COMPILERS:- Software that accepts a program as an input, usually written in a high level language, produces an object program which is machine language equivalent, as an output.

ASSEMBLER:- They were written to automate the translation of assembly language to Machine level language.

INTERPRETER:- Instead of producing object program, it directly executes the operations specified in source program.

LOADER:- places the Program into memory, prepares for execution
→ Same can be done by the assembler, but it wastes the core memory by placing itself alongside with program.

MACRO:- Treats identical parts of the program as definition and substitutes it for all future references.

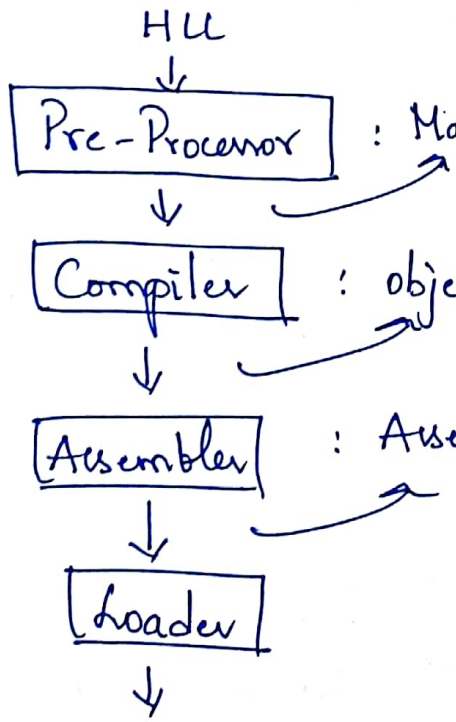


Fig: HLL → MLL journey.

Scanned by CamScanner

STRUCTURE OF A COMPILER:-

Analysis:-

Front-end of compiler.
Breaks the Source program & imposes grammar on it.
Creates an intermediate representation.
Produces Informative messages to user, if the program is unsound.
Contains Symbol table, a Special data structure in which, Source program's information is Stored.

) Synthesis:-

· Back-end of Compiler.
- Constructs desired target program based on Symbol table, intermediate representation.

) COMPONENTS OF A COMPILER:-

) Lexical Analyzer:-

- Also 'Scanner'.
- It reads character stream, groups into Sequences Called as 'lexemes'
- for each lexeme, there will be a 'token' as output.
- Syntax:- <token_name, value>

Input to Next phase.

points to an entry in the Symbol table.

Scanned by CamScanner

- e.g: (position = Initial + rate * 60)

↓

Lexical Analyzer

↓

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

where,

< > : token-form

1	position
2	Initial
3	rate

Symbol table

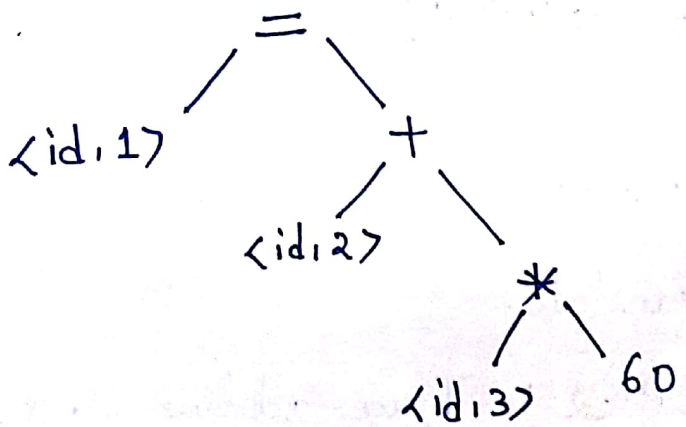
} lexemes, mapped as an identifiers.

→ =, +, *, 60, does not need an attribute value.

→ Spaces are discarded.

2) Syntax Analysis:-

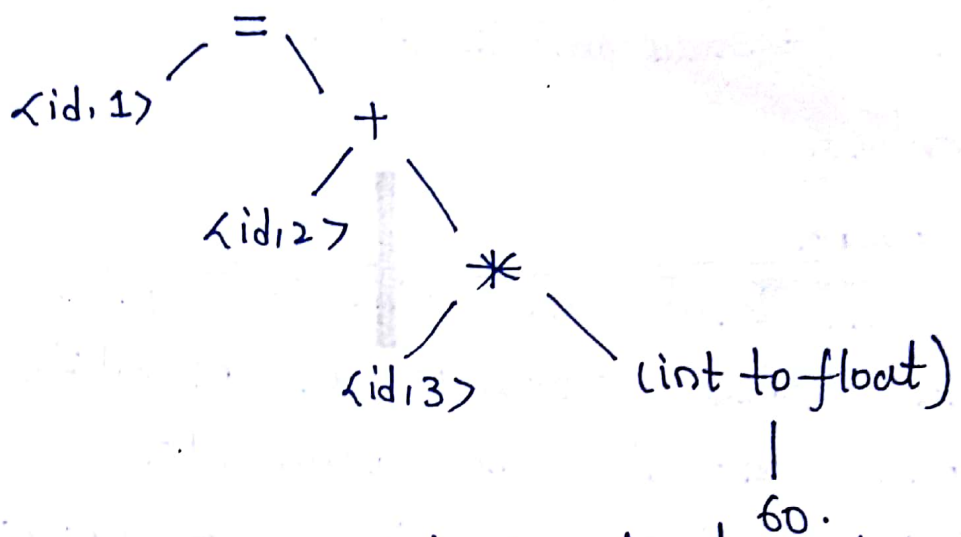
- Also 'Parsing'.
- It takes the input, and builds a syntax tree, in which parent nodes represent an operation, and child nodes represent the arguments.
- Input:- <id, 1><=><id, 2><+><id, 3><*><60>; Symbol-table
- output:-



Scanned by CamScanner

3) Semantic Analysis:-

* - It performs type checking.
* - It saves type information in either the Tree or Table for further use.
- It checks the Source code for consistency with the language defenition.
- It performs necessary 'coercion'.
- Input:- Table & Tree.
- output:- Modified Tree.



- if id, values are provided in float and viceversa.

4) Intermediate Code Generator:-

- properties:-
  - (a) easy to produce
  - (b) easy to translate.
- one form is called "Three-address Code".
  - ↳ each operand acts like a register.
  - ↳ at most one operator on right side.

Scanned by CamScanner

→ 3AC:-

t_1 = inttofloat(60) ..... (1)

t_2 = id3 * t_1 ..... (2)

t_3 = id2 + t_2 ..... (3)

id1 = t_3 ..... (4)

5) Code optimisation:-

t_2 = id3 * 60.0 ..... (5) (1,2 optimized)

id1 = id2 + t_2 ..... (6) (3,4 optimized)

6) Target Code Generation:-

- Input as Intermediate, optimized code and maps it into the target language.

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1.
```

where,

LDF ← Load float

MULF ← Multiply float

ADDF ← Add float

STF ← Store float

Scanned by CamScanner

Charecter stream

↓

Lexical Analyzer

token stream

↓

Syntax Analyzer

↓

Syntax tree

↓

Semantic Analyzer

↓

Modified syntax tree

Analysis

↓

Intermediate Code generator

synthesis

↓

Intermediate representation

↓

Machine Independent code optimizer

↓

Intermediate representation

↓

Code Generator

↓

target-machine code

↓

Machine dependent code optimizer

↓

Target Machine code.

Scanned by CamScanner

SCIENCE OF BUILDING A COMPILER:-

Concept:- Take a problem

↓

Formulate Mathematical abstraction

↓

Solve using Mathematical techniques.

→ **Modelling in Design & Implementation:-**

- Designing Models & Algorithms.
- Fundar- Finite State Machines, Regular expressions, CFG.

→ **Science of Code optimization:-**

→ **Design objectives:-**

- optimization must be Correct.
- must improve the Performance.
- Compilation time must be Kept reasonable.
- Effort required must be manageable.

Conclusion:-

- The Developers should have influence not just over the Compilers, they create, but all the programs that they Compile.
- We should learn not only how to build a Compiler, but also the general methodology of Solving problems
- Approach : Theory + experimentation.

Scanned by CamScanner

⇒ **Boot-Strapping:-** Writing a Compiler in the language, it is trying to produce object code based on already existing. 6-(b)

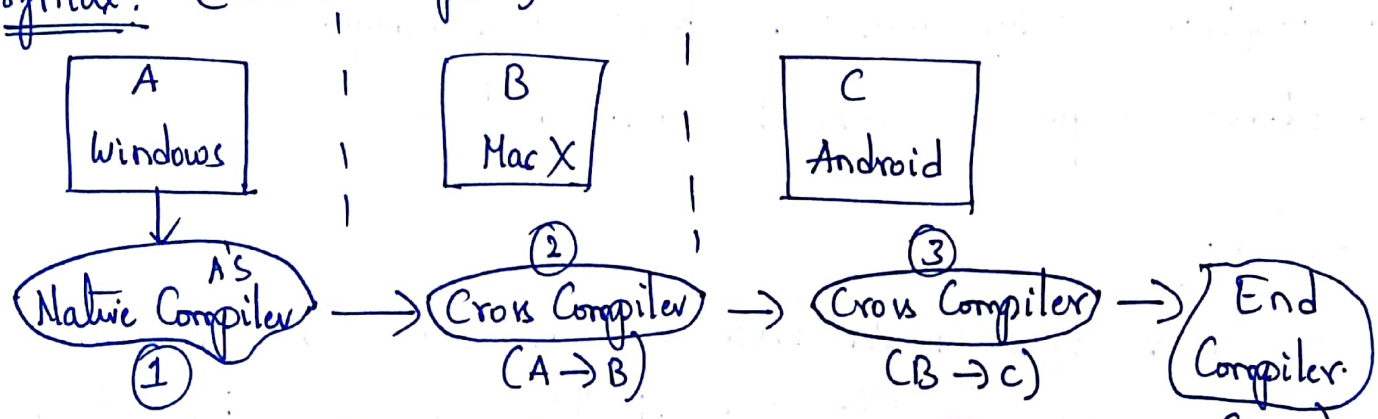
Def2:- The process by which a simple language is used to translate a more complicated program, which in turn may handle even more complicated program. (James. A. Van) 1993.

⇒ **Cross-Compiler:-**

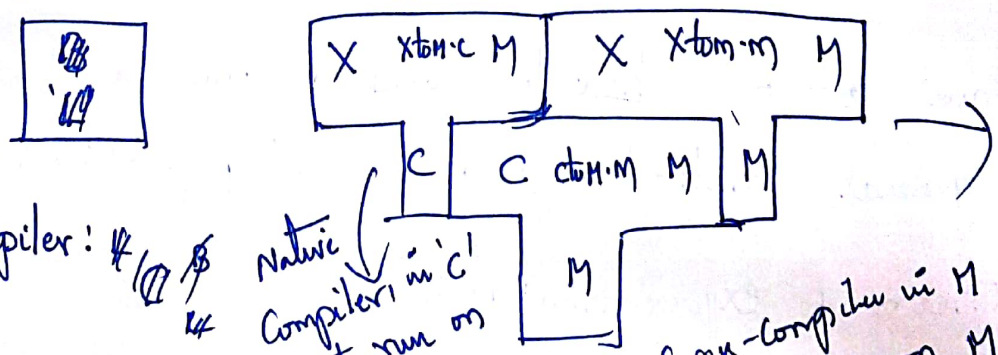
Compiler capable of creating executable code for a platform other than the one, on which the compiler is running.

e.g: Compiler running on Windows, generates a code that can run on linux / android, is a cc.

→ **Syntax:-** (Cross-Compiler).



→ **Syntax:-** (Boot-Strapping)



only B, C, No:4

Boot-strapped Compiler in M, used to run on M, from X.

Compiler: Native Compiler, in 'C' used to run on M from X

Cross-Compiler in M used to run on M from C

Scanned by CamScanner

Lexical Analysis

→ LA - Role:-

- Reads charecters from input, groups them into lexemes. (Main)
- It eliminates the blank spaces and comments. (Secondary)

→ Reading Ahead:-

- LA needs to read ahead to decide on a token to be returned to parser
- e.g: After '>' → if nothing, can be a header file (or) greater than
  → if succeeded by '=', then greater than or equal to.
- This property drives LA, to maintain an 'Input Buffer'.
- A pointer is used to keep track of portion of data that has been analyzed. Backtracking of data possible
- Next input charecter is stored in a variable 'peek'
- peek = NULL, if LA reads symbols, e.g. *.

→ Constants:-

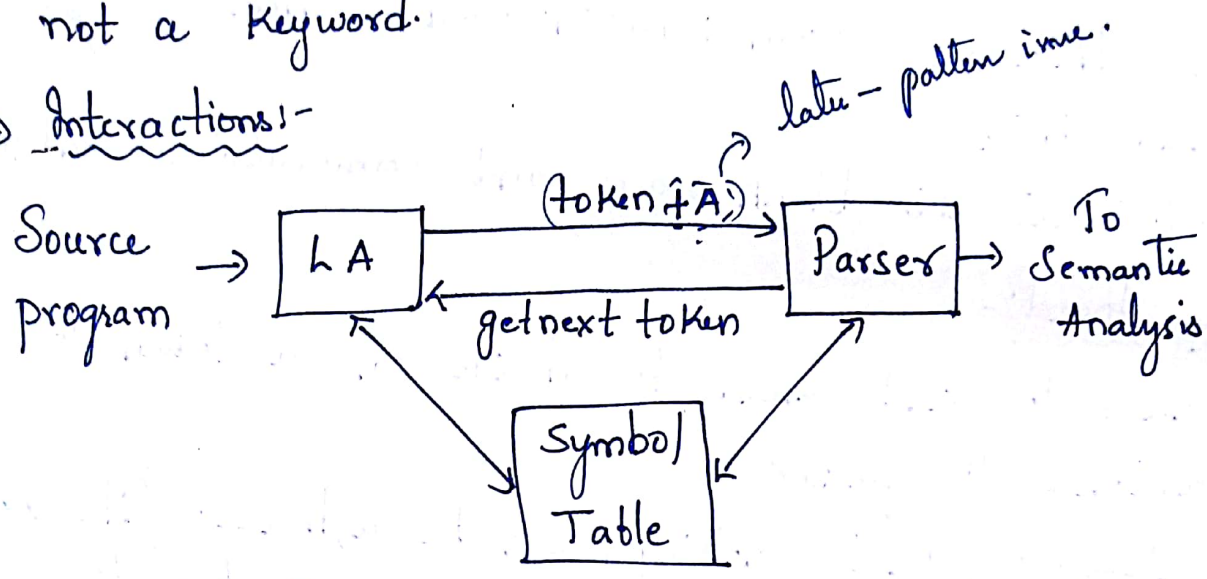
- LA converts the numerical constants to corresponding lexemes.
- e.g: 31 + 28 + 59 ⇒ ⟨num, 31⟩ ⟨+⟩ ⟨num, 28⟩ ⟨+⟩ ⟨num, 59⟩
  ↳ to enter symbol table.

Scanned by CamScanner

→ **Keywords:- & Identifiers:-**

- Most Compilers use fixed strings for Keywords.
- Character strings form an identifier, if it is not a Keyword.

→ **Interactions:-**



→ It Copys the Source program with error messages inserted at appropriate positions and returns back.

→ **Definitions:-**

1) **Lexeme:-** Sequence of characters in Source program that matches the pattern for a token.

2) **Token:-** pair of name & value.

3) **Pattern:-** Description of the form that the lexeme may take.

4) **Literal:-** Anything that is enclosed in Quotes (double).

- e.g: printf ("Total = %d \n", score);
  - printf → lexeme
  - "Total = %d \n" → literal
  - score → lexeme.

- both lexemes matches a Same pattern for token form : <id, 1>, <id, 2> etc).

Scanned by CamScanner

→ Attributes - for tokens:-

- Issue arouses when multiple lexemes, matches a Same pattern
- LA provides the parser not only a token, but also an 'attribute' that describes the lexeme (or) Token

→ Lexical errors:-

- e.g: fi (a == f(x)); ....
- LA, on its own cannot declare, that it is 'misspelled if', Could be identifier.
- LA Communicates with the parser to deal Such issues.

→ Error Recovery Techniques:-

1) 'Panic mode Recovery':-
   - delete Successive charecters from remaining input, until a proper lexeme is formed.
2) Delete one charecter from remaining input.
3) Insert a missing charecter into the input
4) Replace a charecter with another.
5) Transpose two adjacent charecters.

⇒ Regular grammar:-

1) Alphabet:- finite set of symbols.
   e.g: {0, 1} : Binary alphabet.
2) String:- Sequence of Symbols drawn from alphabet.
   - Synonyms: Sentences; words.

Scanned by CamScanner

- length denoted by $|s|$
- empty string : ϵ

3) <u>Language</u>:- Countable Set of strings over fixed alphabets.

→ <u>Operations</u>:-
- Unions.
- Concatenations.
- Closures.

→ <u>Regular expression</u>:-
- for a regular expression 'r', there is a language $L(r)$.
- 'a' is a symbol in alphabet, 'Σ' $\in L(r)$, then a is a regular expression and $L(a) = \{a\}$.

→ <u>Regular definition</u>:-
- if Σ is an alphabet, then Regular definition will

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots \qquad \vdots$$
$$d_n \rightarrow r_n$$

where,

$d_i \rightarrow$ a new symbol not in Σ

$r_i \rightarrow$ a new regular expression over the alphabet $\Sigma \cup \{d_1, d_2 \ldots d_{i-1}\}$

Scanned by CamScanner

→ Extensions for regular expressions:-

1) One or more instances:-

- unary, postfix operator '+'.
- $(r)^+$ denotes $(L(r))^+$
- '+': Same precedence and associativity as '*'.

2) Zero or one instance:-

- unary, postfix operator '?'
- $r? = r \mid \epsilon$

3) Charecter classes:-

- Short hand notation.
- [abc] for a|b|c
- [a-z] for a|b|...|z
- [A-Z] for A|B|...|Z

⇒ Input Buffering:-

1.) Buffer pairs:-

- Two Same Size : 4096 bytes.

- Two Pointers:-

(a) lexeme begin: marks beginning of current lexeme.

(b) forward:- Scans ahead until a pattern is matched.

→ forward is advanced only after completing first buffer.

Scanned by CamScanner

2) Sentinels:-

- Special symbol not a part of program.
- natural choice for eof.
- used in 'Buffer-end' test, when a buffer is loaded.

=) Token - Recognition:-

1) Transition diagrams:-

- patterns (lexemes) → Stylized flow charts (T.D)

↳ T.D:-

- nodes:- States.
- Edges:- labeled by a symbol.
- If we are in some state 's', and next input symbol is 'a', we look for an edge out of s labeled by 'a', and advances the 'forward', and enters that state.
- All T.D's are assumed to be 'deterministic'.
- 'forward' can be back-tracked only once, and only by specifying the state with *.
- The T.D always begins from the 'Start' - state, before any input.
- All actions are to be attatched to the 'final' (or) 'accepting state', denoted by double circles

Scanned by CamScanner

→ Recognition of reserved Keywords - Identifiers:-

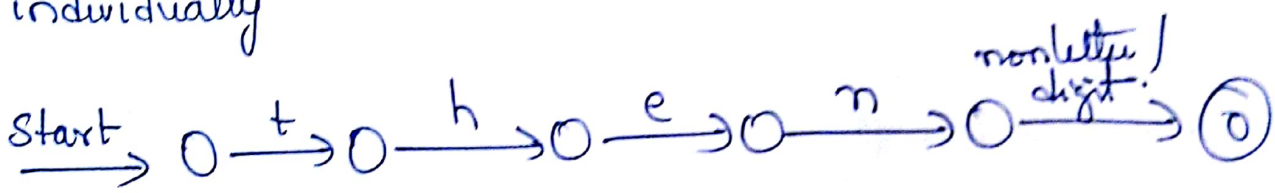
- Issue:- The T.D's for identifying identifiers, also identifies Keywords.

- Solutions:-

1) Install the reserved words in the Symbol table. A field of S.T is reserved for description. Any identifier not in the S.T, during L.A phase is only treated as an identifier.

2) Create Seperate transition diagrams for each Keyword.

- e.g: a T.D to allow Key word then is tested individually



→ Architecture of T.D Based L.A:-

1) As Specified, use individual T.D's for all Seperate Keywords.

2) Run in 'parallel' various T.D s. feed next input to all T.Ds. only one TD forms a lexeme.

strategy:- Take longest input of prefix.

3) Combine all T.Ds to one. We allow TDs to read input, until there is no possible next state. and take the longest lexeme that matched any pattern.

- All the deciding states are Converted to one 'start) state.

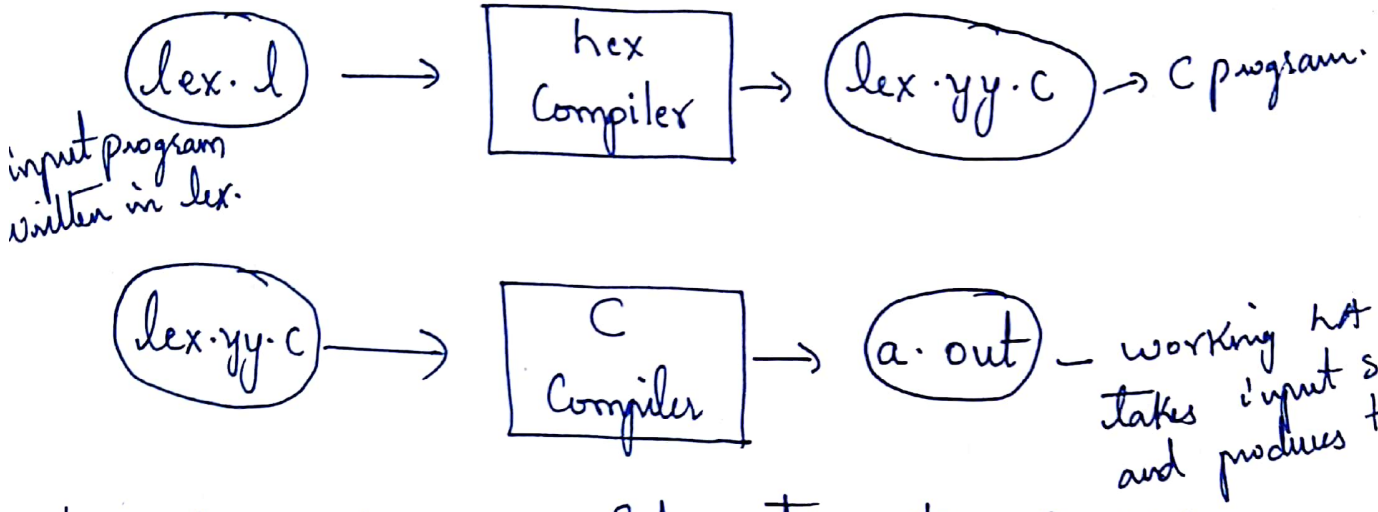
Scanned by CamScanner

→ **LEX:- Lexical Analyzer Generator:-**

- TOOL: Lex / Flex.
- LA Specified by specifying Regular expressions.
- Input:- Lex language.

- Lex Compiler transforms the input to TDs and generates code in a file lex.yy.c.

↳ Overview:-



⇒ 'a.out:- acts as a Sub-routine to parser.

→ It is a 'C' function, that returns an integer, which is a possible code for one of the tokens.

→ All the attributes are placed in another global variable called "yylval", which can be accessible by both LA & Parser.

→ Program - Structure:-

Declarations
%%
translation rules.
%%
auxillary functions

Scanned by CamScanner

1) Declaration Section:-

- Variables; manifest constants, regular definitions.

2) Translation Section:-

- Rule: Pattern { Action } 'c' code.

Reg. exprn :- That uses Reg. defn of declaration Section.

3) Auxillary function Section:-

- Additional functions Specified for Action.
- They are Compiled Seperately and loaded with LA.

→ Working:-

① - parser Calls LA for input

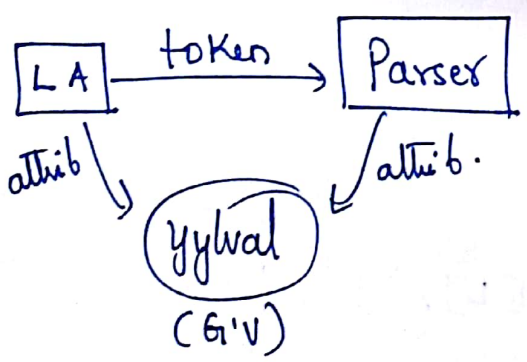
② - LA begins reading, until it finds longest prefix of input that matches one of the patterns P_i.

③ - It then executes its associated action A_i and returns to parser.

④ - If not, then the LA proceeds to find additional lexemes, until step ② is occurred.

- unlike Reg. LA, it passes only token to the Parser, but updates the information regarding that token in 'yylval', iff needed.

→



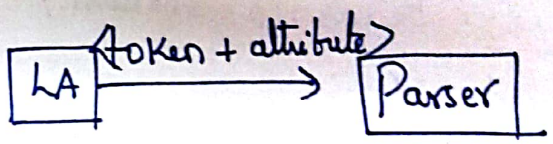


Fig: Regular LA.

Scanned by CamScanner

→ Simple lex Program:-

```
%{
    #include <stdio.h>
    int vowels=0;
    int cons = 0;
%}
%%
[aeiou AEIOU] {vowels ++;}
[a-z A-Z] {cons ++;}
%%
int yywrap()
{
    return 1;
}
main()
{
    printf(" Enter the string at end press ^d \n");
    yylex(); }
    printf(" No.of Vowels =%d \n No.of Consonants
            =%d \n, vowels, cons);
}
```

Declaration

Rule Section

Called when input is exhausted.
↳ Done if 1
↳ More processing if 0.

Main - entry point for lex.

⇒ Predefined functions:-

1) yylex:- Reads i/p stream, generate tokens, returns zero at end.

2) yyin:- points to an input file which is to be Scanned.

Syntax:- yyin = fopen(argv[1], "r");

Scanned by CamScanner

3) yyout:- points to a file where the o/p has to be kept.

Syntax: yyout = fopen (argv[2], "w");

4) yyleng:- lexeme's length

5) yylval:- token's value. (global variable)

6) yyval:- local variable

7) yytext:- input buffer.

8) yymore:- returns next token.

9) yyparse:- parses the lexeme.

=> Conflict Resolution:-

- when several prefixes of input matches more patterns,

1) Always prefer a longer prefix to shorter.

2) yet a longer prefix matches more patterns, then prefer the pattern listed first in lex program.

=> Look Ahead Operator:-

- lex automatically reads one character ahead of the last that forms the lexeme.

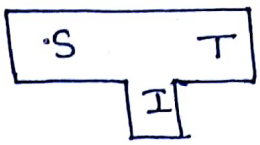
- slash (/) in a pattern indicates the end of the part of the pattern that matches the lexeme.

- The additional pattern is followed, that must be matched, before we can decide on a token.

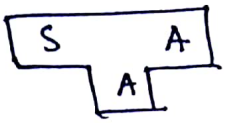
Scanned by CamScanner

⇒ Bootstrapping:- (High-level reading)

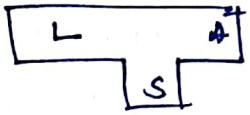
- A Compiler has:-
  - Source language (S)
  - Target language (T)
  - Implementation language (I)



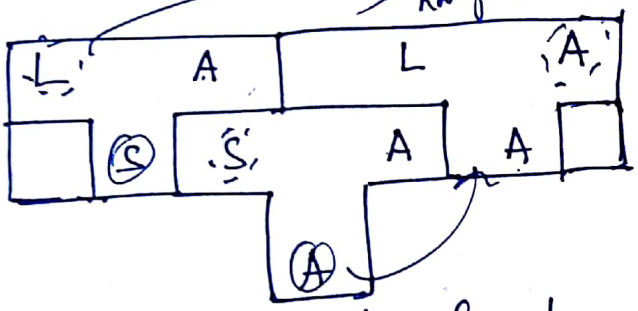
- Notation:- $S\,C\,{}^{T}_{I}$
- To Create a New language 'L', for Machine 'A'. - 'A' Create $S\,C\,{}^{A}_{A}$ (subset) of S using language A, which runs on Machine A.



- Create $L\,C\,{}^{A}_{S}$, a subset of L, for L.



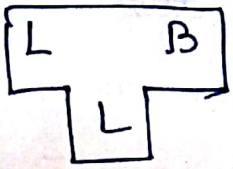
- Compile both, to obtain for language L, runs on machine A.



$L_S A + S_A A = L_A A$

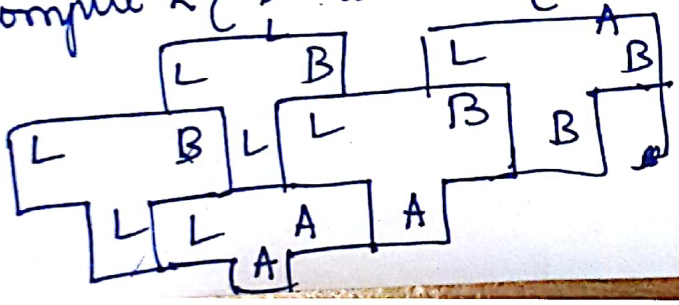
- To produce Compiler for ~~language~~ Machine B:-

- $S\,C\,{}^{A}_{A} \rightarrow L\,C\,{}^{A}_{S} \rightarrow L\,C\,{}^{B}_{L}$



('S' subset of 'L')

- Compile $L\,C\,{}^{B}_{L}$ to produce $L\,C\,{}^{B}_{A}$, runs on 'A'. (cross-compiler) which runs on A produces B.
- Compile $L\,C\,{}^{B}_{L}$ with $L\,C\,{}^{B}_{A}$ to produce $L\,C\,{}^{B}_{B}$.



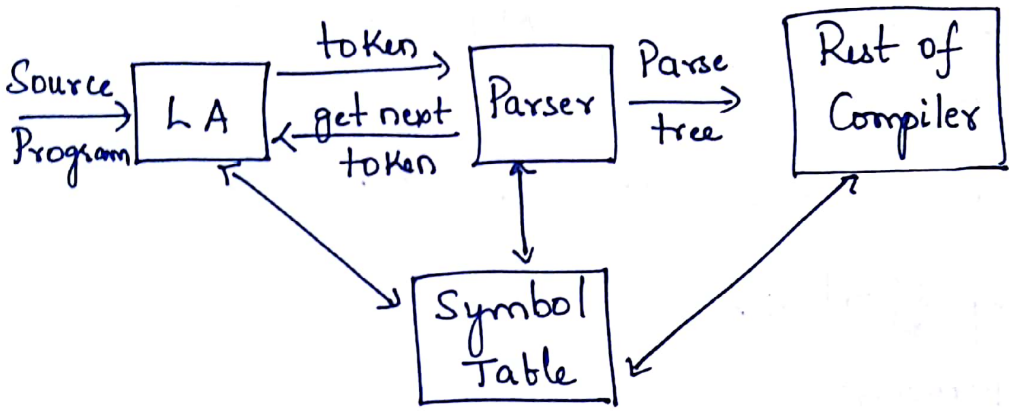
Scanned by CamScanner

SYNTAX ANALYSIS

→ **Role of Parser:-**

- obtains a string of tokens from LA and verifies that they can be generated by the Source language's grammar.
- It reports syntax errors in an intelligible fashion.
- It constructs a parse tree and passes it to the rest of Compiler.

→ **Interaction:-**



→ **Types:-**

- Universal
- Top-Down
- Bottom-Up

→ **Methods:-**

- Top-Down
- Bottom-Up

→ Works mostly for sub-classes of Grammars.

→ They are usually constructed using automated tools

Scanned by CamScanner

⇒ Syntax error Handling:-

→ Errors:-

1) Lexical:- misspellings. (fi)

2) Syntactic:- misplaces. ({ instead of })

3) Semantic:- mismatches. (operand – operators).

4) Logical:- misuse. (= instead of ==).

→ Error - Handler goals:-

1) Report the errors clearly and accurately.

2) Recover from each error quickly and detect subsequent ones.

3) Add minimal overhead to the processing.

→ Error- Recovery Strategies:-

1) Panic-mode-Recovery:-

- Discard input symbols one at a time, until a set of tokens is found.

- Pro:- Simple ; Cons:- Skips input

2) Phrase-level-Recovery:-

- To perform local correction i.e., to replace prefix of the remaining input.

- Replacement must be careful in checking infinite loops

- Pro:- It can correct any input string

- Cons:- Difficulty with copying of error situation.

Scanned by CamScanner

CONTEXT FREE GRAMMARS

- A CFG has four major Components:-

1) Terminals:-
- Also 'TOKens'.
- elementary symbols of input strings.

2) Non Terminals:-
- Also 'Syntactic Variables'.
- Denotes set of strings.
- They impose a structure on the language that is key to syntax analysis and translation.

3) Start Symbol:-
- one non-terminal.
- It denotes the set of strings that is the language generated by grammar.

4) Productions:-
- Specifies the manner in which terminals and non-terminals combine to form strings.
- Each production has:-

(a) Non-terminal:- Head (or) L.H.S.

(b) Symbol:- $\rightarrow$. (or) $::=$

(c) Terminals:- Body (or) R.H.S.
- also contains non-terminals.
- It describes the way in which strings at the head can be constructed. (Simple replacements).

Scanned by CamScanner

e.g. $S \rightarrow AA$

$A \rightarrow \alpha$

$A \rightarrow \beta$

$S \rightarrow \beta A$ $\quad$ $S \rightarrow \alpha A$

$S \rightarrow \beta \alpha$ $\quad$ $S \rightarrow \alpha \beta$

→ **Notations:-**

1) **Terminals:-**

- lower case letters:- a, b, c
- operator symbols:- +, *
- digits:- 0 - 9.
- Boldface strings

2) **Non Terminals:-**

- Uppercase letters: - A, B, C (early)
- lower case italic names; expr

3) **Symbols:-**

- Late uppercase letters: - X, Y, Z
- represents grammar symbols either terminals (or) Non terminals.

4) **String of Terminals:-**

- Late lowercase letters:- x, y, u, v, ... z.

5) **String of Grammar:-**

- lowercase Greek:- α, β, γ.

6) **Alternatives:-**

$A \rightarrow \alpha_1$; $A \rightarrow \alpha_2$ ... $A \rightarrow \alpha_k$, with Common head : A

Scanned by CamScanner

e.g:- id + - * / ( )

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

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

$F \rightarrow (E) \mid id.$

→ Derivations:-

- Parse tree Construction becomes easier in a 'derivational' point of view.
- Consider $\alpha A \beta$, A being non-terminal and $A \Rightarrow \gamma$, then

$$\alpha A \beta \Rightarrow \alpha \gamma \beta.$$

→ If $\alpha \Rightarrow \beta$; $\beta \Rightarrow \gamma$ then $\alpha \Rightarrow \gamma$.

→ Notation:-

(a) $\Rightarrow$: Derives in one-step

(b) $\overset{+}{\Rightarrow}$: Derives in one or more steps.

(c) $\overset{*}{\Rightarrow}$: Derives in zero or more steps.

→ Sentential form:-

$S \overset{*}{\Rightarrow} \alpha$, 'S' : start symbol; 'G' : Grammar

$\therefore \alpha$ is a Sentential form of G.

→ Context-free language:-

- A string of terminals (w), is in L(G), (language generated by G) and w is a Sentence of G ($S \overset{*}{\Rightarrow} w$).,

Such language is called 'Context-free' language.

Scanned by CamScanner

→ **Left most derivation:-**

- The left most non terminal in each Sentential is always choosen.
- $\alpha \Rightarrow \beta$, $\alpha_{lm} \Rightarrow \beta$, Step in which the left most non-terminal in α is replaced.

→ **Right most derivation:-** (Canonical derivations).

→ $\alpha = \beta$, $\alpha_{rm} \Rightarrow \beta$, right most non-terminal in α is choosen for derivation.

→ **Left-Sentential form:-**

$\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$; if $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, α is a L.S. form of Grammar.

→ **Parse Tree:-** Derivation that filters out order in which productions are applied.

- Many-one relation b/w Derivations & P.T.

→ **Ambiguity:-** Grammar that produces more than one P.T. (or) more than one L.M.D (or) R.M.D.

→ A Grammar G generates a language 'L', every string generated by G is in L.

→ **CFG vs RE:-**

- every construct can be described by a grammar
- every language is a CFL.

Scanned by CamScanner

PARSING

⇒ **Top Down:-**

- Constructing a PT from root → nodes.
- finding leftmost Derivation.
- At each step, the problem is to determine the production to be applied on a non terminal.
- once production is choosen, the process is matching the terminals with the input strings.

- **Grammar:-**

$E \rightarrow TE'$

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

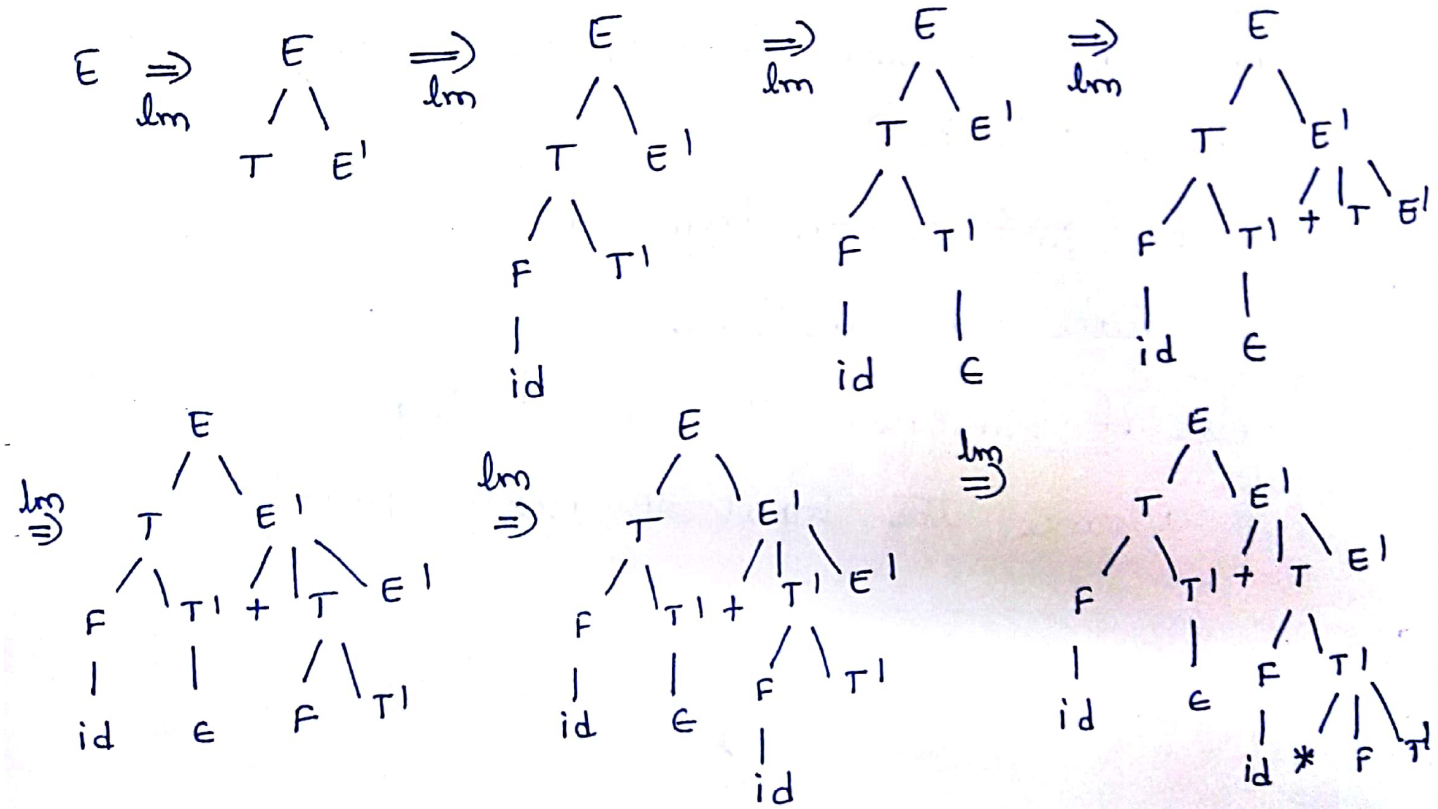
$T \rightarrow FT'$

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

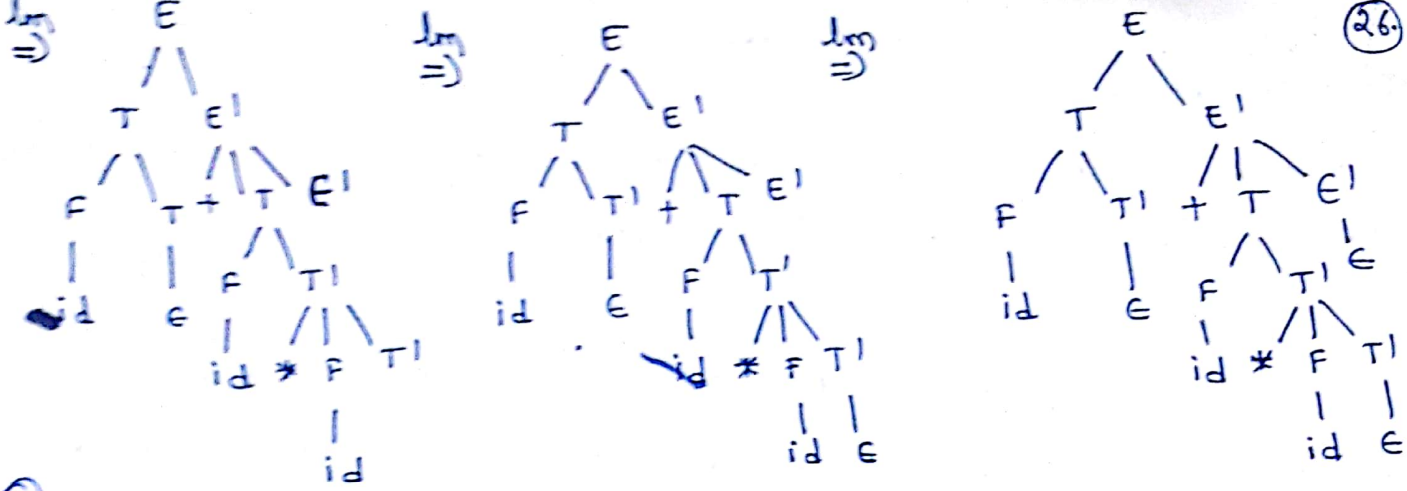
$F \rightarrow (E) \mid id.$

Input:- id + id * id

- **Parse Tree:-**



Scanned by CamScanner



② ⇒ **Recursive - Descent Parsing:-**

- Consists of set of procedures, one for each non terminal.
- Execution begins with start and announces Success, if it scans the entire string.
- Back-tracking allowed but prompts for code modification.
- **Algorithm:-**

```
{
Choose a Production: A → X1, X2 - - . XK;
 for (i=1 to K){
    if (Xi is a non-terminal)
      Call Procedure Xi();
 else if (Xi equals the current input a)
    advance the input to next symbol;
 }
}
```

Scanned by CamScanner

(1). BRUTE FORCE PARSING

-> full Backup methodology (Back tracking)

-> Best case:- Attempt to Create a Syntax tree, until a Correct Set of terminals is reached.

-> worst case:- All possible Combinations are attempted, before recognising the failure.

-> Algorithm:-

— begin —

Step 1: Given:- A Non-terminal to be expanded
- Apply first production.

Step 2: Given:- New string; Select L.S.C and attempt for expansion.

step 3: Repeat step(2), until the process Cannot Proceed further.

step 4: If {The expansion is Correct, string is parsed Successfully}

else

{The expansion is incorrect, undo most recent production, apply next production}.

Step 5: If no further productions are available to replace, then go for further back tracking, until productions are available. Print "unparsable".

if we have reached non-terminal,

— End —

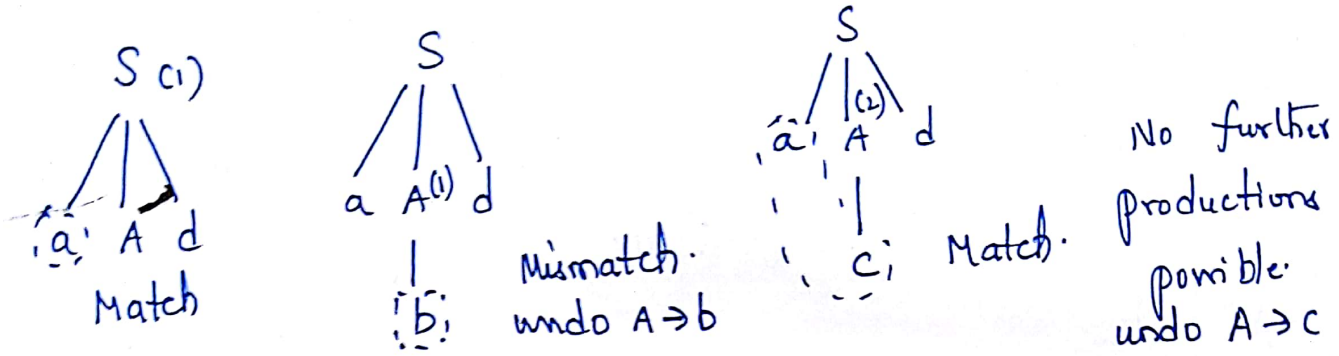
Scanned by CamScanner

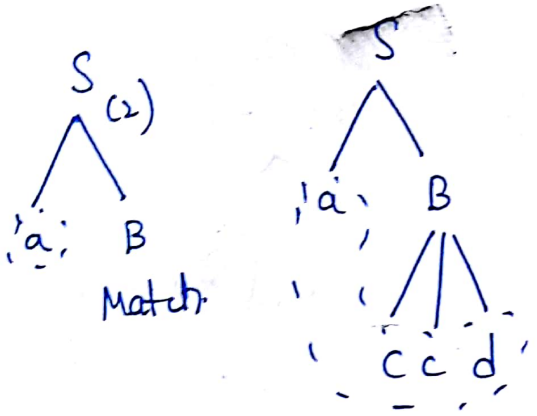
E.g. 1: $S \to a\overset{①}{A}d \,/\, \overset{②}{aB}$

$A \to \overset{①}{b} / \overset{②}{c}$

$B \to \overset{①}{cd} / \overset{②}{ddc}$

String:- accd.



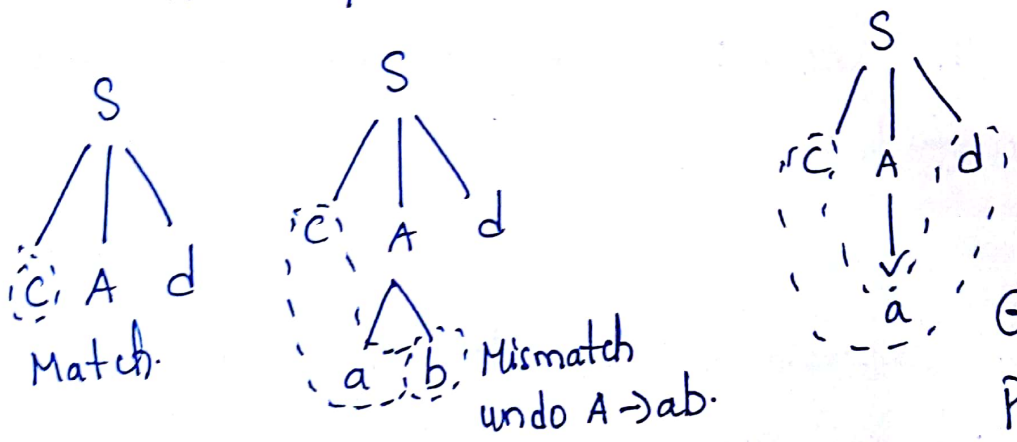


Given string is parsed.

E.g. 2:- $S \to cAd$

$A \to ab / a$

String:- cad



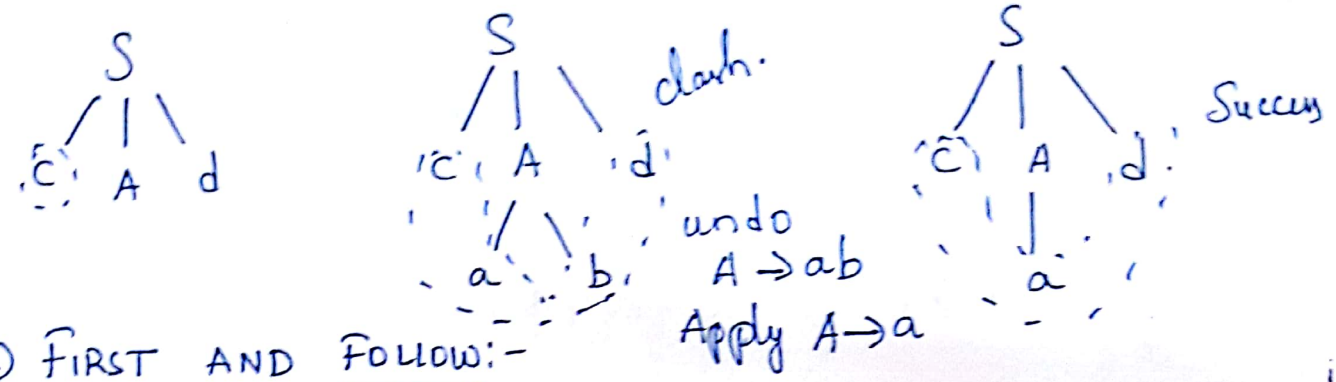
Scanned by CamScanner

e.g.: Grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

string : 'cad'



⇒ **FIRST AND FOLLOW:-**

→ **Rules for FIRST:-**

1) For a Terminal 'a' ∴ FIRST(a) = {a}.

2) For a Rule: $X \rightarrow \epsilon$ ∴ FIRST(X) = {ε}.

3)* For a Rule: $A \rightarrow X_1 X_2 X_3 \ldots X_K$

FIRST(A) = {FIRST(X_1) ∪ FIRST(X_2) . . . }

→ **Methodology:-**

1) FIRST(E) = FIRST(T) = FIRST(F) = {c, id}.

2) FIRST(E') = {+, ε}

3) FIRST(T) = {c, id}.

4) FIRST(T') = {*, ε}

5) FIRST(F) = {(, id}.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$.

→ **Rules for FOLLOW:-**

1) for Start 'S', FOLLOW(S) = $

2) For Production: $A \rightarrow \alpha B \beta$, then.

FIRST(β) without ε is placed in FOLLOW(B)

∴ FOLLOW(B) = {FIRST(β) − ε}

Scanned by CamScanner

3) for productions: $A \to \alpha B \beta$ or $A \to \alpha B$, $FIRST(\beta) = \{\varepsilon\}$ (28)

$FOLLOW(A) = FOLLOW(B)$ $\quad \therefore FOLLOW(B) = \{FIRST(\beta) - \varepsilon\}$

$= \{\varepsilon - \varepsilon\}$

$= \phi$.

→ <u>Methodology</u>:-

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

2) $FOLLOW(E') = FOLLOW(E) = \{), \$\}$ (∵ Rule (3))

3) $FOLLOW(T) = \{FIRST(E') - \varepsilon\} = \{+,), \$\}$ (∵ $ replaced by ϵ).

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

5) $FOLLOW(F) = FIRST(T') \Rightarrow$

Consider $T \to FT'$, $T' \to *FT'$

<u>Case (i)</u>: $T \to FT'$

$A \to \alpha B \beta$

$A : T$; $\alpha : \epsilon$, $B : F$; $\beta : T'$

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

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

$\therefore FOLLOW(F) = \{\{*, \epsilon\} - \{\epsilon\}\}$

$= \{*\}$

<u>Case (ii)</u>:- $T' \to *FT'$

$A \to \alpha B \beta$

$A : T'$; $\alpha = *$; $B : F$; $\beta : T'$

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

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

$= \{*\}$.

→ Consider $T' \to *FT'$

$A \to \alpha B \beta$

$FOLLOW(A) = FOLLOW(B)$

$FOLLOW(T') = FOLLOW(F)$.

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

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

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

Scanned by CamScanner

1) FOLLOW(E):-

$F \to (E) / id$

$A \to \alpha B \beta$

$A : F$; $\alpha : ($, $B : E$, $\beta =)$

$FOLLOW(B) = (FIRST(\beta) - \epsilon)$

$= ( ))$

$\therefore FOLLOW(E) = \{), \$ \}$ ($\because$ E : Start symbol add \$).

2) FOLLOW(E'):-

$E' \to + T E' / \epsilon$; $E \to TE'$

(i) $E \to TE' \Rightarrow TE'\epsilon$

$A \to \alpha B \beta \Rightarrow \alpha B \beta$

$A : E$; $\alpha : T$; $B = E'$; $\beta = \epsilon$

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

3) FOLLOW(T):-

$E \to TE'$; $E' \to + TE'$

i) $E \to TE' \Rightarrow \alpha TE'$

$A \to \alpha B \beta \Rightarrow \alpha B \beta$

$A : E$; $\alpha = \epsilon$; $B = T$; $\beta = E'$

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

$\therefore FOLLOW(T) = \{ FIRST(E') - \epsilon \}$

$= \{ (+, \epsilon \} - \epsilon \}$

$= \{ + \}$.

ii) $E' \to + TE'$.

$A \to \alpha B \beta$

$A : E'$, $\alpha = +$, $B = T$, $\beta = E'$

$FOLLOW(A) = FOLLOW(B)$

$\therefore FOLLOW(E') = FOLLOW(T)$

$= \{), \$ \}$

$\therefore FOLLOW(T) = \{ + \} \cup \{), \$ \}$

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

Scanned by CamScanner

4) FOLLOW (T'):-

$T \to FT'$; $T \to *FT'$

i) $T \to FT'\epsilon$

$A \to \alpha B \beta$

$A : T, \alpha = F ; B = T ; \beta = \epsilon$

$\therefore$ FOLLOW(T') = FOLLOW(T)

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

ii) $T \to *FT'$

$A \to \alpha B \beta$

$A : T, \alpha = *, B = F, \beta = T'$.

FOLLOW (T') = FOLLOW (T)

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

$\therefore \{+,), \$\}$

5) FOLLOW (F):-

$T \to FT'$, $T' \to *FT'$.

Scanned by CamScanner

=> FIRST & FOLLOW:- FIRST: F( ); FOLLOW: f( ) 30-(A)

→ Example:-

1) $S \to ABCDE$
$A \to a / \epsilon$
$B \to b / \epsilon$
$C \to c$
$D \to d / \epsilon$
$E \to e / \epsilon$

FIRST

$F(S) = F(A) = \{a, b\}$ (a if) $F(A), F(B)$ c
$F(A) = \{a, \epsilon\}$
$F(B) = \{b, \epsilon\}$
$F(C) = \{c\}$
$F(D) = \{d, \epsilon\}$
$F(E) = \{e, \epsilon\}$

FOLLOW

$f(S) = \{\$\}$
$f(A) = \{b, c\}$
$f(B) = \{c\}$
$f(C) = \{d, e, \$\}$
$f(D) = \{e, \$\}$
$f(E) = \{\$\}$.

e.g. first: $\{b, \epsilon\}$
$S \to Ab / S\$ / \epsilon$

2) $S \to Bb / Cd$
$B \to aB / \epsilon$
$C \to cC / \epsilon$

FIRST

$F(S) = \{a, b, c, d\}$ ϵ
$F(B) = \{a, \epsilon\}$
$F(C) = \{c, \epsilon\}$

FOLLOW

$f(S) = \{\$\}$
$f(B) = \{b\}$
$f(C) = \{d\}$

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

FIRST

$F(E) = \{id, (\}$
$F(E') = \{+, \epsilon\}$
$F(T) = \{id, (\}$
$F(T') = \{*, \epsilon\}$
$F(F) = \{id, (\}$

FOLLOW

$f(E) = \{\$,)\}$ → default symbol. → following E.
$f(E') = \{\$,)\}$ (∵ follow(E))
$f(T) = \{F(E') = \{+, \$,)\}$ (first(E')) + follow(E))
$f(T') = \{f(T) = \{+, \$,)\}$ (follow(T))
$f(F) = \{F(T') = \{*, +, \$,)\}$ (first T') + follow(T).

4) $S \to AcB / cbB / Ba$
$A \to da / BC$
$B \to g / \epsilon$
$C \to h / \epsilon$

FIRST

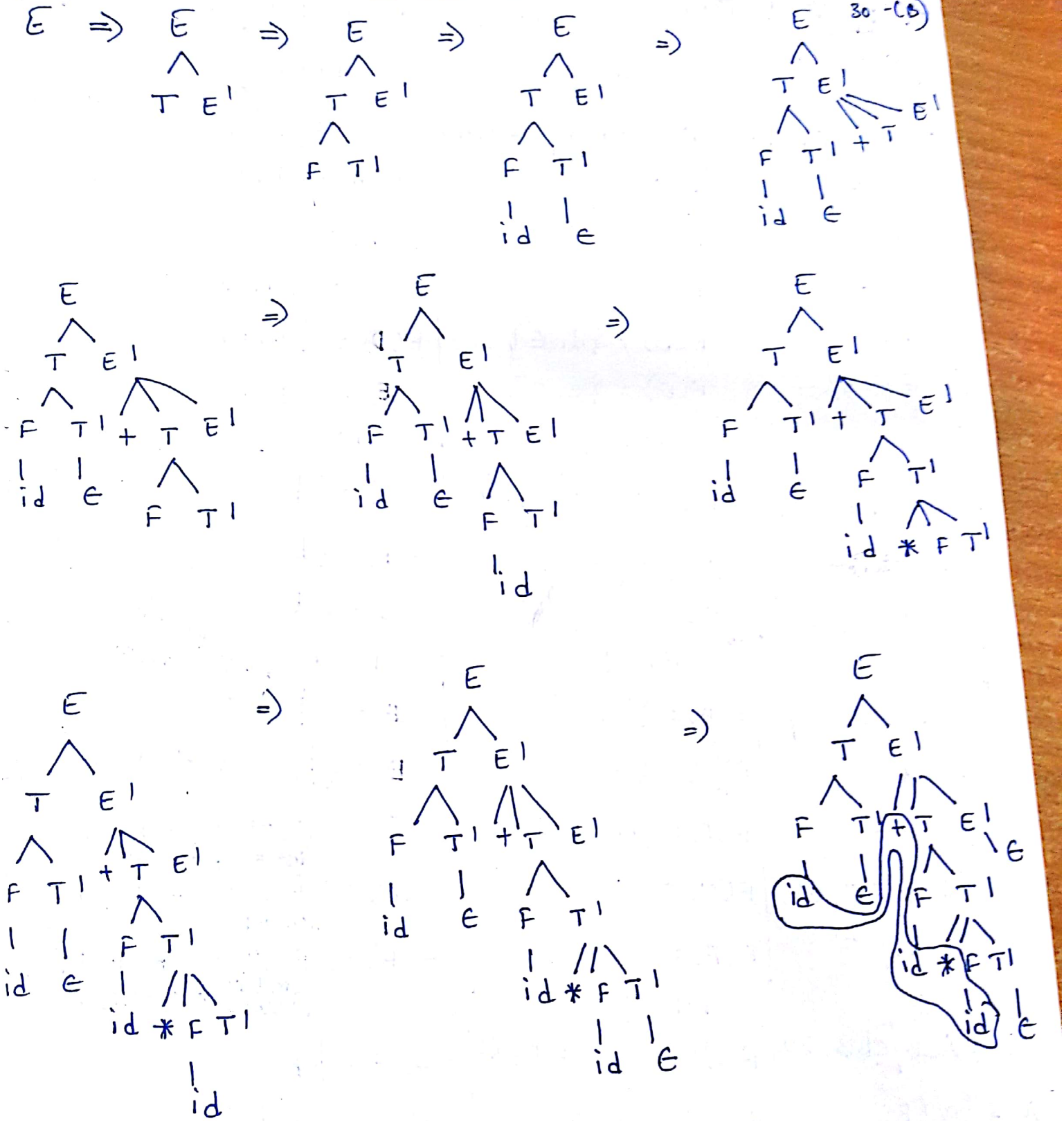
$F(S) = \{d, g, h, \epsilon, b, a\}$
$F(A) = \{d, g, h, \epsilon\}$
$F(B) = \{g, \epsilon\}$
$F(C) = \{h, \epsilon\}$

FOLLOW

$f(S) = \{\$\}$
$f(A) = \{h, g, \$\}$
$f(B) = \{\$, a, h, g\}$
$f(C) = \{g, \$, b, h\}$

Scanned by CamScanner

30 -(B)



String: id + id * id.

Scanned by CamScanner

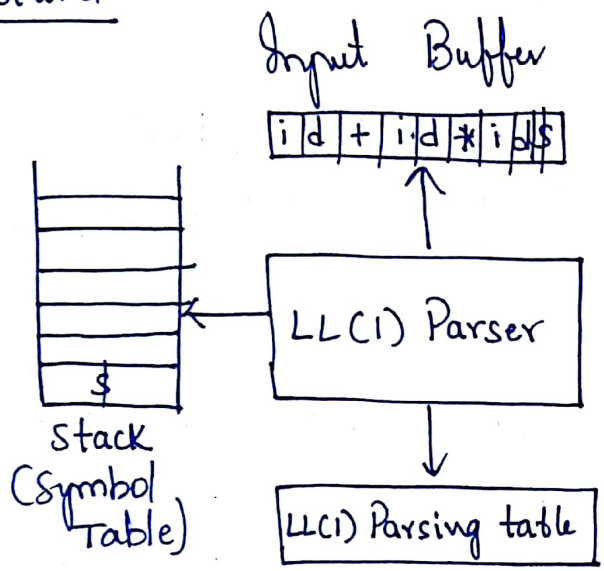
⇒ LL(1) Parsers:- PREDICTIVE PARSER - Table Driven Parser

- L: Left-right Scanning
- L: Takes left most derivation
- 1: Takes / Scans one input character at a time. (Look-Ahead factor)

→ **Not allowed:-**

- Ambiguous grammars.
- Non-Deterministic Grammars.
- Should't be left recursive.

→ **Outline:-**



⇒ **Table Construction:-**

- For a given grammar 'G', Compute FIRST() & FOLLOW()
- place all L.H.S (non-terminals) as 'Row Labels'.
- place all R.H.S (terminals) as 'column labels'.

⇒ **Grammar:-**

Grammar	FIRST	FOLLOW
$E \to TE'$	$F(E) = \{(, id\}$	$FOLLOW(E) = \{\$,)\}$
$E' \to +TE' / \epsilon$	$F(E') = \{+, \epsilon\}$	$FOLLOW(E') = \{FOLLOW(E)\} = \{\$,)\}$
$T \to FT'$	$F(T) = \{(, id\}$	$FOLLOW(T) = \{FIRST(E') + FOLLOW(E)\} = \{+, \$,)\}$
$T' \to *FT' / \epsilon$	$F(T') = \{*, \epsilon\}$	$FOLLOW(T') = \{FOLLOW(T)\} = \{+, \$,)\}$
$F \to (E) / id$	$F(F) = \{(, id\}$	$FOLLOW(F) = \{FIRST(T') + FOLLOW(T)\} = \{*, +, \$,)\}$.

Scanned by CamScanner

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

$E \to TE'$

1) $E \to TE'$ holds good in id & (

FIRST(TE') = {id, (}
= FIRST(T)

2)(i) $E' \to +TE'$ holds good for + (terminal).
= FIRST(+TE') = {+}

(ii) $E' \to \epsilon$ doesn't hold good in L.H.S. ⇒ follow(E') = {$,)}
($E' \to \epsilon$ ∴ FOLLOW(E')).

3) $T \to FT'$ = FIRST(FT')
= FIRST(F) = {id, (}

4)i) $T' \to *FT'$ = FIRST(T') = {*, ε} ; FOLLOW(T') ($T' \to \epsilon$).

ii) $T' \to \epsilon$ = FOLLOW(T') = {+,), $}

5) $F \to id \mid (E)$ → If ε comes in R.H.S, place it in follow of L.H.S.

⇒ <u>Working</u> :- <u>LL(1)</u> <u>Parser</u> :-

- Construct another table.
- Three Column fields :-
  - Stack
  - Input pattern
  - Action to be carried.

⇒ append $ to both Stack & Input.

→ Consider Input : (id + id * id) for above Grammar.

Scanned by CamScanner

⇒ Error Recovery in Predictive Parsing:-

1) Panic Mode:-

→ Add "Synch" entries into Parsing table, under Follow of Non Terminals.

→ Follow (E) : { $,) }

→ Follow (E') : { $,) }

→ Follow (T) : { + , $,) }

→ Follow (T') : { + , $,) }

→ Follow (F) : { * , + , $,) }

→ Modified Table:-

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

2) Phrase level:-

- Blank entries are filled with "error routines".
- Those routines can change, insert, delete, pop input from stack (S.T)
- Can't proceed further parsing. Not so ideal technique.

Scanned by CamScanner

→ String :-)id * + id

SYMBOL-TABLE	INPUT- BUFFER	SEMANTIC ACTION.
E$	)id * + id $	Synch: pop (Skip) ")" as 'E' is start one.
E$	id * + id $	E → TE'
TE'$	id * + id $	T → FT'
FT'E'$	id * + id $	F → id
idT'E'$	id * + id $	Match: id
T'E'$	* + id $	T' → *FT'
*FT'E'$	* + id $	Match: *
FT'E'$	+ id $	Synch, pop 'F'
T'E'$	+ id $	T' → ∈
E'$	+ id $	E' → +TE'
+TE'$	+ id $	Match: +
TE'$	id $	T → FT'
FT'E'$	id $	F → id
idT'E'$	id $	Match: id
T'E'$	$	T' → ∈
E'$	$	E' → ∈
$	$	Parser Halts

Rule 1:- If look-up is 'blank', pop 'Terminal'

Rule 2:- If look-up is 'Synch', pop 'Non-Terminal'

Rule 3: for synch, if pop = start (or) only N.T, then pop = Terminal. (To resume Parsing).

Scanned by CamScanner

Stack	Input	Action to be done
$E\$$	id + id * id $	$E \to TE'$
$TE'\$$	id + id * id $	$T \to FT'$
$FT'E'\$$	id + id * id $	$F \to id$
$id\,T'E'\$$	id + id * id $	id : accepted. (pop: id).
$T'E'\$$	id + id * id $	$T' \to \epsilon$ (pop: T')($T' \to +$)
$E'\$$	id + id * id $	$E' \to +TE'$ ($E' \to +$)
$+TE'\$$	id + id * id $	Match : + : accepted. (pop: +).
$TE'\$$	id + id * id $	$T \to FT'$ (T on id)
$FT'E'\$$	id + id * id $	$F \to id$
$id\,T'E'\$$	id + id * id $	id : accepted (pop: id)
$T'E'\$$	id + id * id $	$T' \to *FT'$ (T' on *)
$*FT'E'\$$	id + id * id $	Match : * : accepted (pop: *)
$FT'E'\$$	id + id * id $	$F \to id$
$id\,T'E'\$$	id + id * id $	id : accepted (pop: id).
$T'E'\$$	id + id * id $	$T' \to \epsilon$. (pop: T'). (T' on $)
$E'\$$	id + id * id $	$E' \to \epsilon$ (pop: E') (E' on $)
$\$$	id + id * id $	End. ($ on $: End)

pattern Matched: (id + id * id)

→ The tokens are entered from right to left in stack, to avoid 'Backtracking'.

Scanned by CamScanner

⇒ Bottom-up parsing:-

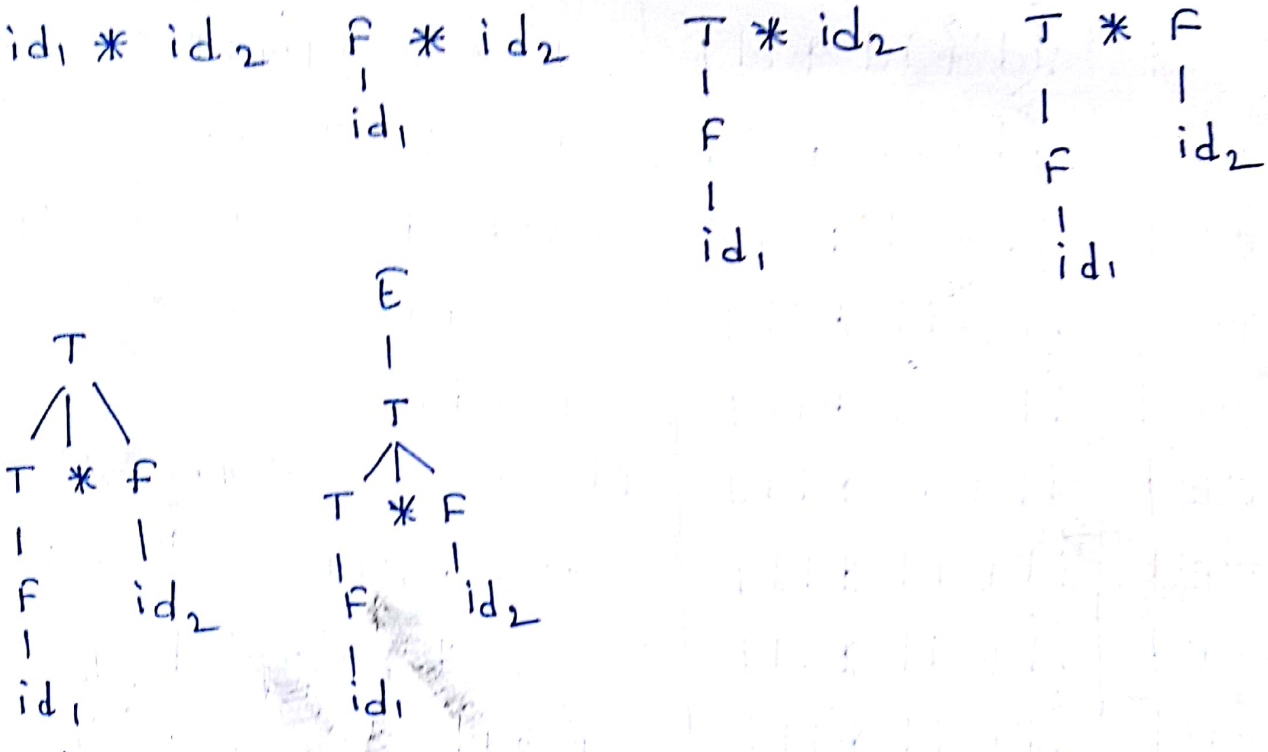
Grammar:- $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id.$

A 29 copies

→ Input:- $id_1 * id_2$



Reductions:-

- Bottomup parsing:- Reducing a string to the start.
- Reduction:- Reverse of a derivation, depicting only roots

→ Reductions:-

$id_1 * id_2$, $F * id_2$, $T * id_2$, $T * F$, T, E.

→ R.M.D:-

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id.$

Scanned by CamScanner

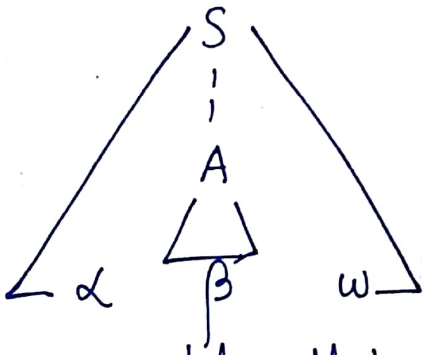
→ Handle Pruning:-

→ Handle:- A substring that matches the body of a production, and whose reduction represents one step along the reverse of R.M.D.

Sentential form	Handle	Reduction
$id_1 * id_2$	id_1	$F \to id_1$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

- Consider,

$$S \Rightarrow \alpha A w \Rightarrow \alpha \beta w.$$



- $A \to \beta$ is a 'handle' of $\alpha\beta w$, replacement should be done at Handle's R.H.S.

- for an unambiguous grammar, every right sentential form of a grammar has exactly one handle.

→ Handle Pruning:- Constructing reverse of a rightmost derivation.

Scanned by CamScanner

Consider,

$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w.$

- Start with 'w' (to be parsed)
- $w = \gamma_n$.
- Locate the handle β_n in γ_n and replace it ($A_n \to \beta_n$) to obtain γ_{n-1}.
- Locate the handle β_{n-1} in γ_{n-1} and reduce to obtain γ_{n-2}, until we get 's'.

⇒ **Shift-Reduce Parsing:-**

→ **Stack:-** Holds grammar symbols.

→ **Input Buffer:-** Holds rest of string to be parsed.

STACK (S.T)	INPUT	ACTION
$	id1 * id2 $	shift id → S.T
$ id1	* id2 $	reduce F → id
$ F	* id2 $	reduce T → F
$ T	* id2 $	shift
$ T *	id2 $	shift
$ T * id2	$	reduce F → id2
$ T * F	$	reduce T → T*F
$ T	$	reduce E → T
$ E	$	accept.

S.TART FIRST APPEARANCE

Scanned by CamScanner

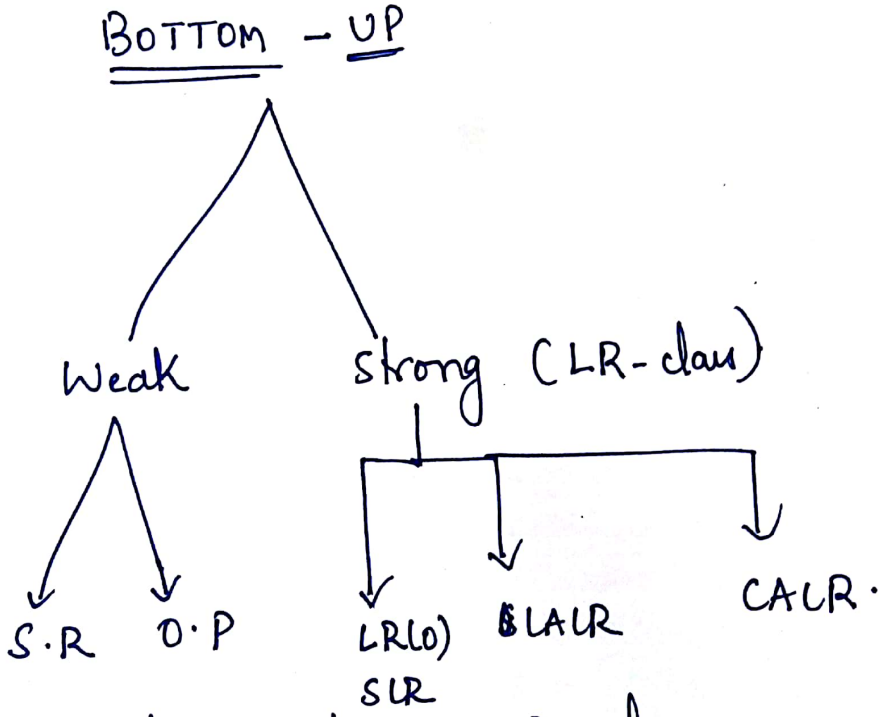
→ Actions:-

1) shift:- move the next input to stack.
2) Reduce:- Replace with possible reduction.
3) Accept:- Succesful Completion of parsing.
4) Error:- discovers Syntax error, and calls its routine

→ Conflicts:-

1) Not applicable for ambiguous grammars.
2) Not applicable when the Setting is unable to decide on a reduction.

~~Top~~ - ~~Down~~ Parsers



- Shift-Reduce, Concepts used in LR-class.

Scanned by CamScanner

⇒ ~~[illegible]~~ - LR Parsing **OP - PR**

- Conflicts :- S/R ; R/R

(38)

→ O-P - Parser :-

- Grammar :- $S \to iEts \mid iEtSeS \mid a$ → No two adjacent NT's
$E \to b \mid c \mid d$

→ No - ε - Productions
→ op - grammar (req).
→ Sentinel.

- Precedence :- $i \cdot\!> t \cdot\!> e$; $\$$: least ; a, b, c, d : highest
↳ direct terminals.

⇒ Relation is $\boxed{a \cdot\!> b \cdot\!> c \cdot\!> d \cdot\!> i \cdot\!> t \cdot\!> e \cdot\!> \$}$.

	i	t	e	a	b	c	d	$
i	≐	·>	·>	<·	<·	<·	<·	·>
t	<·	≐	·>	<·	<·	<·	<·	>
e	<·	<·	≐	<·	<·	<·	<·	·>
a	·>	·>	·>	≐	·>	·>	·>	·>
b	·>	·>	·>	·>	≐	·>	·>	·>
c	·>	·>	·>	·>	·>	≐	·>	·>
d	·>	·>	·>	·>	·>	·>	≐	·>
$	<·	<·	<·	<·	<·	<·	<·	≐

Pros
→ Very Simple
→ Can deal with ambig. gramm.

Cons
→ Can't decide on multiple handles
→ No guarantee for string though parsed is some of that gram.
↳ there the need for pr-relations.

Rules: <· = push / shift
·> = reduce

ravikishore4u.2010@gmail

Scanned by CamScanner

⇒ Input string:- ibtaea$

Stack	Input	Relation	Action
$	ibtaea$	<· (shift)	Push i [$, i]: <·
$i	btaea$	<·	Push b [i, b]: <·
$ib	taea$	·>	Reduce b [b, t]: ·>
$iE Clash	taea$	<·	Push t [$, t]: <·
$iEt	aea$	<·	Push a [t, a]: <·
$iEta	ea$	·>	Reduce a [a, e]: ·>
$iEtS Clash	ea$	<·	Push e [$, e]: <·
$iEtSe	a$	<·	Push a [e, a]: <·
$iEtSea	$	·>	Reduce a [a, $]: ·>
$iEtSeS	$	·>	Reduce: [$, $]: ·>
$S	$	-	ACCEPT

Rule:- 1) Result: <· push.

2) Result: ·> Reduce last element in stack and compare with $ from stack.

Scanned by CamScanner

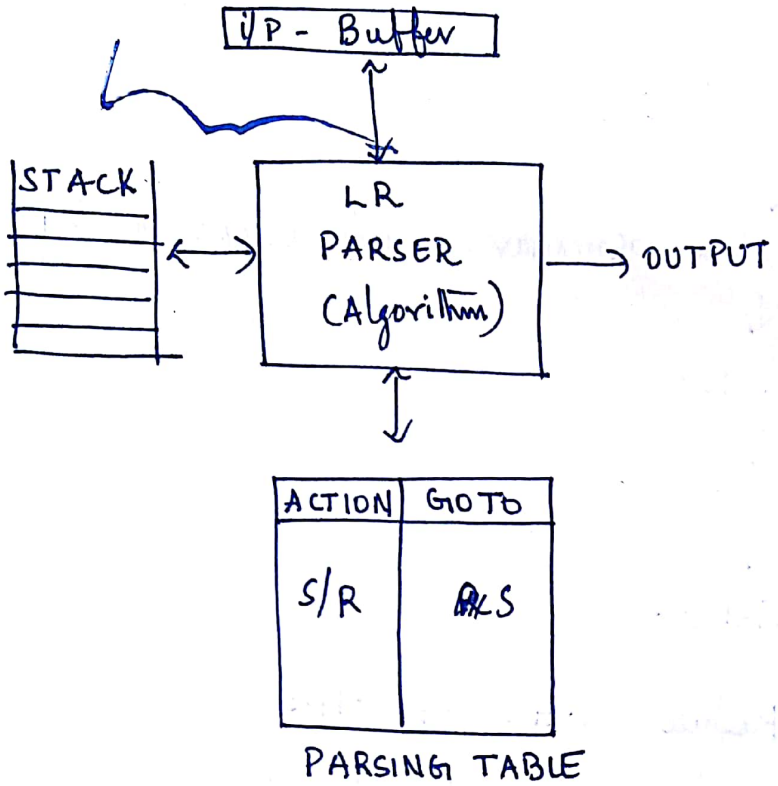
LR Parsers :-

→ L : L → R ; R : Rmd ; K : L.A. factor

→ LR grammars are Superset grammars.

→ Can recognise most languages in which CFG can be written.

→ **Model :-**



→ **Types :-**

- Simple LR (SLR)
- Look Ahead LR (LALR)
- Canonical LR (CALR)

- most general back-tracking
- It detects syntactic errors quickly.

→ **Drawbacks :-**

- Too much work to construct a table. - Need for a tool
- e.g. CFG → [YACC] → Parser.

Scanned by CamScanner

Items:-

$$\left.\begin{array}{l} A \to xyz \\ A \to \cdot xyz \\ A \to x \cdot yz \\ A \to xy \cdot z \\ A \to xyz \cdot \end{array}\right\} \text{LR(0) collection.}$$

Working:- LR (Parser).

LR(0) collection

↓

Augmented grammar (for accepting input)

↓

CLOSURE

↓

GOTO

↓

TABLE.

Reason:- When to Reduce R.H.S → L.H.S.

$xyz \to A$ only if $A \to \underline{xyz \cdot}$ (have seen xyz)

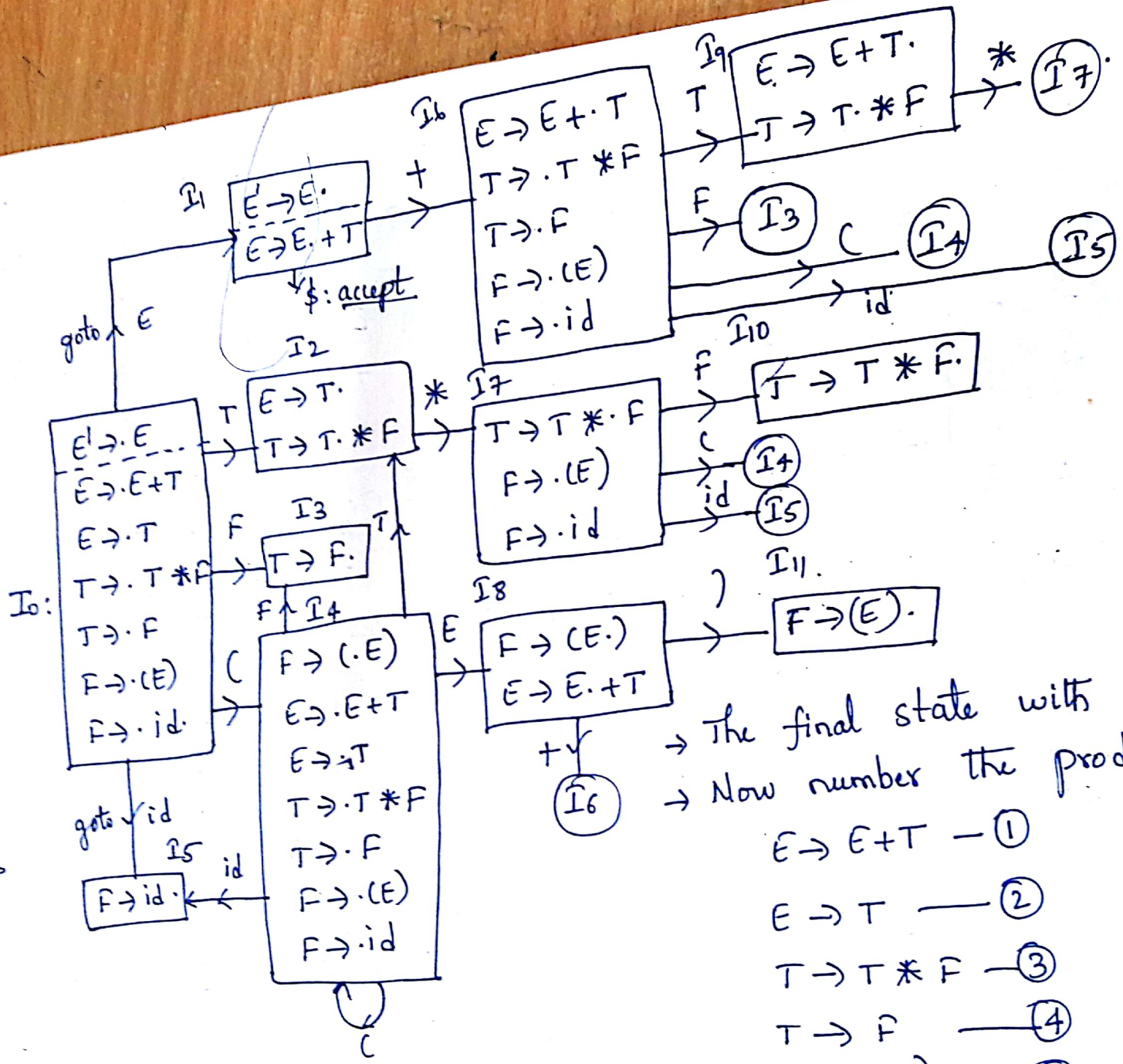
E.g:- Grammar:

$$\left.\begin{array}{l} E \to E+T \\ E \to T \\ T \to T*F \\ T \to F \\ F \to (E) \\ F \to id \end{array}\right\} G$$

Augmented Grammar : $\boxed{\begin{array}{c} E' \to E \\ G \end{array}}$

Scanned by CamScanner

Step 1:- Construct Augmented Grammar, nothing but Jammi.1 (4)



→ The final state with E': accept state

→ Now number the productions

E → E+T — (1)

E → T — (2)

T → T*F — (3)

T → F — (4)

F → (E) — (5)

F → id — (6)

Scanned by CamScanner

Step 2:- Now Compute **FOLLOW:**

1) FOLLOW(E') = {$} (∵ Start symbol)
2) FOLLOW (E) = {+,), $}
3) FOLLOW (T) = {+, *,), $}
4) FOLLOW (F) = {+, *,), $} (∵ FOLLOW(F) = FOLLOW(T))

Step 3:- **LR(1) PARSING TABLE.**

State	ACTION							GOTO		
	id	+	*	(	)	$		E	T	F
0	S5			S4				1	2	3
1		S6				accept				
2		r2	S7		r2	r2				
3		r4	r4		r4	r4				
4	S5			S4				8	2	3
5		r6	r6		r6	r6				
6	S5			S4					9	3
7	S5			S4						10
8		S6			S11					
9		r1	S7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

→ **Rules for ACTION:-**

0 – id : 5 ⇒ S5.

- shifts / Reduce in ACTION.

→ **Rules for GOTO:-**

- only shifts
- AND NUMBERS ONLY.

Scanned by CamScanner

Step 4:- parsing the input : (id * id + id)

- push initial state '0' to stack , (pointer : id)

id * id + id $

↑ ↑ ↑ ↑ ↑

$	0	

[0, id] = S5

Rule: push id, 5 , (pointer: *)
pointer ++

$	0	id	5

[5, *] : r6 (id → F)

Rule: For one reduction pop two characters

$	0	~~id~~	~~5~~

pop: id, 5 ; push: F

$	0	F	

[0, F] : 3 ⇒ push: 3

$	0	~~F~~	~~3~~

[3, *] : r4 (F → T)
pop: 3, F ; push: T

$	0	T	

[0, T] : 2 : push: 2

$	0	T	2

[2, *] : S7
pointer ++ ; (pointer: id)
push: 7, *

$	0	T	2	*	7

[7, id] : S5
pointer ++ ; (pointer: +)
push: id, 5

~~$~~	~~0~~	~~T~~	~~2~~	~~7~~	~~5~~

[5, +] : r6 (id - F)

$	0	T	2	*	7	~~id~~	~~5~~

pop: id, 5 ; push: F

$	0	T	2	*	7	F

[7, F] : 10 ; push: 10

~~$~~	~~0~~	~~T~~	~~2~~	~~*~~	~~7~~	~~F~~	~~10~~

[10, +] : r3. (T * F → T)
pop: 10, F, 7, * ; push: T

$	0	T	2

[0, T] : 2

[2, +] : r2 (T, E)

, F] : 3
= [0, F, 3]
2, NULL) :
= [0, T] : 2 .

Scanned by CamScanner

$	0	~~T~~	~~2~~

pop: 2, T ; push E

id * id + id $

(44)

$	0	E	

: [0, E] : 1 push: 1

$	0	E	1

[1, +] : S6

pointer ++ ; (pointer: id).

push: (+, 6)

$	0	E	1	+	6

[6, id] : S5

pointer ++; (pointer: $)

push (id, 5)

$	0	E	1	+	6	~~id~~	~~5~~

[5, $] : r6 (id → F).

pop (5, id) ; push: F

$	0	E	1	+	6	F	

[6, F] : 3

push: (F, 3)

$	0	E	1	+	6	~~F~~	~~3~~

[3, $] : r4 (F → T)

pop: 3, F, push: T

$	0	E	1	+	6	T

[6, T] : 9 ; push 9

$	0	~~E~~	~~1~~	~~+~~	~~6~~	~~T~~	~~9~~

E.

[9, $] : r1 (E + T → E)

pop: 9, T, 6, + push

$	0	E		
				1

[0, E] : 1 push: 1.

[1, $] : accept.

∴ (id * id + id) is a valid string for the Given Grammar

Scanned by CamScanner

⇒ CALR Parser:-

- Canonical - LR Parser - most powerful - High error detection Capability.
- uses LR(1) items: LR(0) items + 1 Look Ahead symbol.
- e.g. for a production $S' \to S$

  LR(0) item = $S' \to \cdot S$

  LR(1) item = $S' \to \cdot S, \$$
- we Substitute reduce symbol, only in the fields pertaining to a production's look ahead
- e.g. Consider a state in a Grammar. $I_q : C \to aC \cdot, \$$ and the production number is (2). we Keep r_2, in the qth level, and under \$, as \$ being the lookahead.

→ Consider the Grammar, $S \to CC$

$C \to aC$

$C \to d$

→ Augmented Grammar: $S' \to S$

$S \to CC$

$C \to aC$

$C \to d$

→ LR(1) items:- $S' \to \cdot S, \$$ (∵ \$, the standard look ahead for first production of augmented Grammar).

- As 'S' is non-terminal, preceeding after (·), we have to write S's productions, and Complete the closure.

$S' \to \cdot S, \$$

$S \to \cdot CC, ?$

Scanned by CamScanner

- So, for obtaining the look ahead, we have a standard Rule.

→ <u>STD. Rule:-</u> If $A \to \alpha \cdot X\underline{B}, \underline{a}$, and $X \to \gamma$, then write $X \to \cdot\gamma, b$, where 'b' is look-ahead symbol and b: FIRST(B, a)

- So for the above production:-

$S' \to \cdot S, \$$, Comparing with Std. rule $A \to \alpha \cdot XB, a$, we have

$A: S'; \alpha: \varepsilon; X = S; B: \varepsilon$ ($\because$ as there are no relevant symbols, $\alpha, B: \varepsilon$)
$a: \$$

$\Rightarrow S' \to \cdot \underline{S}, \$$

$S \to \cdot \underline{CC}, \underline{b}$, b: FIRST(B, a)

b: FIRST$(\varepsilon, \$)$

$b: \$$

$\therefore S \to \cdot CC, \$$

- We have to write C's productions as it is a non-terminal preceding (·).

$\therefore S \to \cdot CC, \$$

$C \to \cdot aC, ?$

$C \to \cdot d, ?$

$S \to \cdot CC, \$$

$A \to \alpha \cdot XB, a$

$A: S; \alpha: \varepsilon; X: C; B: C; a: \$$

$X \to \gamma$, b: FIRST(B, a)

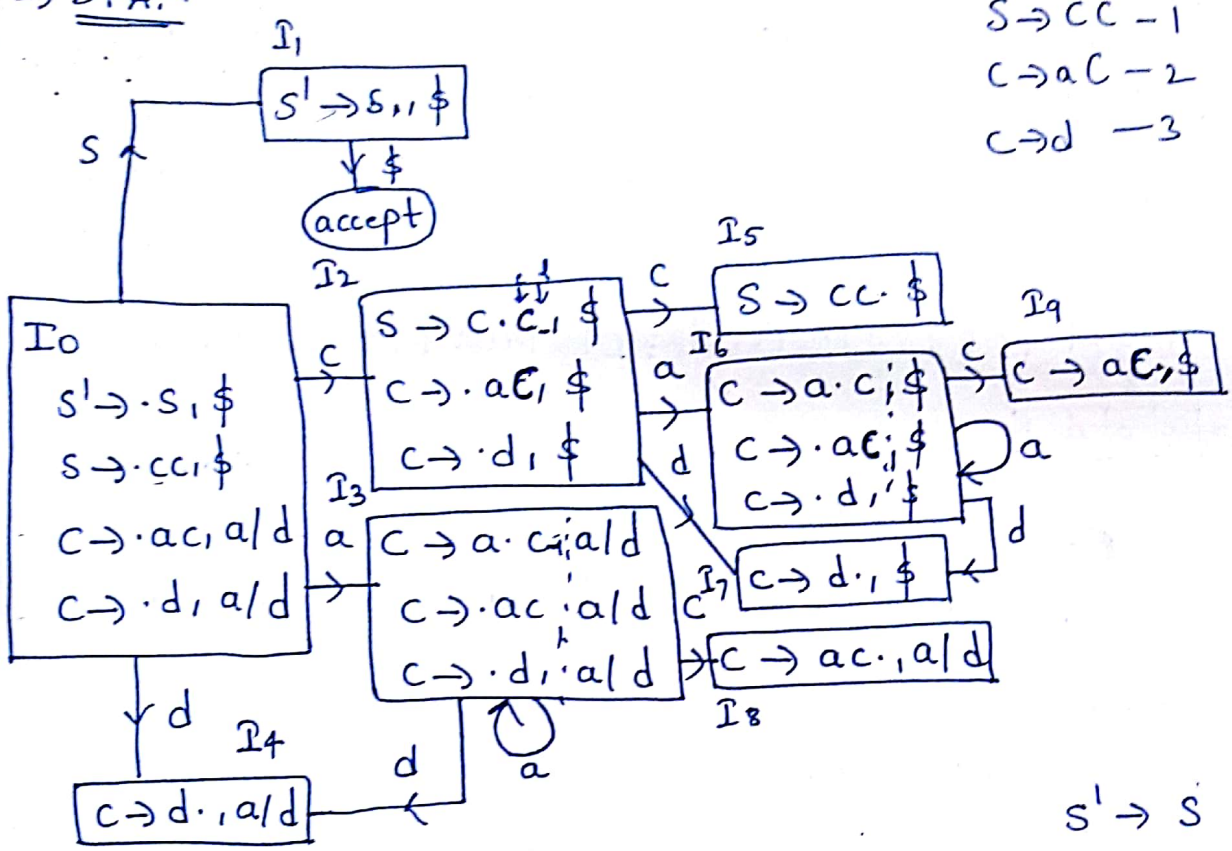
$C \to aC / d$ = FIRST$(C, \$)$

$\therefore ?$ = (Look-Ahead) = (a, d)

Scanned by CamScanner

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

⇒ DFA:-



$S' \to S$
$S \to CC$
$C \to aC / d$

⇒ PARSE - TABLE:-

State	ACTION			GOTO	
	a	d	$	S	C
0	S3	S4		1	2
1			ACCEPT		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

	FIRST	FOLLOW
S'	{a, d}	{$}
S	{a, d}	{$}
C	{a, d}	{a, d, $}

Scanned by CamScanner

⇒ INPUT - PARSING :- aadd$

$|0| [0, a] : S3, push : a, 3, advance the pointer (p: a)

$|0|a|3| [3, a] : S3, push : a, 3, advance the pointer (p: d)

$|0|a|3|a|3| [3, d] : S4, push: d, 4, advance the pointer (p: d)

$|0|a|3|a|3|d|4| [4, d] : r3 (d → c) pop: 4, d push: c

$|0|a|3|a|3|~~d~~|~~4~~|

$|0|a|3|a|3|c| [3, c] : 8 ; push: 8

$|0|a|3|a|3|c|8| [8, d] : r2 (ac → c) pop: 8, c, 3, a ; push: c

$|0|a|3|~~a~~|~~3~~|~~c~~|~~8~~|

$|0|a|3|c| [3, c] : 8 push: 8

$|0|a|3|c|8| [8, d] : r2 (ac → c) pop: 8, c, 3, a ; push: c

$|0|~~a~~|~~3~~|~~c~~|~~8~~|

$|0|c| [0, c] : 2 ; push: 2

$|0|c|2| [2, d] : S7, push: d, 7, advance the pointer (p: $)

$|0|c|2|d|7| [7, $] : r3, d → c, pop: d, 7, push: c

$|0|c|2|~~d~~|~~7~~|

$|0|c|2|c| [2, c] : 5 ; push: 5

$|0|c|2|c|5| [5, $] : r1 (cc → s) pop: 5, c, 2, c push: s

$|0|~~c~~|~~2~~|~~c~~|~~5~~|

$|0|s| [0, s] : 1 push: 1

$|0|s|1| [1, $] : ACCEPT ∴ The string 'aadd' is valid.

Scanned by CamScanner

⇒ LALR- PARSER

- LALR parser works on the same methodology of CALR.
- Difference with parse-table and input-parsing.
- In the above DFA, for Grammar $S \to CC$; $C \to aC$; $C \to d$ the states (3,6) (4,7) (8,9) has same productions but different lookaheads. Hence they are merged to get new hybrid states. They are as follows:-

I_{36}: $C \to a \cdot C$, a/d/\$
$C \to \cdot aC$, a/d/\$
$C \to \cdot d$, a/d/\$

I_{47}: $C \to d \cdot$, a/d/\$

I_{89}: $C \to aC \cdot$, a/d/\$

⇒ PARSE- TABLE:-

- The merging of states will reflect in the Parse table also.
- The New table will be, optimized one having low levels.

State	a	d	\$	S	C
0	S36	S47		1	2
1			ACCEPT		
2	S36	S47			5
36	S36	S47			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Scanned by CamScanner

INPUT- PARSING : aadd$

$|0| [0, a] : S36, push: a, 3, 6, Advance the Pointer

$|0|a|3|6| [36, a] : S36, push: a, 3, 6, Advance the Pointer

$|0|a|3|6|a|3|6| [36, d] : S47, push: a, 4, 7, Advance the Pointer

$|0|a|3|6|a|3|6|d|4|7| [47, d] : r36, d → c pop: d, 4, 7, push: c

$|0|a|3|6|a|3|6|~~d~~|~~4~~|~~7~~|

$|0|a|3|6|a|3|6|c| [36, c] : 89, push: 8, 9

$|0|a|3|6|a|3|6|c|8|9| [89, d] : r2, ac → c pop: a, 3, 6, c, 8, 9
push: c

$|0|a|3|6|~~c~~|~~3~~|~~6~~|~~c~~|~~8~~|~~9~~|

$|0|a|3|6|c| [36, c] : 89, push: 8, 9

$|0|a|3|6|c|8|9| [89, d] : r2, ac → c pop: a, 3, 6, c, 8, 9
push: c

$|0|~~a~~|~~3~~|~~6~~|~~c~~|~~8~~|~~9~~|

$|0|c| [0, c] : 2 ; push: 2

$|0|c|2| [2, d] : S47, push : d, 4, 7, Advance the Pointer (p: $)

$|0|c|2|d|4|7| [47, $] : r3, d → c , pop: d, 4, 7, push: c

$|0|c|2|~~d~~|~~4~~|~~7~~|

$|0|c|2|c| [2, c] : 5, push: 5

$|0|c|2|c|5| [5, $] : r1, cc → s , pop: 5, c, 2, c, push: s

$|0|~~c~~|~~2~~|~~c~~|~~5~~|

$|0|s| [0, s] : 1 , push: 1

$|0|s|1| [1, $] : ACCEPT

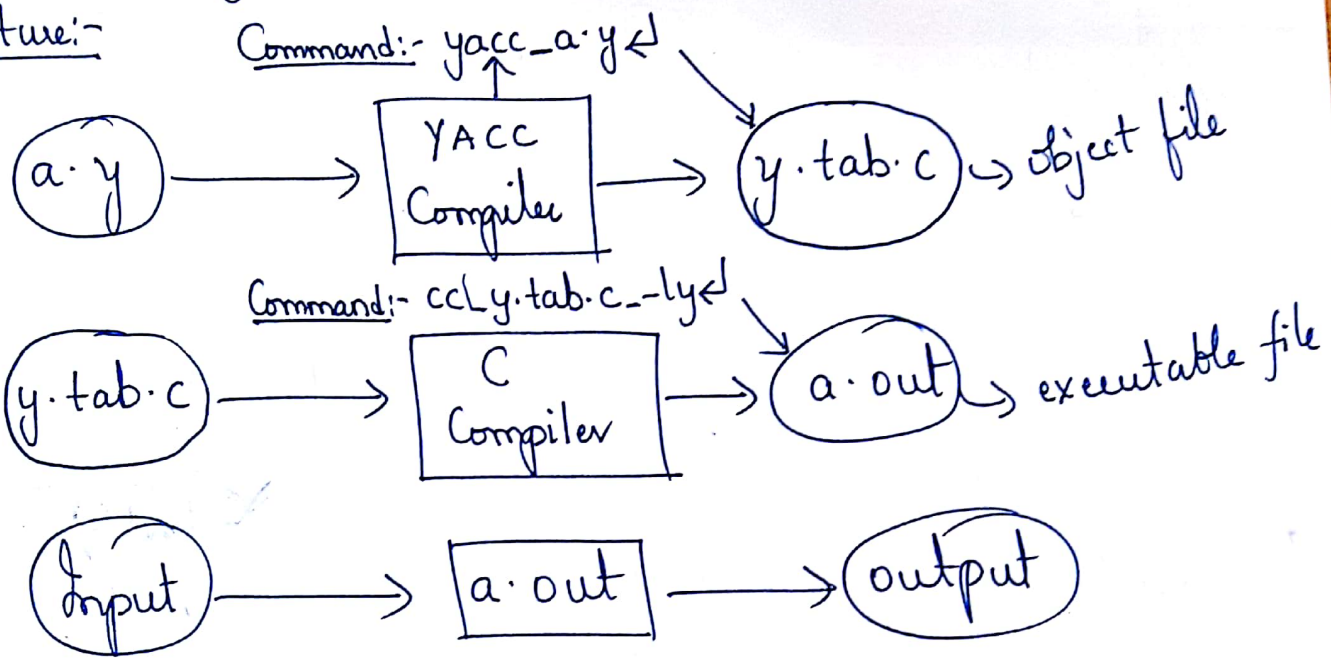
∴ String 'aadd) is valid

Scanned by CamScanner

⇒ AUTOMATIC - PARSER GENERATOR - YACC

- utility from UNIX.
- It generates 'LALR'.
- works along with lex.
- extension : a.y (e.g)

→ **Structure:-**



→ **Overview:-**

```
%{
    Declaration Section
%}
%%
    Translation rule Section - CFG
%%
%%
    C-functions
%%
```

Scanned by CamScanner

UNIT-4

SEMANTIC ANALYSIS

SDD:- Syntax - directed definition.

- CFG together with attributes & rules.
- X : symbol, a : attribute, ⇒ X.a : value of a at X.

Attributes:-

1) **Synthesized:-** Non-terminal : 'A'
Node : 'N'

$S.a\big|_A$ = Production at 'N', where Head = 'A' (only)

→ S.a - Can be defined only at 'N' and its children.

2) **Inherited:-** Non-terminal : 'B'
Node : 'N'

$i.a\big|_B$ = Production at the 'N's Parent', where Head = 'B'.

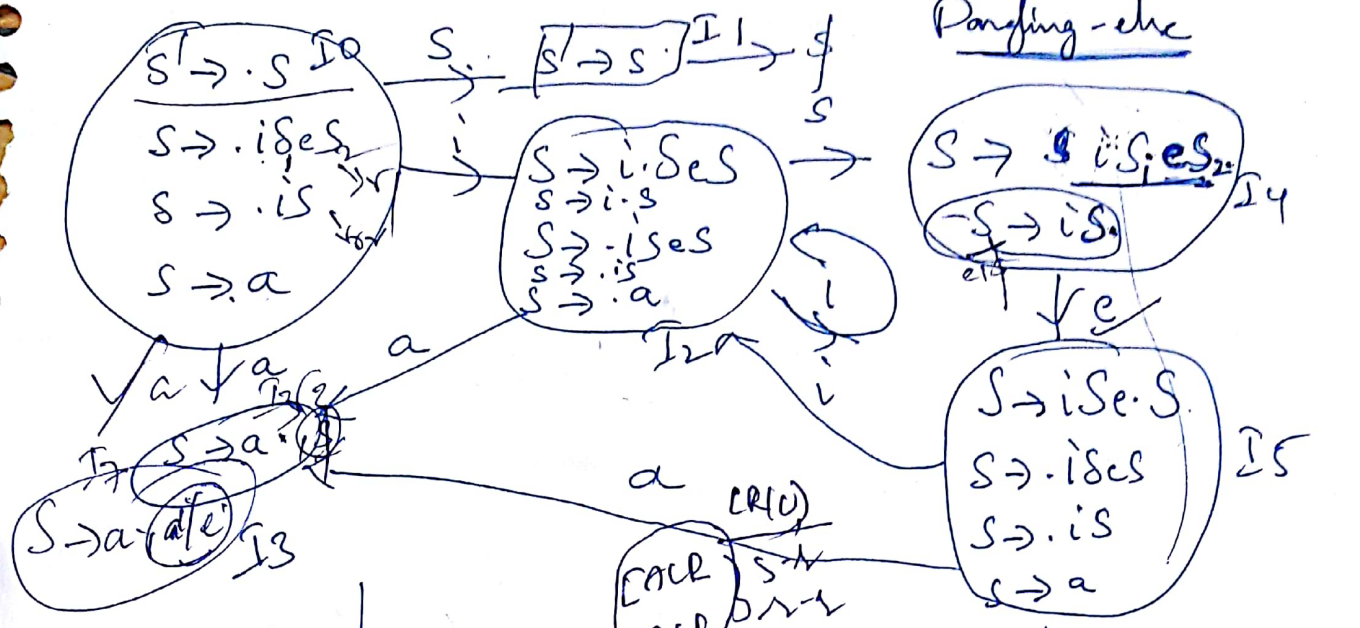
→ i.a - Can be defined at N's parent, itself, siblings

→ **SDD:-**

Production	Rules
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$
$L \to E$	$L.val = E.val$ (final).

Scanned by CamScanner

Dangling-else



	i	e	a	$	S
0	S2		S3		1
1				acc	
2	S2		S3		4
3		r3		r3	
4		S5		r2	
5	S2		S3		6
6		r1		r1	

iiaea$

for every terminal atleast oneshift state

Hence we go for 's' over 1,s' for e on 4

iiaea$ S → iSeS

T.C.

S.T	I.B	ACTION
$0	iiaea$	S2
$0i2	iaea$	S2
$0i2i2	aea$	S3
$0i2i2a3	ea$	r3 (a→S)
$0i2i2S	ea$	(2,S) = 4
$0i2i2S4	ea$	S5
$0i2i2S4e5	a$	S3
$0i2i2S4e5a3	$	r3 (a→S)
$0i2i2S4e5S	$	(5,S) = 6
$0i2i2S4e5S6	$	r1 (iSeS→S)
$0i2S	$	(2,S) = 4
$0S4	$	r2 (iS→S)
$0S	$	(0,S) = 1
$0S1	$	ACCEPT

Scanned by CamScanner

104 - 218
372 - 396
400 - 468
498 - 566
600 - 604
660 - 714

103 - 220
372 - 397
399 - 475
497 - 567
599 - 605
660 - 715

	a	b	c	d	$	S	A	B
0		S3		S5		1	2	4
1					Acc			
2	S6							
3				S9			7	8
4			S10					
5								
6								
7			S11					
8	S12							
9								
10								
11								
12								

bda bdc

$S \to Aa \mid bAc \mid Bd \mid bBa$

LR $A \to d$

I_0: $S' \to .S, \$$; $S \to .Aa, \$$; $S \to .bAc, \$$; $S \to .Bc, \$$; $S \to .bBa, \$$; $A \to .d, a$; $B \to .d, c$

$S \xrightarrow{} I_1$: $S' \to S., \$$ (Acc)

$A \xrightarrow{} I_2$: $S \to A.a, \$$

$b \xrightarrow{} I_3$: $S \to b.Ac, \$$ $S \to b.Ba, \$$ $A \to .d, c$ $B \to .d, a$

$B \xrightarrow{} I_4$: $S \to B.c, \$$

$d \xrightarrow{} I_5$: $A \to d., a/c$ $B \to d., a/c$

$a \xrightarrow{} I_6$: $S \to Aa., \$$

$A \xrightarrow{} I_7$: $S \to bA.c, \$$

$B \xrightarrow{} I_8$: $S \to bB.a, \$$

$c \xrightarrow{} I_9$: $S \to bAc., \$$

$a \xrightarrow{} I_{10}$: $S \to bBa., \$$

$c \xrightarrow{} I_{10}$: $S \to Bc., \$$

$d \xrightarrow{} I_{11}$: $A \to d., c$ $B \to d., a$

$d \xrightarrow{} I_{12}$: $A \to d., a/c$ $B \to d., a/c$

$A \to d \to r5$

$B \to d \to r6$

$S \to Bc., \$$

$S \to Aa$ - (1)
$S \to bAc$ - (2)
$S \to Bc$ - (3)
$S \to bBa$ - (4)
$A \to d$ - (5)
$B \to d$ - (6)

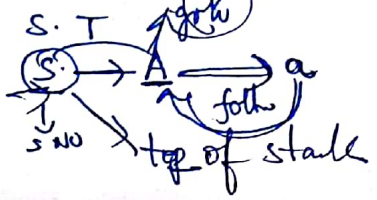
FR	fol
S {b,d}	{$}
A {d}	{a/c}
B {d}	{a/c}

Scanned by CamScanner

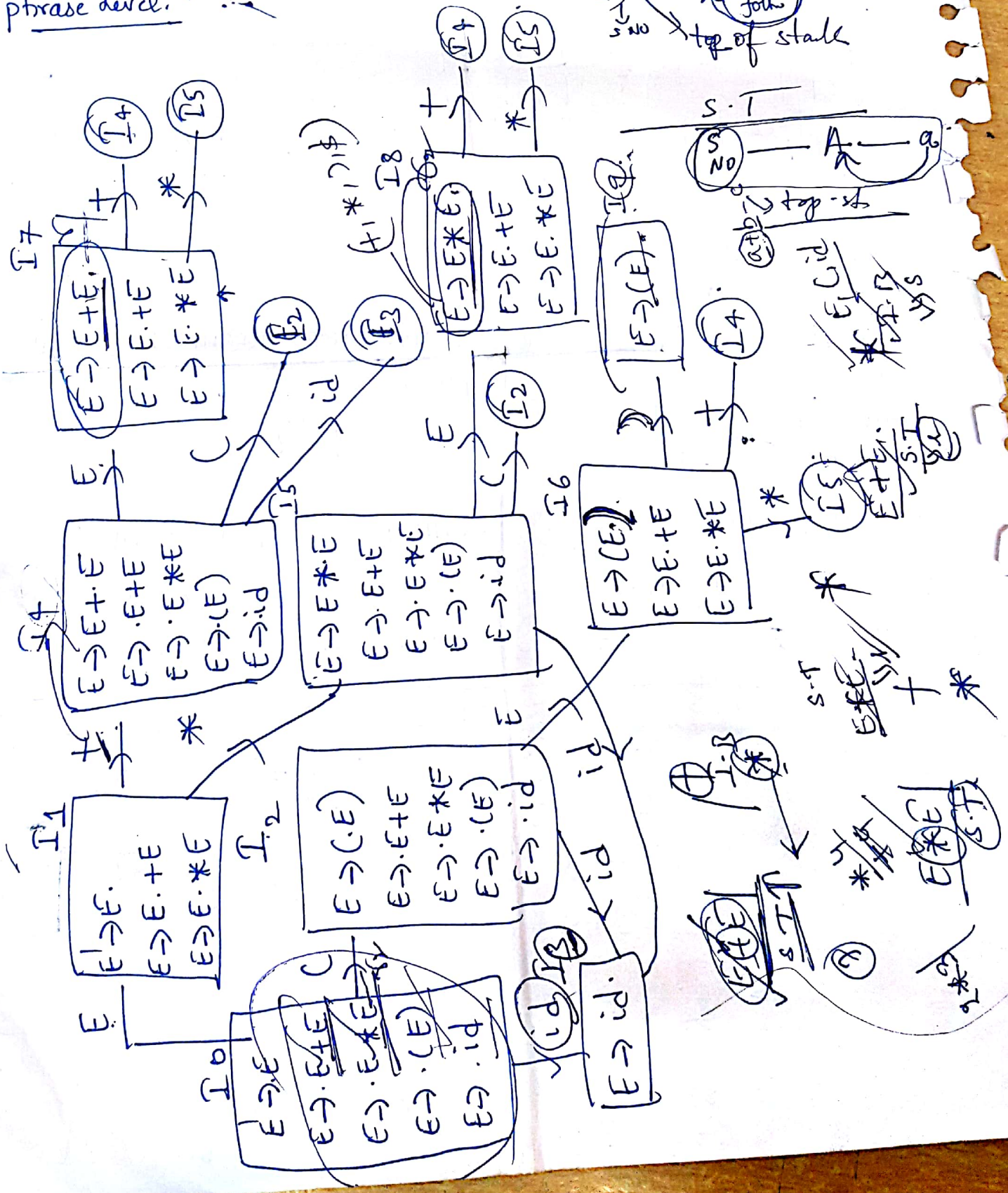
Error Recovery – LR Parsing

1. **Exhaustive Method:-** Examine each entry, Construct recovery procedure based on language.

2. **Panic Mode:-** Scan the S.T, for a state 's' that has goto on N.T 'A'. discard the Input symbols until 'a' is found, where 'a' is in follow of 'A'. add s to the top of stack. parsing continues.



3. **phrase level:-**



Scanned by CamScanner

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

(R10)
(R10)

	+	*
7	S4 / r1	S5 / r1
8	S4 / r2	S5 / r2

Error Routines:-

e1: MISSING OPERAND

0,2,4,5 - id, (

action: push id and goto I3

(0,2,4,5) on id → I3 · (id,3)

e2: UNBALANCED RIGHT PARANTHESIS

0,1,2,4,5 -) on stack, not in

action: pop ') from stack. I-B

e3: MISSING OPERATOR "+"

1,6 Encountered id, (

action: push + and goto I4 (+,4)

e4: MISSING RIGHT PARANTHESIS

6 expects operator) (),9)

action: push) to stack goto I9.

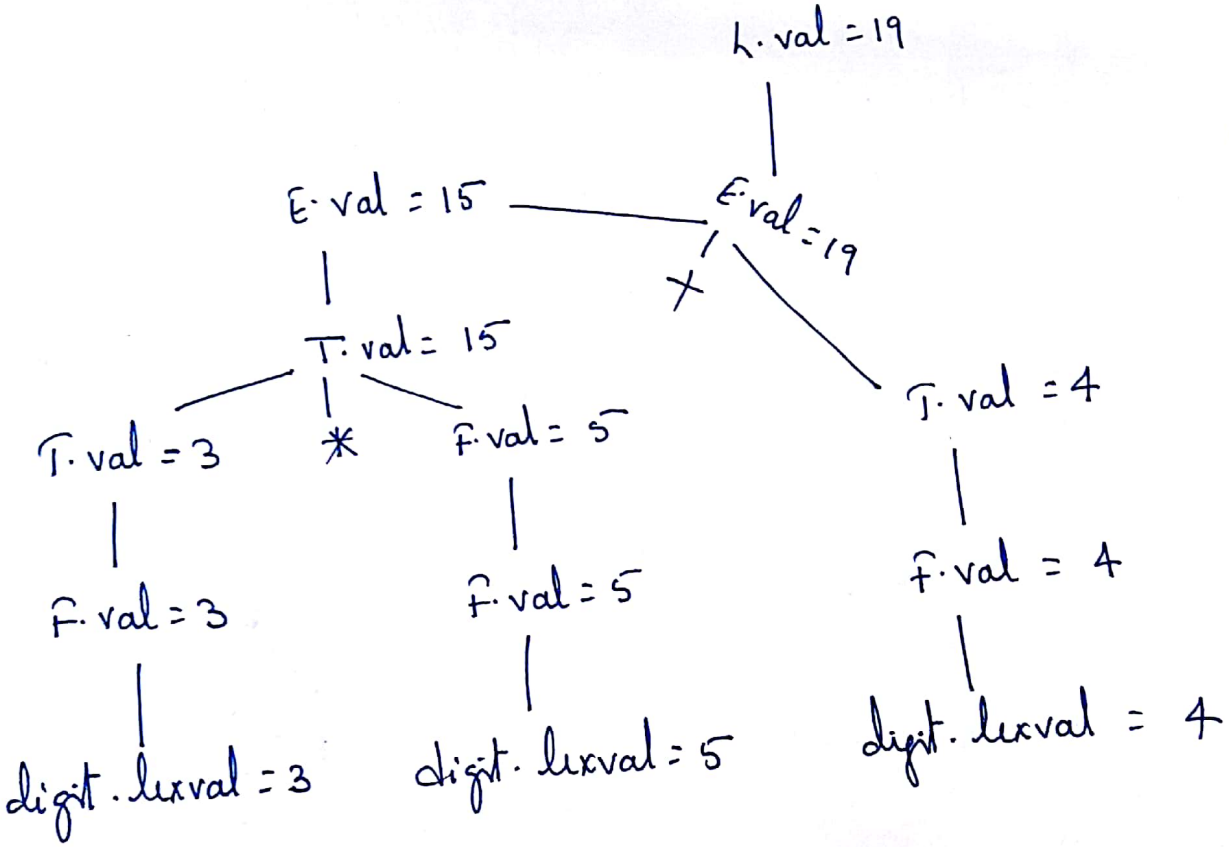
S.T	I.B	ACTION
$ 0	id + ( ) $	
$ 0 id 3		S3
$ 0 id 3		r4 (id → E)
$ 0 E		
$ 0 E 1		clash (0, E) = 1
$ 0 E 1 + 4		S4
$ 0 E 1 + 4		e2: pop)
$ 0 E 1 + 4 id 3		e1: push id goto I3
$ 0 E 1 + 4 E 7		r4 (id → E)
$ 0 E 1 + 4 E 7		
$ 0 E 1 + 4 id 3		
$ 0 E 1 + 4 E 7		
$ 0 E 1 + 4 E 7		
$ 0 E 1 + 4 E 7

→ e.g: 3 * 5 + 4

Step 1: 3, 5, 4 : digit.lexval , Replace with L.H.S. (F → digit)

∴ digit.lexval = F.val = 3, 5, 4.

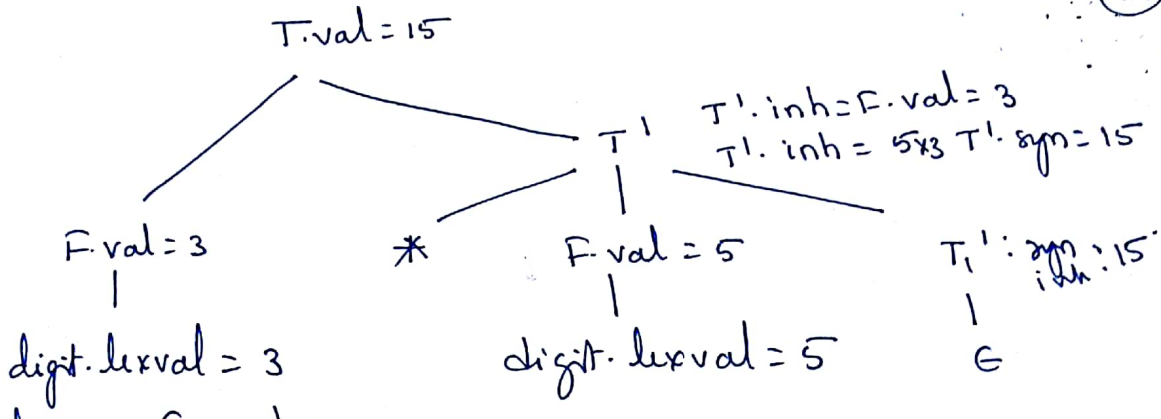
∴ F.val = T.val = 3, 5, 4



→ e.g:- 3 * 5

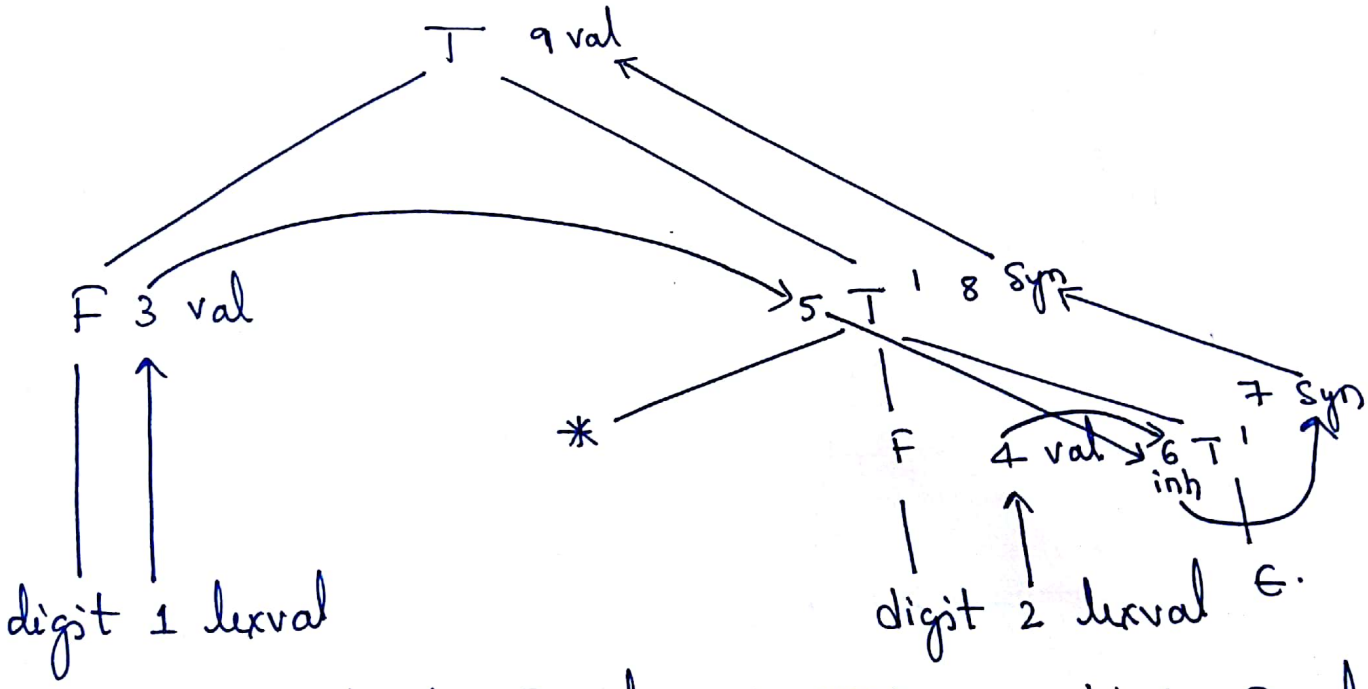
PRODUCTION	RULES
$T \rightarrow FT'$	$T'.inh = F.val$; $T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$; $T'.syn = T_1'.syn$.
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$

Scanned by CamScanner



=) <u>Dependency Graphs</u>:-

- useful tool for determining evaluation order.
- It determines how to compute values for annotated Parse trees.
- It depicts the flow of information among attributes.



→ 3 ⟶ 5 : $T'.inh = F.val$ (R.C, L.C) 4 ⟶ 6 : $T'.inh = F.val$ (R.C, L.C.)

→ 5 ⟶ 6 : $T' : * F T'$

4 ⟶ 6 : ϵ * 3 5 : 15.

Scanned by CamScanner

⇒ S- Attributed & L- Attributed definitions :-

1) **S-attributed** :- An SDD is S-Attrib if every attribute is Synthesized.

2) **L-attributed** :- An SDD is L-Attrib, if

i) The attributes move from left → right,

ii) They are Synthesized (or)

iii) Inherited, and for production $A \to X_1 X_2 \ldots X_n$, and an inherited attribute $X_i.a$, Computed by a rule. Then the rule may use :-

(a) Inherited-attributes associated with 'A' (head).

(b) Either-inherited (or) Synthesized attribute with occurrence of symbols $X_1, X_2 \ldots X_{i-1}$, located to the left of X.

(c) Inh & Syn. attributes associated with X_i itself.

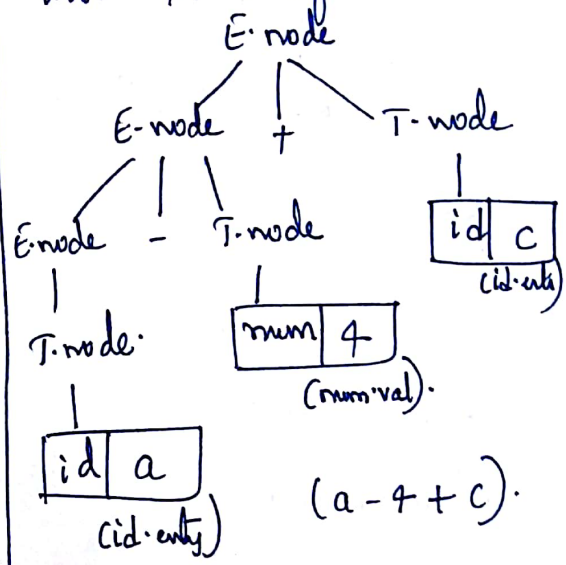
⇒ Application - SDT :-

1) Construction of Syntax Trees.

2) Parser - stack Implementation

3) postfix evaluation

4) Eliminating left recursion.



(a - 4 + c).

⇒ Construction of Parse-Trees :-

$E \to E+T$ / E.node = new Node(+, E1.node, T.node)

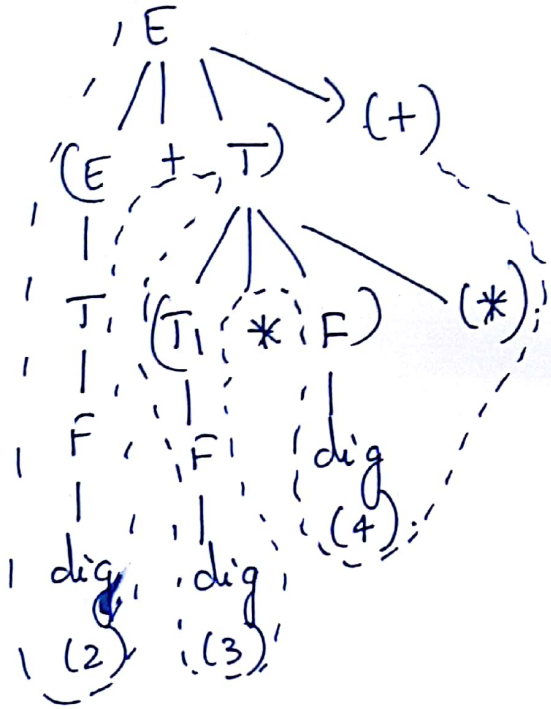
$E \to E_1 - T$ / E.node = new Node(−, E1.node, T.node)

$E \to T$ / E.node = T.node | $T \to id$ / T.node = new Leaf(id, id-entry).

$T \to (E)$ / T.node = E.node | $T \to num$ = new Leaf(num, num.val)

Scanned by CamScanner

⇒ **Post-fix evaluation:-** $(2+3*4) \Rightarrow (234*+)$



$(234*+)$

⇒ **Parser-stack Implementation:-**

	X	Y	Z
	X.x	Y.y	Z.z

↑ top

$A \rightarrow XYZ.$

⇒ Reduction case in LR.

⇒ **Eliminating left-Recursion:-**

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

$A \rightarrow \beta R$

$R \rightarrow \alpha R$

$R - \epsilon$

$E \rightarrow \underset{A}{E} + \underset{\alpha}{T}$ $(A \rightarrow A\alpha)$

$E \rightarrow T$

$(\because \alpha = +T)$

$\therefore E \rightarrow TR$

$R \rightarrow +TR$

$R \rightarrow \epsilon$

Scanned by CamScanner

⇒ **Exercise - Additional Reading**

- Semantic Actions Inside Productions (SDT - concept)

- e.g: $E \rightarrow E+E$ (print +)
  $E \rightarrow E*E$ (print *)
  $E \rightarrow id$ (print id.value)

/ NOTE:- Actions can be placed only at R.H.S */

⇒ **SDT:- Advantages.**

1) To Construct Parse Tree. & syntax trees.
2) To Construct & update symbol table.
3) To check Types, parameters, equivalence.
4) To evaluate algebriac expressions
5) To generate Intermediate code.

⇒ **Grammar:-**

$E \rightarrow E+E$

$E \rightarrow E*E$

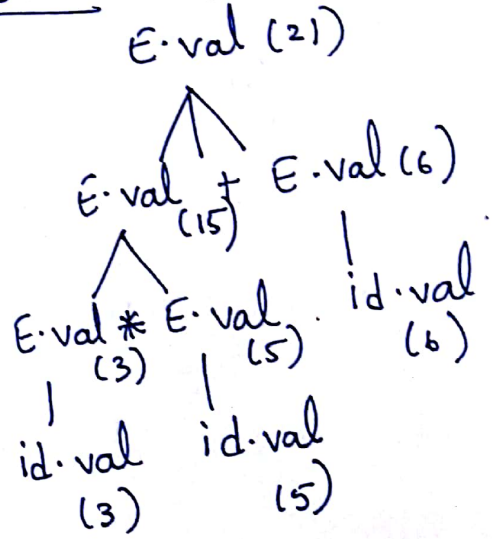
$E \rightarrow id$

String:-

3 * 5 + 6

/* APT should be developed keeping in view - Input */

Annotated Parse-Tree



⇒ **Grammar:-**

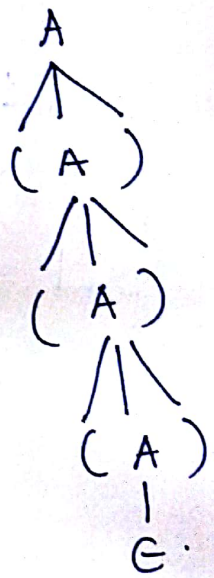
$A \rightarrow (A)$

$A \rightarrow \epsilon$

input:-

((( )))

APT:-



Scanned by CamScanner

→ **SDT- Type checking:-**

- Input :- $(2+3) == 6$

$E \to E == E$ (if $(E_1.type = E_2.type)$ & $(E_1.val = E_2.val)$

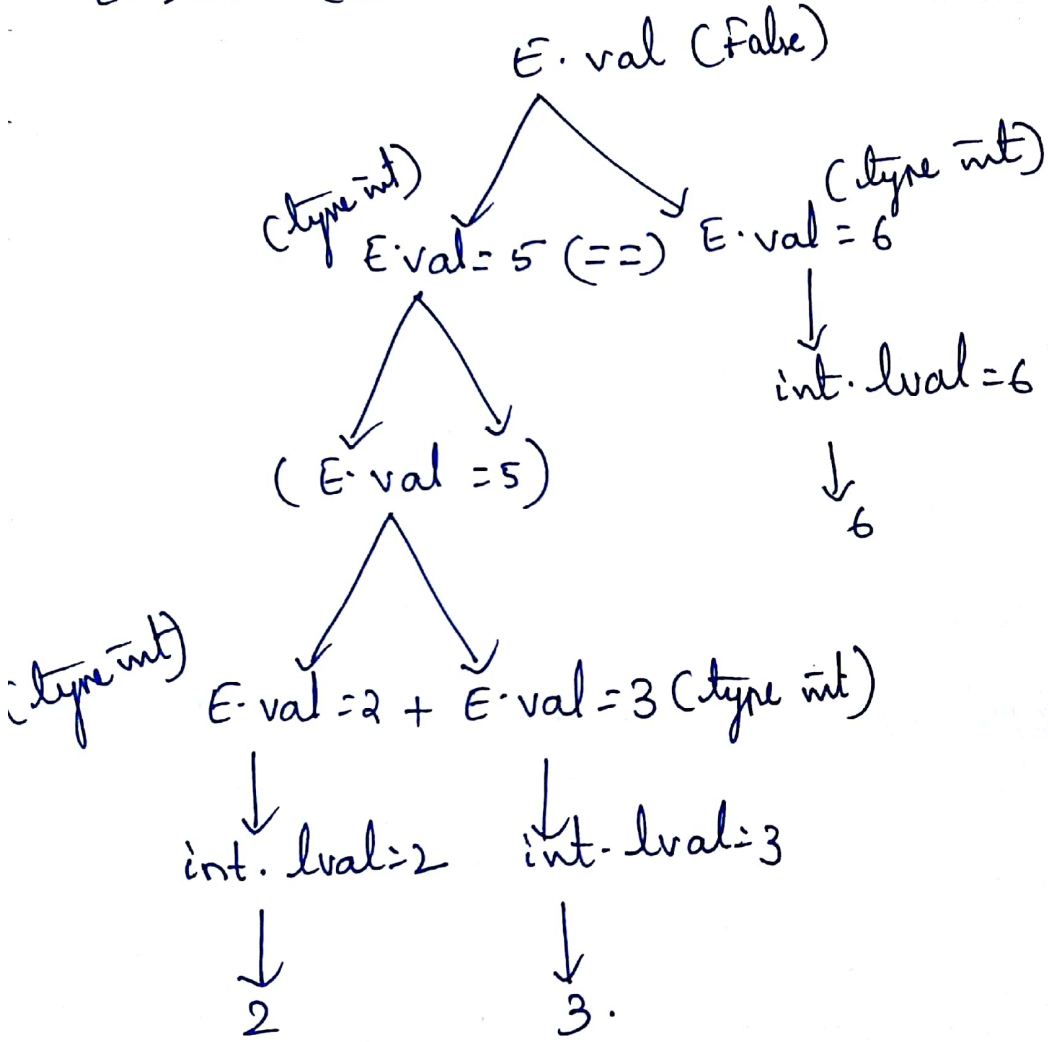
$E \to E + E$ — then $E.type = bool$) — $(E.val = E_1.val + E_2.val)$

$E \to (E)$ $(E.val = E_1.val)$

$E \to True$ $(E.val = Bool)$

$E \to False$ $(E.val = Bool)$

$E \to int$ $(E.val = int.lexval)$.



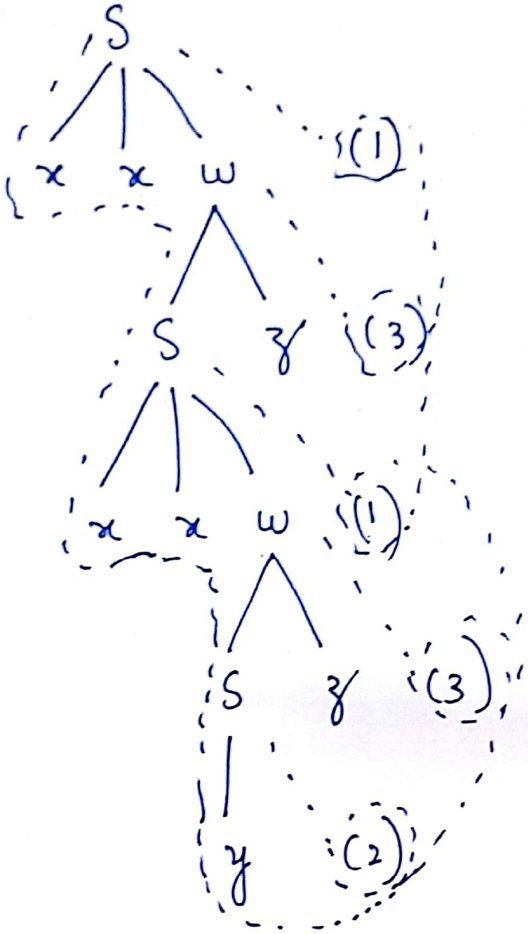
→ $S \to xxw$ (print 1)

$S \to y$ (print 2)

$w \to Sz$ (print 3)

Input :- $xxxxyz$

Scanned by CamScanner



outputs:-

2 3 1 3 1

⇒ SDT- clarification:-

"S"

- only syn. attributes
- actions at R.H.S, and are evaluated during parsing.

e.g: $S \to AB$ (print 1)
$A \to a$ (print 2)
$B \to b$ (print 3).

/ * Actions should be placed only at R.H.S.

"L"

↳ Both syn & inh. attributes.
- inh. attrib restricted to inherit from parent (or) left sibling only.

- e.g: $S \to A$ (print 1) B
$A \to$ print(2) a B
$B \to b$ print (3).

/ * Actions can be placed anywhere.

Scanned by CamScanner

⇒ Conversion:- (L → S)

$S \to A\{print\ 1\}B \Rightarrow S \to AMB$
$M \to \epsilon\{print\ 1\}$

$A \to \{print\ 2\}aB \Rightarrow A \to PaB$
$P \to \epsilon\{print\ (2)\}$

$B \to b\{print\ 3\} \Rightarrow B \to b\{print\ (3)\}$ (∵ No change required).

Scanned by CamScanner

INTERMEDIATE CODE GENERATION

→ **3AC:-** at-most three address

$$a = b \ op \ c$$
$$\downarrow_1 \quad \downarrow_2 \quad \downarrow_3$$

e.g: $a = b + c * d$

3AC:- $t_1 : b + c$
$t_2 : t_1 + d$
$a : t_2$

t_1, t_2 : temporary Names.

→ **Implementation:-**

1) **Quadraples:-** four fields:- op, arg1, arg2, result.

e.g. $x = -a * b + -a * b$

3AC:- t_1 : Uminus a
$t_2 : t_1 * b$
$t_3 : -a$
$t_4 : t_3 * b$
$t_5 : t_2 + t_4$
$x : t_5$

op	arg1	arg2	result
uminus	a		t1
*	a	b	t2
uminus	a		t3
*	t3	b	t4
+	t2	t4	t5
=	t5		x

2) **Triples:-** Temporary Variables avoided by referring to pointers.

Three fields:- Number, op, Arg1, Arg2

Scanned by CamScanner

Number	Op	Arg1	Arg2
(0)	Uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

=) 3AC for Expressions:- (operations within expressions)

$S \rightarrow id = E;$ S.Code = E.code || gen (top.get (id.lexeme) '=' E.addr)

$E \rightarrow E_1 + E_2;$ E.addr = new Temp()
E.Code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr +

$E \rightarrow -E_1;$ E.addr = new Temp()
E.Code = E_1.code || gen (E.addr '=' 'minus' E_1.addr)

$E \rightarrow (E_1);$ E.addr = E_1.addr.
E.Code = E_1.code.

$E \rightarrow id;$ E.addr = top.get (id.lexeme)
E.Code = ''.

Scanned by CamScanner

TRANSLATION OF EXPRESSIONS

- Expression with more than one operator, will translate into instruction with atmost one operator.
- e.g: a = b + -c

  t_1 = -minus c

  t_2 = b + t_1

  a = t_2.

→ **Incremental transition:-**

$S \rightarrow id = E;$ { gen(top.get(id.lexeme) '=' E.addr); }

$E \rightarrow E_1 + E_2;$ { E.addr = new Temp();
gen(E.addr '=' E_1.addr '+' E_2.addr); }

| $-E_1$ { E.addr = new Temp();
gen(E.addr '=' 'minus' E_1.addr); }

| (E_1) { E.addr = E_1.addr; }

| id { E.addr = top.get(id.lexeme); }

→ **Addressing Array elements:-**

- Can be accessed quickly, if stored in Consecutive locations
- **for 1D element:-**

$A[i] = base + i \times w$ → width

↳ element

relative address of Array [A(0)]

- **for 2D element:-** $A[i_1][i_2] = base + i_1 \times w_1 + i_2 \times w_2$.

Scanned by CamScanner

→ for relative elements:-

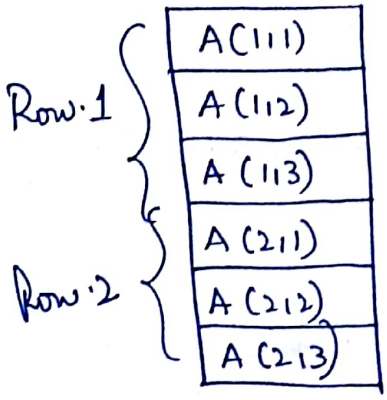
$A[i_1][i_2] = base + (i_1 \times n \times i_2) \times w$ / $A[i] = base + (i - low) \times w$ (1D).

No. of elements

→ when the array size is dynamic, compile-time pre-calculations cannot be applied for MD-Arrays. Then they should be evaluated, as they were written.

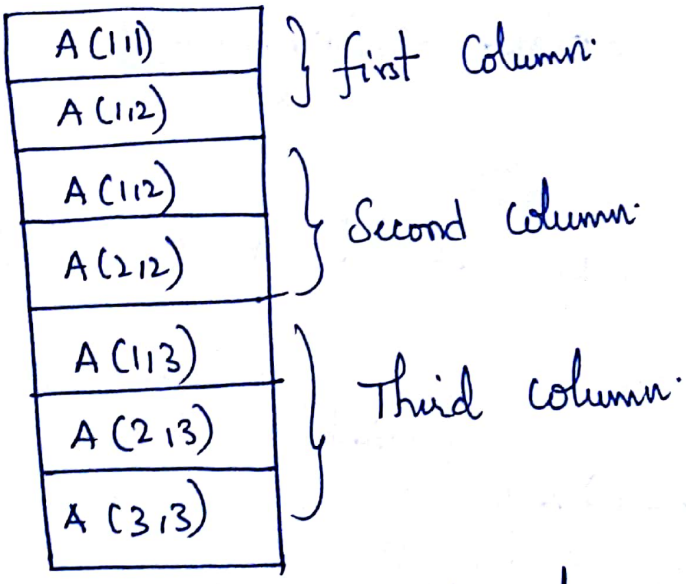
→ Above references are stored either in;

(a) Row-Major Layout:-



, when scanned, right-most subscripts vary fast

(b) Column-Major Layout:-



, when scanned leftmost subscripts vary fast.

Scanned by CamScanner

TRANSLATION OF ARRAY REFERENCES:-

Grammar:- $L \rightarrow L(E) \mid id(E)$

$S \rightarrow id = E;$ { gen (top.get (id.lexeme) '=' E.addr); }

$\mid L = E;$ { gen (L.array.base '[' L.addr ']' '=' E.addr); }

$E \rightarrow E_1 + E_2;$ { E.addr = new Temp();
gen (E.addr '=' E_1.addr '+' E_2.addr); }

$\mid id;$ { E.addr = top.get (id.lexeme); }

$\mid L;$ { E.addr = new Temp();
gen (E.addr '=' L.array.base '[' L.addr ']'); }

$L \rightarrow id[E];$ { L.array = top.get (id.lexeme);
L.type = L_1.type.elem;
t = new Temp();
L.addr = new Temp();
gen (t '=' E.addr '*' L.type.width);
gen (L.addr '=' L_1.addr '+' t); }

→ e.g:- C + a[i][j]

3AC: $t_1 = [i]$

$t_2 = [j]$

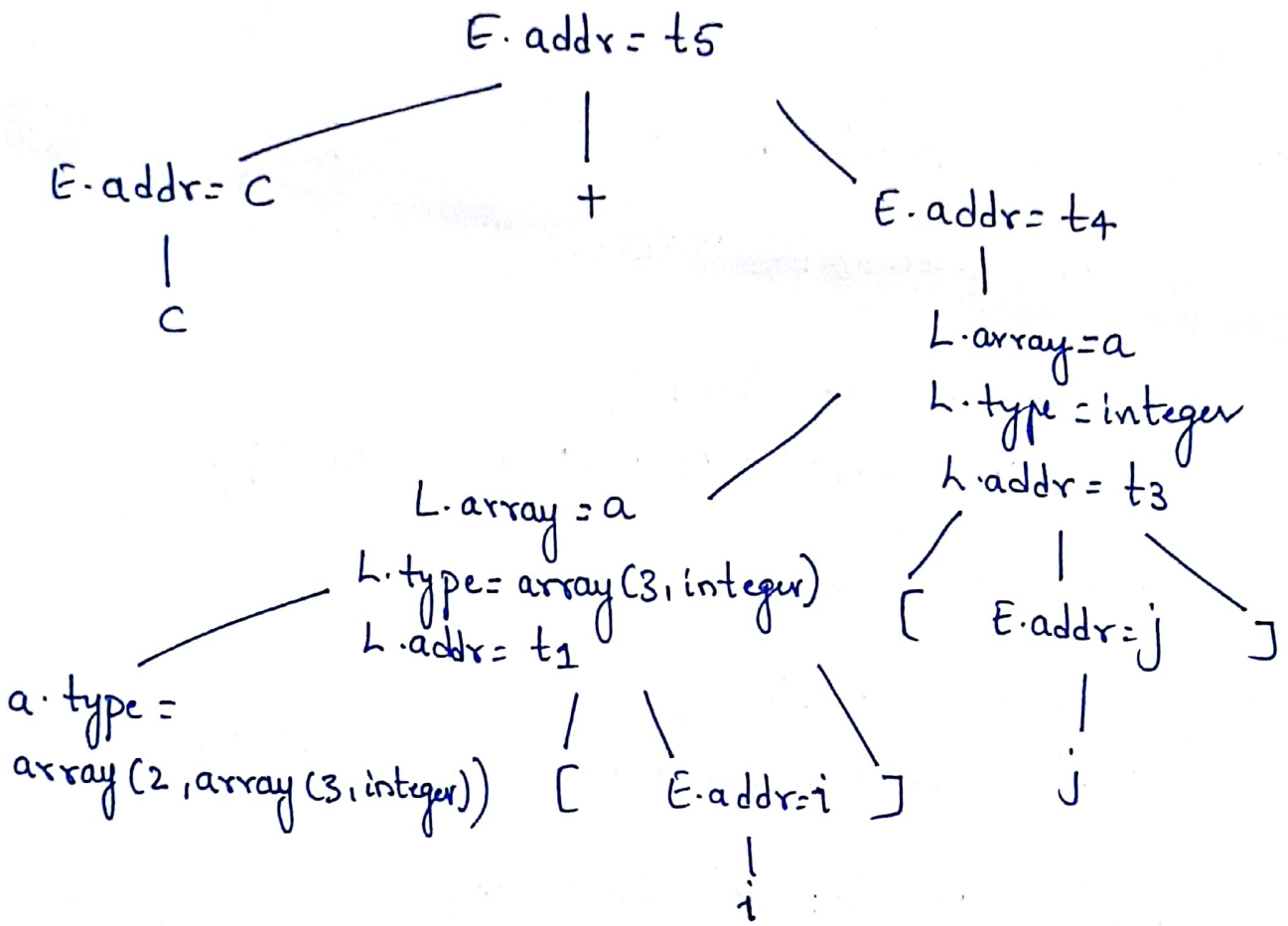
$t_3 = t_1 + t_2$

$t_4 = a[t_3]$

$t_5 = C + t_4$.

Scanned by CamScanner

⇒ Annotated Parse Tree:-



BACK PATCHING

- Consider the task of generating code for a statement:-

  if (B) S

  - The statement contains a 'jump' for 'S', when (B) is false.
  - Issue:- Target of the goto that jumps code for 'S'.

→ Defn:- A complementary approach, in which lists of jumps are passed as synthesized attributes. When a jump is generated, target left unspecified temporarily, put in a lists of jumps, whose labels are filled (to be), when determined properly.

Scanned by CamScanner

UNIT-V

CODE GENERATION

- **Input:-** . IR (Intermediate Representation)
  . Symbol table
- **Output:-** Semantically equivalent target program.

→ **Charecteristics:-**
- It should preserve the Semantic meaning of Source code
- High quality.
- efficient

→ **Challenge:-** getting an optimized code.

⇒ **Issues:-**

1. **Input to the Code generator:-**
- **choices:-** Quadruples, Triples, Indirect triples, byte code
  Stack-Machine code.
- **Assumptions:-** Front-end has Scanned, parsed, translated, errors-detected, type checked, Conversions inserted,

2. **Target Program:-**
- Key role played by Instruction-Set architecture.
- e.g: RISC, CISC, JVM.
- **Codes:-** Absolute ; relocatable.

3. **Instruction Selection:-**
- CG, must map IR to a Code Sequence, that can be executed by Target Machine.

Scanned by CamScanner

⇒ Charecteristics:-

1) IR level:- if high level; Conversion is done using templates.
- more efficient if low level.

2) Instruction Set:- uniformity & Completeness.
Speeds & idioms.

3) Code quality:- determined by speed & size.

4. Register Allocation :-
↳ fastest Computational unit.

(i) Allocation:- Selecting set of variables, that will reside in registers.

(ii) Assignment:- picking the specific register.

5. Evaluation Order:-

Scanned by CamScanner

⇒ **Backpatching:- Concepts:-**

(i) Truelist:- list of jumps and Conditions into which we must insert the label, if Control goes true.

(ii) Falselist:- list of instructions eventually get the label to which, Control goes if false.

(iii) Nextlist:- list of jumps to the instruction immediately following the code.

(iv) makelist:- Contains only 'i', an index into the array of instructions. It returns a pointer to the newly created list.

(v) merge(p1,p2):- Concatenates the lists pointed by p1, p2. returns a pointer to the newly formed list

(vi) backpatch(p,i):- inserts 'i' as the target label for each of the instructions on the list pointed to, by p.

⇒ **Back-patching:- Boolean expressions:-**

Grammar:- $B \rightarrow B_1 \,||\, M B_2 \mid B_1 \,\&\&\, M B_2 \mid !B_1 \mid (B_1) \mid E_1 \text{ rel } E_2 \mid$
true | false

$M \rightarrow \epsilon$

, M is a marker non-terminal, in the grammar, Causes a Semantic action to pick up, at appropriate times, the index of the next instruction to be generated.

Scanned by CamScanner

① $B \rightarrow B_1 \,||\, M B_2$ { backpatch (B1.falselist, M.instr);
B.truelist = merge (B1.truelist, B2.truelist);
B.falselist = B2.falselist; }

② $B \rightarrow B_1 \,\&\&\, M B_2$ { backpatch (B1.truelist, M.instr);
B.truelist = B2.truelist;
B.falselist = merge (B1.falselist, B2.falselist); }

③ $B \rightarrow ! B_1$ { B.truelist = B1.truelist;
B.falselist = B2.falselist; }

④ $B \rightarrow E_1 \text{ rel } E_2$ { B.truelist = makelist (nextinstr);
B.falselist = makelist (nextinstr+1);
gen ('if' E1.addr rel.op E2.addr 'goto' -')
gen ('goto -'); }.

⑤ $B \rightarrow \text{true}$ { B.truelist = makelist (nextinstr);
gen ('goto -'); }

⑥ $B \rightarrow \text{false}$ { B.falselist = makelist (nextinstr); }
gen ('goto -'); }

$M \rightarrow \epsilon$ { M.instr = nextinstr; }.

<u>Consider ①:-</u> $B \rightarrow B_1 \,||\, M B_2$

if B_1 is true, B is also true (B1.truelist = B.truelist)

if B_1 is false, B_2 must be tested, hence a <u>'jump'</u>.

Scanned by CamScanner

, then B_1. falselist, must be the beginning of code, done by Marker 'M'.

- It produces 'M. instr' a synthesized attribute.

$\{M.instr = nextinstr;\}$

→ 'nextinstr' is a Variable, will be patched onto the B_1. falselist.

→ B_1. falselist receives M. instr as its target label.

→ Consider (2):- $B \to B_1 \ \&\& \ MB_2$: Similar Semantics.

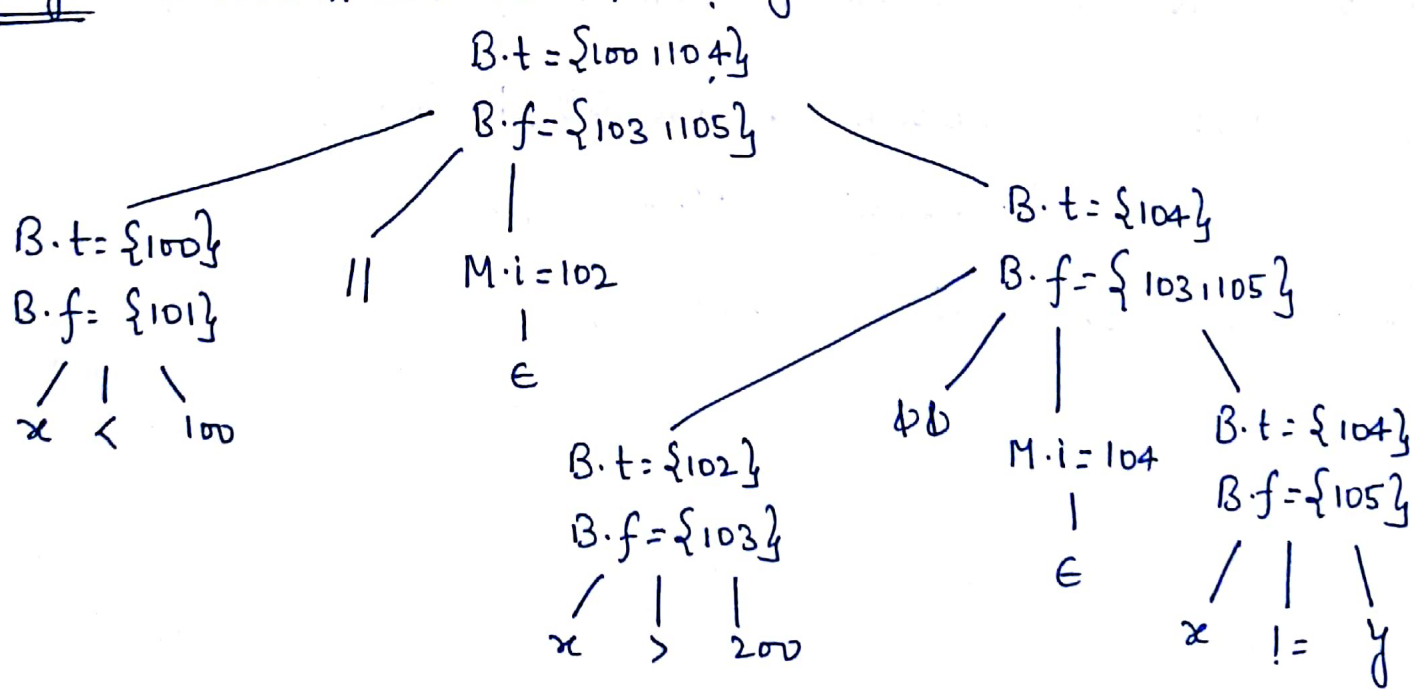
→ Consider (3):- $B \to !B$: B. truelist = B. falselist ;
B. falselist = B. truelist

→ Consider (4):- Ignores Parantheses.

→ Consider (5):- $B \to E_1 \ rel \ E_2$

Generates two instructions: One Conditional, one unconditional put on newlists, pointed by B. trulists, B. falselists.

→ e.g:- $x < 100 \ || \ x > 200 \ \&\& \ x != y$



Scanned by CamScanner

-> Backpatching:- flow of control Statements:-

Grammar:- $S \rightarrow if(B)\,s \mid if\,(B)\,S\,else\,s \mid while(B)\,s$

$\mid \{L\} \mid A;$

$L \rightarrow L\,S \mid S$

where, S : Statement, L : List; A : Assignment; B : Boolean.

$S \rightarrow if(B)\,M\,S_1$ { backpatch (B.truelist, M.instr);
S.nextlist = merge (B.falselist, S_1.nextlist); }

$S \rightarrow if(B)\,M_1\,S_1\,N\,else\,M_2\,S_2$ { backpatch (B.truelist, M_1.instr);
backpatch (B.truelist, M_2.instr);
S.nextlist = B.falselist;
gen ('goto' M_1.instr); }

$S \rightarrow \{L\}$ { S.nextlist = L.nextlist }

$S \rightarrow A$ { S.nextlist = null; }

$M \rightarrow \epsilon$ { M.instr = nextinstr; }

$N \rightarrow \epsilon$ { N.nextlist = makelist (nextinstr); gen ('goto' _'); }

$L \rightarrow L_1\,M\,S$ { backpatch (L_1.nextlist, M.instr);
L.nextlist = S.nextlist; }.

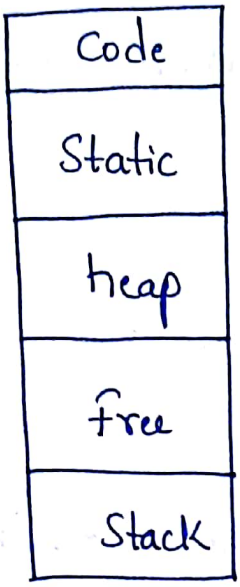
$L \rightarrow S$ { L.nextlist = S.nextlist; }.

Scanned by CamScanner

RUNTIME ENVIRONMENTS

→ Storage Organization:- Target program runs in a logical space, managed by Compiler, OS, System. OS maps log → phy.

- Memory Division:-



- Memory Allocation:-

(i) Static:- Allocation is made by the Compiler, looking only at the Context of the program.

(ii) Dynamic:- Allocation is made by the Compiler, only while the program is running.

- STRATEGIES:-

(i) Stack:- Names, local to a procedure are allocated space on Stack, Supports call / return mechanism.

(ii) Heap:- Area of Virtual memory that allows objects or other data elements to obtain storage, when they are created and to return that storage, when they are invalidated.

Scanned by CamScanner

STACK ALLOCATION OF SPACE:-

1) **Activation tree:-** Represents activation of procedures, during running of an entire program. Each node Corresponds to One activation, root is the main activation. At a node, for an activation, the children Corresponds to activations, are Called in left → right order. One child must finish, before the activation of next begins.

- Sequence of Calls: Pre-order traversal
- Sequence of returns: Post-order traversal.

2) **Activation Records:-**

- Each activation, has an activation record on the Control Stack, with root at the bottom

- Structure:-

Actual parameters
returns
Control link
Access link
Saved status
Local data
Temporaries.

(1) **Temporaries:-** Arising from expressions, whose values Cannot be held in registers.

Scanned by CamScanner

(ii) Local data:- belongs to the procedure.

(iii) Saved status:- Information about system before call to procedure. Contains return addresses, registers,

(iv) Access link:- Needed to locate data needed by called procedure, found in another record.

(v) Control link:- Pointing to caller's activation record.

(vi) Returned values:- returns of called functions

(vii) Actual parameters:- only virtual, actually placed in registers

3) Calling Sequences:- used to implement Procedure calls.

- Contains code that allocates an activation record.

→ Principles for Designing Calling Sequence:-

1) Communicated values between Caller & Callee, are placed at the beginning of callee's activation record.

2) Fixed length items are generally placed in middle.

3) Dynamic size items are placed at the end of AR.

4) Locate the top-of-stack judiciously. Place a pointer to the end of fixed length fields.

Parameters, returned values
Control link
Temporaries

} Caller & Callee's Responsibility, AR.

Scanned by CamScanner

=) Caller's responsibility:-

1. Caller evaluates actual parameters.
2. Caller stores a return address and old value into activation record., Increments the old value.

=) Callee's responsibility:-

1. Callee Saves register values, Status information
2. Callee initializes its local data, begins execution

RETURN SEQUENCE

Caller:- Knows the return value, and may use that value

=) Callee:- - places return value to the next of parameters.
- restores registers, old value and branches to the return address.

4. Variable length data on The Stack:-

- procedure p has three local arrays: a, b, c who are dynamic.
- only a pointer to the beginning of each array appears in activation record

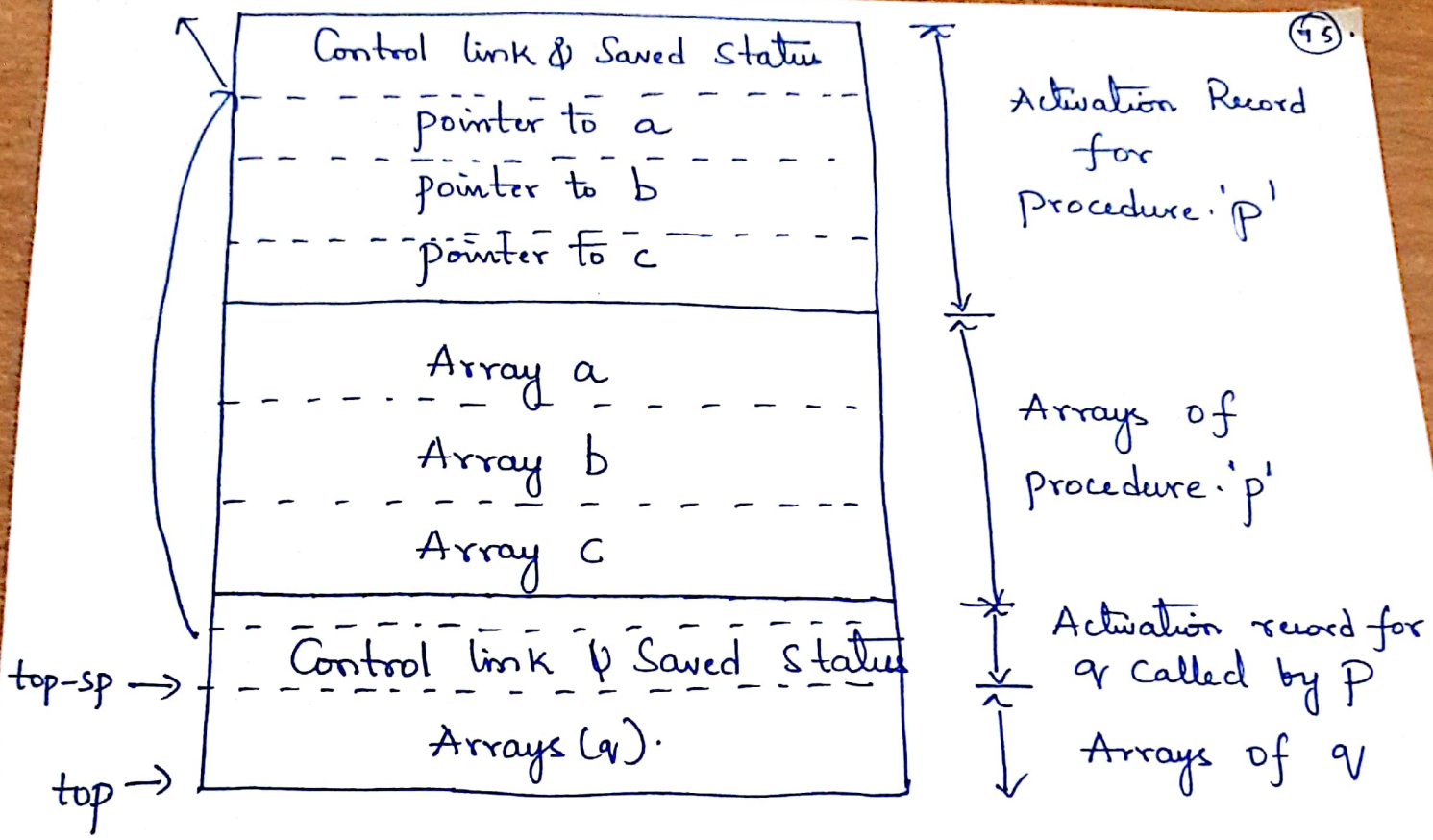
- Two pointers:-

(i) top: points to actual top of stack

(ii) top_sp: points to / uses to find local fixed-length fields of the top activation record.

- procedure 'p' calls procedure 'q', and q's AR begins after a, b, c.

Scanned by CamScanner



Scanned by CamScanner

⇒ **Peephole optimization:-**

- Examining a sliding window of target code, and replacing instructions within that 'peephole' by a shorter & faster sequence.
- can be applied immediately after ICG.

1. **Elimination of Redundancies:-**

e.g: LD R0, a
ST a, R0 → can be eliminated.

2. **Elimination of unreachable code:-**

- unlabeled instruction, following immediately an unconditional jump (unreachable code).
- e.g: if debug ==1 goto L1
goto L2
L1: Print debugging information

L2:

→ **Replacement:-**

→ if debug !=1 goto L2.
Print debugging information

- if (debug is 0), then

→ if 0 != 1 goto L2
Print debugging information

Scanned by CamScanner

3. Flow-of-control optimization:-

(i) goto L1 → goto L2
L1: goto L2 → L1: goto L2
↳ can be even eliminated, if no jumps on L1.

(ii) if a<b goto L1 → if a<b goto L2
L1: goto L2 → L1: goto L2

(iii) goto L1 → if a<b goto L2
L1: if a<b goto L2 → goto L3
L3: → L3:

4. Algebriac Simplification (1) - Strength Reduction:- (2)

(1) → e.g: x = x+0, x = x*1 } 3AC can be avoided.

(2) → e.g: x^2 = x*x - choose R.H.S
x/2 = x*0.5 - choose R.H.S

5. use of Machine Idioms:-
- Identify Machine's strength for faster access.
- e.g: Some machines having auto-increment (or) decrement modes.

Scanned by CamScanner