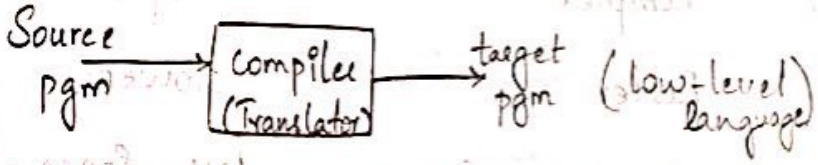
9/10/19

Introduction

Compiler :-

Translates/Converting one Programming language into another.

→ Translate an executable program in one language into an executable pgm usually in another language.



C, C++

Python e.t.c has no compiler. only interepreter.

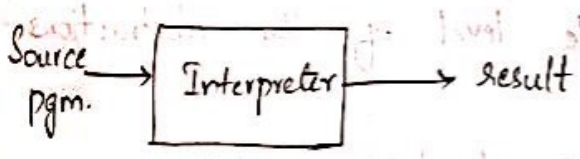
Programming Language :-

It has rigid property

It has no ambiguity.

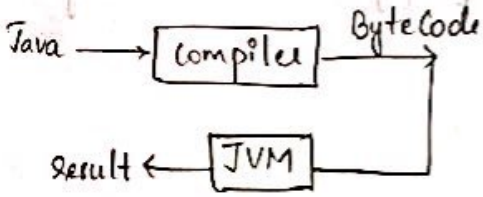
→ A formal language described for expressing computations and also have rigid property and meaning and no ambi

Interpreter :-



* Need not convert into target pgm. it will directly give o/p.

Eg:- Java



Difference between Compiler and Interpreter.

Learning Dimension	Compiler	Interpreter
1) Defn		
2) Examples	C, C++, FORTAN	Python, Perl, Scheme
3) Looking at	Entire piece of code	Line-by-line
4) Easiness to write	complex	Simpler
5) Execution Speed	Faster	Slower.
6) Space	Occupies max space	less space.
7) Target pgm	object code	Simplified intermmediate code.

11/1/19

Why study compiler construction?

Ans: 1. Efficiently handle complex algo problems.
2. App of theory to practice
3. Primary responsibility for application performance.
4. Design the appropriate level of the abstractions.

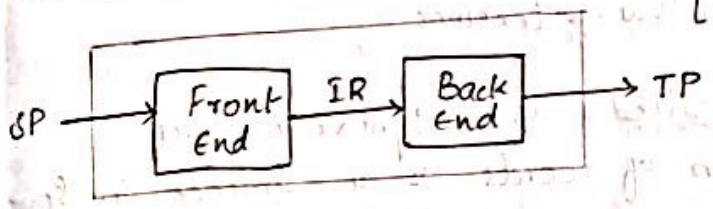
→ Fundemental principles of compiler / objectives / challenges.

(1) preserve meaning.

(2) Must improve performance of pgm. in some way.

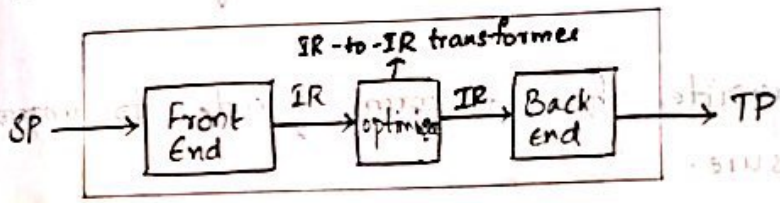
Compiler Structure:-

Two-phase Compiler:



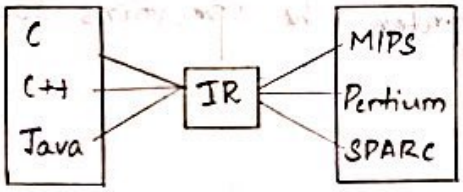
* [IR → Intermediate Rrepresentation
FE o/p → IR
BE I/p → IR]

Three-Phase Compiler:

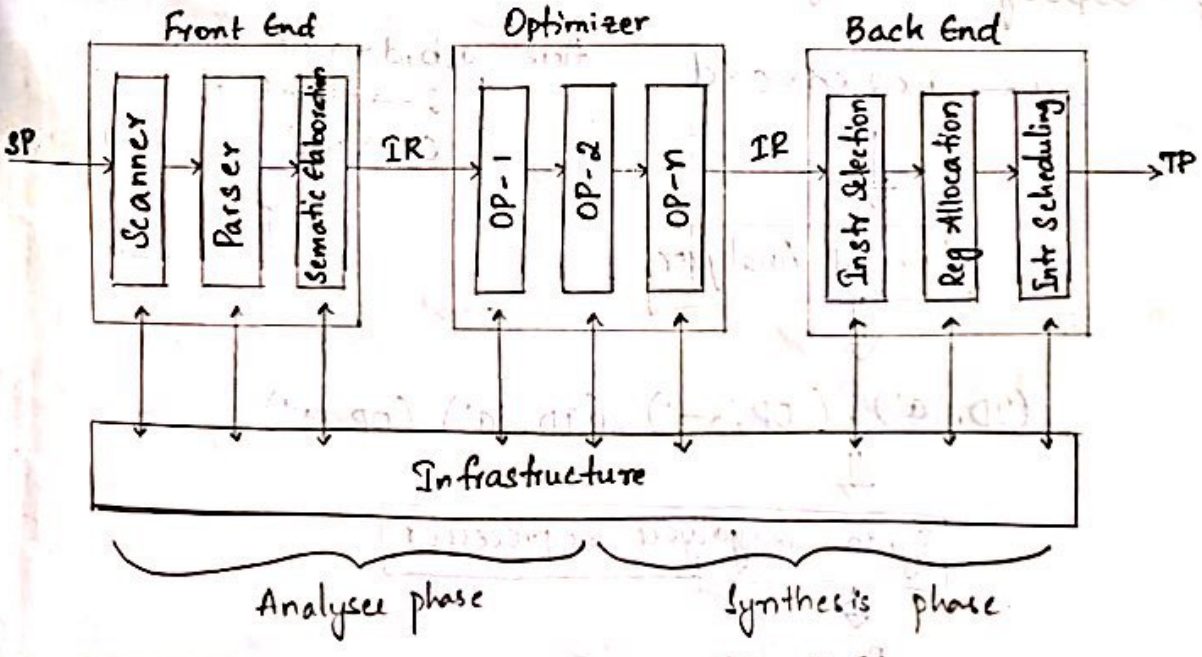


Retargeting:

Defn:- Task of changing compiler to generate code for new processor.



Structure of a Typical Compiler:-



* Exp:-

(1) Scanner is also called as Tokenizer / lexical Analyzer.
it translates stream of char to words

words → Tokens ; char → Lexemes.

(2) Parser is also called as Syntax Analyser.
it determine stream of words is a Sentence in Source la

(3) Sematic Anglyser/Elaboration check deeper meaning and built appr
IR.

(4) Optimiser will rewrite the IR form of code to improve it
in some measure.

(5) Instruction Selection will rewrite the IR into target langua
operations like Assembly level lang pgm.

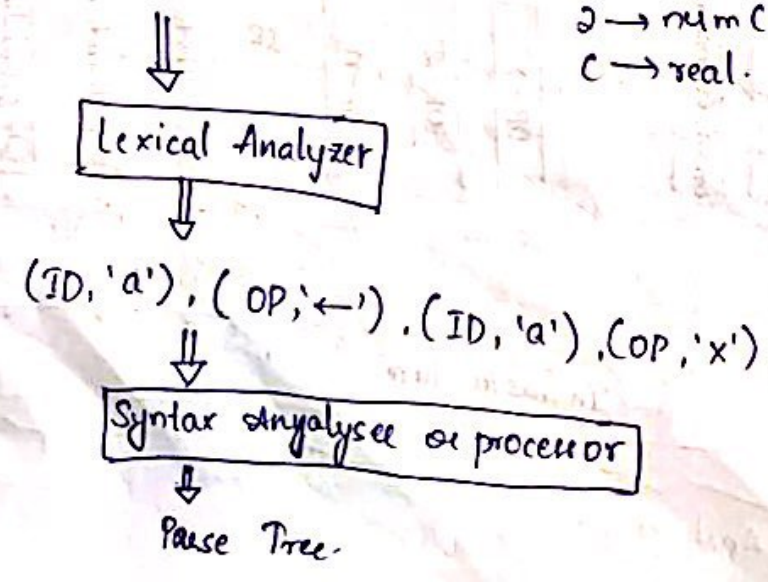
(6) Reg Allocation will rewrite the code to fit to the finite
reg of Set of the target machine.

(7) Intruction Scheduling will reorder the operations so that they
run faster

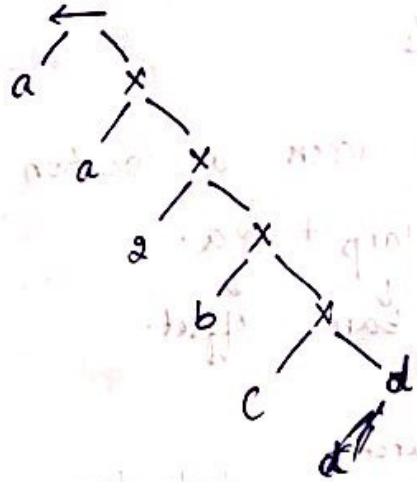
6/11/19

* → Compiling of a given Sentence:-

a ← a x 2 x b x c x d here a,b,d → int
2 → num (int)
c → real.



→ Parse Tree:-



⇓

Semantic Elaboration

⇓

optimizer

⇓

[Three add code] = low level Assembly code

$t_0 = a \times 2$
$t_1 = t_0 \times b$
$t_2 = t_1 \times c$
$t_3 = t_2 \times d$

⇓

Optimizer

⇓

$t_0 = a \times 2$
$t_1 = t_0 \times b$
$t_2 = t_1 \times c$
$a = t_2 \times d$

⇓

Instruction Selection

⇓

writing ILOC code

↓

Intermediate language for an optimising compiler.

(Y) It is an abstract assembly language for a simple RISC processor

→ This ILOC code assumes that a, b, c & d are located at offsets @a, @b, @c, & @d from an add contained in the register [illegible]

* r_{arp} → activation record pointer. This will have the base address.

If we want to access a location of any variable then loc = r_{arp} + @a.
(r_{arp} → base, @a → offset.)

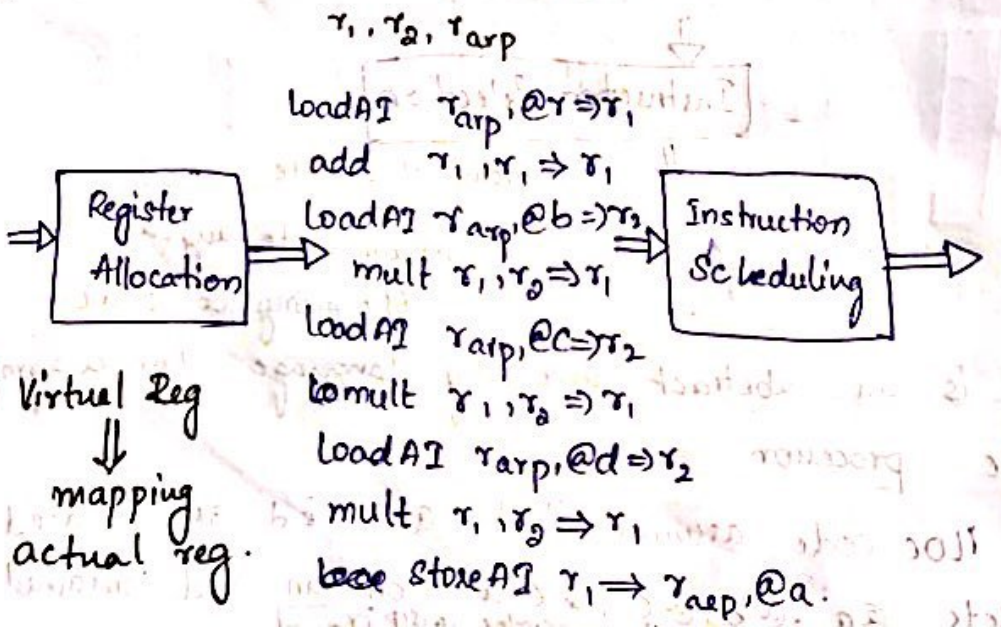
* ILOC code:-

```
loadAI  r_arp, @a  ⇒ r_a      (source, destination)
loadI   2          ⇒ r_2
load AI r_arp, @b  ⇒ r_b
load AI r_arp, @c  ⇒ r_c
load AI r_arp, @d  ⇒ r_d
mult    r_a, r_2   ⇒ r_a
mult    r_a, r_b   ⇒ r_a
mult    r_a, r_c   ⇒ r_a
mult    r_a, r_d   ⇒ r_a
StoreAI r_a        ⇒ r_arp, @a
```

⊗ Note:-
No. of memory cycles req for instr in ILOC
- loadAI = 2
- StoreAI = 2
- all other = 1
- mult = 2

08/11/19

→



Before Recording			After Recording		
Start	End	ILOC code	Start	end	ILOC code
1	3	——	1	3	loadAI r_{arp}, @a $\Rightarrow r_1$
4	4	. — .	2	4	loadAI r_{arp}, @b $\Rightarrow r_2$
5	7	. — —	3	5	loadAI r_{arp}, @c $\Rightarrow r_3$
8	9	— — — .	4	4	add $r_1, r_1 \Rightarrow r_1$
10	12	. — — —	5	6	mult $r_1, r_3 \Rightarrow r_1$
13	14		6	8	loadAI r_{arp}, @d $\Rightarrow r_2$
15	17		7	8	mult $r_1, r_3 \Rightarrow r_1$
18	19		9	10	mult $r_1, r_2 \Rightarrow r_1$
20	22		11	13	storeAI $r_1 \Rightarrow r_{arp}$, @a

12/11/19

Chapter #2

→ Scanners :-

2.1 Intro

2.2 Recognizing words
- Formalism for Recognizers
- Recognizing more complex words

2.3 Regular Expression
- Notation
- Examples.

2.4 From Regular Expression to Scanners

Lexeme :-

Actual text for a word recognized by an fi

Automata.

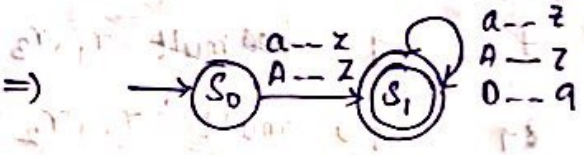
Token :- Type: identifier.

* Identifying variable :-

$(S, \Sigma, \delta, S_0, S_A)$

new

$S_0 \rightarrow S_1$ — —



Regular Expression :-

[a--z] = a|b|c - - -|z

RE = ([a--z]|[A-Z])([a--z]|[A-Z]|[0--9])*

* Unsigned Integer :-

RE = 0 | [1--9][0--9]*

* Unsigned Real Numbers :-

([1--9][0--9]*)(ε|.[0-9]*)

*Scientific Notation :- 10.23E+23

* No. of Reg → R0 - R31 with an optional leading 0 on sin
digit reg name. (is valid)

13/11/19

Regular Expression:-

(1) r_0 to r_{31}

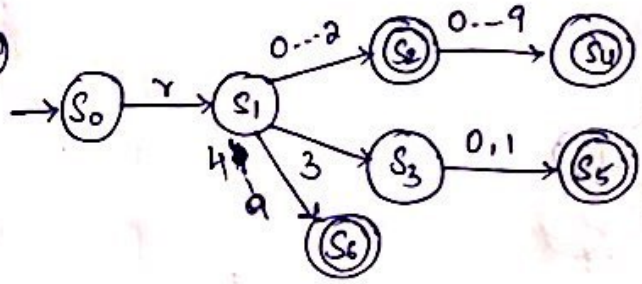
(2) PAN ⇒ format UUUUUDDDDU

(3) Passport-No ⇒ Assumption

↑

Eg: A12 34567.

(1A)



r_{00}
r_{01} r_0
r_{10}
r_{20}
r_{31}
r_{30}
r_{31}

$RE = r[[0,1,2]\cdot[0,--,9] + [3]\cdot[0,1] + [4,--,9]]$

(2A) PAN → UUUUUDDDDU (D = digit)

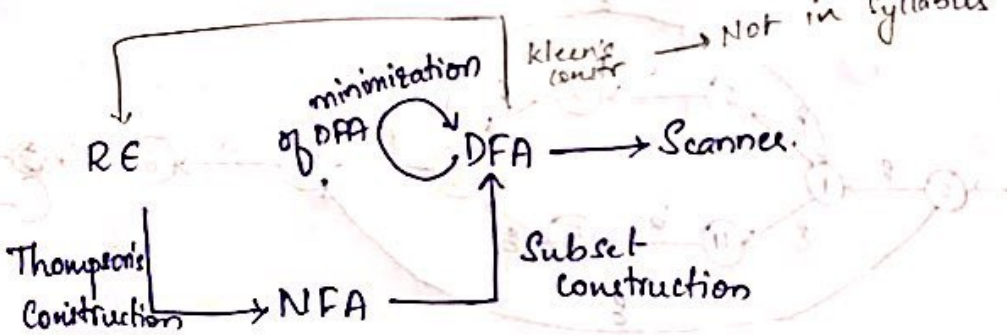
$RE = [A--Z]^5[0--9]^4[A-Z]$

(3A)

RE ≠ First char can't be = x, Q, Z.

$([A--P] \mid [R--W] \mid Y)[1--9][0---9]$ space $?[0--9]^4[1--9]$

From RE to Scanner:-

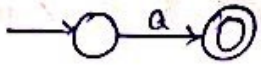


14/11/19

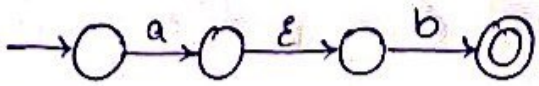
Regular Expression to NFA :-

* Thompson's construction :-

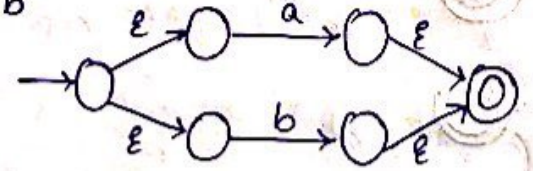
a



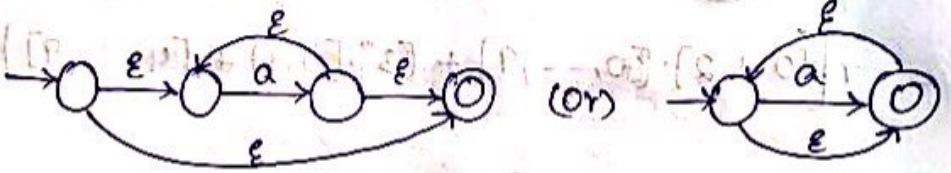
ab



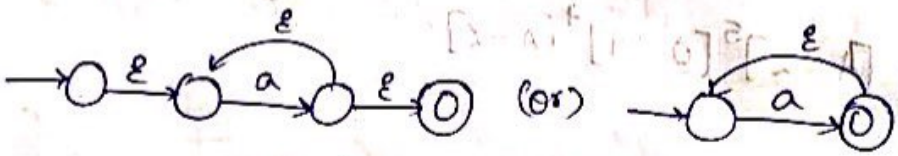
a/b



a*



a+

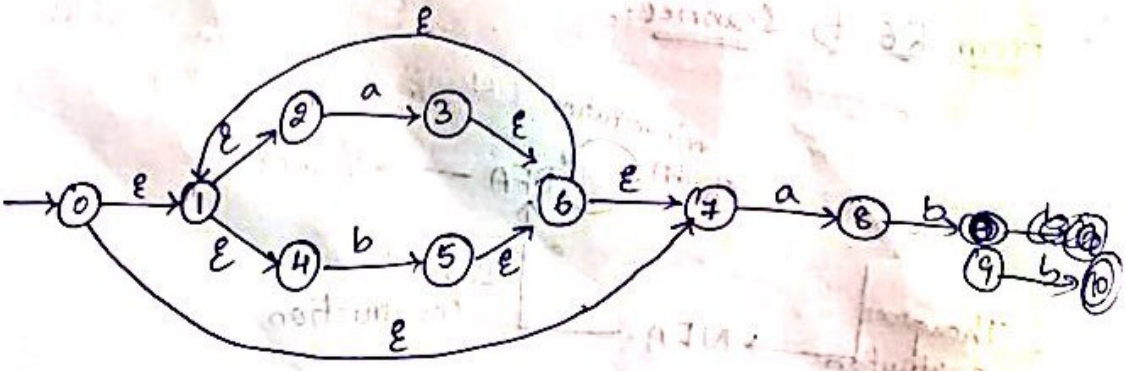


NFA to DFA :-

Subset constr. method

Eg:- $(a/b)^*abb$ to minimized DFA ?

Ans:-



DFA

States	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

A = ε-closure (0) = {0, 1, 2, 4, 7}

B = ε-closure (3, 8) = {3, 6, 7, 1, 2, 4, 8}

= {1, 2, 3, 4, 6, 7, 8}.

C = ε-closure (5) = {5, 6, 7, 1, 2, 4}

= {1, 2, 4, 5, 6, 7}.

Minimization of DFA:-

Two states S1 & S2 are equivalent iff both states are final or nonfinal states

⇒ if $S_1 \equiv S_2$ then replace those 2 states by one representative state.

Eg: from above example dfa table.

A ≡ C [∵ changing C with A]

		a	b
→	A	B	A
	B	B	D
	D	B	E
*	E	B	A

→ Minimized DFA

Eg:-

① $4(0|123)^*$

② $4(1|2|3)^*$

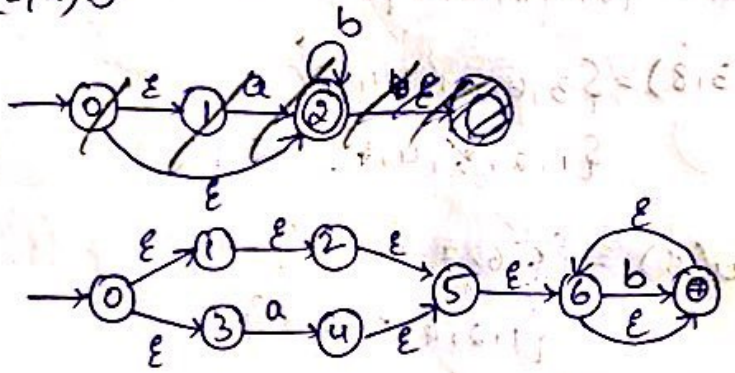
③ $(a|b)^*ab(a|b)^*$

④ $(\varepsilon|a)b^*$

⑤ $(a|b|c)^*(a|b)^*$

Solutions:-

④A $(\varepsilon|a)b^*$



$\rightarrow A \rightarrow \varepsilon\text{-}c(0) \Rightarrow \{0, 1, 3, 2, 5, 6, 7\}$

	a	b
A	a	b
→A	B	C
B	ϕ	C
C	ϕ	C

$B(4) = \{4, 5, 6, 7\}$

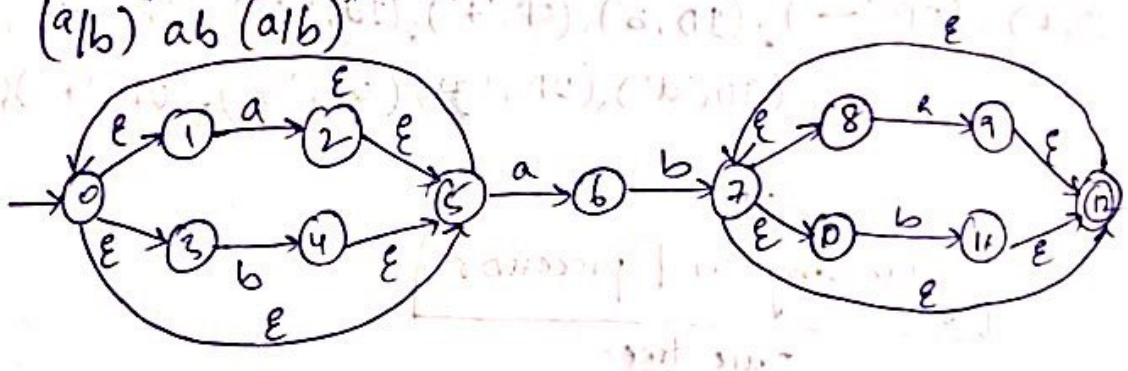
$C(7) = \{6, 7\}$

min

	a	b
→*A	B	B
*B	ϕ	B

$B \equiv C$

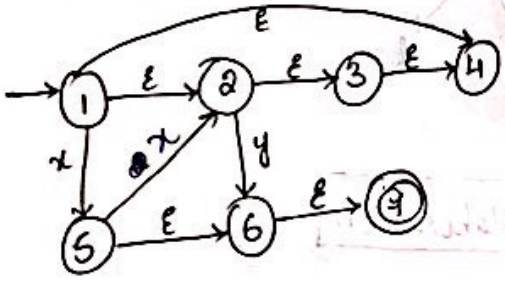
(3A) $(a/b)^* ab (a/b)^*$



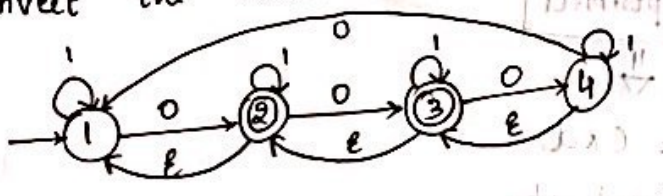
15/11/19

1Q. Show the o/p of the phase compiler for the expression given below. a = (b+(c*d))+e, Here d is real?.

(2Q). Construct a minimal DFA for the following



(3Q) Convert the below NFA to DFA using Subset construction



(4Q) Write RE for Roll no of a university students (UG stud
yr of admission

Sol:- (Cb|am|bg)(EN).U4).(CSE|ECE|MEE|GNG|EEE).
(0[3,4,5,6,7,8,9]/1[1,---,9])(0---9)^3.

Solution:-

1A. a = b+c*d+e. a,b,c,e → Int, d → real

⇓

Lexical Analyser

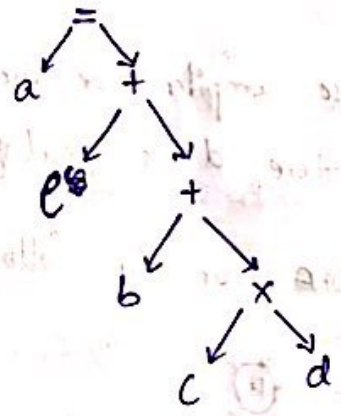
(ID,'a'), (OP,'←'), (OP,'(') (ID,'b'), (OP,'+'), (OP,'(') (ID,'c'), (OP,'*'),
(ID,'d'), (OP,'x') (OP,')'), (OP,'+'), (ID,

⇓

Syntax analyser / processor

parse tree.

⇓



⇓

Semantic elaboration

⇓

Optimiser

⇓

$t_0 = c \times d$

$t_1 = t_0 + b$

$t_2 = t_1 + e$

$t_3 = t_2$

⇓

Optimiser

⇓

$t_0 = c \times d$

$t_1 = t_0 + b$

$a = t_1 + e$

⇓

Instr Selection

⇓

Iloc code

```
loadAI   r_arp, @b ⇒ r_b
loadAI   r_arp, @c ⇒ r_c
loadAI   r_arp, @d ⇒ r_d
loadAI   r_arp, @e ⇒ r_e
mult     r_c, r_d ⇒ r_a
add      r_a, r_b ⇒ r_a
add      r_a, r_e ⇒ r_a
storeAI  r_a ⇒ r_arp, @a
```

⇓

Reg allocation

Start	End	code
1	3	loadAI r_{arp}, @c ⇒ r_1
4	6	loadAI r_{arp}, @d ⇒ r_2
7	8	mult $r_1, r_2 \Rightarrow r_1$
9	11	loadAI r_{arp}, @b ⇒ r_3
12	12	add $r_3, r_1 \Rightarrow r_1$
13	15	loadAI r_{arp}, @e ⇒ r_2
16	16	add $r_2, r_1 \Rightarrow r_1$
17	19	store $r_1 \Rightarrow r_{arp}$, @a

⇓

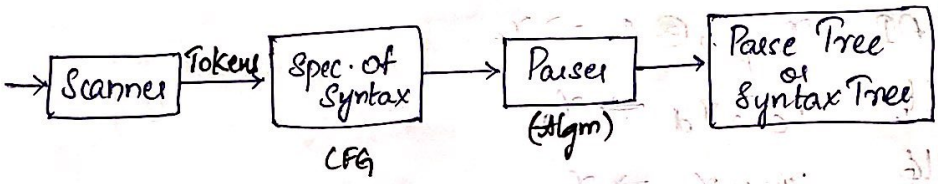
Instr Scheduling

Start	End	code
1	3	loadAI r_{arp}, @c ⇒ r_1
2	4	loadAI r_{arp}, @d ⇒ r_2
3	5	loadAI r_{arp}, @b ⇒ r_3
5	6	mult $r_1, r_2 \Rightarrow r_1$
6	8	loadAI r_{arp}, @e ⇒ r_2
7	7	add $r_1, r_3 \Rightarrow r_1$
9	9	add $r_1, r_2 \Rightarrow r_1$
10	12	storeAI $r_1 \Rightarrow r_{arp}$, @a

9/11/19

Chapter #3

** Parsers [Syntax Analyzer]:-



Parsing
Expressing Syntax → Why not RE?
CFG
Sentence / String
Production
NT Symbol
Teminal Symbol
Deviation
Sentinal form.
Parse Tree or Syntax Tree
L_m deivation
R_m derivation
Ambiguity

* Grammar:-

(1) C → PFclassidentifier XY
(2) P → public
(3) P → ε
(4) F → Final
(5) F → ε
(6) X → extends Identifier.
(7) X → ε
(8) Y → implements I

(9) $Y \rightarrow \varepsilon$

(10) $I \rightarrow$ identifier J

(11) $J \rightarrow , I$

(12) $J \rightarrow \varepsilon$

** Im derivation or Parse Tree :-

1Q) (class Car extends Vehicle
public class JavaIsCrazy
Implement factory, Builder,
listner public final class
President extends Person
implements official)

} One Sentence.

* Derive the given Sentence from above grammar?

Sol:

$C \rightarrow$ PF class identifier XY

$C \rightarrow$ F class identifier XY ($P \rightarrow \varepsilon$)

$C \rightarrow$ class identifier XY ($F \rightarrow \varepsilon$)

$C \rightarrow$ class identifier (Car) extends identifier. (Vehical)

∴ Not possible to drive complete Sentence.

2Q) Start → Paren / bracket

bracket → [Paren] / [ ]

Paren → (bracket) / ( )

Check if string [( ) is derivable ? and [( )]( ) ?

20/11/19

Parsers

Top-down Parser

I/P
(1) left to right Scanning
(2) lm derivation
(3) Parse tree is generated from top to bottom (From roots to leaves)

Bottom-up Parser

left to right Scanning
rm derivation
Parse tree is generated from bottom to top (From leaves to roots)

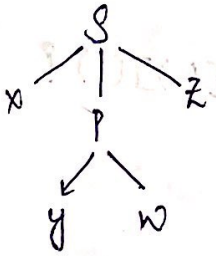
Top-down Parser

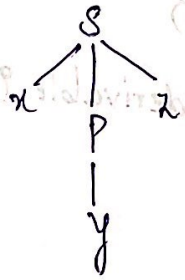
- Recursive Descent Parser [RDP]
  - Non-~~Bracketing~~ Backtracking (Back tree free)
- LL(1) Parser (or) Predictive Parser
  - Left to right
  - No. of look ahead symbols
  - works with LL(1) grammar

Problems with topdown Parser :-

(1) Backtracking :-

$S \rightarrow xPz \qquad P \rightarrow yx \mid y$

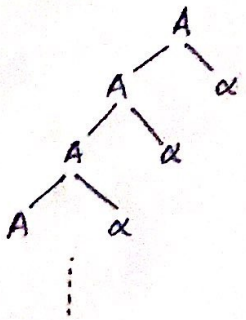




It occurs where there is more than one alternate in production to be tried while passing the input.

(2) Left Recursion :-

LR Grammar $A \overset{+}{\Longrightarrow} A\alpha$



The grammar is left recursive if it has a nonterminal A, such that there is a derivation $A \overset{+}{\Longrightarrow} A\alpha$ for some string α (anything) then this is Left recursive.

* Elimination of Left Recursion :-

If $A \to A\alpha / \beta$ by replacing this pair of productions with (X)

$$A \to \beta A'$$
$$A' \to \alpha A' / \varepsilon$$

Left Recursion type

Immediate

Indirect

Eg:- $E \to E + T / T$ (A = E, α = +T)

$T \to T * F / F$

$F \to (E) / id$

Sol:- $E \to TE'$

$E' \to +TE' / \varepsilon$

$T \to FT' / F$

$T' \to *FT' / \varepsilon$

$F \to (E) / id$.

Suppose:-

If we have multiple α & β values

$$A \to A\alpha_1 / A\alpha_2 / \cdots / A\alpha_n / \beta_1 / \beta_2 / \cdots / \beta_n \Rightarrow A \to \beta_1 A' / \beta_2 A' / \cdots / \beta_n A'$$
$$A' \to \alpha_1 A' / \alpha_2 A' / \cdots / \alpha_n A' / \varepsilon$$

22/11/19

* Problems:-

① $S \rightarrow (L) | a$

$L \rightarrow \overset{A}{L}, \overset{\alpha}{S} | \overset{\beta}{S}$

② $S \rightarrow a | \wedge | (T) | T$

$T \rightarrow T, S | a | \uparrow | (T)$

③ $S \rightarrow Ra | Aa | a$

$R \rightarrow ab$

$A \rightarrow \overset{A}{A}\overset{\alpha}{R} | \overset{A}{A}\overset{\alpha}{T} | \overset{\beta}{b}$

$T \rightarrow Tb | a$

Sol:-

(1A) $S \rightarrow (L) | a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' | \varepsilon$

(2A) $S \rightarrow a | \wedge | (T) | T$

$T \rightarrow (T)T' | aT' | \uparrow T'$

$T' \rightarrow ,ST' | \varepsilon$

(3A) $S \rightarrow Ra | Aa | a$

$R \rightarrow ab$

$A \rightarrow bA'$

$A' \rightarrow RA' | TA' | \varepsilon$

$T \rightarrow aT'$

$T' \rightarrow bT' | \varepsilon$

* Indirect Left Recursion :-

④ $S \rightarrow A$
$A \rightarrow Ba \mid a$
$B \rightarrow Ab$

Sol:- $S \rightarrow A$
$A \rightarrow Aba \mid a \mid Ba$ → Backward Substitution
$B \rightarrow Ab$

⇒ $S \rightarrow A$
$A \rightarrow Aba \mid a \mid Ba$
$B \rightarrow Bab \mid ab$

$S \rightarrow A$
$A \rightarrow Ba \mid a$ → Forward Substitution
$B \rightarrow Bab \mid ab$

⇒ $S \rightarrow A$
$A \rightarrow Ba \mid a$
$B \rightarrow abB' \mid \varepsilon$

⑤ $A \rightarrow Cd$
$B \rightarrow Ce$
$C \rightarrow A \mid B \mid f$

Sol:- $A \rightarrow Cd$
$B \rightarrow Ce$
$C \rightarrow Ce \mid Cd \mid f$

⇒ $A \rightarrow Cd$
$B \rightarrow Ce$
$C \rightarrow fC'$
$C' \rightarrow eC' \mid dC' \mid \varepsilon$.

③ $A \rightarrow AB|Aab|BA|a$
$B \rightarrow Bb|Aa|b$

Sol:- $A \rightarrow BAA'|aA'$
$A' \rightarrow BA'|abA'|\varepsilon$
$B \rightarrow aA'aB'|bB'$
$B' \rightarrow bB'|AA'aB'|\varepsilon$

④ $S \rightarrow Aa|b$
$A \rightarrow Ac|Sd|e$

⑤ $S \rightarrow AA|0$
$A \rightarrow SS|1$

Left factoring:-

The process of factoring out common prefixes of alternate

$A \rightarrow \alpha B_1|\alpha B_2$

Left factored, then the original production become

$A \rightarrow \alpha A'$
$A' \rightarrow B_1|B_2$

Eg: ① $S \rightarrow \overbrace{iEtS}^{\alpha}|\overbrace{iEtSeS}^{\alpha}|\overbrace{a}^{B_2}$
$E \rightarrow b$

Sol:- $S \rightarrow iEtSS'|a$
$S' \rightarrow eS|\varepsilon$
$E \rightarrow b$

26/11/19

LL(1) Parser / Predictive Parser :-

* Steps in constructing LL(1) Parser :

1. For a given grammar G, Eliminate left recursion if any } Make it suitable for LL(1) parser/gram
2. Perform left factoring if required
3. Compute FIRST and FOLLOW on the symbols
4. Construct the predictive parsing table
5. Check if given i/p can be parsed.

Eg:- ① $S \rightarrow (L) / a$

$L \rightarrow L,S / S$

② $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

Hw:- $S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' / \varepsilon$

FIRST :- (Def)

FIRST of α is a set of terminal symbols that of 1st symbol appearing at RHS in derivation of α and if $\alpha \overset{*}{\Rightarrow} \varepsilon$ then ε is also in FIRST(α) = {ε}.

RULES :- $X \rightarrow a\alpha$. (computing FIRST)

① If X is NT

FIRST(X) = a

② If X is T

FIRST(X) = {X}

③ If X → ε is production

FIRST(X) = {ε}

④ If X → $Y_1 Y_2 \cdots Y_k$ and Y_1 is Nullable

FIRST(X) = FIRST(Y_2)

If Y_2 is also Nullable

FIRST(X) = FIRST(Y_3)

*⇒ FIRST for above eg①.

	FIRST
S	(a
L	(a
L'	,ε

⇒ eg② FIRST

	FIRST
E	(id
E'	+ε
T	(id
T'	*ε
F	(id

③ S → Bb | Cd

B → aB | ε

C → cC | ε

Sol:-

	FIRST
S	a, ε, c, b, d
B	a, ε
C	c, ε

④ $S \to ACB / CbB / Ba$

$A \to da / BC$

$B \to g / \varepsilon$

$C \to h / \varepsilon$

Sol:-

	FIRST
S	d, g, ε, h, b, a
A	d, g, ε, h
B	g, ε
C	h, ε

Computation of FOLLOW :- (only NT's)

Def:-

FOLLOW(A) is defined as set of terminal symbols that appear immediately to the right of A

$FOLLOW(A) = \{a / S^* \Rightarrow \alpha A a \beta\}$ where α & β are some grammar symbols (T or NT or ε)

* Computing Rules :-

① FOLLOW(S) = $ if S is a start symbol of grammar.

② If you have prod $A \to \alpha B \beta$, FOLLOW(B) = FIRST(β) except ε

③ $A \to \alpha B$, FOLLOW(B) = FOLLOW(A)

④ $A \to \alpha B \beta$ where FIRST(β) contains ε, FOLLOW(B) = FOLLOW(A)

Eg:- ① $S \to (L)$

② $S \to a$

③ $L \to SL'$

④ $L' \to ,SL'$

⑤ $L' \to \varepsilon$

Sol:- ① $S \rightarrow (L)$

rule 1 ⇒ S is start symbol

rule 2 ⇒ $S \rightarrow \alpha B \beta$ (β → L)

3 X

4 → same as 2.

③ ⇒ $S \rightarrow L \rightarrow \alpha B \beta$ → $\varepsilon S L'$

rule 3 → $L \xrightarrow{\alpha B} S L'$

$FOLLOW(L') = FOLLOW(L)$

rule 4 → $S \xrightarrow{\alpha B \beta} L$

$FIRST(\beta) \rightarrow$ contains $\varepsilon \rightarrow$ true

then $FOLLOW(S) = FOLLOW(L)$.

④ ⇒ $L' \rightarrow , S L'$

rule 2 ⇒ $L' \xrightarrow{\alpha B \beta} , S L'$

rule 3 ⇒ $L' \xrightarrow{A \; \alpha \; B} , S L'$

$FOLLOW(L') = FOLLOW(L')$

rule 4 ⇒ $L' \xrightarrow{\alpha B \beta} , S L'$

$F(\beta) \rightarrow$ contains ε. then

$FOLLOW(\$) = FOLLOW(L')$

②, ⑤ $L' \rightarrow \varepsilon$ → No rule to apply

	FOLLOW
S	$,)
L	)
L'	)

⇒ Final table.

	FOLLOW
S	$, FIRST(L')−ε, FOLLOW(L)
L	FIRST( ) − ε
L'	FOLLOW(L), FOLLOW(L')

27/11/19

Steps/Rules to be followed to make entry in the LL(1) table:-

Step1:- For each prod $A \to \alpha$ of G · do Step 2 and Step 3.

Step2:- For each terminal a in FIRST(α) add $A \to \alpha$ to T[A,a]

Step3:- (a) If ε in FIRST(α) add $A \to \alpha$ to T[A, b] for each terminal b in FOLLOW(A)

(b) If ε in FIRST(α) & \$ in FOLLOW(A) then add $A \to \alpha$ to T[A,\$]

Step4:- Each undefined entry = Error.

Eg:- $S \to (L)$

$S \to a$

$L \to SL'$

$L' \to ,SL'$

$L' \to \varepsilon$

Sol:-

	FIRST	FOLLOW
S	(a	\$,)
L	(a	)
L'	,ε	)

LL(1) Table

	a	,	(	)	\$
S	S→a	Error	S→(L)	Error	Error
L	L→SL'	Error	L→SL'	Error	Error
L'	Error	L'→,SL'	Error	L'→ε	Error

(N) Empty cells are Errors

① $S \to (L)$

FIRST(α) = (

$\Rightarrow T[S, (]$

② $S \to a$

$\Rightarrow T[S, a]$

③ $L \to \overbrace{\cdot S L'}^{\alpha}$

FIRST(α) = S

FIRST(S) = (a

$\Rightarrow$ T[L, a] & T[L, (]

④ $L' \to \overbrace{\cdot , S L'}^{\alpha}$

FIRST(α) = ,

$\Rightarrow$ T[L', ,]

⑤ $L' \to \varepsilon$

If α is ε then add $L' \to \varepsilon$ to T[L', b]

where b = FOLLOW(L') =)

$\Rightarrow$ T[L',)].

Stack	I/p	Action.
$$	(a,a)$	$S \to \overbrace{(L)}$ Push RHS in reverse
$)L(	(a,a)$	
$)L	a,a)$	$L \to SL'$ Push RHS in reverse
$)L'S	a,a)$	$S \to a$ pop
$)L'a	a,a)$	
$)L'	,a)$	$L' \to ,SL'$ push RHS in reverse
$)L'S,	a)$	
$)L'S	a)$	$S \to a$ pop
$)L'a	)$	
$)L'	)$	$L' \to \varepsilon$ pop
$)	$	
$	$	Stop //

28/11/19 (10 Marks)

** Recursive Descent Parser [RDP] :-

Dfn:- A parser that uses a set of recursive procedures to recognize its input with no back tracking is called a RDP. Each NT ⇒ one recursive parser.

Eg:-

Q. Write RDP for grammar given below assume that token is a global variable that contains next input token and a function advance reads the next token and fn error is called if input token is not valid. and $ is end marker.

$E \rightarrow \epsilon + id \mid id$

Sol:- remove LR ⇒ $\epsilon \rightarrow id\epsilon'$

$\epsilon' \rightarrow +id\epsilon' \mid \varepsilon$

⇒ Recursive procedure.

```
→ main()
  {
     E();
     if (token == $)
     {
          print("Parsing Success");
     }
     else {
          Error();
     }.
  }
  Error()
  {
     print("parsing failure");
  }
  Advance()
  {
     token = getnexttoken();
  }
```

```
E( )
{
    if (token=='id')
    {
        Advance( );
        EDASH( );
    }
}
EDASH( )
{
    if (token == '+')
    {
        Advance( );
        if (token == 'id')
        {
            Advance( );
            EDASH( );
        }
    }
    else {
        return;
    }
```

Ⓧ Sample I/p ⇒ | id | + | id | $ |