Book - Aho Ullman Sethi

Only Introduction of Subject.

Class-1 (Not attended)

→ Read about assembler. (2 pass assembler)
→ Read about current microprocessor and their evolution from 8s/86.

Read
High level language → Assembly language → Machine language
→ Linker, loader, relocatable code,
→ how memory is allocated.

Need of compiler → because machine only understands in binary, but it's difficult for us.

Translators → Compiler, interpreter, assembler,

Difference b/w compiler & interpreter

→ Interpreter is easier to implement. Interpretor checks line by line, compiler checks a set of statements.

Difference b/w error and warning.

→ Warning comes when for example, we don't use declared variable, space allocated is a wastage.

→ Context free grammar.

→ Lexical analysis, semantic, syntax analysis, code generation, optimisation of code (reducing size of code), efficient use of memory etc).

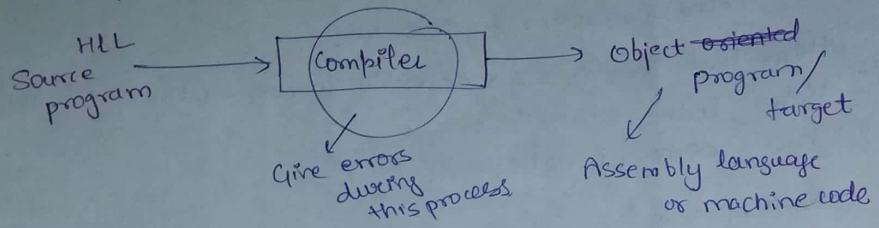
→ Code generation - final generation of another type of code

LEX, YACC (Unix/Linux/Ubuntu)
FLEX, BISON (Windows)
} to install in laptop

Class-2

Compiler : software

A big program that helps us to translate.



→ Set of instructions which change one type of language to another type of language (target)

What is need of having errors?

It doesn't stop when it detects some errors and indicate the errors to us.

How is compiler constructed? What is compilation?

Compilation

Language Analysis

- Syntax + Semantics.
- → Read the input
- → Constructs (whatever) according to the language

Semantics meaning :- check semantics with limited capacity.

(compiler is unable to gather infos about logic of program)

Synthesis

→ Intermediate code generation

↓

Code generation

(cannot convert directly to target)

→ Architectural independent compiler which we can run on different ~~car~~ architecture.

Then we can convert ~~to~~ Intermediate to actual target code acc^n to architecture of machines.

Assignment-1

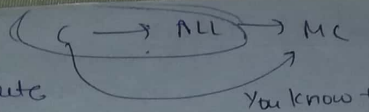
honor 7x

C++ program execution.

Secondary → Main memory Execute

Loading / Linking

Goes to assembly language, Then to machine language.



Analysis (3 Types) :-

- Linear → Ex- This is NSIT
  - ↳ Identify this is a word.
  - A list will be created and entered into symbol table
- Hierarchical → Hierarchical organize words in hierarchial structures like trees (Syntax Tree, Parse Tree)
  - ↳ Involves concept of Grammar & rules.
- Semantic
  - ↳ Type conversion / Division by two
  - Semantic Rules are defined & incorporated these rules in trees.

We will learn about NFA & DFA in linear analysis,

Synthesis :-

We are going to make 3 address code.

3 addresses and 2 operators

(3 variable stored at)

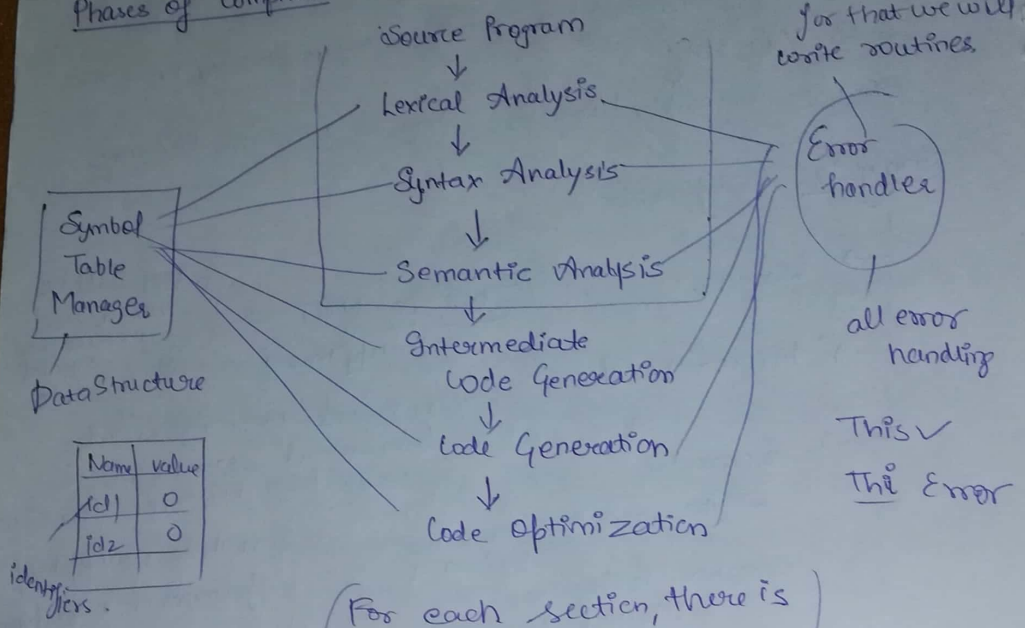
A = B + C ← Easy to convert it into assembly language

DS used :-

- DAG - Directed Acyclic Graph
- Tree → Symbol Tree

In assembly language, we can represent in the form of memonics.

Phases of Compiler



(For each section, there is a chapter)

Optimization :- Write the code which is efficient and used in minimal possible memory phase

Basic principle of compilation

↳ Compiler must preserve the meaning of program.

We can not change the concept written in our code

→ Data Structures
→ Algorithms
→ Heuristic Search
→ Optimization Techniques

} Should be known before

Class-3

Analysis Phase.

Identifier

P = i + r * 60

operator → value

↓

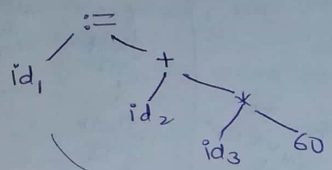
(1) Lexical Analysis

$id_1 := id_2 + id_3 * 60$

words are being formed here →

↓

Syntax Analysis



Now we need to check whether words are forming a sentence or not.

(bottom up) Syntax tree

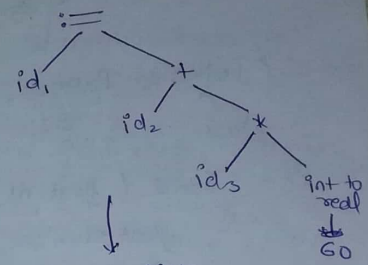
(2) Semantic Analysis

rules handling

div by zero,

type casting etc

the only function.



↓

Intermediate code generation

synthesis phase starts here

```
temp1 := into real (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1   := temp3
```

3 address code, 3 operations, 3 variables

Code written during synthesis phase

Code Optimizer

We try to reduce the no. of temporary variables required to execute.

```
temp1 := id3 * 60.0      (Here only 1 variable is used instead of 3)
id1   := id2 + temp1
```

↓

Code generator

source / dest.

Assembly level code using mnemonics.

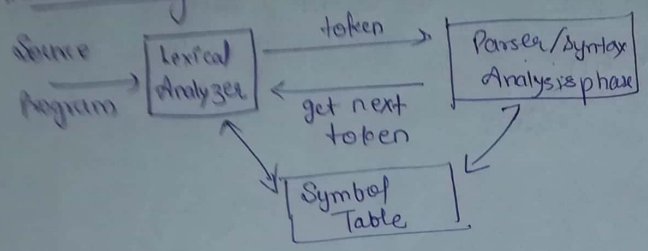
```
MOV  id3, R2
MUL  #60.0 R2
MOV  id2, R1
ADD  R2, R1
MOV  R1, id1
```

values of identifiers & names are kept in Symbol table.

↳ A table which acts as a store of all variables & their initial values.

1	f	
2	i	
3	r.	

Lexical Analysis



In this phase, we develop a recognizer & it will ~~develop~~ recognize the words of the language.

→ Token, Pattern, Lexeme
- Token → class.
- Pattern → rule
- Lexeme → word (some are operators, keywords..)

Ex-
```
Sum1 ( float n1, float n2)
{
    float sum;
    sum = n1+n2;
    return (sum);
}
```

from the character set of language.

```
Sum1    float
(       sum
float   ;
n1      .
;
float
n2
)
```

Rules that defines the class of words in a language is called Pattern.

n1, n2 → token identifier → class of words based on the rules

→ Lexical Analyser passes the token identified to the next phase called parser

→ Whatever tokens are identified by lexical analysis are simultaneously stored in symbol table.

→ Lexical Analyser also responsible for eliminating blank spaces & comments from the program.

→ We do not say that blank spaces & comments do not generate any tokens

Ex- `x = y + z * 2 // Comments //`

lexems → x, =, y, +, z, *, 2

lexeme < token, token attribute >

Ex- 2 <const, 2>
x <id, x>
+ <addop, +>
* <multiplyop, *>
z <id, z>
⋮
(Space) — < >
(comment) // < >
→ Empty
(Nothing is written there or nothing is passed on to the next phase)

(If a lexeme is not matching any pattern, then there will be an error, and errors will be notified).

→ For execution, program is first shifted from sec. to main memory.

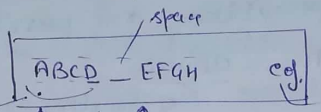
Buffer → temp storage space

To overcome overhead, we use buffering techniques, we use input buffers
↳ a major portion of program is kept in buffer.

→ Buffer size depends on architecture.

→ But we generally say that size of buffer is the same as the size of a disk block.

One buffer →



This is one word.

BP → Beginning Pointer

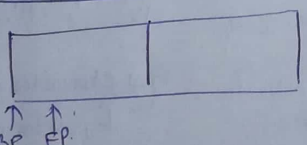
FP (Forward Pointer)

, means program is ended.

(It is difficult to use one buffer always, ∴ Two buffers)

2 pointers are used, initially both at start, FP is incremented when we get a space, whatever is there b/w BP & FP is one word. When space is encountered, BP too move ahead.

Two Buffers



As soon as FP reaches the end of first buffer, second buffer is loaded.
When FP reaches end of second, first is loaded.

Sentinel → any char or concept is used as end of file

As soon as sentinel is encountered, buffering stops.

9/8/2018
Class-4

How recognition is happening in LA?

We use Regular Expressions.

↳ We need to define all the different patterns in some notation.

A = { A ... Z, a ... z }

D = { 0 ... 9 }

for a variable name,

Rule is → A(A/D)*

Ex- int A1
int A1
int AAA
int A11100.

K = { if, while, do, else ... }

↓

Keywords (we define keywords by enlisting all of them in a list.

Specification of Tokens.

→ How we recognize tokens?

→ How do we form strings from expressions?

→ What operations are possible on strings?

- length abaab → 5 length
- Empty String, ∈ representation
- Find prefix, suffix, sequence, substring, subsequence

whatever character is specified above is part of language.

Operations on Languages — ① Union — LUD { A..Z, a..z, 0..9 }

= { a ... z, A ... Z }

= { 0 ... 9 }

② Concatenation → LD { A0, A1 ... B0, ... }

↑ First character of L, second of D

③ Closure

↙ We define 3 types of closure for a language

① Kleen Closure $L^* = \bigcup_{i=0}^{\infty} L^i$ (Empty included)
(0 or more concatenations of L)

② Positive Closure $L^+ = \bigcup_{i=1}^{\infty} L^i$ (Empty not included)
(1 or more concatenation of L)

Regular Expression (RE)

Epsilon. Rules →

1.) ϵ is a RE that ~~defines~~ denotes $\{\epsilon\}$ is empty set.

2.) If 'a' is a symbol in Σ then 'a' is a RE $\{a\}$

3.) If r & s are RE denoting language L(r) & L(s) then –

a) (r)|(s) is a RE denoting language L(r) ∪ L(s)

b) (r)(s) is a RE " " L(r).L(s)

c) r* is a RE " " $(L(r))^*$

Conventions used while writing RE.

1.) Unary operation * has highest precedence is left associated

2.) Concatenation has second highest precedence and is left associated.

3.) | has lowest precedence and is left associated.

Let $\Sigma = \{a, b\}$

RE $a|b = \{a, b\}$

RE $(a|b)(a|b) = \{aa, ab, ba, bb\}$ or $\{aa|ab|ba|bb\}$

$a^* = \{\epsilon, a, aa, aaa \ldots\}$

$(a|b)^*$ = zero or more instances of a or b $\{\epsilon, a, b, ab, ba \ldots\}$

$(a|a^*b) = \{ab, aab, aaab \ldots aaa\ldots b, a, b\}$

→ Find out the reason why we require empty character or empty set in a language?

Algebraic Properties of RE.

1.) $r|s = s|r$ | is commutative

2.) $r|(s|t) = (r|s)|t$ | is associative

3.) $(rs)t = r(st)$ Concatenation is associative

4.) $\epsilon r = r$

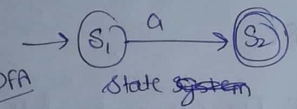
5.) $r\epsilon = r$

6.) $r^* = (r|\epsilon)^*$

7.) $r^{**} = r^*$

8.) $r(s|t) = rs|rt$ Concatenation is distributive over |

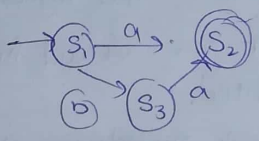
Theory of Automata - NFA, DFA.



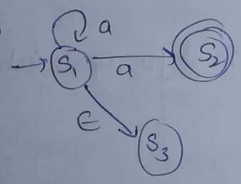
DFA

State ~~System~~ Diagram

	a
S_1	S_2
S_2	–



NFA



	a	ϵ
S_1	$S_2 S_1$	S_3
S_2	–	–
S_3	–	–

→ It is simpler to depict any situation in terms of NFA. because we ϵ there.

→ In DFA, deterministic, so difficult.

→ But no. of states is more so NFA consumes more states.

So we compress NFA to get DFA.

* lexical analyser input is a string and that is fed to the transition diagram and then rules are checked to verify whether the O/p is acceptable or not.

(Aho Ullman Sethi)

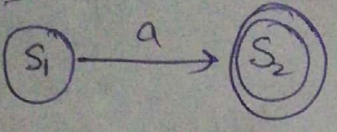
Finite Automata

23/8/18

- Non-deterministic NFA
- Deterministic DFA

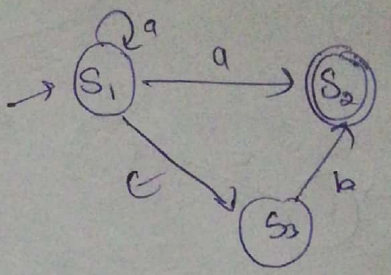
One initial state & more than one output state

Ex DFA



NFA

Can be input as ϵ



	a	b	ϵ
S_1	$\{S_1, S_2\}$		S_3
S_2	-	-	-
S_3	-	S_2	

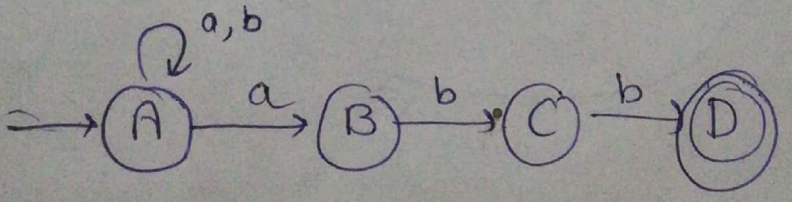
ϵ transitions are not allowed in DFA

Input tokenizer

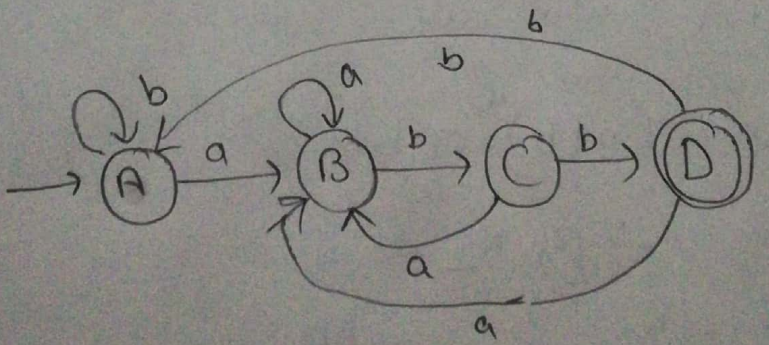
Scanning the input & identify the token.

$$R = (a|b)^* abb$$

Draw NFA first

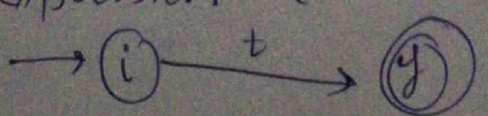


DFA

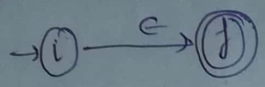


Thompson's Construction rules for constructing NFA.

1.) for an expression 't'



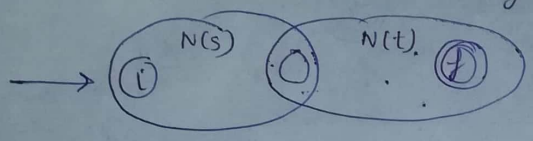
a) NFA for ε



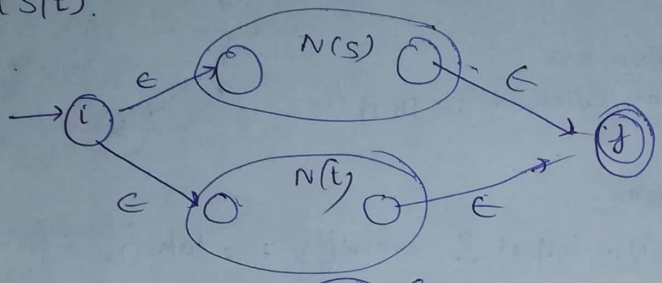
③ For Regular Expression s and t.

there will be merging of final state of s with initial state of t.

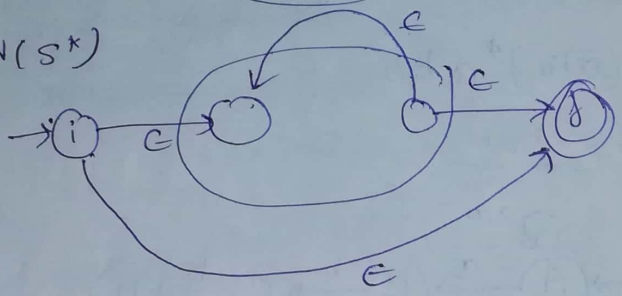
a) N(st)

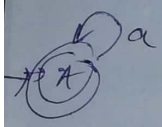


b) N(s|t).



c) N(s*)





Construct NFA using rule.

$R = (a|b)^* abb$

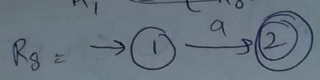
$= R_1 R_2$

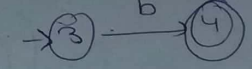
$R_1 = (a|b)^*$ $R_2 = abb.$

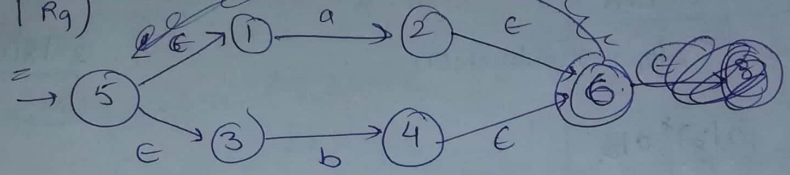
$R_3 = R_4 R_5 R_6$ $R_5 = b$

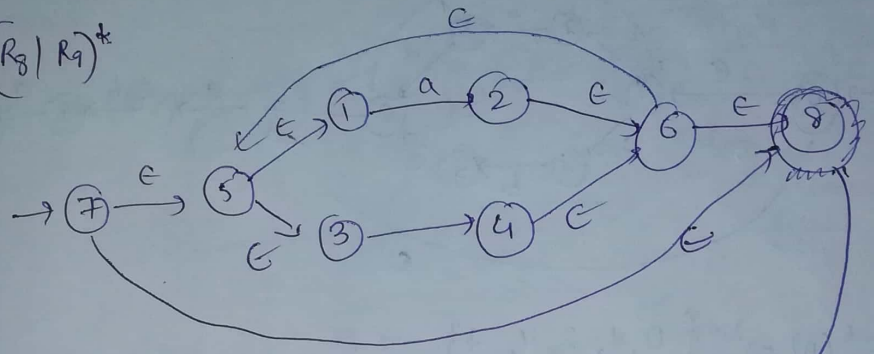
$R_4 = a$ $R_6 = b.$

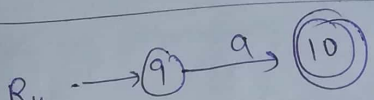
$R_7 = (R_1)^*(R_1)^*$ $R_7 = (R_8/R_9)^*$ $R_8 = a$ $R_9 = b$

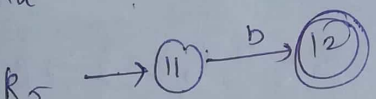
$R_8 =$ 

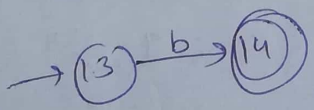
R_9 

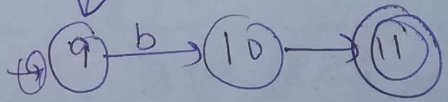
$(R_8 | R_9)$ 

$R_1 = (R_8 | R_9)^*$ 

R_4 

R_5 

R_6 

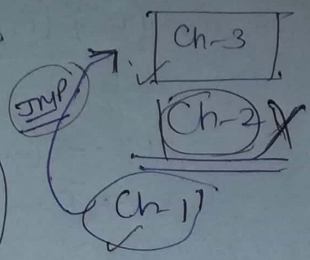
$R_3 = R_4 R_5 R_6$ 

a

Using Thomson Construction, draw NFA,

$R = a a^* | a b b^*$

$R = \left((a(a^*)) | (ab(b^*)) \right)$



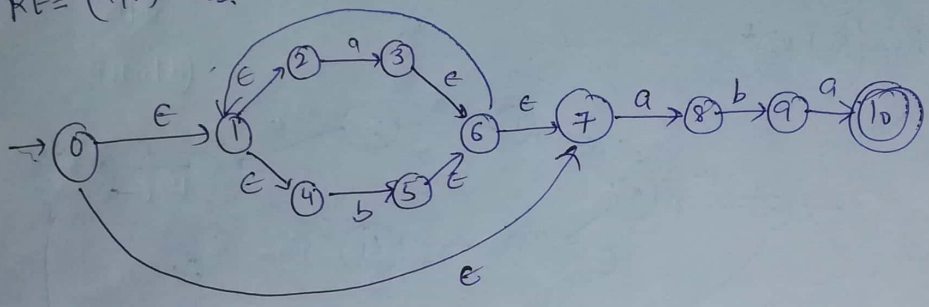
NFA to DFA

↳ Subset Construction

28/8/18

Class-6

RE = $(a|b)^* abb$



$\epsilon^*(0) = \{0, 1, 2, 4, 7\} \rightarrow A$

$(A, a) = \{3, 8\}$

$(A, b) = \{5\}$

$\epsilon^*(A, a) = \epsilon^*(3, 8) = \{3, 8, 6, 7, 1, 4, 2\} \rightarrow B$

$\epsilon^*(A, b) = \epsilon^*(5) = \{5, 6, 7, 1, 4, 2\} \rightarrow C$

$(B, a) = \{8, 3\} \rightarrow B$

$(B, b) = \{9, 5\} = \epsilon^*(9, 5) = \{9, 5, 6, 1, 2, 4, 7\} \rightarrow D$

$[C, a] = \{3, 8\} \rightarrow B$

$[C, b] = \{5\} \rightarrow C$

$[D, a] = \{3, 8\} \rightarrow B$

$[D, b] = \{5, 10\} \rightarrow E$

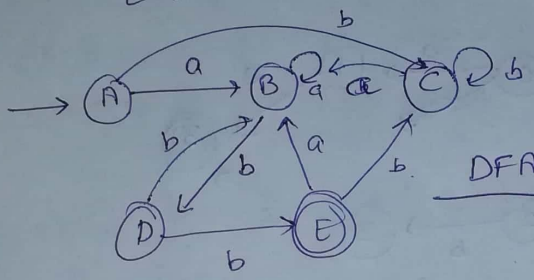
$\epsilon^*(E) = \{5, 6, 1, 2, 4, 7, 10\}$

$[E, a] = \{3, 8\} \rightarrow B$

$[E, b] = \{5\} \rightarrow C$

	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

because E contains final state



DFA

Minimisation of DFA

You can directly see on table and remove some states

or Equivalence State Partition Method.

$P_0 = (ABCD)\ (E)$

$P_1 = (ABC)\ (D)\ (E)$

$P_2 = (AC)\ (B)\ (D)\ (E)$

$P_3 = (AC)\ (B)\ (D)\ (E)$

LEX tool is doing all this in the background.

Phases of Compiler

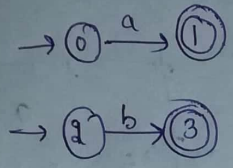
Minimised DFA

Q.1) Draw NFA, DFA and minimized DFA.

(i) RE = (a|b) b (a|b)*

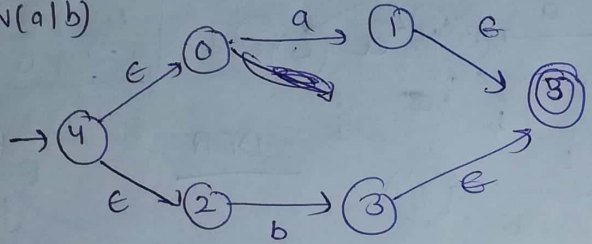
(ii) RE = b(a|b) b.

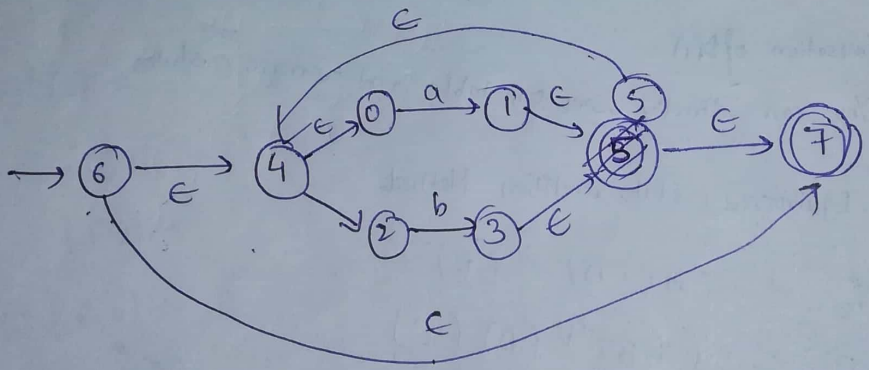
Sol:

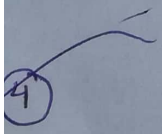


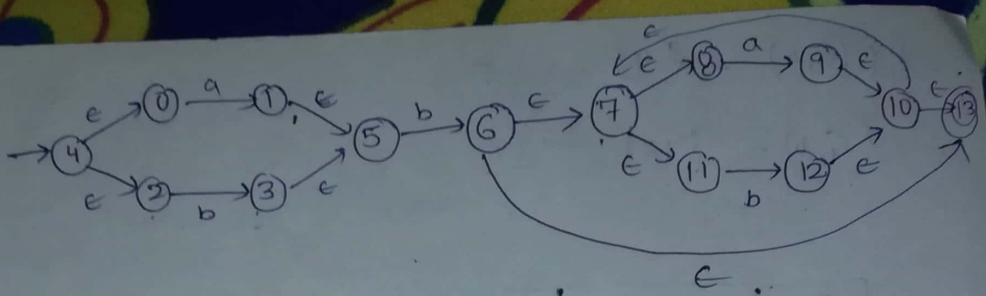
(a|b)

N(a|b)









~~$\epsilon^*(0)$~~

$\epsilon^*(4) = \{0, 2, 4\} \rightarrow A.$

~~Also~~ $(A, a) = \{1\}$

$(A, b) = \{3\}$

$\epsilon^*(A, a) = \{1, 5\} \rightarrow B$ ✓

$\epsilon^*(A, b) = \{3, 5\} \rightarrow C$ ✓

$(B, a) = \phi$

$(B, b) = \{6\}.$

$\epsilon^*(B, a) = \phi \rightarrow D$ ✓

$\epsilon^*(B, b) = \{6, 7, 8, 11\} \rightarrow E$

$(C, a) = \{\phi\}$ D

$(C, b) = \{6\}$ E.

$(E, a) = \{9\}$

$(E, b) = \{12\}$

$\epsilon^*(E, a) = \{9, 10, 13, 7, 8, 11\}$

$\epsilon^*(E, b) = \{12, 10, 13, 7, 8$

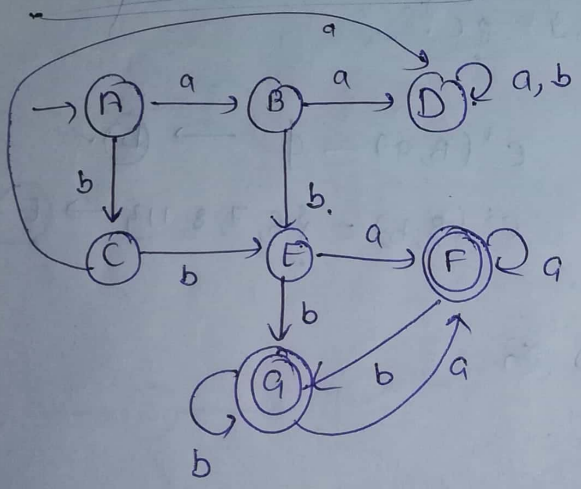
$(F, a) = \{9\} \rightarrow F$

$(F, b) = \{12\} \rightarrow 9.$

$\delta(G, a) = \{g\} \rightarrow F$

$(G, b) = \{12\} \rightarrow G$

PS	I/P = a	I/P = b
→ A	B	C
B	D	E
C	D	E
D	D	D
E	F	G
F	F	G
(G)	F	G



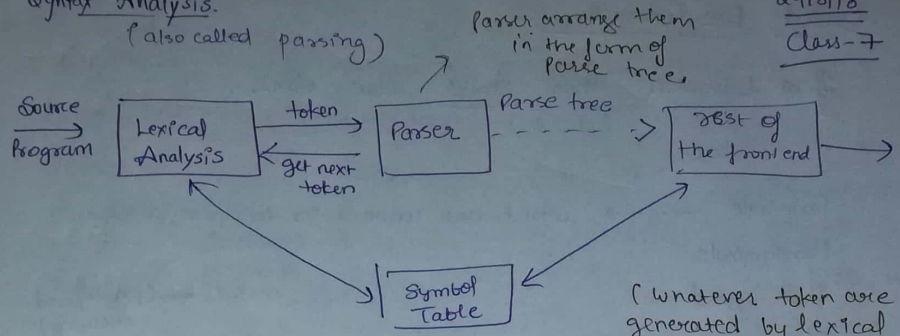
(3 final states aa rahi) (sabki) [check karna hai]

Syntax Analysis.

(also called parsing)

29/8/18

Class-7



(whatever token are generated by lexical analysis is passed to parse tree.

① → It obtain strings of token from lexical analysis.

② → It groups the token in some kind of syntatical ~~nature~~ structure like parse tree or syntax tree.

③ → It report syntax error also.

Errors during lexical analysis

- ↳ lexical errors → errors at the level of token.
- ↳ Syntax errors → you have to follow certain format (by given CFG) — during syntax analysis
- ↳ Semantic errors → division by zero.
- ↳ logical errors → divide by zero, infinite loop. ∞ loop

④ If it encounters errors, it reports them & recovers from them.

- Some rules written as error routines which can handle errors.
- ignore the error
- does small changes by itself.

After encountering errors, it should not stop but continue till end, till all errors detected.

→ This ~~stop~~ should not slow down overall compiler process.

CFG → used to depict rules in syntax analysis.

(In lexical, we used regular expressions) RE is unable to depict longer rules like those needed in syntax analysis phase.

(Static part can be represented by RE but not variable)

Context Free Grammar.

If Expression then stmt1 else stm2.

↓

A ≤ B

Terminals ⟶ which are constant in representation.
Ex- keywords, operators, specials (predefined meaning)

Non-Terminals.

Start Symbol

Production

Non-terminals

(whatever variable is there in sentence is non-terminal

↳ expression, stmt1, stm2.

Ex

stmt ⟶ if expression then stmt else stmt.

if we write it like this, then it called Production (of the language)

{ stmt → Expression.
stmt → if statement }

Start Symbol

(Set of productions, set of rules)

First symbol in first production is start symbol LHS.

If we have n productions, then the top most rule's left side is the start symbol, indicating where the program begins).

Eg.

Set of rules for language

start symbol

stmt → E

Expression → Expression op Expression Ex, A = B + C

Expression → (Expression) (A → B)

Expression → - Expression (A = -B.

Expression → id (can be identifier)

op → + | - | / | * | ↑

→ 6 productions

→ starts, stmt

Terminal

(,), op, id.

Non-terminal

stmt, expression

In short, E → E op E

E → (E)

E → -E

E → id → identifier.

op → +|-|/|*|↑

We can breakdown, expression, statements etc but we cannot breakdown identifies,

S → E → 1 product.

E → E op E | (E) | -E | id. → 4 productions

op → + | - | / | * | ↑ → 5 productions

Either expanded or contracted form

Some Rules (In Book)

* (Read from Book)

we can also write this like

op → +

op → -

op → /

op → *

op → ↑

Every rule is production.

* Terminals can't be expanded further.

Check if it matches or not
(a + b)

Grammar

E → E op E
E → (E)
E → -E
E → id
op → + | - | * | /

① (id + id) E op E ✗ matches.

E → (E)
E → (E op E)
E → (id op E)
E → (id + E)
E → (id + id)

↗ Yes matches.

Ex ② id + id * id.

E → E op E
E → E op E op E
E → id op E op E
~~E → id op id~~
E → id + E op E
E → id + id op E
E → id + id + E
E → id + id + id

we use a process called ↓ derivation

Left Most → This is left most

Right Most ←

Because you have to find out whether your grammar is ~~correct~~ or not

(id + id id * id)

not pass.

30/8/18

Class 8

Q.1 id + (id * id)

Find the leftmost & rightmost.

Soln:- $E \rightarrow E$ op E

$E \rightarrow (E)$

$E \rightarrow -E$

$E \rightarrow$ id

op $\rightarrow +|-|*|/$

Leftmost

$E \rightarrow E$ op E

$\rightarrow$ id op E

$\rightarrow$ id + E

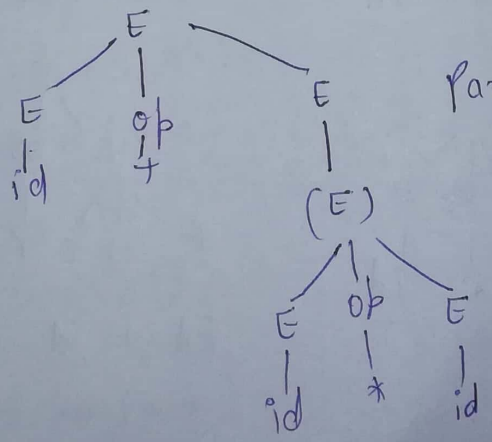
$\rightarrow$ id + (E)

$\rightarrow$ id + $(E$ op $E)$

$\rightarrow$ id + (id op E)

$\rightarrow$ id + (id * E)

$\rightarrow$ id + (id * id)

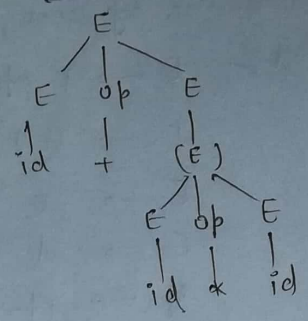


Parse Tree.

Rightmost. id + (id * id).

$E \to E$ op E
$E \to E$ op (E)
$E \to E$ op $(E$ op $E)$
$E \to E$ op $(E$ op id$)$
$E \to E$ op $(E * id)$
$E \to E$ op $(id * id)$
$E \to E$ + $(id * id)$
$E \to id + (id * id)$.

Parse Tree



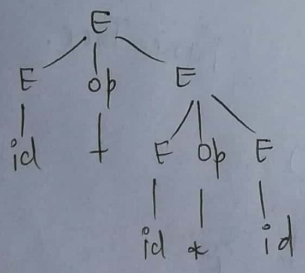
Q. Check whether for this, leftmost & rightmost are same or not.

id + id * id.

(Not same) (Depends on you) can be same

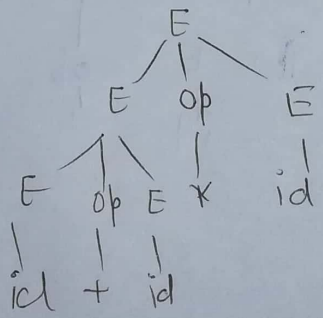
Sol. Left

$E \to E$ op E
$E \to$ id op E
$E \to$ id + E
$E \to$ id + E op E
$E \to$ id + id op E
$E \to$ id + id * E
$E \to$ id + id * id.



Right

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



This grammar is ambiguous

For ex, we can do this like

$E \to E+E$ or $E \to E*E$.

$op \to + | - | / | *$

is ambiguous.

Grammar

$S \to (L) | a$

$L \to L, S | a$.

S, L - Non-terminals

( ), a, → Terminals

Find Parse tree for ① (a,a)

② (a, (a,a))

Construct leftmost & rightmost derivation. Find grammar is ambiguous / not,

(For type declaration grammar,

$T \to$ type id / type id, id)

① $S \to (L)$

~~$S \to (a,a)$~~

Left

$S \to (L,S)$

$S \to (a,S)$

$S \to (a,a)$

$S \to (L)$

$S \to (L,S)$

Right

$S \to (L,a)$

$S \to (a,a)$

② w/t

$S \to (L)$

$S \to (L,S)$

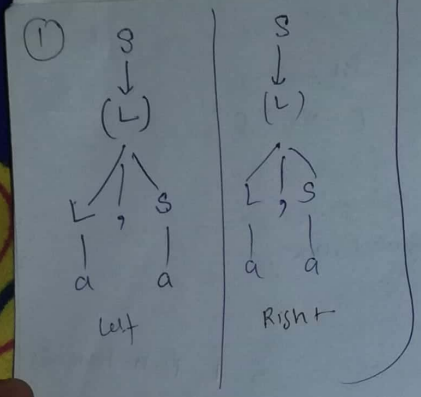
$S \to (a,S)$

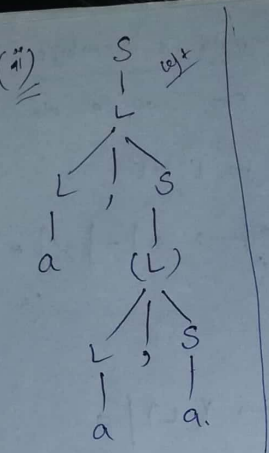
$S \to (a,(L))$

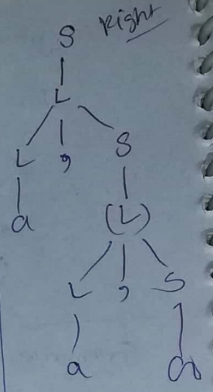
$S \to (a,(L,S))$

$S \to (a,(a,S))$

$S \to (a,(a,a))$







(2) Right

$S \to (L)$
$S \to (L, S)$
$S \to (L, (L))$
$S \to (L, (L,S))$
$S \to (L, (L, a))$
$S \to (L, (a, a))$
$S \to (a, (a, a))$

Result
Not ambiguous

(Ambiguity is difficult for compiler to decide which path to follow)

Leftmost → Left sentencial form, rightmost → right sentencial form

→ Language generated by CFG, is called CF language
→ If two grammars generate same language then grammar are said to be equivalent.

→ useless symbol
→ Cycle
Do in automata

6/9/18
Class-9

Removing Ambiguity.

$S \rightarrow \alpha S \beta S Y \mid \alpha_1 \ldots \mid \alpha_n$

$S \rightarrow \alpha S \beta S' Y \mid S'$

$S' \rightarrow \alpha_1 \mid \alpha_2 \ldots \mid \alpha_n$.

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

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

→ Check whether this is ambiguous or not.

id + id * id.

Do two leftmost and two rightmost.

Leftmost

$E \rightarrow E + E$

$\rightarrow id + E$

$\rightarrow id + E * E$

$\rightarrow id + id * E$

$\rightarrow id + id * id$

$E \Rightarrow E * E$

$\rightarrow E + E * E$

$\rightarrow id + E * E$

$\rightarrow id + id * E$

$\rightarrow id + id * id$.

Now the above strings has two leftmost & two rightmost

∴ ambiguous.

Using soln rule

1 id + id * id

$E \to E + T \to T + T \to F + T \to id + T \to id + T * F \to id + F * F \to id + id * id$

We can't reach the above soln via $E \to T$

∴ There is only 1 way to derive the above string.

∴ Ambiguity in Grammar is removed.

Left factoring. (When left symbol is same in production, how ll complete know the right path)

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

$$A \to \alpha A'$$

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

Soln:- Whatever common is dealt with first and other passes on.

$$E \to \underbrace{E}_{\alpha} \underbrace{+E}_{\beta_1} \mid \underbrace{E}_{\alpha} \underbrace{*E}_{\beta_2} \mid id$$

$$E \to EE' \mid id$$

$$E' \to +E \mid *E$$

Example:-

$$S \to \underbrace{iEtS}_{\alpha(\beta_1)} \mid \underbrace{iEtSeS}_{\alpha\ \beta_2} \mid a$$

α → expression, β_1 → then statement, β_2 → and else statement, a → identifier

$$S \to iEtSS' \mid a$$

$$S' \to \epsilon \mid eS$$

Left Recursion.

Non-terminal on both LHS, RHS

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

$A \rightarrow \beta A'$

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

Remove left recursion

Example

(1) $E \rightarrow E + T \mid T$

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

$E \rightarrow TE'$

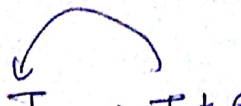
$E' \rightarrow +TE' \mid \epsilon$

(2) $T \rightarrow T * F \mid F$

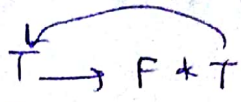
$A \rightarrow A \alpha$

$T \rightarrow FT'$

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



$T \rightarrow T * F$ Left Recursion



$T \rightarrow F * T$ Not left recursive

---

1. Check left recursion

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

Left Recursion

$A \rightarrow \underbrace{Ac}_{A\alpha} \mid \underbrace{Aad}_{A'\alpha} \mid \underbrace{bd}_{\beta} \mid \epsilon$

remove recursion

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \epsilon$

This is called indirect left recursion

Rules :- (1) Write systematically all the products.

(2) Replace Top with bottoms (with 2 and so when req

(3) If a left recursion exist in 2nd, then it is removed first

Before passing start with (syntax based)?

RE to CFG.

For all RE, we have CFG,
but for all CFG, we not have RE.

Why not using CFG for all?
- Difficult to understand.
- For modularising, syntax & lexical.
  ↓ why
  - easy constraint
  - Independent of each other.

→ There is a way to convert RE to CFG.

CFG is strongest notation compared to RE. For identification of words, we don't require such a strong tool. ∴ we use RE.

Rules

① For each state i of NFA create non-terminals symbols A_i.

② If state i has a transition to state j on input symbol a, introduce production like this,

$$A_i \longrightarrow a A_j$$

③ If state i goes to j on ϵ, then introduce production

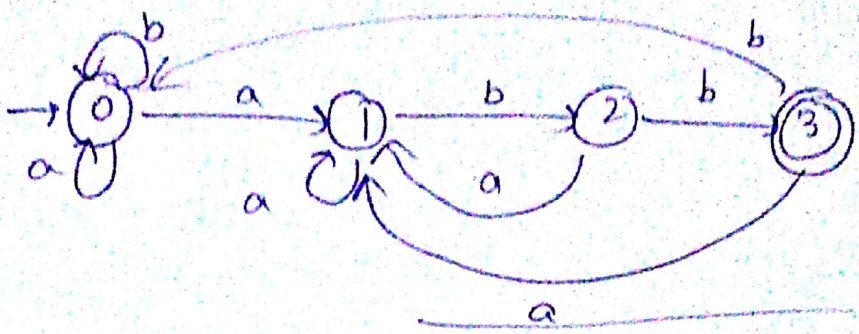
$$A_i \longrightarrow A_j$$

④ If i is an accepting state, then introduce,

$$A_i \longrightarrow \epsilon$$

⑤ If i is the start state, then make A_i as the start symbol of Grammar

Q>/ $(a|b)^* abb$.



NFA

Soln

$$A_0 \longrightarrow aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \longrightarrow aA_1 \mid bA_2$$
$$A_2 \longrightarrow aA_1 \mid bA_3$$
$$A_3 \longrightarrow bA_0 \mid aA_1 \mid \epsilon$$

Ch-1
Ch-3
Ch-4 till 4.3

1.1
1.2
1.3 (Evolution) X (For mid sem)
1.4 X

Ch-3 Imp.

3.1
3.2
3.3
3.4
3.5 (Lex Tool) officially you should know them
3.6
3.7 (Conversion of NFA to DFA) only subportion only

4.1
4.2
4.3

3.8 X
3.9 (Only minimisation whatever has been taught in class)