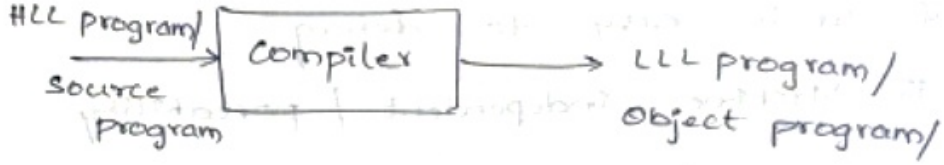
UNIT-1

INTRODUCTION:-

Definition Of Compiler:- Converts high level language (HLL) to low level language (LLL)



→ There are three classifications of programming languages. They are:-

(i) low level language (or) Machine level (or) binary language

(ii) Assembly language (or) Symbolic language

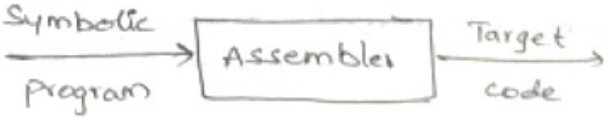
(iii) High level language

Low level language:-

Drawbacks:- 1. It is difficult to write a program
2. Testing & debugging is difficult.
3. Machine dependent (one must know about the computer hardware i.e., Registers)

Assembly language:-

- It is a symbolic language which uses mnemonic codes.
- It can be easily interpreted by programmer



Drawbacks:- platform dependent

High level language :-

Very close to human speaking languages.

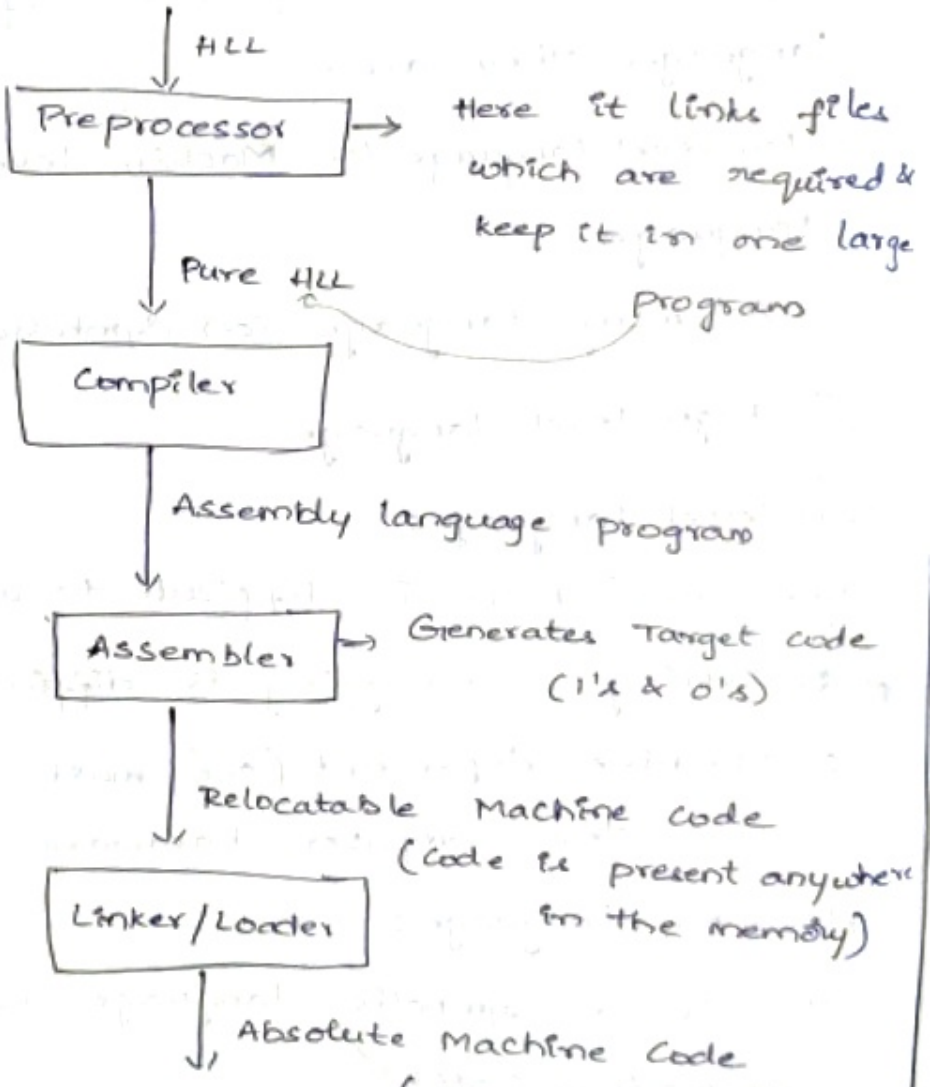
Advantages :-

(i) Faster program development

(ii) It is easy to debug

(iii) Machine independent / portability

Language Processing System :-



linker - Resolves external memory references (several modules)

loader - loads codes into the memory

PREPROCESSOR :-

- A source prog may be divided into modules stored in separate files
- Processes HLL program by including file content and Expanding macros
- Code is get ready for compilation after processing.
- The task of collecting the source prog is sometimes entrusted to a separate prog called preprocessor
- The preprocessor may also Expand shorthands called macros, into source lang statements.

Functions Of PreProcessing :-

(i) File Inclusion
(ii) Macro Expansion
(iii) Rational preprocessor
(iv) Language extension

File Inclusion :-

When a preprocessor encounters #include keyword then it replaces it with by its content.

→ There are two ways to include files

1. Angle brackets - #include <file.h>
2. double Quotes - #include "file2.h"

When it is represented with Angle brackets then the compiler checks in standard library files located i.e., default directory.

When it is represented in double quotes then first it checks in present directory & then it goes to the default directory i.e., directory above all the standard library files located.

Macro Expansion:- It is a rule (or) pattern which maps certain i/p sequence into o/p sequence

# define	i/p sequence	o/p sequence
# define	identifier	replacement

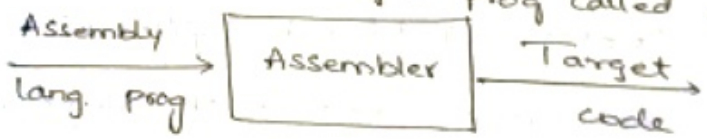
When a preprocessor sees #define keyword it replaces identifier with replacement in the rest of the program.

Rational Preprocessors:- It consists of built in macros for complex constructs like while, if, etc

Language Extension:- Add new capabalities to a language.

ASSEMBLER:- It translates assembly language program into target code.

- The modified source prog is then fed to compiler
- The compiler may produce an assembly lang prog as its o/p bcz assembly lang is easier to produce as o/p & easier to debug
- The assembly lang is then processed by a prog called an assembler that produces relocatable m/c code as its o/p



→ There are 2 types of assembler

i. One pass assembler - Generates target code in single pass i.e, by scanning all the statements once

ii. Two pass assembler - Generates target code in two passes (or) scans

→ In 1st pass – it enters all symbols into symbol table

In 2nd pass – Uses those symbols to generate target code

→ In one pass assembler – symbols are defined in advance before their use

→ In Two pass assembler – symbols can be defined anywhere in the program

→ One pass assemblers are faster than the two pass assembler.

Linker:–

It is a program which takes one (or) more object files generated by a compiler and make them into single program

Tasks performed by linker:–

1. Search for the library routines
2. Determine the location of each object file and relocate them to make single program
3. Resolve references among files (i.e., linking address)

Loader:–

Loader is a program that loads target code into main memory

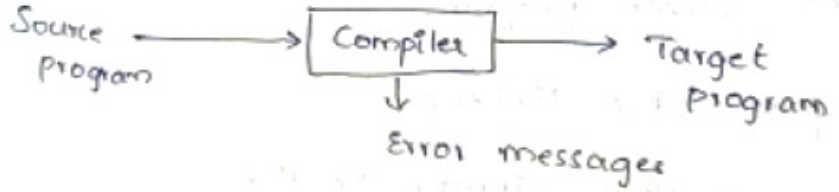
Tasks performed by loader:–

1. Determine the size of the file.

2. Create new address space (i.e, to create space in main memory)
3. Copies instructions and data to that address space.
4. Initializing stack pointer
5. Initialize various machine registers
6. Calls main program routine

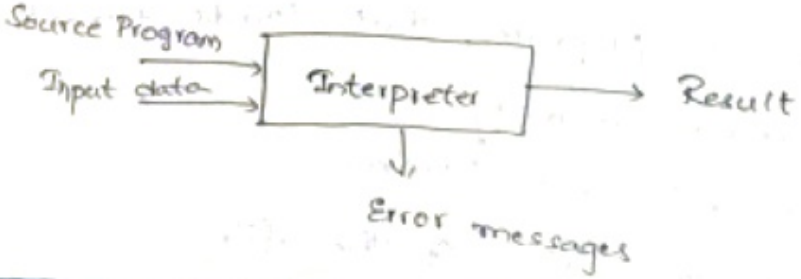
COMPILER :-

It is a program that take source program written in one language (HLL) and translate it into equivalent program in another language (LLL) and reports Error messages to users



Interpreter :-

It is a program which performs statement by statement translation and performs operations on the specified input data



1. Comp
Entire
2. If th
Error f
generate
3. Objec
is not
compiler
4. It pr
Statemen
physical
5. It p
program
Exactly
6. Process
less
7. Compil
Size
memor
8. Some
langua
FORTRA

Compiler	Interpreter
1. Compiler scans the Entire program at once	1. Interpreter translates one statement at a tim
2. If the program is Error free then compiler generates object code	2. If the program is Error free then Interpreter generates Result.
3. Object program Execution is not carried out by compiler.	3 Object program executio is carried out by interpre
4. It processes the program statements in their physical I/p sequence	4. It processes accordi to the logical flow of control through the program
5. It processes each program statements Exactly once	5. It might processes some statements repeated and Ignore others
6. Processing time is less	6. Processing time is more
7. Compilers are larger in size and occupy more memory.	7. Interpreters are smalle than compilers.
8. Some compilation languages are C, C++, FORTRAN, PASCAL, Adaet.	8. Some interpretation languages are LISP, ML, Prolog, small Talk etc

Structure Of the Compiler / Phases of the Compiler :-

Compilation is divided into two major parts

i. Analysis :- Breaks source code into pieces & impose some grammatical structure on those pieces.
   - Lexical analysis, Syntax analysis, Semantic analysis, Intermediate code
   - It is also called as front end of compiler

ii. Synthesis :- Generates target code
   - code generation
   - It is also called as back end of compiler

→ Compiler operates in sequence of phases
   i. Lexical Analysis
   ii. Syntax Analysis
   iii. Semantic Analysis
   iv. Intermediate code generation
   v. Code Optimization
   vi. Code generation

Symbol table manager

LEXICAL ANALY

It reads

group them in

There lexem

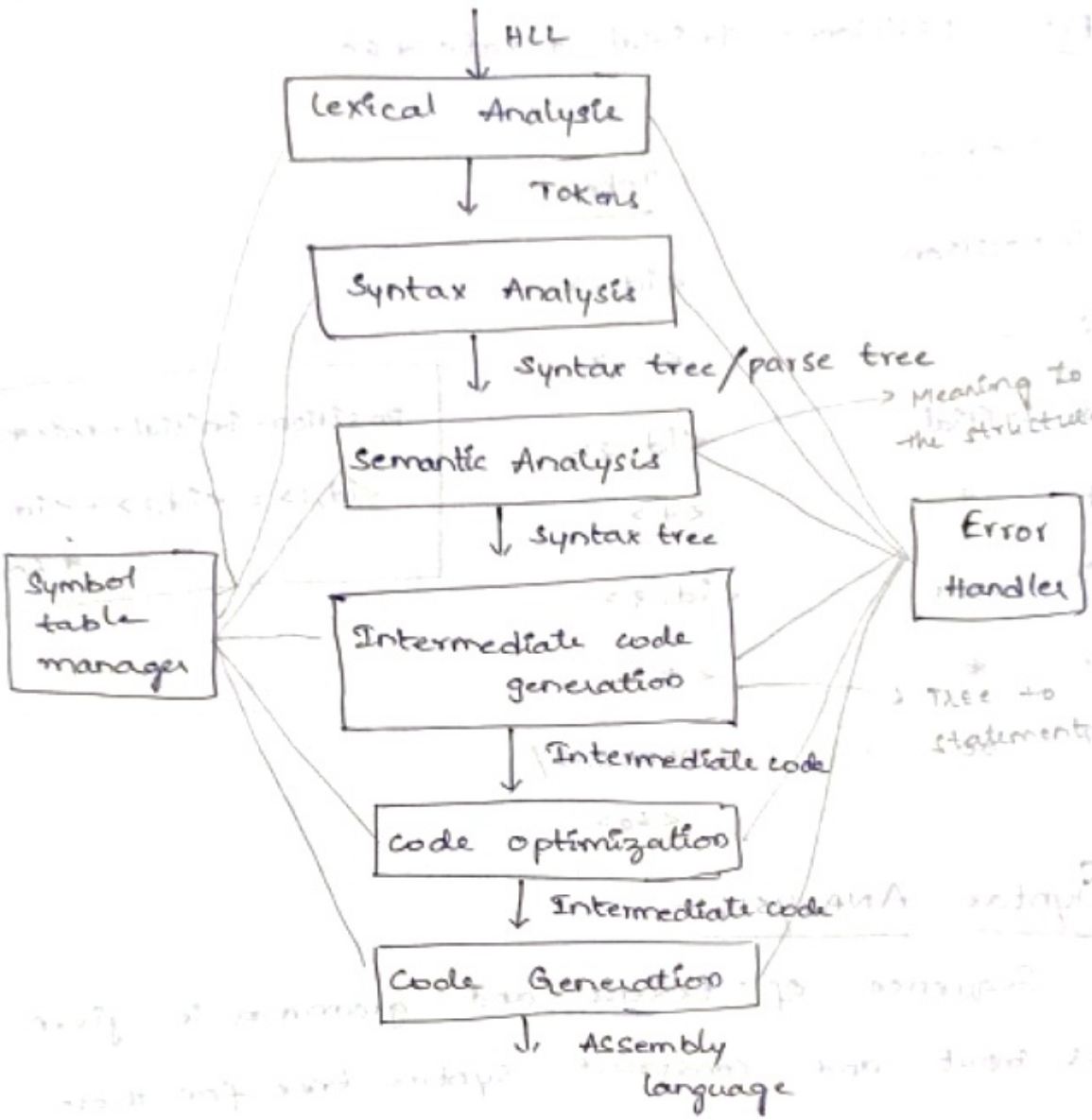
certain patte

→ Tokens ar

i.e., < To

< id

< nu



LEXICAL ANALYSIS :—

It reads source program characters and group them into meaningful sequences called lexemes. These lexemes mapped to tokens based on certain pattern.

→ Tokens are represented with 2 fields

i.e., < Token_name , attribute_value >

< id , pointer to symbol table entry >

< num, 2 >

Ex:- position = initial + rate * 60

lexemes	Token
1. position	<id,1>
2. =	<=>
3. initial	<id, 2>
4. +	<+>
5. rate	<id, 3>
6. *	<*>
7. 60	<num, 60>/ <60>

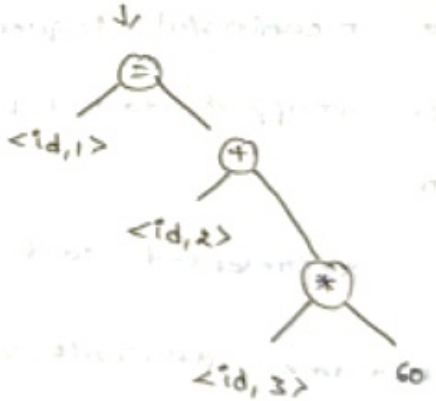
position = initial + rate * 60

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

Syntax Analysis:-

Sequence of tokens and grammar is given as input and construct syntax tree for those tokens.

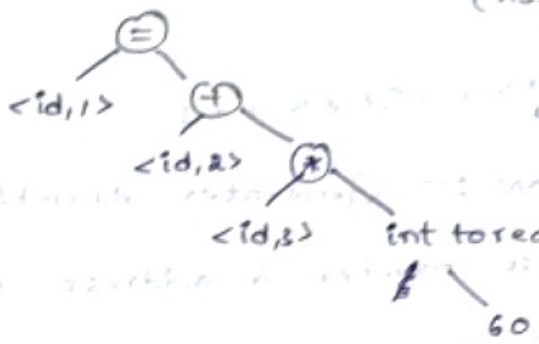
<id,1> = <id, 2> + <id, 3> * 60



SEMANTIC ANALYSIS :- Gives meaning to the syntax tree

Syntactic structure is consistent with language definition. It performs type checking

Ex



(Assume all are in float type so we convert int to real (i.e. 60))

INTERMEDIATE CODE GENERATION:-

- Machine like intermediate code for the syntax tree
- Most popular intermediate code is "three address code"
- Maximum we can have three addresses
- Every intermediate result to be stored in temporaries

t1 = (int to real) (60)

t2 = <id, 3> * t1

t3 = <id, 2> + t2

<id, 1> = t3

- In this intermediate code, the tree is represented in statement form
- It is simpler than HLL
- Here for every computation it will create one temporary.

CODE OPTIMIZATION :-

Improves intermediate code by removing redundant statements. so that better target code is generated

```
t1 : <id,3> * 60.0

t2 is
<id,1> = <id,2> + t1
```

CODE GENERATION :- Generates Assembly language code.

- TO differentiate number & address we use '#' symbol

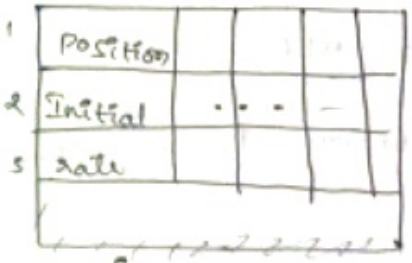
```
LDF   R1 , #60
MULF  R1, R1, <id,3>
LDF   R2, <id,2>
ADDF  R1, R1, R2
STF   <id,1> , R1
```

SYMBOL TABLE MANAGER :-

It is a data structure used to store additional information of various symbols encountered in the source program

Variables - string, type, scope

Functional Variables - No. of arguments, types of arguments, method of passing (i.e., call by value call by reference)



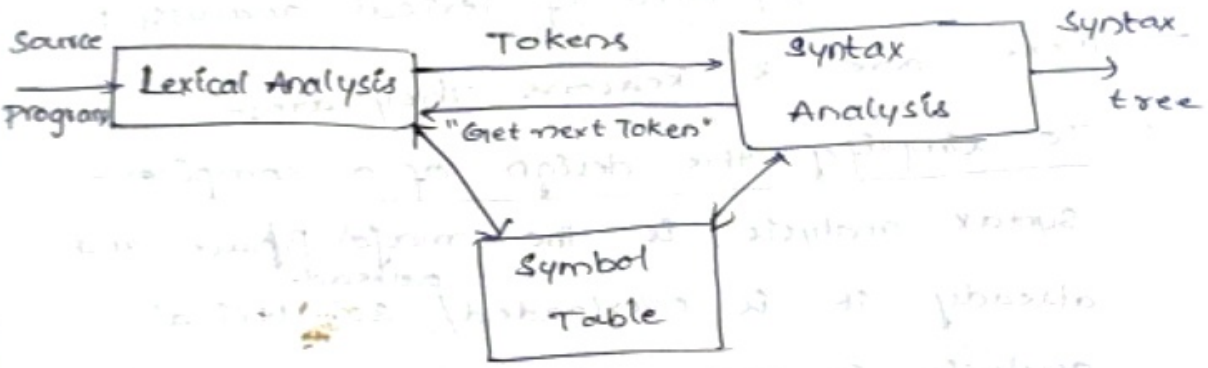
Symbol table

ERROR HANDLER:- It handles errors at every phase level

LEXICAL ANALYSIS:-

Role Of Lexical Analysis:-

- Major role is to recognise Tokens.



- Here both lexical analysis and Syntax Analysis work together i.e., Syntax analyzer initiates lexical analyzer.
- When Lexical Analyzer receives "get next token" command from Syntax analysis then it starts reading characters to recognise next token and then it returns the token to the parser.
- When it doesn't give any command the meanwhile it constructs Syntax tree/parse tree
- lexical analysis performs some other tasks also. They are:-

(i) Strip out comments and whitespaces

(ii) Associates Error messages to the source program by giving line numbers to source code.

Lexical Analysis Vs Syntax Analysis :-

(Reasons for Separating lexical analysis from syntax analysis) / Need of lexical analysis :-

— There are 3 Reasons. They are :-

(i) To simplify the design of a compiler -

Syntax analysis is the major phase and already it is overloaded/overhead. so lexical analysis is not added to the syntax Analysis.

- so simpler tasks are performed by lexical analysis.

(ii) To enhance compiler efficiency :-

Generally we read characters from I/p devices, it takes more time for character reading process. So we use Input buffering technique - which speeds up the character reading.

(iii) To improve portability :-

Portability - (i) computer architecture
(ii) Operating system

i.e., To become machine independent

- Device pecularities are restricted in the first phase itself.

Ex:- In PASCAL lang '↑' - Multiplication

In C - lang * - Multiplication

After first phase it generates tokens and sends to syntax analysis as 'Mult-op'

TOKENS, PATTERNS, LEXEMES:-

Token:- It is an abstract symbol having some collective meaning

Lexeme:- Sequence of characters in the source program that is matched by a pattern for a token.

Pattern:- It is a rule which is describing how set of lexemes can be mapped to a particular token

Tokens	Sample lexeme	Description of Pattern
if	if	if
relational-op	<, <=, >, >=, ==, !=	< (OR) <= (OR) > (OR) >= (OR) == (OR) !=
id	Pi, count, D1	letter(letter/Digit)*
num	3.5245, 54, 3.26E21	Any numeric constant
literal strings	"Compiler Design"	Any set of characters Enclosed in double quotes.

- In most of the programming languages keywords, operators, identifiers, constants, numbers, literal strings, punctuation symbols, commas, semicolons are treated as tokens

TOKEN ATTRIBUTES:-

- Attributes describe tokens
- Attributes provides additional information to tokens.

Ex:- E = M * C * * 2

E - < id, pointer to symbol entry for M >

= - < Assignment-op >

M - < id, pointer to symbol Entry for C >

** - < Exp-op >

2 - < Num, 2 >

Lexical Errors:-

Prefix of remaining input does not match to any particular pattern then it is called lexical error.

Ex:- fi(a == f(x))

Here it is misspelled keyword i.e., if is declared as fi

But in lexical analysis it gives an Error as undeclared function identifier where as it is syntax error.

- But lexical analysis provides some recovery actions.

i) inserting missing character.

ii) Deleting extra character

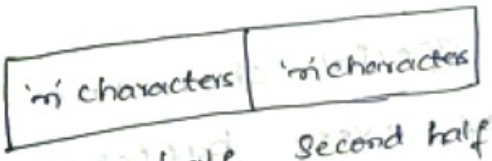
iii) Replacing incorrect character by correct character

iv) Transposing two adjacent characters

INPUT BUFFERING:-

- lookahead some characters so that we can speed up reading process.
- All these characters are placed in buffers.

Buffer Pair:- It is divided into 2 halves.



n - disk block size

- We read disk in block size

How to recognise tokens:- To recognise tokens we use two pointers. They are:-

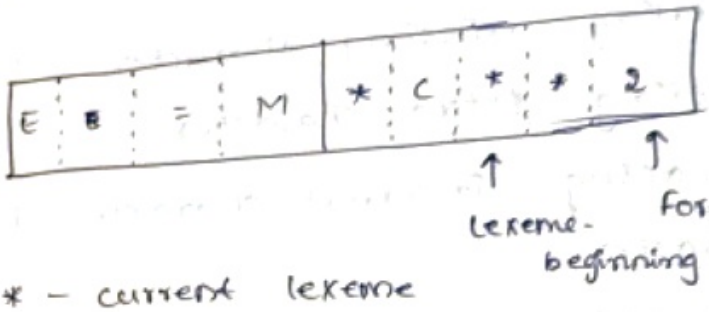
i) **Lexeme - beginning:-** Points to first character of a lexeme

ii) **forward:-** Initially point first character then it advances until it recognises next token.

- When we reach at the end of the first half second half is reloaded. Similarly

when we reach at the end of the second half first half is reloaded.

Ex:-



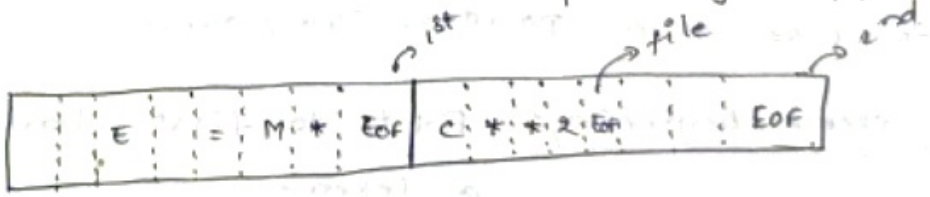
** - current lexeme

- This is called as basic approach

Drawback:- In this approach everytime we have to check whether it is reached to the end of the buffer (or) not.

SENTINALS:-

- It is introduced to overcome the drawback of basic approach
- Sentinals are special characters inserted at the end of every buffer.
- The character which is used as sentinal is "EOF" at the end of every buffer.



- There are three EOF's. They are
  1. EOF at the end of 1st buffer
  2. EOF at the end of 2nd buffer
  3. EOF at the end of the file
- It works in circular manner.

Code:-

```
switch (*forward ++)
{
  case EOF : if forward is at the end of
             first buffer
               reload second buffer
               forward = forward + 1
             else if forward is at the end of
               second buffer
               reload first buffer
               forward = forward + 1
             else Retrieve
               Reached at the end of the file
               terminate lexical analysis
    cases for all other characters
}
```

Regular Expressions :-

Regular expressions are defined over an alphabet. Let Σ be an alphabet set

Regular expressions over Σ and the sets they denote are

ϕ is RE that denotes empty set $\phi = \{\}$

ϵ is RE that denotes set containing ϵ i.e. $\epsilon = \{\epsilon\}$

a is RE that denotes set containing a i.e. $a = \{a\}$.

- If r & s denotes Re that denotes language L(r) and L(s)

- $(r)|(s)$ is RE that denotes lang $L(r) \cup L(s)$
- rs is RE that denotes lang $L(r)\,L(s)$
- $(r)^*$ is " " " " $[L(r)]^*$

→ Algebraic properties of a RE given as follows

Axiom	Description
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is Associative
$(rs)t = r(st)$	concatenation is Associative
$r(s\|t) = rs\|st$	concatenation distribution over |
$\epsilon r = r\epsilon = r$	ϵ is an identity Element for concatenation
$r^* = (r\|\epsilon)^*$	relation b/w r & $*$
$r^{**} = r^*$	$*$ is Idempotent

Ex:- i, $(0+1)^*$ - Denotes all strings of 0's & 1's

ii, $(0+1)^*011$ - all strings of 0's & 1's ends with 011.

iii, $0^*1^*2^*$ - Any no. of 0's followed by any no. of 1's followed by any no. of 2's

iv, $(0+(+2)^+$ - r

v, $a|a^*b$ – a (or) all strings consisting of zero (or) more instances of 'a' followed by 'b'

Regular Definition:– Regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

⋮

$d_n \rightarrow r_n$

Where d_i is the name given to regular expression r_i

i, Pascal identifiers:– Set of strings of letters & digits begin with letter

letter → A|B|...|z|a|b|....|z

digit → 0|1|....|9

id → letter (letter|digit)*

ii, Pascal numbers:–

digit → 0|1|....|9

digits → digit digit*

optional-fraction → (. digits) | ϵ

optional-exponent → (E (+|-|ϵ) digits) | ϵ

num → digits optional-fraction optional-exponent

EXTENSION TO REGULAR EXPRESSION:–

– One (or) more instances :– The unary operator '+' is one (or) more instances. If 'r' is a language L(r) then

v, $a|a^*b$ – a (or) all strings consisting of zero (or) more instances of 'a' followed by 'b'

Regular Definition:- Regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

$\vdots$

$d_n \rightarrow r_n$

Where d_i is the name given to regular Expression r_i

i) **Pascal identifiers:-** Set of strings of letters & digits begin with letter

letter $\rightarrow$ A|B|...|z|a|b|....|z

digit $\rightarrow$ 0|1|....|9

id $\rightarrow$ letter (letter|digit)*

ii) **Pascal numbers:-**

digit $\rightarrow$ 0|1|....|9

digits $\rightarrow$ digit digit*

optional_fraction $\rightarrow$ (. digits) | ϵ

optional_Exponent $\rightarrow$ (E (+|-|ϵ) digits) | ϵ

num $\rightarrow$ digits optional_fraction optional_Exponent

EXTENSION TO REGULAR EXPRESSION:-

– **One (or) more instances :-** The unary operator '+' is one (or) more instances. If 'r' is a RE that denotes a language L(r) then

r^+ is a regular expression that denotes a language $[L(r)]^+$

- 2 algebraic identities

$$r^* = r^+ / \epsilon$$

$$r^+ = rr^* = r^* r$$

(ii) Zero (or) one instance :- The unary post fix operator '?' means zero (or) one instance

$r?$ is shorthand for r/ϵ

(iii) Character classes :- The notation $[xyz]$ denotes a RE $x|y|z$. An abbreviated character class such as $[a-z]$ denotes RE $a|b|\ldots|z$

Identifiers :-

letter → [A-Za-z]

digit → [0-9]

id → letter (letter/digit)*

Numbers :-

digit → [0-9]

digits → digit+

optional_fraction → (. digits)?

optional_Exponent → ((E(+/-)?) digits)

num → digits optional_fraction optional_Exponent

Recognition Of Tokens :-

- Pattern match takes place we Recognise token

stmt → if expr then stmt

→ if expr then stmt else stmt

→ ϵ

expr → term relop term

→ term

term → id
→ num

Grammar for branching statements.

→ The tokens for the above grammar are

if, then, else, relop, id, num

num → digits (.digits)? ((E(+|-)?)digits)?

relop → < | <= | <> | > | >= | = (In PASCAL lang '!=', PASCAL lang '==')

id → letter(letter| digit)*

if → if

else → else

then → then

delim → space | tab | newline | comment

ws → (delim)+

- All terminals are referred as tokens
- If pattern match takes place to the white space then it doesn't return anything to the parser this is called as stripe out

TRANSITION DIAGRAM :-

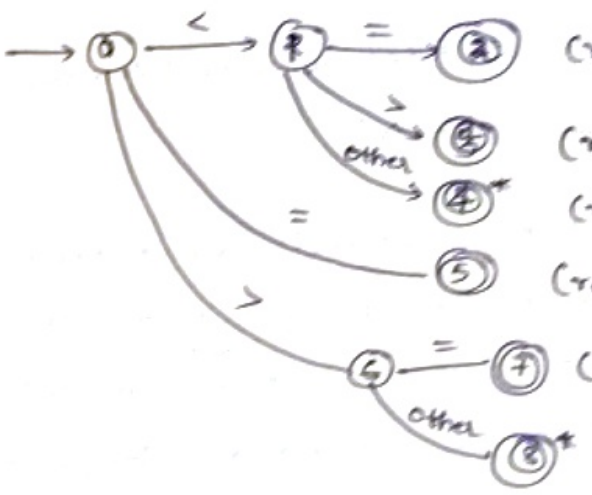
lexeme	Token	Attribute_value
ws	-	-
if	if	-
then	then	-
else	else	-
identifier	id	pointer to symbol table
number	num	pointer to symbol table
<	relop	LT
<=	relop	LE
>	relop	GT

- If lexeme & Token are same then there will be no attribute value

>=	relop	GE
=	relop	EQ
<>	relop	NE

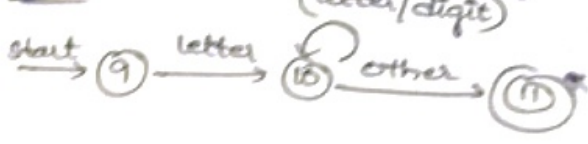
- In relation operators if we have represented lexemes and tokens are same then there is no need for attribute value. But if we represented by some abstract symbol then we have to take represent Attribute values to

TRANSITION DIAGRAM :-

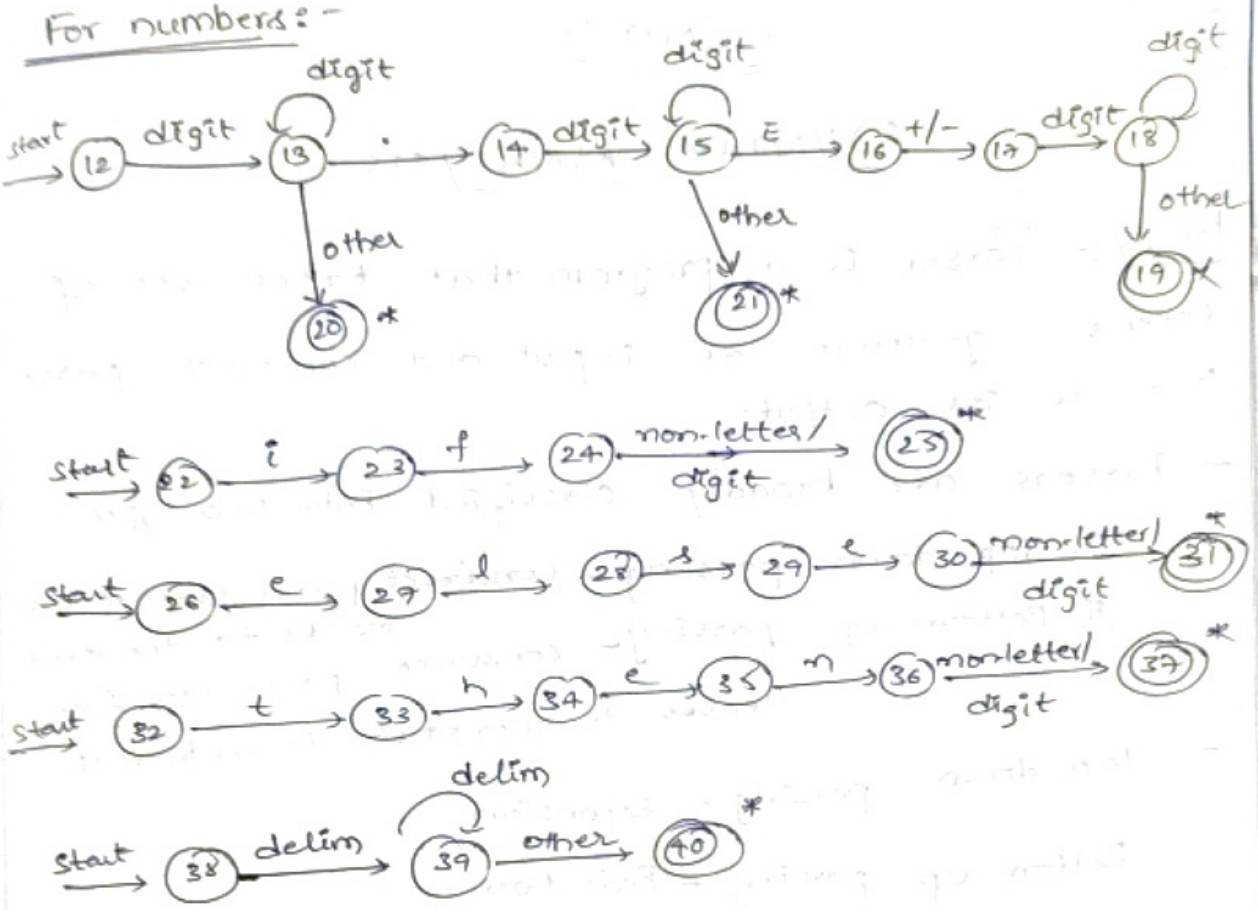


a < b
LT stop pointer
a < b.
a < b
we represent * because after reading < / > it reads another char so to conclude it whether it is LT / LE we represent it as * then it goes to one step back

For id:-



For numbers:-


start
12
digit
13
digit
.
14
digit
15
E
16
+/-
17
digit
18
digit
other
20
*
other
21
*
other
19
*
start
22
i
23
f
24
non-letter/
digit
25
*
start
26
e
27
l
28
s
29
e
30
non-letter/
digit
31
*
start
32
t
33
h
34
e
35
n
36
non-letter/
digit
37
*
start
38
delim
39
delim
other
40
*

4/7/2016

UNIT-II

SYNTAX ANALYSIS

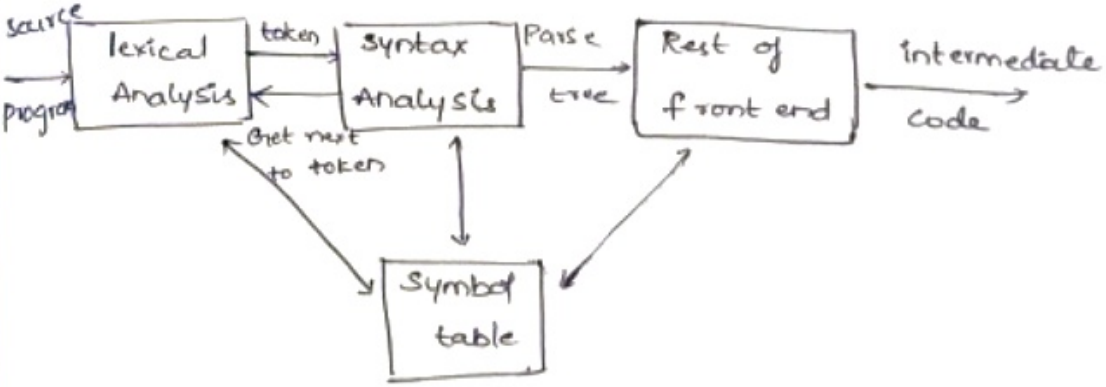
Parser:- Parser is a program that takes set of tokens grammar as input and construct parse tree as its output.

- Parsers are broadly classified into two types
  - i) Top down parsing - Construct parse trees from root to the leaves
  - ii) Bottom up parsing - construct parse tree from leaves & workout towards root

- Top down parsing - Expansion

  Bottom up parsing - Reduction

Role of the parser:-



CONTEXT FREE GRAMMAR:-

- Grammars are introduced to systematically describe the syntaxes of programming language constructs stmts, expr's

terminals - tokens

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

Notational conversions :-

i) Terminals :-

- lowercase letters
- digits
- operators
- paranthesis
- special symbols
- constructs with bold italic face

ii) Non terminals :-

- uppercase letters
- lowercase italic constructs

iii) last symbols of uppercase letters X, Y, Z treated as terminals

iv) u, v, z - strings

v) α, β, γ - string of grammar symbols

vi) set of production $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, $A \rightarrow \alpha_3$ ...
can be rewritten as $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 \ldots$

Ex:- expression → expression + term
→ expression − term
→ term

term → term * factor
→ term / factor
→ factor

factor → (expression)
→ id

By following Notational conversions

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid T/F \mid F$

$F \rightarrow (E) \mid id$

Derivation:- Sequence of application of productions of the grammar to get the desired string.

- Two types of derivation

Left most derivation:- At each step of derivation if a production is always applied to the left most variable

Ex:- $S \rightarrow aAa \mid a$
$A \rightarrow sbA \mid ss \mid ba$

i/p string : aabbaa

$S \rightarrow aAa$
$\rightarrow asbAa$
$\rightarrow aabAa$
$\rightarrow aabbaa$

Right Most derivation:- At each step of derivation if a production is always applied to the Right most variable.

Ex:- $S \rightarrow aAa \mid a$
$A \rightarrow sbA \mid ss \mid ba$

i/p : aabbaa

$S \rightarrow aAa$
$\rightarrow asbAa$
$\rightarrow asbbaa$
$\rightarrow aabbaa$

→ $S \rightarrow ss+ \mid ss* \mid a$

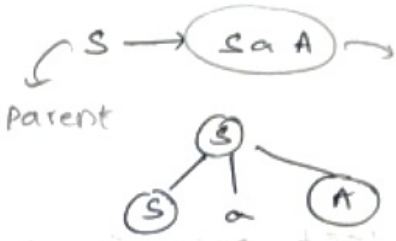
i/p : aa+aa*

$S \rightarrow ss*$
$\rightarrow ss+s*$
$\rightarrow aa+a*$

PARSE TREE:- Graphical representation of a derivation. Three types of nodes

i. Root node - start symbol
ii. Interior node - Non terminals
iii. leaf node - Terminal

- childrens of any parent constructed by some production

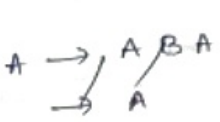


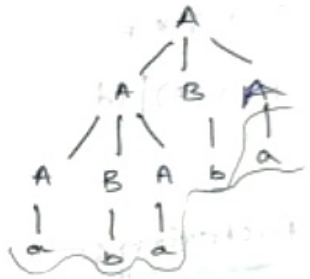
Ambiguous Grammar:— A grammar is said to be ambiguous if it generates more than one parse tree for the same i/p string.

Ex:- A → ABA
A → a
B → b

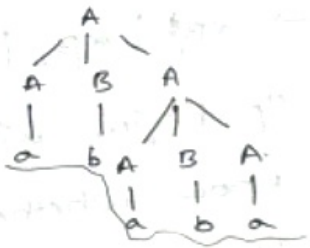
i/p: ababa

A → ABA
→ ABABa
→ abaBa
→ ababa

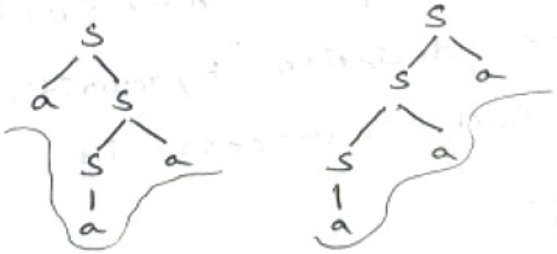




A → ABA
→ ABABa
→ ababa



Ex:- S → as | sa | a



Elimination of ambiguous Grammar:— Ambiguous grammar can be converted into unambiguous if we remove left recursion & left factoring is done by doing

Elimination of left recursion:— A grammar is said to be left recursive if it has a non terminal 'A' such that there is a derivation A → Aα

- Top down parsers cannot handle left recursion

Ex:- $A \rightarrow A\alpha$
$\rightarrow A\alpha\alpha$
$\rightarrow A\alpha\alpha\alpha$
$\rightarrow A\alpha\alpha\alpha\alpha$
:

- It causes infinite loops

- The general form of left recursion is

$$A \rightarrow A\alpha \mid \beta$$

$$\Downarrow$$

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

Ex:- $E \rightarrow E+T$
$T \rightarrow T*F$
$F \rightarrow (E) \mid id$

$\Rightarrow$

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

Left factoring:-

If $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ are three productions and the i/p begins with a non-empty string 'α', we don't know whether to expand $A \rightarrow \alpha\beta_1$ (β_2, β_3) $A \rightarrow \alpha\beta_2$. We differ the decision by rewriting the A's production. Such a process is called left factoring.

i.e., $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

$$\Downarrow \text{ left factoring}$$

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

Ex:- $S \rightarrow iEtS \mid iEtSeS \mid b$ (common prefix)

$E \rightarrow a$

$\Downarrow$

$S \rightarrow iEtSS' \mid b$

$S' \rightarrow \epsilon \mid eS$

$E \rightarrow a$

- Constructing a parse tree for the i/p string, starting from the root and creating the nodes of the parse tree in pre-order.

TOP DOWN PARSING :-

- Begins with start symbol and obtain given input string by applying productions of the grammar.
- Top down parsing resembles left most derivation
- There are 3 types. They are :-
  (i) Back tracking (Brute Force Method) (Trial & Error)
  (ii) Recursive decent parser
  (iii) Non-Recursive predective parsing

Difficulties with top-down parsing :-

(i) Left recursion :-

- There is non-terminal 'A' such that there is a production $A \rightarrow A\alpha$, then it is left recursion
- left recursion causes infinite loops, we need to be eliminate

(ii) Back tracking :-

- If we make sequence of erraneous expansion and finally discover a mismatch. Then we have to undo all those erraneous expansion. this is called back tracking.

iii) Order of alternatives tried :-

- order of alternatives tried can also effect language accepted.

Ex:- $S \rightarrow cAd$

$A \rightarrow abla$

i/p: cabd

⇓

2 ways

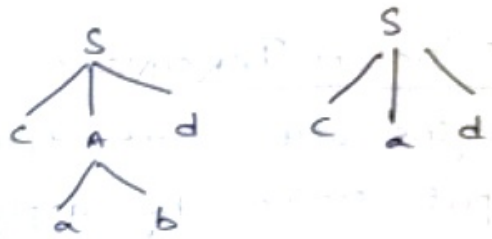
$S \rightarrow cAd$

$S \rightarrow cabd$

(OR)

$S \rightarrow cAd$

$S \rightarrow cabd$



iv) Failure Report :-

- Very little idea about the error i.e., where the error has come in Top-down parser.

BACK TRACKING (Brute force Approach):-

- Top-down parsing with full back-up is called Back Tracking.
- This method operates as follows
  1. Begin with start symbol and expand it by applying first production.
  2. Within the newly expanded substring, choose left most non-terminal expand it by applying first production.
  3. Repeat step 2 for all subsequent non-terminal until step2 cannot be continued further.

- Terminate the process by 2 reasons

i) string has been successfully parsed.

ii) It may results incorrect expansion that does not match the given i/p string.

Ex:- $S \rightarrow aAd \mid aB$

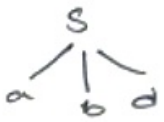
$A \rightarrow b \mid c$

$B \rightarrow ccd \mid ddc$

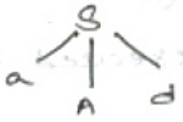
i/p :- accd

Step 1:- $S \rightarrow aAd$

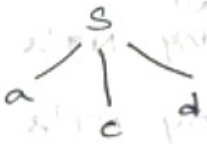
Step 2:-



Step 3:- There is only one N.T so step 2 need not to be repeated

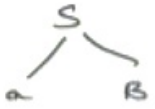


Step 4:-

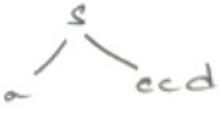


Step 5:-

Step 6:-



Step 7:-



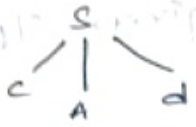
→ Consider a grammar

I/P:- cad

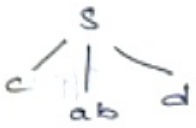
S → cAd

A → ab|a

Step 1:-

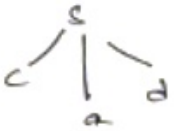


Step 2:-



Step 3:-

Step 4:-



RECURSIVE DECENT PARSER:-

- set of recursive procedures executed to parse given i/p string.
- Procedure is defined for all NT's.
- Procedure is defined for any NT's with the help of their production.

Ex:- $E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow (E) \mid id$

Procedure :—

```
E()
begin
  T();
 Eprim();
 end
Eprim()
 begin
   if (inputsymbol = '+')
   then
    Advance;
    T();
     Eprim();
    end
 end
 T()
 begin
     F();
     Tprim();
  end
 Tprim( )
  begin
   if (input symbol = '*')
     begin
      Advance;
      F();
      Tprim();
     end
  end
```

- Recursive decent parser Encodes state info in its run-time stack & call stack.

- Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

```
F()
begin
  if( input symbol = '(')
    begin
      Advance;
      E();
      if (input symbol = ')')
      Advance;
      else
        error();
      end
      else if (input symbol = 'id')
        Advance;
       else
         error();
      end
```

NOTE:-

In backtracking, unnecessary steps are done & going back to overcome their unnecessary steps. It is its drawback i.e., advancing & going back.

PREDICTIVE PARSER METHOD:—

- To avoid recursive calls we are using non-recursive predictive calls i.e., we are implementing tabular representation of recursive calls. Such a method is called

first & follow sets.

First Set Computation:-

- If α is any string of grammar symbols (i.e., both Terminals & Non-Terminals) then First(α) is set of terminals that begins string derived from α.

If $\alpha \rightarrow \epsilon$ then ϵ is also in first(α)

Rules:-

To compute first(x), for all grammar symbols x. Apply the following rules until no more terminals @, ϵ can be added to any first set.

Rule 1:- If x is terminal, then First(x) = {x}

Rule 2:-

If X is a Nonterminal and $X \rightarrow a\alpha$ (a: Terminal) is a production then add a to First(X)

If $X \rightarrow \epsilon$ is a production then add ϵ to First(X)

Rule 3:-

If X is a non terminal and $X \rightarrow Y_1 Y_2 \cdots Y_n$ is a production if $Y_1 Y_2 \cdots Y_{i-1} \rightarrow \epsilon$ then add every non-ϵ symbols of First(Y_i) to First(X)

Ex:-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

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

First(E') = { +, ϵ }

First(T') = { *, ϵ }

→ S → (L) | a

L → LS | S

⇓

S → (L) | a

L → SL'

L' → SL' | ϵ

S → (L)
⇒ (LS)
⇒ (SS)

First(S) = { (, a }

First(L) = First(S) = { (, a }

First(L') = { (, a, ϵ }

Follow Set Computation:-

Generally follow will be computed only for Non-terminals. For Non-terminal A follow(A) be the set of terminals that can appear immediately to the right of A in some sentential form.

- To compute follow set first we have to compute first set.
- To compute follow(A) for all non terminals. Apply the following rules until no more symbols (or) terminals can be added to any follow set.

In derivation
Every expansion is called sentential form

Rule 1:- $ is
the st

Rule 2:-
If there is
everything
in follow(B)

Rule 3:-
If there is
A → αB (β) A —
First(β) conta
Follow(A) is

Ex:- E → TE'
E' → +TE' | ϵ
T → FT'
T' → *FT'
F → (E) | id

Follow(E) = {
Follow(E') =
Follow(T) =
Follow(T') =
Follow(F) =

→ Follow(S) =
Follow(L) =
Follow(L') =

Rule 1:- $ is in follow(S), where S is the start symbol.

Rule 2:-

If there is a production $A \longrightarrow \alpha B \beta$ then everything in first(β) except for ϵ is in follow(B)

Rule 3:-

If there is a production $A \longrightarrow \alpha B$ (β), $A \longrightarrow \alpha B \beta$ where first(β) contains ϵ then everything in follow(A) is in follow(B)

$ - default symbol

Ex:- $E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow (E) \mid id$

follow(E) = {$,)}

follow(E') = {$,)}

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

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

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

→ follow(S) = {$, (, a,)}

follow(L) = {)}

follow(L') = {)}

Construction of predictive parsing table:-

M[r, c]

Row name - NT

Column name - Terminal

I/p :- Grammar 'G'

O/p :- Parsing table M

Method:-

Rule 1:- For each production $A \rightarrow \alpha$ of the grammar do the following

Step I:- For each terminal 'a' in First(α) add $A \rightarrow \alpha$ to M[A, a]

Step Rule 2:- If ϵ is in First(α) add $A \rightarrow \alpha$ to M[A, b] for each terminal b in Follow(A)

Step Rule 3:- If ϵ is in First(α) & $ is in Follow(A) add $A \rightarrow \alpha$ to M[A, $]

Rule 2:- Make each undefined entry of M be an Error

Parsing Table:-

i) $E \rightarrow TE'$ - First(TE') = First(T) = { (, id }

ii) $E' \rightarrow +TE'$ - First($+TE'$) = { + }

iii) $E' \rightarrow \epsilon$ - Follow(E') = { $,) }

iv) $T \rightarrow FT'$ - First(FT') = First(F) = { (, id }

v) $T' \rightarrow *FT'$ - First($*FT'$) = { * }

vi) $T' \rightarrow \epsilon$ - Follow(T') = { +, $,) }

vii) $F \rightarrow (E)$ - ~~follow~~ First((E)) = { (}

viii) $F \rightarrow id$ - First(id) = { id }

Ex:-

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

- When the top of the stack is terminal and it is the required i/p then we have to pop it.
- When the top of the stack is NT then replace the NT by its reproduction rule in reverse order so that the first production should be at the T.O.S (Stack

It is represented as left to right)

- When there is a match in the stack & input then pop the element in the stack & advance the i/p pointer

Ex:-

Stack	Input	Production
$\$ E$	$id+id\$$	
$\$ E'T$	$id+id\$$	$E \to TE'$
$\$ E'T'F$	$id+id\$$	$T \to FT'$
$\$ E'T'id$	$id+id\$$	POP
$\$ E'T'$	$+id\$$	
$\$ E'$	$+id\$$	$T' \to \epsilon$
$\$ E'T+$	$+id\$$	POP
$\$ E'T'F$	$id\$$	$T \to FT'$

$ E' T' id	id $	POP
$ E' T'	$	
$ E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

When we get '$' in both stack & I/P then the input is parsed & the string is accepted.

→

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

$S \to (L)$ - First[(L)] = { (}

$S \to a$ - First(a) = {a}

$L \to SL'$ - First(SL') = First(S) = {(, a}

$L' \to SL'$ - First(SL') = {(, a}

$L' \to \epsilon$ ∴ Follow(L') = {) }

- Initial configuration

stack	I/p	Operation
$ starting-symbol	w$	

$ E' T' id	id $	POP	
$ E' T'	$		
$ E'	$	$T' \to \epsilon$	
$	$	$E' \to \epsilon$	

When we get '$' in both stack & i/p then the input is parsed & the string is accepted.

→

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

$S \to (L)$ - First[(L)] = { (}

$S \to a$ - First(a) = { a }

$L \to SL'$ - First(SL') = First(S) = { (a }

$L' \to SL'$ - First(SL') = { (, a }

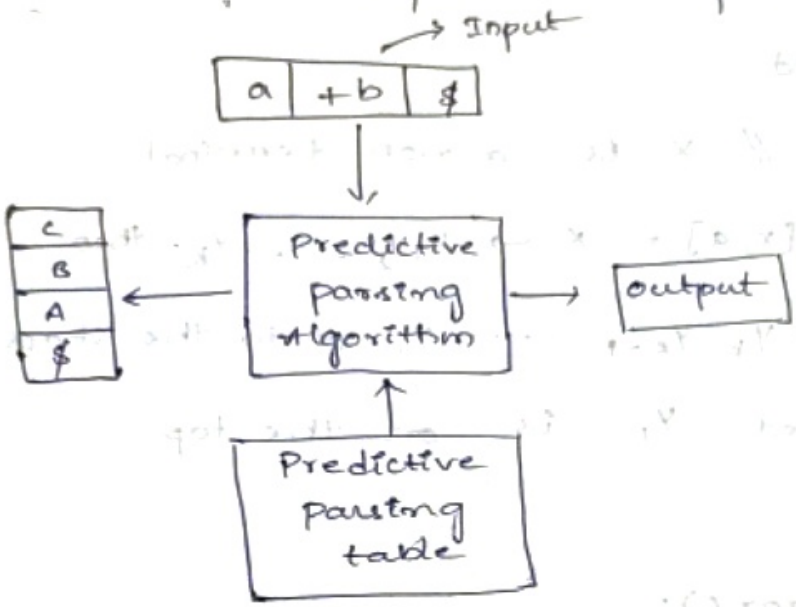
$L' \to \epsilon$ = Follow(L') = {) }

- Initial configuration

stack | I/P | Operation

$ starting-symbol | w$

PREDICTIVE PARSING ALGORITHM :-



- Parsing table, input string ending with $
- Based on action of algorithm it will push string of grammar symbols on to stack which is preceeded by $.
- Initial configuration of predictive parsing input w = id + id

stack	input
$ s	id + id $

Algorithm :-

Repeat
begin
  Let x be the start symbol and
  a be the next input symbol

  If x is a terminal
  begin
    if x = a
    begin
      Pop x from stack and
      add value advance input pointer

```
        else
            error
        end
    else        // X is a non terminal
begin
    if M[X, a] = X → Y1 Y2 ..... Yk then
        push Yk Yk-1 ..... Y1 on to the stack
        provided Y1 is on the top
    else
        error();
end
until X = $
```

Ex:- (id + id) $

Stack	Input	Operation
$ E	(id+id)$	
$ E' T	(id+id)$	E → T E'
$ E' T' F	(id+id)$	T → F T'
$ E' T') E (	(id+id)$	F → (E)
$ E' T') E	id+id)$	POP
$ E' T')	+id)$	POP
$ E' T') E' T	id+id)$	E → T E'
$ E' T') E' T' F	id+id)$	T → F T'
$ E' T') E' T' id	id+id)$	F → id POP
$ E' T') E' T'	+id)$	POP

$ E'T'	+ id) $	$T' \rightarrow \epsilon$
$ E'T' +	+ id) $	POP $E' \rightarrow +TE'$
$ E'T'	id) $	POP
$ E'T'F	id) $	$T \rightarrow FT'$
$ E'T'id	id) $	$F \rightarrow id$
$ E'T'	) $	POP
$ E'	) $	$T' \rightarrow \epsilon$
$ E'T'	) $	$E' \rightarrow \epsilon$
$ E'T'	$	POP
$ E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

LL(1) Grammar :-

A grammar is said to be LL(1) whose parsing table has no multiple defined entries then it is said to be LL(1) grammar.

First 'L' stands for left to right scanning of input.

Second 'L' stands for left most derivation

'1' stands for look ahead at most one symbol.

- The grammar who table has multiple defined entries

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

$E \rightarrow b$

$\Downarrow$

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid e$

$E \rightarrow b$

First (S) = { $, a } ; First (S') = { e, ϵ }

First (E) = { b }

Follow (S) = Follow (S') =

Follow (S) = { $,

$S \to iEtSS' / a$

$S' \to eS / \epsilon$

$E \to b$

Follow (S) = { $, e }

Follow (S') = { $, e }

Follow (E) = { t }

- If we have to find follow of a NT then we have to find the First of its next symbol i.e., If it is terminal write directly.

- If it is NT write the non-ϵ symbols directly to the Follow (NT) and if it has ϵ-symbols write its parent's Follow set i.e., Follow (parent)

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

- A grammar is said to be LL(1) if and only if there is a production $A \to \alpha / \beta$ and holds the following conditions

i. No terminal 'a' begins with strings derived from both α and β

ii. Atmost one of α and β can derive empty string

(ii) If $\beta \to \epsilon$ then α doesn't derive any strings beginning with a terminal in follow(A)

(or)

If $\alpha \to \epsilon$ then β doesn't derive any strings beginning with a terminal in follow(A)

Ex:- $S' \xrightarrow[A]{} \underset{\alpha}{e}\, \underset{\beta}{s} \mid \epsilon$

In α - first(α) values should not be equal to follow(parent)

ERROR Recovery STRATEGIES in Predictive Parsing:-

Error is detected in 2 ways

(i) When a terminal on top of the stack does not match the current i/p symbol.

(ii) When a NT A on top of the stack a be the current i/p symbol if M[A, a] is a blank.

Recovery

- There are two ways for recovering from Error

(i) Panic mode Error recovery

(ii) Phase level Error recovery

Panic Mode Error Recovery:- This method is based on the idea of skipping input symbol until a token in set of synchronising tokens appear

- Synchronising sets are chosen so that parser quickly recover from Error.

Some methods for/choosing symbol

Some methods for choosing symbols for Synchronising sets

(i) All symbols of follow(A) added to Synchronising set for Nonterminal 'A'

(ii) Symbols that begin high level constructs can be added to synchronizing set
Ex:- statement, Expressions, blocks etc

(iii) First(A) Symbols can be chosen for synchronising sets for nonterminal.

(iv) ε production that derive empty string ε can be added to synchronizing set

Method 1 Example:-

Follow set is taking as Example to choose symbols of synch set.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Follow(E) = {), $ }

Follow(E') = {), $ }

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

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

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

Synch set E = {), $ }

Synch set E' = {), $ }

Synch set T = { +,), $ }

Synch set T' = { +,), $ }

Synch set F = { *, +,), $ }

E
E'
T
T'
F

Error Recove

- Parsing
rules

(i) Parser
blank

(ii) If th
of the
continu

(iii) If a
not m
is po

- If top
should

Stack im

Stack

$ E

$ E

$ E' T

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

Error Recovery Rules:-

- Parsing can be continued with the following rules

i) Parser look up an entry $M[A,a]$ for which it is blank then i/p symbol 'a' is skipped out.

ii) If the entry is synch, then NT on the top of the stack is popped then parsing will continue.

iii) If a token (terminal) on top of the stack does not match the current i/p symbol then token is popped out from stack.

- If top of the stack is start symbol then we should not pop it. we skip the i/p symbol.

Stack implementation for Erraneous i/p string)id*+id:-

Stack	Input	Output
$ E	)id*+id$	
$ E	id*+id$	Error skip)
$ E'T	id*+id$	$E \to TE'$

$E'T'F	id * + id $	$T \rightarrow FT'$
$E'T'id	id * + id $	$F \rightarrow id$
$E'T'	* + id $	POP
$E'T'F*	* + id $	$T' \rightarrow *FT'$
$E'T'F	+ id $	POP
$E'T'F	+ id $	M[F, +] = synch so POP F.
$E'T'id	id $	$F \rightarrow id$
$E'T'	$	POP
$E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

Phase level Error Recovery :—

- In phase level error recovery the blank in the table are filled with pointers to error recovery routine.
- The error routine may change insert & delete symbols on the i/p and issue appropriate error message.

→

$E'T'F	+ id $	M[F, +] = synch POP F.
$E'T'	+ id $	
$E'	+ id $	$T' \rightarrow \epsilon$
$E'T+	+ id $	$E' \rightarrow +TE'$
$E'T	id $	POP

$ E' T' F	id $	T → FT'
$ E' T' id	id $	F → id
$ E' T'	$	pop
$ E'	$	T' → ε
$	$	E' → ε.

- Sometimes user may give some wrong i/p then these Error recovery strategies itself will correct the given wrong statements i.e, they will check where the Error is araised and it will try to correct it & will give correct results

19/2/2016

UNIT-III

BOTTOM UP PARSING

- Begins with input string and obtain start symbol of the grammar.
- Construct parse from leaves and continue towards root.

Ex:- $E \rightarrow E+T$

$E \rightarrow T$

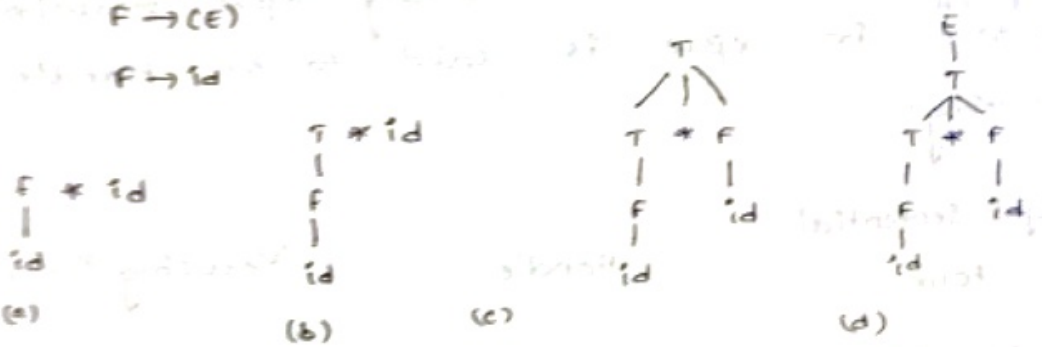
$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

i/p: id * id



Right most derivation for the i/p: id * id

$E \rightarrow \underline{T}$

$\rightarrow \underline{T * F}$

$\rightarrow T * \underline{id}$

$\rightarrow \underline{F} * id$

$\rightarrow \underline{id} * id$

- Bottom up parsing resembles "RMD in reverse."

REDUCTION:-

- Bottom up parsing is a process of reducing string 'w' to start symbol of the grammar.
- At each step of reduction substring matching the production right hand side is replaced

by left hand side NT

- Reduction is a reverse step of derivation

HANDLE:- Handle is a substring that matches to the production of right hand side

Ex:- for the above example handles are

id, F, id, T*F, T (i.e., which are reduced to its left hand side)

Handle Pruning:- If there is a production $A \to \beta$ then β is said to be handle since it is reduced to 'A' in the string '$\alpha\beta w$', reducing $\beta \to A$ in $\alpha\beta w$ is said to be handle pruning.

Right Sentential form	Handle	Reducing Production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$
E (Starting Symbol)		

Shift Reduce Parsing:-

- It is a simple bottom up parsing
- There are 4 actions in shift reduce parsing.

Actions

i. Shift :- Shi

ii. Reduce :- If
re

iii. Accept :- Wh
+
then

iv. Reject :- Err

Ex:- Stack Imple

- If any han
the top of
to shift
- We have t
until we
the stack.

Stack

$
$ id
$ E
$ E +
$ E + id
$ E + E
$ E

Ex:- $
$a
$S
$Sa
$SS
$SS+
$S
$Sa
$SS

Actions:

i. Shift :- Shift i/p Symbols on to the stop of the stack.

ii. Reduce :- If a handle appears on top of the stack reduce it by left hand side NT

iii. Accept :- When we get start symbol on the top of the stack & i/p $ is present then it is accepted otherwise not accepted

iv. Reject :- Error:- state where we cannot perform actions i.e., shift/Reduce

Ex:- Stack Implementation:-

- If any handle is not present at the top of the stack then we have to shift i/p to stack.
- We have to shift i/p's to stack until we get handle on the top of the stack.

$E \to E+E$
$E \to E*E$
$E \to E-E$
$E \to E/E$
$E \to id$

Action i/p:- id+id

Stack	Input	Operations
$	id+id$	shift id
$ id	+id$	Reduce
$ E	+id$	shift +
$ E +	id$	shift id
$ E + id	$	Reduce
$ E + E	$	Reduce
$ E	$	Accept

Ex:-

$S \to SS+$
$S \to SS*$
$S \to a$
i/p: aa+a*

$	aa+a*$	shift a
$a	a+a*$	Reduce
$S	a+a*$	shift a
$Sa	+a*$	Reduce
$SS	+a*$	Reduce shift +
$SS+	a*$	Reduce
$S	a*$	shift a a
$Sa	*$	Reduce
$SS	*$	shift *

$SS+ $ Reduce
$S $ Accept

Initial Configuration of shift reducing parser:

Stack	Input	Action
$	w$	

Conflicts in shift reducing parsing:-

- There are two types of conflicts
  i. shift / reduce conflict
  ii. Reduce / Reduce conflict

Shift / Reduce Conflict:-

We know the content of the stack and curr i/p symbol (or) next i/p symbol but we cannot decide whether to take shift action (or) reduce action.

Pg NO - 281

Ex:- Dangling else grammar

stmt → if expr then stmt
→ if expr then stmt else stmt
→ a
Expr → b.

- We reached one configuration in stack implementation

Stack	Input	Action
$ if expr then stmt	else stmt $	

- Here we can perform 2 action i.e., If we can reduce it by stmt (or), we can shift input to the stack. This conflict is called shift / Reduce conflict.

Reduce / R
We
we cann
-ons to
conflict.
Ex:- E
A
B
we rea
implement
stack
$ .....
Here
so here
result to
Reduce

Operator

Operator
to be
f satisf
i. No
ii. No t
right
Ex:-

E

Reduce / Reduce conflict:—

We know the content of the stack but we cannot decide which of the several reductions to make, choose is called Reduce/Reduce conflict.

Ex:- $E \rightarrow ABc$

$A \rightarrow b \mid c$

$B \rightarrow b$

we reached one configuration in stack implementation.

Stack	i/p	Action
$ ..... b	... $	

Here we can reduce 'b' as $A \rightarrow b$ or $B \rightarrow b$ so here we have two reductions which will result to conflict, such conflict is called Reduce/Reduce conflict.

Operator Precedence Parser:— (Bottom up parsing)

Operator precedence Grammar:— A grammar (CFG) is said to be operator precedence grammar when it satisfies two rules.

i. No production derives empty string

ii. No two adjacent NT's present in any right hand side production.

Ex:- $E \rightarrow EAE \mid id$

$A \rightarrow + \mid - \mid * \mid /$

} It is not Operator precedence grammar since it violates Rule-2.

⇓

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

Precedence Relation:-

- identifiers has the precedence over all operators

- Operator grammar relay on precedence relation

Relation	Meaning
$a \lessdot b$	a yields precedence to b
$a \doteq b$	a has same precedence as b
$a \gtrdot b$	a takes precedence over b

- To differentiate precedence relation and relational Operator we will represent it by dot i.e, $\lessdot$, $\gtrdot$:

Precedence Table:-

	id	+	*	$
id	-	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	←

- Insert precedence relations b/w every terminal of the string
- Precedence relations allows to identify handles as follows:

i. Scan the string from left to right until seeing '$\gtrdot$'

ii. Scan the string backwards from right to left until seeing '$\lessdot$'

iii. Everything b/w two relations $\lessdot$ and $\gtrdot$ forms the handle

- we will insert $ on both sides of the string inorder to fix the precedence relation for first & last symbol of I/P

Ex:- $ id +

$ $\lessdot$ id

$ $\lessdot$ +

$ $\lessdot$

$ $\lessdot$

$ $\lessdot$

$ $

Operator P

let '@'

I/P Symbo

begi

if (a

else

else

else

end

Ex:-

Ex:- $ id1 + id2 * id3 $

→ Handle

$ <· id1 ·> + <· id2 ·> * <· id3 ·> $

$ <· + <· id2 ·> * <· id3 ·> $

$ <· + <· * <· id3 ·> $

$ <· + <· * ·> $ - Here it evaluates the exp by this operator

$ <· + ·> $

$ $ — Accept

Operator Precedence Parsing Algorithm:—

Let 'a' be the top of the stack, 'b' be the i/p symbol

begin

if (a is $ and b is $)

return

else if (a <· b or a ≐ b) then

push 'a' onto the stack

advance ip_pointer to point next i/p symbol

else if (a ·> b)

pop the terminals until top of the stack terminal is related by '<·' to the most recently popped.

else

error()

end

Ex:-	Stack	Input	Operation
	$	id + id * id $	push id
	$id	+ id * id $	pop
	$ +	+ id * id $	push
	$ +	id * id $	push

$+id	* id $	Pop
$ +	* id $	Push
$ + *	id $	push
$ + * id	$	Pop
$ + *	$	Pop
$ +	$	Pop
$	$	

Precedence functions :-

- Inorder to reduce the size of precedence table. We are moving to predence function
- The size of Precedence table = $O(n^2)$
  size of precedence functions - $O(2n)$

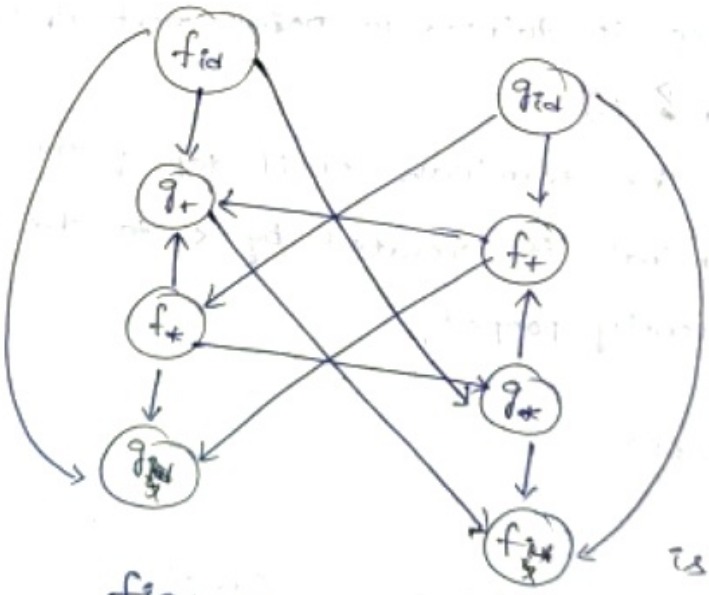


fig: Precedence graph

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

- Values given in the table is the length of largest procedure path from the respective function

Ex:- $g_{id} - f_* - g_* - f_+ - g_+ - f_\$$

LR PARSERS:—

Generally we consider it as LR(k) parser

where L = it stands for left to right scanning of input.

R = it stands for RMD in reverse

k = no. of i/p symbols lookahead while taking parsing decision.

Why LR Parser:—

- LR parsers can handle all types of programming language constructs. (i.e., if, else ..)
- LR parsers are non backtracking shift reduce parser i.e, it is more efficient when compared to basic shift reduce parser. (i.e., here it tells about the conflicts of shift reducing parser)
- LR parsers can detect Errors as soon as it is possible.
- LR parsers handles superset of grammars of predictive parsers.

Types Of LR Parsers:— There are three types of LR parsers. They are:—

i. SLR (simple LR) — LR(0) items

ii. CLR (Canonical LR) } — LR(1) items

iii. LALR (Lookahead LR) } (C compiler is designed by LALR parser)

Simple LR Parser:-

Item (or) LR(0) item

Item is a production of grammar G with dot (.) inserted at some point at right hand side.

Ex:- $A \longrightarrow XYZ$ is a production

Items:- $A \longrightarrow \cdot XYZ$ (we have yet to see XYZ)

$\rightarrow X \cdot YZ$ (we saw x terminal & yet to see YZ)

$\rightarrow XY \cdot Z$

$\rightarrow XYZ \cdot$

- If dot is extreme right then we have to go for reduce

CONSTRUCTION OF SLR PARSING TABLE:- (Closure, goto, augmented)

Augmented Grammar:- A grammar G' is a grammar for G where we will insert an extra production i.e., the starting symbol of grammar G

i.e., $G' \longrightarrow G$.

- we can't construct parser if we didn't consider $S' \rightarrow S$

- The items of the above production is

$G' \longrightarrow \cdot G$ - (starting)

$G' \longrightarrow G \cdot$ (parsing completed)

- we have to look add for productions until we did not get a new NT preceeded by dot (.). (i.e., items)

Ex:-

G	G' (Augmented Grammar)	Closure
$E \rightarrow E+T$	$E' \rightarrow E$	$I_0 : E' \rightarrow \cdot E$
$E \rightarrow T$	$E \rightarrow E+T$	$E \rightarrow \cdot E+T$
$T \rightarrow T*F$	$E \rightarrow T$	$E \rightarrow \cdot T$
$T \rightarrow F$	$T \rightarrow T*F$	$T \rightarrow \cdot T*F$
$F \rightarrow (E)$	$T \rightarrow F$	$T \rightarrow \cdot F$
$F \rightarrow id$	$F \rightarrow (E)$	$F \rightarrow \cdot (E)$
	$F \rightarrow id$	$F \rightarrow \cdot id$

Goto :-

$I_0 : E' \rightarrow \cdot E$

$E \rightarrow \cdot E$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T$

$T \rightarrow \cdot$

$F \rightarrow \cdot$

$F \rightarrow \cdot$

Goto (I_0, E)

$I_1 : E \rightarrow E \cdot$

$E \rightarrow E$

Goto $(I_0, T$

$I_2 : E \rightarrow T \cdot$

$T \rightarrow T \cdot$

Goto $(I_0,$

$I_3 : F \rightarrow F \cdot$

Goto $(I_0,$

$I_4 : F \rightarrow$

$E \rightarrow$

$E \rightarrow$

$T \rightarrow$

$T \rightarrow$

$F \rightarrow$

$F \rightarrow$

Goto (I_0

$I_5 : F$

Goto :-

$I_0 : E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Goto (I_0, E)

$I_1 : E' \to E \cdot$

$E \to E \cdot + T$

Goto (I_0, T)

$I_2 : E \to T \cdot$

$T \to T \cdot * F$

Goto (I_0, F)

$I_3 : T \to F \cdot$

Goto $(I_0, ()$

$I_4 : F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Goto (I_0, id)

$I_5 : F \to id \cdot$

Goto $(I_1, +)$

$I_6 : E \to E + \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Goto $(I_2, *)$

$I_7 : T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Goto (I_4, E)

$I_8 : F \to (E \cdot)$

$E \to E \cdot + T$

Goto (I_4, T)

$I_2 : E \to T \cdot$ } I_2
$T \to T \cdot * F$ } Here we will give the same name bcz it has the same items

Goto (I_4, F)

$F \to (F)$

$E \to$

$I_3 : T \to F \cdot$

Goto $(I_4, ()$

$I_4 : F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Goto (I4, id)

I_5: F → id.

Goto (I_6, T)

I_9: E → E+T.

T → T*F.

Goto (I_6, F)

I_3: T → F.

Goto (I_6, ()

I_4: F → (.E) } I_4

Goto (I_6, id)

I_5: F → id.

Goto (I_7, F)

I_{10}: T → T*F.

Goto (I_7, ()

I_4: F → (.E) } I_4

Goto (I_7, id)

I_5: F → id.

Goto (I_8,))

I_{11}: F → (E).

Goto (I_8, +)

I_6: E → E+.T

T → .T*F

T → .F

F → .(E)

F → .id

Goto (I_9, *)

I_7: T → T*.F

F → .(E)

F → .id

Drawbacks

- SLR is not parsed by every grammar so this is the conflict we are going to CLR
- CLR is parsed by every language constructs.

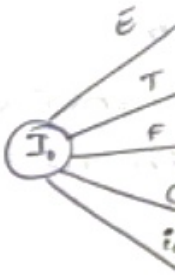
S → L=R/R

L → *R/id

R → L

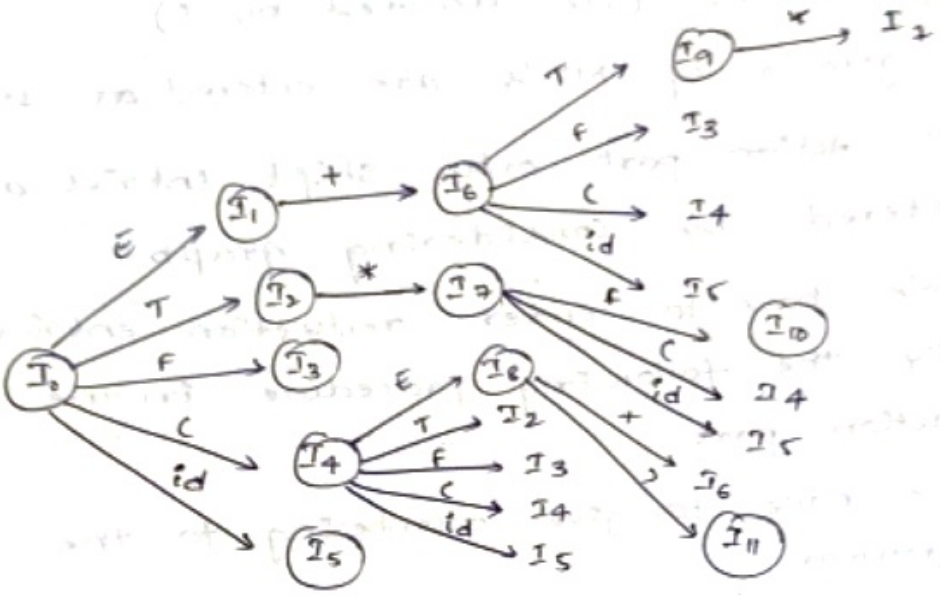
for this grammar SLR Parsing table has multiple entries

GRAPH :-



	+	
0		
1	S_6	
2	r2	
3	r_4	
4		
5	r_6	
6		
7		
8	S_6	
9	r_1	
10	r_3	
11	r_5	

GRAPH :-



	Action						Go to		
	+	*	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1	S_6					ACCEPT			
2	r_2	S_7		r_2		r_2			
3	r_4	r_4		r_4		r_4			
4			S_4		S_5		8	2	3
5	r_6	r_6		r_6		r_6			
6			S_4		S_5			9	3
7			S_4		S_5				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			

- All goto 's of terminals are entered as shift entries (i.e., denoted by S)
- All goto 's of NT 's are entered as state no.
- In Action part only shift entries are entered by considering graph
- We have to enter reduction entries by the following procedure in the action part.

(i) We have to give numbering to the grammar

i.e., r_1 1) $E \rightarrow E+T$ (reduction at 1st production)

r_2 2) $E \rightarrow T$

r_3 3) $T \rightarrow T*F$

r_4 4) $T \rightarrow F$

r_5 5) $F \rightarrow (E)$

r_6 6) $F \rightarrow id$

(ii) First & follow sets of the grammar

First(E) = First(T) = First(F) = { (, $ id }

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

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

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

(iii) We have to check in which the respective production 's can be reduced (i.e., dot (.) will be at the right most)

- In that particular item set we have to enter the reduction entries for the respective follow (Grammar production)

$E \rightarrow E+T.$

Ex:- $r_1 : E \to E+T$.

In I_9 item set

we have to enter r_1 at follow(E) = {+,), $}

in 9^{th} state

We have to enter r_1 in 9^{th} state for follow(E) = {+,), $}

- We will give numbering to original grammar but not for augumented grammar because $S' \to S.$ is an accept state where we cannot reduce it.
- we have to enter 'Accept' entry at $ terminal.

Closure (I) :-

Begin

J = I ;

repeat

for each item $A \to \alpha . B\beta$ in I

for each production $B \to \gamma$ add

$B \to .\gamma$ to J if it is not already there

until no more items added to J

end

Goto (I, X) :-

Goto(I,X) where I is the item set and X is (terminal / NT)

the grammar symbol

Goto(I, X) is the closure set of all items $A \to \alpha X . \beta$ if $A \to \alpha . X \beta$ is in I

Construction Of SLR Parsing table Algorithm: (LR(0) items)

Input :- Grammar 'G'

Output :- Parsing table

Method :-

Step 1 :- collect LR(0) item sets

$$C = \{I_0, I_1, I_2, \ldots I_n\}$$

Step 2 :- Parsing table action for state i is constructed as follows

(i) If $A \to \alpha . a\beta$ is in I_i and $goto[I_i, a] = I_j$ then set action $[i, a]$ = "shift j"

(ii) If $A \to \alpha .$ is in I_i then set action $[i, a]$ to "reduce $A \to \alpha$" for each terminal a in Follow(A).

(iii) If $S' \to S.$ is in I_i then set action$[i, \$]$ to "accept"

Step 3 :- // Goto (Non terminals)

If $A \to \alpha . X \beta$ (X: NT) is in I_i and $goto[I_i, X] = I_j$ then set action $[i, X] = j$

Step 4 :-

All entries which are not defined by Rule 2 and Rule 3 are considered as Errors.

Step 5 :-

Initial State is constructed from $S' \to .S$

Canonical LR

LR(1) Item :-

The general

Closure (I) :-

Begin

repeat

for each

for ea

termina

to I if

until

end

Goto (I, x) :-

Goto (I, x)

a gramma

all items

is in I.

Construction

Grammar :-

G

S → CC

C → cC

C → d

Canonical LR Parser (CLR) More Powerful Parser:-

(LR(1) items)

LR(1) Item:-

The general form of LR(1) item is as follows

$$[A \longrightarrow \alpha \cdot B\beta, a]$$

- core (same as LR(0))
- lookahead symbol

Closure (I):-

Begin

repeat

for each item $[A \longrightarrow \alpha \cdot B\beta, a]$ is in I

for each production $B \rightarrow \gamma$ and for each terminal b in First(βa) add $[B \rightarrow .\gamma, b]$ to I if it is not already there

until no more items are added to I

end

Goto(I, x):-

Goto (I, x) where I is item set and x is a grammar symbol is the closure set of all items $[A \longrightarrow \alpha x \cdot \beta, a]$ if $[A \longrightarrow \alpha \cdot x\beta, a]$ is in I.

Construction of CLR Parsing table:-

Grammar:-

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

Closure

closure ([S' → · S, $])

first ($) = {$}

first (βa) = first (C $)

first (C $) = first(C), {.

[S' → · C C, $]

[C → · c C, c/d]

[C → · d , c/d]

Go to :-

Go to (I_0, S)

I_1 : [S' → S. , $]

Go to (I_0, C)

I_2 : [S' → C. C, $]

[C → · c C, $]

[C → · d , $]

Go to (I_0, c)

I_3 : [C → c. C, c/d]

[C → · c C, c/d]

[C → · d , c/d]

Go to (I_0, d)

I_4 : [C → d. , c/d]

Go to (I_2, C)

I_5 : [S' → C C. , $]

Go to (I_2, c)

I_6 : [C → c. C, $]

[C → · c C, $]

$[C \to \cdot d, \$]$

Goto (I_2, d)

$I_7 : [C \to d\cdot, \$]$

Goto (I_3, C)

$I_8 : [C \to cC\cdot, c/d]$

Goto (I_3, c)

$I_3 : [C \to c\cdot C, c/d]$

$[C \to \cdot cC, c/d]$

$[C \to \cdot d, c/d]$

Goto (I_3, d)

$I_4 : [C \to d\cdot, c/d]$

Goto (I_6, C)

$I_9 : [C \to cC\cdot, \$]$

Goto (I_6, c)

$I_6 : [C \to c\cdot C, \$]$

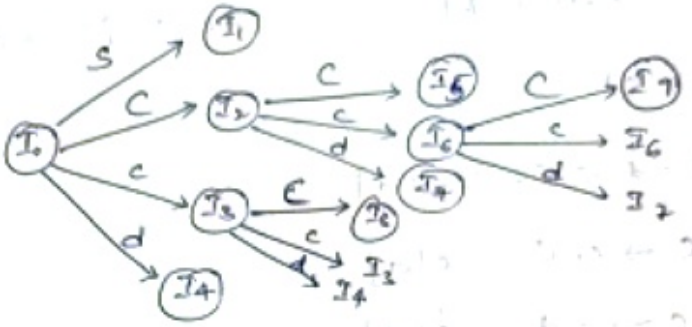
$[C \to \cdot cC, \$]$

$[C \to \cdot d, \$]$

Goto (I_6, d)

$I_7 : [C \to d\cdot, \$]$

Graph:-



	Action			Go to	
	c	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			
9			r_2		

Construction of CLR parsing table Algorithm:-

Input :- Grammar 'G'

output :- Parsing table (i.e., CLR)

Method :-

Step 1 :- Collect LR(1) item sets

$$C = \{ I_0, I_1, I_2, \ldots I_n \}$$

Step 2 :- Parsing table action for state i is constructed as follows

i. If $[A \to \alpha \cdot a\beta, b]$ is in I_i and $goto[I_i, a] = I_j$ then set action $[i, a]$ = "shift j"

ii. If $[A \to \alpha \cdot, a]$ is in I_i then set action $[i, a]$ to "reduce by $A \to \alpha$"

iii. If $[S' \to S \cdot, \$]$ is in I_i then set action $[i, \$]$ to "Accept".

Step 3 :-

If $[A \to \alpha \cdot X\beta, a]$ is in I_i and $goto[I_i, X] = I_j$ then set action $[i, X] = j$.

Step 4 :-

All entries which are not defined by step 2 and steps are considered as errors

Step 5 :-

Initial state is constructed from $[S' \to \cdot S, \$]$

Drawbacks :-

- It results more no. of states and item sets when compared to SLR parsing parser

LALR (Lookahead LR) :-

- It is more efficient and powerful
- LALR states are equal to SLR states

- Idea

i. Collect LR(1) item sets

ii. Identify LR(1) items that are having same core and different look aheads and merge them together into one set.

i.e., I_3, I_6 & I_4, I_7 & I_8, I_9

I_{36} : $[C \to c.C, c/d/\$]$
$[C \to .cC, c/d/\$]$
$[C \to .d, c/d/\$]$

I_{47} : $[C \to d., c/d/\$]$

I_{89} : $[C \to cC., c/d/\$]$

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

Construction

Input :- Gr

Output :- L

Method :-

Step1 :- Co

Collec

Step2 :-

Find the

different

by their

Step3 :- Let

LR(1) item

State i a

manner a

Step4 :-

The Goto

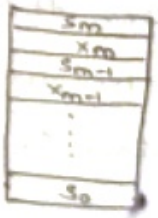
If J is

item sets

goto(J, x)

sets of ite

Model Of



Stack

Construction of CALR Parsing table Algorithm

Input:- Grammar 'G'

Output:- LALR Parsing table

Method:-

Step1:- Construct $C = \{ I_0, I_1, I_2, \ldots I_n \}$ collection of the sets of LR(1) items sets

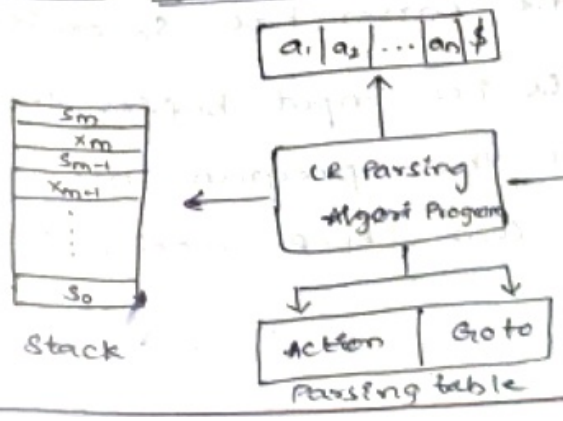
Step2:-
Find the sets with the common core and different lookaheads and replace those sets by their union

Step3:- Let $C' = \{ J_0, J_1, J_2, \ldots J_n \}$ be the resultant LR(1) item sets. The parsing table actions for state i are constructed from I_i in the same manner as in LR/CLR Parsing table

Step4:-
The Goto part is constructed as follows

If J is the union of one (or) more LR(1) item sets i.e., $J = \{ I_1 \cup I_2 \cup \ldots I_k \}$ then goto(J, x) = k where k be the union of sets of items having the same core.

Model Of an LR Parser (LR Parsing Algorithm):-



Stack:-

It consists combination of states & grammar Symbols

S_0 is the initial state present over top of the stack.

Function of LR Parser:-

Let S_m be the top of the stack and 'a' be the current input symbol then it consult action $[S_m, a_i]$ of parsing table, where they have following values

(i) Shift S, where S is a state

(ii) Reduce by $A \rightarrow \beta$

(iii) Accept

(iv) Error

LR Parsing Algorithm:-

Input:- An input string 'w' and LR parsing table

output:- If w is in L(G) the reduction steps of bottomup parser for 'w' will be produced otherwise Error

Method:- Initially the parser has S_0 on its stack and w$ is in input buffer then the parser executes the program below until an ACCEPT (or) ERROR action is encountered.

Program:-

```
Set ip to point to the first symbol of w$
Repeat
begin
    let s be the state on the top of the stack
        a is the symbol pointed by ip
  if action[s, a] = shift s then
  then begin
        push 'a' then s on to the top of the
      stack and then advance ip to the next
      input symbol
       end
  else if action[s,a] = reduce A → β then
    begin                          → since here we are
                                     shifting 2 symbols
        pop 2 * |β| of the stack
                                   so A = |β|
      let s' be the state on the top of the stack
      push 'A' then Goto[s', A] on to top of
            the stack (i.e. s) output the
        Production A → β

  else if action[s,a]= Accept
            return

    else
            Error
```

STACK IMPLEMENTATION:-

Stack	Input	Action
0	id + id $	Shift 5 -
0 id 5	+ id $	Shift 5
0 F 3	+ id $	Reduce F → id
0 T 2	+ id $	Reduce T → F
0 E 1	+ id $	Reduce E → T
0 E 1 + 6	id $	Shift 6
0 E 1 + 6 id 5	$	Shift 5
0 E 1 + 6 F 3	$	Reduce F → id
0 E 1 + 6 T 9	$	Reduce T → F
0 E 1	$	Accept Reduce E → E+T
0 E 1	$	Accept

Ambiguou

1. E → E
   E → E
   E → (E
   E → id

2. G' E'
   E' → E+
   E → E*
   E → (E'
   E → id

S_0 : E' →
E →
E →
E →

Go to I(

I_1 : E' →
E →
E →
E/

Go to (

I_2 : E
E
E

Go to (

I_3 :

Go to (

I_4 : E →
E →

Ambiguous Grammar:—

i) $E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to id$

ii) $S \to iSeS \mid iS \mid a$

---

i) G' $E' \to E$

$E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to id$

I_0 : $E' \to \cdot E$

$E \to \cdot E+E$

$E \to \cdot E*E$

$E \to \cdot(E)$

$E \to \cdot id$

Goto (I_0, E)

I_1 : $E' \to E \cdot$

$E \to E \cdot + E$

$E \to E \cdot * E$

Goto $(I_0, ($

I_2 : $E \to (\cdot E)$

$E \to \cdot E+E$

$E \to \cdot E*E$

$E \to \cdot(E)$

$E \to \cdot id$

Goto (I_0, id)

I_3 : $E \to id \cdot$

Goto $(I_1, +)$

I_4 : $E \to E+\cdot E$

$E \to \cdot E+E$

$E \to \cdot E*E$

$E \to \cdot(E)$

$E \to \cdot id$

Goto $(I_1, *)$

I_5 : $E \to E*\cdot E$

$E \to \cdot E$

$E \to \cdot E+E$

$E \to \cdot E*E$

$E \to \cdot(E)$

$E \to \cdot id$

Goto (I_2, E)

I_6 : $E \to (E \cdot)$

Goto $(I_2, $

$E \to E \cdot + E$

$E \to E \cdot * E$

Goto (I_2, C)

I_2 : $E \to (\cdot E)$

$E \to \cdot E+E$

$E \to \cdot E*E$

$E \to \cdot(E)$

$E \to \cdot id$

Goto (I_2, id)

I_3 : $E \to id \cdot$

Goto (I_4, E)

I_7 : $E \to E+E \cdot$

$E \to E \cdot + E$

$E \to E * E$

Goto (I_4, E)

$I_2: E \to (\cdot E)$

$E \to \cdot E + E$

$E \to \cdot E * E$

$E \to \cdot (E)$

$E \to \cdot id$

Goto (I_4, id)

$I_3: E \to id\cdot$

Goto (I_5, E)

$I_8: E \to E * E\cdot$

$E \to E\cdot$

$E \to E \cdot + E$

$E \to E \cdot * E$

Goto $(I_5, ()$

$I_2: E \to (\cdot E)$

$E \to \cdot E + E$

$E \to \cdot E * E$

$E \to \cdot (E)$

$E \to \cdot id$

Goto (I_5, id)

$I_3: E \to id\cdot$

Goto $(I_6,))$

$I_9: E \to (E)\cdot$

Goto $(I_6, +)$

$I_4: E \to E + \cdot E$

$E \to \cdot E + E$

$E \to \cdot E * E$

$E \to \cdot (E)$

$E \to \cdot id$

Goto $(I_6, *)$

$I_5: E \to E * \cdot E$

$E \to \cdot E + E$

$E \to \cdot E * E$

$E \to \cdot (E)$

$E \to \cdot id$

Goto $(I_7, +)$

$I_4: E \to E + \cdot E$

$E \to \cdot E + E$

$\vdots$

Goto $(I_7, *)$

$I_5: E \to E * \cdot E$

$E \to \cdot E + E$

$\vdots$

Goto $(I_8, +)$

$I_4: E \to E + \cdot E$

$\vdots$

Goto $(I_8, *)$

$I_5: E \to E * \cdot E$

Goto

Graph

	+	
0	e1	
1	S4	
2	e1	
3	R4	
4	e1	
5	e1	
6	S4	
7	S4, R1	
8	S4, R2	
9	R3	

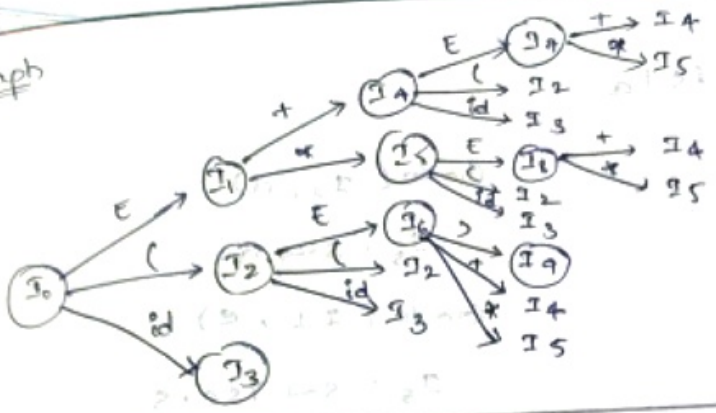
R1 1) E →

R2 2) E →

R3 3) E →

R4 4) E →

Graph



	Action						Go to
	+	*	(	)	id	$	E
0	e1	e1	S2	e2	S3	e1	1
1	S4	S5	e3	e2	e3	ACCEPT	
2	e1	e1	S2	e2	S3	e1	6
3	r4	r4	r4	r4	r4	r4	
4	e1	e1	S2	e2	S3	e1	7
5	e1	e1	S2	e2	S3	e1	8
6	S4	S5	e3	S9	e3	e4	9
7	S4, r1	S5, r1	r1	r1	r1	r1	
8	S4, r2	S5, r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

r1 1) E → E+E

r2 2) E → E*E

r3 3) E → (E)

r4 4) E → id

First(E) = { (, id }

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

- If any row contains only reduction fill the remaining empty fields with reduction only

$S \to iSeS \mid iS \mid a$

G'

$S' \to S$
$S \to iseS$
$S \to iS$
$S \to a$

I_0 : $S' \to .S$
$S \to .iseS$
$S \to .iS$
$S \to .a$

Goto (I_0, S)

I_1 : $S' \to S.$

Goto (I_0, i)

I_2 : $S \to i.SeS$
$S \to i.S$
$S \to .iseS$
$S \to .iS$
$S \to .a$

Goto (I_0, a)

I_3 : $S \to a.$

Goto (I_2, S)

I_4 : $S \to iS.eS$
$S \to iS.$

Goto (I_2, i)

I_2 : $S \to i.SeS$
$S \to i.S$
$S \to .iseS$
$S \to .iS$
$S \to .a$

Goto (I_2, a)

I_3 : $S \to a.$

Goto (I_4, e)

I_5 : $S \to iSe.S$
$S \to .iseS$
$S \to .iS$
$S \to .a$

Goto (I_5, S)

I_4 : ~~$S \to iS.eS$~~
$S \to iS.$

Goto (I_5, i)

I_5 : $S \to i.SeS$
$S \to i.S$
$S \to .iseS$
$S \to .iS$
$S \to .a$

Goto (I_5, a)

I_3 : $S \to a.$

Goto (I_5, S)

I_6 : $S \to iSeS.$

Goto (I_5, i)

I_2 : $S \to i.SeS$
$S \to i.S$
$S \to .iseS$
$S \to .iS$
$S \to .a$

Goto (I_5, a)

I_3 : $S \to a.$

Graph

I_0

	i
0	S2
1	
2	S2
3	
4	
5	S2
6	

First(S

Follow(

Stack

0

0 i 2

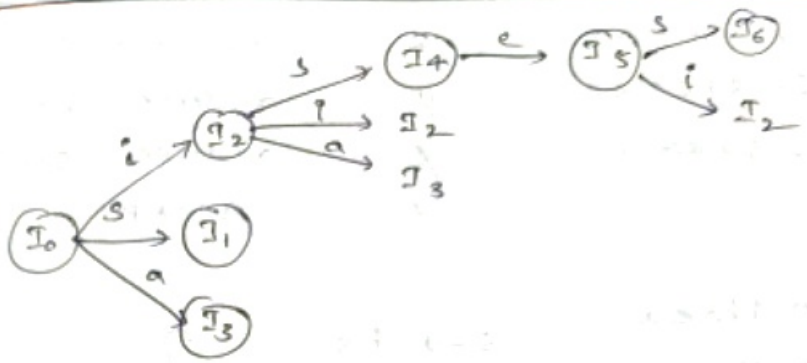
0 i 2 i 2

0 i 2 i 2

0 i 2 i 2

0 i 2 i 2

Graph



	Action				Go to
	i	e	a	$	S
0	S2		S3		1
1				ACCEPT	
2	S2		S3		4
3		r3		r3	
4		S5, r2		r2	
5	S2				6
6		r1		r1	

First(S) = { i, a }

Follow(S) = { e, $ }

r1 1) S → iSeS

r2 2) S → iS

r3 3) S → a

Stack	Input	Action
0	iiaea$	shift 2
0 i2	iaea$	shift 2
0 i2 i2	aea$	shift 3
0 i2 i2 a3	ea$	reduce S → a
0 i2 i2 S4	ea$	shift 5
0 i2 i2 S4 e5	a$	Shift 3

0 i 2 i 2 S 4 e 5 a 3	$	reduce S → a
0 i 2 i 2 S 4 e 5 S 6	$	reduce S → i S e S
0 i 2 i 2 S 4 ~~e S a~~	$	reduce S → i S
0 S 1	$	ACCEPT

stack
$
$
$
$

$S \rightarrow iiaea$
$\Rightarrow iiaea$

$S \rightarrow iS$
$\Rightarrow iises$
$\Rightarrow iisea$
$\Rightarrow iiaea$

} These are the handles (i.e., Reduce keys in Example)

stack
$ id
$ id
$ id

ERROR RECOVERY IN LR PARSING :—

1. Resolve / Completes The error recovery routines are defined as follows

e1 : This Error routine is called from 0, 2, 4, and 5 states to indicate "MISSING OPERAND"

e2 : This Error routi recovery routine is called from 0, 1, 2, 4 & 5 to indicate "unbalanced right parathesis"

e3 : This Error recovery routine is called from states 1 to 6 to indicate "missing Operator"

e4 : This is called at State 6 to indicate "Missing right paranthesis"

stack
$ C id
$ C id
$ C id

LL Parser

Top-down

LL

1. L - left
   Scanning
   L - left M

2. Begin with
   Symbol an
   obtain th

stack	input	Error
$	+	Missing operand
$	*	Missing operand
$	$	Missing operand
$	)	Unbalanced right paranthesis

stack	input	Error
$ id	id	Missing operator
$ id	(	Missing operator
$ id	)	Unbalanced right paranthesis

stack	input	Error
$ (id	id	Missing operator
$ (id	(	Missing operator
$ (id	$	Missing right paranthesis

LL Parser Vs LR Parser

(or)

Top-down Vs Bottom-up Parsing:-

LL Parser	LR Parser
1. L - Left to right scanning of input L - left Most Derivation	1. L - Left to right scanning of input R - Right most derivation in reverse
2. Begin with start symbol and try to obtain the given string	2. Begin with given string and try to obtain the start symbol.

3. Can have sequence of expansions (i.e., derivations)	3. Sequence of reductions
4. The two actions we can perform here are i. Predict - choose the production ii. Match - Match the correct i/p symbol	4. The two actions performed are shift & reduce. shift :- Shift the i/p symbol reduce :- reduce the handle by i/p symbol.
5. Less powerful parsers	5. More powerful parsers
6. LL parsers accept small class of grammars than LR parsers.	6. LR Parser accept large class of grammar than LL Parsers.
7. Simple construction than LR.	7. Complex construction than LL.
8. No tools are available for LL parsers	8. Tools are available for LR parser i.e., YACC tool (Yet another compilex compilex)

Classificatio

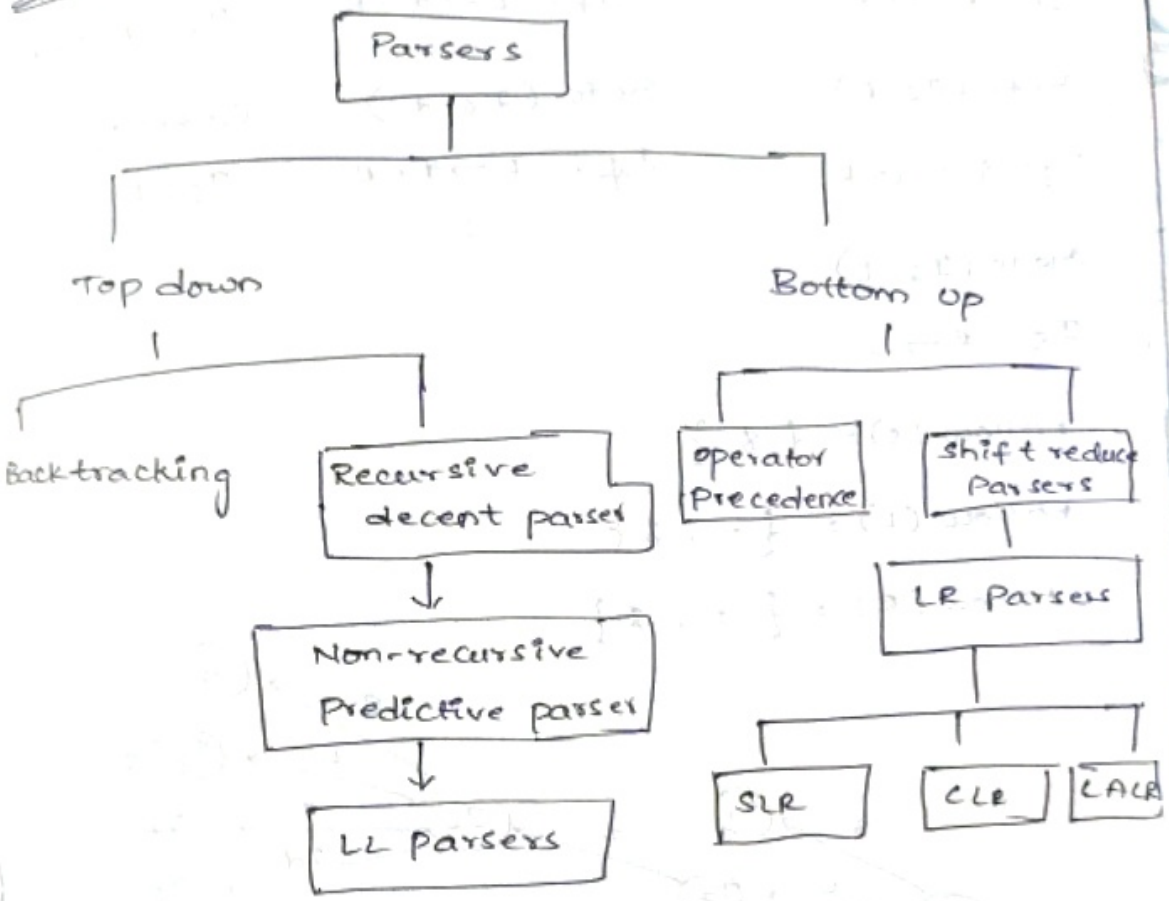
Top down

Backtracking

Ex:- S → L
L → *
R → L

Goto (I0, S)
I1 : S → S.
Goto (I0, L)
I2 : S → L. =
R → L.
Goto (I0, R)
I3 : S → R.
Goto (I0, *
I4 : L → *
R → .
L → .

Classification Of parsers:—



Ex:- $S \rightarrow L=R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

I_0 : $S' \rightarrow .S$
$S \rightarrow .L=R$
$S \rightarrow .R$
$L \rightarrow .*R$
$L \rightarrow .id$
$R \rightarrow .L$

Goto (I_0, S)

I_1 : $S' \rightarrow S.$

Goto (I_0, L)

I_2 : $S \rightarrow L.=R$
$R \rightarrow L.$

Goto (I_0, R)

I_3 : $S \rightarrow R.$

Goto $(I_0, *)$

I_4 : $L \rightarrow *.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

Goto (I_0, id)

I_5 : $L \rightarrow id.$

Goto $(I_2, =)$

I_6 : $S \rightarrow L=.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

Goto (I_4, R)

I_7 : $L \rightarrow *R.$

Goto (I4, L)

$I_8 : R \to L.$

Goto (I6, R)

$I_9 : S \to L = R.$

Goto (I6, L)

$I_8 : R \to L.$

Goto (I4, *)

$I_4 : L \to * . R$

Goto (I6, *)

$I_4 : L \to * . R$

Goto (I4, id)

$I_5 : L \to id.$

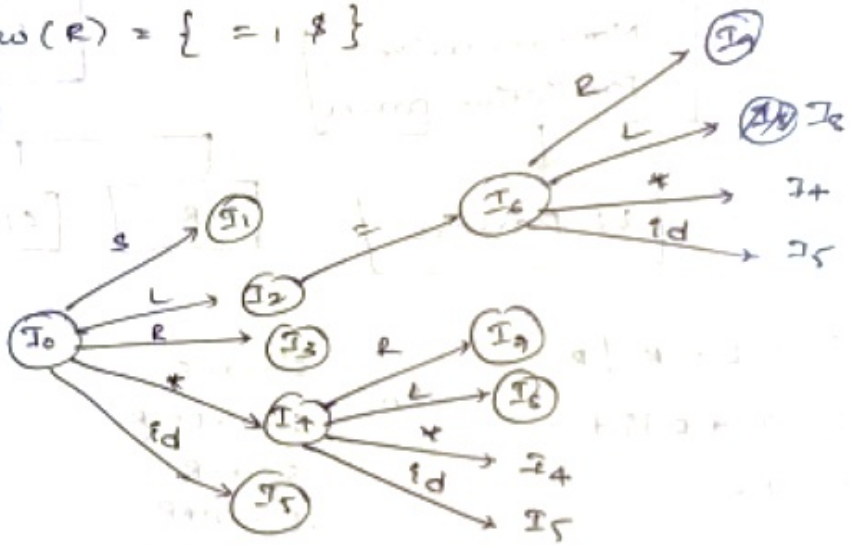
Goto (I6, id)

$I_5 : L \to id.$

Follow (S) = { $ }

Follow (L) = { =, $ }

Follow (R) = { =, $ }



19/8/2016

UNIT-IV

- Syn' Semantic Analysis
- Intermediate code Generation

Syntax directed translation schemes:-

syntax directed definition :- (SDD)

CFG + Semantic rules

- Each and every grammar symbol is associated with ~~a~~ a set of attributes and each and every production is associated with set of semantic rules
- semantic rules are used to compute attributes of the grammar symbols (or) to compute a attributes at ~~the~~ particular Node.

Ex:- Node x is Non terminal, whose attributes are by the semantic rule associated with that node.

- There are two types of attributes
  i) Synthesized attributes
  ii) Inherited attributes

Synthesized attributes:-

- value of a synthesized attribute at a node is computed from the values of attribute at the childrens of that node ~~in~~ in the parse tree. value of a parent/ left hand side is computed ~~from~~ from the attributes values of its children/ right hand side production

Inherited attributes:-

The value of an inherited attribute is computed from the values of the attributes at the siblings and parent of that node.

NOTE:-

- All terminals have only Synthesized attributes
- Root node doesn't have inherited attribute

Ex:-

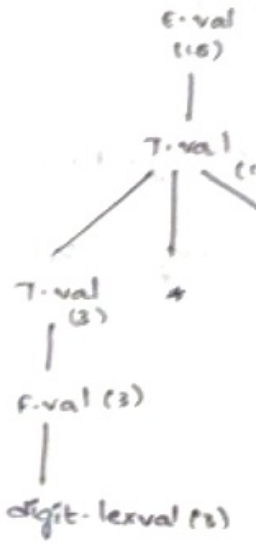
Productions	Semantic rules
$L \rightarrow E_n$	Print (E.val)
$E \rightarrow E+T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T*F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = (E.val)
$F \rightarrow$ digit	F.val = digit.lexvalue

lex value - value given in the program

$L \rightarrow E_n$ - Prints the final value

- Here in the above example we are using Synthesized attributes and the synthesized attribute is 'val'.

Ex:- 3 * 5 +



Annotated Parse

attribute values at each

parse tree.

SDD for Inh

Declaration g

$D \rightarrow TL;$

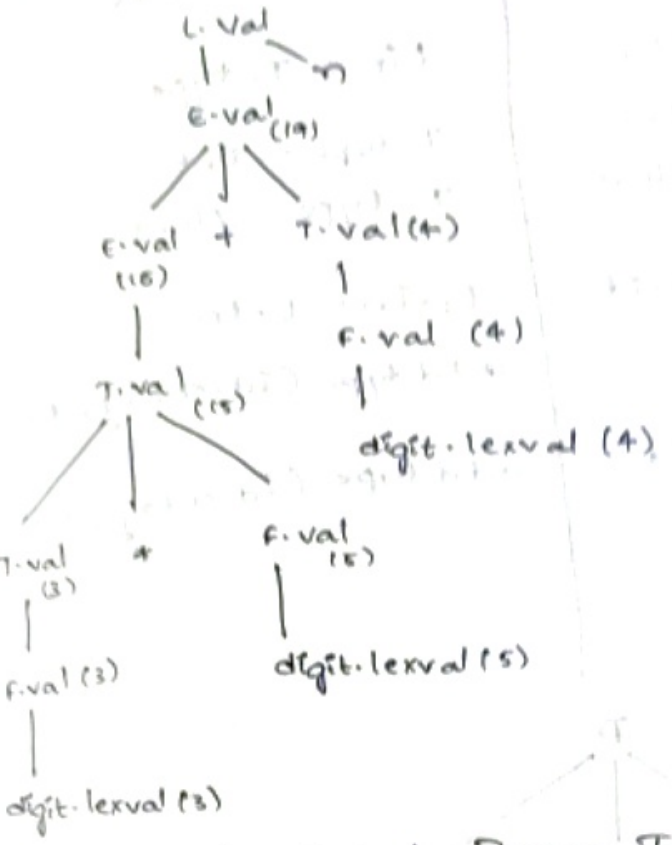
$T \rightarrow$ int/flo

$L \rightarrow L, id$

$L \rightarrow id$

Ex:- int id,

Ex: 3 * 5 + 4 n



Synthesized attribute

Example

Annotated Parse Tree

Annotated Parse tree:— Parse tree is showing attribute values at each node is called Annotated parse tree.

SDD for inherited attributes:—

→ for inherited we consider top down procedure

for synthesized we follow bottom-up procedure

Declaration grammar

$D \rightarrow TL;$

$T \rightarrow int/float$

$L \rightarrow L, id$

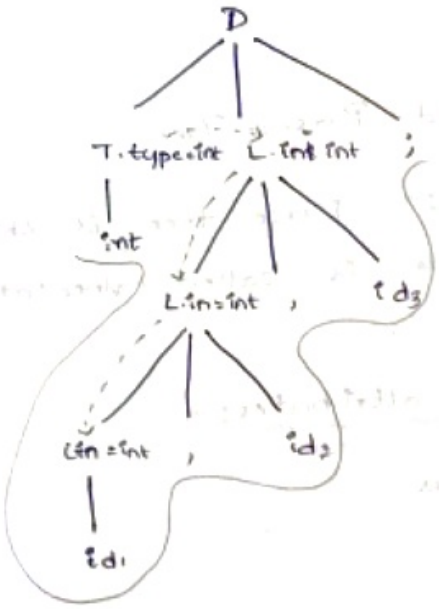
$L \rightarrow id$

Ex:- int id, id, id;

Production	Semantic rules
$D \rightarrow TL;$	$L.in = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L_1, id$	$L_1.in = L.in$ add type (id.entry, L.in)
$L \rightarrow id$	add type (id.entry, L.in)

entry attributes which represents symbol table

- type which is given for id should be represented for list



SDD for non left recursive version of terms 3*5, 6*4 ... :-

not

- Here we are considering expression terms in order to get simplicity.

Synthesized are val, syn

Inherited are inh

Production

1. $T' \rightarrow$ *F

2. $T' \rightarrow$ *F

3. $T' \rightarrow \epsilon$

4. $F \rightarrow$ digit

Ex:- 3*5

F.val

digit.lexval

Evaluation

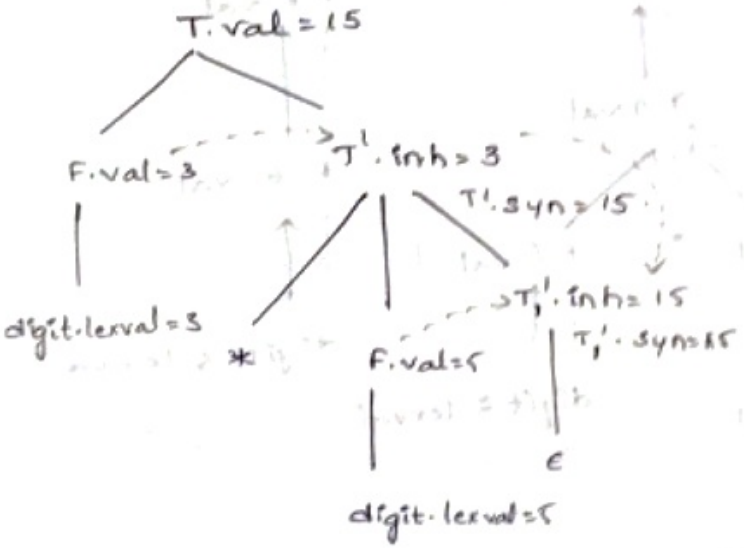
- It sho attribute
- In wha

Dependere

- In gener dependen
- It sho attribute

Production	Semantic Rules
1. $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2. $T' \rightarrow * F T_1'$	$T_1'.inh = F.val * T_0'.inh$ $T'.syn = T_1'.syn$
3. $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4. $F \rightarrow digit$	$F.val = digit.lexval$

Ex: 3*5



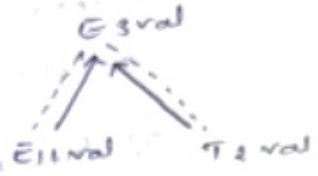
Evaluation Order Of SDD's :-

- It shows the evaluation of order of attributes in SDD.
- In what order the attributes are computed.

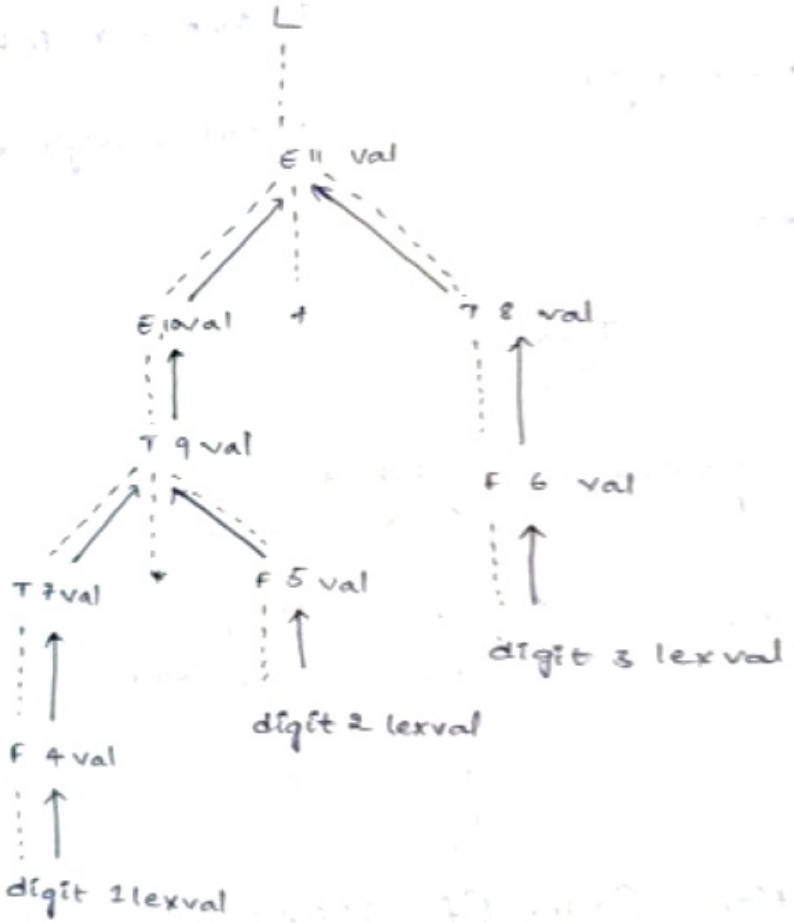
Dependency Graph :-

- In general these are useful to express dependencies in programming language constructs
- It shows flow of information from one attribute to another attribute in the parse tree

Ex:- $E \rightarrow E_1 + T$ $\quad E.val = E_1.val + T.val$

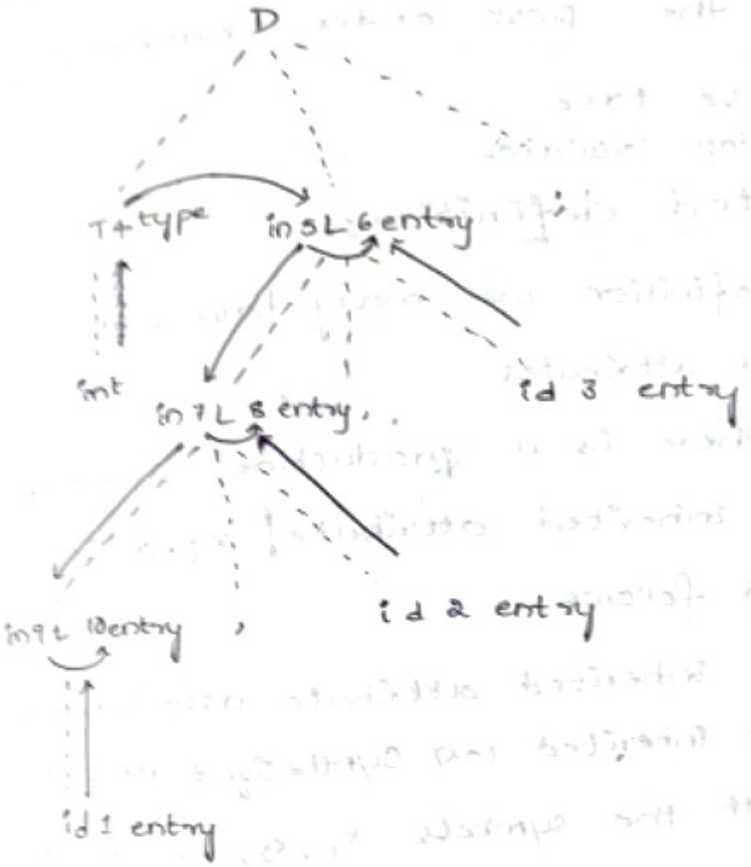


Ex:- 3 * 5 + 4

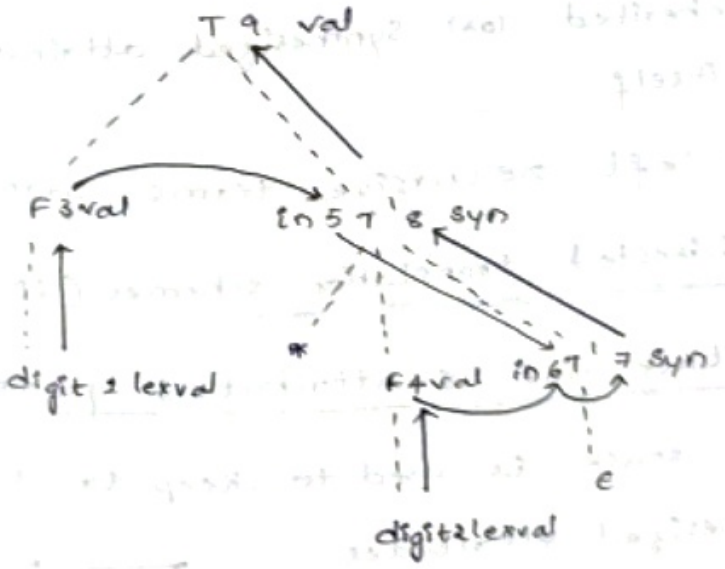


- lexval has more precedence than val
- first we have to copy the values to their at respective productions (i.e., E / T / F) i.e, give first precedence to them then we have to give order for computations (precedence)

Dependency graph for declaration grammar.



3 * 5



S-attributed definition:-

An SDD is said to be an S-attributed if every attribute is synthesized. We can evaluate its attributes in the bottom up manner.

- It is simple to perform attribute evaluation by the post order traversal of the parse tree.

Ex:- Desktop calculator.

L-attributed definition:-

In this definition we may have both Synthesized and inherited attributes.

Ex:- suppose there is a production $A \rightarrow X_1 X_2 \cdots X_n$ where the inherited attribute $X_i.a$ is computed as follows

Rule 1:- The inherited attribute associated with A

Rule 2:- either inherited (or) Synthesized attributes associated with the symbols $X_1, X_2, \cdots X_{i-1}$ located to the left of X_i.

Rule 3:- Inherited (or) Synthesized attribute of X_i itself.

Ex:- Non-left recursive terms grammar

Syntax Directed translation schemes (SDT schemes)

Parser stack for S-attributed grammar:-

Parser stack is used to keep track of the values of synthesized attributes

$X \rightarrow ABC$

attribute a

$X.a = A.a\ B.a\ C.a$

State	value
C	C.a
B	B.a
A	A.a

X	X.a

Parser stack.

- Here the @ e
fix evaluation o
stored in x C

Parser stack ac

Production

$L \rightarrow En$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$ {

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow digit$

Ex:- Parser sta

2 * 3n :-

Input string

2 * 3n
* 3n
* 3n
* 3n
3n
n
n
n
n

attribute evalua-

traversal of

have both Synthesized

tion $A \rightarrow X_1 X_2 \cdots X_n$

e $X_i.a$ is

te associated with A

thesized attributes

$, X_2, \cdots X_{i-1}$ located

d attribute of X_i

rms grammar

hemes (SDT schemes)

ted grammar :-

keep track the values

State	value
C	C.a
B	B.a
A	A.a

X	X.a

Parser stack.

- Here the @ evaluation or is similar to postfix Evaluation and the final result will be stored in x (i.e., in parser stack)

Parser stack actions for desktop calculations:-

Production	Action
$L \rightarrow En$	{ Print (stack[top-1].val) top = top - 1 }
$E \rightarrow E_1 + T$	{ stack[top-2].val = stack[top-2].val + stack[top].val top = top - 2 }
$E \rightarrow T$	top = top - 2 }
$T \rightarrow T_1 * F$	{ stack[top-2].val = stack[top-2].val * stack[top].val
$T \rightarrow F$	top = top - 2 }
$F \rightarrow (E)$	{ stack[top-2].val = stack[top-1].val
$F \rightarrow$ digit	top = top - 2 }

Ex:- Parser stack implementation for the string 2 * 3n :-

Input string	State	value	Production
2 * 3n	-	-	-
* 3n	2	2	-
* 3n	F	2	F → digit
* 3n	T	2	T → F
3n	T *	2 *	-
n	T * 3	2 * 3	-
n	T * F	2 * 3	F → digit
n	T	6	T → T * F
n	E	6	E → T

INTERMEDIATE CODE GENERATION:-

- Why we need intermediate code generation?

Intermediate code is required so that code generation process will become simple

- Two types of intermediate code
  (i) Three address code (Most Popular)
  (ii) Abstract syntax tree

Three adress Code :-

At most three addresses (operands)

Addresses and instruction:-

Addresses:-
(i) Names appear in source program
(ii) constants
(iii) compiler generated temporaries

$t_1 = x * 2$
↳ temporaries

Instructions:-

Types of three address instruction :-

1. Assignment statement of the form (binary operator)

$x = y \text{ op } z$

(ii) Assignment statement with urary operator.

$x = \text{minus } y$

(iii) Copy statement of the form

$x = y$

(iv) Unconditional goto statement

goto L1

nditional goto with x (single Operand)

if x then goto L1

vii) conditional goto with relational Operator

if x relop y then go to L1

ix) Procedure calls

```
Param   x1
Param   x2
Param   x3
  :
Param   xn
```

- First we have to initialize the Parameters then we have to call the procedure

call P, n

where P = name of the procedure

n = No. of parameters passed

viii) indexed copy statement

x = a[i]

a[i] = x

ix) Pointer assignments

x = &a

&a = x

Ex:-

do i = i + 1 while (a[i] < v);

```
L1 : t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a[t2]
     If a t3 < v then goto L1
```

with table

Since array is consecutive and for each element we will give 8 units of memory so the index is multiplied by 8.

using positions

100 : $t_1 = i + 1$

101 : $i = t_1$

102 : $t_2 = i * 8$

103 : $t_3 = a[t_2]$

104 : if $t_3 < v$ then goto 100

How to represent three address code in data structure :-

There are 3 such representations, as a record with fields for op, arg1, arg2, result

i. Quadraple - op, arg1, arg2, result (4 fields)

ii. Triple - op, arg1, arg2 (3 fields)

iii. Indirect Triple :- op, arg1, arg2 (3 fields)

Ex :- $a = b * -c + b * -c$ (unary minus)

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

unary minus has more precedence than *

Based on precedence they are Evaluated

i. Quadraple :-

For the above Example

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Exceptions :-

i. For copy

arg2 is

ii. operators

are ab

ii. Triple :-

- Results

than by

- Here w

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

- wastage

inde

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

→ Memory Wastage

Exceptions:-

i. For copy and unary assignment statements arg2 is absent

ii. operators like param both arg2 and result are absent

fields)

iii. Triple :- Fields op, arg1, arg2

ds)

- Results can be accessed by its position than by Explicit temporary names
- Here we reduce memory wastage

location

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

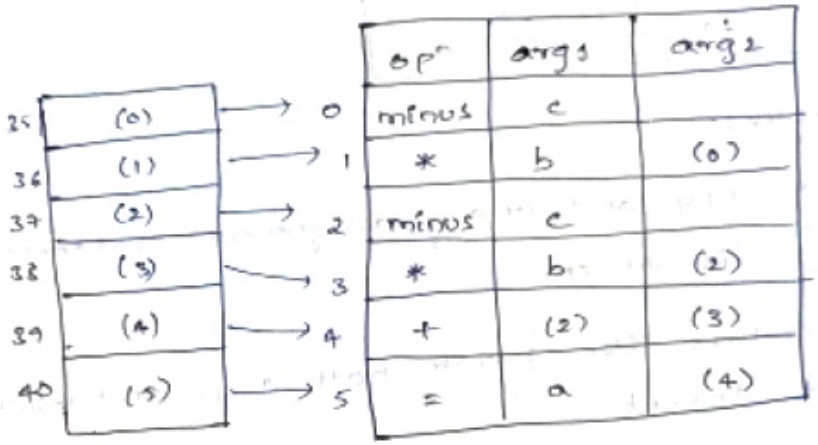
0th location (-c)

(0) t1 = minus c
(1) t2 = b * t1
(2) t3 = minus c
(3) t4 = b * t3
(4) t5 = t2 + t4
a = t1

- wastage in triple is less than Quadraple,

Indirect Triple :-

- In triple when a code is moved to different memory locations we have to change pointer references.
- To overcome that problem we moved to indirect triple notation



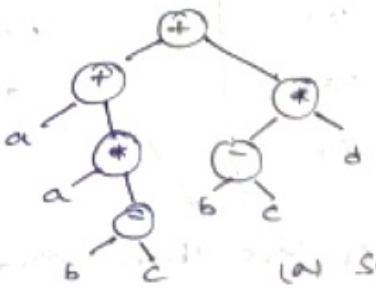
			op	arg1	arg2
35	(0)	0	minus	c	
36	(1)	1	*	b	(0)
37	(2)	2	minus	c	
38	(3)	3	*	b	(2)
39	(4)	4	+	(2)	(3)
40	(5)	5	=	a	(4)

Abstract Syntax tree (Direct Acyclic Graph) :-

(OR) condensed Syntax tree :-

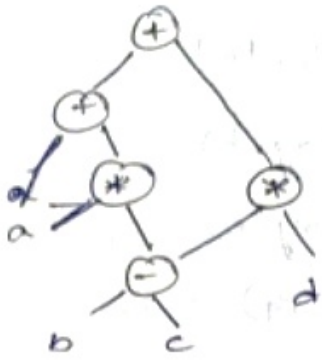
- In direct acyclic graph, a node that represents common sub expression has more than one parent.
- In Syntax tree, subtree that represents common sub expression replicates as many times as it appears in original expression

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



→ 13 nodes

(a) Syntax tree



→ 9

(b) DAG (Directed Acyclic Graph)

SDT scheme for construction of syntax tree (or) DAG:-

Production	Action
$E \rightarrow E_1 + T$	E.node = newnode('+', E_1.node, T.node)
$E \rightarrow E_1 * T$	E.node = newnode('*', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = newleaf(id, id_entry)
$T \rightarrow num$	T.node = newleaf(num, num.val)

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

P_1 = leaf (id, entry-a)

P_2 = leaf (id, entry-a) = P_1

P_3 = leaf (id, entry-b)

P_4 = leaf (id, entry-c)

P_5 = node (leaf '-', P_3, P_4)

P_6 = node ('*', P_2, P_5)

P_3 = node('+', P_1, P_2)

P_5 = leaf(id, entry-b) = P_5

P_6 = leaf(id, entry-c) = P_6

P_{10} = node('-', P_8, P_9) = P_{10}

} Where 4 nodes are constructed

P_{11} = leaf(id, entry-d)

P_{12} = node('*', P_{10}, P_{11})

P_{13} = node('+', P_3, P_{12})

Types and declarations:-

Type Expressions:- (Types can be specified)

- Basic Type is TE
  Ex:- int, char, float, double
- Name given to basic type is a TE
- constructor applied to basic type is a TE

  int [2][3] a;

  i.e., array(2(array(3, integer))

- Constructor record applied to various fields of that record is a TE

  Ex:- record { int x; int y; }

- If S and t are two type expressions function constructor $S \rightarrow t$ applied over S and t is a T.E
- If s and t are two type expression their cartesian product $S \times t$ is a TE

SDT for computing types and widths/storage layout for local variables:-

Production

Declaration:-

$D \rightarrow T\ id;\ D/\epsilon$

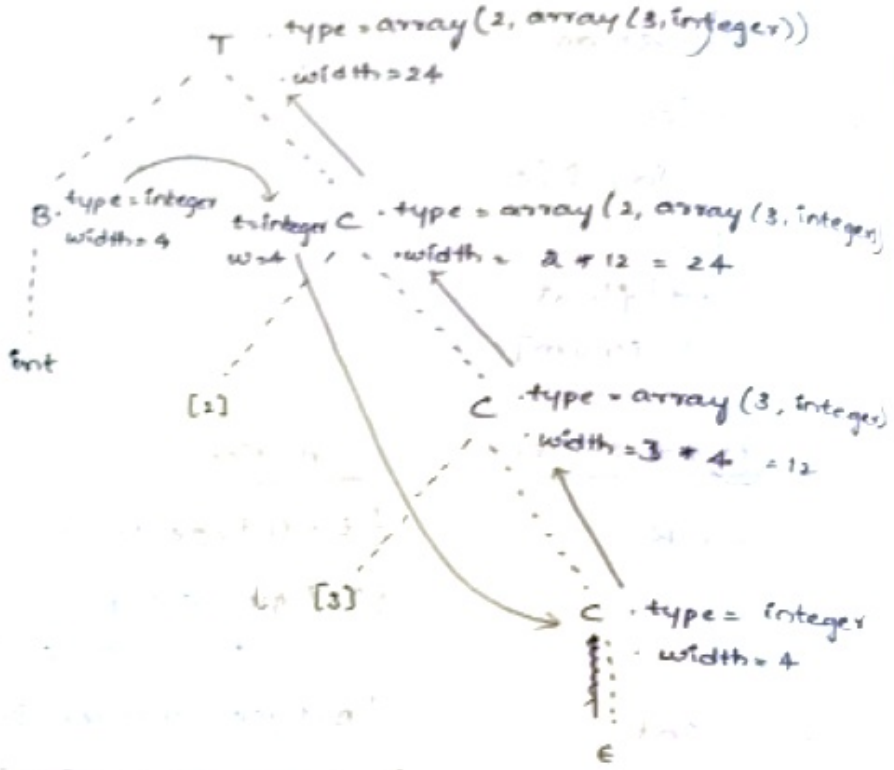
$T \rightarrow BC$

$B \rightarrow int | float$

$C \rightarrow \epsilon |\ [num]C$

Production	Action
$T \rightarrow B$	{t = B.type, w = B.width;}
C	{T.type = C.type, T.width = C.width;}
$B \rightarrow$ int	{B.type = integer; B.width = 4;}
float	{B.type = float; B.width = 8;}
$C \rightarrow \epsilon$	{C.type = t, C.width = w}
$\rightarrow [num]\ C_1$	{C.type = array(num.val, C.type), C.width = num.val * C_1.width;}

Ex:- int [2][3]



TYPE CHECKING :-

- To perform any operation we must check whether we have compatible types (or) not
- Every source program language is given type expression

$$c = a + b$$

- Source program - type system - logical rules

Rules for type checking :-

There are 2 rules

1. Synthesis :- A type of an expression is derived from the types of its subexpressions.

$$E \longrightarrow E_1 + E_2$$

Inference :- Type of an expression is derived from the way it is used

- Typical rule for synthesis :-

If f has a type $s \to t$ and x has type s then f(x) has type t.

- Typical rule for inference :-

If f has a type $\alpha \to \beta$ from some α and β then x has type α.

TYPE CONVERSION :-

$$x + i$$

where x is float type
i is an integer
i is converted to float

Ex:- 2 * 3.14

$t_1 = (\text{float}) 2$

$t_2 = t_1 * 3.14$

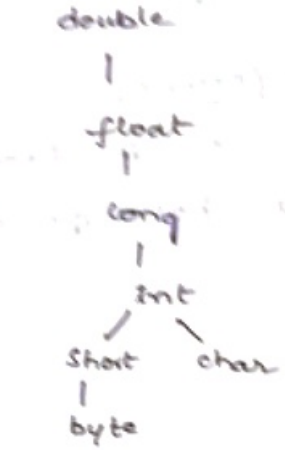
Implicit Conversion :- Compiler does the conversion. This is also called as coersion.

Explicit Conversion :- User explicitly convert It is also called as type casting.

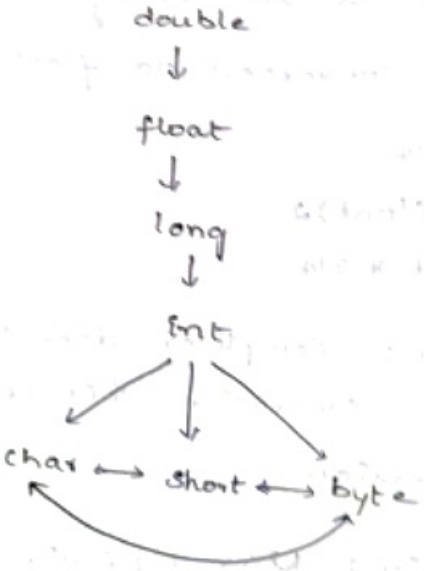
- In JAVA type languages they are called as widening conversions, Narrowing conversion.

Widening conversion :- Moving to higher type which preserves the information

Narrowing conversion :- Moving to lower types which looses the information



(a) Widening Hetrarchy



(b) Narrowing Hetrarchy

- char, byte, short are pairwise convertable

If we have
2 functions
1. max(t1, t2)
2. widen(a, t

where a :
t :
w :

Addr widen(
begin
if (t = w)
retu
else if (t
temp :
gen(
re
end

Production

$E \longrightarrow E_1 + E_2$

$E \rightarrow E_1 + E_2$

If we have to convert, then we have to follow 2 functions:

1. max (t_1, t_2) - which returns maximum type
2. widen (a, t, w)

where a = actual type identifier
t = type of actual type
w = widening

```
Addr widen( Addr a, type t, type w)
begin
    if (t = w)
        return a;
    else if (t = integer and w = float)
        temp= new temp();
        gen( temp = (float) a)
        return temp;
end
```

Production	Semantic Action
$E \rightarrow E_1 + E_2$	{ $E.type = max(E_1.type^{int}, E_2.type^{float})$ $a_1 = widen(E_1.addr, E_1.type^{int}, E.type^{float})$ $a_2 = widen(E_2.addr, E_2.type^{float}, E.type^{float})$ $E.addr = new\ temp()$ $gen(E.addr = a_1 + a_2)$ }

19/2016

UNIT-V

- Symbol Tables
- Routine storage Organization
- code Generation.

SYMBOL TABLES:—

It keeps track of name and scope information of various symbols encountered in source program.

Need Of Symbol table:—

- Every phase is interacting with symbol table
- lexical analysis enters names and their information into symbol table
- Syntax analysis access symbol table names to check syntaxes.
- Semantic analysis access symbol table to resolve type conflict issues.
- Code generation access symbol table to know how much memory is allocated to names.

Name Representation Of Symbol Table:—

- Names are variable names, constants procedure (or) function names, literal strings.
- Attributes are scope, type, offset in storage, if it is procedure name no. of parameters passed, types of parameters, whether they passed by values (or) references etc.

Names	Attributes

Name representation of Symbol Table:

- Names are represented in two ways

1. Fixed length names :- Fixed amount of storage is given irrespective of their size

Ex:-

Names									Attributes
c	a	l	c	u	l	a	t	e	
a									
b									
s	u	m							

Drawbacks:-

- It is simple to organize / implement but its drawback is wastage of memory so we are moving to variable length representation.

2. Variable length Names:— Names are given space as much as they require

Ex:-

Names		Attribute
Starting index	length	
0	9	
9	1	
10	1	
11	3	

mbol Table:-
wo ways
amount of
of their size

Attributes

plement
: of
to variable
given
re

Implementation Of Symbol table :—

Symbol tables can be implemented through 4 data structures.

i. Arrays
ii. Linked list
iii. Hashing
iv. Trees

Arrays :—

Names are stored at consecutive location.

Name	Attribute

Advantages :—

Simple to organise.

Disadvantages :—

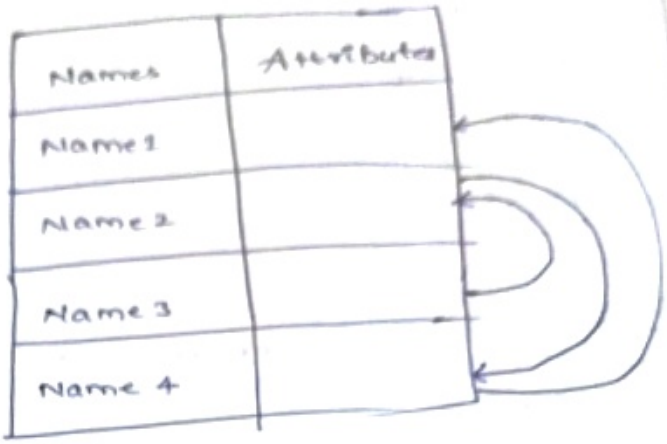
i. Search time is more if it is located at last part of an array.

ii. If we want to delete name in first part of an array i.e., name 1, we have to rearrange all other names (i.e., upside one position)

Linked Lists :—

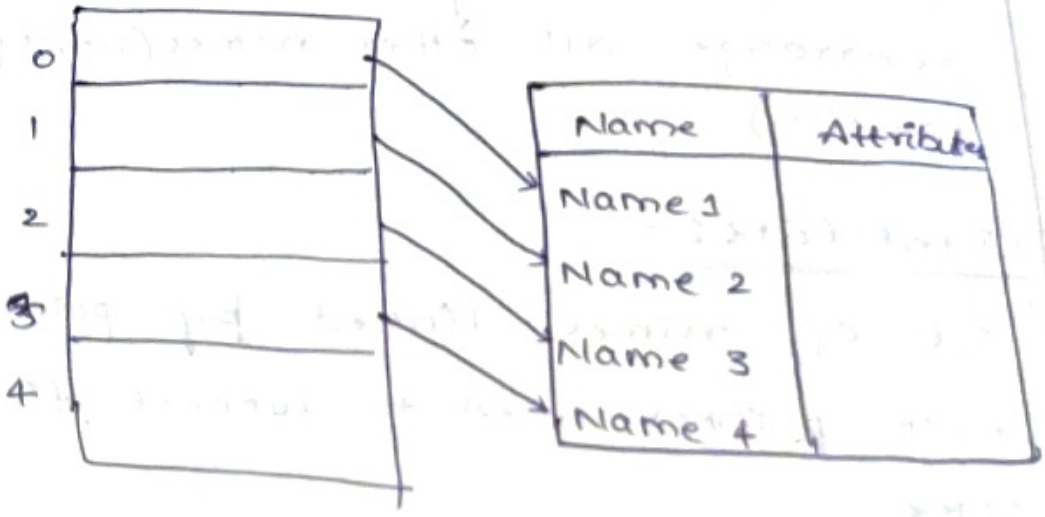
- set of names linked by pointers.
- first pointer used to connect first name.
- Name3, Name2, Name4, Name1 is the Order

- Search names by traversing across links
- In deletion simply we can change pointer connection



Hashing :-

- Quick search possible with hashing
- All the names are applied hash function while searching
- Hash function provides value, which gives the pointer to symbol table entry for that hash table name.



Runtime En

Runtime sto

Code area :-
compilation
program
in code

static :-
as static
Here who
can be
data are
Ex:- Gl

stack :-
Ex:-
- If a
- also

Heap :-
of
- If
Ex

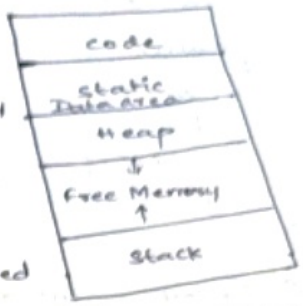
Runtime Environment :-

Runtime Storage Organisations :-

Code area :- At the time of compilation the size of the program is fixed and stored in code area.

static :- It is also called as static data area Here whose address are fixed can be allocated in static data area

Ex :- Global constants.



Runtime Memory

stack :- Runtime information of procedure calls

Ex :- Activation Record

- If a program terminates then the information related to the procedure will be eliminated

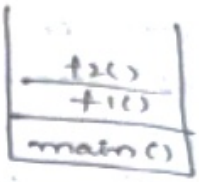
Heap :- Information outlive after the termination of procedure.

- If we want to know the result of every procedure calls if it is at the end of the procedure i.e; it will store the information then we will use heap memory
- Heap and stack both are used to store runtime information
- Heap memory will go down and stack will grow upwards (i.e.) free memory)

Stack allocation of Space:

- If the procedures are nested then the stack data area is effectively used

```
main()
{
    f1()
    {
        f2(){ .. }
    }
}
```



Ex:- Quicksort

```
int a[]
void readarray()
int partition(int m, int n)
                    ith location is pivot.
void quicksort(int m, int n)
{
  int i;
  if (n > m)
  {
    i = partition(m, n)
    quicksort(m, i-1);
    quicksort(i+1, n);
  }
}
Void main()
{
  readarray();
  quicksort(1, 9);
}
```

Recursive Calls:-

Enter main()

Enter readarray()

leave readarray()

Enter quicksort(1,9)

Enter partition(1,9)

leave partition(1,9)

then the
ly used

f2()
f1()
main()

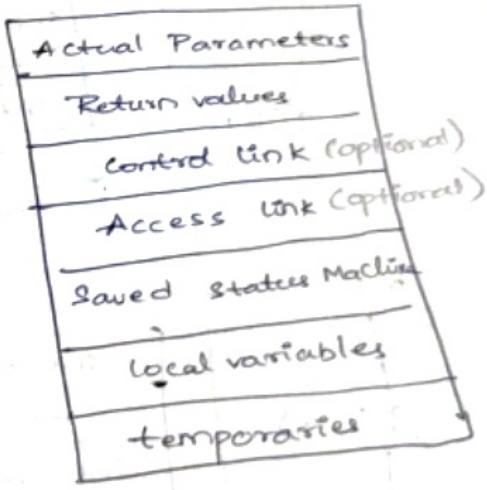
tion (int m, int
location is
pivot.

Enter quicksort(1,3)
:
leave quicksort(1,3)
Enter quicksort(5,9)
:
leave quicksort(5,9)
leave quicksort(1,9)
leave main()

Activation Record :—

Activation record is a block of memory used to manage information about the execution of a single procedure.

Block diagram Of AR :—



Actual Parameters :—

Actual parameters passed to called procedures.

Return value :— value returned by called procedure.

control link :— It is used to link to call

Access link :— used to access non local data of other activation record.

Saved machine status :— We have to save machine status before we jump to called procedure.

Local variables: — Memory allocation to local variables called procedures.

Temporaries: — Used to store intermediate results of called procedure while evaluating expressions

Position in Activation Tree	Activation Record in stack	Remarks
m	int : a[] m	Frame for main
m r	int : a[] r int : ?	r is activated
m r q(1,9)	int : a[] m int m,n q(1,9) int : ?	Frame for r has been popped out and frame for q(1,9) is activated
m r q(1,9) p(1,9) q(1,3) p(1,3) q(1,0)	int : a[] m int : m,n q(1,9) int : ? int : m,n q(1,3) int : ?	control has just returned to q(1,3)

Calling Sequence

call sequence :-
called proced
various field

Return Sequence
caller so th
part of Exec
- called ac
caller a

- In

cation to local

rmediate
ıile evaluating

Remarks

me for
main

e activated

e for x has
popped out
rame for
e is activated

has just

ned to

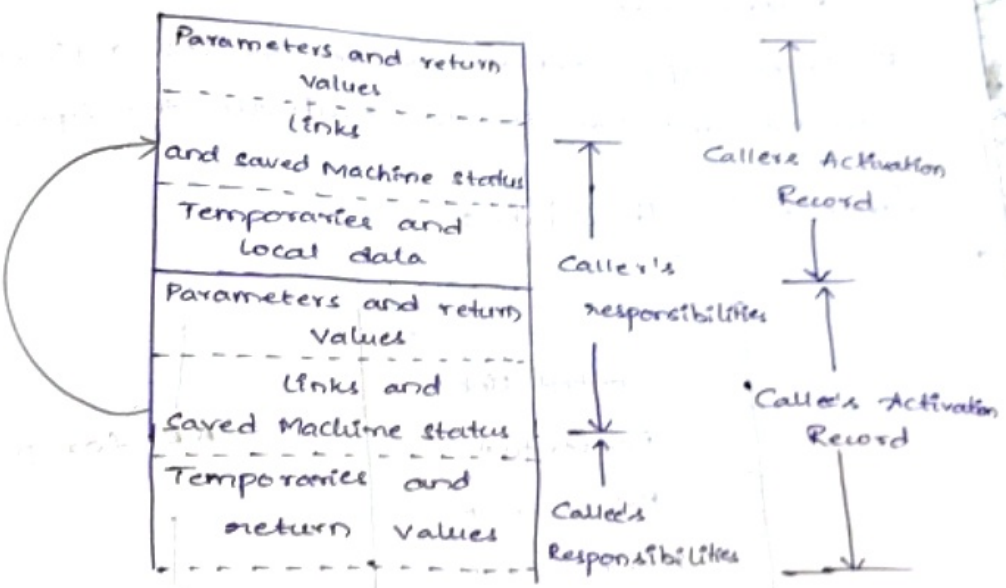
(1, 3)

Calling Sequence :—

Call Sequence :— Allocates activation record to called procedure and enters information to various fields of that record.

Return Sequence :— Reset Restores the status of caller so that caller can continue remaining part of Execution

called activation is just below that of Caller activation record



- In call sequence the following parameters are performed

i) Caller Evaluates the actuals ii) Caller stores return address, of old value of top SP (i.e., Stack pointer) to the called Activation record

iii) Callee saves register values and other status information

(iii) Callee initialise its local data and begins its execution

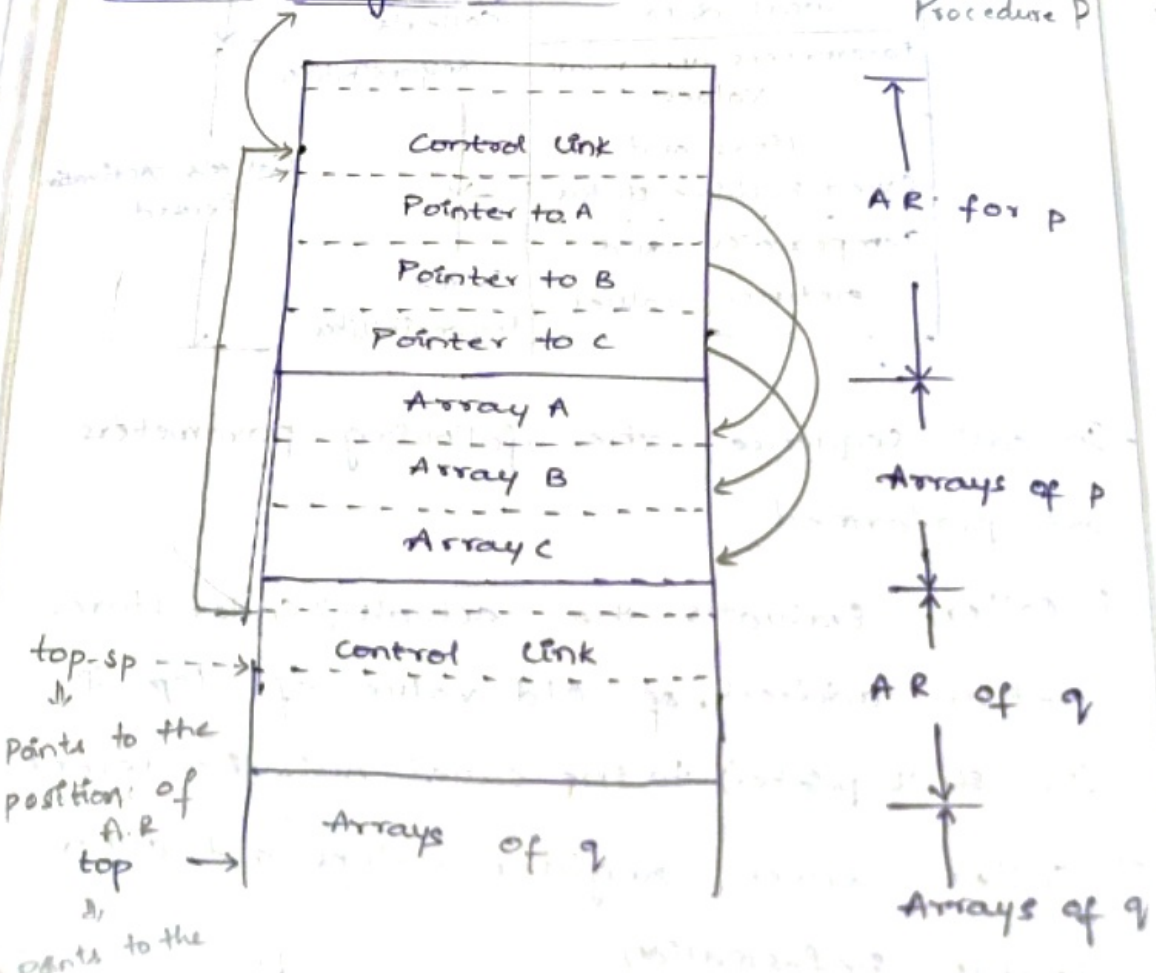
- In return sequence the following parameters are performed.

1. Callee places a return value.

(ii) Using information in status field, callee restores top SP and other registers and branch to the return address in the caller's code.

(iii) Caller copy return value to its activation record and use it to evaluate the expression

Variable length data:—



Access to

scope

1. Static/ lexical

2. Dynamic

Nesting dep

Rules to

1. Add 1 new

2. Subtrac Exit fr

3. The v -ted

- The ret caller

Ex:- --p

depth=3

Access to non local data :— (Block Structure lang)

scope

1. static/ lexical scope
   - Access link
   - Displays
2. Dynamic
   - Shallow Access
   - Deep Access

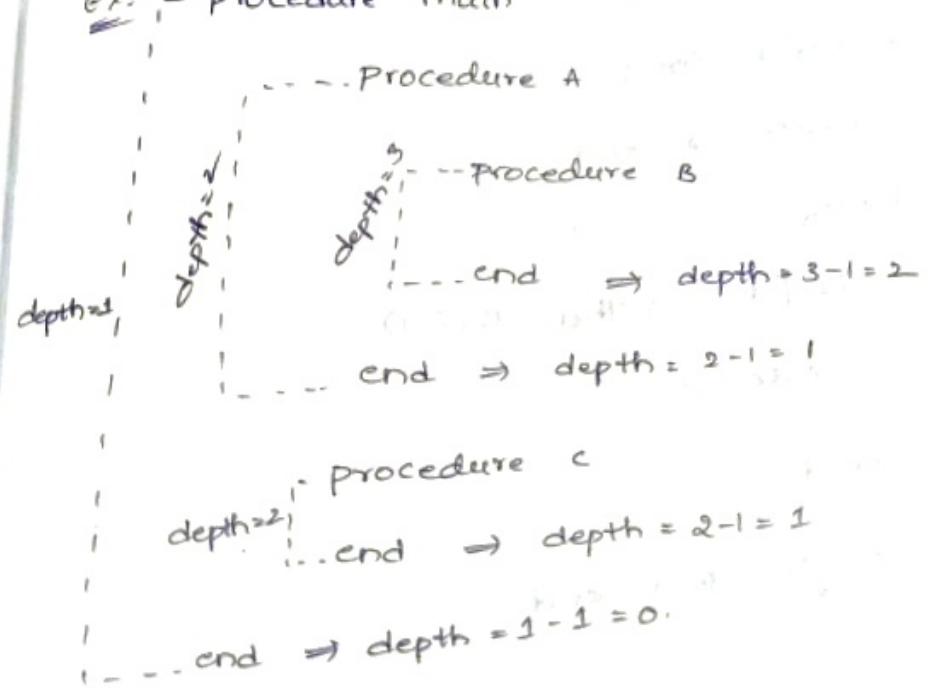
Nesting depth:— Access links are implemented by nesting depth.

Rules to compute nesting depth

1. Add 1 to the nesting depth when a new procedure begins.
2. Subtract 1 from nesting depth when you exit from a nested procedure.
3. The variable declared in a procedure associated with corresponding nesting depth.

- The nesting depth value will be 1 between caller and callee.

Ex:—



Access link :-

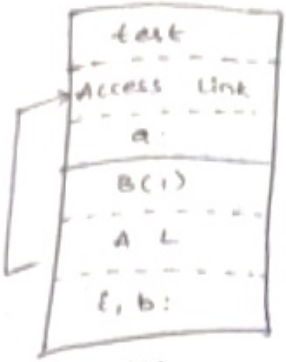
- Lexical scope can be implemented by maintaining pointers to each AR these pointers are called as "access links"
- If procedure P is nested within procedure Q then access link of P points to the access link of Q

Ex:- Program in PASCAL language.

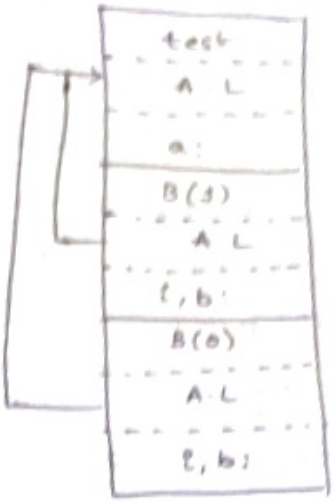
```
program test;
  var a: int
 Procedure A;
   var d: int
 begin A := 1  end
  Procedure B (i: int);
   var b: int
    Procedure C
     var k: int
    begin  A; end
    begin
      if (i<>0) then  B(i-1)
       else  C
  end
begin  B(1) end
```

maintaining
are called

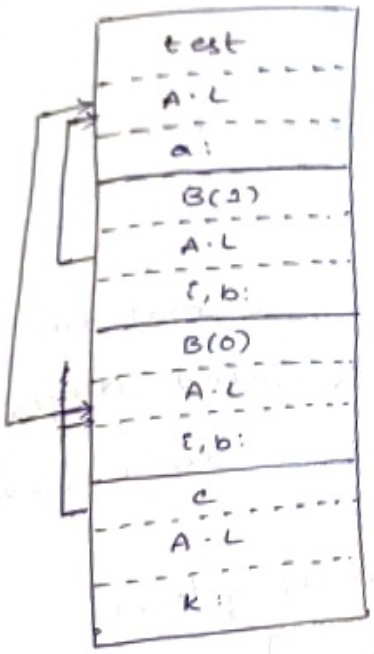
procedure
the access



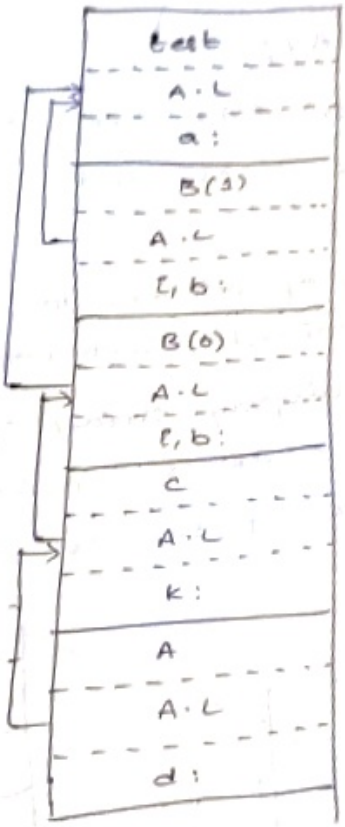
(a)



(b)



(c)



(d)

Rules to be used to set up Access links:-

1. If procedure 'A' at depth n_A calls procedure 'B' at depth n_B

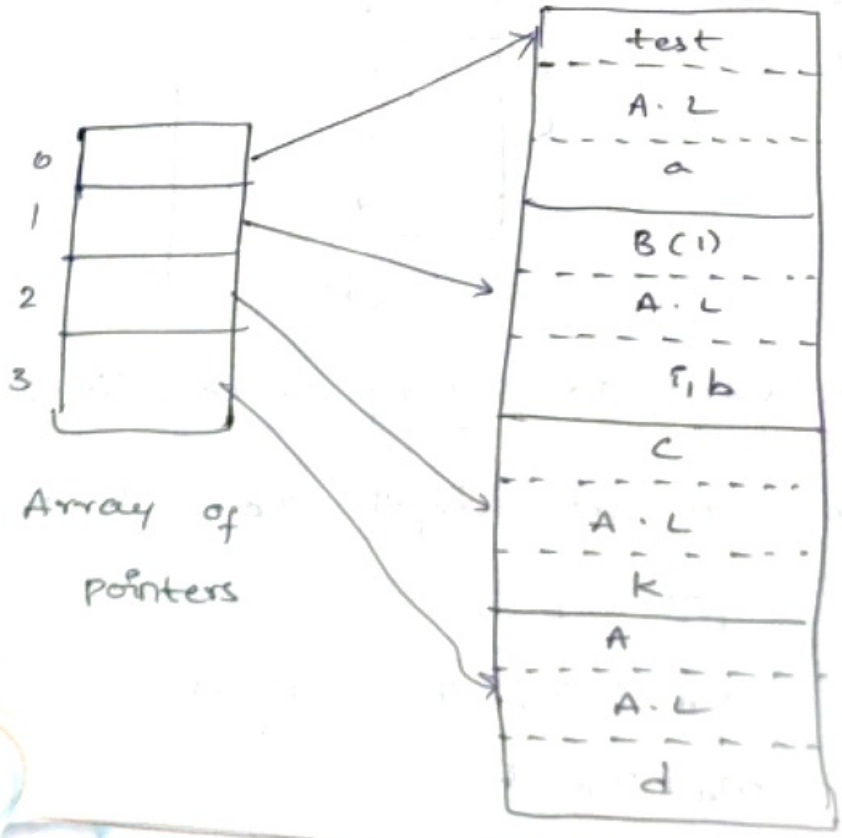
Case 1:- if $n_A < n_B$ then B is enclosed in A and $n_A = n_B - 1$

Case (ii): if $n_A \geq n_B$ then it is either a recursive call or calling to previous procedure

2. The access link of A.R of procedure A points to the A.R of procedure B where procedure A is nested in procedure B.
3. The A.R of B must be active at the time of pointing.
4. If there are several A.R's for B then most recent AR will be pointed

DISPLAY:-

- Access links are expensive to traverse across links and access non local data.
- Array of pointers to access non local data maintained. Such concepts is called display
- In display Array of pointers are used to access AR is maintained.
- Array Index is the nesting level.



- Array pointers are used to only accessible activation record.
- It faster to access but the draw back is it occupies more space.

Comparision between Access link and Display:-

- Access links takes more time to access non local variables especially when non local variables are at many nesting levels
- Display is used to generate code efficiently
- Display requires more space at run time than access links.

Heap Management :-

Two operations are performed (or) two basic functions

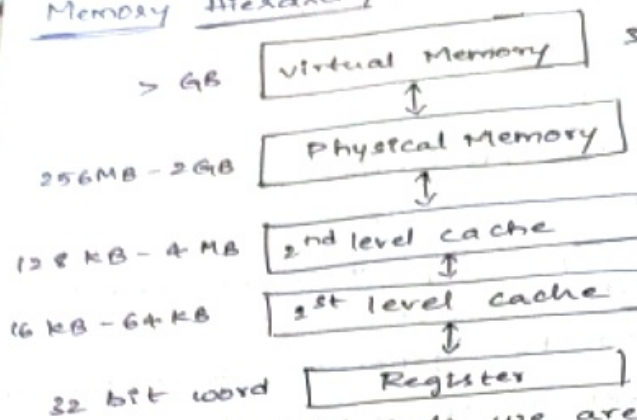
(i) Allocate Memory

(ii) Deallocate Memory

Properties :-

- Space efficiency :- Allocate memory to programs efficiently.
- Program Efficiency :- We have to allocate memory to programs to run faster.
- low Overhead :- No need to spend much time while allocating and deallocating.

Memory Hierarchy :—



Locality Of Programs :— 2 types of locality (what part of code we are executing (accessed))

Temporal locality :— Piece of code that frequently accessed

Spacial locality :— Piece of code accessed under closed locations.

- Programs spend 90% of time executing 10% of code
- Programs often contain main instructions that never executed.
- Only a small fraction of code is executed.
- Typical program spend most of time inner most loops and type recursive calls.

Parameter Passing :—

- Call by value
- Call by reference
- Call by restore
- Call by name.

Garbage Collection via Reference Counting:—

Garbage collection is another method of LISP management. When an application needs some free space to allocate the nodes and if there is no free space available then a s/m routine call garbage collector invoked.

This routine search the system for the nodes that are no longer accessible from an external pointer. Then these nodes are made available for re use by adding them to available pool. the system can make use of these free available space for allocating node.

Reference count is a special counter used during implicit memory allocation. If any block is referred by some other block then its reference count is incremented by '1'. That also means reference count (R.C) of particular block drop down to '0' that means that block is not referred one and hence it can be deallocated R.C's are best used when pointers blw blocks never appear in cycle.

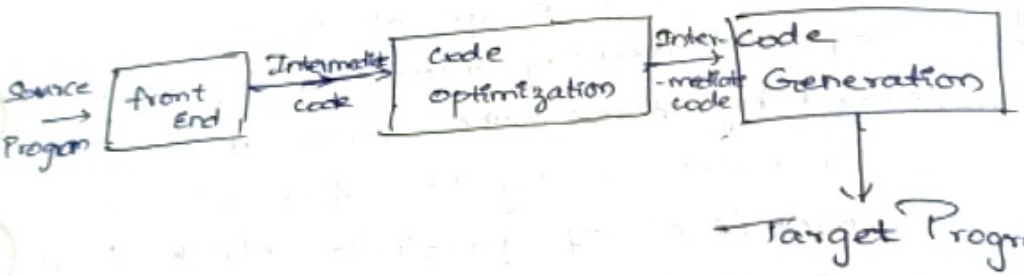
Advantages Of Garbage Collection:—

1. The manual memory management done by the programmer (using malloc() and free()) is time consuming and Error prone. Hence automatic memory management is done.
2. Reusability of memory can be achieved with the help of garbage collection.

Disadvantages:-

1. The execution of the program is paused and stopped during the execution of garbage collector.
2. Some time situations like thrashing may occur due to garbage collection. thrashing → Instead of executing the program more time spend to the garbage collection execution then the problem is called "thrashing".

CODE GENERATION :-



ISSUES TO CODE GENERATION :- The following are the issues to code generation

i. Input to code generation (i.e., I/p may be in 3-address code, Byte code, DAG

v intermediate code byte code Postfix notation

ii. Target Program

Based on Architectures

→ complex addressing modes

i. CISC - 3-address instruction

ii. RISC - 2-address instruction

↓

simple addressing modes

Stack based method

Drawback:- More no of copy statements are required

Three types of target program

Absolute Machine code:- Memory addresses are fixed at compile time. If the memory addresses are known at compile time then it is Efficient

Relocatable Machine code:-

Sub programs are to be compiled separately later they can be linked and loaded into the memory.

Assembly code:- It is efficient to generate assembly code at code generation phase. It is efficient to generate assembly code because it is platform independent.

Instruction Selection

Mapping intermediate program into code sequence. For one intermediate (IR) program several code sequences are possible i.e., it depends on the type of mnemonic code which we have chosen.

$$a = b + c$$

$$d = a + e$$

```
LD   R0, b
ADD  R0, c
ST   a, R0
LD   R0, a
ADD  R0, e
ST   d, R0
```

Here we are performing LD & ST on same register and these are unnecessary steps

```
MOV   R0, b
ADD   R0, c
ADD   R0, e
MOV   d, R0
```

} here we removed those unnecessary steps

(iv) Register Allocation & Assignment

The registers selected for set of values is also considered in code generation.

Register Allocation :-

Choosing register for set of values at a particular point in the program.

Register Assignment :-

Choosing a register that a particular value residing is called Register Assignment

(v) Evaluation Order

In what order the target code is generated also matters.

(vi) Target Machine :- The following should be considered for a target machine

Types of Instructions

(i) Load Operation :- Load memory operand into register

Ex:- LD R0, a

ST x, R0

(ii) Store Operation :- Store register operand into memory

Ex:- ST a, R0

(iii) Computational Operation :- means addition

Ex:- OP R1,

(iv) Unconditional Location

Ex:- BR

(v) Conditional

'R' value

Ex:- Bcon

ADDRESSING

1. Direct Add

variable

a(R0)

100(R

* R

Co

op

* 100

#

INSTR

Ex:-

=

(iii) Computational Operation:— Computational operations means addition, subtraction.

Ex:- OP R1, R1, R2

(iv) Unconditional jumps:— jump to the memory location

Ex:- BR (L) → label ⇒ Branch to label 'L'

(v) Conditional jumps:— Based on the status of 'R' value we can jump to label 'L'

Ex:- Bcond R, L

ADDRESSING MODES:—

1. Direct Addressing Mode:— We can name with a memory variable 'x'

→ Indexed mode ⇒ index of a is added to the e

a(R0) - The content of 'a' is added to the register R0 to know the address of the operand.

100(R2) - 100 is added to R2 to know the address of the operand.

*R - Register indirect addressing mode. Content of the register gives the address of the operand.

→ Indirect indexed Addressing mode.

*100(R) - (100 + content of R) gives the address where address of the operand is found.

#100 - It is an absolute immediate value

INSTRUCTION // COST :-

Ex:- b = a[i]

```
LD  R1, i
MUL R1, R1, 8
LD  R2, a[R1]
ST  b, R2
```

X = *P

```
LD  R1, P
LD  R2, 0(R1)
ST  X, R2
```

Instruction Cost :-

- for every op code one unit of cost is required. For every memory variable the cost is 1. for registers cost is 0.

Ex:- (i) LD R1, a → memory variable
↳ 2 units

(ii) MUL R1, R1, R2
↳ 1 unit

(iii) MOV a, b ⇒ 3 units

Peephole Optimization :-

- Peephole Optimization is machine dependent optimization.

(i) Removing unnecessary loads and stores.

Ex:- LD R1, a
ST a, R1

(ii) Unreachable code :- Remove all unreachable code inorder to decrease the cost.

Ex:- if debug = 1 goto L1
goto L2
L1 : print debugging information
L2 : .....

Now this can be modified as

if debug ≠ 1 goto L2
Print debug information
goto L2 :

of cost is
variable the
et is 0.
iable

pendent

stores.

unreachable
et.

if debug is system defined variable and its value of debug is 0

if 0 ≠ 1 goto L2

Print debug information

L2:

(3) Unnecessary jumps:—

Remove unnecessary jumps to jumps, jumps to conditional jumps, conditional jumps to jumps

Ex:- go to L1

L1: go to L2

L2:

Now it can be modified as

go to L2

L1: go to L2 ✗ Remove this statement bcz there is no possibility to reach L2 so we can remove L2

go to L2

L2: ...

Ex:- conditional jumps to jumps

if a<b go to L1

L1 : go to L2

L2 : ....

Modified as

if a<b go to L2

L1 : go to L2 ✗ "

L2 : ...

→ if a<b go to L2

L2: ...

Ex:- jumps to conditional jumps

```
    go to L1
L1: if a<b goto L2
    go to L3
```

modified as

```
if a<b go to L2
   go to L3

L2: ...
L3: ...
```

Algebraic Simplification:- Algebraic simplification is done at the intermediate code generation. It is

Ex:- $x = x + 0$, $x = x * 1$ } Identity rules

These rules can be simplified at Intermediate code generation itself. It is not necessary to go to target machine and then removing.

Machine Idioms:-

Inc A

This is also

$A = A + 1$

Inc operation is already present in the machine (in-built). then it requires only one operation.

```
LD R0, A
ADD R0, R0, 1
ST A, R0.
```

If we want to perform by our own then we have to perform 3 operations so we can use directly machine idioms.

This representation is constructed as follows:

1. Partition the intermediate code into basic blocks which are the maximum sequence of consecutive 3-address statements by holding the properties as
   i) The flow of control can only enter the basic block through the first instruction i.e., there are no jumps in the middle of the block.
   ii) Control will leave the block without halting (or) branching except possibly at the last instruction in the block
   iii) The basic blocks becomes the nodes of the program whose edges can indicate which blocks can follow each other block.

BASIC BLOCK :—

Algorithm for constructing basic blocks

I/P :- A sequence of 3-address instructions

O/P :- A list of basic blocks in which each instruction is assigned to exactly one basic block.

Method :—

1. Firstly, we determine those instructions in the intermediate code that are leaders.

Rules for finding leaders :—

1. The first 3-address statement in the intermediate code (IC) is a leader.

2. Any instruction that is a target of conditional or unconditional jump is a leader

3. Any instruction that immediately follows a conditional (or) unconditional jump is a leader.

Ex:- If $a < b$ then goto L1

i.e., After completing L1 then the next immediate instruction is considered as leader

- For each leader its basic block consists of itself and all the instructions up to but not including the next leader

Ex:- Source code which turns a 10x10 matrix into identity matrix where array elements take 8 bytes each

Source Code:-

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0
    for i from 1 to 10 do
        a[i, i] = 1.0
```

1. $i = 1$ (leader)
2. $j = 1$ (leader)
3. $t_1 = i * 10$ (leader)
4. $t_2 = t_1 + j$
5. $t_3 = t_2 * 8$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. $j = j + 1$
9. if $j <= 10$ goto (3)
10. $i = i + 1$ (leader)
11. if $i <= 10$ go to 2
12. $i = 1$ (leader)

is a target of condition
is a leader
immediately follows
nditional jump is a

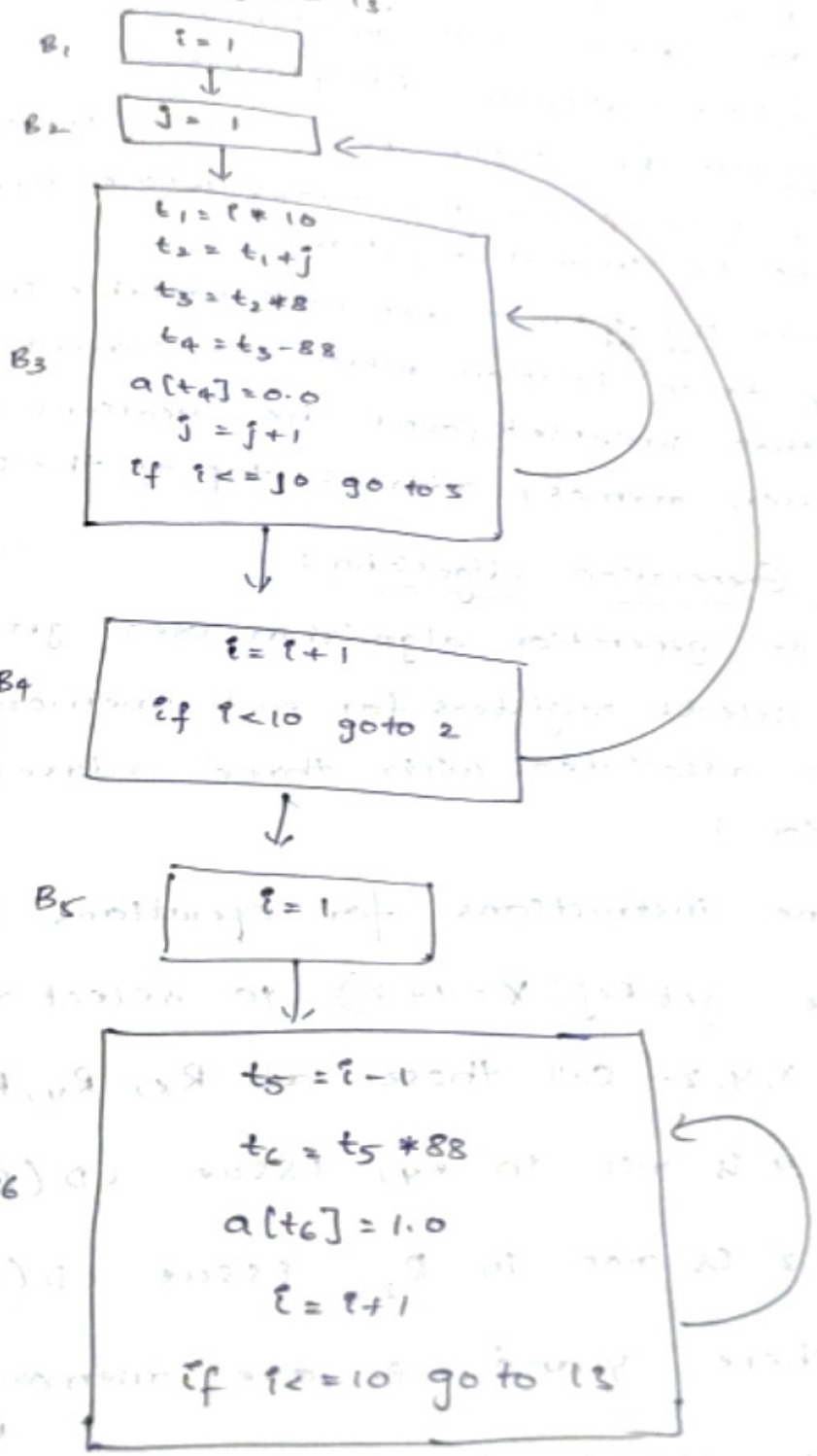
goto L1
1 then the
considered as

: block consists
tructions up to
leader

10x10 matrix
array elements

to (3)
der)
to 2

```
13  t5 = i-1 (leader)
14  t6 = t5 * 88
15  a[t6] = 1.0
16  i = i+1
17  if i<=10 go to 13
```



A Simple Code Generator:-

- Efficiency of the code generations depends on how to use registers
- Registers are used to store operands, temporary, global values, stack pointers etc.

Register & Address descriptors:-

Register Descriptor:- for each register it will keep track of variable names whose current value is in that registers.

Address Descriptor:- for each variable it will keep track location where the current value of that variable is found. the locations may be register, memory address, top of the stack etc

Code Generation Algorithm:-

Code generation algorithm uses getReg(I) which selects registers for each memory location associated with three address instruction i.

get Register

Machine instructions for operations X = Y+Z

i. Use getReg(X = Y + Z) to select registers of X, Y, Z. Call those as R_x, R_y, R_z

ii. If Y is not in R_y, issue LD(R_y, Y)

iii. If Z is not in R_z, issue LD(R_z, Z)

where Y and Z are memory locations

iv. Issue ADD(R_x, R_y, R_z)

Machine Instructions for Copy Statements:-

1. Issue LD (R4, y) if y is not in R4

Managing Register & Address descriptors:-

Note: For the instruction LD(R, x)

(a) change register description for R so it holds only x

(b) change address descriptor for x by adding R as additional location

2. For ST(x, R) change address descriptor (A.D) to include the own memory location

3. For ADD(Rx, Ry, Rz)

(a) Change Register Descriptor (RD) Rx so it holds only x.

(b) Change AD for x the only location is Rx.

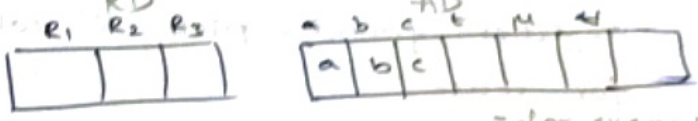
(c) Remove Rx AD's of any other variable

4. For the instruction x = y LD(Ry, y)

(a) Add x to the RD of Ry

(b) Change AD for x so the only location is Ry

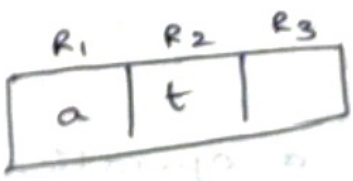
Ex:- t = a - b
u = a - c
v = t + u

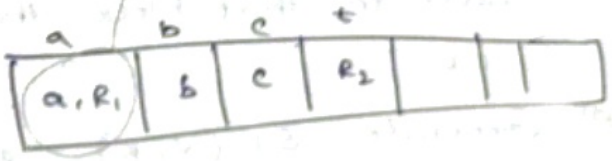


- for every LD operation we have to keep Register locates in AD.

3-address code:-

t = a - b
LD R1, a
LD R2, b
SUB R2, R1, R2



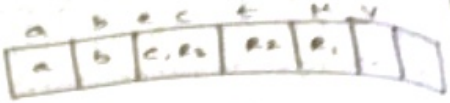


u = a − c

LD R3, c

SUB R1, R1, R3

R1	R2	R3
u	t	c

a	b	c	t	u	v
a	b	c, R3	R2	R1	

v = t + u

ADD R3, R2, R1

R1	R2	R3
u	t	v

a	b	c	t	u	v
a	b	c	R2	R1	R3

getReg() :—

- How to pick registers for memory variable y

1. If y is currently in register pick it as R_y.
2. If y is not in Register but there is a register currently empty then pick it as R_y.
3. When y is not in register and no empty register found pick 'R' which holds 'v' for reuse. (Here register contains 'v')
   - (a) If AD for 'v' says v is somewhere beside 'R' then we can use it (i.e., v is present somewhere else also)
   - (b) If 'v' is x which is computed in 'i' then ok (i.e., x = y + z, now we are computing it for new values so there is no use of old value then we can remove it)
   - (c) If 'v' is not used later in that block then ok. (i.e., If v is not used anywhere in the block)
4. (d) Generate ST (V, R) such a operation is called "spill" (i.e., no register is empty so we have to store V in memory and store value in R)

Register Allocation:- At a particular point in program, what values should be present in the registers

Register Assignment:- In which register each value should reside

Global Register Allocation:-

Assign registers to frequently used variables and keep registers consistently across boundaries

Usage Count:-

Usage count means how many times the variable is used in the block.

The formula for usage count is

$$\sum_{\text{Blocks } B \text{ in } L} use(x,B) + 2 \times live(x,B)$$

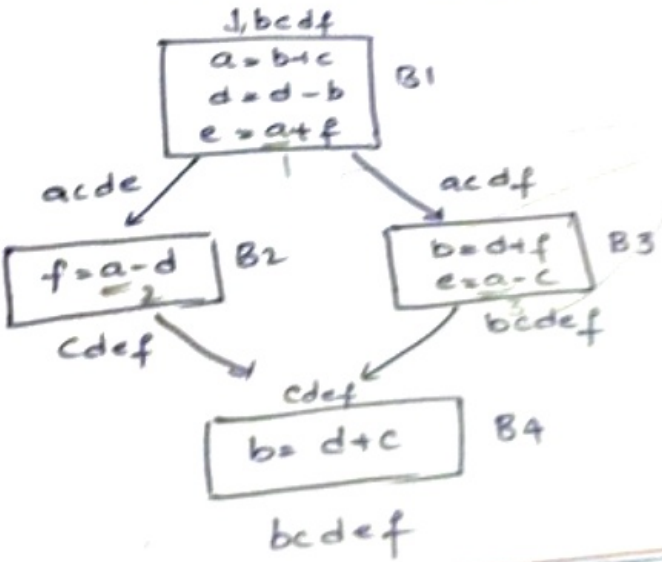
L → Loop

use(x,B) = No. of times x is used in B

$$live(x,B) = \begin{cases} 1 \Rightarrow \text{if } x \text{ is live on exit from } B. \\ 0 \Rightarrow \text{if } x \text{ is not used other than block} \end{cases}$$

Here 2* live(x,B) because a write cost is saved

Ex:-



4/10/2016

UNIT - V

Quick Sort:-

```
void quicksort (int m, int n)
{
   int i, j, v, x;
   if(m<=n)
     return;
   i=m-1;
   j=n;
   v=a[n];
   while(1)
   {
    do
    {
      i=i+1;
    } while(a[i]<v);

    do
    {
      j=j-1;
    } while (a[j]>v);

    if (i>=j)
      break;
    x=a[i];
    a[i]=a[j];
    a[j]=x;
   }

   x=a[i];
   a[i]=a[n];
   a[n]=x;

  quicksort (m, j);
  quicksort (i+1, n);
}
```

3- address code:-

```
1.  i=m-1      (lea
2.  j=n
3.  t3=4*n
4.  v=a[t3]
5.  i=i+1      (l
6.  t2=4*i
7.  t3=a[t2]
8.  if (t3<v)
9.  j=j-1
10. t4=4*j
11. t5=a[t
12. if(t5>
13. if  i>
14. t6=4
15.  x=
16.  t7=
17.  t8=
18.  a[
19.  a[t
20.  t
21.  a
22.
23.
24
25.
26
2
2
```

i = m-1

3- address code:—

i = m-1 (leader)

1. j = n
2. t1 = 4 * n
3. v = a[t1]
4. i = i+1 (leader)
5. t2 = 4 * i
6. t3 = a[t2]
7. if (t3 < v) go to 5.
8. j = j-1 (leader)
9. t4 = 4 * j
10. t5 = a[t4]
11. if (t5 > v) go to, 9.
12. if i >= j go to 23. (leader)
13. t6 = 4 * i (leader)
14. x = a[t6]
15. t7 = 4 * i
16. t8 = 4 * j
17. ~~a[t7]~~ t9 = a[t8]
18. a[t7] = t9
19. t10 = 4 * j
20. a[t10] = x
21. go to 5
22. t11 = 4 * i (leader)
23. x = a[t11]
24. t12 = 4 * i
25. t13 = 4 * n
26. t14 = a[t13]
27. a[t12] = a[t14] t14
28. t15 = 4 * n
29. a[t15] = x

Here array size is 4 bytes

Here we are checking a condition. So it is better to store in a temporary

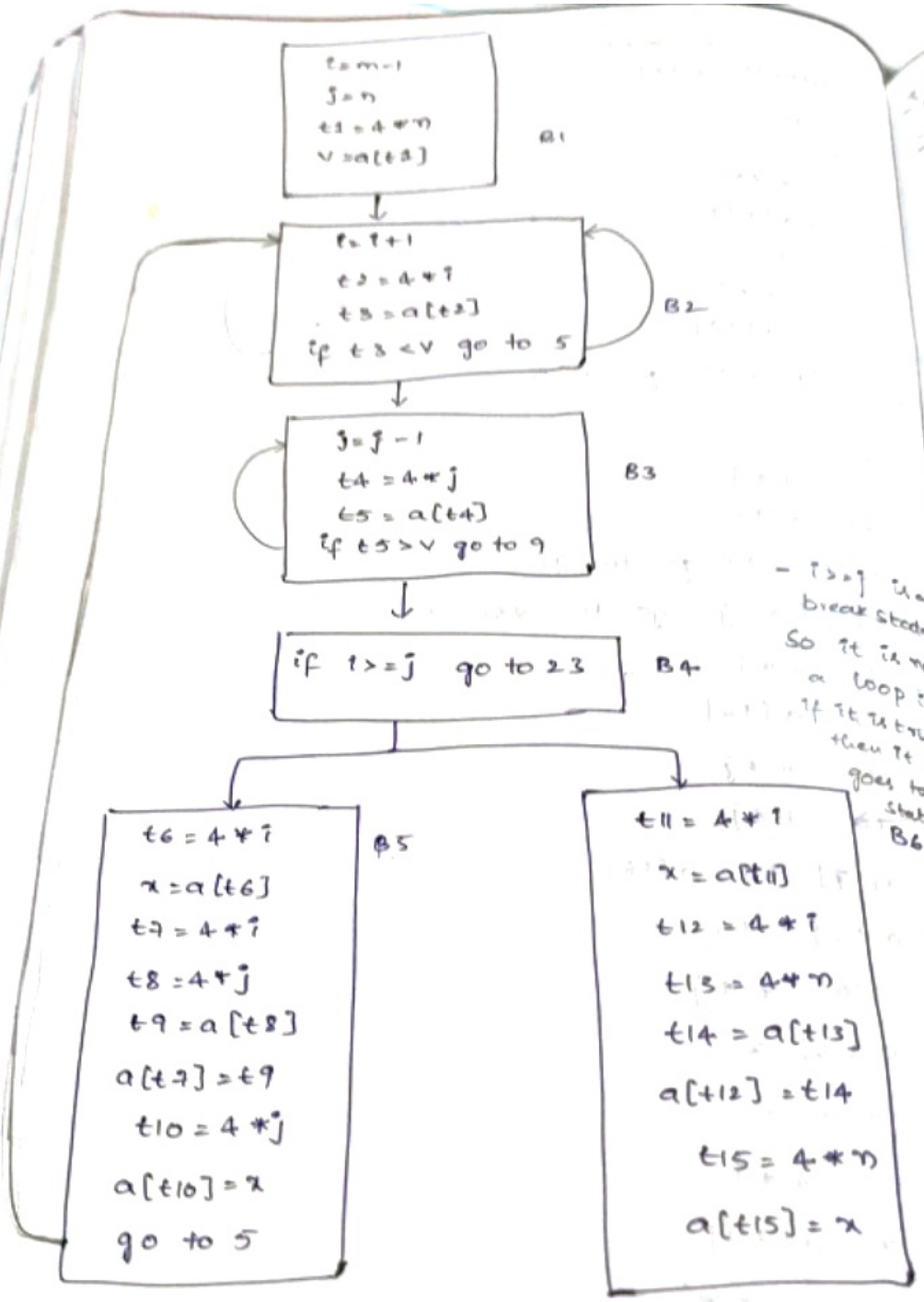
t8 = a[t7]

t9 = 4 * j

a[t9] = t8

a — We cannot directly assign one array value to another array

i.e., a[i] = a[j] is not possible


i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v go to 5
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v go to 9
B3
if i >= j go to 23
B4
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
go to 5
B5
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
B6

Optimization techniques:-

Common Sub-Expression Elimination:- (CSEE)

An occurrence of expression 'E' is called CSEE, if E was previously completed and the values of variable in E have not changed since previous computation.

From the block B5, remove t7, t10. Use t6 instead of t7 and t8 instead of t10

t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
go to 5

B5

t11 = 4*i
x = a[t11]
t13 = 4*n
t14 = a[t13]
a[t11] = t14
a[t13] = x

B6

- Remove t6 and t8, t11 in B5 and B6. Use t2 instead of t6 and t4 instead of t8 (in B5) t2 instead of t11 and t3 instead of t13.

x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
go to 5

B5

B6

x = a[t2]
t14 = a[t4]
a[t2] = t14
a[t3] = x

- Remove t9. Use t5 instead of t9.

x = a[t2]
a[t2] = t5
a[t4] = x
go to 5

B5

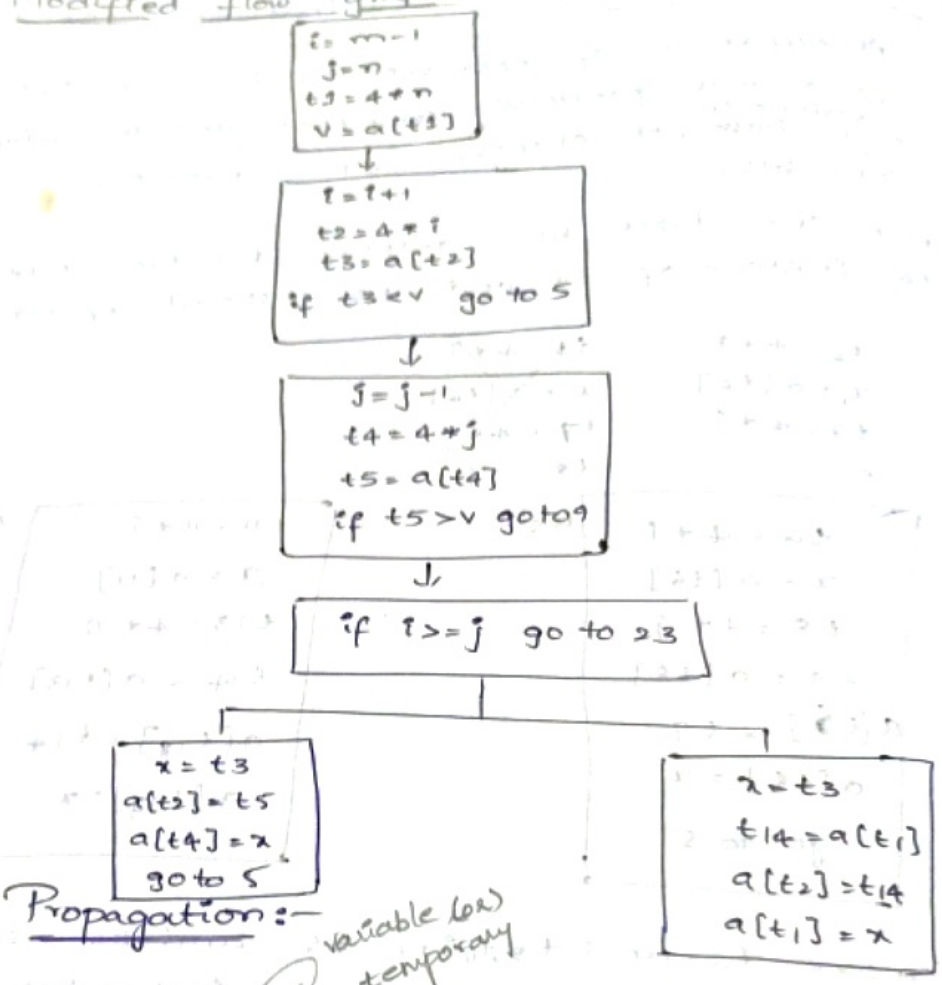
x = t3
where t3 = a[t2]

B6
x = t3

x = a[t2]
t14 = a[t3]
a[t2] = t14
a[t3] = x

- [3>=] is a break statement so it is not a loop i.e, if it is true then it directly goes to 23rd statement B6

Modified flow graph:

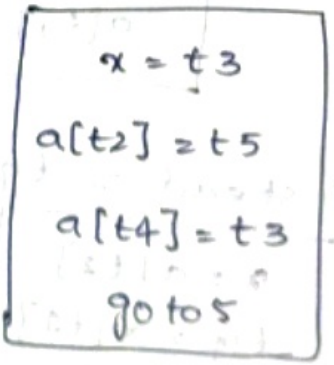


Copy Propagation:-

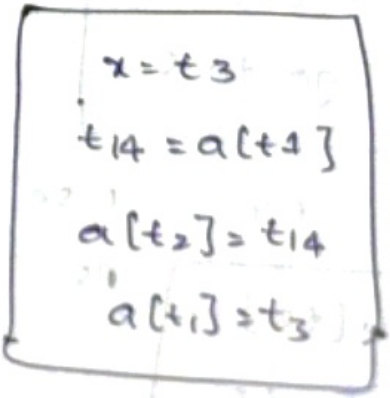
- The form $f = g$ (variable (or) temporary) is called copy statement
- The idea behind copy propagation is to use 'g' for 'f' after the copy statement.

i.e., for $x = t3$ use $t3$, for 'x'

- we will not consider array references



B5



B6

Dead code elimination:-

A variable is live at a point in a program if its value can be used subsequently in some statements otherwise it is dead at that point wherever their values never get used.

```
a[t3] = t5
a[t4] = t5
goto s
```

```
t14 = a[t3]
a[t2] = t14
a[t1] = t5
```

i.e., here we are removing $x = t_5$ because it is a dead code i.e, x value never used in the subsequent statements.

Loop Optimization:-

The running time of a program may be improved if you decrease the no. of instructions in an inner loop even if you increase the amount of code outside of the loop

- There are three techniques in loop optimization.
  (i) Code Motion
  (ii) Induction variable elimination
  (iii) Reduction in strength.

Code Motion:-

Code Motion moves the code outside of the loop. This transformation takes the expression that yields the same result in each and every iteration of the loop and place that expression outside of the loop

Eg:- while (i <= limit - 2)
{
}

Here even let us consider limit = 10 and every time if we want to go for while loop we have to check the condition i.e., we have to perform subtraction every time; so costlier results in

This can be modified as

```
t = limit - 2
while (i <= t)
{
  ....
}
```

Induction Variable Elimination:-

Induction variable is of the form

$v_i = \pm c_1$, $v_i = v_i \pm c_2$

where c_1 , c_2 are constants

i.e., B_2 :-

$i = i + 1$	whenever i is incremented by 1
$t_2 = 4 * i$	corresponding t_2 incremented by 4

replaced by

$t_2 = t_2 + 4$

B3 :-

$j = j - 1$	whenever j is decremented by 1
$t_4 = 4 * j$	corresponding t4 is decremented by 4

replaced by

$t_4 = t_4 - 4$

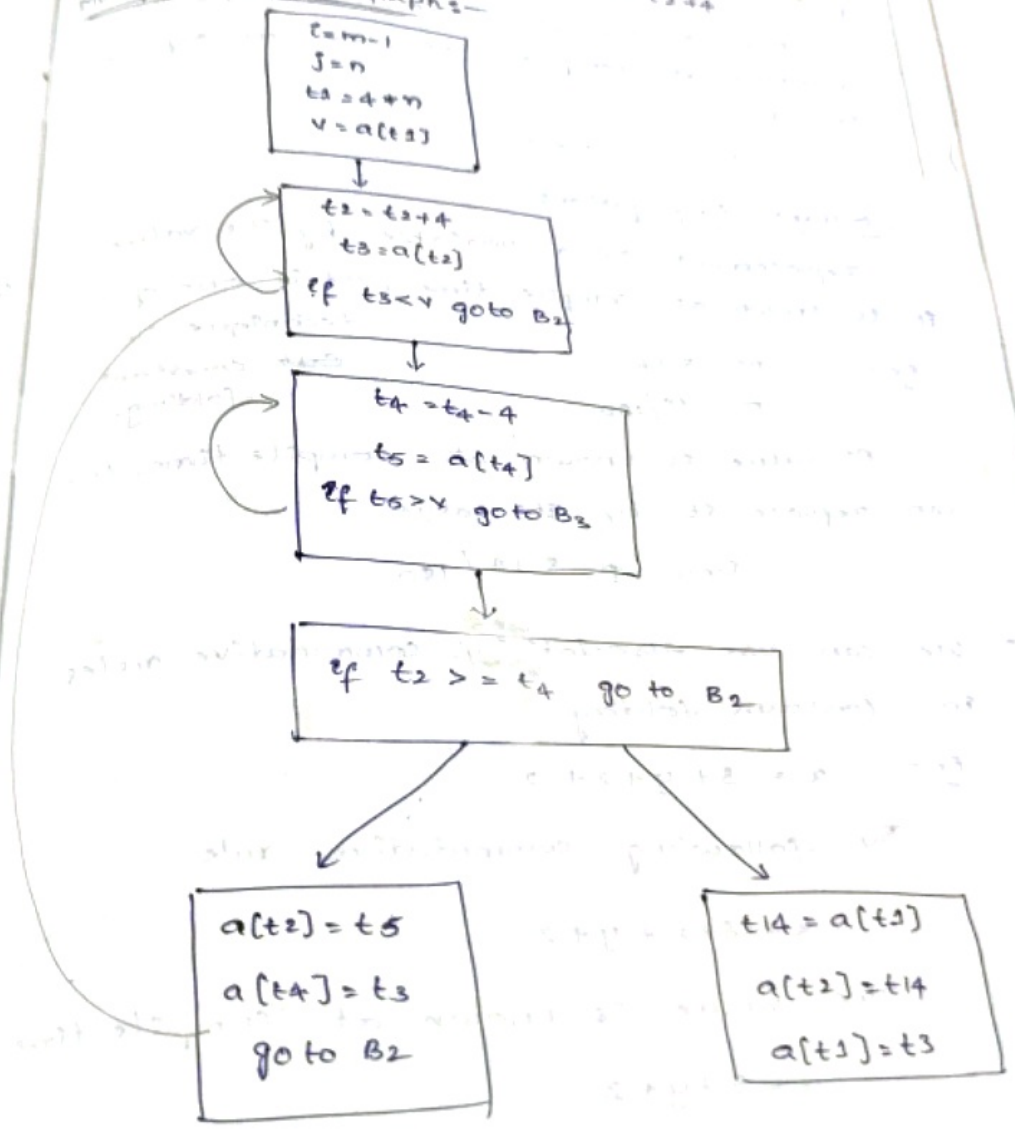
Strength Reduction:
Replacing expens...
is called strength ...
form B_3, multiplication ...
replaced by addit...
Modified flow ...

Strength Reduction:-

Replacing expensive operator by cheaper operator is called strength reduction

i.e. Multiplication operation in $t_3 = 4 * i$ is replaced by addition operation $t_3 = t_3 + 4$

Modified flow Graph:-



Constant folding:-

Constant folding is the replacement of an expression by its value, if that expression is evaluated at compile time

Ex:- $z = 3+4+y+x$

Here in the above expression 3+4 is evaluated at compile time, we can replace it by 7

$\therefore z = 7+y+x$

Constant Propagation:-

Replacement of a variable by its value, if it is known at compile time. It is one of the technique in constant folding.

Ex:- $PI = 3.14$

$P = PI/180$

PI value is known at compile time, we can replace it by its value

i.e., $P = 3.14/180$

- We can use associativity, commutative rules in constant folding.

Ex:- $x = 3+y+2+2$ (i.e., they are at different places)

By following commutative rule

$x = 3+2+y+2$

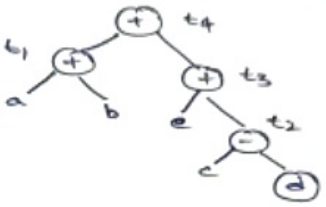
The value is known at compile time

$\therefore x = 5+y+2$

Instruction scheduling:-

The order of the 3-address code effects the cost of object code being generated. In the sense that by changing the order we can obtain the object code with minimum cost. Thus code optimization can be achieved by scheduling the instructions in proper order, such a technique is called instruction scheduling.

Ex:- $t_1 = a+b$
$t_2 = c-d$
$t_3 = e+t_2$
$t_4 = t_1+t_3$



Object code:- (Source, Destination)

```
MOV  a, R0
ADD  b, R0
MOV  c, R1
Sub  d, R1
MOV  R0, t1
MOV  e, R0
ADD  R0, R1
MOV  t1, R0
ADD  R1, R0
MOV  R0, t4
```

Now we are changing the order of code

(1) $t_2 = c-d$ (2) $t_3 = e+t_2$
(3) $t_1 = a+b$ (4) $t_4 = t_1+t_3$

```
MOV  c, R0
Sub  d, R0
ADD  e, R0
MOV  a, R1
Add  b, R1
Add  R1, R0
MOV  R0, t4
```

Inter procedural Optimization:-

Optimizations applied across multiple procedures is called inter procedural optimization.

We have to explain optimization techniques

Semantic preserving Optimization:

Semantic meant meaning i.e., we should not change the meaning of the expression.

We have to write the same optimization techniques.

1. Draw the block diagram of phases of compiler and indicate the main functions of each phase

2. Define lexeme, token & pattern. Identify the lexemes that make up the tokens in the following program segment, indicate corresponding token, pattern

```
void swap (int i, int j)
{
    int t;
    t = i;
    i = j;
    j = t;
}
```

3. Write a RE for identifiers & Reserved words. Design the transition diagram for them

4. Explain the three general approaches for the implementation of lexical analysis

5. Compare compiler & interpreter with suitable examples.

6. Construct NFA equivalent to R.E R=(a+b)ab

7. Give general format of LEX program

8. Write RE for the set of words having a, e, i, o, u appearing in that order, although not necessarily consecutively.

9. State the reasons for separating lexical analysis & Syntax analysis.

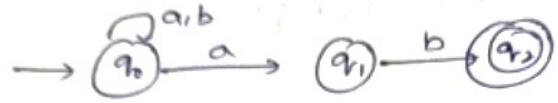
10. Describe the lexical errors and various Error recovery strategies

11. Write R.E for relational operator & design transition diagram for them

---

→ NFA equivalent to RE

Given $R = (a+b)^* ab$



→ alpha →

const → b|c|d....z

string → (const)* a (const)* e (const)* i (const)* o (const)* u (const)*

→

Tokens	lexemes	Pattern
void	void	void
int	int	int
id	swap, i, j, t	letter (letter|digit)*
relop	=	< (or) <= (or) > (or) >= (or) <> (or) ==
Spl-Sym	(), {}, ;, ,	(|)|{|}|;|,

Give general format of LEX program

A lex program has the following form:

declarations
%%
translation rules
%%
auxiliary functions

declarations:- The declaration section includes declaration of variables, manifest constants (identifiers declared to stand for a constant Ex:- name of a token), and regular definitions

Ex:- digit [0-9]

letter [A-Za-z]

Translation rules:- The translation rules each have the form

Pattern {Action}

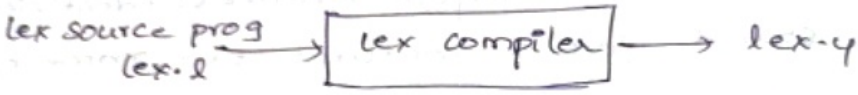
Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created.

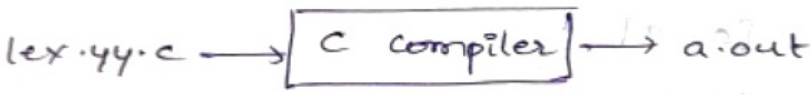
Auxiliary functions:- It holds the additional functions which are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

Creating a lexical analyzer with Lex:-

An i/p file, which we call lex.l is written in the lex language & describes the lexical

analyzer to be generated. The lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c. The later file is compiled by the C compiler into a file called a.out, as always. The C-compiler o/p is a working lexical analyzer that can take a stream of i/p characters & produce a stream of tokens.





(UNIT-II)

(a) Lexical Analysis Vs Syntax Analysis

(b) construct the predictive parser for the following grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(c) classification of parsing techniques

2. (a) show the grammar is ambiguous

$S \rightarrow 0S1 \mid SS \mid e$

(b) Explain non-recursive predictive parsing with an example.

(c) limitations of recursive decent parser.

3. (a) what is left recursion & left factoring

(b) verify whether the following grammar is LL(1) (or) not

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

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

(c) Discuss about Error recovery strategies in predictive parsing.

4. (a) what is an LL(1) grammar? when the grammar is said to be LL(1)

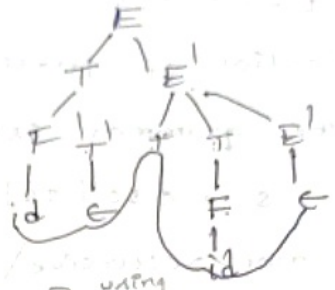
(b) Design Non-recursive predictive parser for the following grammar

$S \rightarrow AaAb \mid BbBb$

$A \rightarrow e$

$B \rightarrow e.$

(c) Discuss how Brute force approach operates in top down parsing.



8 recursive calls

Stack Memory

Memory- Info about 8 recursive calls

- By using recursive calls, we have to store records

to by using predictive parser the memory storage will be less when compared to Recursive decent parser i.e., It has table where will check the productions

(W-17-5M)

2. a) Parse the i/p string int id, id; using shift reduce parser for the grammar S → TL;

T → int / float

L → L, id

L → id

b) Write the steps for efficient construction of LALR parse table. Explain with an example.

↳ Algorithm for construction of table.

3. (a) Write the steps for construction of CLR Parsing table

(b) Explain the compaction of LR Parsing table with an example

3. (a) construct the collection of LR(0) items and goto graphs for the grammar

S → SS | a | ∈

(b) Explain the process of handling dangling else ambiguity.

4. (a) Draw the structure of LR Parser

(b) Compute closure & goto for the grammar.

S → Aa | bAc | Bc | bBa

A → d

B → d

(c) Compare bottomup approaches & topdown approaches.

UNIT IV

(a) Translate the assignment $x = A[4, 2]$ into three address statements.

(b) Define Type checker and write down the specifications for simple type checker

2. (a) Write down quadruple, Triple, Indirect Triple for the Expression $-(a * b) + (c + d) - (a + b + c + d)$

(b) Write an algorithm for dependency graph for a given parse tree

3. (a) Construct syntax tree and postfix notation for the expression $(a + (b * c)) \uparrow d - e / (f + g)$

$= abc * + d \uparrow efg + / -$

(b) Explain in detail how an L-attributed grammar can be converted into a translation scheme.

4. (a) Construct the syntax tree and draw the DAG for an Expression

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

(b) Write syntax directed definition for constructing syntax tree of an Expression Delve from the grammar

$E \rightarrow E + T / E - T / T$

$T \rightarrow (E) / id / num$

2. (a)

$t_1 = a * b$

$t_2 = c *$

$t_3 = a$

$t_4 =$

Quadruple :—

	op	arg
*		a
-		
+		
+		
+		
+		
+		
-		

Triple :

	op
0	*
1	
2	
3	
4	
5	
6	
7	

= A [i, 2] into

ite down the
checker

Indirect Triple
+d)- (a+b)*(c+d)
dency graph

postfix notation
)↑d-e/(f+g)
attributed
into a

nd draw
(a+b)

ition for
n Expression

2.(a)

$t_1 = a + b$
~~t_2 = minus t_1~~
$t_3 = c + d$
~~$t_4 = c + d$~~
$t_5 = a + b$
~~$t_6 = c + d$~~

1) $t_1 = a + b$
2) $t_2 = -t_1$
3) $t_3 = c + d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a + b$
6) $t_6 = t_5 \cdot d$
7) $t_7 = t_6 + t_4 d$
8) $t_8 = t_4 - t_7$

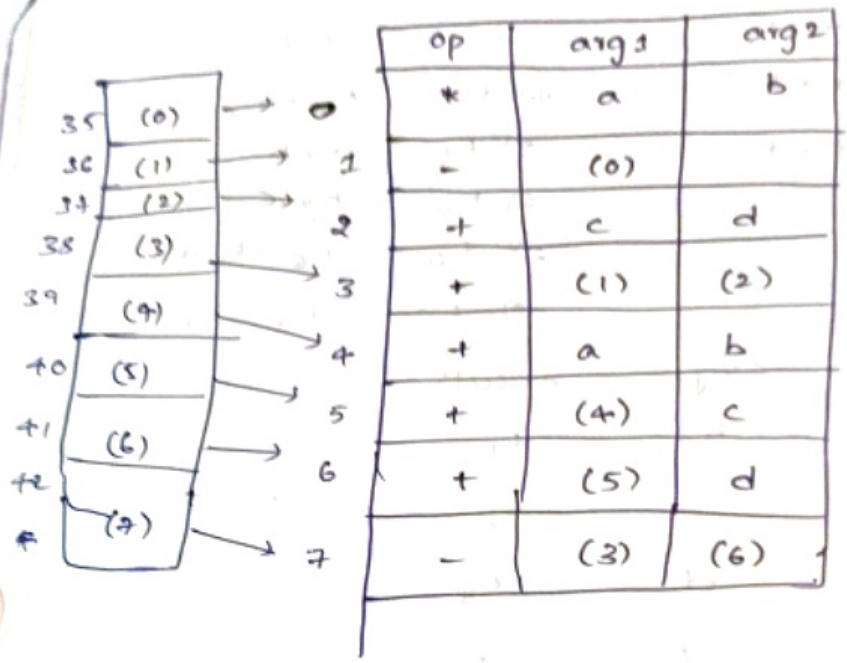
Quadruple :—

	op	arg1	arg2	result
	*	a	b	t_1
	-	t_1		t_2
	+	c	d	t_3
	+	t_2	t_3	t_4
	+	a	b	t_5
	+	t_5	c	t_6
	+	t_6	d	t_7
	-	t_4	t_7	t_8

Triple :—

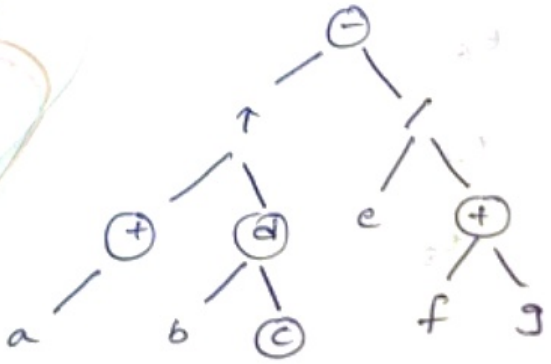
	op	arg1	arg2
0	*	a	b
1	-	(0)	
2	+	c	d
3	+	(2)	(1)
4	+	a	b
5	+	(4)	c
6	+	(5)	d
7	-	(3)	(6)

Indirect Triple:-



op	arg1	arg2
*	a	b
-	(0)	
+	c	d
+	(1)	(2)
+	a	b
+	(4)	c
+	(5)	d
-	(3)	(6)

3(a) abc * + d ↑ ef g+/-



4(a)