28 Compiler Design

~ B. prasad sir

- Basic of Compiler
- lexical Analysis
- Syntax Analysis
- Syntax Directed translation
- Intermediate code generation
- code optimization
- Run time Environment.

Reference book:-

- Ullman

Marks:-

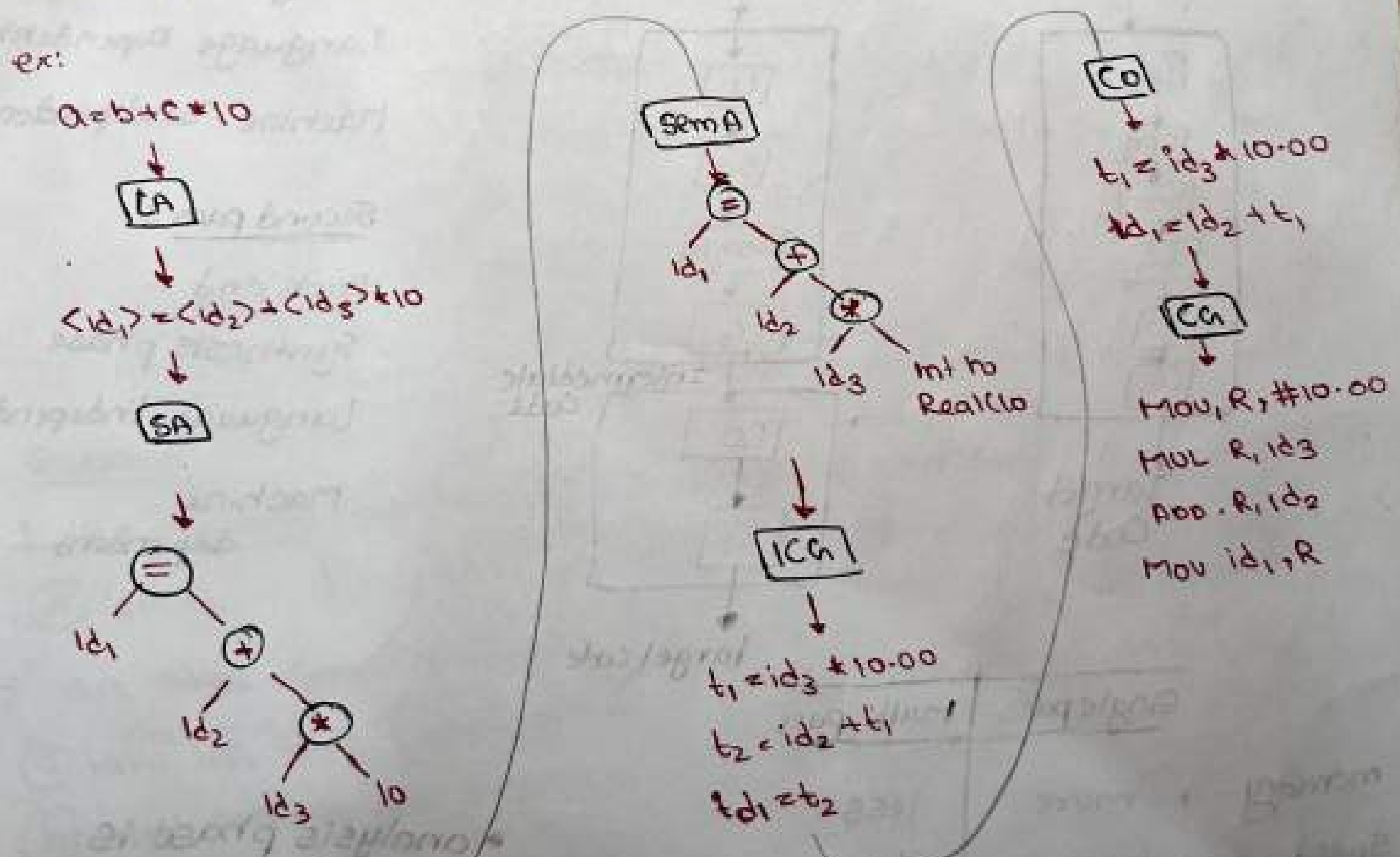
$\boxed{4 \text{ to } 9}$

- Lexical analysis
  - Functions of Lexical Analyser
  - Lexical Error.
  - Error Recovery Methods
  - Finding the no. of Tokens.
  - Lex Tool.

- Lexical analysis
- take the source code as i/p & divide it into tokens
  - Check wheather the code is able to divide into token (only divid the token not count)
- Semantic analysis
  - Checking of the meaning of source code is done or start from these phase
- the semantic analysisor take parse tree as i/p & produces Annotated parse tree as o/p
- Code optimization
- In these no. of instn will be reduced if possible without affecting the o/p of the program

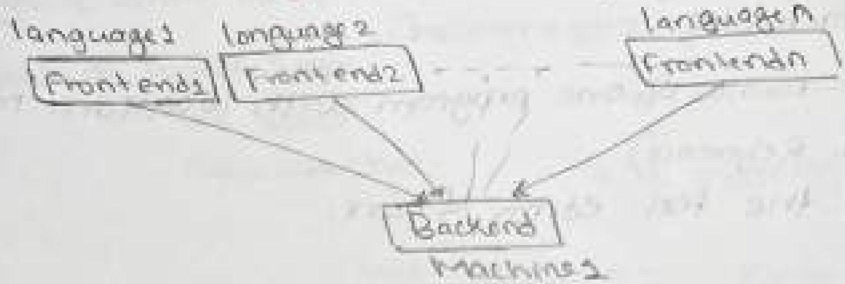
- Syntax Analysis
- the token's produce by lexical Analysisor will be given as i/p to the syntax analysisor to produce a parse tree
  - Check wheather the code is able to divide into sytax
- Intermediate code generation
- it take annotated parse tree as i/p & produces Intermediate Code as o/p
- Code generation
  - finally the target code will be generated by code generation

ex:

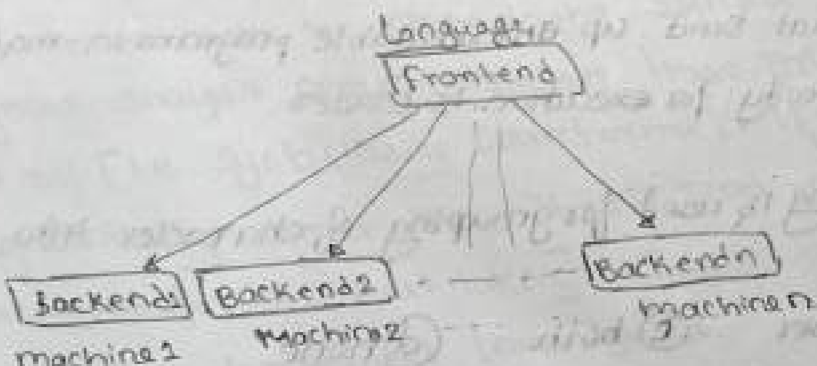


- using multipass compiler we can solve the following 2 problem

1. Suppose we want to design a compiler for different languages which work of same machine then we design different frontend for different language but only one backend for corresponding machine



2. Suppose we want to design a Compiler for some language which works on different machine then we design the frontend for the given language & different backend for different machine



Question

1. the keyword of a language are recognized during
   (a) lexical (b) syntax (c) semantic (d) All

2. an ideal compiler should be
   (a) take less time for compilation
   (b) must produce an object code which execute faster.
   (c) small in size
   (d) All of these

Lexical Analysis

• It scan every character of the source code & the following will be done by it. (also known as lexer or scanner).

1. divides into token
2. Ignores comments.
3. Ignore white space char.

   Such as blank space.
   Tab space
   new line char

4. It counts the no. of lines in the programs
5. It create a symbol table
6. It produces lexical error with line no. & column no. if any.
7. It interacts with Syntax Anaylsis

• the lexical analyser scan every character of the source code & [the token will be given to the syntax analyser then the syntax analyser check wheather those tokens are will formed or not [i.e syntactically correct or not]. If the tokens are syntactically correct then it will send a request to the lexical analyser for the next token]

• Until it get the request from syntax analyser Laxical analyser will not generate a token.

Why should we separate. Laxical analyser & syntax analyser

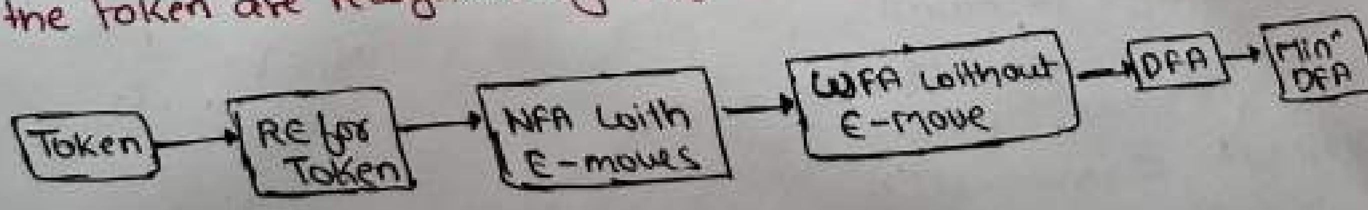
1. To simplify the design of compiler.
2. To increase the efficiency of compiler
3. To increase the portability of compiler

• **note**

the program used in Lexical analysis in FA

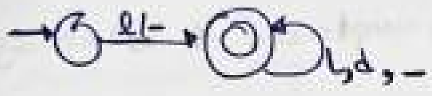
the program used in Lexical analysis using FA.

i.e the token are recognized by Regular Expressions using FA.

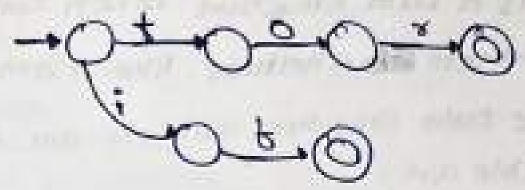


• Recognization of Identifier.

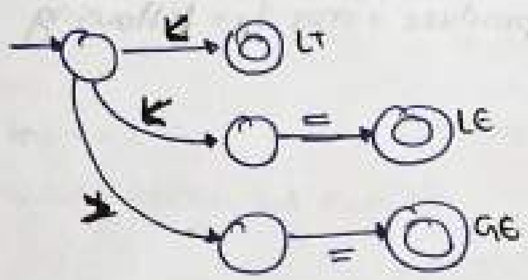
$id \rightarrow (l/\_)(l/d/\_)^*$



• Recognization of Keywords.



• Recognization of Relational operators.



• Panic Mode error Recovery:

ex: Forrr — identifier; For — Keyword

• In these technique the successive group of character will be remove/delete from the remaining i/p string until it get the well formed token

For~~rr~~

Lexical error.

✓ the scanned group of characters if it is not matching with any of the rules of the table then the Lexical analyser produces lexical error.

• unterminated comments → a=b+c; /* adding b & c

• unmatched string → a="Bhopal"; missing

• Invalid Identifiers;

• Invalid Constant

ex: int a = 2.53; ← no lexical error.

int a = $10; ← invalid constant.

• Exceeding the length of the identifier.

Delete: it deletes one single character from the remaining i/p str. Ex:- Fo~~r~~r → for

INSERT: It insert the missing character
Ex: prntf → printf

REPLACE: It replace one character by another character
Ex: ~~P~~nt → int

TRANSPOSE: It transpose two successive character.
ex:- itn → int

• Symbol table.

• It is a Data Structure which contain the information about the variable & there attribute. like. name, value, size, locatn

• the Data Structure which are used to implement the symbol table are

• Hash table • Binary search tree • Linear list

Question.

1. find wheather the lexical analyser can produce error for following Statements.

① $\underset{id}{\underline{iit}}$ $\underset{id}{x}$,y; no lexical error.

② Fi(a>b
a=a1 -
else
b=b+1

no lexical error

③ Fro(i=0, i<n; i+-) ← no lexical error

④ $\underset{id}{\underline{printf}}$("Bhopal is capital of India"); ← missing
lexical error (Unmatch string)

⑤ a= b+cd* ; /* Big lake ← unterminated comment

⑥ int 1a, 1b; ← invalid identifier

⑦ int a=079; ← invalid constant [Any no. starting with '0' will be treated as octal no.]

⑧ int a=0b12 ← invalid const. Any no. starting with '0b' will be treated as binary no.

⑨ int a= 0xaAG;
invalid Constant
octal no.

Q] find the no. of token produce by laxical analyser for following Statements.

1. int a,b; 5
2. int aaa; 3
3. int aaa ; 3
4. int a a a; 5

- lexical error
- Here are some common errors.

① invalid characters.
ex: x = @

② (unmatched) unterminated string.
ex: String str = "millin"; — " missing

③ unterminated comment
ex /* not closed */

④ Exceeding the length of the identifiers.
max. length.
suppose C lang"
max. is 32 characters.

⑤ unrecognized Token
ex int 42x = 10;
not valid token
↳ when the lexical analyzer encounter a sequence of characters that do not form a valid token.

⑥ Invalid identifier.
ex: int 2x=5
identifier can't start with digit

⑦ Illegal escape characters
• message = "Mi\lin";
occurs when an escape sequence is used incorrectly in a string.

⑧ Missing operator or operand.
ex: int x = 5 + ;
missing opera nd after "+".

---

11 a<=b; 6

12 a<<=b; 9

whitespace is ignored

10, 20 (number)
considered as 1 token

▷ * *
considered different token not 1

- Syntax Analysis
- Ambiguity
- Left Recursion & Right recursion
- left factor
- Parsing techniques

▸ the token produce by lexical analyzer will be given as i/p to the syntax analyser. the syntax analyzer check wheather the token syntactically correct or not. if the token are syntatic correct then it will construct a parse tree for this token.

▸ this whole will be done by using context free grammer.

the program used in syntax analysis is PDA

the [illegible] we do can be represented as tree,

- Common error in Syntax analyzer (parser)

① Syntax Error/missing semicolon

ex if(x>5){...};

missing.

② Missing closing parenthesis

ex: int sum =(3+5);

missing.

③ unexpected token

ex: int x=*5;

unexpected

④ missing operator

ex: int sum=3 + * 5

missing operator after "+"

⑤ invalid statment placement

ex: int x=5;

x+3

expression w/o assignment or print stmt

b=b+!

③ For(i=0, i<n;i+-) ← no lexical error

④ printf("Bhopal is capital of India"); ← missing lexical error (unmatch string)

id

⑤ a= b+cd* ; /* Big lake ← unterminated comment

⑥ int 1a, 1b; ← invalid identifier

⑦ int a=079; ← invalid constant [Any no. starting with '0' will be treated as octal no.]

⑧ int a=0b12 ← invalid const. Any no. starting with '0b' will be treated as binary no.

⑨ int a= 0x9AG;

↓ octal no.

invalid constant

Q] find the no. of token produce by lexical analyser for following

5. if (a>b)
 a = a++
else
 b = b--

15

6. for (i=0; i<=n; i++); 13

7. printf ("MadeEasy"); 5

8. printf("%d %d", a, b); 9

9. printf("%d", *a); 8

10. a = b++++----d; 9

11. a<<b+c; 6

12. a<<=b; 4

13. b=a+*c---; 9

14. a>>=b; 4

15. if(a>b)
 a=a+b /* Adding nos
 b=b c;
 d=b+a; */
 b=b-a;

17

▸ Variable name is also consider as 1 token

note:-
<<= : 1
>>= : 1
++ : 2
"String" = 1
/* aabb */ = 0
whitespace is ignored
10, 20 (number) consider as 1 token

▸ * *
 4 4
Consider different token not 1

• Syntax Analysis
• Ambiguity
• Left Recursion & Right recursion
• Left Factor
• Parsing techniques.

▸ the token produce by lexical analyzer will be given as i/p to the Syntax analyser. the syntax analyzer check wheather the token syntatically correct or not. if the token are syntatic correct then it will construct a parse tree for this token.

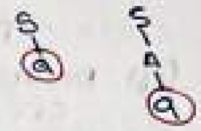
▸ this whole will be done by using context free grammer.
 the program used in syntax analysis is PDA

▸ The derivation what ever we do can be represented as tree, Such tree are called as derivation tree or parse tree.

- Note: there is no algorithm to decide wheather a grammer is ambiguous or not. there can be done only by try & error method

note: if an grammer is ambiguous than it is not suitable for any kind of parsing except backtracking & operator precedence parsing

ex: $G = \{S \to A/a, Aa\}$



∴ a is ambiguous

// The grammer is ambiguous

- if we want to apply the remaining parsing technique also then we must eliminate the ambiguity from the grammer but there is no algorithm to eliminate the ambiguity from the grammer.

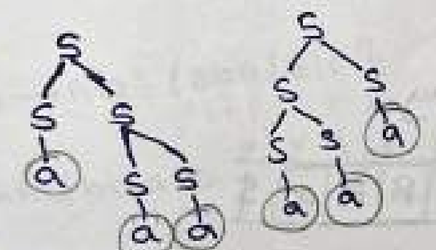
ex: $G = \{S \to A/a,\ \cancel{A \to a}\}$ is ambiguous

$L(G) = a$

$G' = \{S \to a\}$ is unambiguous $\Rightarrow$

note:-
- A grammer which is both left & right recursive is known as Ambiguous.

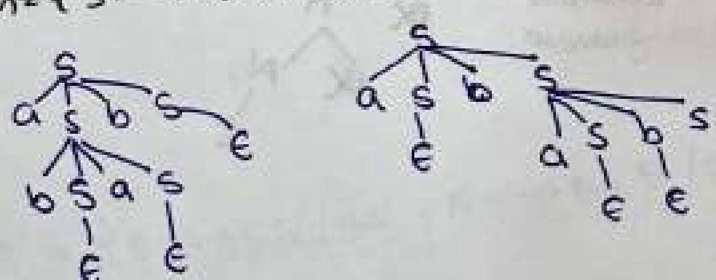
ex:- $G = \{S \to SS/a\}$ $\Rightarrow$



$L(G) = aaa$

$G' = \{S \to aS/a\}$ is unambiguous

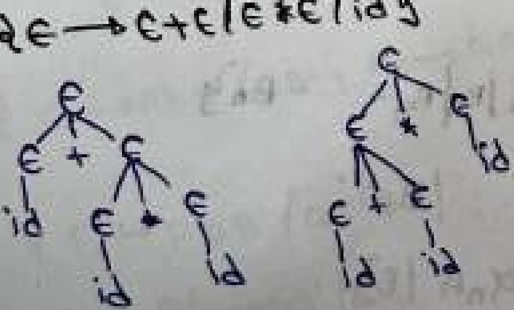
Q) $G = \{S \to aSbS \,/\, bSaS \,/\, \epsilon\}$ $\Rightarrow$ $G' = A/S/\to$ ambiguity can not be removed.



Q) $G = \{E \to E+E \,/\, E*E \,/\, id\}$ $\Rightarrow$ $G' = \{E \to E+T/T$

$T \to T*F/F$

$F \to id\}$ is unambiguous.



is ambiguous

Recursive CFG

Left Recursive CFG

i.e $\underline{A} \to \underline{A}\alpha / \beta$

Right Recursive CFG.

i.e $\underline{A} \to \alpha\underline{A} / \beta$

Eliminating left Recursion CFG.

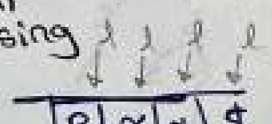
$G' = \{ A \to \beta A'$
$A' \to \alpha A' / \epsilon \}$

↳ In the case of Topdown parsing if the grammer is left recursive then it may fall into infinite loop, therefore we must eliminate left recursion from the grammer.

↳ to eliminate the Left recursion we must convert it into Right recursive grammer.

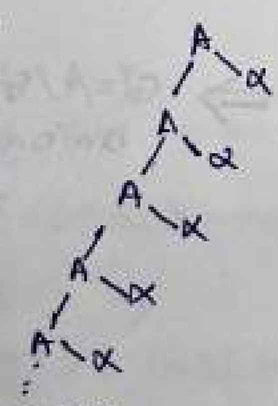
Start
↓
String

Top down parsing



$G = \{ A \to A\alpha / \beta \}$

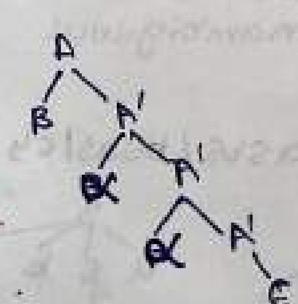
$A \to A\alpha$
$\to A\alpha\alpha$
$\to \beta\alpha\alpha$



infinite loop

eliminate →

Left recursive grammer.



$G = \{ A \to A\alpha_1 / A\alpha_2 / A\alpha_3 \cdots \cdots / A\alpha_n / \beta_1 / \beta_2 / \beta_3 \cdots \beta_n \}$

$G' = \{ A \to \beta_1 A' / \beta_2 A' / \beta_3 A' \cdots \cdots \cdots / \beta_n A'$

$A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \cdots \cdots / \alpha_n A' / \epsilon \}$

ex: eliminate left recursion from following grammer

1. $G = \{S \to Sa \mid b \mid a\}$

$G' \Rightarrow \{S \to bS' \mid aS'$
$S' \to aS' \mid \epsilon\}$

2. $G = \{S \to Sa \mid Sbb \mid aa \mid b \mid \epsilon\}$

$G' \Rightarrow \{S \to aaS' \mid bS' \mid S'$
$S' \to aS' \mid bbS' \mid \epsilon\}$

3. $G = \{S \to Ab \mid aSb \mid a,\ A \to Abs \mid sbA \mid b \mid \epsilon\}$

one direct left Recursion

& 2 more indirect left Recursion

$S \to Ab$ $A \to SbA$
$S \to SbAb$ $A \to AbbA$

$G' = \{S \to Ab \mid aSb \mid a',\ A \to \underbrace{Abs}_{\alpha_1} \mid \underbrace{AbbA}_{\alpha_2} \mid \underbrace{aSabbA}_{\beta_1} \mid \underbrace{abA}_{\beta_2} \mid \underbrace{b}_{\beta_3} \mid \underbrace{\epsilon}_{\beta_4}\}$

$G'' \Rightarrow \{S \to Ab \mid aSb \mid a,\ A \to aSbbAA' \mid abAA' \mid bA' \mid A'$
$A' \to bSA' \mid bbAA' \mid \epsilon\}$

4. $G = \{S \to Sa \mid Ab \mid a,\ A \to Aas \mid sbA \mid b \mid \epsilon\}$

~~$G' = \{S \to Sa \mid AaSb \mid SbAb \mid bb \mid a \mid b,\ A \to AaS \mid SabA \mid AbbA \mid abA \mid b \mid \epsilon\}$~~

~~$G'' = \{S \to AaSbS' \mid bbS' \mid aS' \mid bS',\ A \to SabAA' \mid abAA' \mid bA' \mid A'$~~
~~$S' \to aS' \mid bAbS' \mid \epsilon$ $A' \to aSA' \mid bbAA' \mid \epsilon\}$~~

as both as direct & indirect than 1st solve one & then solve other.

(there may be possibility that if you subsititude any one then you may get more state, so go for minimized option)

$G' = \{S \to AbS' \mid aS';\ A \to AaS \mid SbA \mid b \mid \epsilon,$
$S' \to aS' \mid \epsilon\}$

$G'' = \{S \to AbS' \mid aS',\ A \to AaS \mid AbS'bA \mid aS'bA \mid b \mid \epsilon,\ S' \to aS' \mid \epsilon$

$G''' = \{S \to AbS' \mid aS'$, $A \to aS'bAA' \mid bA' \mid A',$
$S' \to aS' \mid \epsilon$ $A' \to aSA' \mid bS'bAA' \mid \epsilon\}$

- Parsing
  - Top down parsing — Left most derivation
    - with Backtracking (Brute force technique)
    - without Backtracking (Predictive parsing)
      - Recursive Desent Parsing
      - non-recursive Desent parsing (or) LL(1) parsing.
      - both should not be
        1. ambiguous
        2. Left recursive
        3. left factor or non-Deterministic
  - Bottom up parsing. (Shift - Reduce Parsing) — It generates Right most derivation in reverse order.
    - Operator precedence parsing
    - LR(K) parsing.
      - LR(0)
      - SLR(1)
      - LALR(1)
      - CLR(1) | LR(1)

• parsing conflict can be resolved using both precedence & associativity of operators.

ex: $A \rightarrow abs$

two adjacent terminals in the RHS of a production will have "=" relation & in precedence relation a=b never be same as b=a

Parsing (all grammer will be CFG)

Top down parsing

- From Root to leaves
- Uses LMD
- Uses derivation process

$\underline{S}$
$\underline{A}B$
$a\underline{A}B$
$aa\underline{B}$
aab

$$G = \{ S \rightarrow AB,\ A \rightarrow aA/a,\ B \rightarrow bB/b \}$$

Bottom up parsing

- From leaves to Root
- use RMD in reverse
- Uses reduction process.

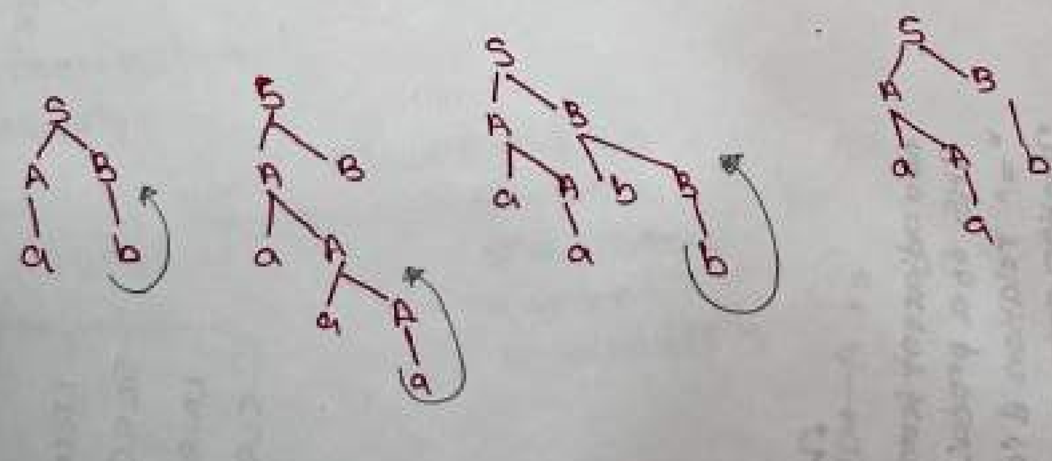
$$S \rightarrow AB,\ A \rightarrow aA/a,\ B \rightarrow bB/b$$

aab
$a\underline{A}b$
$\underline{A}b$
$A\underline{B}$
$\underline{S}$

- Brute force technique. [with backtracking]
- there are no rules & regulation in these technique.
- Backtracking is a process of trying with different production until if we get the required string.
- if the grammer is having so many production & the length of the string is too large then the parser has to go for several trials b/c of these several trials it take more time.
- Hence we don't prefer backtracking

$G = \{ S \rightarrow AB,$
$A \rightarrow aA/a,$
$B \rightarrow bB/b \}$

l
$\downarrow$
aab$



• Recursive Desent parsing:

• It is a program consists of set of produre. one for each non-terminal

The execution begins with the procedure of the start symbol. then halt & annonces that the successful parsing has done if its procedure body scans the entire i/p string.

ex: $A \rightarrow X_1 X_2 X_3 \cdots\cdots X_n$ where each X_i is either a terminal or non-terminal

```
A(){
  for(i=1 to k){
     if(Xi is a non-terminal)
        Xi();
     elseif(Xi is matched with current i/p symbol)
        Advance the current i/p symbol
     else
        error;
  }
}
```

• it should not contain left fact & left recussion

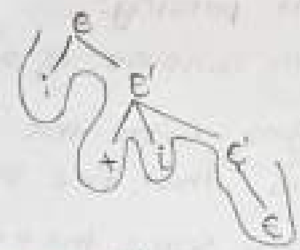
Ex: $G = \{E \rightarrow iE',$
$E' \rightarrow +iE' \mid \epsilon\}$

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

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

```
match(char t){
  if(l==t)
    l=getch();
  else
    printf("error");
}
```

```
main(){   ← calling
  E();    ← called
  if(l=='$')
    printf("success");
  else
    printf("error");
}.
```



Note —.

- an ambiguous grammer is not suitable for recursive decent parser.
- A left recursive grammer can't be parsed using recursive decent parsing b/c the parser may fall into "∞" loop
- A left factor grammer can't be parsed using recursive decent parsing b/c it may given an ~~error~~ msg even for valid string.

- **First Set**

First(A) is the set of terminals that are the 1st symbols on the right side of the arrow. in every production of A

Procedure :-

1. first (a) = {a}
2. if A → a → First(A) = {a}
3. if A → ε → First (A) = {ε}

A → BCDE, B → aB/b/ε , C → cC/a

First(A) = First(B)

⇒ {a,b,ε} - {ε} ∪ First(C) - {ε} ∪ First(D) - {ε} ∪ First(E)

4. If A → BC → First(A) = First(B) if First(B) doesn't contain 'ε'

5. If A → BC → first (A) = First (B) - {ε} ∪ first(C)

• find the first set of each non-terminal of the following grammar.

ex: $G = \{S \to ABC, A \to aA / b, B \to bB / \epsilon, C \to c\}$

First(S) = {a, b}
First(A) = {a, b}
First(B) = {b, ε}
First(C) = {c}

ex: $G = \{S \to ABC,$
$A \to aA / \epsilon,$
$B \to bB / a,$
$C \to cA / a\}$

First(S) = {a, b}
First(A) = {a, ε}
first(B) = {b, a}
first(C) = {c, a}

ex: $G = \{S \to AbcC,$
$A \to Ba / b,$
$B \to dB / \epsilon,$
$C \to Bc / a\}$

First(S) = {a, b, d}
First(A) = {d, a, b}
First(B) = {d, ε}
First(C) = {a, d, c}

ex: $G = \{S \to ABC / Ba / Cb,$
$A \to da / BC$
$B \to g / \epsilon,$
$C \to h / \epsilon\}$

First(S) = {a, b, d, g, h, ε}
First(A) = {d, g, h, ε}
First(B) = {g, ε}
First(C) = {h, ε}

ex: $G = \{S \to ABb / CA,$
$A \to Ba / Cc / \epsilon,$
$B \to AB / \epsilon,$
$C \to aC / Ab / \epsilon\}$

First(S) = F(A) ∪ F(C) {a, b, c, ε}
First(A) = {ε} ∪ F(B) ∪ F(C) {a, b, c, ε}
First(B) = {ε} ∪ F(A) = {a, b, c, ε}
First(C) = {a} ∪ F(A) ∪ {ε}
b
{a, b, c, ε}

Q] $G = \{S \to AaB / bc,$
$A \to BC / bA,$
$B \to aB / \epsilon,$
$C \to a / SA\}$

First(S) = {b} ∪ F(A) = {a, b}
First(A) = {a, b} ∪ F(C) = {a, b} ∪ {a, b} = {a, b}
First(B) = {a, ε}
First(C) = {a} ∪ F(S) = {a} ∪ {a, b} = {a, b}

ex: $G = \{S \to AaSb / BS,$
$A \to Sc / \epsilon,$
$B \to bB / dS / \epsilon\}$

First(S) = {a, b, d}
First(A) = {ε, a, b, d}
First(B) = {b, d, ε}

Q) $G = \{ S \to SA/a/\epsilon,$
$A \to ABb/\epsilon,$
$B \to cB/d/\epsilon \}$

First(S) = {a, b, c, d, ε}
First(A) = {c, d, b, ε}
First(B) = {c, d, ε}

If there is interdependent b/w the non-terminals then solve partly part.
- whatever you have put that & solve the other & after getting the value. Solve it again.

Q) $G = \{ E \to TE'.$
$E' \to +TE'/\epsilon$
$T \to FT'$
$T' \to *FT'/\epsilon$
$F \to (E)/id$

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

Follow Set

- Follow(A) is the set of terminals that present immediately to the right side of 'A' whenever 'A' is present on the RHS of the arrow.

{S → Sa/b}   {S → aS/b}

Follow(S) = {a, $}   Follow(S) = {$}

Procedure:-

1. Follow(S) = {$}
2. If $A \to \alpha B \beta \to$ Follow(B) = First(β) · if First(β) doesn't contain 'ε'
3. If $A \to \alpha B \beta \to$ Follow(B) = First(β) − {ε} ∪ Follow(A)
4. If $A \to \alpha B \to$ Follow(B) = Follow(A)

ex: $G = \{ S \to AaAb/BbBa,$
$A \to AA/\epsilon,$
$B \to bB/\epsilon \}$

Follow(S) = {$}
Follow(A) = {a, b}
Follow(B) = {a, b}

Ex: $G = \{ S \to ABC,$
$A \to aA/\epsilon,$
$B \to bB/c,$
$C \to cC/\epsilon \}$

Follow(S) = {$}
Follow(A) = {b, c}
Follow(B) = {c, $}
Follow(C) = {$}

ex: $G = \{S \rightarrow ABC \mid Cb \mid Ba$

$A \rightarrow dc / BC,$

$B \rightarrow g \mid \epsilon,$

$C \rightarrow h / \epsilon\}$

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

Follow(A) = $\{g, h \$\}$

Follow(B) = $\{a, h, \$, g\}$

Follow(C) = $\{g, h, \$, b\}$

Q) $G = \{S \rightarrow aS / ABC,$

$A \rightarrow Sb / \epsilon,$

$B \rightarrow AaB / c,$

$C \rightarrow cB / \epsilon\}$

Follow(S) = $\{\$, b\}$

Follow(A) = $\{a, c\}$

Follow(B) = $\{c, \$, b\}$

Follow(C) = $\{\$, b\}$

Q) $G = \{S \rightarrow ASB / aS / BC,$

$A \rightarrow SaB / cbB,$

$B \rightarrow CA / bCd,$

$C \rightarrow BC / a\}$

first(S) = $\{a, b\}$
-- " -- (A) = $\{a, b\}$
-- " -- (B) = $\{a, b\}$
-- " -- (C) = $\{a, b\}$

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

Follow(A) = $\{a, \$, b\}$

Follow(B) = $\{\$, b, a\}$

Follow(C) = $\{\$, b, a, d\}$

ex: $G = \{S \rightarrow aSb / SAa / a,$

$A \rightarrow BC / \epsilon,$

$B \rightarrow AaB / b\epsilon,$

$C \rightarrow cC / Ad / \epsilon\}$

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

Follow(A) = $\{a, d\}$

Follow(B) = $\{c, a, d \$ b\}$

Follow(C) = $\{a, d, c \$ b\}$

ex: $G = \{S \rightarrow S + A / A,$

$A \rightarrow A * B / B,$

$B \rightarrow (E) / id,$

$C \rightarrow aCa / b\}$

Follow(S) = $\{\$, +\}$

Follow(A) = $\{\$, +, *\}$

Follow(B) = $\{\$, +, *\}$

Follow(C) = $\{), a\}$

LL(1) parsing

* "ε" is not a terminal

LL(1)
- Look ahead symbol of length 1.
- use LMD
- left to Right Scan of the i/p string.

• construction of LL(1) parsing table

we consider ever production of the form "$A \to \alpha$" then proceed as follows.

1. we add $A \to \alpha$, in $M[A,x]$ where $x \in \text{first}(\alpha)$

   M: table name

   x: x^{th} column

ex: $S = \{S \to AB,\ A \to aA/b,\ B \to bB/\epsilon\}$

① we add $A \to \alpha$ in $M[A, x]$ where $x \in \text{First}(\alpha)$

$\boxed{S \to AB}$

• $M[S,x]$. $x \in \text{First}(\underline{A}B) = \{a, b\}$

[S,a]

[S,b]

• $A \to b$

$M[A,x]$

$x = \text{first}(\underline{b}) = \{b\}$

• ~~$B \to \epsilon$~~

~~$M[B,x]$~~

~~$x = \text{first}(\epsilon) = \{\epsilon\}$~~

• $A \to aA$

$M[A,x]$

$x = \text{first}(\underline{a}A) = \{a\}$

• $B \to bB$

$M[B,x]$

$x = \text{First}(\underline{b}B) = \{b\}$

$B \to \epsilon$

$M[B,y]$

$y = \text{follow}(\underline{B}) = \{\$\}$

A/c to 2nd point

M:

	a	b	$
S	$S \to AB$	$S \to AB$	Syntax error
A	$A \to aA$	$A \to b$	Syntax error
B	Syntax error	$B \to bB$	$B \to \epsilon$

In above table all the entries are single. ∴ the grammer is LL(1).

ii) If First $(\alpha) = \{\epsilon\}$ then add $A \rightarrow \alpha$ in $M[A, y]$ where $y \in \text{Follow}(A)$

iii) unfilled entires in the table are called as Syntax errors.

iv) If all the entires in the table are single then the grammer is LL(1).

Q] construct a LL(1) parsing table for the following grammer.

$G = \{E \rightarrow TE',\ E' \rightarrow +TE' / \epsilon,\ T \rightarrow FT',\ T' \rightarrow *FT' / \epsilon,\ F \rightarrow (E) / id\}$

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

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

follow
$E' = \{), \$\}$
$E = \{), \$\}$
$T' = \{+,), \$\}$
$T = \{+,), \$\}$
$F = \{*, +,), \$\}$

"—": Syntax error

All the entries in the table are single
$\therefore$ the grammer is LL(1).

• properties of LL(1) grammer.

1. A left recursive grammer is not LL(1)
2. A left factor grammer is not LL(1).
3. An ambiguous grammer is not LL(1)
4. Every LL(1) grammer is unambiguous but every unambiguous is not LL(1)
5. if any production are of the form $\boxed{A_1 \rightarrow \alpha_1 / \alpha_2}$ then if $\text{First}(\alpha_1) \cap \text{first}(\alpha_2) \neq \phi$ $\Rightarrow$ not LL(1).

$A \rightarrow \alpha_1 / \alpha_2$
if First(α_1) have some common in first(α_2) then it not LL(1)

6. If any production are of the form

$A \to \alpha_1 / \alpha_2 / \alpha_3 \cdots\cdots / \alpha_n$ then

if first set any two production having common elements then it is not LL(1)

7. If any productions are of the form $A \to \alpha / \epsilon$ then

First(α) ∩ Follow(A) ≠ ϕ

⇒ not LL(1)

- check whealther the following grammar is LL(1) or not.

1. $G = \{S \to Sa / b\}$

the grammar is left recursive

⇒ not LL(1).

First(Sa) ∩ First(b)

⇒ {b} ∩ {b} ≠ ϕ

is not LL(1)

2. $G = \{S \to Aab / a,$
$A \to bA / \epsilon\}$

first(Aab) ∩ First(a)

First(b, ~~ϵ~~, a) ∩ First(a)

{b, a} ∩ {a} ≠ ϕ

∴ not LL(1)

3. $G = \{S \to AbS / Aa / b,$
$A \to bA / a\}$

→ not LL(1) b/c left factor.

→ First(AbS) ∩ First(Aa)

{b, a} ∩ {b, a} ≠ ϕ.

4. $G = \{S \to aSbA / \epsilon,$
$A \to SaA / b\}$

- First(aSbA) ∩ Follow(S)

{a} ∩ {b, a} ≠ ϕ

- First(SaA) ∩ First(b)

{a} ∩ {b} = ϕ.

exceptn: $G = \{A \to Ba / Bb,\ B \to \epsilon\}$

Still these LL(1) Still after having left factor.

→ First(Ba) ∩ First(Bb)

{~~B~~a} ∩ {~~B~~b}

= ϕ

ex: $G = \{S \to dBSA / aS, Ab,$
$A \to cB / \epsilon,$
$B \to bB / \epsilon\}$

$\overbrace{\text{First}(bB) \cap \text{Follow}(B)}^{B}$

$\{b\} \cap \{d, a\} \cap \{c, b\} \neq \phi$

first(S) first(A)

& putting & solving

A: First(c) ∩ Follow(A)

{c} ∩ {b, c, \$} ≠ ϕ

not LL(1)

ex: $G = \{S \rightarrow iEtSS' / a,$

$S' \rightarrow eS / \epsilon,$

$E \rightarrow b\}$

S': First(eS) $\cap$ First(E) Follow(S') → Follow(S) → First(S') = {e}

{ e} $\cap$

$\Rightarrow$ {e} $\cap$ {e, \$} $\neq \phi$

So, it is not LL(1).

E: First(b) $\Rightarrow$ {b}

S: First(iEtSS') $\cap$ First(a)

{i} $\cap$ {a} = ϕ

ex: $G = \{S \rightarrow bSA / \epsilon, A \rightarrow aSaB / \epsilon, B \rightarrow b\}$

First(bSA) $\cap$ Follow(S)

{b} $\cap$ {First(A) $\cup$ {a}}

{b} $\cap$ {a, ~~ε~~, \$} = ϕ

First(aSaB) $\cap$ Follow(A).

{a} $\cap$ {Follow(S)}

{a} $\cap$ {a, \$} $\neq \phi$

	a	b	\$
S	$S \rightarrow \epsilon$	$S \rightarrow bSA$	$S \rightarrow \epsilon$
A	$A \rightarrow aSaB$ $A \rightarrow \epsilon$		$A \rightarrow \epsilon$

Find the entries of M[S, b] & M[A, a] in the LL(1) parsing table.

- $\boxed{LL(k)}$
  - At a time lookahead Symbols of length 'k'
  - LMD
  - Left to right

ex: $G = \{S \rightarrow aA / ab,$

$A \rightarrow a\}$

LL(1):- {a} $\cap$ {a}

= {a}

$\neq \phi$ [not LL(1)]

LL(2): a.a (a· First(A))

First(a$\boxed{A}$) $\cap$ First(ab)

{aa} $\cap$ {ab} = ϕ

LL(2)

∴ the grammer is LL(2) but not LL(1)

- which of the following is LL(3) but not LL(1) and not LL(2)?

a) {S → aS / b} LL(1) → {a} ∩ {b} = φ

b) {S → aaS / ab / b} LL(2)

c) {S → aaaS / aab / ab / b} LL(3).

d) none

→ note
- LL(1) ⊂ LL(2) ⊂ LL(3) - - - - ⊂ LL(k)

{aa} ∩ {ab} ∩ {b} = φ [LL(2)]

{aa} ∩ {aa}

{aa} ≠ φ not LL(2)

$LL_3 \Rightarrow$ {aaa} ∩ {aab} ∩ {ab} ∩ {b}

= φ

LL(3)

ω-B (Chapter 2)

17. $G = \{S \rightarrow \cancel{AB} / CA,$
$\quad A \rightarrow a,$
$\quad \cancel{B \rightarrow BC / AB},$ useless
$\quad \cancel{C \rightarrow aB} / b\}$

⇒ B is useless

S → AB
aB
aBC
aABC

$G' = \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\}$

- BOTTOM up parsing

$G = \{E \rightarrow E+E / E*E / id\}$

Handing Pruning

Right sentential form	Handle	Production
id * id * id	id	E → id
E + id * id	id	E → id
E + E * id	E+E	E → E+E
E * id	id	E → id
E * E	E*E	E → E*E
E		

Handle

- A Handle is a substring of a string that match with any of the Right side of production. then that handle will be reduce with left side of the production

E
E+E
E+E*E
id+E*E
id+id*E
id+id*id

Sentential form (any)

Sentence

If you are able to generate the string from start symbol

Handle purning

- Bottom of parsing is a process of finding the handle & using them in reduction to ge the Start symbol. these entire process of reducting the String to the start symbol is know as handle purning

- any String that can be derived from the start symbol is called as sentential form.
- If all the symbol form in sentential form are terminals then it is Called as Sentence.

• every sentence is a Sentential form but every Sentential form is not sentence

• Right sentence form

• A Sentence form that occurs in the right most derivation of some sentence, is called right sentence form

• Left Sentence form

• A sentence form that occur's in the left most derivation of some sentence is called left sentence form

• Shift-Reduce parsing:

• in these we perform 4 action
[Shift, Reduce, Accept, error]

• in these we use 2 Data Structure

1. I/p Buffer :- it is used to store the I/p String.

the initial configuration of the I/p buffer is w$

a	b	a	b	b	$

2. Stack :- used to store the symbol of the grammer.

the initial Configuration of the stack is $ or $

• Consider the following grammer. and parse the I/p string id+id*id using shift-reduce parsing.

Stack	input	Action
$	id+id*id $	Shift
$id	+id*id $	reduce by E→id
$E	+id*id $	Shift
$E+	id*id $	Shift
$E+id	*id $	reduce by E→id
$E+E	*id $	Reduce by E→E+E
$E	*id$	Shift
$E*	id $	Shift
$E*id	$	Reduce by E→id
$E*E	$	Reduce by E→E*E
$E	$	Accept.

last one is Accept not shift

• Viable prefixes :-

• Viable prefixes are the set of prefixes of a right sentential form that appear on the top of the stack of a shift-Reduce parser.

• Set of valable prefixes of CFL is a regular language.

• consider the following grammer.

Q) G = {S → CC, C → aC/b}
which of the foll owing are vaiable prefix?

(a) ab (b) Cb ~~(c) bb~~ (d) Ca ~~(e) ba~~

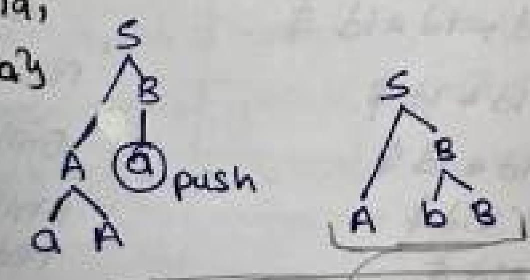
// we can't push non-terminal.

Stack	I/p Buffer	Action
$	____$	Shift
⋮	⋮	
$ab	____$	C → b
$ aC	__$	C → aC
$ C	b$	push b
$ Cb	$	C → b.
$ CC		
$ S		

bcb
bcc
bS

Ca
Cab
CaC
CC
S

Q) G = {S → AB,
A → aA/a,
B → bB/b}

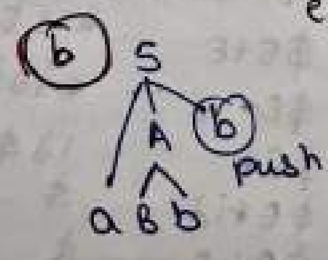


a) aA
b) aab
c) AbB
d) aaB
e) ABb

here you can either open A or B. but only one at a time

Q) G = {S → aAb/bS,
A → Bb/a,
B → a}

a) bbA.
b) aBb
c) bbaA
d) bb
e) Aab



• consider the following grammer & relational table b/w the operator. pass the i/p string id*id+id using operator precedence parsing.

highest precedence is 'id' & least precedence is '$'

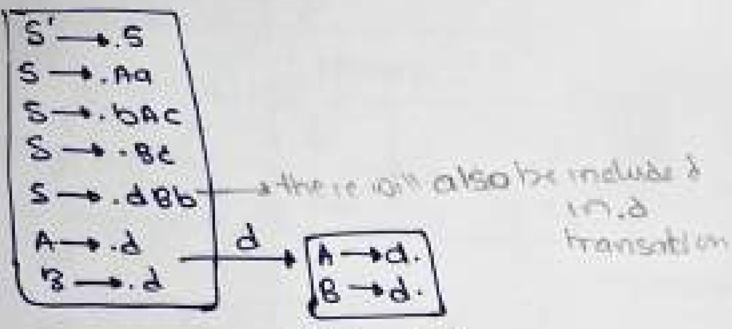
G = {E → E+E / E*E / id}

	id	*	+	$
id	–	⋗	⋗	⋗
*	⋖	⋗	⋗	⋗
+	⋖	⋖	⋗	⋗
$	⋖	⋖	⋖	–

Stack		I/p	Buffer
$	⋖	id*id+id$	Shift
$id	⋗	*id+id$	Reduce by E→id
$E	⋖	*id+id$	Shift
$E*	⋖	id+id$	Shift
$E*id	⋗	+id$	Reduce by E→id
$E*E	⋖	+id$	Shift
$E*E+	⋖	id$	Shift
$E*E+id	⋗	$	Reduce by E→id
$E*E+E	⋗	$	by E→E+E
$E*E	⋗	$	by E→E*E
$E		$	Accept.

• LR(k) parsing
- L → left to Right scan of the i/p string.
- R → Use RMD in rewrite
- (k) → Look Ahead symbol of length 'k'

Q) G = { S → Aa / bAc / Bc / dBb, A → d, B → d }

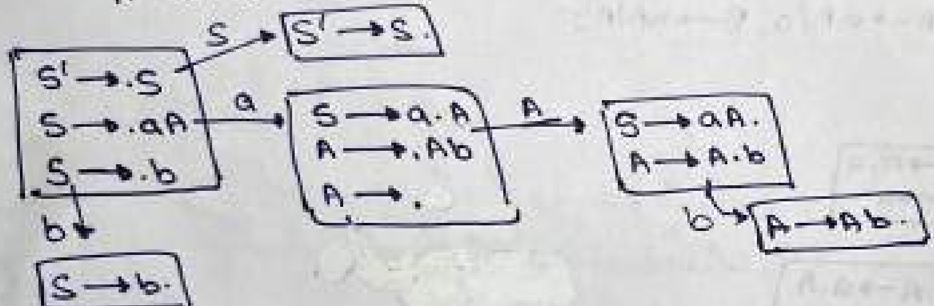


{d} ∩ {d}

not SLR(1)

RR conflict.

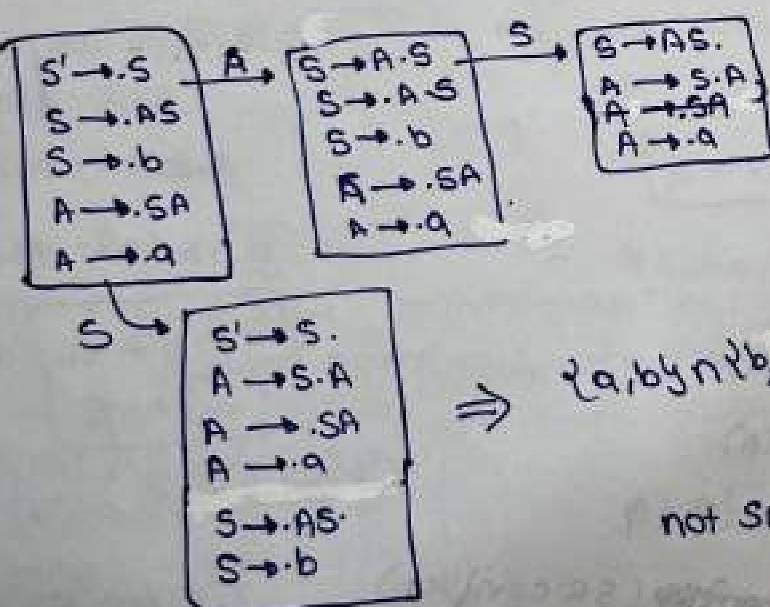
Q) G = { S → aA | b,
A → Ab | ε }



∴ It is SLR(1)

no conflict

Q) G = { S → AS | b,
A → SA | a }



⇒ {a, b} ∩ {b, a} ⇒ {b, a} ≠ φ

not SLR (SR conflict)

Q) $G = \{S \to Bb / aA,\ A \to Sa/\epsilon,\ B \to b\}$

$$\boxed{\begin{array}{l} S' \to .S \\ S \to .Bb \\ S \to .aA \\ B \to .b \end{array}} \xrightarrow{a} \boxed{\begin{array}{l} S \to a.A \\ A \to .Sa \\ A \to . \\ S \to .Bb \\ S \to .aA \end{array}}$$

Follow(A) $\cap$ $\{a\}$

$\{a, \$\} \cap \{a\} \neq \phi$

SR conflict.

Q) $G = \{S \to SaA / Ab,\ A \to BaA/a,\ B \to aS/\epsilon\}$

$$\boxed{\begin{array}{l} S' \to .S \\ S \to .SaA \\ S \to .Ab \\ A \to .BaA \\ A \to .a \\ B \to .aS \\ B \to . \end{array}}$$

Follow(B) $\cap$ $\{a\}$

$\{a\} \cap \{a\} \neq \phi$

SR conflict.

- LR(0) parsing:-
  - It can be divided into 2 parts.

1. Construction of canonical set of items (or) LR(0) items:- Construction of LR(0) items in LR(0) parser is same as Construction of LR(0) item in SLR(1).

2. Construction of Parsing table: It is also same as Construction of parsing table in SLR(1) except that if $A \to \alpha.$ in I_i, then we write Reduce by $A \to \alpha$ in entire Action part of i^{th} row of the table (we put the Reduction in the entire row we don't see the follow).

- Conflict in LR(0) parser
- Shift-Reduce conflict

I_i $\boxed{\begin{array}{l} A \to \alpha.aB \\ B \to \gamma. \end{array}}$ → in entire of I_i.

- S/R Conflict

not LR(0).

If $\{a\} \cap$ Follow(B) $\neq \phi$

- Reduce-Reduce conflict.

I_i $\boxed{\begin{array}{l} A \to \alpha. \\ B \to \gamma. \end{array}}$ If follow(A) $\cap$ follow(B) $\neq \phi$

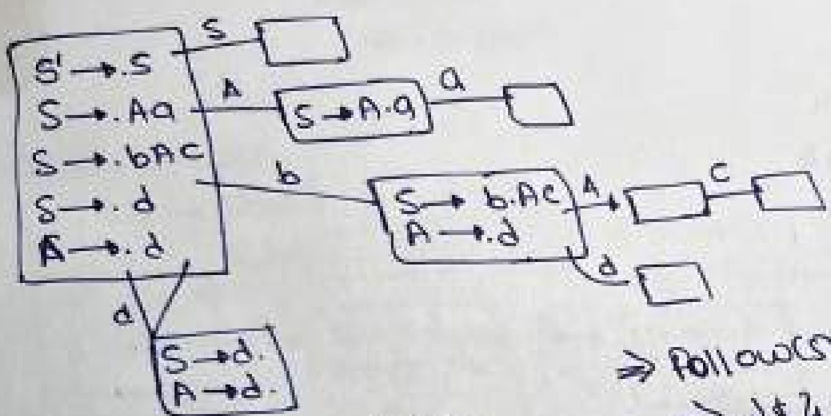
- R/R conflict

not LR(0)

Q] The following grammer is

$G = \{ S \to Aa / bAc / d,\ A \to d \}$

a) LR(0) but not SLR(1)
b) SLR(1) but not LR(0)
c) Both LR(0) & SLR(1)
d) neither LR(0) nor SLR(1)



R/R conflict in LR(0)
not LR(0)

$\Rightarrow$ Follow(S) ∩ Follow(A)
$\Rightarrow \{\$\} \cap \{a, c\}$
$\Rightarrow \phi$
$\Rightarrow$ no conflict.

∴ no conflict for SLR(1)
$\Rightarrow$ The grammer is SLR(1) but not LR(0).

Q] The following grammer is.

$G = \{ E \to aEb / Tb,$
$T \to bT / af / \epsilon,$
$F \to fa / b / \epsilon a \}$

$E' \to E$
$E \to .aEb$
$E \to .Tb$
$T \to .bT$
$T \to .af$
$T \to .$

$\Rightarrow$ Follow(T) ∩ {b, a}
{a, b} ∩ {a, b}
{a, b} ≠ ϕ
So not SLR(1)

$\Rightarrow T \to .af \cap T \to .$
{a} ∩ in whole row ≠ ϕ

$\Rightarrow$ not LR(0)

LR(0) Items

- LR(0) parser
- SLR(1) parser.

LR(1) items

- LALR(1) parser
- CLR(1) parser.

LR(1) items = LR(0) items + Look Ahead symbols.

$A \rightarrow \alpha . a\beta$, $a|b|\$$

LR(0) item LAS

LR(1) item.

- **CLR(1) parsing**

• It can be divided into 2 parts

1. Construction of canonical set of item (or) LR(1) items.
2. Construction of parsing table.

• Construction of canonical set of items (or) LR(1) items.

It has 2 part.

(i) Construction of closure operatn

(ii) construction of GOTO operatn.

• construction of closure operatn.

(i) initially, add $S' \rightarrow .S, \$$

(ii) If $A \rightarrow \alpha . B\beta, a$ is in I_i & if $B \rightarrow \gamma$ is production

then add $B \rightarrow . \gamma, x$ in I_i where $x \in First(\beta a)$

I_i: $\boxed{\begin{array}{l} A \rightarrow \alpha . B\beta, a \\ B \rightarrow . \gamma, x \end{array}}$ where $x \in First(\beta a)$.

Construction of GoTo operation:-

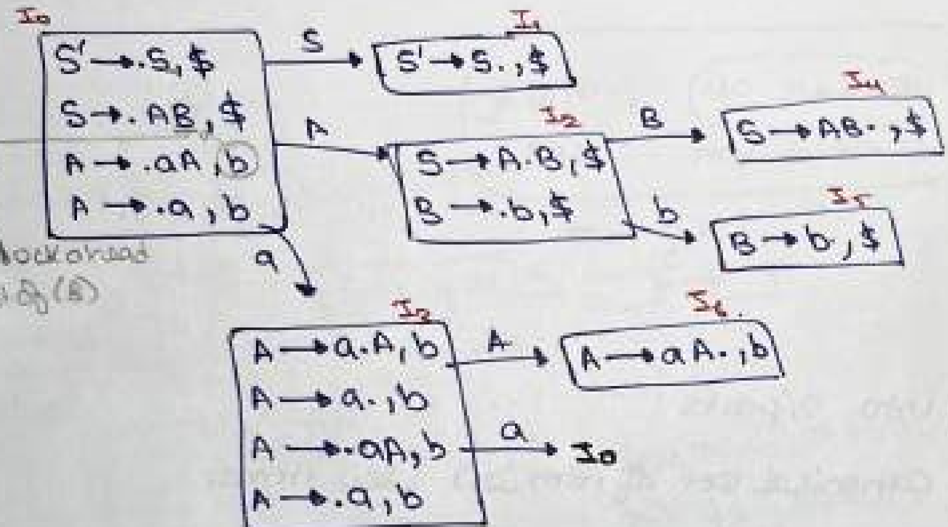
① If $A \to \alpha.X\beta, a$ is in I_i then Goto $(I_i, X) = A \to \alpha X.\beta, a$

Here X can be a terminal or non terminal.

Q] Construct LR(1) items for the following grammer.

$G = \{S \to AB, A \to aA/a, B \to b\}$

due to this we get b as lookahead as first of (B)



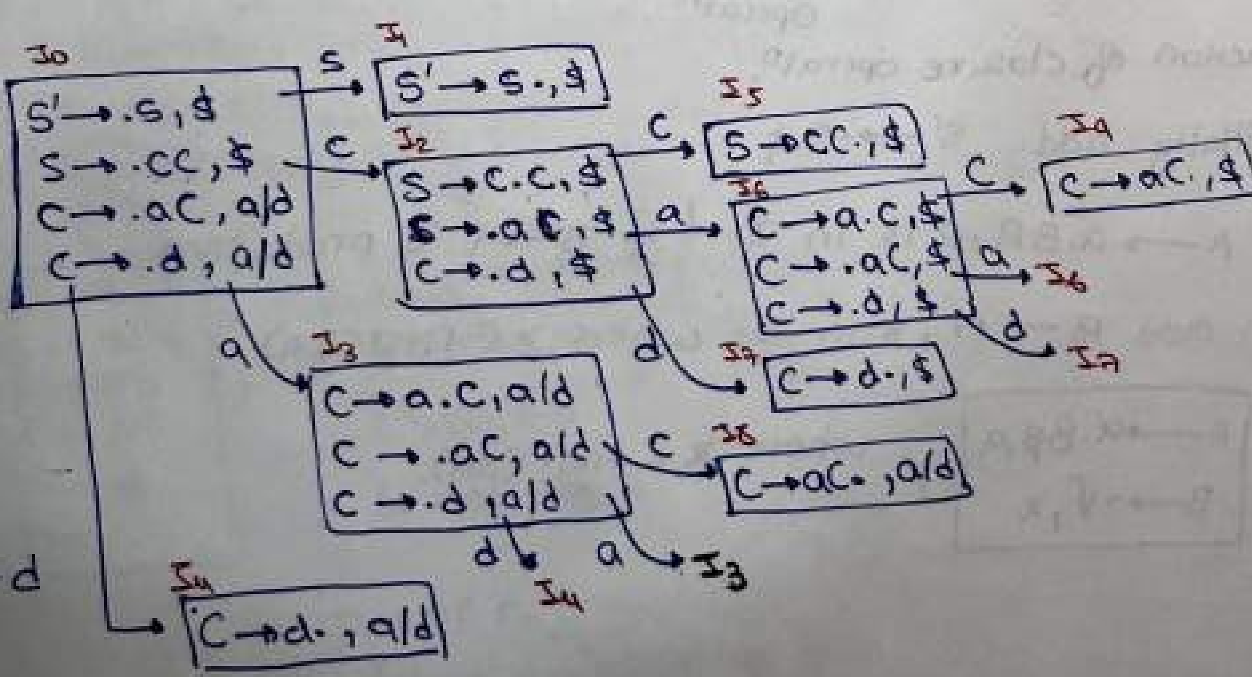
→ if $.S$ nothing then $ as lookahead

→ if $.A$ nothing then copy the pervious lookahead.

→ if $.AB$ then go for 1st of B i.e First(B)

Q] Construct LR(1) parsing table or CLR(1)

$G = \{S \to CC, C \to aC/d\}$



Q] G = {S → aSa / Aa , A → aA / b}

I_0
S' → .S, \$
S → .aSa, \$
S → .Aa, \$
A → .aA, a
A → .b, a

$\xrightarrow{a}$

S → a.Sa, \$
A → a.A, a
A → .aA, a
A → .b, a
S → .aSa, a
S → .Aa, a

$\xrightarrow{A}$

A → aA., a
S → A.a, a

S/R conflict
not CLR(1)

• LALR(1) parsing :-

• In CLR(1) parsing those states having everything as same but with different look ahead, we will separate them into into 2 different state. But in LALR(1) parsing we will merge them into single state

merging of above diagram.

I_3

C → a.C, a/d
C → .aC, a/d
C → .d, a/d

\+

I_6

C → a.C, \$
C → .aC, \$
C → .d, \$

=

I_{36} Goto(I_0, a)

C → a.C, a/d/\$
C → .aC, a/d/\$
C → .d, a/d/\$

Goto(I_0, d)

$I_4 + I_7 = \boxed{C \rightarrow d., a/d/\$}$

Goto(I_{36}, C)

$I_8 + I_9 = \boxed{C \rightarrow aC., a/d/\$}$

note
• construction of LALR(1) parsing table is same as CLR(1)

State	Action			Goto	
	a	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		

Goto(I_2, a) = I_{36}
Goto(I_2, d) = I_{47}
Goto(I_{36}, c) = I_{89}
Goto(I_{36}, a) = I_{36}
Goto(I_{36}, d) = I_{47}.

All the enteries are single
∴ the grammer is LALR(1)

SLR(1) : not SLR 1.

$S' \to .S$
$S \to .AaAb$
$S \to .BbBa$
$A \to .$
$B \to .$

b/c
Follow(A) ∩ Follow(B)
{a,b} ∩ {a,b}
$\neq \phi$

LR(0)
Can't be b/c it's not SLR(1).

CLR(1):

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

$\xrightarrow{A}$ $S \to A.aAb, \$$

$\xrightarrow{a}$ $S \to Aa.Ab, \$$
$A \to ., b$

So, no SR or RR conflict in CLR(1).

So, no LALR(1) too.

Q) $S \to CC$
$C \to cC \mid d$

LALR:

$S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .d, c/d$

$\xrightarrow{C}$ $S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

$\xrightarrow{c}$ $C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

$\xrightarrow{d}$ $C \to d., c/d$

$\xrightarrow{c}$ $C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

Syntax Directed Translation → SDT

- Definition of SDT
- Type of attribute
- Construction of SDT's.

↓

the token produce by lexical Anayzer will be given as I/p to the Syntax analyser then a parser tree will be produce by the Syntax analyser using the production of the CFG but the Syntax analyser can't check mean of the source code. the meaning of the source code will be verified in the semantic analyser. there will be done by adding some extra rules. to every production of the grammer. the grammer with, these extra information is called as Syntax directe translation (SDT).

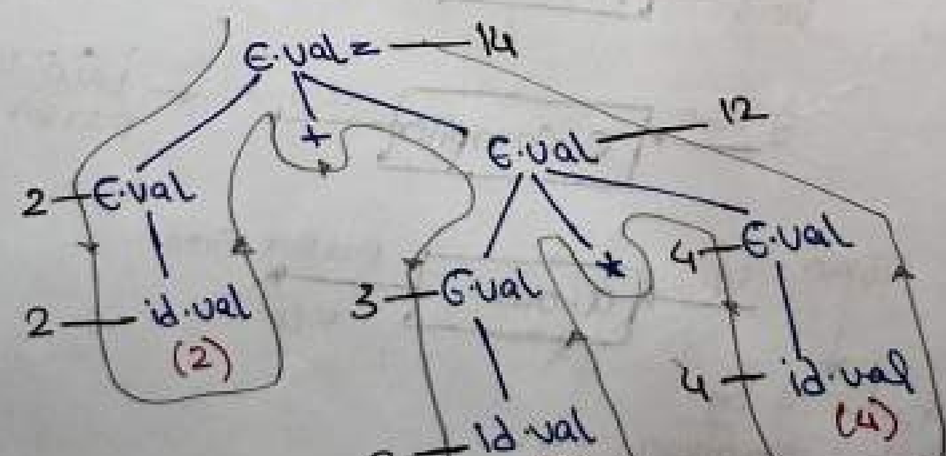
CFG + Semantic Rules = SDT

- consider the following SDT. If the I/p is 2+3*4 then o/p?

SDT
$E \rightarrow E + E \; \{E.val = E.val + E.val\}$
$\quad / E * E \; \{E.val = E.val * E.val\}$
$\quad / id \; \{E.val = id.val\}$

2+3*4

$\boxed{id + id * id}$



both 14 or 20 are correct as the grammer is ambiguous.

- Common error encounter in semantic anlyser.

① undeclared variable

ex: int x=y+5

undeclared

② Type mismatch

ex: String str="Hello";

int num=str;

Incompatible type

③ Redeclared variable

ex: int x=5;

int x=10;

x is already declared.

④ Missing return stmt

ex: int add(.....){

//return missing

}

⑤ Funct^n argument mismatch.

ex: int add(int x, int y){

.....

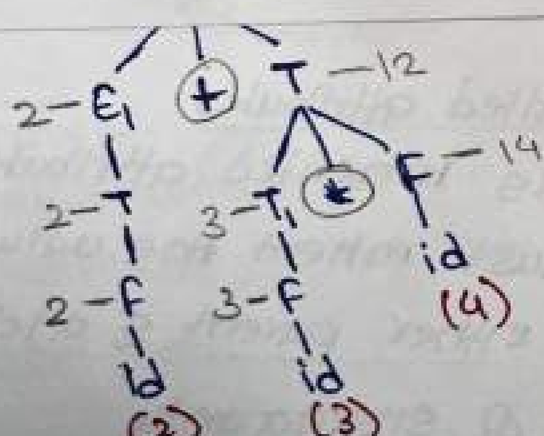
}

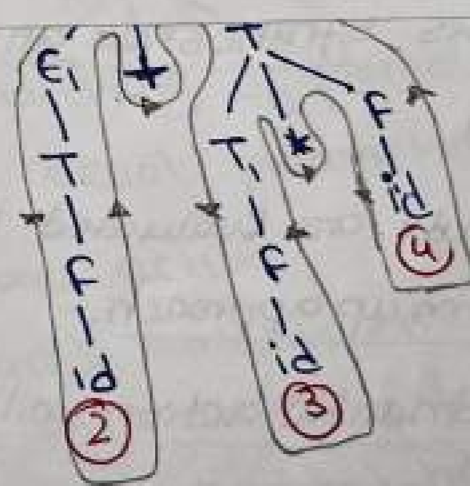
int sum=add(5)

mismatch

⑥ Division by zero:

ex: result=10/0;

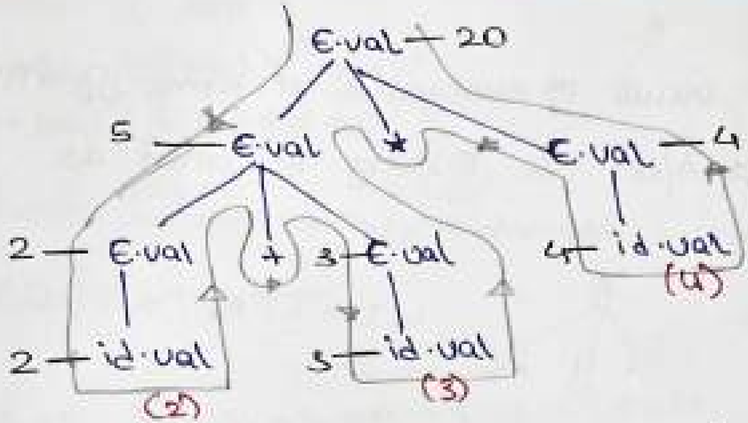




⇒ 2 3 4 * +

- Type of attribute

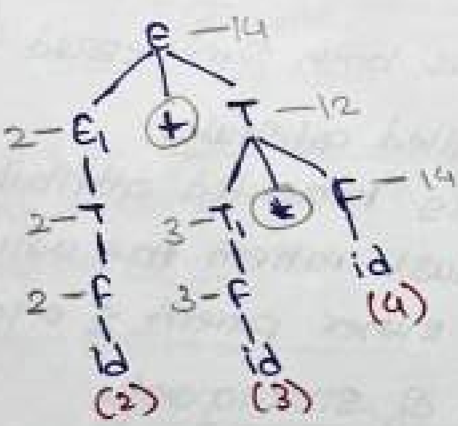
• Synthesized attribute: The attribute whose value is evalua[ted] ... attributes values of its children. is called

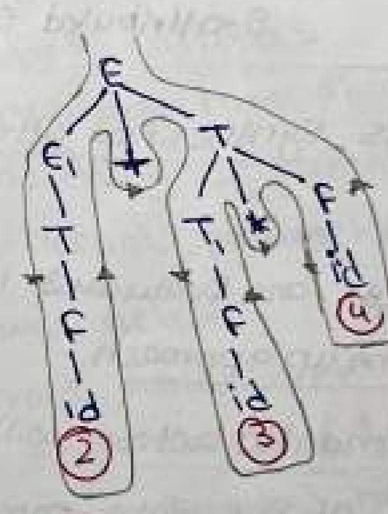


(2) String 2+3*4

Q. $E \rightarrow E_1 + T$ { E.val = E_1.val + T.val }
| T { E.val = T.val }
$T \rightarrow T_1 * F$ { T.val = T_1.val * F.val }
| F { T.val = F.val }
$F \rightarrow id$ { F.val = id.val }

$E \rightarrow E_1 + T$ { printf('+'); }
| T { }
$T \rightarrow T_1 * F$ { print('*'); }
| F { }
$F \rightarrow id$ { printf(id.val); }





⇒ 2 3 4 * +

- Type of attribute
  - Synthesized attribute: The attribute whose value is evaluated in terms of attributes values of its children. is called as synthesized attribute. (here we can calculate only parent value)
  - It is an attribute of a non terminal which is on the LHS of the production.

A.s = f(x.s / y.s / z.s).

ex: $A \rightarrow XYZ$

- the following SDT is

$A \rightarrow BC \; \{A.l = B.l\}$ — S-att^n & L-att^n

$C \rightarrow BA \; \{A.l = B.l + C.l\}$

finding parent value — finding child value — L-att^n not S-attribute

∴ whenever finding child then it is not S-attribute

So, its L-attribute.

Q) $A \rightarrow B + D \; \{D.val = B.val - 1\}$ — finding child value — L-attribute.

$B \rightarrow C * D \; \{C.val = B.val * D.val\}$

finding child value — Right sibling

So not L-attribute

So, (a) S-attribute
(b) L-attribute
(c) Both
(d) none

- Semantic Errors :-
  - undeclared variables
  - In compatable type of operands
  - not matching of formal parameter with actual parameters.

Note :-
- every S-attribute SDT is also L-attribute SDT but not vice-versa

- Error recovery action
  - If undeclared variable error is encounter then an entry for that variable in the symbol table will be done by compiler.
  - when incompatible type of operand error has encountered then all the operand will be converted into large datatype of the operand by the compiler these is called as automatic type conversion or implicit type conversion

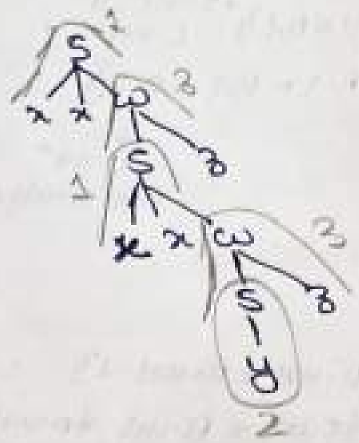
Q] consider the following SDT.

S → xxW {print('1');}
  / y {print('2');}

W → Sz {print('3');}

if the
I/p = xxxxyzz

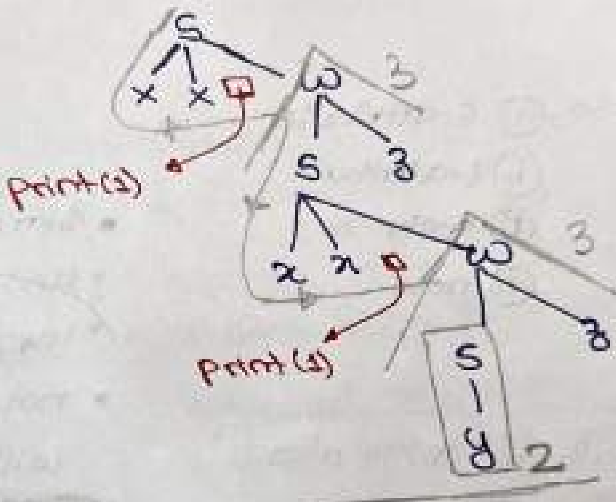
O/p: 23131



S → xx {print('1');} W.
  / y {print('2')}

W → Sz {print('3')}

I/p string: xxxxyzz
O/p: 11233



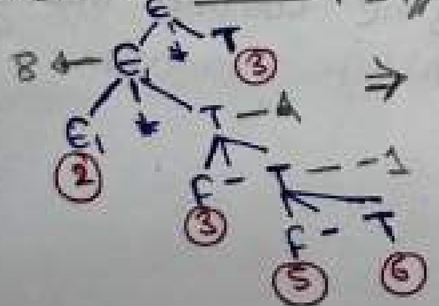
Shortcut

Q] E → E_1 * T {E.val = E_1.val * T.val}
  / T {E.val = T.val}

T → F − T {T.val = F.val − T.val}
  / F {T.val = F.val}

F → id {F.val = id.val}

I/p = 2*3−5−6*3 ⟶ 24//



⇒

going up the 1st operand that you get will have higher precendence

& E → E_1 * T
left associative
so, "*" will be

// but if you can go back to upper level than we can decide the precendence like

T → E*T
E → T*F } now we can't decide the precendence

- $2 * 3 - 5 - 6 * 3$

$- > *$

$\Rightarrow 2 * (3 - 5 - 6) * 3$

- is right associative

$\Rightarrow 2 * (3 - (5 - 6)) * 3$

$\Rightarrow ((2 * 4) * 3)$

$\Rightarrow 24$

Q] consider the following SDT

$L \rightarrow LB$ {L.val = L.val + B.val}
$/ B$ {L.val = B.val}

$B \rightarrow 0$ {B.val = 1}
$/ 1$ {B.val = 0}

I/p = 0100
O/p = 3
Count the no. of zero.

if
$B \rightarrow 0$ {B.val = 0}
$/ 1$ {B.val = 1}

then count no. of 1's

if
$B \rightarrow 0$ {B.val = 1}
$/ 1$ B.val = 1}

then count the length of string.

- $L \rightarrow LB$: {L.val = L.val * 2 + B.val}
$/ B$. {L.val = B.val}

$B \rightarrow 0$ {B.val = 0}
$/ 1$ {B.val = 1}

$(0)_2 = (0)_{10}$
$(1)_2 = (1)_{10}$

I/P: 1 0 1 0 1

$1 * 2 + 0$

$2 * 2 + 1$

$5 * 2 + 0$

$10 * 2 + 1$

21

$N \to L_1 \cdot L_2$ $\{N.val = L_1.val + \frac{L_2.val}{2^{L_2.count}}\}$

$L \to L_1 B$ $\{L.count = L_1.count + B.count;\ L_1.val = L_1.val * 2 + B.val\}$

$\quad / B$ $\{L.count = B.count;\ L.val = B.val\}$

$B \to 0$ $\{B.val = 0;\ B.count = 1\}$

$\quad / 1$ $\{B.val = 1;\ B.count = 1\}$

ex: $\frac{101}{5} \cdot 11 = 5.75$

$\frac{1}{2}$

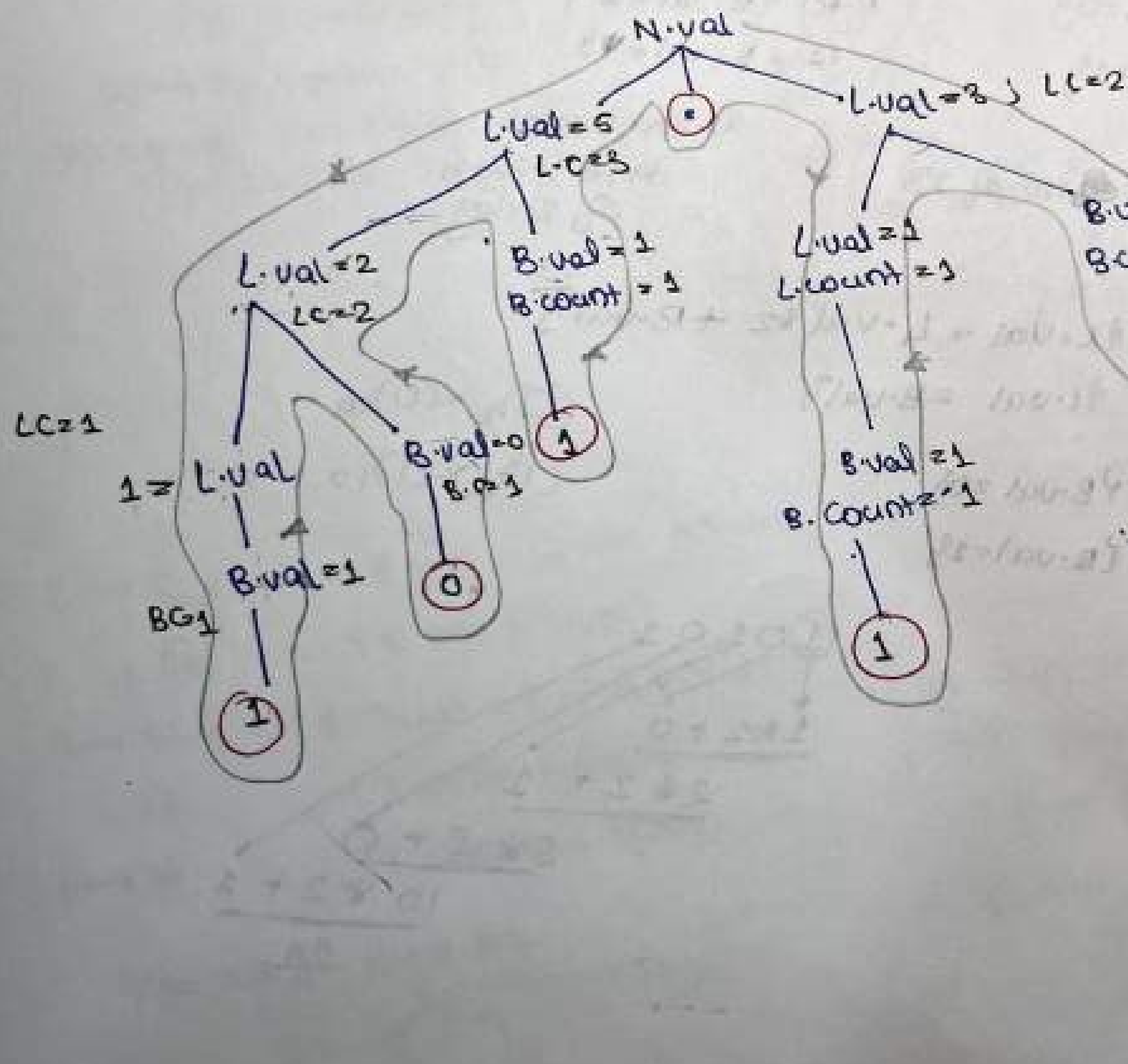
$\frac{1}{2^2}$

$101 \cdot 11$

$2^2\ 2^1\ 2^0 \cdot 2^{-1}\ 2^{-2}$

$5 + \frac{3}{2^2} = \frac{3}{4} = 0.75$

$\boxed{5.75}$



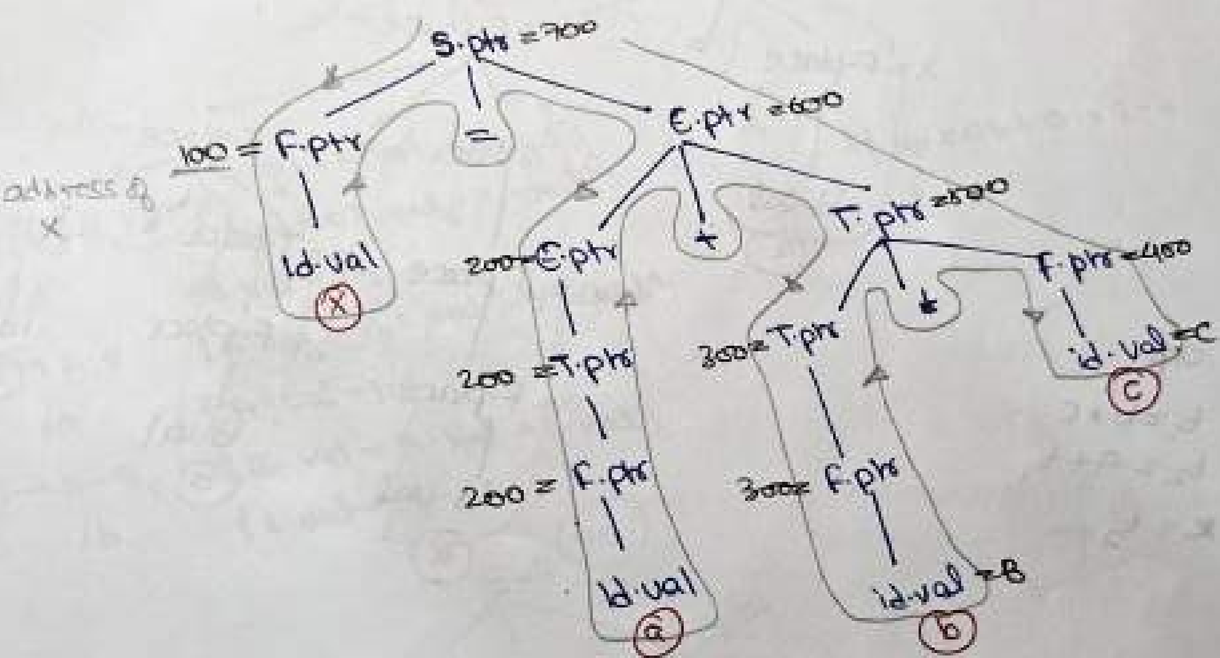
i.e $5 + \frac{3}{2^2}$

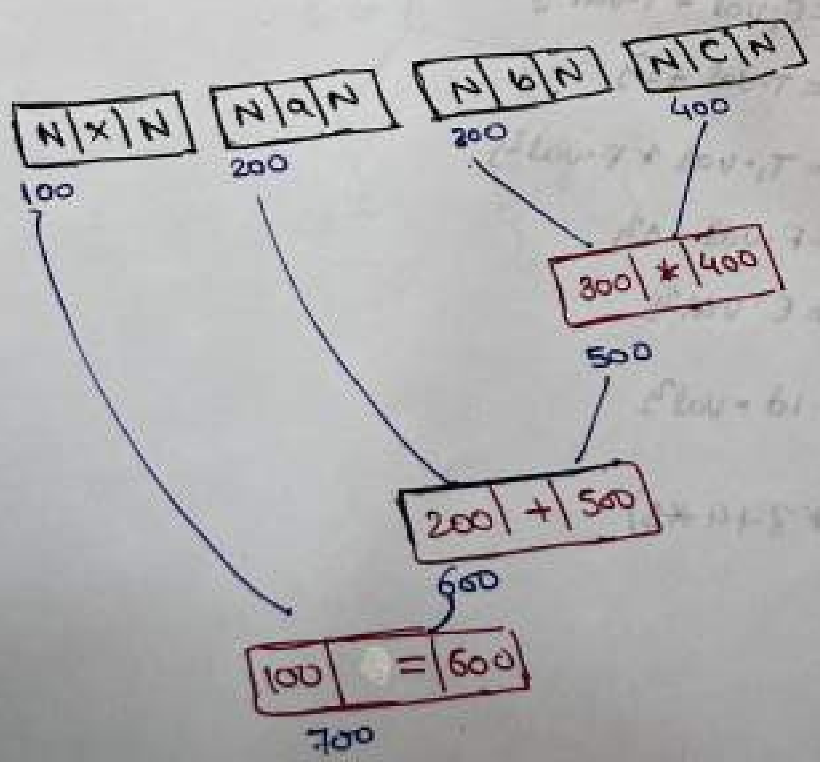
$\Rightarrow 5 + \frac{3}{4}$

$\Rightarrow 5.75.$

Q] Construct a SDT which create a syntax tree for the expression $x = a + b * c$

$S \rightarrow F = E$		{S.ptr = makenode (F.ptr, =, E.ptr)}
$E \rightarrow E + T$		{E.ptr = makenode (E₁.ptr, +, T.ptr)}
	$\mid T$	{E.ptr = T.ptr}
$T \rightarrow T * F$		{T.ptr = make node (T₁.ptr, *, F.ptr)}
	$\mid F$	{T.ptr = F.ptr}
$F \rightarrow id$		{F.ptr = make node (Null, id.val, Null)}





Q] Construct an SDT which create 3 address code for the same expression.

$S \rightarrow F = E$ { F.place = E.place }

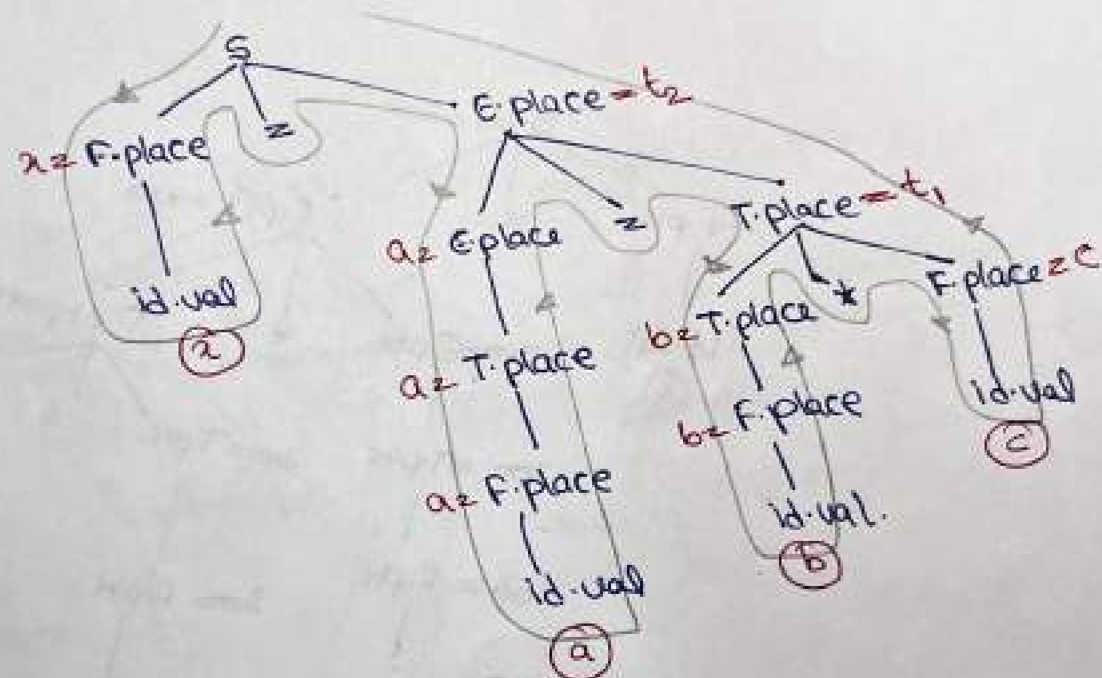
$E \rightarrow E_1 + T$ { E.place = new temp(); Gen(E.place = E_1.place + T.place) }

$\mid T$ { E.place = T.place }

$T \rightarrow T_1 * F$ { T.place = new temp(); Gen(T.place = T_1.place * F.place) }

$\mid F$ { T.place = F.place }

$F \rightarrow id$ { F.place = id.val }



O/P:

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

Q] $E \rightarrow E_1 * T$ { E.val = E_1.val + T.val }

$\mid T$ { E.val = T.val + 1 }

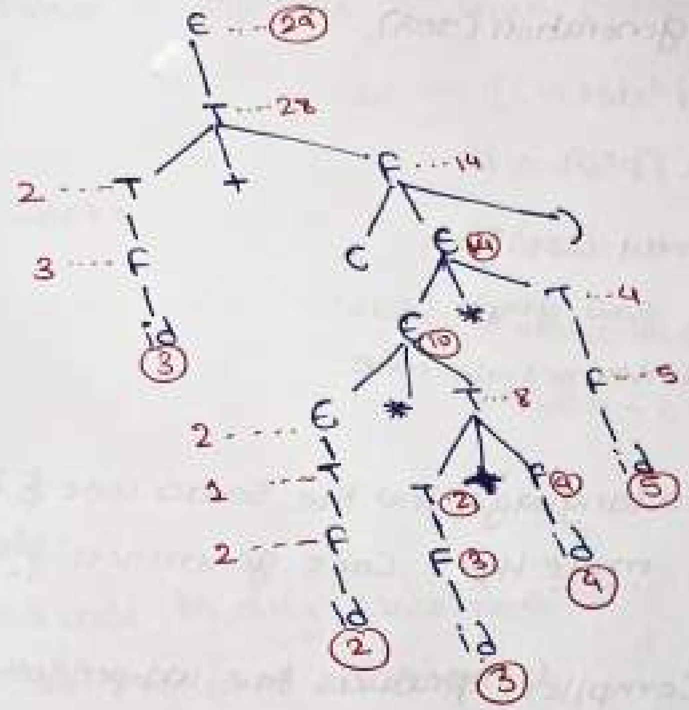
$T \rightarrow T_1 + F$ { T.val = T_1.val * F.val }

$\mid F$ { T.val = F.val − 1 }

$F \rightarrow (E)$ { F.val = E.val }

$\mid id$ { F.val = id.val }

i/p: = 3+(2*3+4*5)



Q] $S \rightarrow A+B$ {S.val = A.val * B.val}

$\quad | B * S_1$ {S.val = B.val − S_1.val}

$\quad | a$ {S.val = 2}

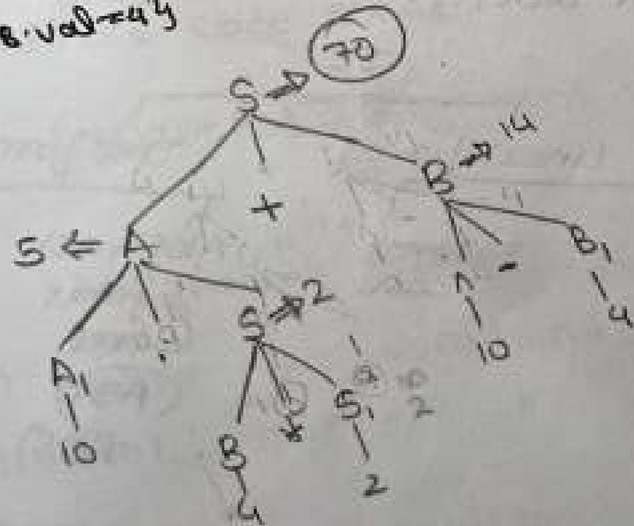
$A \rightarrow A_1 S$ {A.val = A_1.val ÷ S.val}

$\quad | a$ {A.val = 10}

$B \rightarrow A - B_1$ {B.val = A.val + B.val}

$\quad | b.$ {B.val = 4}

If i/p = ab*a+a-b



• Postfix notation :- Operand 1st than operator.

• a+b : ab+

• b÷c : bc÷

• a+(b*c) : abc*+

ex: x= ((a+(b*c))-d)*((a÷c)-d)

⇒((a+bc*)-d)*(ac÷ -d)

⇒ (abc*+-d)*ac÷d-

~~x~~=abc*+d-ac÷d-*

xabc*+d-ac÷d-*=

• 3-address code:

• In 3-address code, in any statement there will be maximum 3 reference <u>two for operands & one for result.</u> & in 3-address code in any stmt there will be max. one operator other than assignment operator.

the sequence of 3-address stmts is called as 3 address code.

Some valid 3-address code stmts.

a= b op c (relational assignment stmt)

a= op b "

a=b Assignment stmt.

if (relation) Goto L Conditional Jump.

ifz (relation) Goto L "

if zero

Goto L unconditional Jump

a[i] =b
a= b[i]
} Array Index assignment

a= &b address assignment stmts.

a= *b
*a=b
} pointer assignment stmts.

param x_1
param x_2
⋮
param x_3

The parameter to a procedure or a function call are defined by param instruction

call p, n — Invocation of procedure p that take n argument

y = call p, n — " and the result of the p is copied/store is y.

return — passes the control to the instruction following call instruction that invoked a procedure p.

return x — " and the value of x is returned

exits : exits the program.

Q) write a 3 address code for the following stmt.

Stmt are executed Sequential

```
if (a<b)            if (a<b) Goto L1
   a = a+1          b = b+1
else                Goto L2
   b = b+1.         L1: a = a+1
                    L2: ____
```

```
ex: for(i=0; i<=n; i++){      i=0
       x = x+1;               L1: if (i<=n) Goto L2
       y = y-1;                   Goto L3
    }                         L2: x = x+1;
                                  y = y-1;
                                  i = i+1;
                                  Goto L1;
                              L3: ____
```

(or)

```
i=0
L1: if (i>n) Goto L2
    x = x+1
    y = y-1
    i = i+1
    Goto L1
L2: ____
```

- Implementation of 3-address code.

1. Quadruple
2. triple — we can't change the representation
3. Indirect Triple

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

$t_1 = a+b$

$t_2 = c/d$

$t_3 = t_1 - t_2$

$t_4 = t_1 - c$

$t_5 = t_3 * t_4$

$x = t_5$

1. Quadruple by default

S.no.	operator	Arg1	Arg2	Result
1.	+	a	b	t_1
2.	/	c	d	t_2
3.	−	t_1	t_2	t_3
4.	−	t_1	c	t_4
5.	*	t_3	t_4	t_5
6.	=	t_5		x

Advantage:
- one can access the value quickly using tempor^n variable
- statement can be moved around

Disadvantage:
- more no. of temporary variable required more space.

2. Triple

1. c/d
2. a+b
3. (c/d)-(a+b)
4. c/d -c

S.no.	Opr	Arg1	Arg2
1.	+	a	b
2.	/	c	d
3	−	1	2
4	−	1	c
5.	*	3	4
6.	=	5	x

here we write the reference

Adv.: less memory required

Dis: Stmt can't be moved around
Adv.

• Loop Invariant: It is a technique in which avoiding the Computation inside a loop whose value is remain unchange even though the loop run for multiple time

ex: c=0; y=2; a=3; b=4;

```
for(i=0; i<=n; i++){
    x=y+a*b;
    c=c+b.
}
```

→

```
c=0; y=2; a=3; b=4
for( i=0; i<=n; i++){
    c=c+b
}
x=y+a*b;
```

• Loop functn / Loop jamming | Loop Combine.

→ it's a technique in which merging the several loop ~~[illegible]~~ into one loop.

```
for(i=0; i<=n; i++){
    x=x+a*i;
}
for(j=0; j<=n; j++){
    y=y*b÷j;
}
```

→

```
for(i=0; i<=n; i++){
    x=x+a*i;
    y=y*b÷i;
}
```

• loop unrolling:- it is a technique in which no. of jump & test will be reduce by writting the code multipletime.

ex.

```
for(i=0; i<=n; i++){
    a[i]=b[i]
}
```

→

```
for(i=0; i<=n; i++){
    a[i]=b[i]
    i=i+1;
    a[i]=b[i]
}
```

Liveness Analysis

- a variable x is said to be live at any stmt if it is used in the stmt or anywhere in the subsequent program on RHS (before it is defined)

Should not be on LHS before RHS

- a variable is said to be live for a particular stmt if in the subsequent stmt or the upcoming stmt in should be on RHS & before that it should not be in LHS.

1. a = b + c
2. b = a - b
3. c = d + a
4. d = c + b.

liveness	a	b	c	d
1	✗	✓	✓	✓
2	✓	✓	✗	✓
3	✓	✓	✗	✓
4	✗	✓	✓	✗

1 b = a - c
2 c = a + d
3 a = b + e
4 if(a>d) goto L 3
5 b = a + c
6 e = b - d
7 return e

as it using

	a	b	c	d	e
1.	✓	✗	✓	✓	✓
2.	✗	✓	✗	✓	✓
3.	✗	✓	✓	✓	✓
4.	✓	✓	✓	✓	✓
5.	✓	✗	✓	✓	✗
6.	✗	✓	✗	✓	✗
7.	✗	✗	✗	✗	✓

Q] Find min register.

1. a = b + c
2. c = a - b
3. f = d + e
4. a = c + f
5. e = b - a
6. f = c + d
7. b = f + f
8. a = b + e

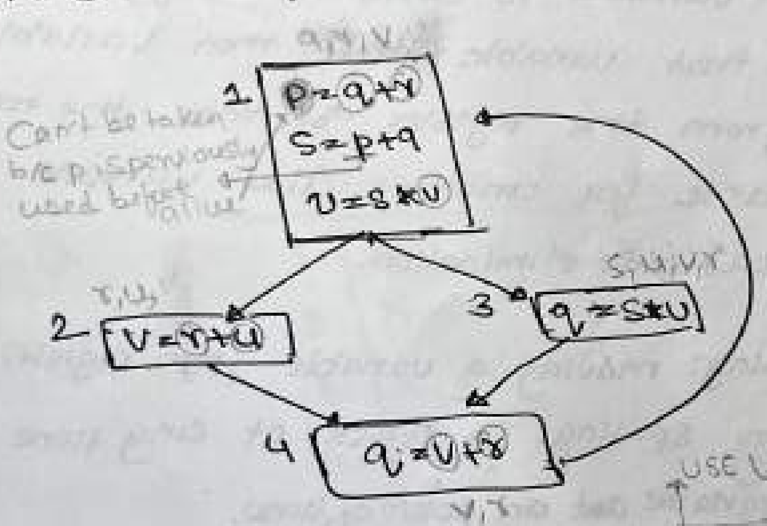
$\overset{c}{R_3} = R_3 - R_1$

$\overset{f}{R_4} = R_5 + R_6$

	a	b	c	d	e	f
1	✗	✓	✓	✓	✓	✗
2	✓	✓	✗	✓	✓	✗
3	✗	✓	✓	✓	✓	✗
4	✗	✓	✓	✓	✗	✓
5	✓	✓	✓	✓	✗	✗
6	✗	✗	✓	✓	✓	✗
7	✗	✗	✗	✗	✓	✓
8	✗	✓	✗	✗	✓	✗

4 min. register required

Q] Consider the following CFG. Find the variable that are live at both block 2 & 3.



Variable that are live at

$IN\,B_2 = \{r, u\}$

$IN\,B_3 = \{s, u, v, r\}$

$IN\,B_2 \cap IN\,B_3 = \{r, u\}$

USE ∪ {OUT - DEF}

Node	USE	DEF	First Go IN	First Go OUT	Second Go IN	Second Go OUT
1	{q, r, v}	{p, s, u}	{q, r, v}	{r, u, s}	{q, r, v}	{s, u, v, r}
2	{r, u}	{v}	{r, u}	{v, r}	{r, u}	{v, r}
3	{s, u}	{q}	{s, u}	{v, r}	{s, u, v, r}	{v, r}
4	{v, r}	{q}	{v, r}	{q, r, v}	{v, r}	{q, v, r}

only USE

all successor USE

IN = The set of variable that are live at the beginning of the block.

OUT = The set of variable that are live at the exit of the block

USE/GEN: The set of variable that are used in the block before they are defined i.e The variable that are on RHS but prior to that stmt they should not be defined in that block.

DEF/KILL = The set of variable that are defined i.e the variable the LHS.

note:
$IN = USE \cup (OUT - DEF)$
$OUT = \cup IN(S)$
successor

Out is to calculate using the successor IN (union)

node	p	q	r	s	u	v
1	×	✓	✓	×	×	✓
2	×	×	✓	×	✓	×
3	×	×	✓	✓	✓	✓
4	×	×	✓	×	×	✓

the variable that come 1st in RHS than live or else in LHS than dead

are in variable

• lexical error

• Here are Some Common error.

① Invalid characters.

ex: x = @

(unmatched)

② unterminated String.

ex: String str = "milin"; — missing

③ unterminated Comment

ex /* not clogsed */

⑧ Missing operators or operand.

ex: int x = 5 + ;

missing opera nd after "+".

④ Exceeding the length of the identifier.

max. length.

Suppose c lang"

max. is 32 characters.

⑤ unrecognized Token

ex int 42x = 10;

not valid token

↳ when the lexical analyzer encounter a sequence of Characters that do not from a valid token.

⑥ Invalid Identifier.

ex: int 2x = 5

Identifier can't start with digit

⑦ Illegal escape character

• message = "Mi\nlin";

occurs when an escape sequence is used incorrectly in a string.

- Common error in Syntax analyzer (parser)

① Syntax Error/missing semicolon

ex if(x>5){...};

missing.

② Missing closing parenthesis

ex: int sum=(3+5);

missing

③ unexpected token

ex: int x=*5;

unexpected

④ missing operator

ex: int sum=3+*5

missing operator after "+"

⑤ invalid statment placement

ex: int x=5;

x+3

expression w/o assignment or print stmt

- Common error encounter in semantic anlyser.

① undeclared variable

ex: int x = y + 5

undeclared

② Type mismatch

ex: String str = "Hello";

int num = str;

Incompatible type

③ Redeclared variable

ex: int x = 5;

int x = 10;

x is already declared.

④ Missing return stmt

ex: int add(.....){

//return missing

}

⑤ Funct^n argument mismatch.

ex: int add(int x, int y){

-- ..

}

int sum = add(5)

mismatch

⑥ Division by zero:

ex: result = 10/0;