Page :
Date :

System Programming & Compiler Construction

Note: Nombering of topics/chapters are ~~done~~ jombled because it is done according to the imp chapter first basis

* Introduction

* Assemblers

* Macros and Macroprocessor

Page :
Date :

* Loaders and Linkers

* Compilers: Analysis Phase

Page :
Date :

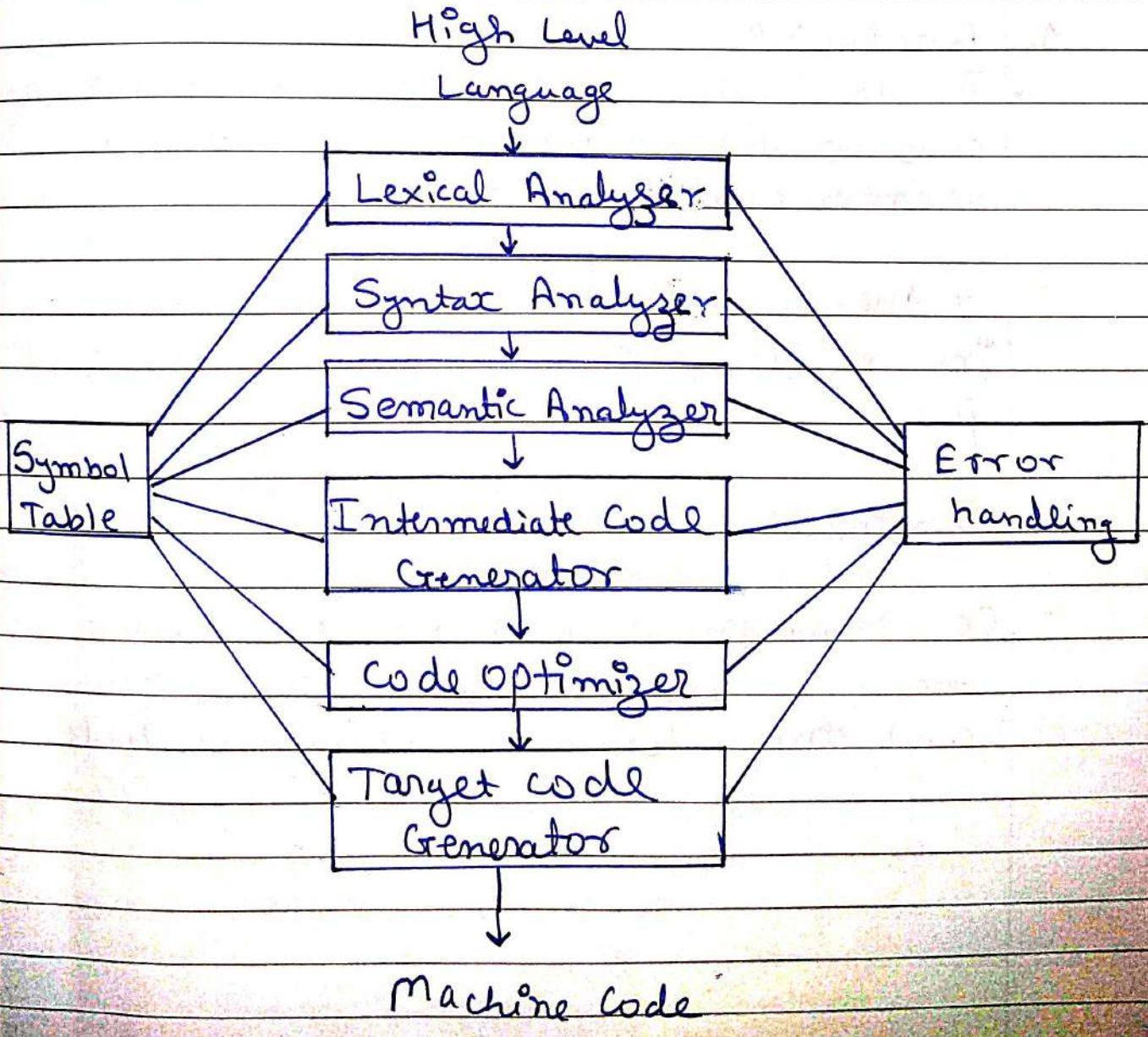
* Compiler's Synthesis Phase

Compiler's Analysis Phase

Phases of compiler

First of all we need to see what is compiler? Compiler is a converter or translator that converts high level language like C++, java, python into low level language or machine code.

This procss of conversion includes different phases of compiler which are as follows:



The phases of compiler are devided into two part Analysis part and Synthesis part.

The Analyses part is often called as front end of the compiler and the Synthesis part is the back end.

Lexical Analyzer, Syntax Analyzer and Semantic Analyzer comes under Analysis part and code optimizer with the code generator is in Synthesis part.

1. Analysis:

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them

- It then uses this structure to create an intermediate representation of the source program.

2. Synthesis

- Constructs the desired target program from the intermediate representation and the information in the symbol table.

1. Lexical Analyzer: Lexical analyzer or Linear Analyzer breaks the sentence into tokens. For example following assignment statement:-

Example

position = initial + rate * 60

Would be groupe into the following tokens:

a. The identifier: position
b. The assignment operator: =
c. The identifier: initial
d. Arithmetic operator: +
e. The identifier: rate
f. Arithmetic operator: *
g. The number: 60

Everything gets stored in symbol table

Symbol Table:

position	Id1 & attributes
Initial	Id2 & attributes
rate	Id3 & attributes

An expression of the form:

position = initial + rate * 60

gets converted to → id1 = id2 + id3 * 60

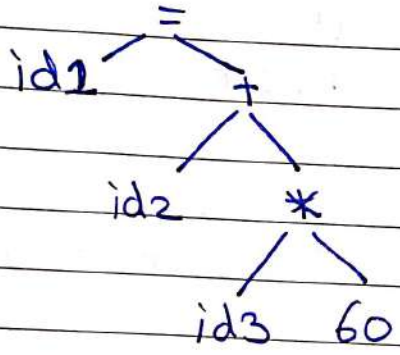
Also it removes spaces & other unnecessory things like comments etc.

2. Syntax Analysis : Syntax analysis is also called PARSING. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. It checks the code syntax using CFG : ie. the set of rules.

If everything is correct parse tree is generated.

Input: id1 = id2+id3*60

parse tree:

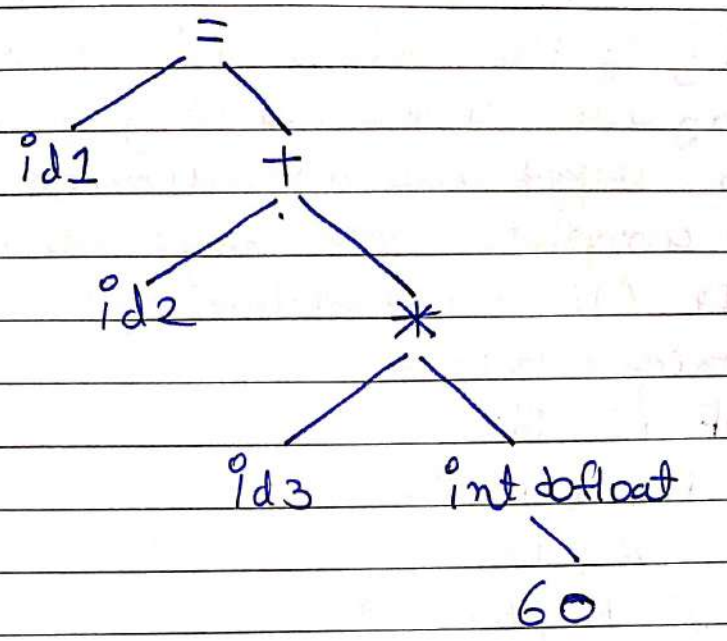


3. Semantic Analysis : It uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

- An important part of semantic anylis is type checking, where the compiler checks that each operator has matching operands. For Ex: many programming language definitions require an array index to be an integer; the compiler must report an error if a floating

point number is used to inder an array.

Syntax tree (updated):



4. Intermediate Code generator:

In this phase Intermediate code is generated which is 3 address code in most of the cases.

t_1 = int to float (60)
t_2 = id3 * t_1
t_3 = id2 + t_2
id1 = t_3

5. Code optimizer:

In this phase the Intermediate code generated is optimized for performance gain.

t_1 = id3 * 60.0
id1 = id2 + t_1

6. Code Generation

The Final phase of the compiler is the generation of the target code, consisting normally of the relocatable machine code or assembly code. compilers may generate many types of target codes depending on M/C while Some compilers make target code only for Specific M/C. Translation of the ~~to~~ taken code might become

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

Illustrate compilers internal representation of Source program or following statement after each phase. (consider each variable as float)

$a = X + X * Y * Z / 100$

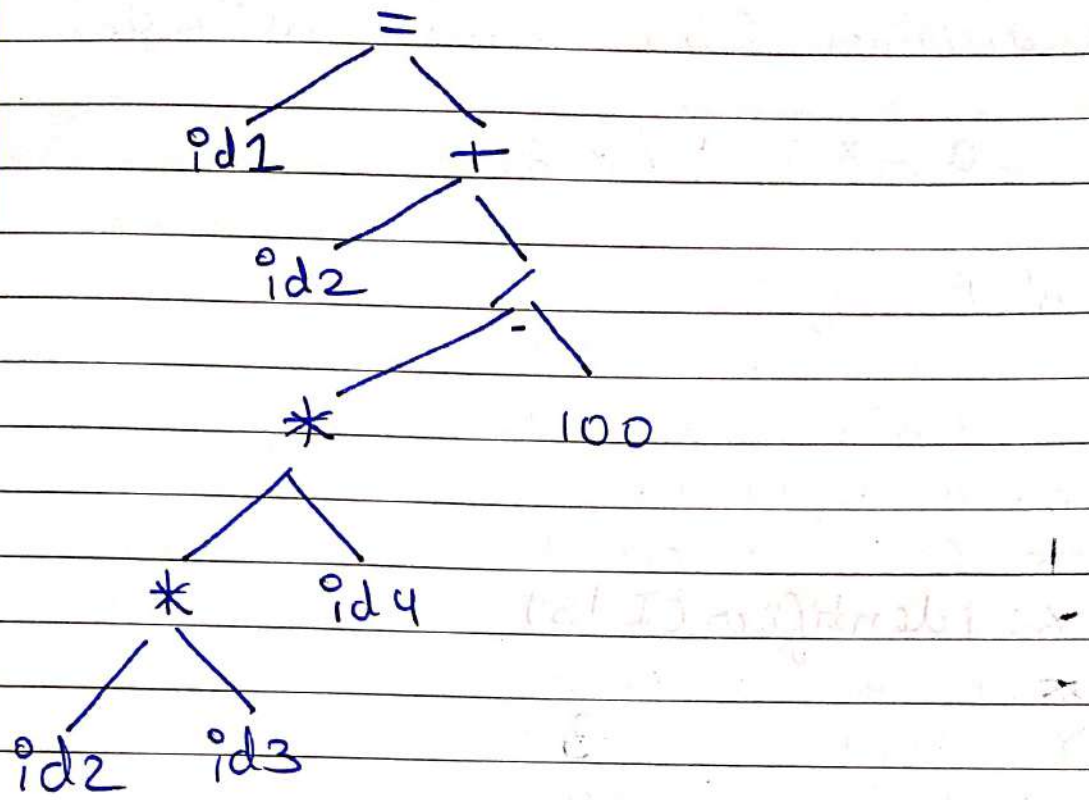
1. Lexical Analyzer
   - a: identifier (Id1)
   - =: assignment operator
   - X: identifier (Id2)
   - +: arithmetic operator
   - *: arithmetic operator
   - Y: identifier (Id3)
   - Z: identifier (Id4)
   - /: arithmetic operator
   - 100: Number / constant.

Symbol table:

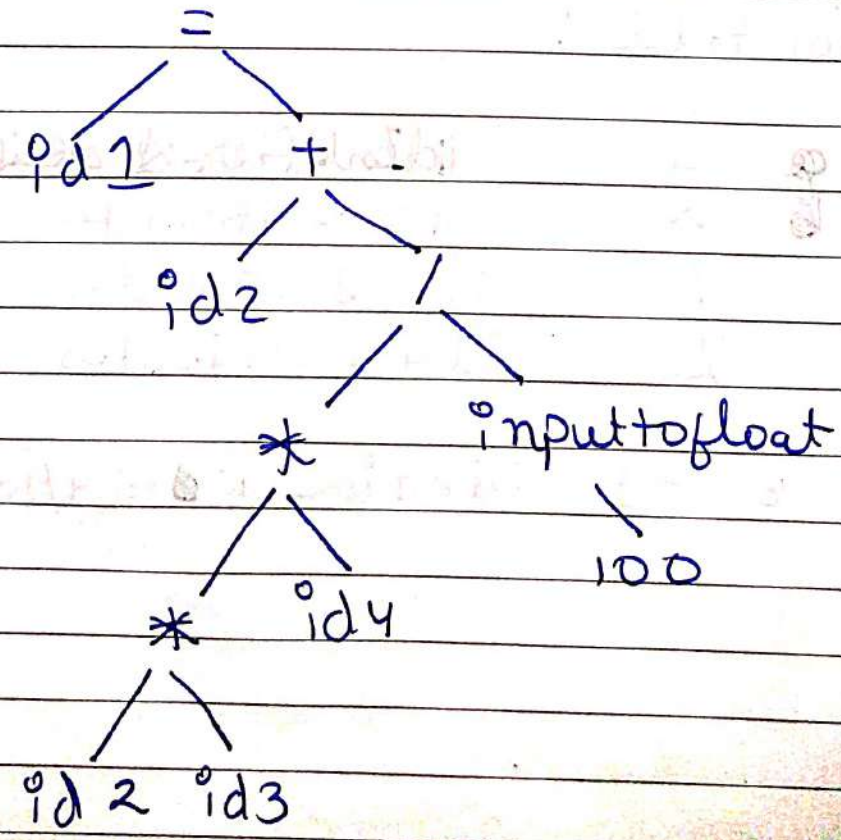
a	id1 & attributes
X	id2 & attributes
Y	id3 & attributes
Z	id4 & attributes

Output : id1 = id2 + id3 * id4 / 100

2. Syntax Analyzer



3. Semantic Analyzer



4. Intermediate code generator

Input: id1 = id2 + id2 * id3 * id4 / 100

```
t1 = int to float (100)
t2 = id2 * id3
t3 = t2 * id4
t4 = t3 / t1
t5 = t4 + id2
id1 = t5
```

5. code optimizer

```
t1 = id2 * id3
t2 = t1 * id4
t3 = t2 / 100.0
id1 = id2 + t3
```

6. Code generation

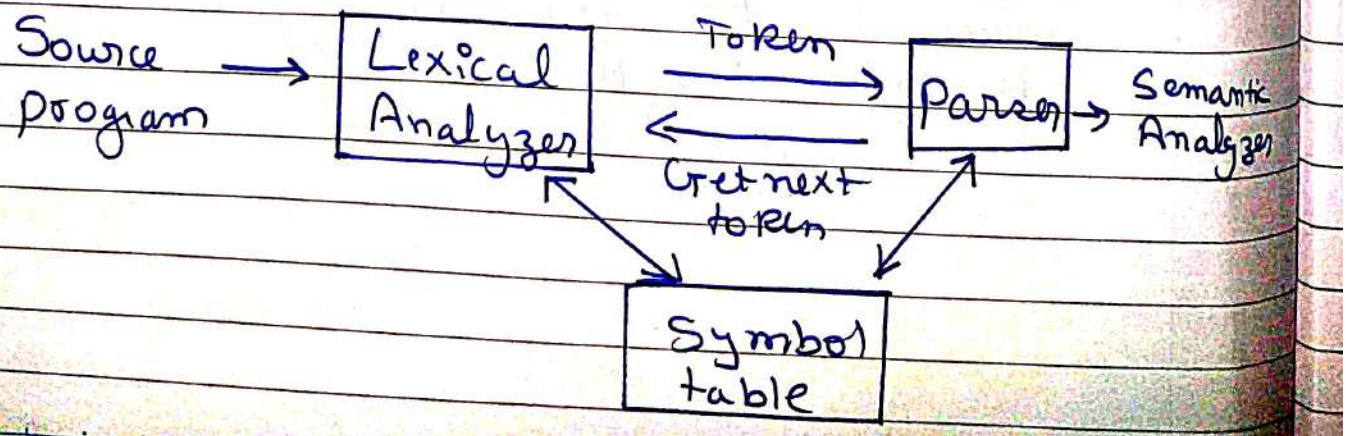
```
MOVF  id2, R1
MOVF  id3, R2
MULF  R2, R1
MOVF  id4, R3
MULF  R2, R3
DIVF  R2, #100.0
ADDF  R1, R2
MOVF  R1, id1.
```

Lexical Analyzer

Lexical Analyzer is a very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences.

In other words it helps you to transform a sequence of characters into that of tokens. The lexical Analyzer breaks this syntax into a series of tokens. It removes any extra space, comment written in the source code.

Programs that performs Lexical Analysis in compiler design are called lexical Analyzers or lexers. A lexer contains tokenizer or Scanner. If the lexical analyzer makes out that the token is invalid, it generates an error. The role of Lexical Analyzer in compiler design is to read Character streams from the source code, Check for legal tokens and pass the data to the Syntax analyzer when it demands.



Sometims Lexical Analyzer is also called the subroutine of Syntax Analyzer. Whenever Syntax analyzer generates the command (Get Next token) So after receiving this command Lexical Analyzer reads the source program and generates the sequence of tokens and provides it to the next phase. Symbol table is used while this process so the token ~~symbo~~ enters Symbol table while Scanning the source program character by character.

- Lexical analyzer is also called as lexical Analysis, Scanner or Tokenizer because it scans and generates tokens which are given to the next phase.

Roles of Lexical Analyzer

- Removes whitespaces and comments from the source program
- Correlates Error messages with the Source program
- Helps you to expand the macros if it is found in the Source program
- Read input characters from the Source program.
- Helps to identify tokens into the Symbol table.

Advantages of Lexical Analysis.

- It is used by web browsers to format and display a web page with the help of parsed data from java script, HTML, CSS.
- A separate lexical analyzer helps you to ~~Cost~~ construct a specialized and potentially more efficient processor for the task.

Disadvantages of Lexical Analysis.

- More efforts is needed to develop and debug the lexer and its token descriptions
- Additional runtime ~~overload~~ overhead is required to generate the lexer tables and construct the tokens.

Q. consider the following grammer
$E \rightarrow E+E \mid E*E \mid id$

Find
1. Left Most Derivation
2. Right Most Derivation
3. Derivation tree for the expression 'id+id*id'
4. Check grammar is ambiguous or not.

=> let us number the productions as:
1. $E \rightarrow E+E$
2. $E \rightarrow E*E$
3. $E \rightarrow id$

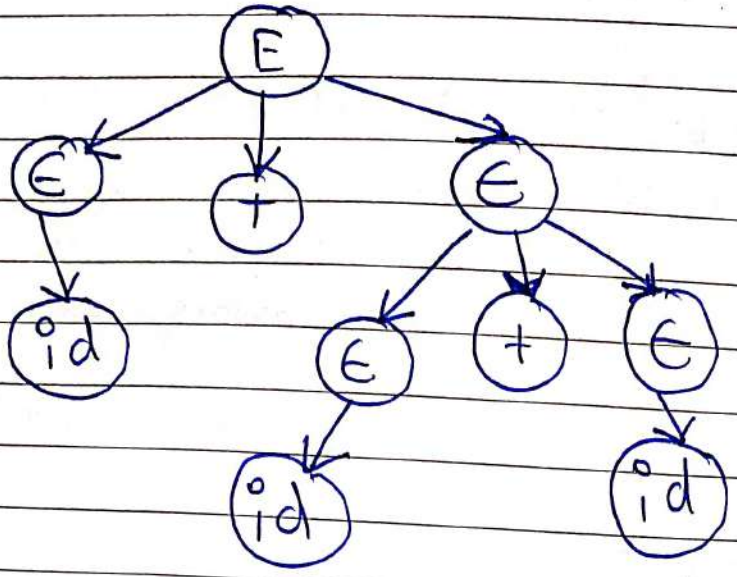
I. Leftmost Derivation for string 'id+id*id'

$E \rightarrow E+E$	Using Rule 1 ie $E \rightarrow E+E$
$E \rightarrow id+E$	Using Rule 3 ie $E \rightarrow id$
$E \rightarrow id+E*E$	Using Rule 2 ie $E \rightarrow E*E$
$E \rightarrow id+id*E$	Using Rule 3 ie $E \rightarrow id$
$E \rightarrow id+id*id$	Using Rule 3 ie $E \rightarrow id$

II. Rightmost Derivation for String 'id+id*id

$E \rightarrow E+E$	Using Rule 1 ie $E \rightarrow E+E$
$E \rightarrow E+E*E$	Using Rule 2 ie $E \rightarrow E*E$
$E \rightarrow E+E*id$	Using Rule 3 ie $E \rightarrow id$
$E \rightarrow E+id*id$	Using Rule 3 ie $E \rightarrow id$
$E \rightarrow id+id*id$	Using Rule 3 ie $E \rightarrow id$

III Derivation



IV Ambiguity check

Find other way of deriving Leftmost Or Rightmost derivation

Leftmost derivation

∈ → ∈*∈ (using Rule 2 ie ∈ → ∈*∈)
∈ → ∈+E*∈ (using Rule 1 ie ∈ → ∈*∈)
∈ → id+∈*∈ (using Rule 3 ie E → id)
∈ → id+id*∈ (using Rule 3 ie ∈ → id)
∈ → id+id*id (using Rule 3 ie ∈ → id)

Rightmost derivation

∈ → ∈*∈ Using Rule 2 ie ∈ → ∈*∈
∈ → ∈*id Using Rule 3 ie ∈ → id
∈ → ∈+∈*id Using Rule 1 ie ∈ → ∈+∈
∈ → ∈+id*id Using Rule 3 ie ∈ → id
∈ → id+id*id Using Rule 3 ie ∈ → id

We can also draw different parse tree/ derivation tree to depict the same string. There are two different ways of deriving string 'id+id*id'.
Hence the given grammer is ambiguous.

=> Consider :
1. S → aB / bA
2. A → a / aS / bAA
3. B → b / bS / aBB
derive "bbaaba" using leftmost & rightmost derivations

i) Leftmost derivation

S → bA	rule (1)
S → bbAA	rule (2)
S → bbaA	rule (2)
S → bbaaS	rule (2)
S → bbaabA	rule (1)
S → bbaaba	rule (2)

ii) Rightmost derivation

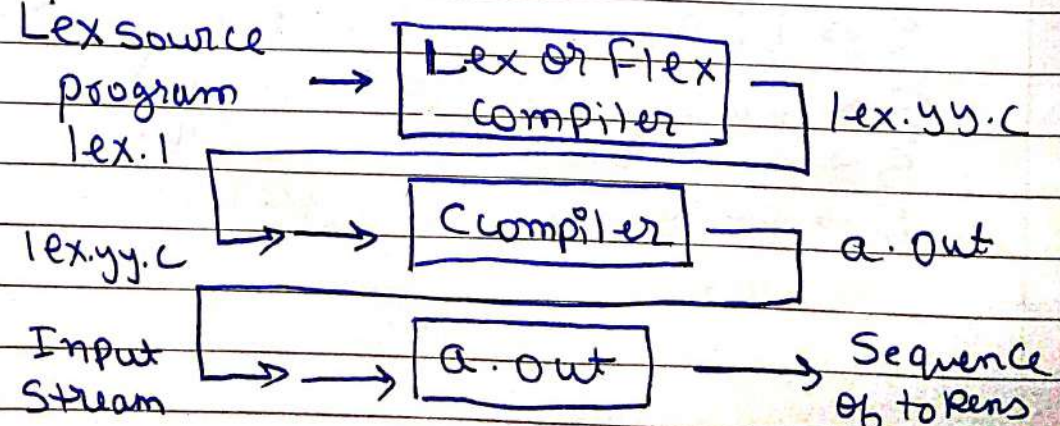
S → bA	rule (1)
S → bbAA	rule (2)
S → bbAa	rule (2)
S → bbaSa	rule (2)
S → bbaaBa	rule (1)
S → bbaaba	rule (3)

Lex

- Lex is the program that generates lexical Analyzer. It is used with YACC parser generator
- The lexical Analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and products the source code as a output through implementing the lexical analyzer in the C program

The Function of Lex is as follows

- Firstly lexical analyzer creates a program lex.l in the lex language. Then Lex Compiler runs the lex.l program and produces C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produce an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex File Format

A Lex program is separated into three sections by %%% delimiters. The formal of Lex source is as follows:

```
{definitions}
%%%
{rules}
%%%
{user Subroutines}
```

Definitions includs declarations of constant, variable and regular definitions.

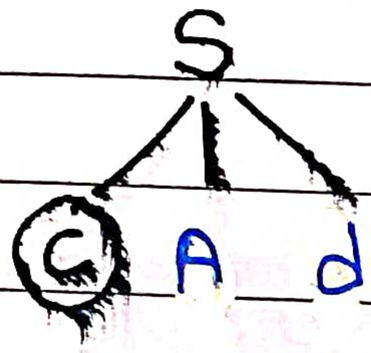
Rules define the statement of form p1{action1} p2{action2}... pn{action}

Where pi describes the regular expression and action1 describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

Recursive Descent parser

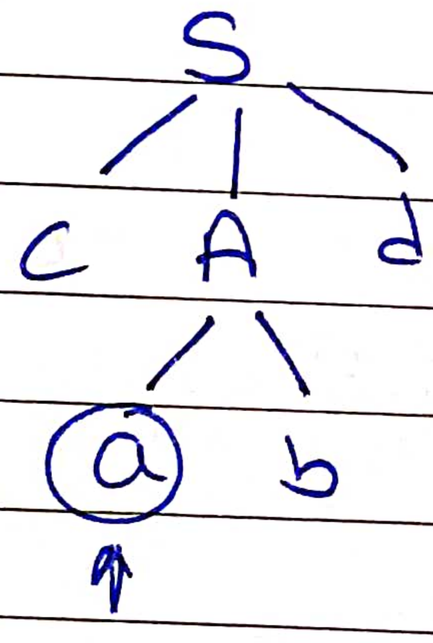
Grammer:
S→cAd String: S=cad
A→ab|a

Step 1: Keep input in buffer
Buffer = c a d
↑



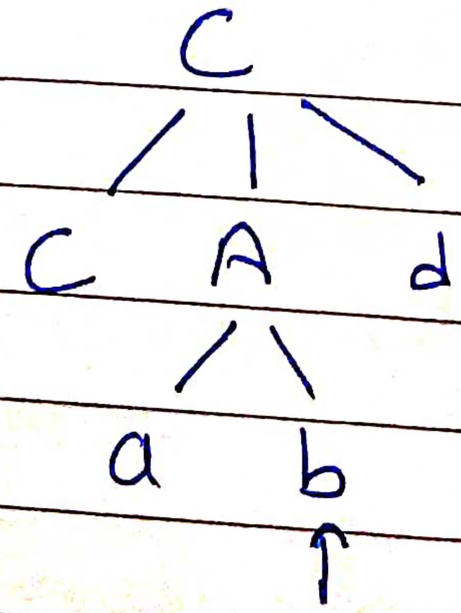
Match

Buffer = C a d
↑



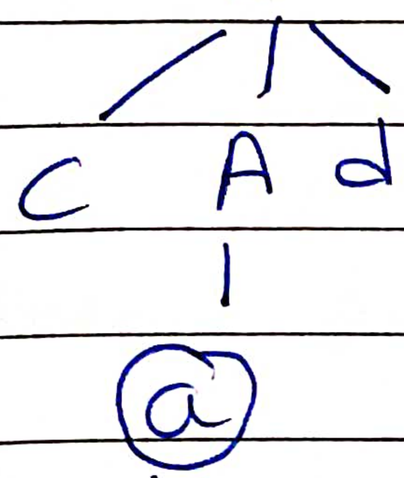
Match

Buffer = C a d
↑



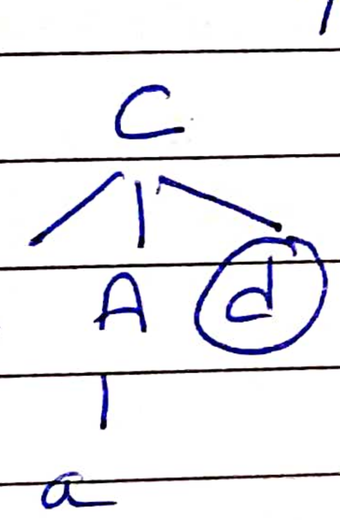
backtrack

Buffer = c a d
         ↑



Match

Buffer = c a d
           ↑



match and the string is generated

This is the parse tree generated by recursive Descent parser.

Leftmost Recursion

$A \rightarrow A\alpha$

$A \rightarrow A\alpha 1 \mid A\alpha 2 \mid A\alpha 3 \ldots \beta 1 \mid \beta_2 \ldots$

Remove left Recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Example 1:

$A \rightarrow \underbrace{ABd}_{\alpha 1} \mid \underbrace{Aa}_{\alpha_2} \mid \underbrace{a}_{\beta 1}$

$\Rightarrow A \rightarrow aA'$

$A' \rightarrow BdA' \mid aA' \mid \varepsilon$

Example 2:

$B \rightarrow \underbrace{Be}_{\alpha 1} \mid \underbrace{b}_{\beta_1}$

$\Rightarrow B \rightarrow bB'$

$B' \rightarrow eB' \mid \varepsilon$

Example 3:

$E \rightarrow \underbrace{E+E}_{\alpha 1} / \underbrace{E \times E}_{\alpha 2} / \underbrace{a}_{\beta_1}$

$E \rightarrow aE'$

$E' \rightarrow +EE' \mid \times EE' \mid \varepsilon$

=> Example 4 : (Indirect left recursion)

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

Solution :

Substitute production 1 in $A \rightarrow Sd$

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

The Grammer will be

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

So,

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

=> Example 4:

$A \rightarrow Ba \mid Aa \mid c$
$B \rightarrow Bb \mid Ab \mid d$

Solution:

Step1: Remove left recursion for A

$A \rightarrow cA' \mid BaA'$
$A' \rightarrow aA' \mid \varepsilon$

Step2: Substitute A in $B \rightarrow Ab$

$B \rightarrow Bb \mid cA'b \mid BaA'b \mid d$

Step3: Remove left recursion for B

$B \rightarrow cA'bB' \mid dB'$
$B' \rightarrow bB' \mid aA'bB' \mid \varepsilon$

Grammer after removal left recursion

$A \rightarrow cA' \mid BaA'$
$A' \rightarrow aA' \mid \varepsilon$
$B \rightarrow cA'bB' \mid dB'$
$B' \rightarrow bB' \mid aA'bB' \mid \varepsilon$

Page :
Date :

Examples: $A \rightarrow A+B / B$

$A \rightarrow BA'$

$A' \rightarrow +BA'$

$A' \rightarrow \varepsilon$

Example 6: $E \rightarrow E+T / T$

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \varepsilon$

Example 7: $A \rightarrow AB / AC / aB / cd$

$A \rightarrow aBA' / cdA'$

$A' \rightarrow BA' / CA' / \varepsilon$

Left Factoring the Grammer

- Lefft factoring is a grammer transformation that is useful for producing a grammer suitable for predictive or top-down parsing

- Consider following grammer:

Stmt → if expr then Stmt elee Stmt
| if expr then Stmt

- on seeing input if id is not clear for the parser which production to use

If we have

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$

then we replace it with

$A \to \alpha A' \mid \gamma$

$A' \to \beta_1 \mid \beta_2$

Example:

$$S \to \underbrace{aab}_{\alpha\ \beta} \mid \underbrace{aaA}_{\alpha\ \beta} \mid \underbrace{c}_{\gamma}$$

$$A \to c$$

Resultant grammar after doing left factoring

$$S \to aaS' \mid c$$
$$S' \to b \mid A$$
$$A \to c$$

Example 2:

$$S \to iEtS \mid iEtSeS \mid a$$
$$E \to b$$

Solution:

$\alpha = iEtS \quad \gamma = a$

$\beta_1 = \varepsilon$

$\beta_2 = eS$

Resultant grammar after left factoring

$$S \to iEtSS' \mid a$$
$$S' \to eS \mid \varepsilon$$
$$E \to b$$

HW

Example 3:

S → aAd / aB
A → a / ab
B → ccd / ddc

Solution:

S → aS'
S' → Ad | B
A → aA'
A' → ∈psilon | b
B → ccd | ddc

Rules for First and follow

1. Rules for Follow:

Follow is a function which gives set of terminals that can appear immediately to the right of given symbol

Rules:

a. Place $ in Follow(S), where S is the start symbol, and $ is the input right endmarker.
b. If there is a production A → α Bβ, then everything in first(β) except ∈ is in Follow(B)
c. If there is a production A → α B, or a production A → α Bβ, where FIRST(β) contains ∈, then everything in FOLLOW(A) is in FOLLOW(B).

2. Rules for first:

1. If X is terminal then First (X) = X
2. If $X \to \varepsilon$ is a production, then add ε to the Set of first[X].
3. If X is non terminal then,
   a. if $X \to Y$, first(Y) is an element of one Set of first(X)
   b. If first(Y) has ε as an element and $X \to YZ$ then

$$first(X) = first(Y) - \varepsilon \cup first(Z)$$

		first	Follow
1.	$S \to KMNOP$	{R, m, n}	{$}
	$K \to R \mid \varepsilon$	{R, ε}	{m, n}
	$M \to m \mid \varepsilon$	{m, ε}	{n}
	$N \to n$	{n}	{o, p, $}
	$O \to o \mid \varepsilon$	{o, ε}	{p, $}
	$P \to p \mid \varepsilon$	{p, ε}	{$}

		first	Follow
2.	$S \to aBDh$	{a}	{$}
	$B \to cC$	{c}	{g, f, h}
	$C \to bc \mid \varepsilon$	{b, ε}	{g, f, h}
	$D \to EF$	{g, f, ε}	{h}
	$E \to g \mid \varepsilon$	{g, ε}	{f, h}
	$F \to f \mid \varepsilon$	{f, ε}	{h}

3.

	First	Follow
$S \to B\,b \mid D\,d$	$\{a, b, c, d\}$	$\{\$\}$
$B \to a\,B \mid \varepsilon$	$\{a, \varepsilon\}$	$\{b\}$
$D \to c\,D \mid \varepsilon$	$\{c, \varepsilon\}$	$\{d\}$

4.

	First	Follow
$S \to A\,C\,B \mid C\,b\,B \mid B\,a$	$\{d, g, h, \varepsilon, b, a\}$	$\{\$\}$
$A \to d\,a \mid B\,C$	$\{d, g, h, \varepsilon\}$	$\{h, g, \$\}$
$B \to g \mid \varepsilon$	$\{g, \varepsilon\}$	$\{\$, a, h, g\}$
$C \to h \mid \varepsilon$	$\{h, \varepsilon\}$	$\{g, \$, b, h\}$

		First	Follow
5	S → ABC DE	{a, b, c}	{$}
	A → a | ε	{a, ε}	{b, c}
	B → b | ε	{b, ε}	{c}
	C → c	{c}	{d, e, $}
	D → d | ε	{d, ε}	{e, $}
	E → e | ε	{e, ε}	{$}

6	S	First	Follow
	S → iEtSS' | a	i, a	{$, e}
	S' → eS | ε	e, ε	{$, e}
	E → b	b	{t}

I.		First	Follow
	E → TE'	{(, id}	{$,)}
	E' → +TE' | ε	{+, ε}	{$,)}
	T → FT'	{(, id}	{+, $,)}
	T' → *FT' | ε	{*, ε}	{+, $,)}
	F → (E) | id	{(, id}	{*, +, $,)}

8.		first	Follow
°	S → A B C	{a, b, c, d, e, f, ε}	{$}
	A → a | b | ε	{a, b, ε}	{c, d, e, f, $}
	B → c | d | ε	{c, d, ε}	{e, f, $}
	C → e | f | ε	{e, f, ε}	{$}

Predictive Parser or LL(1) Parser

- Predictive Parsers are those recursive decent Parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means Scanning input from left to right
  - The second L means leftmost derivation.
  - and 1 Stands for using One input Symbol for lookahead.

LL(1) Parser Numericals

1. $E \rightarrow E+T \mid T$
   $T \rightarrow T*F \mid F$
   $F \rightarrow (E) \mid id$

Input → id+id*id$

Step 1: Remove Left Recursion

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

Step2: First & Follow

	first	Follow
$E \to TE'$ (1)	{(, id}	{$,)}
$E' \to +TE'$ (2) | ε (3)	{+, ε}	{$,)}
$T \to FT'$ (4)	{(, id}	{+, $,)}
$T' \to *FT'$ (5) | ε (6)	{*, ε}	{+, $,)}
$F \to (E)$ (7) | id (8)	{(, id}	{*, +, $,)}

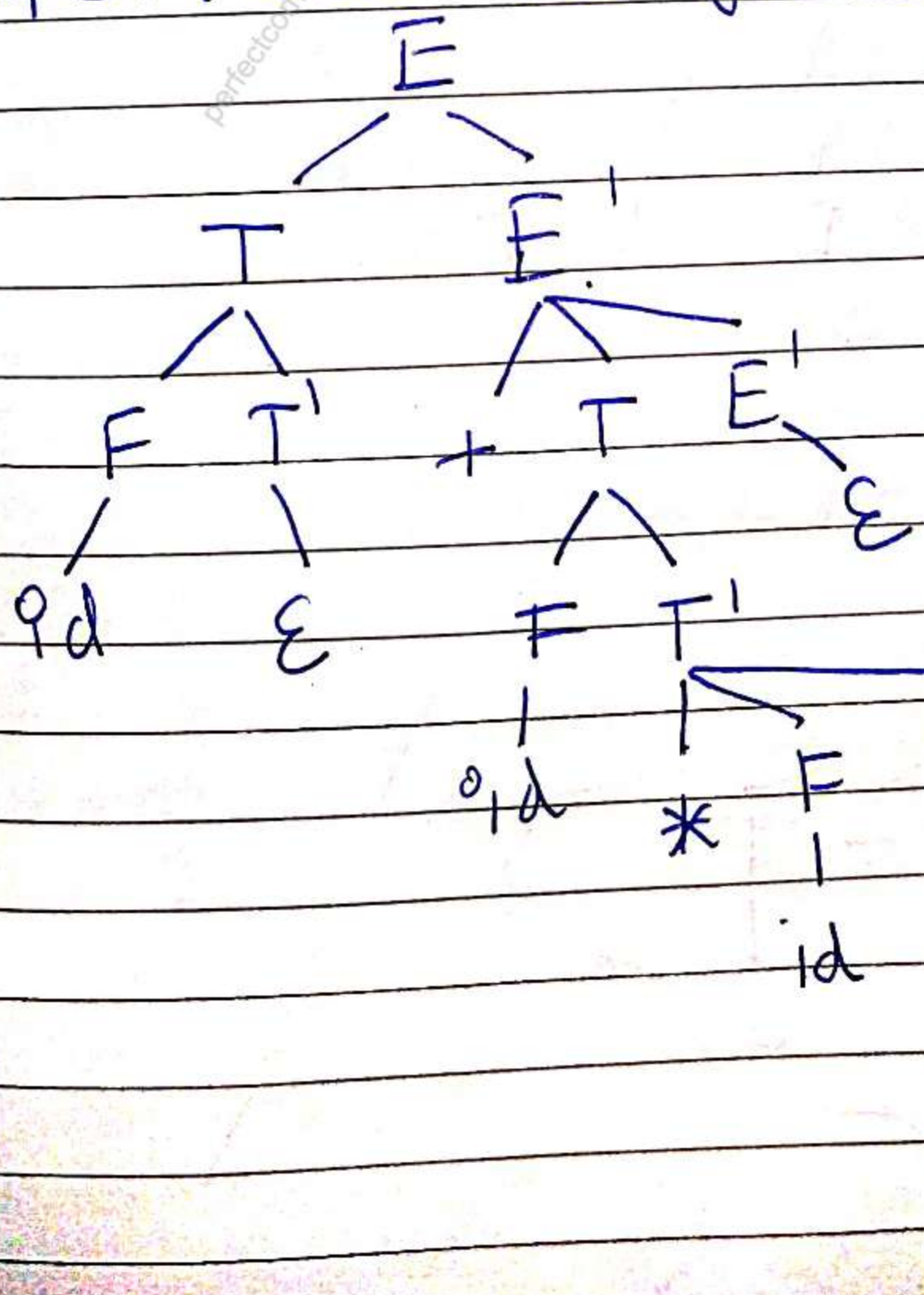
Step3: Parsing Table

	+	id	*	(	)	$
E		1		1		
E'	2				3	3
T		4		4		
T'	6		5		6	6
F		8		7		

Step 4: Stack implementation

Stack	Input	Implementation
E $	id+id*id$	
TE'$	id+id*id$	E → TE'
FT'E'$	id+id*id$	T → FT'
idT'E'$	id+id*id$	F → id
T'E'$	+id*id$	T' → ε
+TE'$	+id*id$	E' → +TE'
FT'E'$	id*id$	T → FT'
idT'E'$	id*id$	F → id
*FT'E'$	*id$	T' → *FT'
idT'E'$	id$	F → id
T'E'$	$	T' → ε
E'$	$	E' → ε
$	$	

Step 5: Parse Tree generation.



3. S → aBh | Ce
B → cC
C → bd | ε

Step 1 : First and Follow

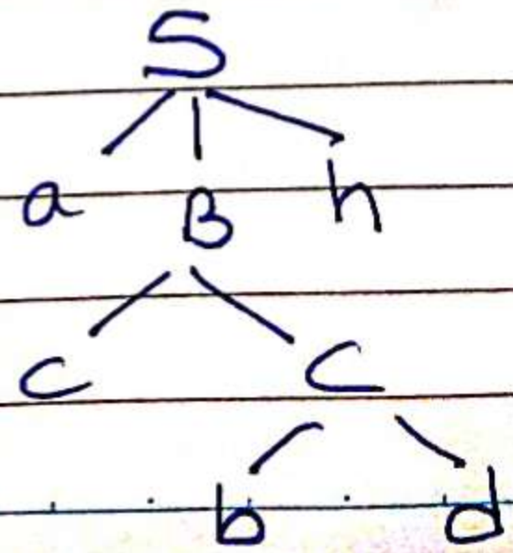
	1	2	First	Follow
S → aBh			{a, b, e}	{$}
B → cC	3		{c}	{h}
C → bd	4	5	{b, ε}	{e, h}
ε				

Step 2 : Generate a parsing table.

	a	b	c	d	e	h	$
S	1		2		2		
B			3				
C		4			5	5	

Step 3 :

Stack	input	Implementation
S$	acbdh$	
aBh$	acbdh$	S → aBh
cCh$	cbdh$	B → cC
~~bdh$~~	~~bdh$~~	C → bd
$	$	



4. $S \to aBDh$
$B \to cC$
$C \to bC \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

~~Inpu~~
String: acbcgfh$

Step 1: First and follow

	First	Follow
$S \to aBDh$	{a}	{$,}
$B \to cC$	{c}	{g, f, h}
$C \to bC \mid \varepsilon$	{b, ε}	{g, f, h}
$D \to EF$	{g, f, ε}	{h}
$E \to g \mid \varepsilon$	{g, ε}	{f, h}
$F \to f \mid \varepsilon$	{f, ε}	{h}

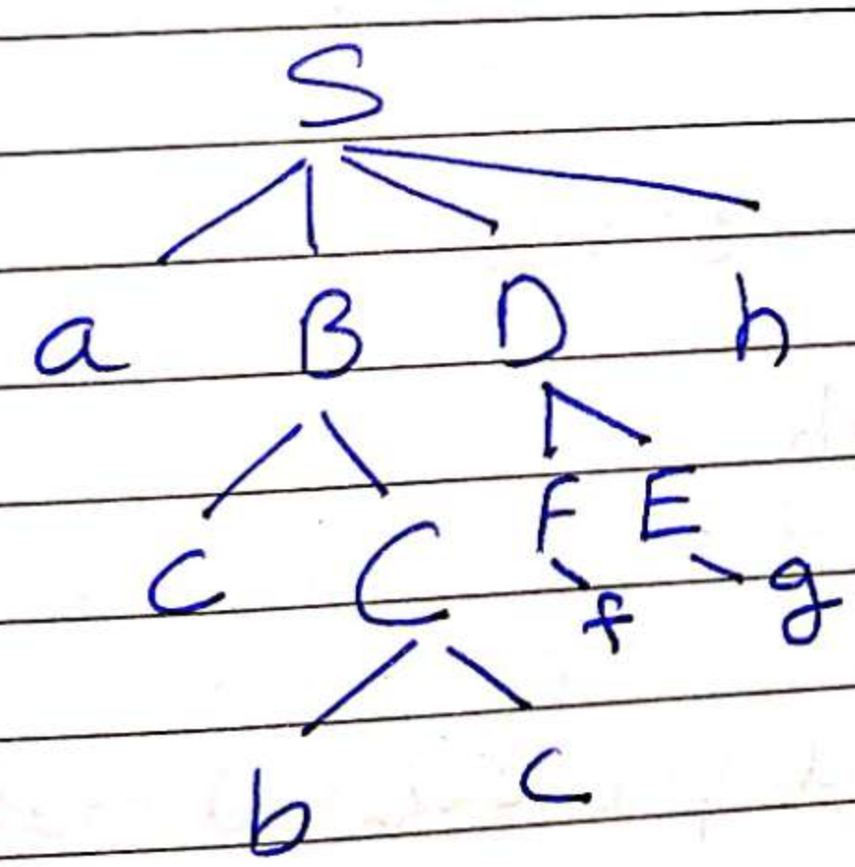
Step 2: Create a parsing table

	a	b	c	d	e	g	f	h	$
S	1								
B			2						
C		3				4	4	4	
D						5	5	5	
E						6	7	7	
F							8	9	

Step 3:

Stack	& input	Implementation
S$	acbcgfh$	
aBDh$	acbcgfh$	S→aBDh
cCDh$	cbcgfh$	B→cC
bcDh$	bcgfh$	C→bC
EFh$	gfh$	D→EF
gFh$	gfh$	E→g
fh$	fh$	F→f
$	$	

Step 4: parse tree



5. $S \to iEtSS' \mid a$
$S' \to eS \mid \varepsilon$
$E \to b$

Step1: first & Follow

	first	Follow
$S \to iEtSS' \mid a$	$\{i, a\}$	$\{\$, e\}$
$S' \to eS \mid \varepsilon$	$\{e, \varepsilon\}$	$\{\$, e\}$
$E \to b$	$\{b\}$	$\{t\}$

step2: Parsing table

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

As there are multiple defined entry in the table the grammer is not LL1

Bottom up parsing

- Bottom-up parsing starts from the leaf nodes of a tree and works in upword direction till it reaches the root node.

- We start from a sentence and then apply production rules in reverse manner in order to reach the start symbol

- Here parser tries to identify R.H.S of production rule and replace it by corresponding LHS. This activity is ~~known~~ known as reduction

- Also known as LR parser, where L means tokens are ~~used~~ read from left to right and R means that it constructs rightmost derivative.

- A general style of bottom up syntax analysis, known as SR (Shift Reduce) parsing.

- Two types of bottom up parsing:
  1. Operator Precedence parsing
  2. LR Parsing.

Operator Precedence Parser.

- Small, but an important class of grammars
- We may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.

- In an operator grammar, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.

Example

$E \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b \mid \varepsilon$

Not operator grammar

$E \rightarrow EOE$
$E \rightarrow id$
$O \rightarrow + \mid * \mid /$

Not operator grammar

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

operator grammar.

- In ~~order~~ operator precedence parsing we difine three disjoint precedence relations between certain pairs of terminals.

$a \lessdot b$	b has higher precedence than a
$a \doteq b$	b has same precedence as a
$a \gtrdot b$	b has lower precedence than a.

Note:
- id has higher precedence than any other symbol
- $ has lowest precedence
- if two Operators have equal precedence, then we check the associativity of that particular Operator.

- The intension of the precedence relations is to find the handle of a right-Sentential form
  - $\lessdot$ with marking the left end.
  - $\doteq$ appearing in the interior of the handle,
  - $\gtrdot$ marking the right hand

- In our Input string $\$a_1a_2 \ldots a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in the pair).
- Precedence function: compilers using operator precedence parsers do not need to store the table of precedence relations. The table can be encoded by two precedence functions f and g that map terminals symbols to integers.
- for symbols a and b:
  $f(a) \lessdot g(b)$ whenever $a \lessdot b$
  $f(a) = g(b)$ whenever $a \doteq b$
  $f(a) \gtrdot g(b)$ whenever $a \gtrdot b$.

1. T → T+T / T*T / id
String : id+id*id

Step 1. Operator Precedence Relation table

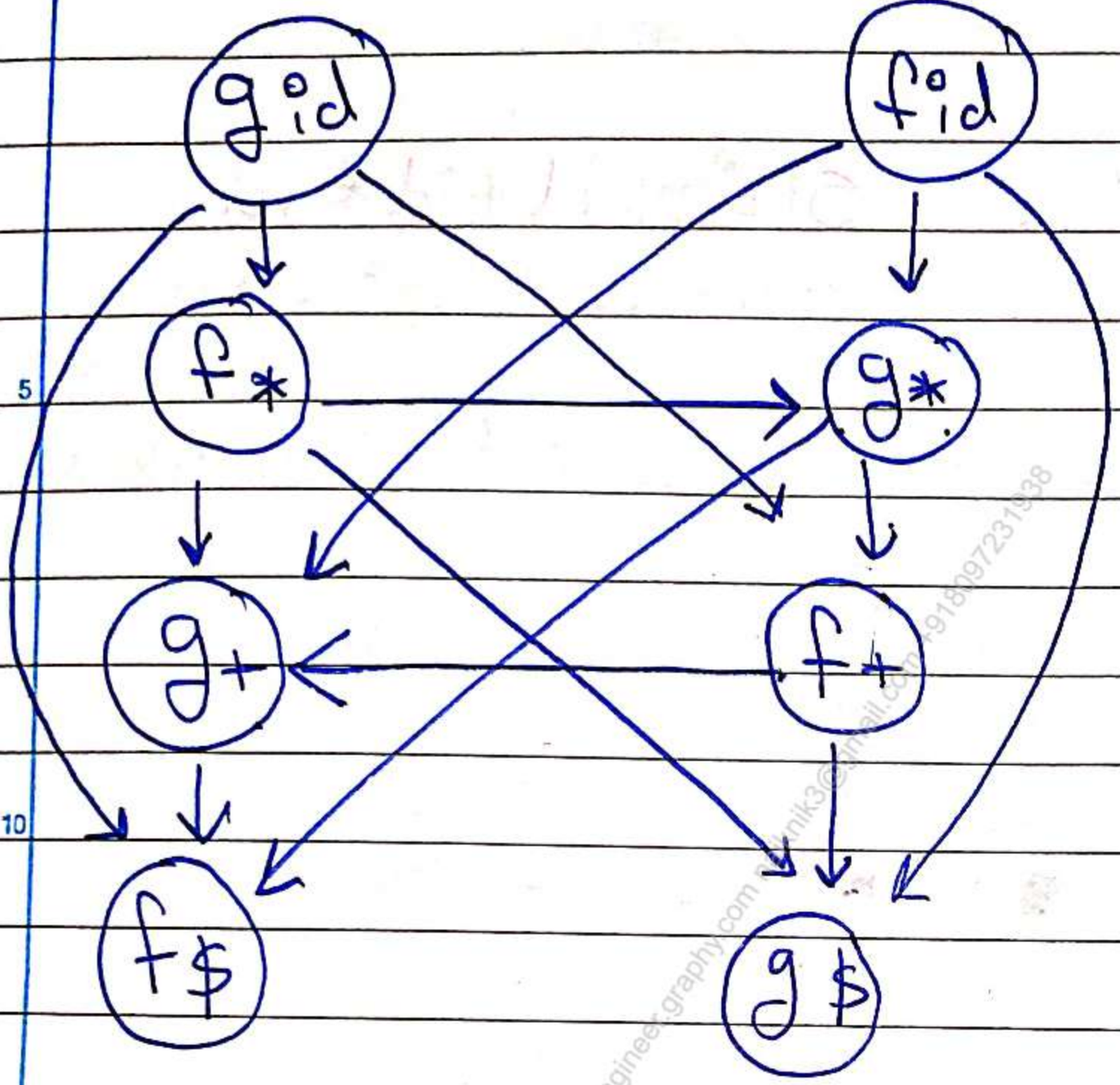
	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	-	>
$	<	<	<	A

Stack	Input	comment
$	id+id*id$	Shift id
$id	+id*id$	Reduce T→id
$T	+id*id$	Shift +
$T+	id*id$	Shift id
$T+id	*id$	Reduce T→id
$T+T	*id$	Shift *
$T+T*	id$	Shift id
$T+T*id	$	Reduce T→id
$T+T*T	$	Reduce T→T*T
$T+T	$	Reduce T→T+T
$T	$	Accept

Step 2: Operator precedence table

Step 3: Precedence function

Page: 42
Date:



Step 4: Table showing the longest path.

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

Shift-Reduce Parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied

- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production

- The key decisions during bottom up parsing are about when to reduce and about what production to apply

- A reduction is a reverse of a step in a derivation

- The goal of a bottom up parser is to construct a derivation in reverse:
  - E → T → T*F → T*id → F*id → id*id

1. Grammar:

$T \to T+T$

$T \to T-T$ String: $C_1-(C_2+C_3)$

$T \to (T)$

$T \to c$

Stack	input	Action
\$	$c_1-(C_2+c_3)\$$	Shift c_1
$\$c_1$	$-(c_2+c_3)\$$	Reduce $T \to c$
\$T	$-(c_2+c_3)\$$	Shift -
\$T-	$(c_2+c_3)\$$	Shift c
\$T-c	$c_2+c_3)\$$	Shift c_2
$\$T-(c_2$	$+c_3)\$$	Reduce $T \to c$
\$T-(T	$+c_3)\$$	Shift +
\$T+(T+	$c_3)\$$	Shift
$\$T+(T+c_3$	)\$	Reduce $T \to c$
\$T+(T+T	)\$	Reduce $T \to T+T$
\$T+(T	)\$	Shift
\$T+(T)	\$	Reduce $T \to (T)$
\$T+T	\$	Reduce $T \to T+T$
\$T	\$	Accept

2. Grammar :

A → 5A5 String : 75957

A → 7A7

A → 9

Stack	Input	Action
$	75957$	Shift
$7	5957$	Shift
$75	957$	Shift
$759	57$	Reduce A→9
$75A	57$	Shift
$75A5	7$	Reduce A→5A5
$7A	7$	Shift
$7A7	$	Reduce A→7A7
$A	$	Accept

LR Parser

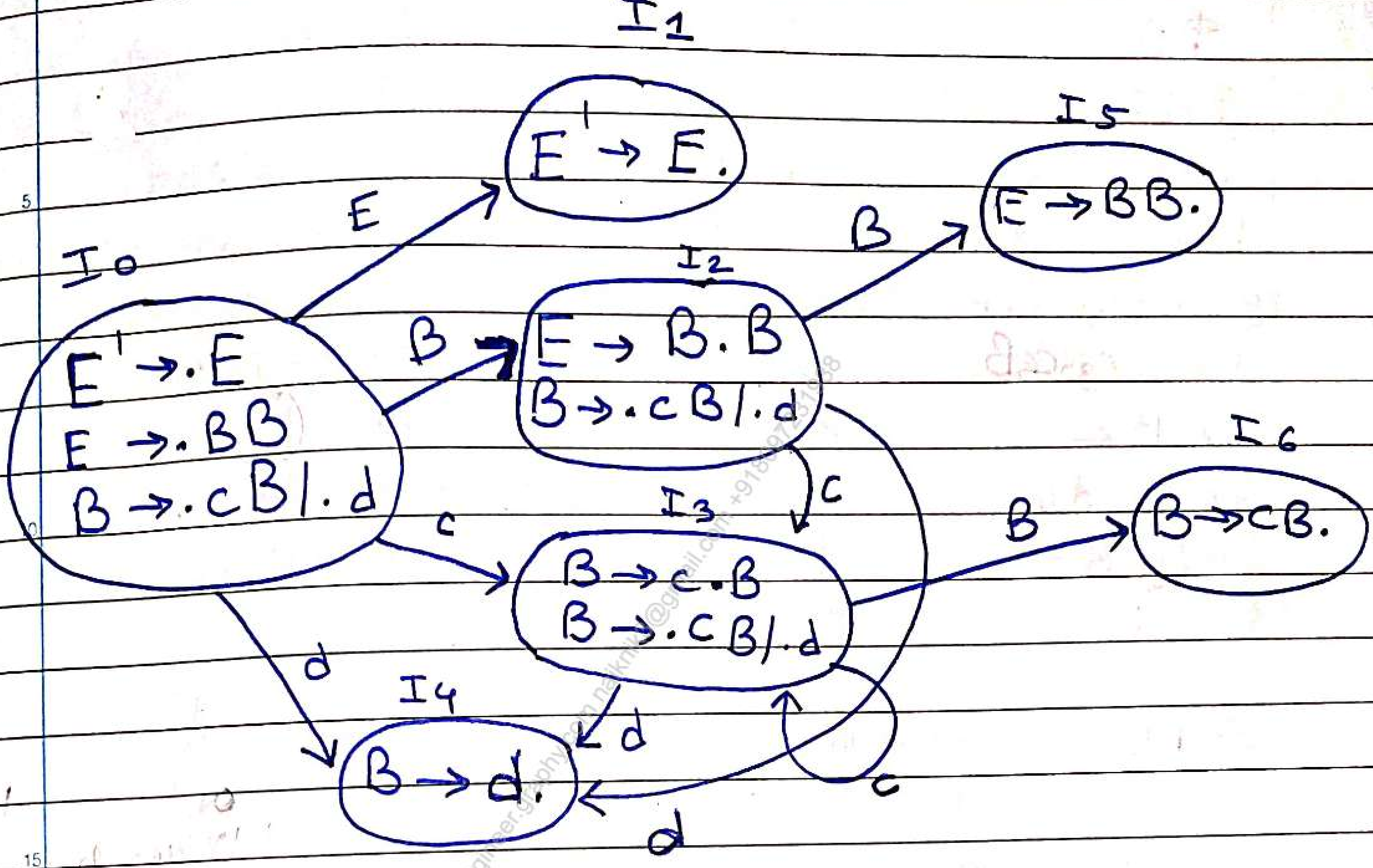
E → BB
B → cB | d

String : ccdd$

Step 1: Augment the given grammar

E' → E
E → BB
B → cB | d

Step2: Draw the DFA diagram



Numbering the productions

$E' \rightarrow E$
$E \rightarrow BB$ (1)
$B \rightarrow cB \mid d$
(2) (3)

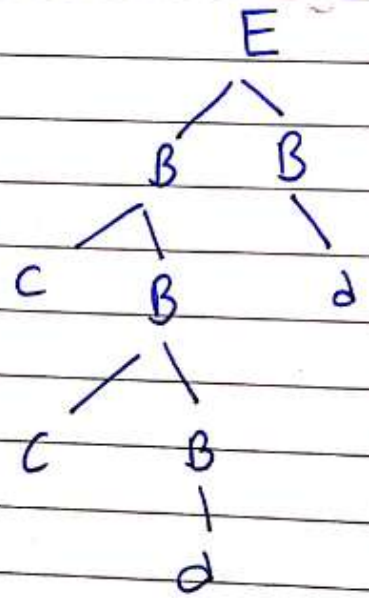
step3: Parsing table

State	Action			Goto	
	c	d	\$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Step 4: Stack Implementation

Stack	Input	Action
$0	ccdd$	Shift c → S_3
$0 C_3	cdd$	Shift c → S_3
$0 $C_3 C_3$	dd$	Shift d → S_4
$0 $C_3 C_3 d_4$	d$	~~Shift~~ reduce r_3 B → d
$0 $C_3 C_3 B_6$	d$	reduce B → cB
$0 $C_3 B_6$	d$	reduce B → cB
$0 B2	d$	Shift d → S_4
$0 $B_2 d_4$	$	reduce B → d
$0 $B_2 B_5$	$	reduce E → BB
$0 E_1	$	Accept

Step 5: Parse tree

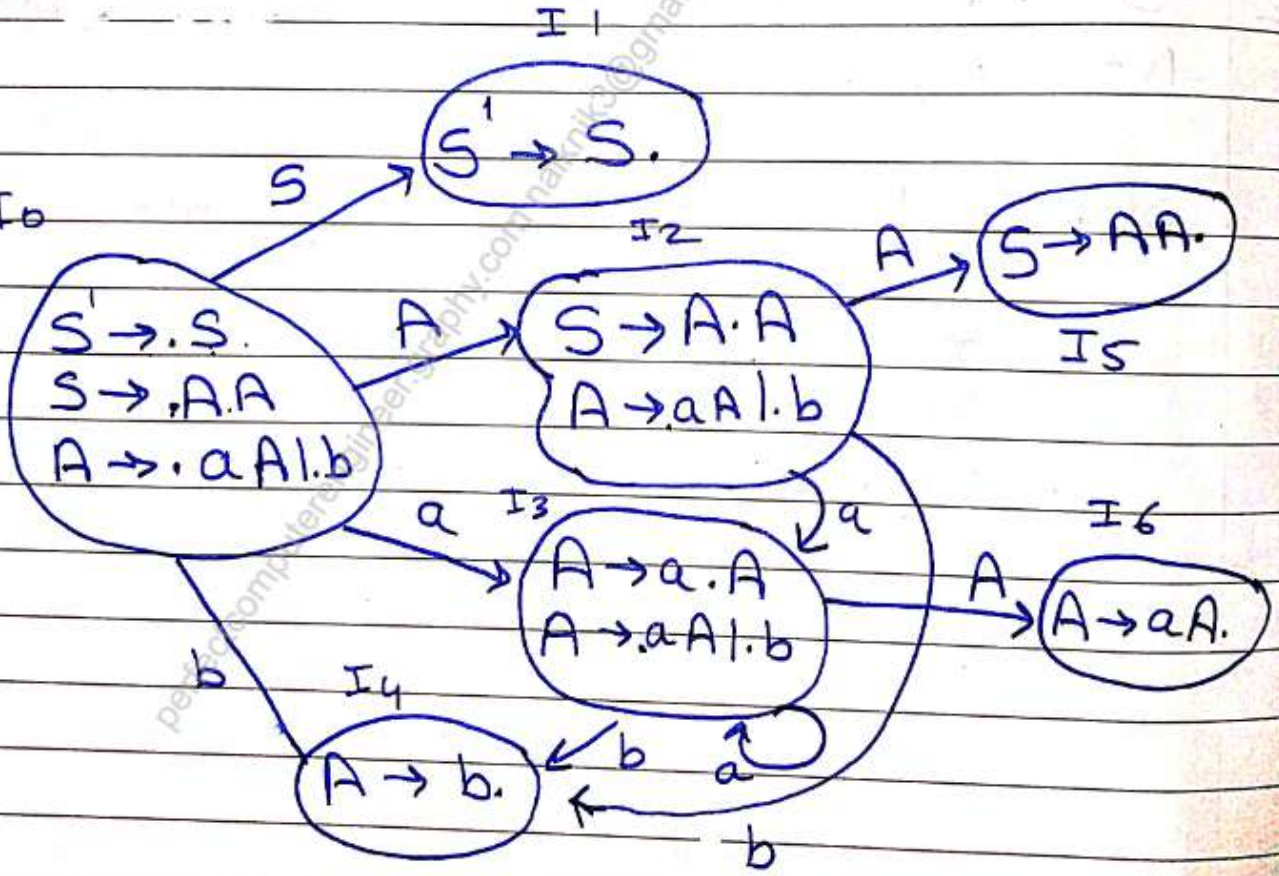


2. $S \to AA$ Check wether LR(0) or not
$A \to aA/b$

Step 1: Augment the given Grammar

$S' \to S$
$S \to AA$
$A \to aA/b$

Step 2: Draw the DFA diagram



	Action			goto	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accpt		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

$S \rightarrow AA$ ①

$A \rightarrow aA \mid b$

② ③

Yes the grammar is LR(0) grammar.

SLR Parser

- SLR Stands for Simple LR
- This is basically a method of adding lookahead to LR(0) Parsers as simply as possible
- The reduced productions are written only in the FOLLOW of the variable whose production is reduced.

1. S → cAd
   A → able
   String: ced $

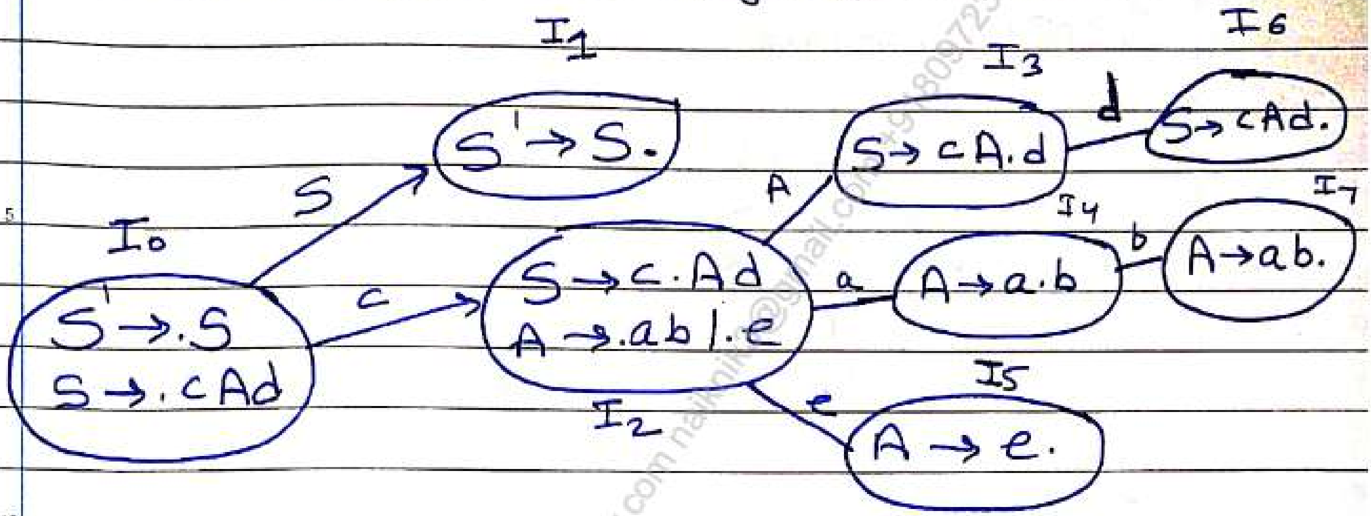
Step1: Create Augmented grammar

S' → S	0
S → cAd	1
A → able	2, 3

Step2: Compute first

	First	Follow
S' → S	c	$
S → cAd	c	$
A → able	a, e	d

Step3: construct DFA Diagram



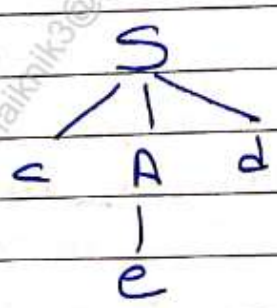
Step4: Parsing table

	Action						goto	
States	a	b	c	d	e	$	S	A
0			S2				1	
1						A		
2	S4				S5			3
3				S6				
4		S7						
5				r3				
6						r1		
7				r2				

4:00 Step 5: Stack Implementation

Stack	input	action
0	ced $	Shift c → S2
0c2	ed $	Shift e → S5
0c2 e5	d$	r: A → e
0c2 A 3	d $	shift d → S6
0 c2 A3 d6	$	r: S → cAd
0 S1	$	Accept

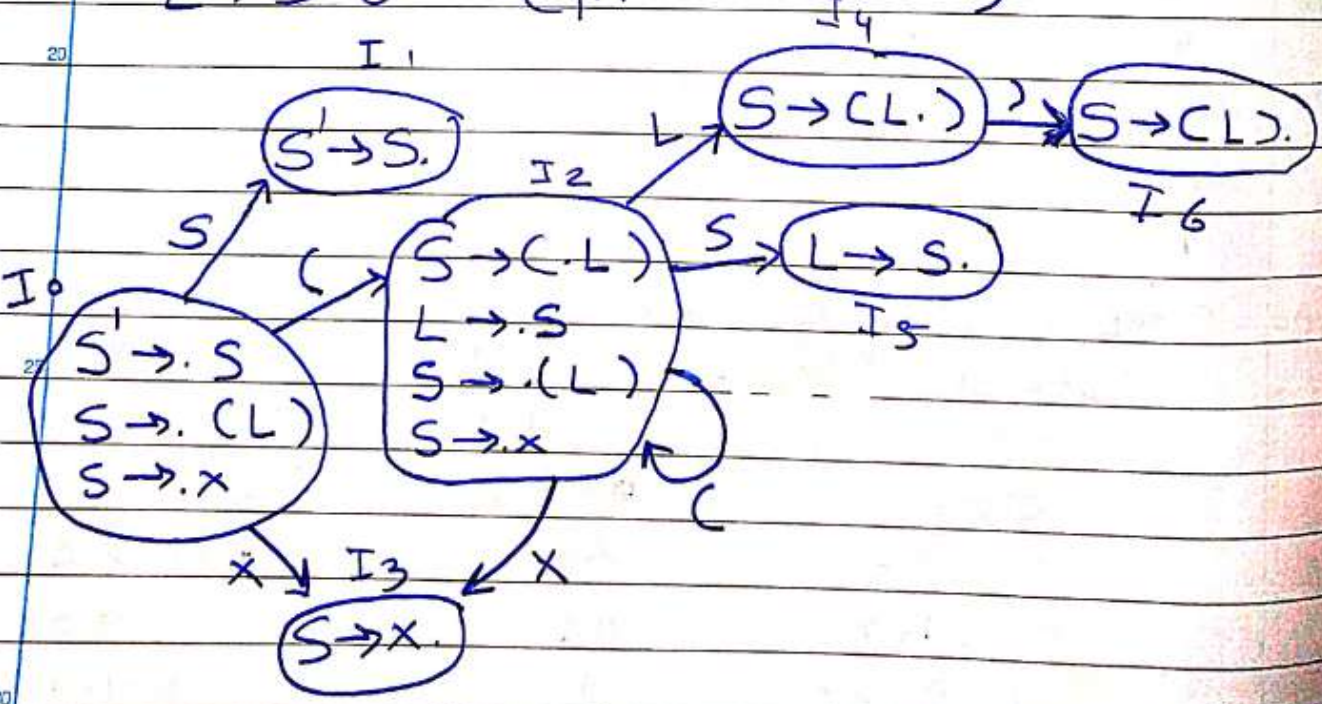
Step6: Parse tree



2. $S \to (L)$
$S \to x$
$L \to S$

Step 1: Augment the grammar & Step2: First & Follow

		first	Follow
$S' \to S$	⓪	(, x	$
$S \to (L)$	①	(, x	$,)
$S \to x$	②	x	$,)
$L \to S$	③	(, x	)



Step3: Construct A DFA diagram

Step 4 : Parsing Table

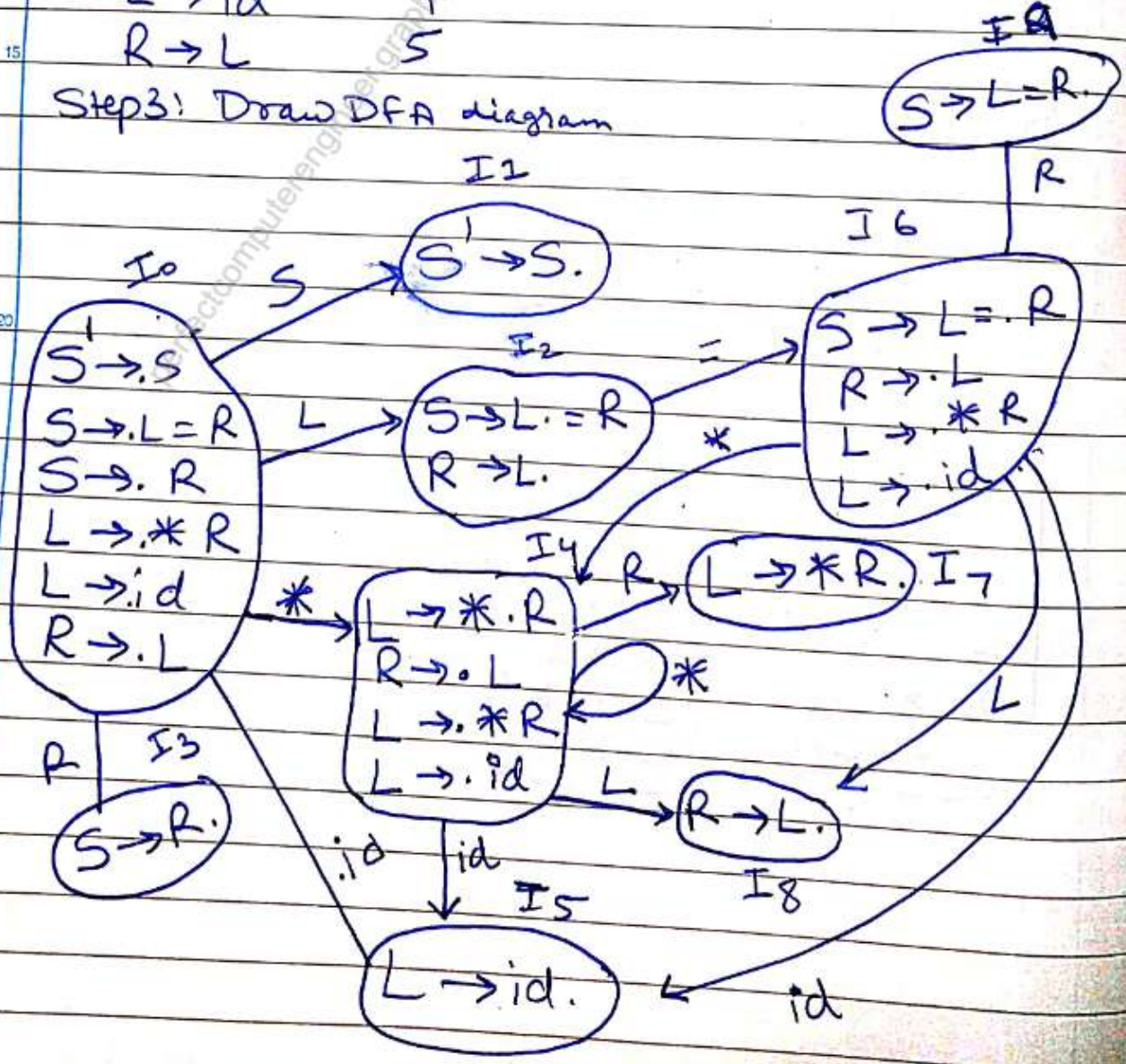
States	Action				Go to	
	(	)	x	$	S	L
I_0	S_2		S_3		1	
I_1				A		
I_2	S_2		S_3		5	4
I_3		r_2		r_2		
I_4		S_6				
I_5		r_3				
I_6		r_1		r_1		

3. $S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

Step1: Augment the grammar and calculate follow

$S' \to S$	0	FOLLOW(S) = {$}
$S \to L = R$	1	FOLLOW(R) = {$, =}
$S \to R$	2	FOLLOW(L) = {=, $}
$L \to *R$	3	
$L \to id$	4	
$R \to L$	5	

Step3: Draw DFA diagram



Step 4: Parsing table

State	Action				GoTo		
	*	=	id	$	S	L	R
I_0	S4		S5		1	2	3
I_1				A			
I_2	S6/r5			r5			
I_3				r2			
I_4	S4		S5			8	7
I_5	r4			r4			
I_6	S4		S5			8	9
I_7	r3			r3			
I_8	r5			r5			
I_9				r1			

Shift and reduce conflict so it is not SLR

=> ~~re~~ Note: reduce reduce conflicts are also possible

for these conflict other two parsers are keen designed CLR and LALR

LALR and CLR parsers are explained on youtube Channel.

Macros and Macroprocessor

Macro → Introduction

- A macro instruction is a notational convenience for the programmar.
- It allows the programmer to write shorthand version of a program (Module programming)
- The macro processor replaces each macro invocation with the corrosponding sequence of statements. (expanding)
- When assembly language programmer wants to repeat some blocks of code many times in the course of a program
- Block can be - code to save / exchange sets of registers or perform series of arithmetic operations.
- Macro instructions are single - line abbreviations for groups of instructions.
- Macro instructions are considered as extension of the basic assembler language and macro processor viewed as an extension of basic assembler algorithm.

Example:

```
        .
        .
        .
A    1, DATA   add contents of DATA to register 1
A    2, DATA   add contents of DATA to register 2
A    3, DATA   add contents of DATA to register 3
        .
        .
        .
A    1, DATA   Add contents of DATA to register 1
A    2, DATA   Add contents of DATA to register 2
A    3, DATA   Add contents of DATA to register 3
        .
        .
        .
DATA DC F'5'
```

=> Source

```
MACRO
INCR
     A      1, DATA
     A      2, DATA
     A      3, DATA
MEND
        .
        .
        .
INCR
        .
        .
        .
INCR
        .
        .
        .
DATA DC F'5'
        .
        .
        .
```

Expanded Source

```
        .
        .
        .
{ A    1, DATA
{ A    2, DATA
  A    3, DATA
        .
        .
        .
{ A    1, DATA
{ A    2, DATA
  A    3, DATA
        .
        .
        .
DATA DC F'5'
```

Macro Instruction definition

Start of definition ⟶ MACRO

Macroname ⟶ ( )

Sequence to be abbreviated { — — — }

End of definition ⟶ MEND

One difference between a macro and function is, in function a value can be returned but in macro cannot return a value.

Macro Instruction Arguments

Example (Problem):

```
        .
        .
        .
A       1, DATA1
A       2, DATA1
A       3, DATA1
        .
        .
        .
A       1, DATA2
A       2, DATA2
A       3, DATA2
        .
        .
        .
DATA1   DC  F'5'
DATA2   DC  F'10'
```

Example (Solution)

we are passing 1 argument

Source	Expanded Source code
MACRO	
INCR &ARG	
A 1,&ARG	
A 2,&ARG	
A 3,&ARG	
MEND	
.	
INCR DATA1	A 1,DATA1 A 2,DATA1 A 3,DATA1
.	.
INCR DATA2	A 1,DATA2 A 2,DATA2 A 3,DATA2
.	
DATA1 DC F'5'	
DATA2 DC F'10'	
.	

2. Example (Problem)

```
      :
LOOP1  A   1, DATA1
       A   2, DATA2
       A   3, DATA3

LOOP2  A   1, DATA3
       A   2, DATA2
       A   3, DATA1
      :
      :
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
```

Example (Solution)

```
&LAB  JAZZ  &ARG1, &ARG2, &ARG3
&LAB  A     1, &ARG1
      A     2, &ARG2
      A     3, &ARG3
  MEND
    :
LOOP1 JAZZ DATA1, DATA2, DATA3
    :
LOOP2 JAZZ DATA3, DATA2, DATA1
    :
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
    :
```

Expanded Source

```
   :
LOOP1 A  1, DATA1
      A  2, DATA2
      A  3, DATA3
   :
LOOP2 A  1, DATA3
      A  2, DATA2
      A  3, DATA1
   :
```

Nested Macro calls

=> Example (Problem)

```
MACRO
ADD1   &ARG
A      1, &ARG
A      2, &ARG
A      3, &ARG
MEND
MACRO
ADDS   &ARG1, &ARG2, &ARG3
ADD1   &ARG1
ADD1   &ARG2
ADD1   &ARG3
MEND
```

(these 3 Instructions)

(Nested Macro call)

Example (Solution)

```
MACRO
ADD1   &ARG
A      1, &ARG
A      2, &ARG
A      3, &ARG
MEND
MACRO
ADDS   &ARG1, &ARG2, &ARG3
ADD1   &ARG1
ADD1   &ARG2
ADD1   &ARG3
MEND
ADDS   D1, D2, D3
D1     DC F'1'
D2     DC F'2'
D3     DC F'3'
```

Expansion Level 1

```
EXPANSION OF ADDS
ADD1   D1
ADD1   D2
ADD1   D3
```

(Not an Assembly Instruction)

Expansion Level 2

Expansion of ADD1

```
A   1, D1
A   2, D1
A   3, D1
```

```
A   1, D2
A   2, D2
A   3, D2
```

```
A   1, D3
A   2, D3
A   3, D3
```

Conditional Macro Expansion

```
        .
        .
        .
LOOP1 A     1, DATA1
      A     2, DATA2
      A     3, DATA3
        .
        .
LOOP2 A     1, DATA3
      A     2, DATA2
        .
        .
        .
LOOP3 A     1, DATA1
        .
        .
DATA1  DC   F'5'
DATA2  DC   F'10'
DATA3  DC   F'15'
        .
        .
```

Conditional Macro Expansion

```
        :
        MACRO
&ARGO   VARY  &COUNT, &ARG1, &ARG2, &ARG3
&ARGO   A     1, &ARG1
        AIF   (&COUNT EQ 1).FINI
        A     2, &ARG2
        AIF   (&COUNT EQ 2).FINI
        A     3, &ARG3
.FINI   MEND
        :
        :
LOOP1   VARY  3, DATA1, DATA2, DATA3
        :
        :
        :
        :
        :
LOOP2   VARY  2, DATA3, DATA2
        :
        :
        :
        :
        :
LOOP3   VARY  1, DATA1
        :
        :
        :
DATA1   DC    F'5'
DATA2   DC    F'10'
DATA3   DC    F'15'
```

```
LOOP1  A  1, DATA1
       A  2, DATA2
       A  3, DATA3
       :
       :
```

```
LOOP2  A  1, DATA3
       A  2, DATA2
```

```
LOOP3  A  1, DATA1
       :
       :
```

=> Two important macro procesor pseudo-ops: AIF, AGO permits conditional reordering of the sequence of macro expansion.

Allows conditional selection of machine instructions that appear in expansions of a macro call.

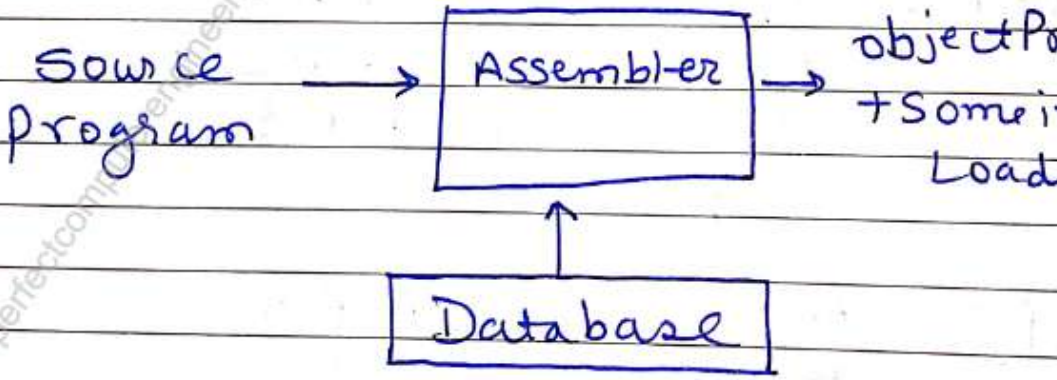
```
~~LOOP~~          :

LOOP1   A       1, DATA1
        A       2, DATA2
        A       3, DATA3
                :

LOOP2   A       1, DATA3
        A       2, DATA2
                :

LOOP3   A       1, DATA1
                :

DATA1   DC      F'5'
DATA2   DC      F'10'
DATA3   DC      F'15'
                :
```

Solution in the Page behind

Introduction to System Programming

Assembler

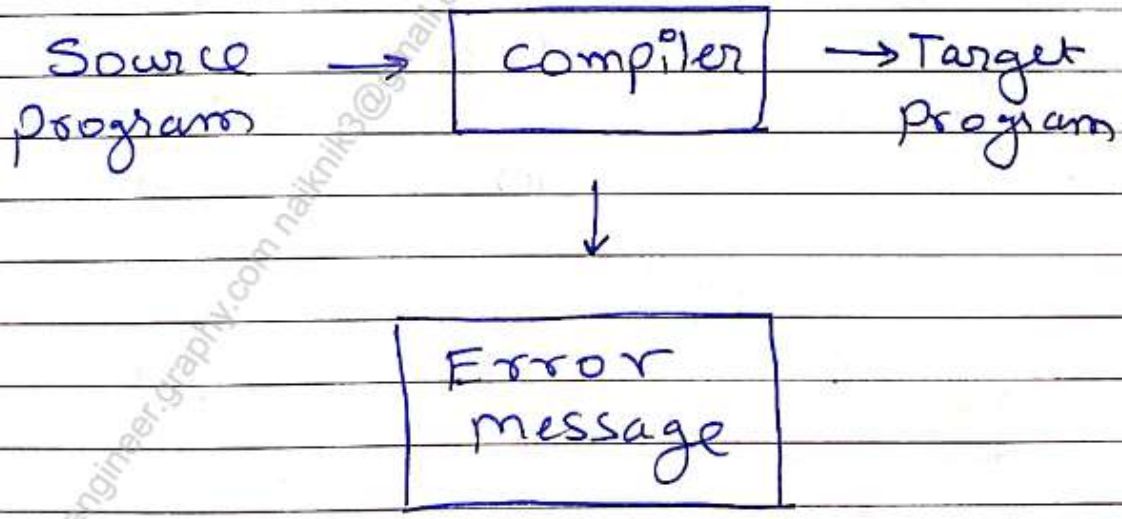
- Assembler is a system program which acts as a translator for translating assembly language code into machine code.
- Assemblers accept the assembly language program and generate its equivalent machine language program



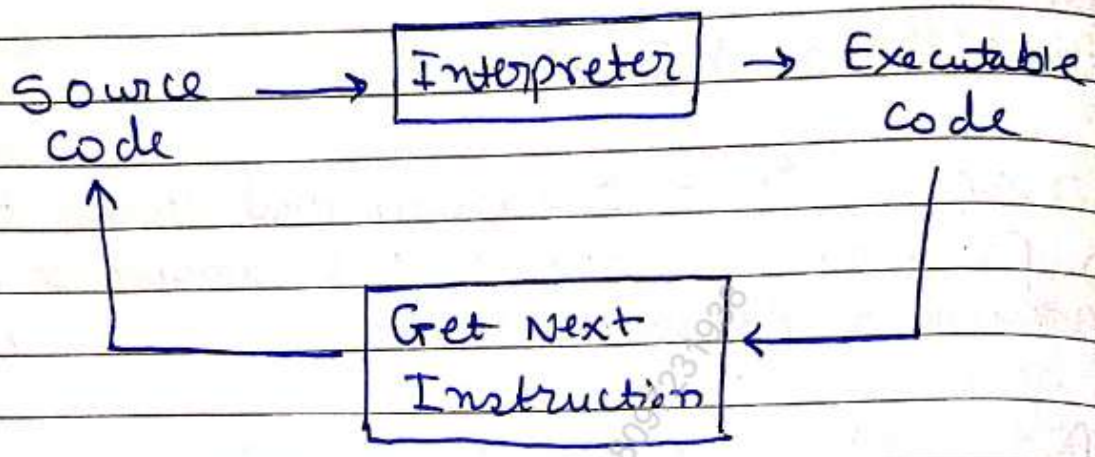
- The input assembly language program is also called as source program and the generated output is referred as object Program.
- Along with the object program, assembler also generates some useful information for loader.

compiler and Enterpreter

- compiler is a program that converts a program written in one language into another ~~Programs~~ language.
- A target language may be any other programming language or is a machine language
- During translation process, compiler also reports presence of errors, if any

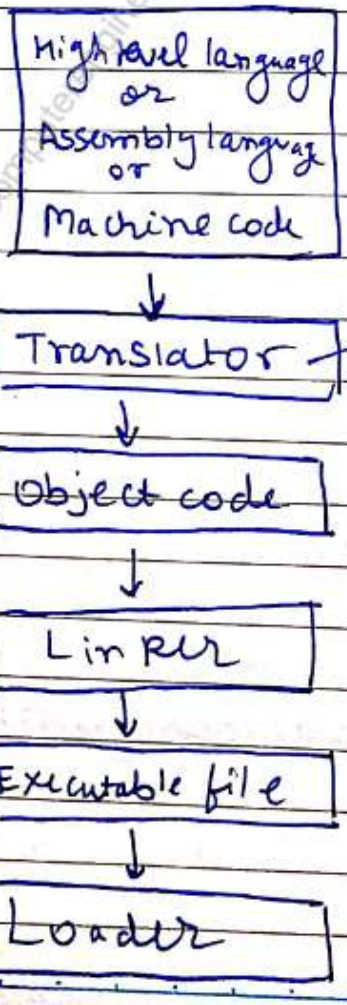


- The job of Interpreter is same as that of compiler.
- Thus an interpreter is a program that translates high level program code line by line into intermediate language code and Execute it.



#10 Loader and Linker

- It loads this program into the memory and then transfers the control of the first instruction of the program in order to initiate its execution.

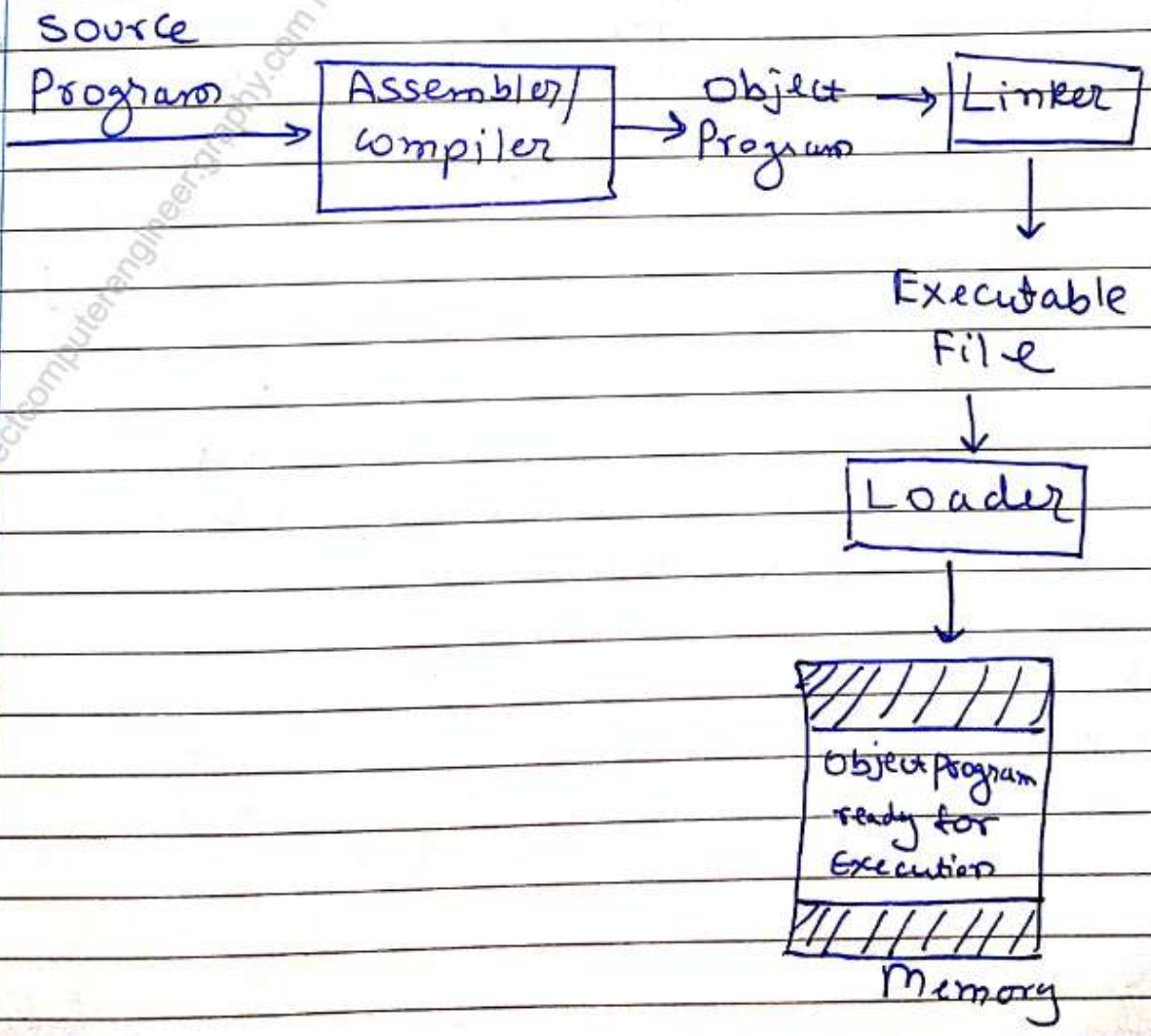


Note: few topics are discussed here in short. They are going to be discussed in detail in upcomming chapters.

- Linker is system software that combines two or more Object programs and supplies information needed to allow references between them.

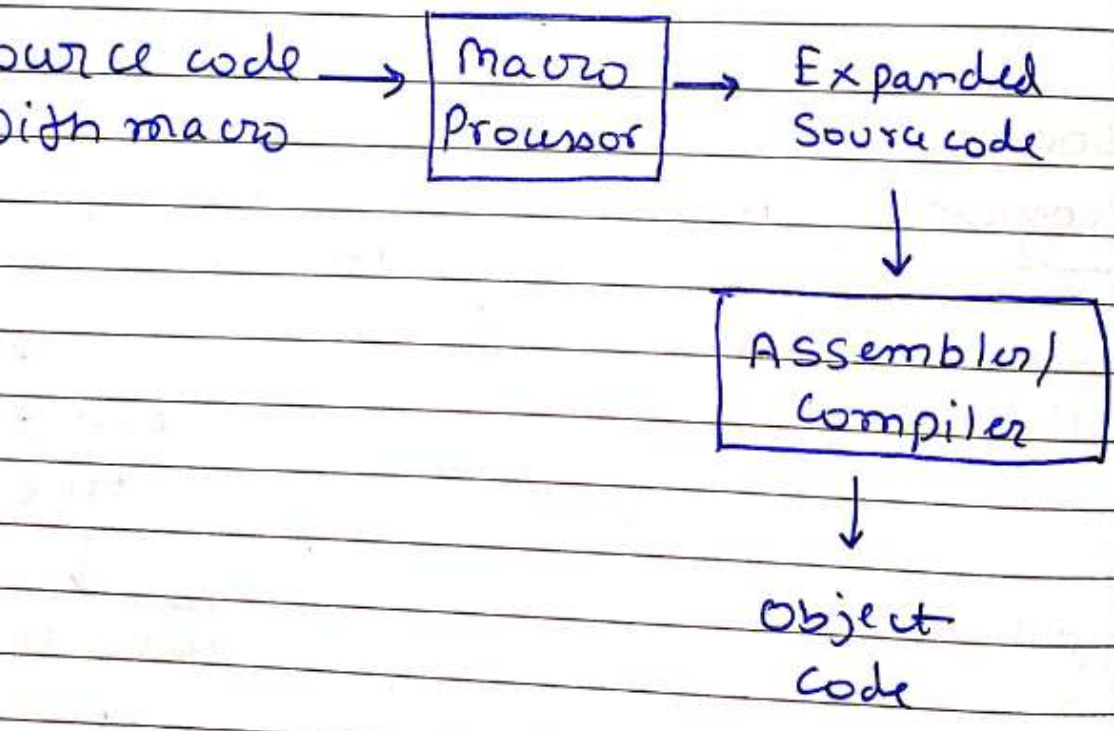
- Thus linker is system software used to link the functions, resources to their respective references.

- It performs the last step of converting the object file into the machine executable form



Macro

- Macro is a single line abbreviation for the series of repeated instruction.
- Simply it is a name given to some block of code which is repeated in a program
- The macro instructions are processed by the macro processor.



Operating System

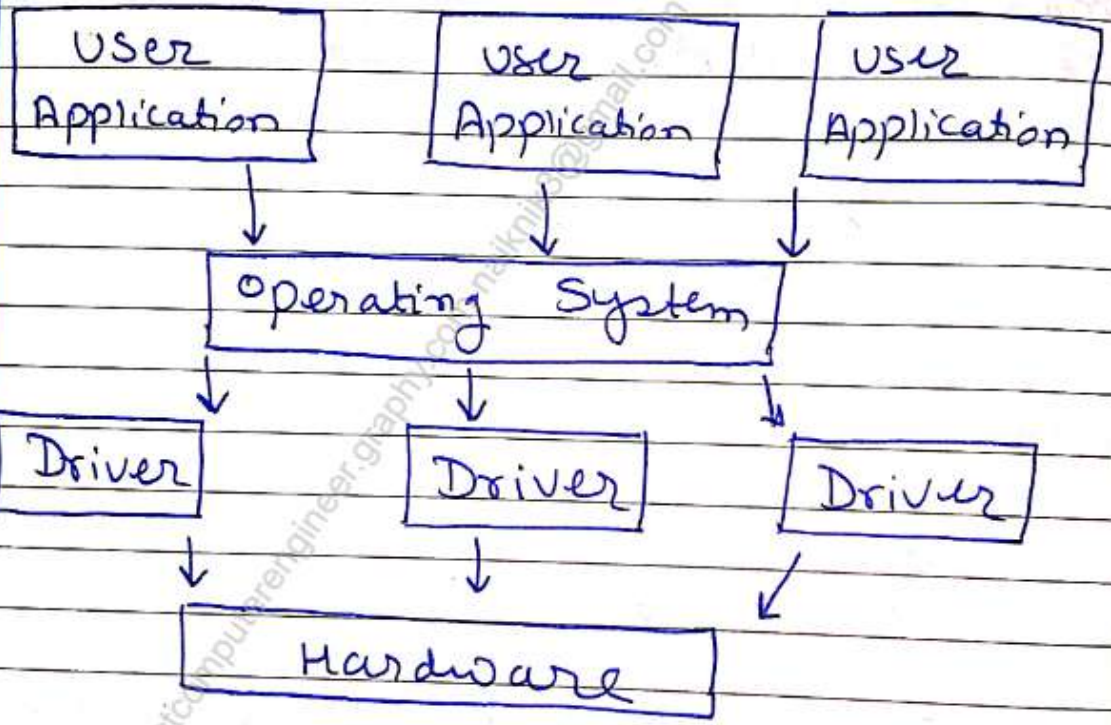
- Operating System plays the role of mediator in between Users and hardware of a System
- It is an interface between User and hardware which assists User to access the various resources of System
- It is also responsible for efficient management and Sharing of various resources attached to the System.

User
↕
Operating System
↕
Computer Hardware

- There are various functions performed by operating System, Some of them are listed below
  - Job scheduling
  - I/O programming
  - Memory Management
  - Security
  - Error handling
  - Device Management
  - Resource Allocation
  - File management

Device Drivers

- Device driver is a System program that helps User applications to communicate with the hardware devices attached to the system.
- These devices include printers, CD-ROM, camera, displays and so.



Text Editor

- Allows User to create / modify a file having only plain text
- ~~Exd~~ Edit contents to get an immediate visual feedback.

  - Emacs (Unix)
  - Notepad (Microsoft)
  - Simple Text, Text Edit. (Macos)

Debuggers

- Debugging in computer programming and engineering , is a multistep process that involves identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a way to work around it.

- A debugger tells the programmer ~~to~~ what types of errors it finds and often marks the exact lines of code where the bugs are found.

- Debuggers also allow programmers to run a program step by step so that they can determine exactly when and why a program crashes.

Loader & Linker

:- A program that accepts the object program decks, prepares these programs for execution by the computer, and initiates the execution.

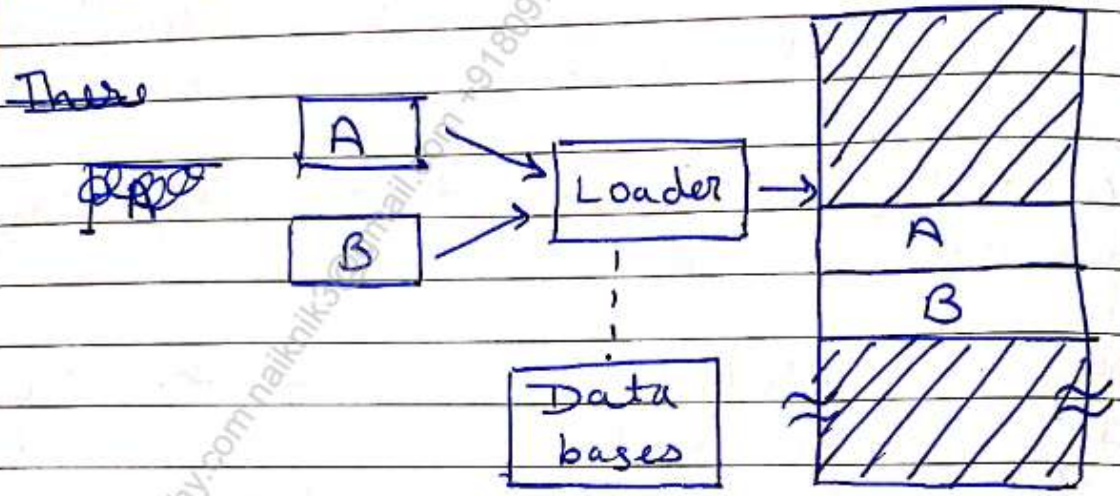


Diagram 1

Program loaded in memory ready for execution

- There are 4 functions of Loaders

1. Allocation & Loading: Allocates memory location and brings the object program into memory for execution - done by loader

2. Linking: Combines two or more separate object programs and supplies the information needed to allow references between them - done by linker.

3. Relocation: modifies the object program so that it can be loaded at an address different from the location originally specified - by Linking loader.

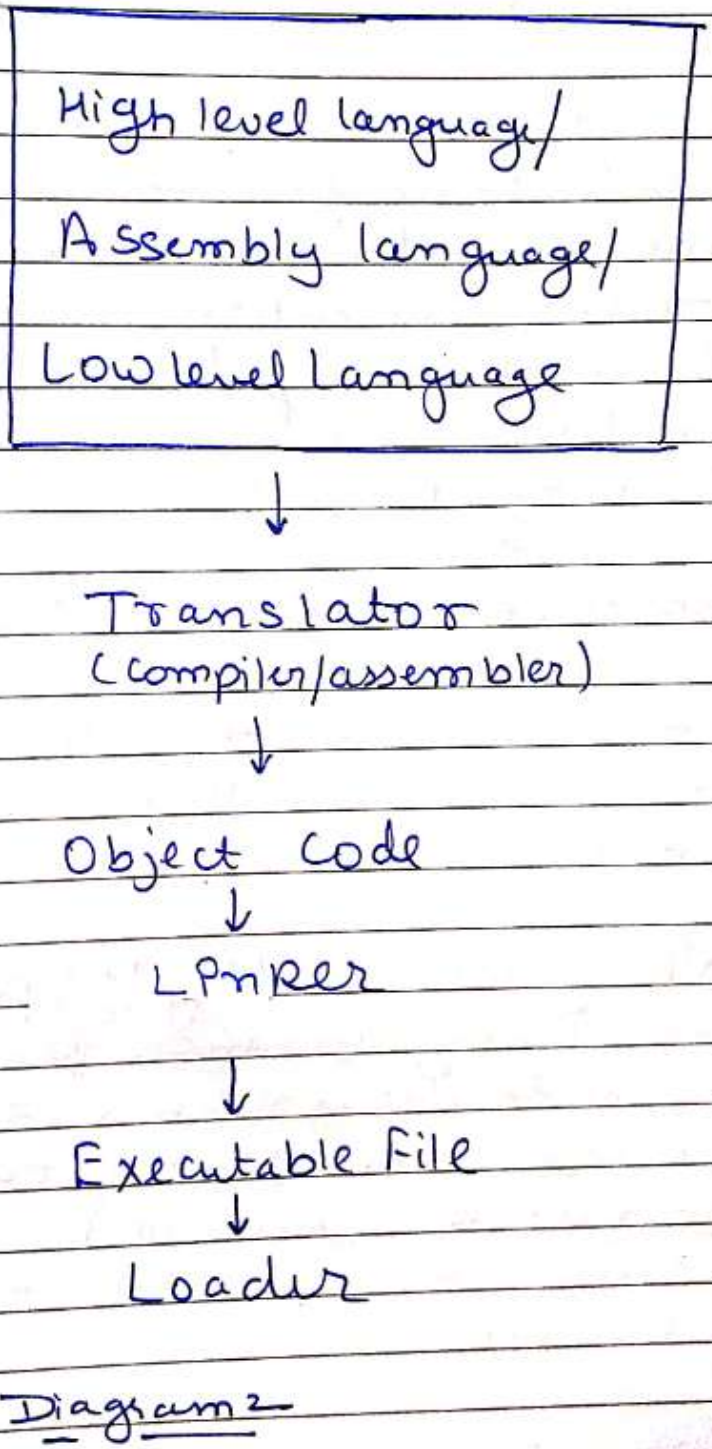


Diagram 2

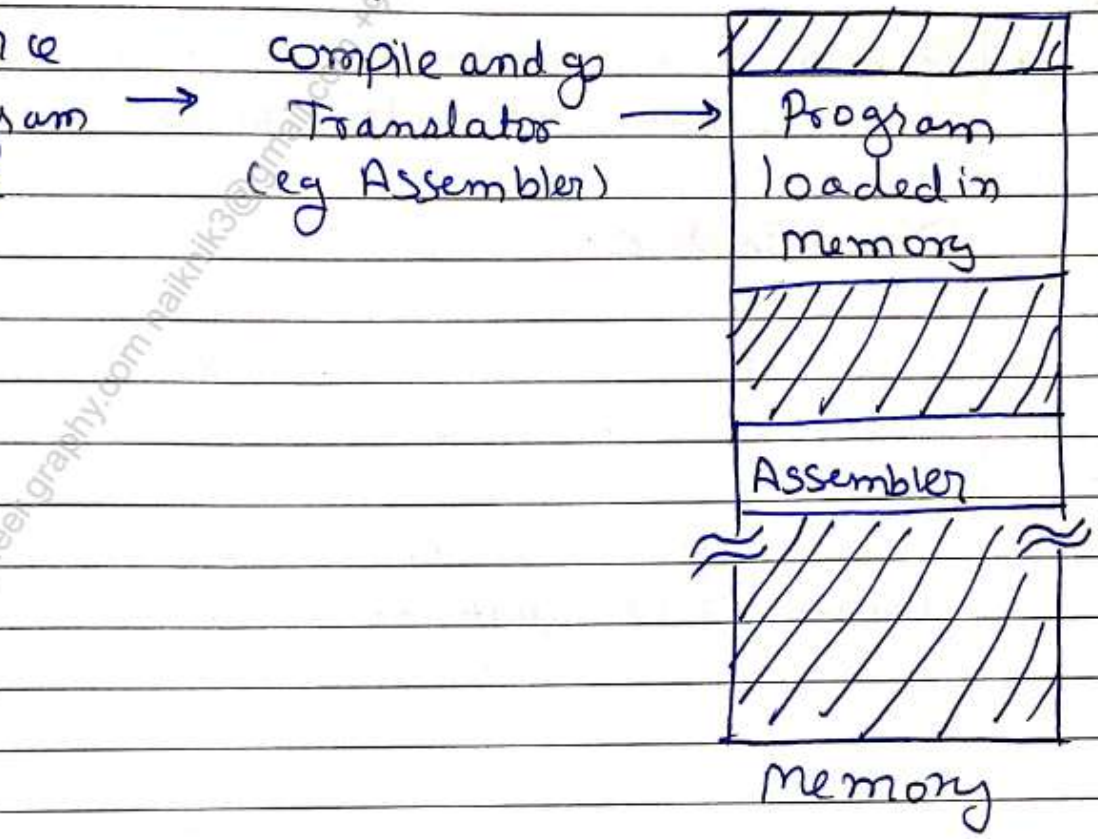
- The program is converted into object code the last thing remaining is to load the program in memory and start the execution. This job will be done by last system programs which are Loader and Linker.

- Loader is a system program which will load the object program prepared by the translator in the memory and start the execution. To do this job the loader has 4 basic functions as discussed earlier (Allocation, Linking, Relocation and Loading)

- As shown in diagram 1. loader also have different data bases it will take different object programs → prepare the necessary databases and will load the programs in the memory.

Along with loading the program in the memory one more job is done by the loader which is transfer the control to the start of the program so that execution will begin.

Compile and GO Loader



- Instructions are read line by line, its mechine code is obtained and it is directly put in the main memory at some known address
- After completion of assembly praes, is assigns the Starting address of the program to the location counter.
- Assembler is first executed, when finished causes a branch straight to the first instruction of the program.
- There is no stop between the compilation, link editing, loading and execution of the program.

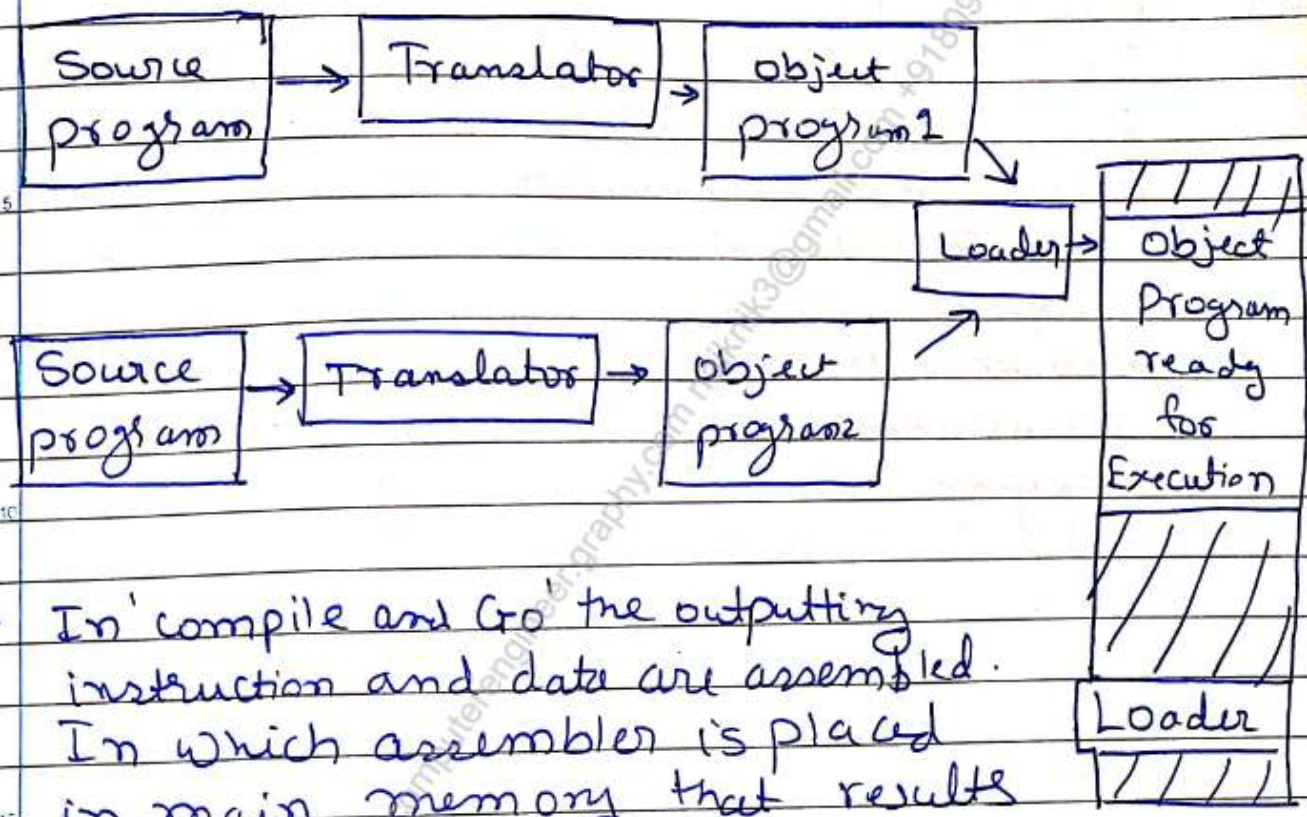
Advantages:

- Simple & easier to implement.
- No additional routines are required to load the compiled code into the memory

Disadvantages:

- Wastage in memory space due to presence of assembler
- No production of .obj file, source code is directly converted to executable form.

General Loader Scheme



⇒ In 'compile and Go' the outputting instruction and data are assembled. In which assembler is placed in main memory that results in wastage of memory

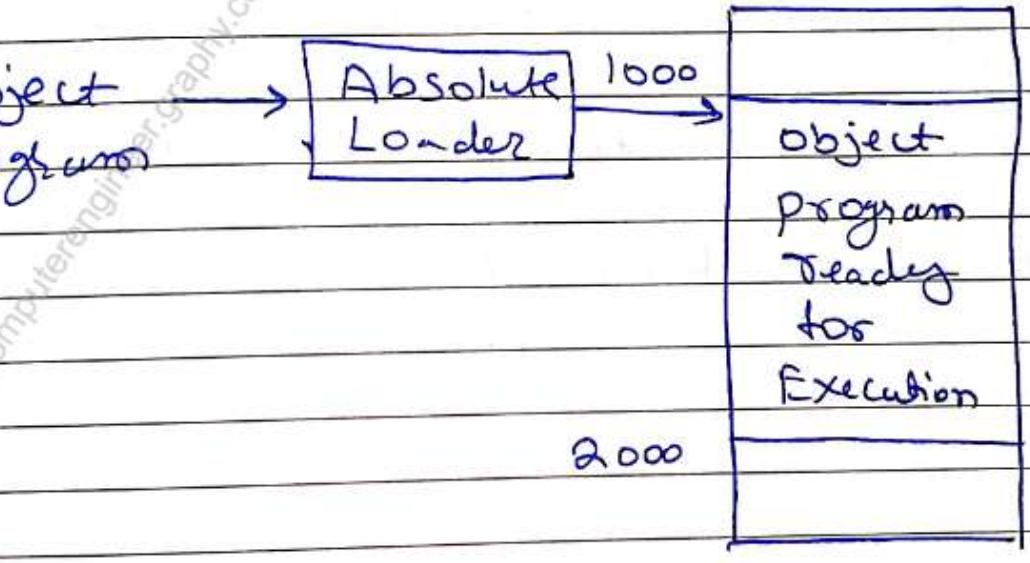
⇒ To overcome that we require the addition of the new program of the system, a loader

- Generally the size of loader is less than that of assembler
- The loader accepts the assembled machine instructions, data and other information present in the object format and places machine instructions and data in core in an executable computer form.
- The reassembly is no longer necessary to run the program at a later date.

Advantages:
In this scheme the source program translators produce compatible Object Program deck formats and it is possible to write subroutines in several different languages. Since the object decks to be processed by the loader will all be in the same 'language' than is in 'machine language'.

Absolute Loader

- The loader accepts these objects decks and places it in memory at the locations supplied by assembler.
- Assembler is not present in memory all the time so more space is available for user. At load time only loader will be available and will complete the loading job.



- Absolute loader is a simple scheme but not that much effective.

o In this scheme the four major functions of loader are accomplished in following manner.

- Allocation is done by programmer
- Linking is done by programmer
- Reallocation is done by assembler
- Loading is done by loader.

Advantages:

- The only advantage this scheme has that it is very simple

Disadvantages

- There is extra burden for the programmer that they have to specify the locations, where the translated code will be placed in memory

- If subroutines are present then programmer has to remember all the absolute addresses while linking these subroutines.

- If any change is done in main program and its length is increased then programmer has to verify it is not overlapping with the locations of subroutines. If overlapping is there then programmer should restuffle the addresses of all subroutines.

- Consider that the main program is stored at location 100-200 and there is one Subroutine ADD which is stored at 250 – 300 location.

- Suppose main program is changed and its length is increased. Due to this, now main program will occupy locations from 100-270. These locations will overlap with the ADD Subroutine.

- In such situation it is necessary to change the address of ADD subroutine and reassemble it.

- Also the programmer has to modify all other subroutines which refer the ADD Subroutine

- The manual allocation of addresses and their reshuffling is quite complex and tedious for programmer.

Linkers

- As we all know that Assembler ~~with com~~ converts assembly code into object code.

- but the object file might have some unresolved symbols and functions such as printf ie here our program has a printf statement. the compiler only verifies wether the printf function follows the correct prototype. It wont bother about actual implementation of printf function exists or not

- So here comes the need of Linker which is the utility program which resolves the unresolved simbols and functions and combines the multiple object file in a single executable file ~~which~~. So this object ~~file~~ code is passed to the linker which will link the actual implementation of printf function code from the run time library and makes it an executable file

- Now suppose compilation process generates multiple object files, may be an unresolved Simbol from an object file might reside in another objectfile. Here linker will solve this problem too. and finally creats an executable file

- What if the linker is unable to find the unresolved Simbol or function code, well in this case is will throw a linker error.

Source code ⟶ compiler → Assembler → Linker

compiler ↓ ↑ Preprocessor

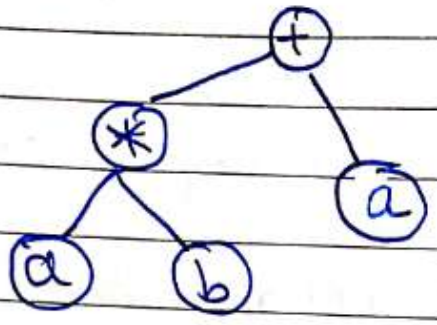
Linker ↓ executable file

Chapter 6: Compilers - Synthesis Phase

DAG

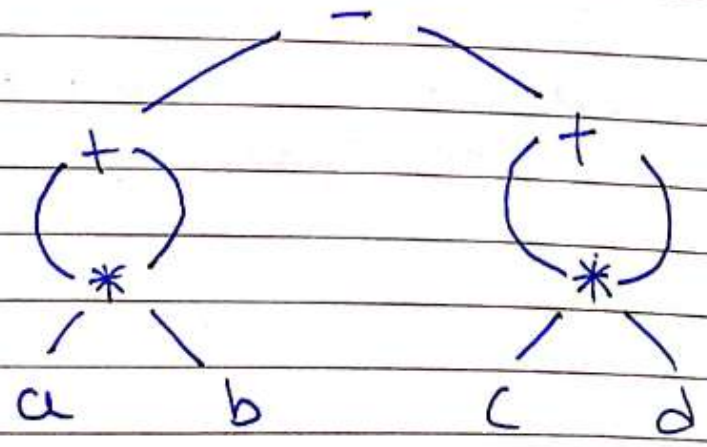
- DAG Stands for Directed Acyclic Graph
- DAG is a graphical representation as that like syntax tree but syntax tree does not find the common sub expressions which DAG does
- In compilers a tree has one route from root to leaf node. DAG is used to share the common subtree of a given expression.
- DAG is used to represent more than one path from Start symbol to terminals.
- They are difficult to construct but are used to save up space.

Q. a*b+a construct Syntax tree and DAG for it

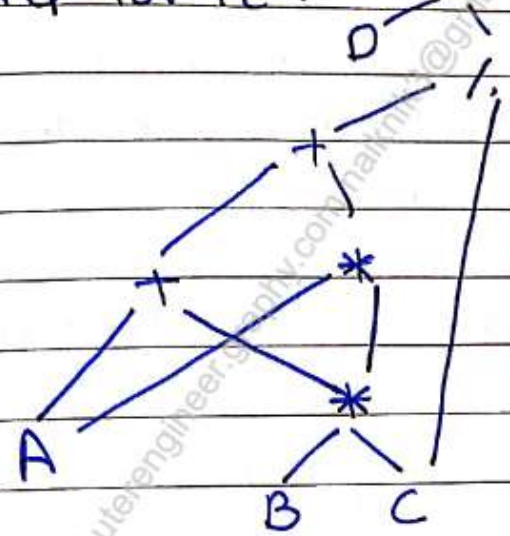


Syntax tree DAG

Q. ((a*b)+(a*b))-(((c*d)+(c*d))
construct a DAG

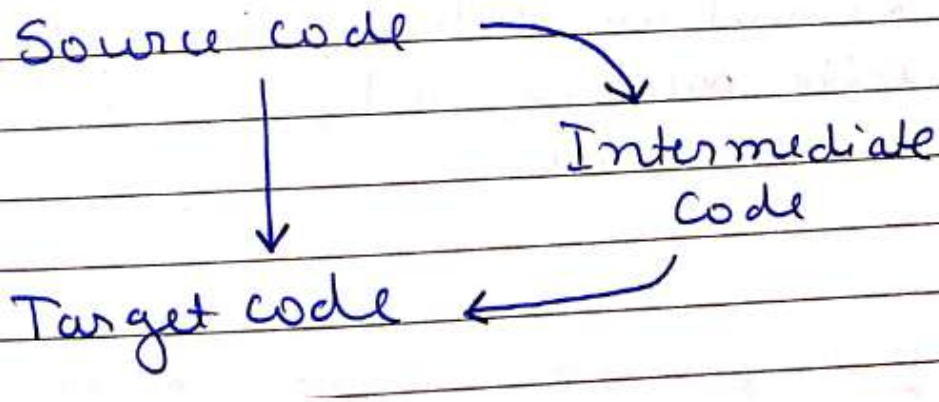


Q. D = ((A+B*C)+(A*B*C))/C construct a DAG for it.



Intermediate Code

* If source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code



- for each new machine, a full native compiler is required if a compiler translates the source language to its target machine language without having the option of generating intermediate code.

- Intermediate code erases the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers

- Synthesis which is the second part of compiler is changed according to the target machine.

- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

- Intermediate code can either be language specific or language independent for example 3 address code.

3 address code

- Intermediate code generator (ICG) receives input from its predecessor phase, Semantic analyzer, in the form of an annotated Syntax tree. That Syntax tree can be converted into a linear representation eg. postfix notation. Intermediate code tends to be machine independent code. Therefore code generator assumes to have unlimited number of memory storage to generate code.

Eg:

$w = x + y * z$:

This expression would be divided into subs expressions and then generate the corresponding code.

$r_1 = y * z$;
$r_2 = x + r_1$;
$w = r_2$

r is used as registers in target program
Three address code has at the most 3 address locations to calculate the expression. A 3 Address code can be represented in two forms: Quardraples and triples.

~~Quad~~

Triplees:

Every instruction in triples presentation has three fields: arg1, arg2 & op. The results of subexpressions are denoted by the position of expression.

Triples represent similarity between Syntax tree and DAG. Triples are equivalent to DAG while representing expressions

OP	arg1	arg2
*	y	3
+	3	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization as the result are positional and changing the order or position of an expression can cause problem

Indirect Triples

It is a modification of triples representation. Here is a use of pointers instead of position to store results. It enables the optimizers to freely reposition the sub expression to produce an optimized code.

Quadraples:
Presentation is divided into four fields: arg1, arg2, result & operator. The above example is represented below in quadraples format.

op	arg1	arg2	result
*	y	z	r1
+	x	r1	r2
+	r2	r1	r3
=	r3		w

⇒ Code Optimization

firstly we need to know what is optimization - It is a process of converting a block of code in more efficient code in terms of space and time complexity. The converted modified code must have:

(A) The ~~out~~ result or output of the program must remain same without having any side effects

(B) The optimized or modified code must execute faster and it should consume less memory

Compilers has a core problem which is optimization or optimizing code. ~~An~~ Alot of research is going on ~~for~~ to solve this problem.

There are two types of optimization:

(A) optimization of speed

(B) optimization of memory

The basic aim of code optimization is

1. Reduce the running time of the compiled code
2. Try to ~~reeeee~~ improve space, power consumption
3. It must preserve the meaning of the code.

- Optimization is very important because of three main reasons

(1) Slow input operation
(2) Differing in the working speed of a human operator and a computer
(3) The volume of code required for newer applications.

- Code optimization is the fifth phase in the compiler.

- Code optimization is divided into two parts Platform dependent techniques and platform independent techniques

- Platform Dependent Techniques
  - Peephole Optimization
  - Instruction level parallelism
  - Data level Parallelism
  - Cache optimization
  - Redundant resources

- Platform Independent Techniques
  - Loop optimization
  - Constant folding
  - Constant propagation
  - Common subexpression Elimination

Chapter 2: Assemblers

What are Assemblers

Assembly languages comprises of low level languages for programming computers, macro controllers, microprocessors and other integrated circuits. They implement Symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. An assembly level language is thus specific to a certain physical or virtual computer architecture.

An Assembler is basically a program which is used for conversion of assembly level language Statements into the target machine code.
An Assembler performs one to one mapping from mnemonic instructions to machine Statements and data.
Mnemonic is a kind of code which is usually from 1 to 5 letters that represents an opcode followed by one or more then one numbers. This is in Contrast with high level languages in which Single statement generally results in many machine instructions.
An assembly language is a machine dependent, low level programming language which is specific to a particular computer System.

Advantages of Assembly language

- helps in getting better performance out of the Processor as possible.

- It helps in getting access to specific Characteristics of the hardware that might not be possible from the higher level language.

Disadvantages of Assembly language.

- It everytime requires the use of an assembler to translate a source program into an object code.

- Assembly languages are specific to a given micro processor or computer and hence they are not portable.

Features of Assembler.

- An Assembler converts mnemonic operation code to their machine language equivalents and resolve Symbolic names for memory locations and other entities. Assigning machine addresses to the Symbolic Saves tedious calculations and manual address updates after Program modifications has been done

There are three basic features Provided by assemblers.

1. Opcode Mnemonics

An opcode is basically symbolic name for a single executable machine instruction, and there is atleast one opcode mnemonic defined for each of the machine language instruction. It eliminates the need to memorize numeric operation codes.

2. Data Sections

They are basically instructions which are used to define data elements to hold data and variables. They define type of data and also length and alignment of data.

3. Assembly directives / Pseudo Operators

They are instructions that are executed by an assembler at the assembly time. Symbolic assemblers allow the programmers to associate arbitrary names with memory locations.