Unit: 1

* Compiler:

Before we run any program it first must be translated into a form in which it can be executed by a computer.
The system software that do this translation are called compilers.
A compiler is a program that can read it into an equivalent program in another language - target language.
The another role of compiler is to report any errors in the source program that it detects during translation process.

Source Program / HLL

⇓

[Compiler]

⇓

Target Program / M/C code or Low level lang.

* Difference between compiler & interpreter.

* Compiler	* Interpreter
Input → [Target Program] → Output	Input (Source Program) → [Interpreter] → Output

Scanned by CamScanner

If the target program is an executable M/C language program, then it can be carried by the user to process, input & produce output.

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execuse the operations specified in the source program with the i/p supplied by the user.

The m/c language target program produced by a compiler is usually faster than an interpreter at mapping inputs to outputs. An interpreter however can usually give better error handling & error diagnostic than a compiler because it executes the source program statement by statement.

temp.c

⇓

I/P ⇒ temp.exe ⇒ O/P

BOOK: Compilers, Principles, Techniques & Tool by Jeffery Ullmen.

Compiler provide easier mapping b/w i/p & o/p. Compiler is more easier & better than interpreter.

HLL ⇒ [Compiler] ⇒ LLL

Source language

M/C language (target lang.)

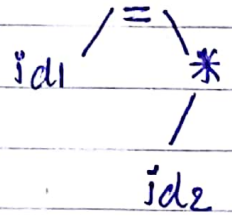
⇓

Processor

Eg: $x = a + b * c$

$id1 = id2 + id3 * id4$

$id1 = id2$

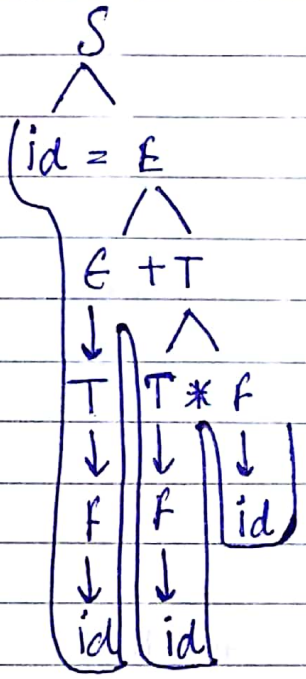


$S \rightarrow id = E$

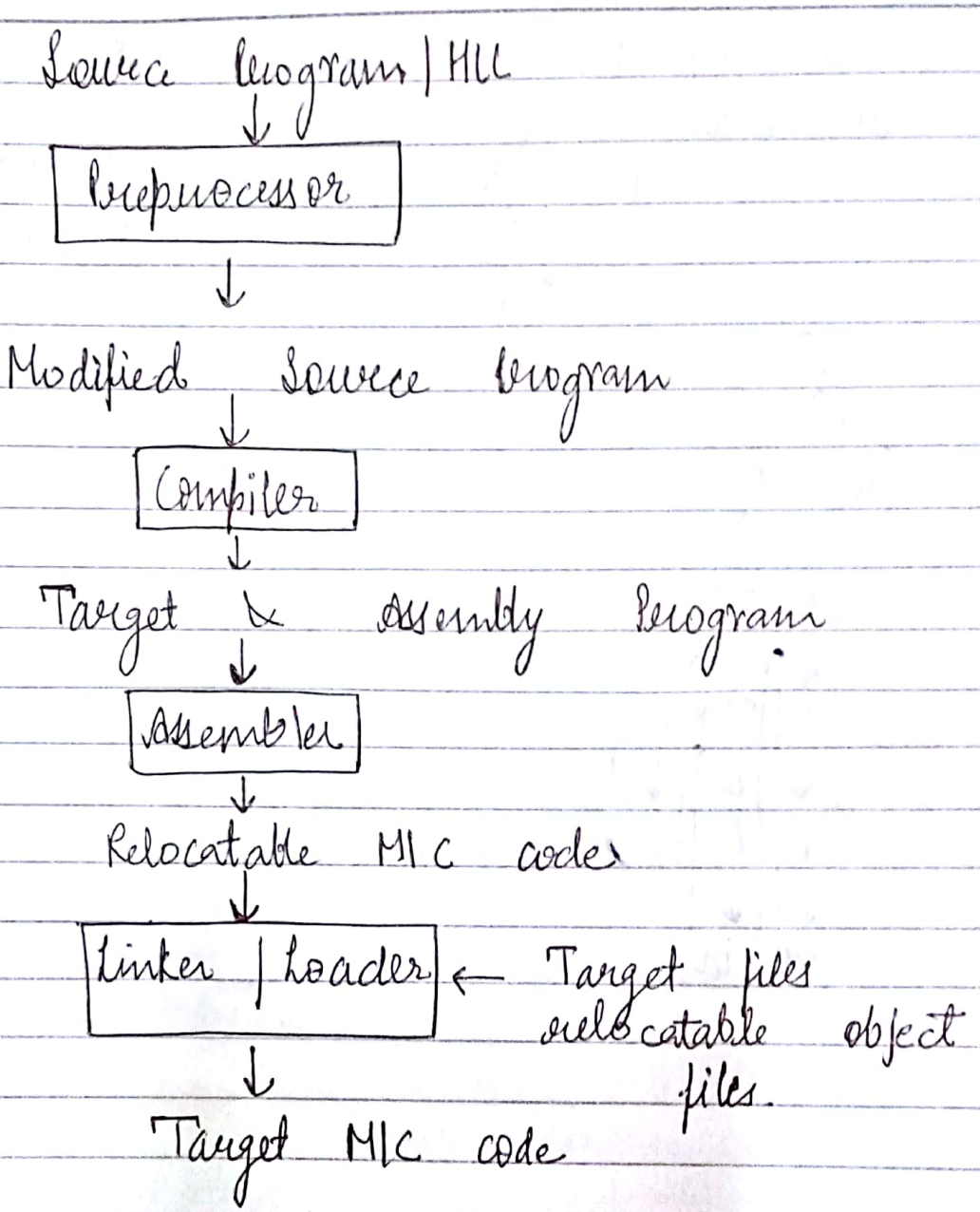
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

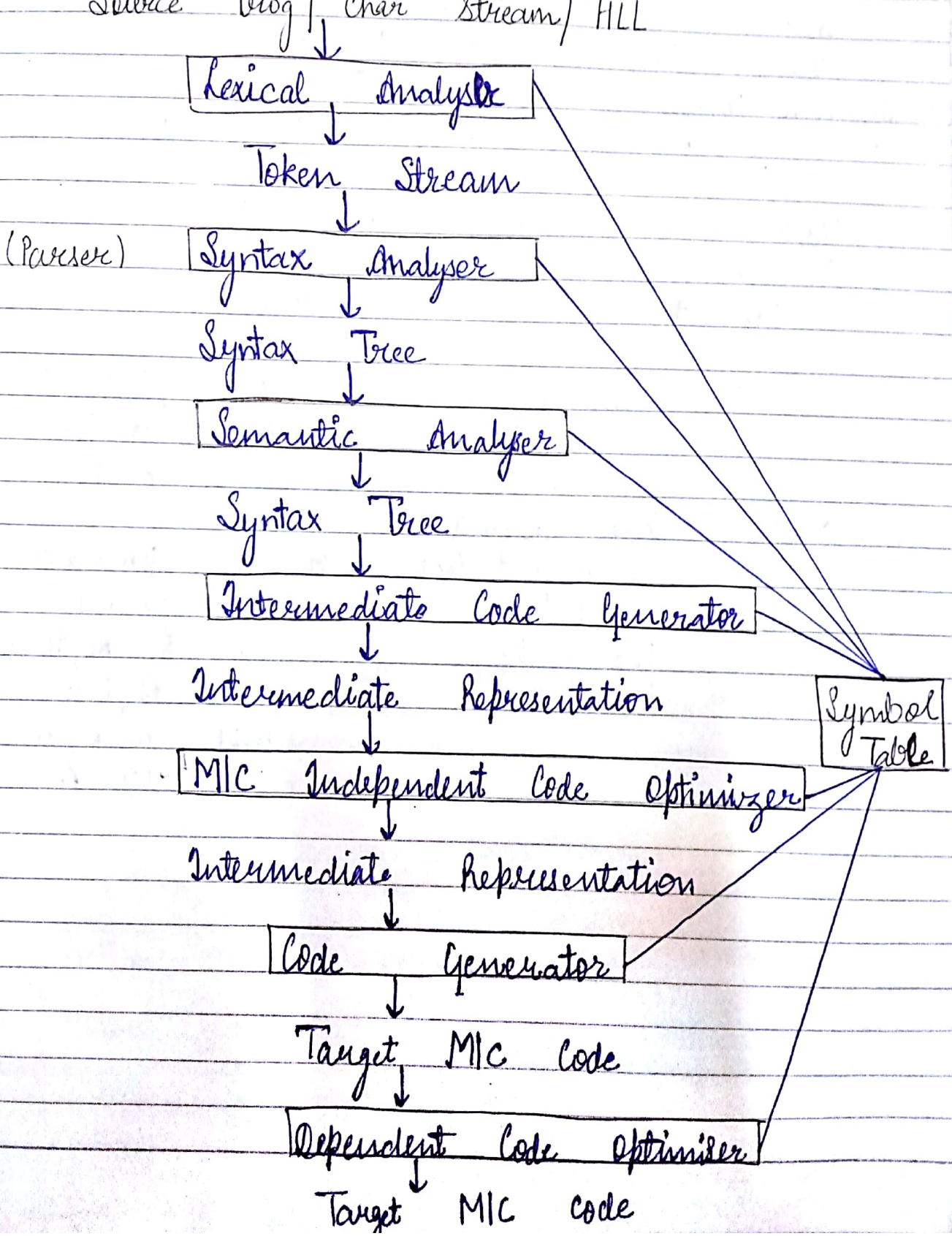


Scanned by CamScanner



Scanned by CamScanner

* Structure of Compiler (functionality)



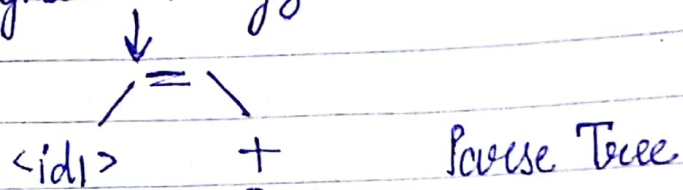
Scanned by CamScanner

Eg: Position = initial + rate * 60;

② Symbol Table
Position → id1
initial → id2
rate → id3

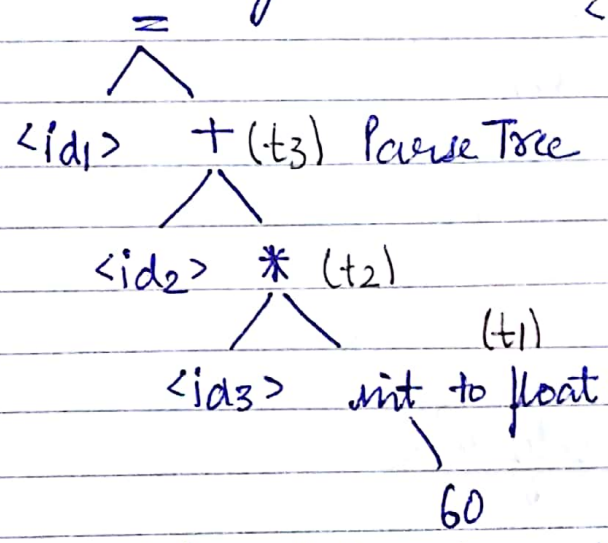
① (Lexical Analyzer)
<id1> = <id2> + <id3> * 60

③ (Syntax Analyzer)



Parse Tree

④ (Symantic Analyser)



Parse Tree

⑤ (Intermediate Code Generator)
t1 = int to float (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

⑥ (Code optimiser)
t1 = id3 * 60.0
id1 = id2 + t1

⑦ (Code Generator)
LDF R2, id3
MLF R2, R2, # 60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Scanned by CamScanner

- We treated compiler as a single box that maps source program into an equivalent targetted program.
- If we open this box little we see that there are 2 parts of this mapping
  1) Analysis
  2) Synthesis

- The Analysis part breaks a source program into pieces & imposes a grammatical structure on them.
- The analysis part also collects the information about the source program and stores it in a data structure called symbol table which is passed along with the intermediate representation to the synthesis part.

- The synthesis part construct the desired target program from intermediate representation and information in symbol table.
- The analysis part also called front end of the compiler and synthesis part is called back-end of the compiler.

- The symbol table which stores information about the entire source program is used in all phases of compiler.

Scanned by CamScanner

* Phases of Compiler :

→ Phase 1: Lexical Analyser

The first phase of compiler is lexical analysis or scanning. The lexical analysis reads the stream of characters from the source program and groups the characters into meaningfull sequence called lexims. for each lexims, the lexical analyser produces as output a token form. Information of each token is stored in symbol table which is needed by syntax analyser, semantic analyser and other code generation phases. lexical analyser also removes the comments and the white spaces from the source code.

→ Phase 2: Syntax Analyser

The second phase of compiler is syntax analysis or parsing. The parser uses the first components of the token produced by lexical analyser to create tree like intermediate representations that depicts the grammatical structure of the token stream which is also called syntax tree or parse tree where each interior node represents an operator and the child node represents the operands or arguements of that operator.

Scanned by CamScanner

Compilers uses context free grammar (CFG) to specify the grammatical structures of programming language & constructing efficient syntax analyser automatically from certain classes of grammar.

→ Phase 3: Semantic Analyser

The semantic analyser uses the syntax tree and the information in the symbol table to check source program semantic consistency with the language definition. The important part of the semantic analysis is type checking where the compiler check that each operator has matching operands. for eg: Type checker in symantic analyser discover the operator * is applied to a floating point number rate and an integer 60. In this case integer may be converted into floating point number.

→ Phase 4: Intermediate Code Generator

In this process of translating a source program into target program, compiler may construct one or more intermediate representation which can have variety of form. This intermediate representation should have two important properties:

1) It should be easy to produce.
2) It should be easy to transmit into target M/C code.

Scanned by CamScanner

The most common representation is used by compiler is three address code where it allows each instruction can have maximum 3 operands.

→ Phase 5: Code Optimisation

The M/C independent code optimisation phase attempt to improve intermediate code so that better target code is generated. Usually better means faster and shorter code which consume less power.

The optimisation phase significantly improve the running time of target program without slowing down the compilation process.

→ Phase 6: Code Generation

The code generator take intermediate representations of the source program and maps it into the target language. (Assembly language)

In this phase, register or memory location are selected for each of the variable use by the program then intermediate instructions are translated in the sequence of machine instructions that perform the same task.

Scanned by CamScanner

→ Symbol Table Management:

An essential function of a compiler is to record the variable name used in source program and collect information about various attribute about each name.

These attribute may be information about storage allocated for a name, its type, its scope.

In case of function and procedure name it stores the number of arguement, their type, the method of passing of each and the type returns.

Scanned by CamScanner

* Bootstrapping : (SIT)

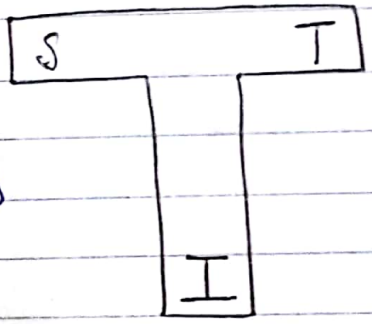
Bootstrapping is the most important concept for building new compilers.

Bootstrapping is the process of writing compiler in target programming language which it is intended to compile.

For constructing any compiler we require three languages.

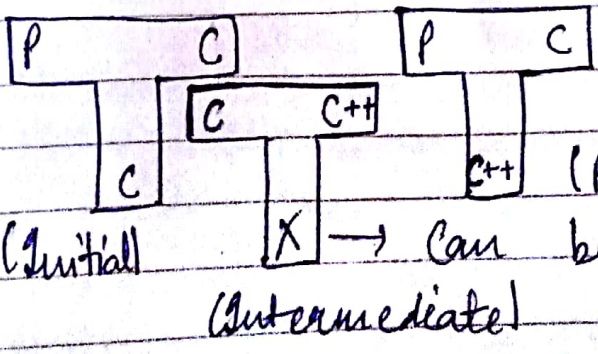
1) Source language (S)
2) Target language (T)
3) Implementation language (I).

With respect to three langugas we use "T" diagram.



Implementation language is the language in which compilers are written.

Q We have pascal translator written in C language that takes pascal code as input & produces C language as output. Create a new pascal translator which is written in C++ & performing the same task.



→ Can be any language

Changing the compiler of C to C++ using boot-strapping.

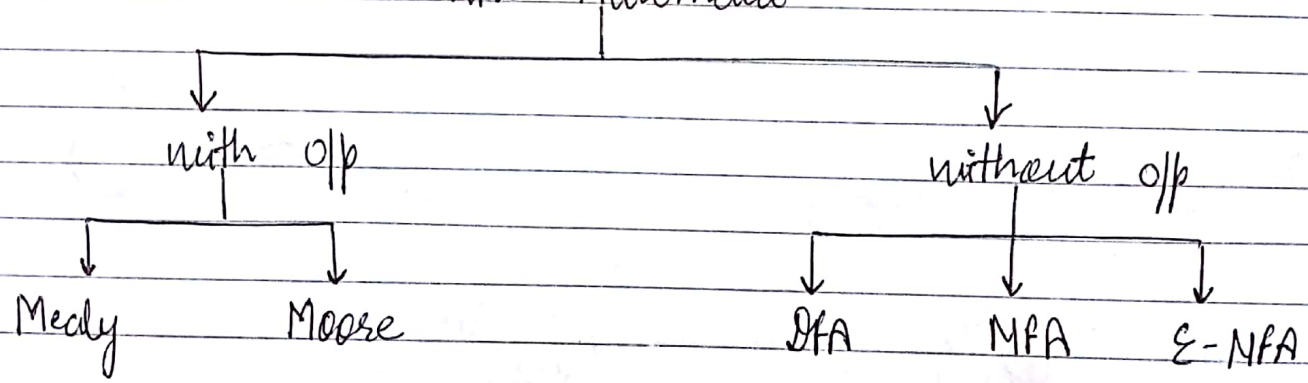
Scanned by CamScanner

with the help of bootstrapping we can create more complex compiler.

* Finite State Automata / Finite State Machine

A finite state M/C or finite state automata (FSA), or finite automata (FA) is a mathematical model of computation which has finite no. of states at any given time.

Symbols → 0-9, A-Z, +, *, _
language → a collection of string
alphabet → {a, b}, {0, 1}, {a, b, c}
String → {a, b, aa, ab, ba, bb}



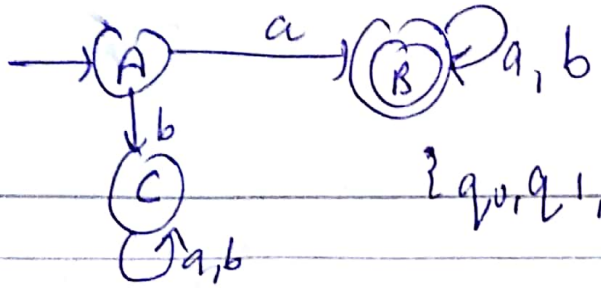
* DFA (Deterministic Finite Automata)

It can be represented by 5 tupples:
(Q, Σ, δ, q0, F)

Q → finite set of states & cannot be empty
Σ → Set of I/P alphabets
δ → Transition function. (Q X Σ → Q)
q0 → start state
F → Set of all final states

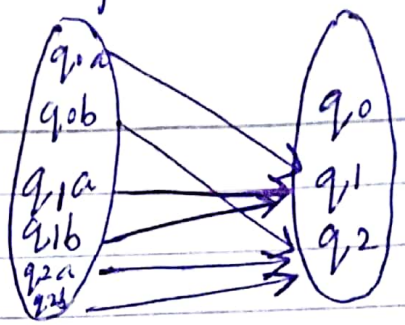
Q. Construct a DFA over {a, b} where each string starts with 'a'.

Scanned by CamScanner



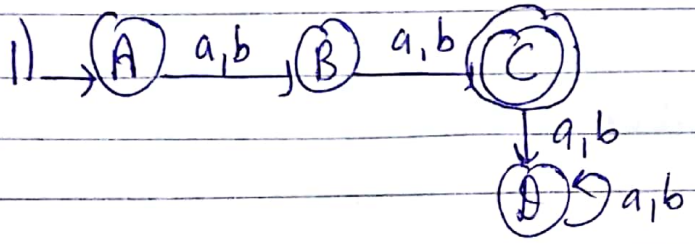
$\{q_0, q_1, q_2\}$ $\{a, b\}$

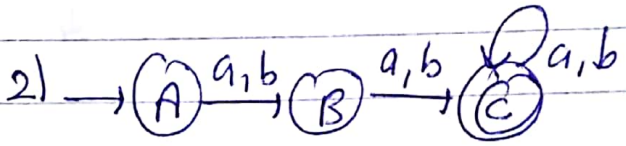
$\phi X \Sigma \rightarrow \phi$

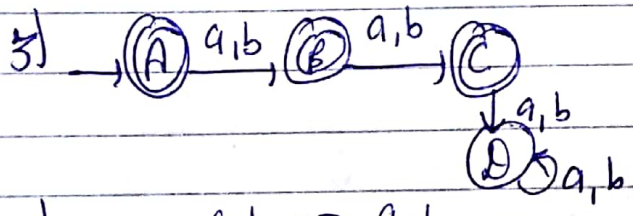


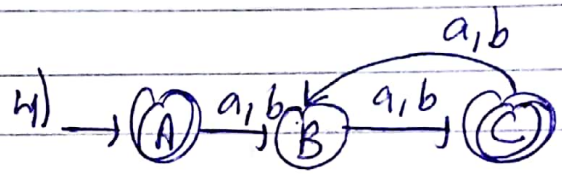
Q. Construct the following DFAs.

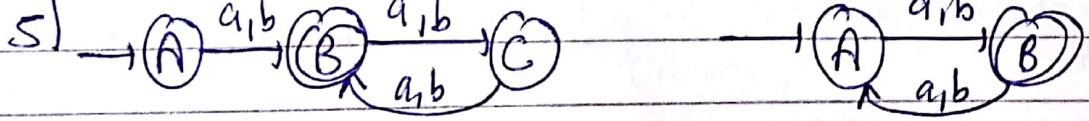
1) accepts set of all string $\{a, b\}$ of length 2.
2) accepts $\{a, b\}$ of length at least 2.
3) accepts string less than or equal to 2.
4) accepts string having $|w| \bmod 2 = 0$. (even)
5) accepts string having $|w| \bmod 2 = 1$.

1)


2)


3)


4)


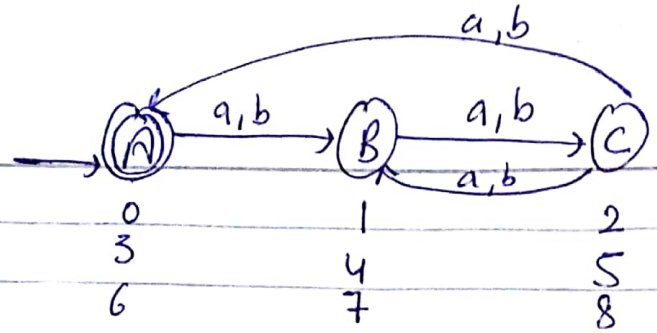
5)


Q. Construct DFAs over $\{a, b\}$

1) that accept $|w| \bmod 3 = 0$
2) $|w| \bmod 3 = 1$
3) $|w| \bmod 3 = 2$
4) $na(w) = 2$
5) $na(w) \geq 2$
6) $na \leq 2$

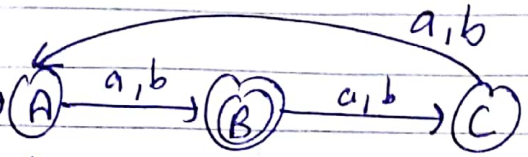
Scanned by CamScanner

1)



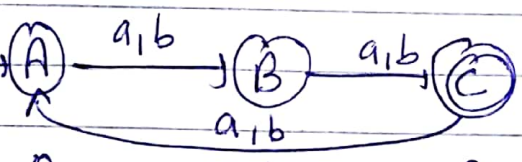
0 1 2
3 4 5
6 7 8

2)



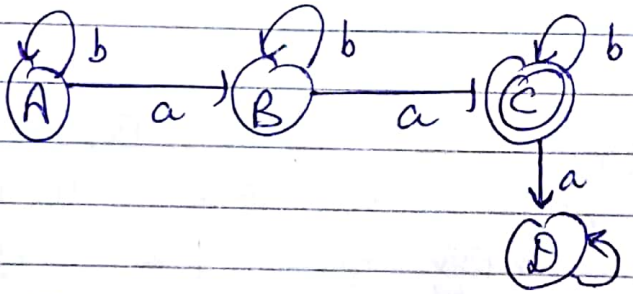
0 1 2
3 4 5
6 7 8

3)

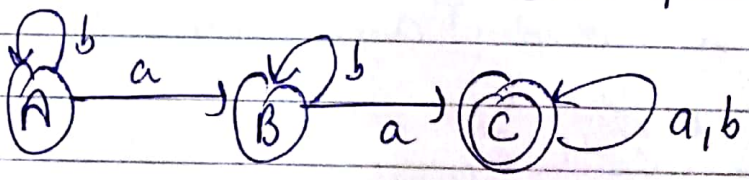


0 1 2
3 4 5

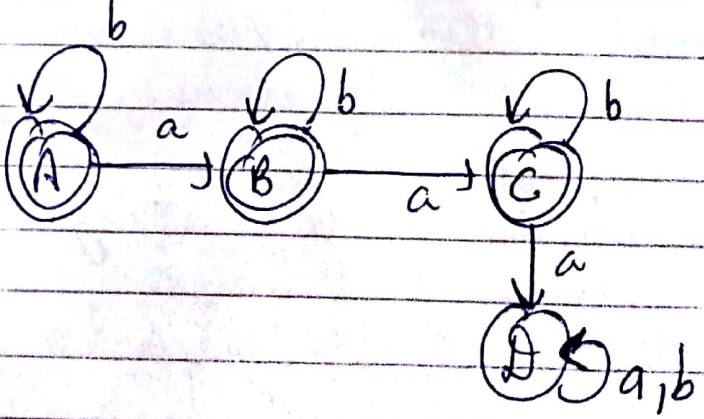
4)



5)

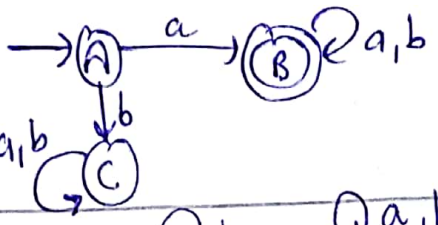


6)

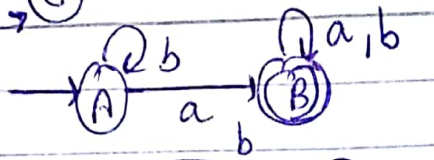


Scanned by CamScanner

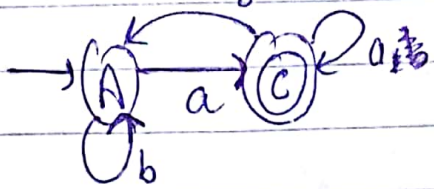
① Start with 'a'



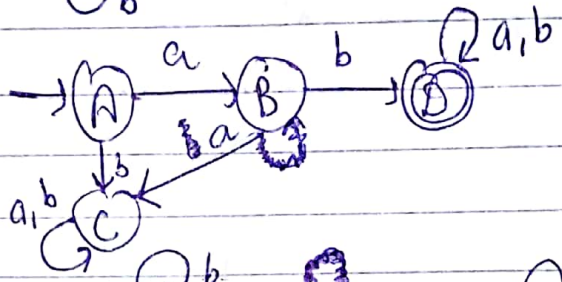
② containing 'a'



③ ends with 'a'

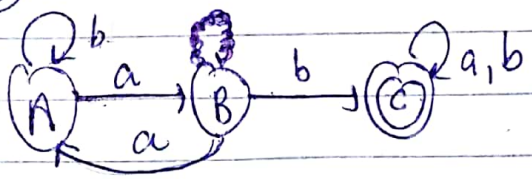


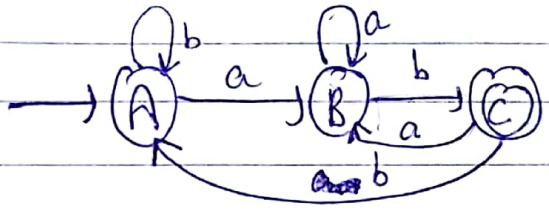
④ starts with 'ab'



⑤ contains 'ab'

⑥ ends with 'ab'





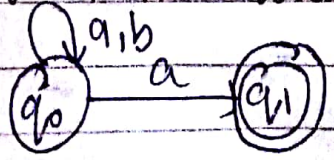
* NFA (Non-Deterministic Finite Automata)

If we are starting from a state and go to a particular state then there is no guarantee that we land a only on one state. $(Q, \Sigma, \delta, q_0, F)$

$\delta : Q \times \Sigma \rightarrow 2^Q$

Every DFA is an NFA but not vice-versa
NFA is more powerful than DFA.
NFAs are more easier to implement.

Q. Construct an NFA where every string ends with 'a'.

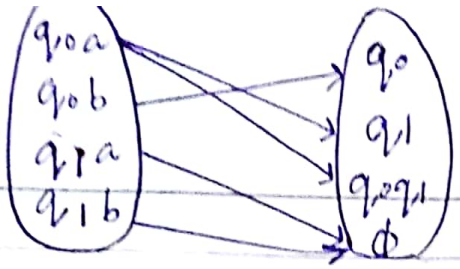


$Q \times \Sigma$

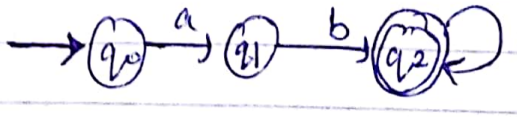
$\{q_0, q_1\} \times \{a, b\}$

$\Rightarrow \{ q_0, q_1, q_0q_1 \}$

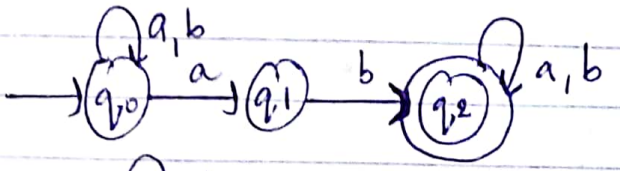
Scanned by CamScanner



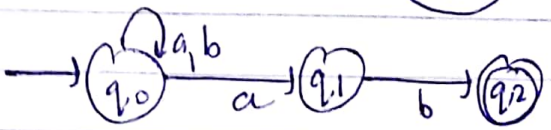
Q. ① Starts with 'ab'



② Contains 'ab'



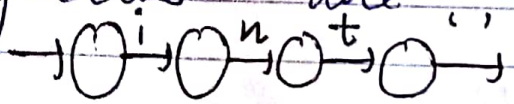
③ Ends with 'ab'



* Use of DFAs in Lexical Analyser

The lexical analyser converts the high level language in streams of tokens and while scanning the characters if it finds any meaningful sequence of characters then it will convert it to a token.

Meaningful sequence of characters are called lexims. And we match these leximes with the existing patterns available in the compiler then the lexical analyser generates the tokens.



ipt a ; int | a | ;

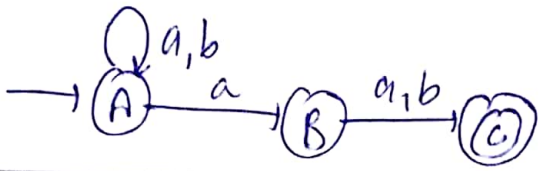
If any pattern is not matched acc. to the compiler then it will give an error.

At the time of matching we always prefer longest match.

Q. Construct an NFA over {a, b} where all string from second symbol to RHS is a. Convert it to DFA.

Scanned by CamScanner

NFA

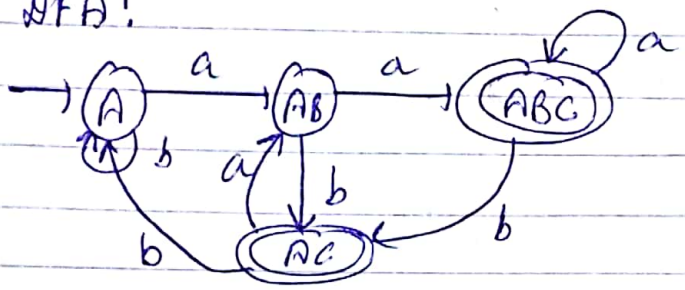


Transition table

	a	b
A	[AB]	[A]
B	[C]	[C]
C	ϕ	ϕ

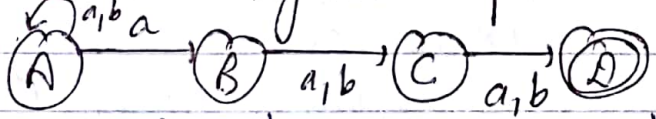
	a	b
[AB]	[ABC]	[AC]
*[AC]	[AB]	[A]
*[ABC]	[ABC]	[AC]
ϕ	ϕ	ϕ

DFA:



Q. Convert NFA to DFA for "All strings in which third symbol from RHS is 'a'".

NFA

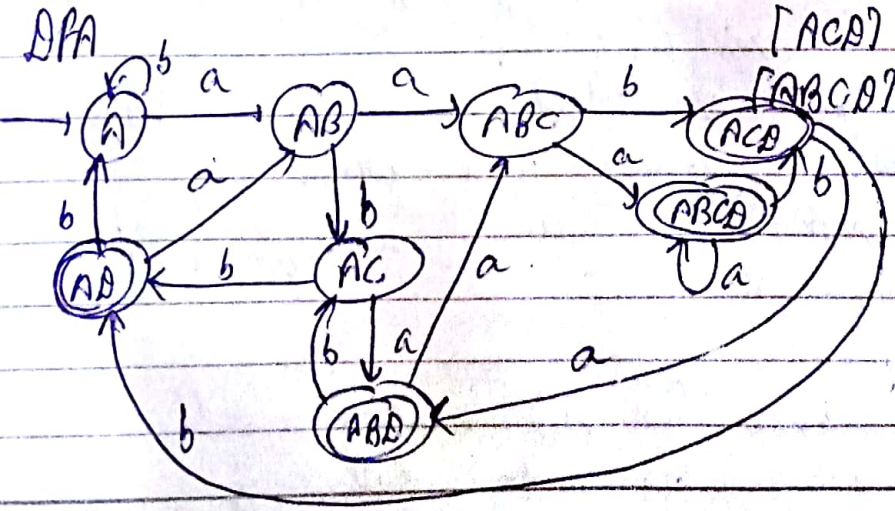


Transition table

	a	b
→A	[AB]	[A]
B	[C]	[C]
C	[D]	[D]
(D)	ϕ	ϕ

	a	b
→[A]	[AB]	[A]
[AB]	[ABC]	[AC]
[AC]	[ABD]	[AD]
[ABC]	[ABCD]	[ACD]
[AD]	[AB]	[A]
[ABD]	[ABC]	[AC]
[ACD]	[ABD]	[AD]
[ABCD]	[ABCD]	[ACD]

DFA



Scanned by CamScanner

* Regular Expressions:

Just like as finite automata are used to recognise the pattern of strings, RE are used to generate patterns of string.
A RE is an algebric formula to represent patterns.

→ Operations On RE
(.) Concatenation
(+) Union
(Σ^*) closure

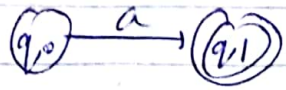
Q. Write RE for:

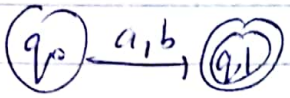
1) language of exactly two length — (a+b)(a+b)
2) atleast two length — (a+b)(a+b)(a+b)*
3) atmost two length — (a+b+ε)(a+b+ε)
4) Even length string — ((a+b)(a+b))*
5) odd — ((a+b)(a+b))*(a+b)
6) strings divisible by 3 — ((a+b)(a+b)(a+b))*
7) containing a — (a+b)* a (a+b)*
8) starts with a — a(a+b)*
9) ends with a — (a+b)* a
10) ends with different symbol and start with different — (a(a+b)*b + b(a+b)*a)
11) Ends and start with same symbol. — ((a(a+b)*a) + (b(a+b)*b))

Scanned by CamScanner

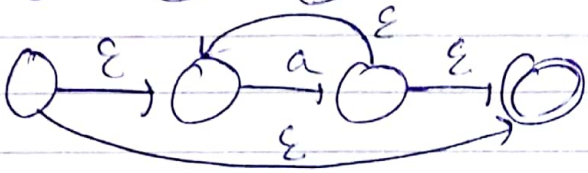
Q. Conversion of RE to DFA

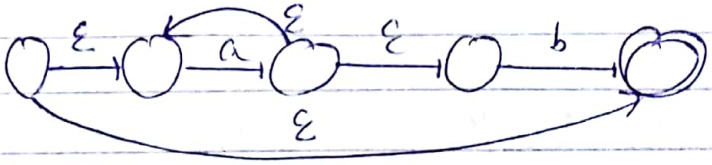
ϕ 

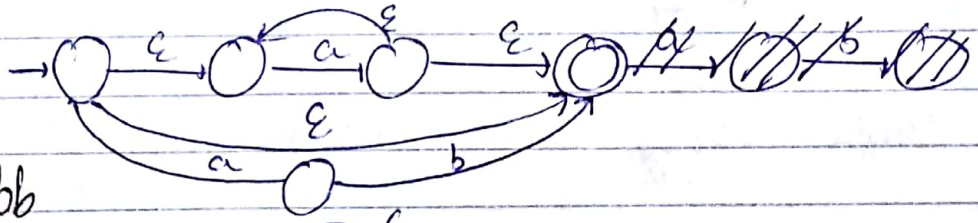
a 

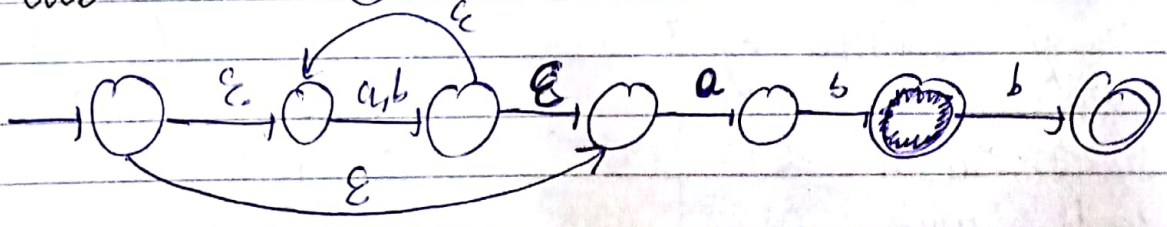
a+b 

a.b 

a* 

a* b 

a*+ ab 

(a+b)* abb 

* Optimisation of DFA based Pattern Matchers:

Multiple forms of DFA's can be possible but minimal DFA's are unique.

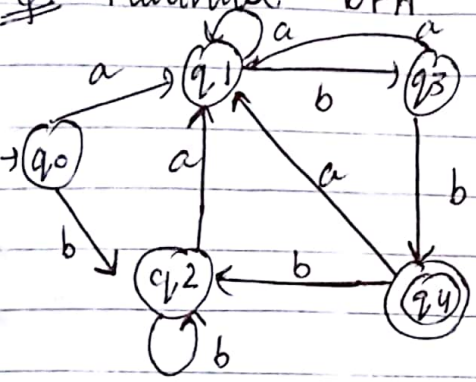
To construct a minimal DFA we can use two methods:

1) Subset Construction Method
2) Myhill Nerode Theorem.

Scanned by CamScanner

Subset Construction Method:

Q. Minimise DFA



$\delta(p,w) \in F$	$\delta(p,w) \notin F$
$\delta(q,w) \in F$	$\delta(q,w) \notin F$
$(p,q) \in F$ (Equivalent)	(Equivalent)
(Equivalent)	$(p,q) \notin F$

$\pi_0 \Rightarrow$ 0 Equivalence

$\pi_1 \Rightarrow$ 1 Equivalence

$\pi_2 \Rightarrow$ 2 Equivalence

$\pi_n \Rightarrow$ n Equivalence

Before solving this question we need to find different equivalent states.

Step:1. Identify start state & final state.

Step:2. Find states which are not reachable from initial state, remove them.

Step:3. Draw transition table.

q	a	b
→ q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
(*) q_4	q_1	q_2

$\pi_0 \Rightarrow [q_0\ q_1\ q_2\ q_3]\ [q_4]$

$\pi_1 \Rightarrow$

q_0q_1: a → q_1q_1, b → q_2q_3

q_0q_3: a → q_1q_1, b → q_2q_4 (Non-Eq.)

q_0q_2: a → q_1q_1, b → q_2q_2

q_0q_4: a → q_1q_1, b → q_2q_2

Step:4 Find 0 equivalence or 0 equivalent set.

- Separate final states from non-final states. (π_0)

Step:5. Find 1 equivalence (π_1).

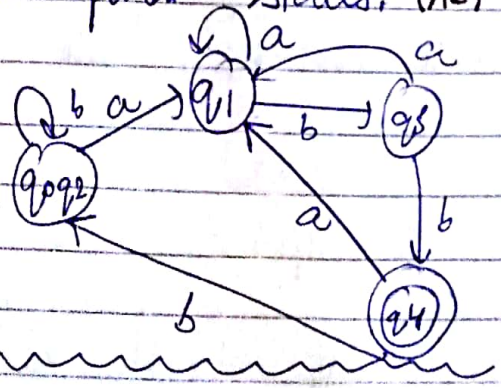
$\pi_1 \Rightarrow [q_0\ q_1\ q_2]\ [q_3]\ [q_4]$

Step:6. Find 2 equivalence (π_2)

$\pi_2 \Rightarrow [q_0\ q_2]\ [q_1]\ [q_3]\ [q_4]$

Step:7. Find 3 equivalence (π_3)

$\pi_3 \Rightarrow [q_0\ q_2]\ [q_1]\ [q_3]\ [q_4]$



π_2 & π_3 are same, hence we will stop here.

Scanned by CamScanner

→ Tokens:

Tokens are group of similar type of words written in the source language program, mainly the tokens are of 5 types:

1) Keywords: These are the fix meaning words or reserved words.
2) Constants | literals: int, float, char, string, a=10
3) Operators: < , > , + , - , / , * , <= , >= , etc.
4) Punctuation Symbols: ; , {} , () , [] , etc
5) Identifiers: Names provided by programmers.

→ Patterns:

Patterns are the rules associated with every token which must be followed by a string under that token.

→ Lexemes:

lexemes are actual strings or words written in the source program. A particular lexeme will be matched by following prescribed pattern for the token under which it comes.

```
void main (int a, int b)
  {
  if (a > b)
  printf ("%d is greater than %d", a, b);
  else
  printf ("%d is greater than %d", b, a);
  }
```

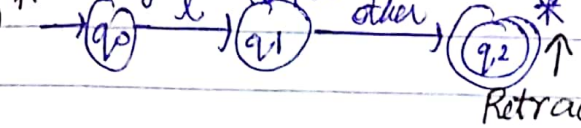
void | main | (| int | id1 | , | id2 |) | Keyword

Scanned by CamScanner

* Designing a Lexical Analyser

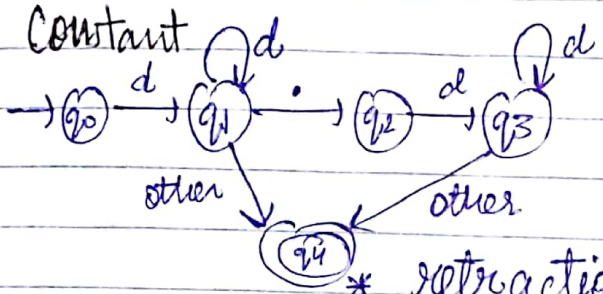
Step:1 List out all the alphabets, tokens & their patterns.
Step:2. Construct transition diagram for all the tokens.

Identifier → l (l + d + _)*

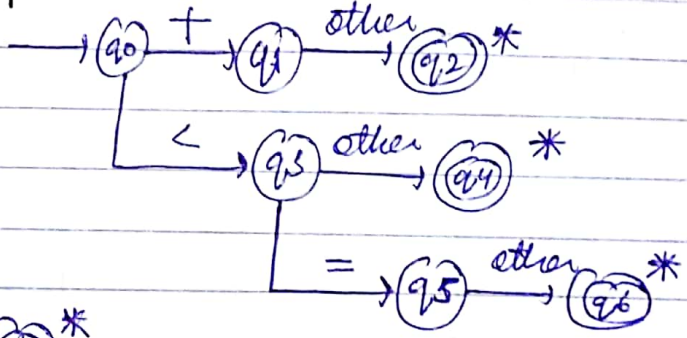


l = letter
d = digit
_ = underscore

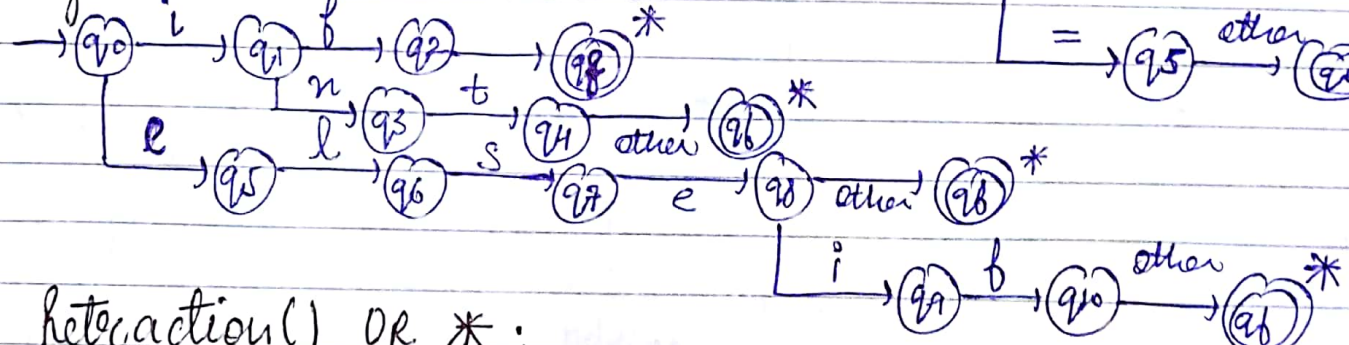
Constant



Operator



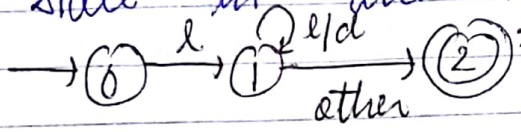
Keywords



Retraction() OR *:
Every final state in the transition diagram labelled with (*) which is called retraction. It is needed once lexical analyser has recognised one string under a top run (scanning from start to end) by reading a delimiter (other)
In case of identifier "other" can be punctuation symbol or an operator but this delimeter is not the part of actual token/identifier. So we will not store these symbols with our token.
For lexical analyser to announce a new token has found, it must go to one character before the delimiter & this is called retraction (process).

Scanned by CamScanner

Step: 3. Implementation of transition diagrams.
For implementing, we will write code for every state in the transition diagram.




```
Begin
  c = getchar()
// code for state 0
  if (isletter(c))
      goto state 1;
  else
      fail();
// code for state 1
  if (isletter(c) or isdigit(c))
      goto state 1;
  else if (isdelimeter(c))
      goto state 2;
  else
      fail();
// code for state 2
  retraction();
     get_id();
     install_id();
End
```

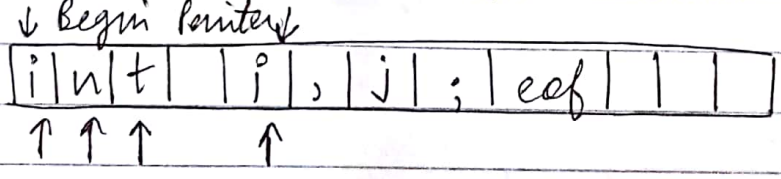
* get_id → This function will be invoked once control has reached to the final state of transition diagram for an identifier. For an identifier, get_id() will look into the symbol table whether newly found token is already there in symbol table or not. If it is already exist in symbol table then get_id() returns true otherwise it return false.

* install_id() → If get_id() returns false then install_id() will be invoked. It will store the new identifier in symbol table & returns its address through pointers.

* Input Buffering:
Program may be stored in secondary memory for the compilation process. Fragment of the program will be loaded into main memory buffer for the compilation process.

Scanned by CamScanner

These buffers helps the compiler to scan I/P stream character by character.
In I/P Buffering there are two types of pointers which helps at the time of scanning:
1) Begni Pointer
2) forward Pointer
Begni Pointer will be fixed at first character of the I/P stream (lexemes) & forward pointer will move left to right, character by character in order to recognise a token.
Once a new token has found, begni pointers and forward pointers will be shifted to the first character of the next lexemes.
forward Pointer is also called look ahead pointer

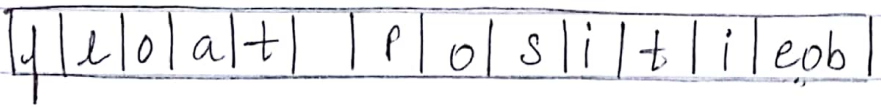


Input Buffering follows two types of Buffer Schemes:
1) One Buffer Scheme: In this scheme only one buffer will be used for input buffering.
→ Drawback:
In case of larger lexemes, complete statement can not be loaded into fixed one buffer scheme In this case half of the statement will be loaded into the buffer after recognizing it half of the statement will be loaded.

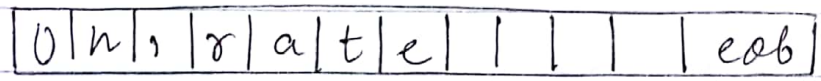
Scanned by CamScanner

This process is bit time consuming because refilling of buffer for a single statement will take time.

2) Two Buffer Scheme: For avoiding this refilling time we can use two Buffer scheme which can store the larger lexemes or a complete statement simultaneously. float position rate;



Buffer 1



Buffer 2.

→ Algorithm for Two Buffer Scheme:

```
if (forward_pointer == eob (buffer1))
  {
   /* Refill buffer 2 */
    forward_pointer++;
  }
else if (forward_pointer == eob (buffer2))
  {
   /* Refill buffer 1 */
    forward_pointer++;
  }
else if (forward_pointer == eof (input))
     return ; /* Terminate */
else
  forward_pointer++ ;
     /* Read rest of the input */
```

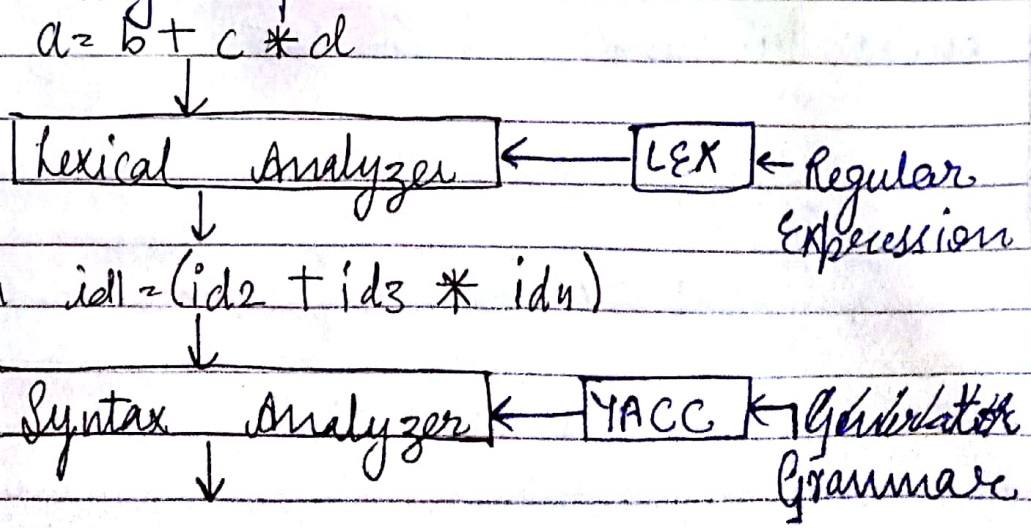
Scanned by CamScanner

WAP in C to calculate No. of characters/words, lines, blank spaces

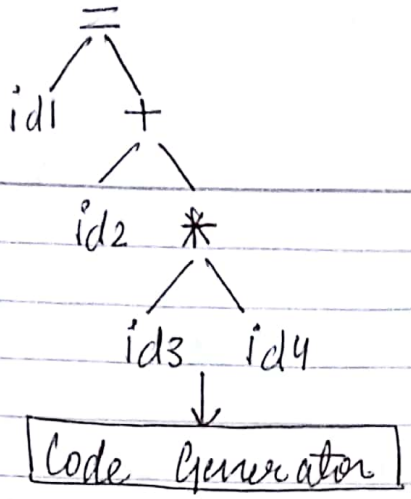
WAP in C to identify a keyword from the entered string. (Program for Lab)

* Lexical Analyzer Generator (lex compiler)

- Various tools are available for automatically generating most of the phases of a compiler.
- Tools for generating lexical analyser is called lexical analyzer generator.
- lex Compiler is such a tool by using we can automatically generate a C program for lexical analyzer phase.
- If we provide lex specification program (Declaration, Rules, procedures) as an input to the lex compiler, it will generate a C program for lexical analysis phase.
- lex Compiler is available as a utility program in UNIX OS
- Lesk & Johnson published a research paper on LEX and YACC in 1975, which greatly simplifies Compiler writing process.

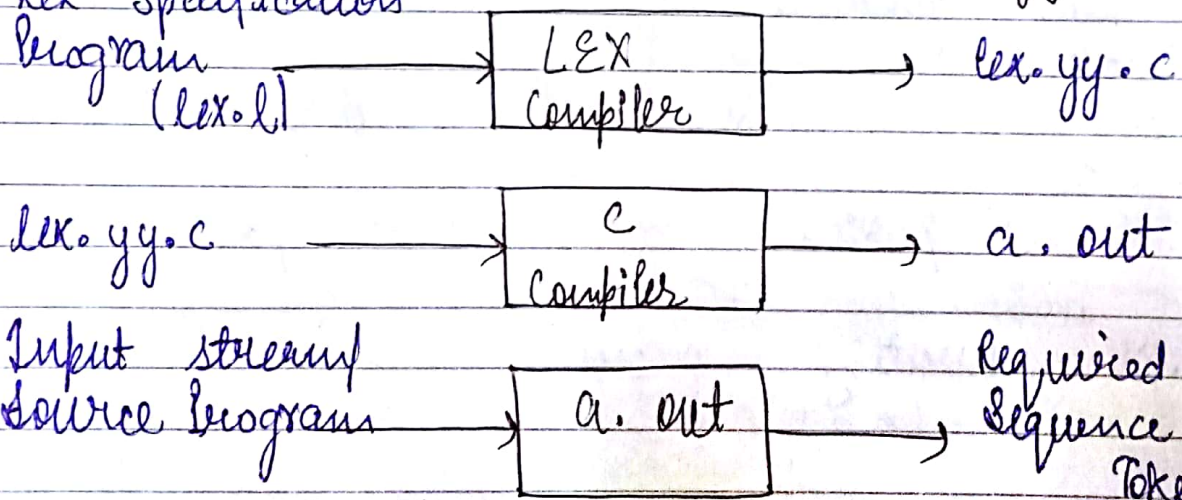


Scanned by CamScanner



LEX:

It is a scanner generator. Here, input is a set of regular expressions & associated actions & rules. (written in C). LEX recognizes regular expressions whereas YACC recognizes grammar. LEX divides input stream into the tokens while YACC uses these tokens & group them logically to create Parse Tree. Output of the LEX phase is a table driven scanner. (lex.yy.c)



→ LEX Specification Program

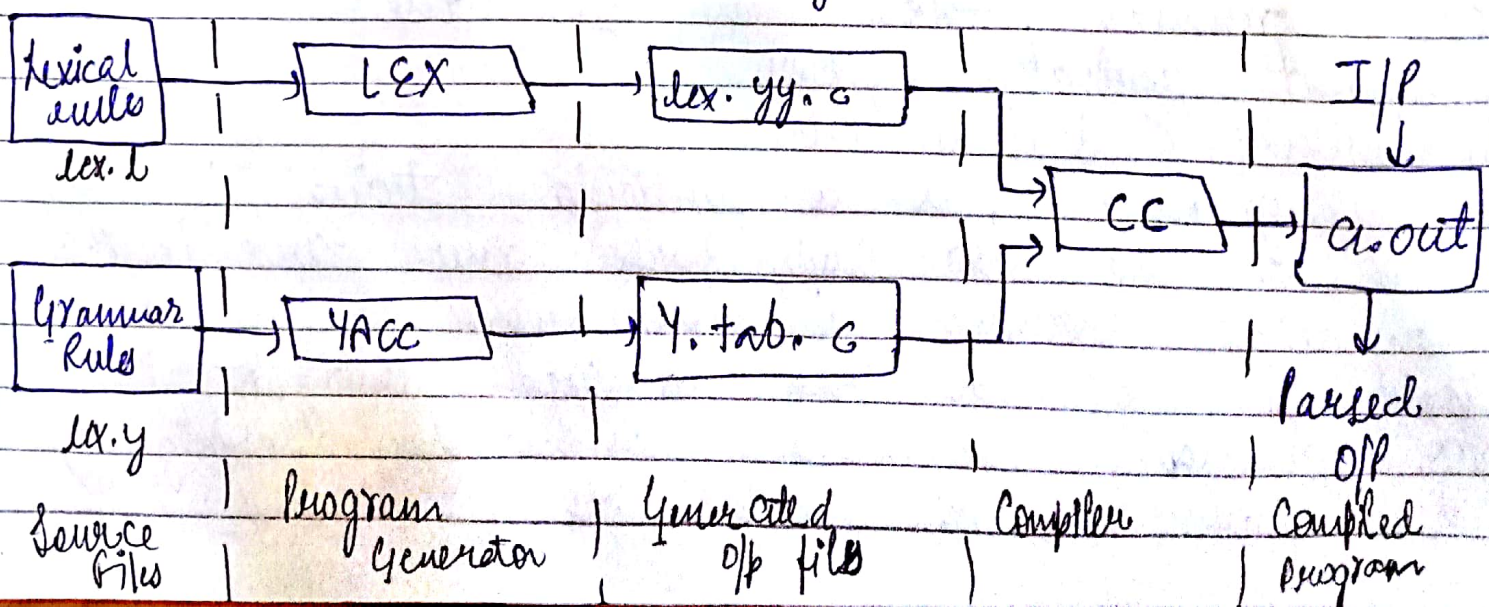
%%
Declaration / Definition
%%
Translation Rules
%%

Scanned by CamScanner

%% Auxilary Procedures/ Sub Routine

1) Declaration: In declaration part, we have to provide set of operators, alphabets, keywords & punctuation symbol in a prescribed format.
2) Translation Rules: In this section we have to provide patterns & rules associated with every token.
3) Auxilary Procedures: In this part, all procedures & all functions are provided of lexical analysis phase. Eg: getid(), install id()

* YACC (Yet Another Compiler Compiler)
YACC will read grammar and generate 'code' for a syntax analyser or parser. It is used to generate 100's of compilers. YACC is utility of Unix System.
The YACC can report conflicts or ambiguities in the form of error message to the user screen.



Scanned by CamScanner

* Commands Required for LEX & YACC (on UNIX Terminal)

% lex bas.l ↵ → bas.yy.c

% yacc -d bas.y ↵ → y.tab.c , y.tab.h

(y.tab.c → Grammar specification; y.tab.h → Tokens into help in making symbol table.)

% cc bas.yy.c y.tab.c -O ↵ → bas.exe

* Formal Grammar & its Applications in Syntax Analysis.

1) Formal Grammar: A FG (sometimes simply called Grammar) is a set of formation rules for strings in a formal language. The rules describe how to form a string from the language alphabet that are valid according to the language's syntax. $G = (V, T, P, S)$

V → V is a non empty set of finite variables

T → T is finite set of terminals

P → Production rules.

S → Start symbol

E (G) = E → E+E | E*E | id

V = {E}

T = {+, *, id}

P = {E → E+E | E*E | id}

S = [E].

If we want to generate a string from any grammar then we can use:

1) Leftmost derivation (LMD)

2) Rightmost derivation (RMD)

For a grammar, for a particular string, if more than one LMD, RMD and parse tree are possible then that grammar is ambiguous.

There is no algo to produce ambiguity of a grammar or we can say ambiguity of grammar is undecidable.

Scanned by CamScanner

To prove grammar is ambiguous we use hit & trial method.

Eg. $E \to E+E \mid E*E \mid id$

LMD

$E \to E+E$
$E+E*E$
$id+E*E$
$id+id*E$
$\to id+id*id$

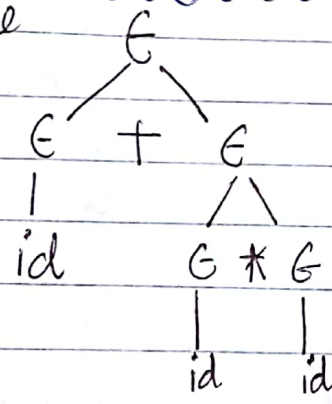
$E \to E*E$
$E+E*E$
$id+E*E$
$\to id+id*id$

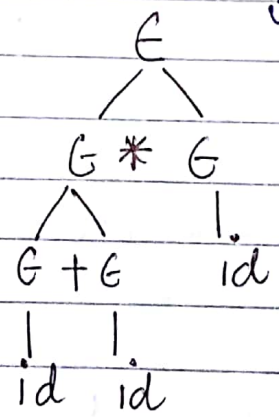
RMD

$E \to E+E$
$E+E*E$
$E+E*id$
$E+id*id$
$\to id+id+id$

$E \to E*E$
$E+E*E$
$E+E*id$
$E+id*id$
$\to id+id*id$

Parse tree





There is no algo. which can prove a grammar is ambiguous or unambiguous. Hence the ambiguity problem is undecidable in nature. To check whether a grammar is ambiguous we need to perform hit & trial.

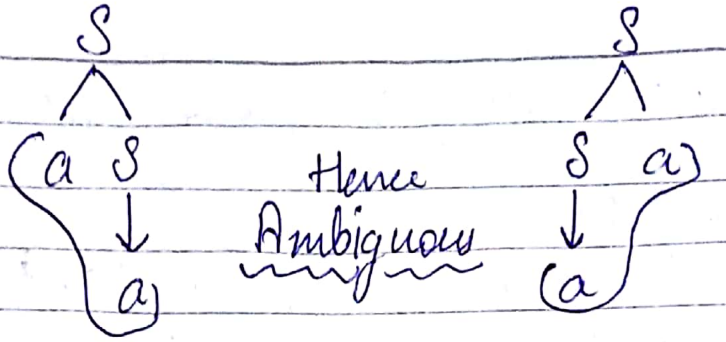
Q-1. $S \to aS \mid Sa \mid a$, $w = aa$

2. $S \to aSbS \mid bSaS \mid \varepsilon$, $w = abab$

3. $R \to R+R \mid R.R \mid R^* \mid a \mid b \mid c$, $w = a+b.c$

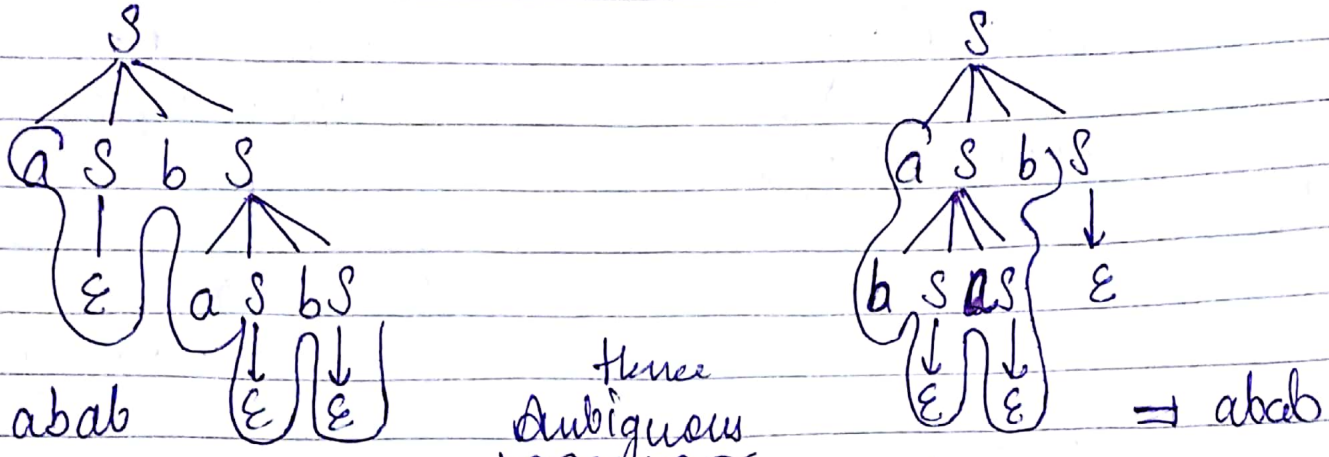
Scanned by CamScanner

1) Parse Tree



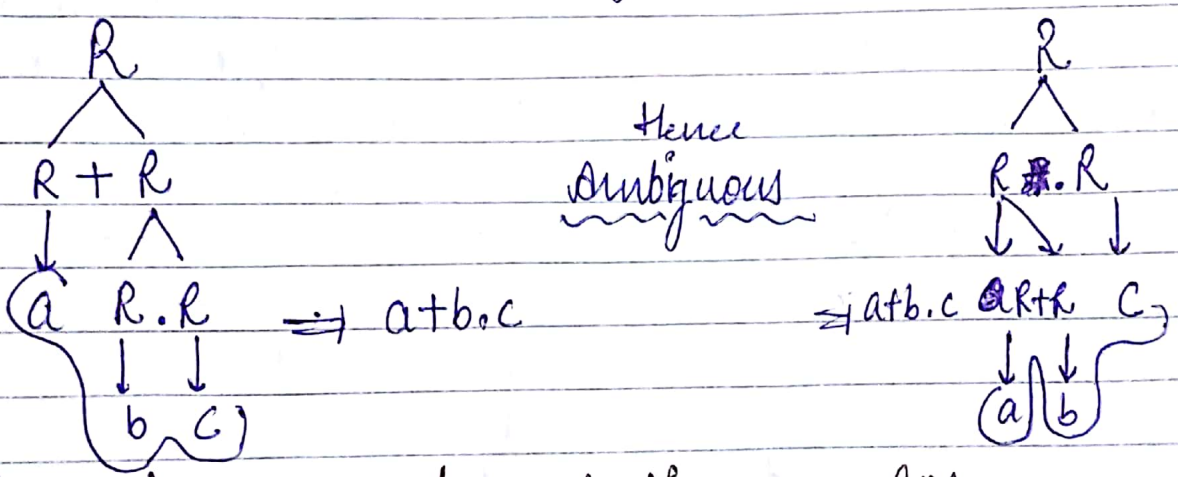
$\Rightarrow aa$ Hence Ambiguous $\Rightarrow aa$

2)



$\Rightarrow abab$ Hence Ambiguous $\Rightarrow abab$

3)



$\Rightarrow a+b.c$ Hence Ambiguous $\Rightarrow a+b.c$

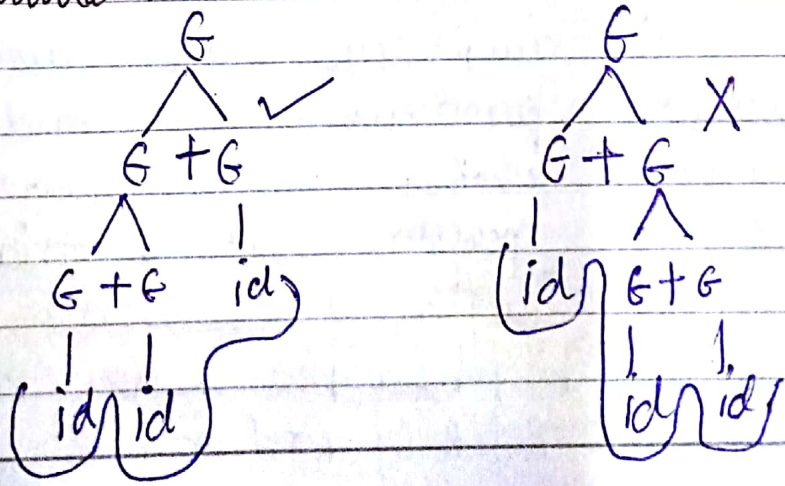
* How to convert Ambiguous Grammar to unambiguous Grammar.

1) Associativity Issue
2) Precedence Issue

(Associativity Issue)

$G \rightarrow G+G \mid G*G \mid id$

$w \rightarrow id+id+id$

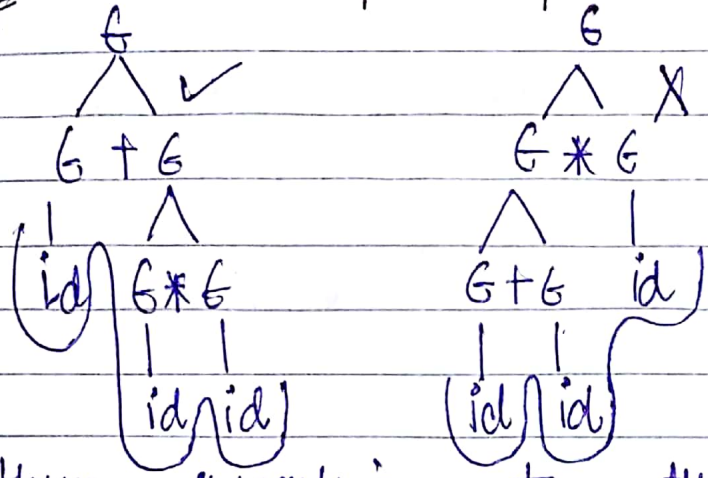


Scanned by CamScanner

The first tree is left associated, so it is correct. It is left recursive (growing in left direction) If any ~~~~ operator is right associative it is not right recursive.

- Thus, the associativity problem can be removed by recursion (Precedence Issue)

Q. $E \rightarrow E+E \mid E*E \mid id$ $w = id + id * id$



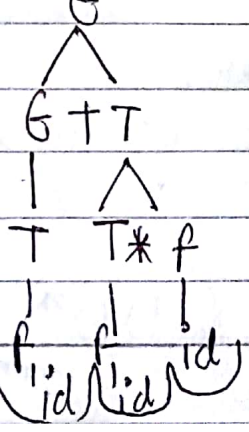
Operators which has high precedence has to be at the bottom of the parse tree so that they can be evaluated first.

Hence, according to this concept parse tree one is a correct version & it will return correct answer.

- Thus, the Precedence problem can be solved if we create multiple levels in parse tree so that the operators which has high precedence can settle themselves at the bottom.

To create levels we need to modify the above given productions.

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



The associativity problem is taken care with the help of recursion & precedence problem is taken care with the help of levels.

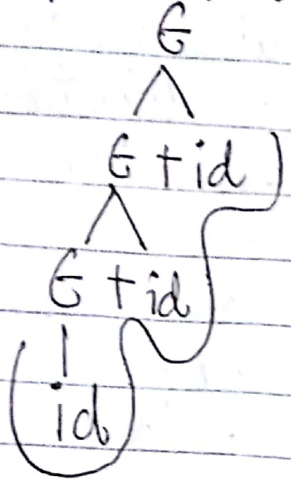
id + id * d

Modified (Precedence)

Scanned by CamScanner

Modified (Associativity) $E \rightarrow E+id/id$

$W = id+id+id$



id+id+id

* Classification of Grammar / Chomsky Classification.

The grammar is classified into 4 categories:

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted Grammar	Recursively Enumerable Language	Turing Machine
Type 1	Context Sensitive Grammar	Context Sensitive Language	Linear Bounded Automata
Type 2	Context Free Grammar	Context Free Language	Pushdown Automaton
Type 3	Regular Grammar	Regular Language	Finite State Automaton

1) Type 0 (Unrestricted Grammar)

Language defined by Type 0 grammar are accepted by Turing Machines.

Rule: $\alpha \rightarrow \beta$

$\alpha \in (V \cup T)^+$ $\beta \in (V \cup T)^*$

Eg: $S \rightarrow aBC$

$aB \rightarrow cA$

$Ac \rightarrow d$

Scanned by CamScanner

In the above rule LHS of the production cannot have ε. And RHS can have ε more.

2) Type 1 (Context Sensitive grammar)
Languages defined by Type 1 grammar are accepted by linear bounded automaton.
Rule: $\alpha \to \beta$
$|\alpha| \leq |\beta|$
& $\alpha, \beta \in (V \cup T)^+$

Eg: $S \to abc \mid aAbc$
$Ab \to bA$
$Ac \to Bbcc$
$bB \to Bb$
$aB \to aa \mid aaA$

3) Type 2 (Context Free Grammar)
Languages defined by Type 2 grammar are accepted by deterministic PDA or Non-deterministic PDA.
Rule: $\alpha \to \beta$
$\alpha \in V$
$\beta \in (V \cup T)^*$

Eg: $S \to aSa$
$S \to bSb$
$S \to \varepsilon$

4) Type 3 (Regular Grammar)
Languages defined by Type 3 grammars are accepted by finite state machine.
Rule: $A \to \alpha B \mid \beta$
$A, B \in V$
$\alpha, \beta \in T^*$
(Right linear grammar)

Eg: $A \to aB \mid a$
$B \to aB \mid bB \mid a \mid b$

$A \to \alpha B \mid \beta$
$A, B \in V$
$\alpha, \beta \in T^*$
(Left linear Grammar)

Eg: $A \to Ba \mid a$
$B \to Ba \mid Bb \mid a \mid b$

$A \to Ba \mid a$
$B \to aB \mid a$
} It is not Type 3 grammar because it contains both left & right linear grammar.

Scanned by CamScanner

* Context Free Grammar:

It belongs to Type 2 grammar and its tupple set is $G = (V, T, P, S)$ Eg: $S \to aAB$
$A \to bBb$
$B \to A/\varepsilon$

- **Derivation:**
Production rules are used to derive strings. The generation of language using specific symbol/rule is called derivation. In CFG, we have two types of derivation.

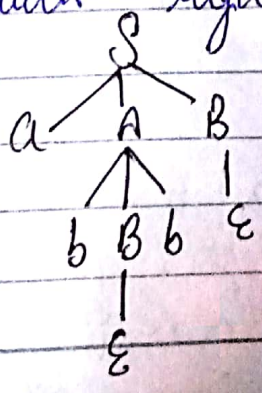
1) Left Most Derivation (LMD): Here, each step of left most variable in sentential form is replaced.
2) Right Most Derivation (RMD): At each step the right most variable in sentential form is replaced.

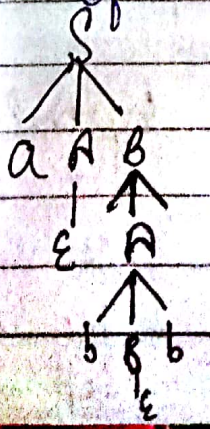
Q. $S \to aAB$
$A \to bBb$
$B \to A/\varepsilon$
$w = abb$

LMD
$S \to aAB$
$S \to abBb$
$S \to abb$

RMD
$S \to aAB$
$S \to aA$
$S \to abBb$
$S \to abb$

- **Derivation Tree/ Parse Tree:**
We can show productions by derivation or parse tree. A derivation tree is in which nodes are labelled with left side of production & in which child nodes are represented with right side of production.





Scanned by CamScanner

Properties of Derivation Trees

1) The root node has to be labelled with start symbol S.
2) Every leaf has a lable from $T \cup \{\varepsilon\}$
3) Every interior vertex has a label from V.
4) If a vertex has label $A \in V$, and its children are labelled $a_1, a_2 \ldots a_n$ then P must contain a production form as $A \to a_1, a_2, \ldots a_n$.
5) A leaf labelled ε has no siblings i.e. a vertex with a child labelled ε can have no other children.

Q $S \to SA$
$S \to A$
$A \to bSe$
$A \to be$
Derive bbebee

RMD
~~$S \to SA$~~
~~$S \to SbSe$~~

LMD
~~$S \to SA$~~
~~$S \to AA$~~
~~$S \to bSeA$~~
~~$S \to bAeA$~~
~~$S \to bbeeA$~~
$S \to bbebe$

LMD
$S \to A$
$\to bSe$
$\to bSAe$
$\to bAAe$
$\to bbeAe$
$\to bbebee$

Q. $S \to S+T$
$S \to T$
$T \to T*f$
$T \to f$
$f \to a$
$f \to (S)$
derive $a*(a+a)+a$

Scanned by CamScanner

* Backus Naur Form (BNF)

BNF is a language that enables you to specify legality in any language either human or computer. Basically, it's a language that defines the rules of another language. It is a Meta language used to describe grammar of a programming language. BNF is also used as a notation for CFG. and it is essential in compiler construction.

Eg: A smartphone consist of a touchscreen, speaker & microphone. ::= means consist of | defined by (Human language)

| means OR

{} means optional

AND, :', :: means AND

→ Smartphone ::= touchscreen AND speaker AND microphone.

We can define/explain the above BNF in syntactic notation.

→ <Smartphone> ::= <touch screen> :' <speaker> :' <microphone>

→ <Smartphone> ::= <touchscreen> :' <speakers> :' <microphone> :' <mini USB> | <apple port>

→ <Smartphone> ::= <touchscreen> :' <speaker> :' <microphone> :' <mini USB> | <apple port> :' {micro SD card}

(Computer language)

Eg: <digits> ::= 0|1|2|3|4|5|6|7|8|9

<if statement> ::= if (<condition>) <statement>

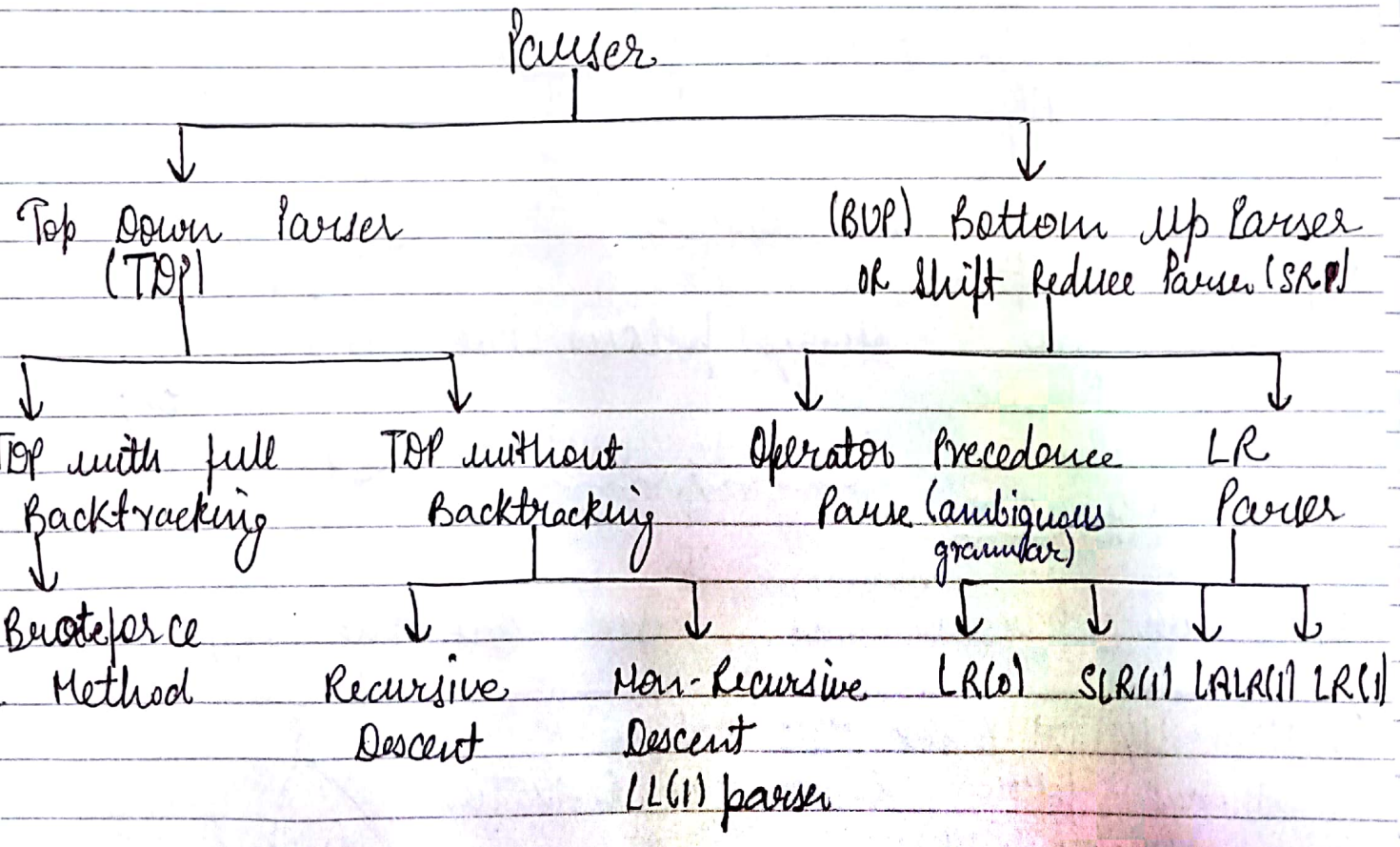
| if (<condition>) <statement> else <statement>

Scanned by CamScanner

* Capabilities of CFG:

1) CFG gives precise syntactic specification of programming languages.
2) A parser can be constructed automatically by using CFG.
3) The syntax entity specified in CFG can be used for translating into object code.
4) CFG is also useful for describing Nested structures such as balanced parenthesis, matching begin & ends and corresponding if, then, else, etc.

Unit : 2



Scanned by CamScanner