to 'n' goes through 'd'. It is represented as 'd dom n'

Note :- Each node dominates itself.

Flowgraph



Dominators

dominators[0] = {0}
[1] = {0, 1}
[2] = {0, 1, 2}
[3] = {0, 1, 3}
[4] = {0, 1, 2, 4}
[5] = {0, 1, 2, 4, 5}

→ For an edge in a flow graph a → b

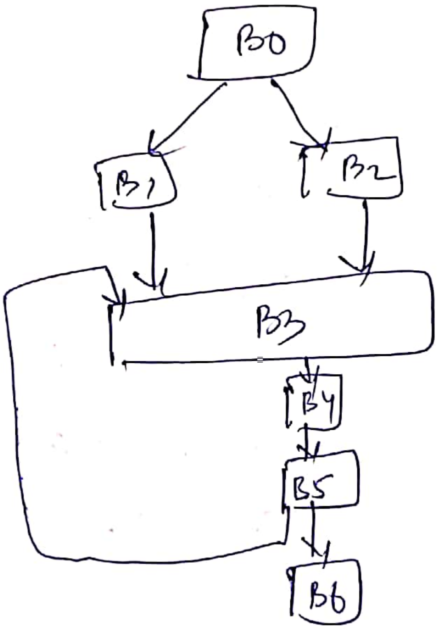
$$\begin{tikzcd} a \arrow[d] & b \arrow[d] \\ \text{tail} & \text{head} \end{tikzcd}$$

Edges & dominators for head & tail

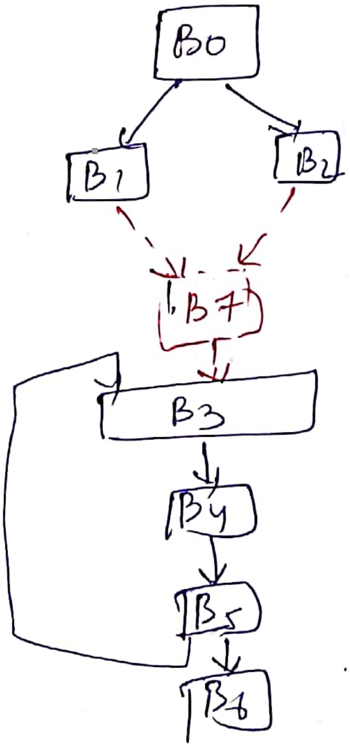
Edge	Head	Tail	dominator [head]	dom[tail]	Remarks
0→1	1	0	{0,1}	{0}	
1→2	2	1	{0,1,2}	{0,1}	
2→4	4	2	{0,1,2,4}	{0,1,2}	
4→5	5	4	{0,1,2,4,5}	{0,1,2,4}	
1→3	3	1	{0,1,3}	{0,1}	
3→1	1	3	{0,1}	{0,1,3}	Back edge

→ If dominators[tail] contain the head, then the edge is called a back edge.

→ Presence of a back edge indicates the existence of a loop in a flow graph.



Before the introduction of preheader



After the introduction of preheader

→ In one of the loop optimization Techniques, say code motion, it is required to move several quads from within the loop to outside of the loop.

- In the optimised code, these quads would typically need to be executed before entering the loop

→ A pre-header block serves as a placeholder for the quads that need to be executed just before entering the loop.

- The pre-header is a basic block introduced during the loop optimization to hold the quads that are moved from within the loop.
- It is a predecessor to the header block.

Identifying the Basic Blocks forming a loop

→ Given a back edge $n_1 \rightarrow n_2$, A natural loop is n_1, n_2 & the set of nodes that can reach 'n_1' without going through 'n_2'.

→ We apply reaching definition ~~&o~~ in one of the loop optimisation called as loop invariant code motion

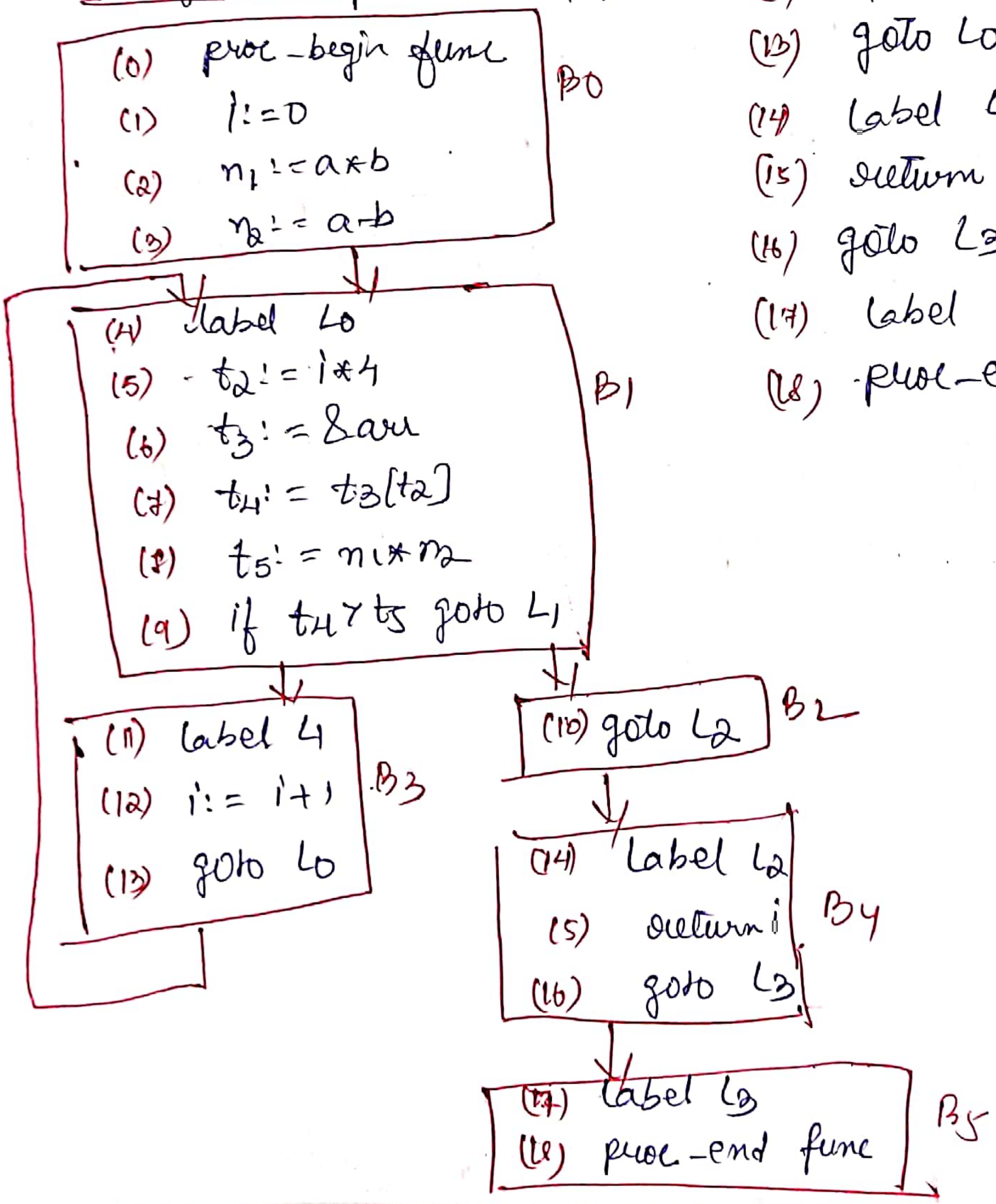
Input source

```
int arr[1000];
int func(int a, int b)
{ int i;
  int n1,n2;
  i=0;
  n1=a*b;
  n2= a-b;
  while(arr[i] > (n1*n2))
   { i=i+1;
   }
  return(i);
}
```

(i) Locally optimised TAC

```
(0)  proc_begin func
(1)   i:=0
(2)   n1:=a*b
(3)   n2:=a-b
(4)  Label L0
(5)   t2:= i*4
(6)   t3 := &arr
(7)   t4:= t3[t2]
(8)   t5 := n1*n2
(9)  if t4 > t5 goto L1
(10) goto L2
(11) Label L1
(12)  i:=i+1
(13) goto L0
(14) Label L2
(15) return i
(16) goto L3
(17) Label L3
(18) proc_end func
```

TAC after local opt & flow graph



```
B0:
(0) proc_begin func
(1)  i:=0
(2)  n1:=a*b
(3)  n2:=a-b

B1:
(4) Label L0
(5) t2:=i*4
(6) t3:=&arr
(7) t4:= t3[t2]
(8) t5:=n1*n2
(9) if t4>t5 goto L1

B2:
(10) goto L2

B3:
(11) label L1
(12) i:= i+1
(13) goto L0

B4:
(14) Label L2
(5)  return i
(16) goto L3

B5:
(17) label L3
(18) proc_end func
```

- To understand the terminology that is required for computing the reaching definitions →

The universal set of definitions L consists of any definition that appear in the statements.

- Here definitions can be seen in the quads 1, 2, 3, 5, 6, 7, 8, 12

∴ L = {1, 2, 3, 5, 6, 7, 8, 12}

→ A block generates a definition 'd', if the definition made reaches the end of the block. We use the term rd_GEN[B] to denote the set of definitions generated by a basic block B.

→ A block kills all the definitions of a variable 'x' made outside the block, if it assigns a value to 'x'. We use the term rd_KILL[B] to denote the definitions killed by a basic block B.

The data flow equations are

rd_OUT[B] = rd_GEN[B] ∪ (rd_IN[B] − rd_KILL[B])

rd_IN[B] = ∪ rd_OUT[P], for all the predecessors of P of the block.

#	TAC	rd_GEN	rd-IN & rd-OUT
B0	(0) proc-begin func (1) i:=0 (2) $n_1:=a*b$ (3) $n_2:=a-b$	rd_GEN[B0]={1, 2, 3} rd_KILL[B0]={12}	rd-IN[B0]={} rd-OUT[B0]={1, 2, 3}
B1	(4) Label L0 (5) $t_2:=1*4$ (6) $t_3:=$ &arr (7) $t_4:=t_3[t_2]$ (8) $t_5:=n_1*n_2$ (9) if $t_4>t_5$ goto L_1	rd_GEN[B1]={5, 6, 7, 8} rd-KILL[B1]={}	rd_IN[B1]={1, 2, 3, 12} rd-OUT[B1]={1, 2, 3, 5, 6, 7, 8, 12}
B2	(10) goto L2	rd-GEN[B2]={} rd-KILL[B2]={}	rd-IN[B2]={1, 2, 3, 5, 6, 7, 8, 12} rd-OUT[B2]={1, 2, 3, 5, 6, 7, 8, 12}
B3	(11) Label L1 (12) i:=i+1 (13) goto L0	rd-GEN[B3]={12} rd-KIL[B3]={1}	rd-IN[B3]={1, 2, 3, 5, 6, 7, 8, 12} rd-OUT[B3]={2, 3, 5, 6, 7, 8, 12}
B4	(14) Label L2 (15) return i (16) goto L3	rd_GEN[B4]={} rd_KILL[B4]={}	rd-IN[B4]={1, 2, 3, 5, 6, 7, 8, 12} rd_OUT[B4]={1, 2, 3, 5, 6, 7, 8, 12}
B5	(17) Label L3 (18) proc-end func	rd-GEN[B4]={} rd-KILL[B4]={}	rd_IN[B5]={1, 2, 3, 5, 6, 7, 8, 12} rd_OUT[B5]={1, 2, 3, 5, 6, 7, 8, 12}.

{Initial value of r-OUT[B] for every block B is assumed as rd-GEN[B]}

→ with these values repeat the whole process as second iteration.

Loop Invariant Code Motion Optimization using RD Analysis

There are two steps required for performing the loop invariant code motion optimization. They are:

1) The detection of loop invariant statements in the loop. This is based on the ud-chain inf. obtained from the reaching definition analysis.

2) The moving of the loop invariant stmts to the pre-header of the loop.

Detection of loop invariant statements

A stmt 's: x = y+z' in a loop L is considered as loop invariant if one of the following conditions hold good:

1) All the reaching definitions of 'y' & 'z' at 's' are from outside the loop as indicated by the ud chains for the quad q.

2) The operands 'y' & 'z' are constants

② Moving the Loop Invariant Statements to the Pre-header

For a stmt 's: a=b+c', to be moved into the pre-header, the following are the conditions that should be met.

1) There should be no other stmt 's' which defines 'a' within L.

2) The reaching definition for all the uses of 'a' in the loop should be from s only.

3) The stmt 's' should be in a block that dominates all the exits of the loop L.

2nd iteration

#	TAC	rd-GEN & rd-KILL	rd-IN & rd-OUT
B0	(0) proc-begin func (1) i:=0 (2) r_1:=a*b (3) r_2:=a-b	rd-GEN[B0] = {1,2,3} rd-KILL[B0] = {12}	rd-IN[B0] = {} rd-OUT[B0] = {1,2,3}
B1	(4) Label L0 (5) t_2:=i*4 (6) t_3:=&arr (7) t_4:=t_3[t_2] (8) t_5:=r_1*r_2 (9) if t_4>t_5 goto L_1	rd-GEN[B1] = {5,6,7,8} rd-KILL[B1] = {}	rd-IN[B1] = {1,2,3,5,6,7,8,12} rd-OUT[B1] = {1,2,3,5,6,7,8,12}
B2	(10) goto L2	rd-GEN[B2] = {} rd-KILL[B2] = {}	rd-IN[B0] = {1,2,3,5,6,7,8,12} rd-OUT[B2] = {1,2,3,5,6,7,8,12}
B3	(11) Label L1 (12) i:=i+1 (13) goto L0	rd-GEN[B3] = {12} rd-KILL[B3] = {1}	rd-IN[B3] = {1,2,3,5,6,7,8,12} rd-OUT[B3] = {2,3,5,6,7,8,12}
B4	(14) Label L2 (15) return i (16) goto L3	rd-GEN[B4] = {} rd-KILL[B4] = {}	rd-IN[B4] = {2,3,5,6,7,8,12} rd-OUT[B4] = {2,3,5,6,7,8,12}
B5	(17) Label L3 (18) proc end func	rd-GEN[B5] = {} rd-KILL[B5] = {}	rd-IN[B5] = {1,2,3,5,6,7,8,12} rd-OUT[B5] = {1,2,3,5,6,7,8,12}

rd-OUT in the first & 2nd iteration :

Block #	rd-OUT in the iteration #1	in the iteration #2
0	rdOUT[B0] = {1,2,3}	# same.
1	rd-OUT[B1] = {1,2,3,5,6,7,8,12}	
2	rd-OUT[B2] = {1,2,3,5,6,7,8,12}	
3	rd-OUT[B3] = {2,3,5,6,7,8,10}	
4	rd-OUT[B4] = {1,2,3,5,6,7,8,12}	
5	rd-OUT[B5] = {1,2,3,5,6,7,8,12}	

UD-chain (use-definition)

It is a set holding all definitions reaching a quad, for each variable used in the quad.

ud-chain $(8, n_1)$ = {2}
ud-chain $(8, n_2)$ = {3}

ud-chain Information

Block	Quad	ud-Chain Information	Explaination
B_1	(5) $t_2 := i*4$	ud-Chain(5,i) ~~info~~ = {1,12}	
B_1	(6) $t_3 := \&arr$	None	
B_1	(7) $t_4 := t_3[t_2]$	ud-chain $(7, t_3)$ = {6} ud-chain $(7, t_2)$ = {5}	
B_1	(8) $t_5 := n_1 * n_2$	udchain $(8, n_1)$ = {2} ud-chain $(8, n_2)$ = {3}	
B_1	(9) if $t_4 > t_5$ goto L_1	udchain $(9, t_4)$ = {7} ud-chain $(9, t_5)$ = {8}	