FIRST TERM EXAMINATION [FEB. 2016]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 1.5 hrs. **M.M. : 30**

Note: *Attempt any three questions including Q. no. 1 which is compulsory.*

Q.1. (a) Discuss Merits and Demerits of Single pass compiler and Multi pass compiler.

Ans: Single pass Compiler is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code.

Merits:

1. They are faster.
2. It is also called as a Narrow compiler.
3. The external storage for the intermediate file between two passes is slow or is inconvenient to use.

Demerits:

1. It has less efficient code optimization and code generation.
2. Large Memory is required by compiler.
3. It has limited scope of passes.

Multipass Compiler: It reads the program several times. Each time transforming it in to different form.

Merits:

1. Better code optimization and code generation.
2. It is called wide compiler as they scan each and every portion of program.

Demerits:

1. Slower, as no. of passes means more execution time.

Q. 1. (b) What do you mean by Bootstrapping in Compilation?

Ans: Bootstrapping is the process of writing a compiler (or assembler) in the source programming language that it intends to compile. Applying this technique leads to a self-hosting compiler. An initial minimal core version of the compiler is generated in a different language; from that point, successive expanded versions of the compiler are run using the minimal core of the language.

Bootstrapping a compiler has the following advantages:

- It is a non-trivial test of the language being compiled, and as such is a form of dogfooding.
- Compiler developers only need to know the language being compiled.
- Compiler development can be done in the higher level language being compiled.
- Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself.
- It is a comprehensive consistency check as it should be able to reproduce its own object code.

1.Q. (c) What is the need for separating the Parser from Scanner?

Ans: Lexer: character stream to token stream

Parser: token stream to syntax tree

Advantages:

Simpler design: Based on related but distinct theoretical underpin
Compartmentalizes some low-level issues, e.g., I/O, internationalization.

Faster

Lexing is time-consuming in many compilers (40-60% ?)

By restricting the job of the lexer, a faster implementation is usually feasib

Q.1. (*d*) Find left most derivation, right most derivation, and derivation of a string baababa.

{S → bB | aA, A→b | bS | aAA, B→ a | aS | bBB

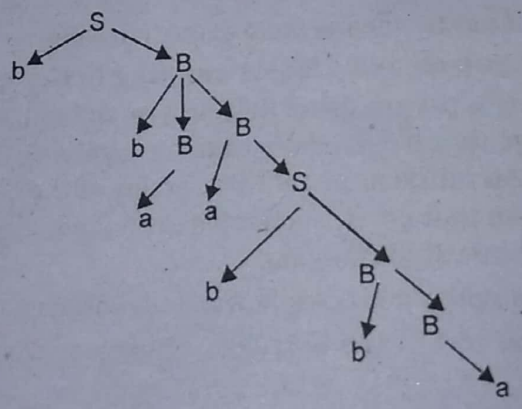
Ans: Left Most Derivation:

S –> bB
S –> bbBB
S –> bbaB
S –> bbaaS
S –> bbaabB
S –> bbaabaS
S –> bbaabab B
S –> bb aababa

Right Most Derivation

S –> bB
S –> bbBB
S –> bbBaS
S –> bbBabS
S –> bbBabaS
S –> bbBababB
S –> bbBababa
S –> bbaababa

Derivation tree



Q.1. (*e*) Explain handle pruning with an example.

Ans: A right most derivation in reverse can be obtained by *handle pruning*.

i.e., start with a string of terminals w that is to parse. If w is a sentence grammar at hand, then $w = \gamma_n$, where γ_n is the nth right sentential form of some unknown rightmost derivations γ_{n-1} $\gamma_n = w$.

Example for right sentential form and handle for grammar

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

Right sentential form	Handle	Reduction production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Q.1. (f) What are the responsibilities of Loader, Linker, and Assembler in the compiler environment?

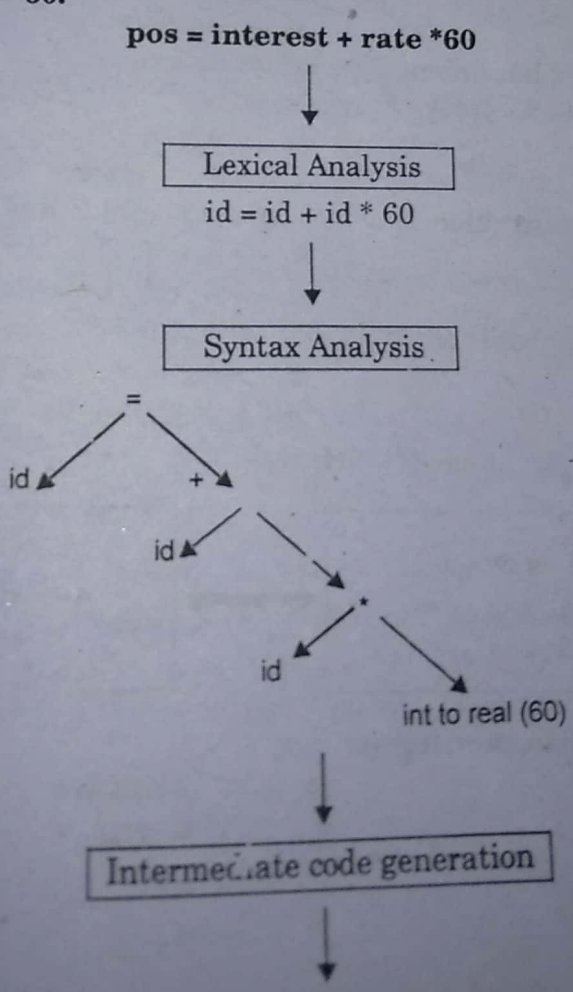
Ans: Assembler: An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker: Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/ routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader: Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

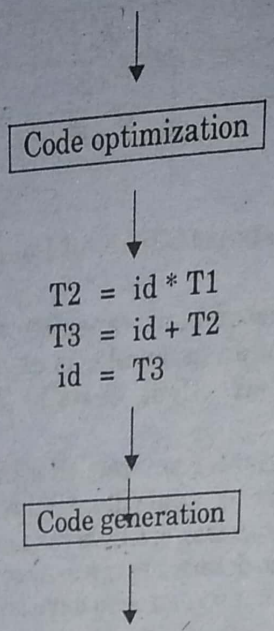
Q.1. (g) Write steps involved in compilation of following sentence pos = interest + rate * 60.

Ans:



4–2016

T1 = in to real (20)
T2 = id * T1
T3 = id + T2
id = T3



Q2. Attempt both parts

(a) Remove left recursion from the following grammar:

$$S \rightarrow Aa/b$$
$$A \rightarrow Ac/Sd/e$$

Ans: As we don't have immediate left recursion,

Step (1) Rename S, A as A_1, A_2 respectively

$$A_1 \text{ -> } A_2a \mid b$$
$$A_2 \text{ -> } A_2\ c \mid A_1\ d \mid e$$

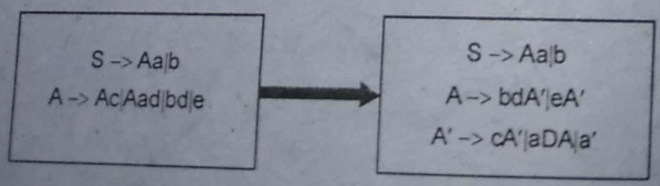
Step (2) Substitute value A_1 of statement (1) in R.H.S of statement (2)

$$A_1 \text{ -> } A_2a \mid b$$
$$A_2 \text{ -> } A_2\ c \mid A_2\ a\ d \mid bd \mid e$$

Step (3) Again substitute $A_1 = S, A_2 = A$

$$S \text{ -> } Aa \mid b$$
$$A \text{ -> } Ac \mid Aad \mid bd \mid e$$

Step (4) It has now immediate left recursion



Q.2. (b) Prove whether following:

$$S \longrightarrow XS \mid dS \mid \epsilon$$
$$X \longrightarrow Y \mid Zb \mid aY$$
$$Y \longrightarrow cZ$$
$$Z \longrightarrow e$$

Ans: A Grammar is LL(1) if for every production A $\alpha \mid \beta \longrightarrow$

First (α) ^ First (β) = ϕ

If $\lambda \in$ First (α) then First (β) ^ Follow (A) = ϕ

If $\lambda \in$ First (β) then First (α) ^ Follow (A) = ϕ

First(S) = First(X) –> First(Y) –> First(Z) –> e

First(X) = First(Y) –> First(Z) –> e

First(Y) = First(Z) –> e

First(Z) = e

First (S) = {e} ∪ {d} ∪ {ϵ}

First (X) = {c} ∪ {e} ∪ {a}

This Grammar is LL(1)

Q.3. Write Algorithm to construct SLR parser. Show all the steps of SLR parsing for following grammar.

$$S \rightarrow R$$
$$R \rightarrow Rb$$
$$R \rightarrow a$$

Ans: Simple LR or SLR parser is a type of LR parser with small parse tables and a relatively simple parser generator algorithm. As with other types of LR(1) parser, an SLR parser is quite efficient at finding the single correct bottom-up parse in a single left-to-right scan over the input stream, or backtracking. The parser is mechanically generated from a formal grammar for the language.

Algorithm:

Input: An Augmented Grammar G'

Output: goto

Method:

1. Initially construct set of items

C = {I_0, I_1, I_2, ... In} where C is a collection of LR(0) items for grammar

2. Parsing actions are based on each item or state I_i.

Various Actions are:

(a) If $\rightarrow \alpha.a\beta$ is in I_i and goto $(I_i, a) = I_j$ then set Action [i, a] = "shift j".

(b) If $A \rightarrow \alpha.$ is in I_i then set Action[i , a] to reduce A –> α for all symbols a, where a belongs to Follow(A).

(c) If $S \rightarrow S.$ is in I_i then entry in action table Action [i , $] = "accept"

3. The go to part of SLR table can be filled as: The go to transitions for state i is considered for non termialsonly. If goto$(I_i, a) = I_j$ then goto [i ,A] = j.

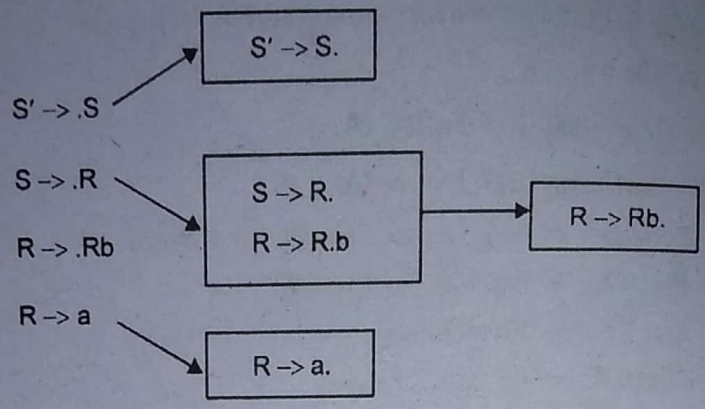
4. All entries not defined by rule 2 and 3 are considered to be "error".

S R →

RRb →

Ra →

Item set 10



Q.4. Attempt both parts

(a) Draw the finite automata which can be used by lexical analyser to recognize identifier, number and operators.

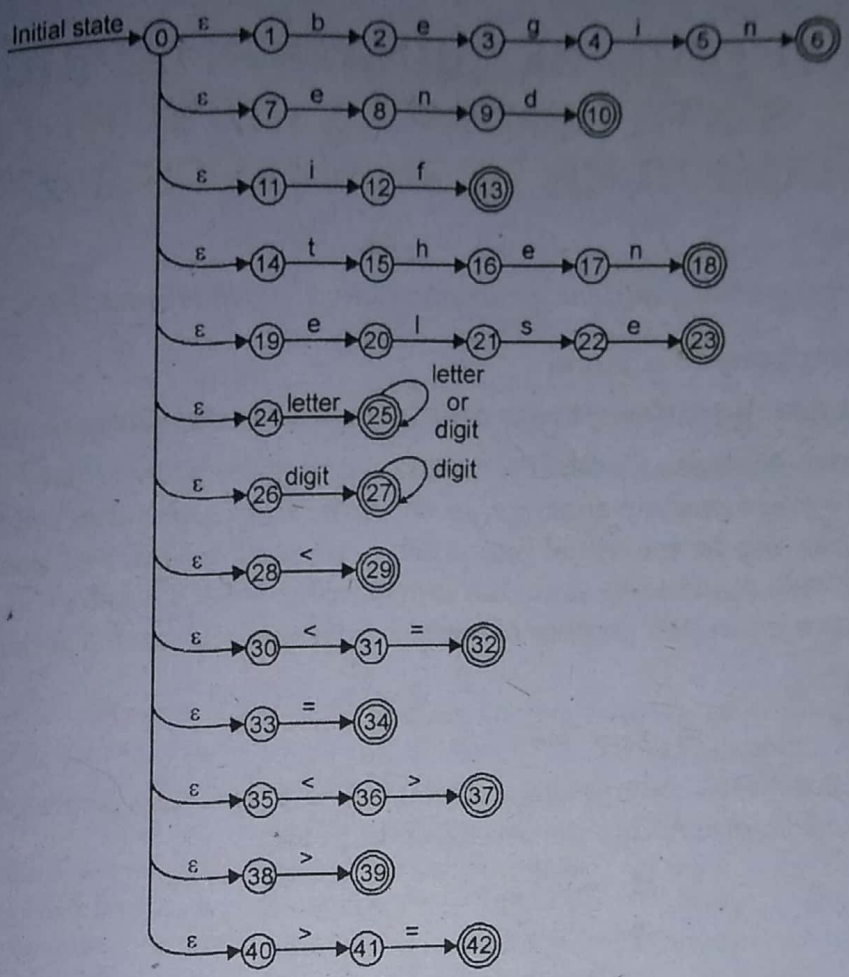
Ans: **Auxiliary definations**

letter = A|B|C|.....|Z

digit = 0|1|2|.....|9

Translation rules

begin	{ return 1}
end	{return 2}
if	{return 3}
then	{return 4}
else	{return 5}
letter (letter + digit)⁻	{Value = install ( ): return 6}
digit⁺	{Value = install ( ): return 7}
<	{Value = 1 : return 8}
<=	{Value = 2 : return 8}
=	{Value = 3 : return 8}
<>	{Value = 4 : return 8}
>	{Value = 5 : return 8}
>=	{Value = 6 : return 8}



Q.4. (*b*) What is operator grammar? Write operator precedence parsing table for following grammar:

$$E \rightarrow E + E|E*E|(E)|a$$

Ans: An operator precedence grammar is a context-free grammar that has the property that no production has either an empty right-hand side or two adjacent non terminals in its right-hand side. These properties allow precedence relations to be defined between the terminals of the grammar. A parser that exploits these relations is considerably simpler than more general-purpose parsers such as LALR parsers. Operator-precedence parsers can be constructed for a large class of context-free grammars.

First attach $ at starting and ending of string i.e. $ a1 + a2 + a3$.

Put precedence relation between operators & symbos using precedence relation table

$$\$ \rightarrow \cdot a1 <\cdot + <\cdot a2 \cdot> * <\cdot a3 \cdot> \$$$

String	Handle	Production Used
$<·a1·> + <·a2·> *<·a3·>$	<·a1·>	E –> a
$E + <·a2·> *<·a3·>$	<·a2·>	E –> a
$E + E *<·a3·>$	<·a3·>	E –> a
$E + E * E$	Remove all non terminals	
$+*$	Insert Precedence relation between operators	
$<·+<·*·>$	<·*·> i.e. E*E	E –> E * E
$<· +·>$	<·+·> i.e. E+E	E –> E + E
$		

FIRST TERM EXAMINATION [FEB. 2017]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 1.5 Hrs. **M.M. : 30**

Note: *Attempt three questions. Question No. 1. is compulsory. Each question carries 10 marks.*

Q.1. (a) Mention the role of Lexical Analyzer in compiler Design. **(2)**

Ans. The main task is to read the input characters and produce as output sequence of tokens that the parser uses for syntax analysis.

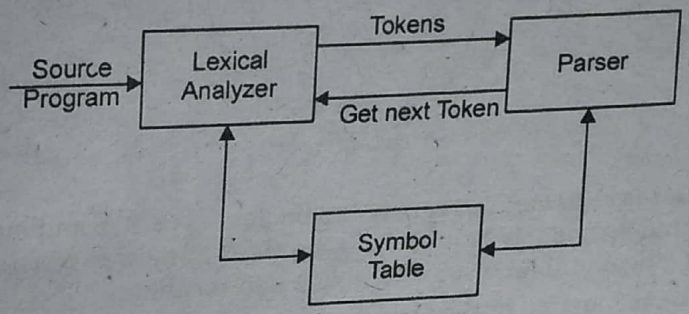


Fig. Role of the lexical analyzer diagram

Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Its secondary tasks are:

- One task is stripping out from the source program comments and white space is in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.

Sometimes lexical analyzer is divided in to cascade of two phases.

(1) Scanning (2) Lexical analysis.

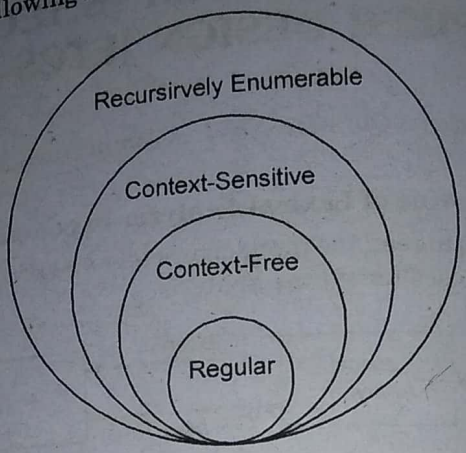
The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Q.1. (b) Explain Chomsky classification of languages. **(2)**

Ans. According to Noam Chomosky, there are four types of grammars " Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other –

Grammar Type	Grammar Accepted	Language Accepted	Automaton
1. Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
2. Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
3. Type 2	Context-free grammar	Context-free language	Pushdown automaton
4. Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of grammar



Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X →a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

X→ε

X → a | aY

Y→ b

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A →** γ

where **A ∈ N** (Non terminal)

and γ ∈ **(T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

S → X a

X→ a

X → aX

X → abc

X→ε

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings α and β

The rule **S → ε** is all

languages generated by th

Example

AB → AbBc

A → bcA

B →b

Type - 0 Grammar

Type-0 grammars g

have no restrictions. The

grammars.

They generate the lang

The productions can b

nonterminals with at leas

terminals and non-termina

Example

S → ACaB

Bc → acB

CB→ DB

aD → Db

Q.1. (c) What are the

method is the most pow

Ans. There are three

parser:

- SLR(1) - Simple LR
  - o Works on smallest
  - o Few number of sta
  - o Simple and fast co
- LR(1) - LR parser
  - o Also called as Can
  - o Works on complete
  - o Generates large ta
  - o Slow construction.
- LALR(1) - Look ahea
  - o Works on interme
  - o Number of states

Q.1. (d) An ambiguo

with an example.

Ans: An ambiguous gra

aren't designed to handle a

problem, if employed upon

is.

Q.1. (e) Design the N

Ans.

The strings α and β may be empty, but γ must be non-empty.

The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

$AB \rightarrow AbBc$

$A \rightarrow bcA$

$B \rightarrow b$

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of $\alpha \rightarrow \beta$ where α is a string of terminals and nonterminals with at least one non-terminal and α cannot be null. β is a string of terminals and non-terminals.

Example

$S \rightarrow ACaB$

$Bc \rightarrow acB$

$CB \rightarrow DB$

$aD \rightarrow Db$

Q.1. (c) What are the three methods to construct LR Parsing tables.Which method is the most powerful? Justify. **(2)**

Ans. There are three widely used algorithms available for constructing an LR parser:

- SLR(1) - Simple LR
  - o Works on smallest class of grammar.
  - o Few number of states, hence very small table.
  - o Simple and fast construction.
- LR(1) - LR parser
  - o Also called as Canonical LR parser.
  - o Works on complete set of LR(1) Grammar.
  - o Generates large table and large number of states.
  - o Slow construction.
- LALR(1) - Look ahead LR parser
  - o Works on intermediate size of grammar.
  - o Number of states are same as in SLR(1).

Q.1. (d) An ambiguous grammar can never be LR .Justify the statement with an example. **(2)**

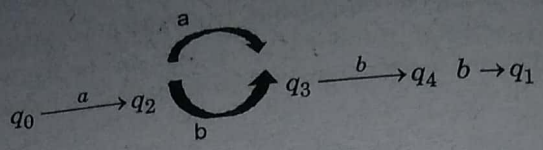
Ans: An ambiguous grammar can never be LR(k) for any k, because LR(k) algorithm aren't designed to handle ambiguous grammars. It would get stuck into undecidability problem, if employed upon an ambiguous grammar, no matter how large the constant k is.

Q.1. (e) Design the NFA for the regular Expression a (a+b)* bb **(2)**

Ans.

$$q_0 \underline{a(a+b)^* bb}\ q_1$$

$$q_0 \underline{a}\ q_{2(a+b)}{}^* q_3 \underline{b}\ q_4 \underline{b}\ q_1$$



Q.2. (a) Let G be CFG: (5)

$$S \rightarrow bB \mid aA$$
$$A \rightarrow b \mid bS \mid aAA$$
$$B \rightarrow a \mid aS \mid bBB$$

For the string bbaababa find
(i) Left Most Derivative
(ii) RightMost Derivative
(iii) Parse Tree

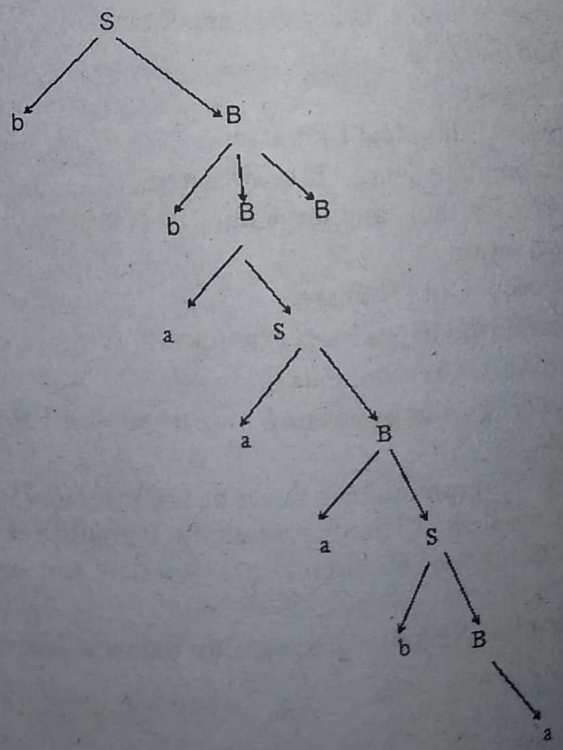
Ans. (i) Left Most Derivative:

$S \rightarrow bB$
$\rightarrow bbBB$
$\rightarrow bbaSB$
$\rightarrow bbaaB$
$\rightarrow bbaabBB$
$\rightarrow bbaabaSB$
$\rightarrow bbaababa$

(ii) RightMost Derivative:

$S \rightarrow bB$
$\rightarrow bbBB$
$\rightarrow bbBa$
$\rightarrow bbaSa$
$\rightarrow bbaaAa$
$\rightarrow bbaabSa$
$\rightarrow bbaabaAa$
$\rightarrow bbaababa$

(iii) Parse tree



Q.2. (b) Construct predictive parsing table for the following grammar(S is Start Symbol) **(5)**

$$S \rightarrow iCtSS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$C \rightarrow b$$

Ans. Step 1. Eliminate left-Recursion and Left Factor the grammer if required.
As, there is no left. Recursion and grammar is already left Factored.
Step 2. Computation of FIRST

(i) $S \rightarrow iCtSS' \mid a$

$\therefore$ FIRST (S) = {i, a}

(ii) $S' \rightarrow eS \mid \varepsilon$

$\therefore$ FIRST (S') = {e. ε)

(iii) $C \rightarrow b$

$\therefore$ FIRST (C) = {b}

FIRST (S) = {i, a}
FIRST (S')= {e, ε}
FIRST (C) = {b}

Step 3. Computation of FOLLOW:
Applying Rule (1) of FOLLOW

FOLLOW (S) = {$}

(1) $S \rightarrow iCtSS'$

Applying Rule (2) of FOLLOW.

$S \rightarrow$	i	C	tSS'
$A \rightarrow$	α	B	β

$\therefore$ FIRST (β) = FIRST (t SS') = {t}.
$\therefore$ FIRST (β) does not contain ε.
$\therefore$By Rule (2a) FOLLOW (C) = {FIRST (t SS')}

$\therefore$FOLLOW (C) = [t]

Again, Apply Rule (2) on different combination

$S \rightarrow$	iCt	S	S'
$A \rightarrow$	α	B	β

FIRST (β) = FIRST (S') = {e, ε} contain ε.
By Rule (2b) FOLLOWS (S) = FIRST (S') – {ε} ∪ FOLLOW (S)
FOLLOW (S) = {e, ε} – {ε} ∪ FOLLOWS (S)

FOLLOW (S) = {e} ∪ FOLLOW (S)

Applying Rule (3) of Follow

$S \rightarrow$	$iCtS$	S'
$A \rightarrow$	α	B

A = S, α = i C t S, B = S'

$\therefore$ FOLLOW (S') = {FOLLOW (S)}

$S' \rightarrow eS$

(2)
Rule (2) can not be applied on this Production.
Applying Rule (3) of FOLLOW

$S' \rightarrow$	e	S
$A \rightarrow$	α	B

FOLLOW (S) = {FOLLOW (S')}

The Renaming Production $S \rightarrow a$, $S' \rightarrow \varepsilon$ and $C \rightarrow b$ do not match with an Rule
Combining Statements (1) to (5)

FOLLOWS (S) = {$}
FOLLOWS (C) = {t}
FOLLOW (S) = {e} $\cup$ FOLLOW (S)
FOLLOW (S') = {FOLLOW (S)}
FOLLOW (S) = {FOLLOW (S')}

$\therefore$ FOLLOW (S) = FOLLOW (S)={$; e}
FOLLOW (C) = {t}

Step 4: **Construction of Predictive Parsing Table.** Put all Non- terminals S'C'S Row wise and All terminals. i, a, b, e, $, t Column wise

(1) $S \rightarrow i\ C\ t\ S\ S'$

Comparing $S \rightarrow i\ C\ t\ S\ S'$ with $A \rightarrow \alpha$

$S \rightarrow$	$iCtSS'$
$A \rightarrow$	α

FIRST (α) = FIRST (i C t S S') = {i}

Applying Rule (1) of predictive Parsing Table:-
Add $S \rightarrow iCtSS'$ to M[S, i]

Write $S \rightarrow$ i C t S S' in front of Row (S) and Column (i) ...(1)

(2) $S \rightarrow a$
$A \rightarrow \alpha$

Comparing it with

$S \rightarrow$	a
$A \rightarrow$	α

$A = S, \alpha = a$

$\therefore$ FIRST (α) = FIRST (a) = {a}

$\therefore$ Put $S \rightarrow a$ onto M[S, a]

Write $S \rightarrow$ a in front of Row (s), column (a)

(3) $S' \rightarrow eS$

Compairing it with $A \rightarrow \alpha$

$S' \rightarrow$	eS
$A \rightarrow$	α

$\therefore$
$\therefore$
$\therefore$

(4) $S' \rightarrow \varepsilon$
Comparing it with

$\therefore$
$\therefore$
$\therefore$ Applying Rule (2
Then FOLLOW
$\therefore$ Add

wr

(5)
Comparing it with

$\therefore$
$\therefore$ Add $C \rightarrow b$ to M [

w

$\therefore$ Combining staten

	a	b
S	$S \rightarrow a$	
S'		
C		C –

This Grammer is not
Entries for productions

$S' \rightarrow eS$ appear
$S' \rightarrow \varepsilon$ appears a
This Grammer is not

Q.3. (a) Construct S

Ans: Step 1: Constru

$A = S', \alpha = eS$

$\therefore$ FIRST (α) = FIRST (eS) = {e}

$\therefore$ Add S' $\rightarrow$ eS to M [S', e]

$\therefore$ Write S' $\rightarrow$ eS in front of Row (S'), column (e)

(4) S' $\rightarrow \varepsilon$

Comparing it with $A \rightarrow \alpha$

S' $\rightarrow$	ε
A $\rightarrow$	α

$A = S', \alpha = \varepsilon$

$\therefore$ FIRST (α) = FIRST (ε) = {ε}

$\therefore$ Applying Rule (2) of Predictive Parsing Table. i.e. If ε is in FIRST (α)

Then FOLLOW (S') = {e, $}

$\therefore$ Add S' $\rightarrow \varepsilon$ to M [S', e] M [S,' $]

$\therefore$ write S' $\rightarrow \varepsilon$ in front of Row (S') Column {(e, $)}

(5) $C \rightarrow b$

Comparing it with $A \rightarrow \alpha$

C $\rightarrow$	b
A $\rightarrow$	α

$\therefore$ First (α) = FIRST (b) = {b}

$\therefore$ Add C $\rightarrow$ b to M [C, b]

$\therefore$ write C $\rightarrow$ b in front of Row (C), Column (b)

$\therefore$ Combining statements (1) to (5) we get following Predictive Parsing table.

	a	b	e	i	t	$
S	S $\rightarrow$ a			S $\rightarrow$ i C t S S'		
S'			S' $\rightarrow$ eS S' $\rightarrow \varepsilon$			S' $\rightarrow \varepsilon$
C		C $\rightarrow$ b				

This Grammer is not LL (1) because In Row (S') and column (e) we have multiple Entries for productions ...(1)

S' $\rightarrow$ eS

S' $\rightarrow \varepsilon$

S' $\rightarrow$ eS appears as FIRST (S') = {e}

S' $\rightarrow \varepsilon$ appears as FOLLOW (S') = {ε}

This Grammer is not LL (1) Grammer.

Q.3. (a) Construct SLR parsing Parsing table for the following grammar: **(5)**

S $\rightarrow$ xAy | xBy | xAz

A $\rightarrow$ aS | q

B $\rightarrow$ q

Ans: Step 1: Construct augmented Grammar

(0) S' $\rightarrow$ S

(1) S $\rightarrow$ **xAy**

(2) S $\rightarrow$ **xBy**

(3) S $\rightarrow$ **xAz**

(4) A $\rightarrow$ aS

(5) B $\rightarrow$ q

Step 2. Find Closure & goto functions to construct LR (0) items. Here Boxes represent New States and Circles represent the repeating state.

Closure (S' → • S)

Io:
S' → •S
S → •x A y
S → •x B y
S → •x A z

I4 = goto (I2,B)
S → x B • y

I8 = goto (I4,y)
S → x By •

I5: goto (I2,q)
A → q • S
A → q •
B → q •
S → •x A y
S → •x B y
S → •x A z

I1 = goto (I0,S)
S' → S •

I9 = goto (I5,S)
A → qS•

I2 = goto (I0, x)
S → x • Ay
S → x • By
S → x • Az
A → • qS
A → • q
B → • q

I2 = goto (I5, x)
S → x • Ay
S → x • By
S → • Az
A → • q
B → • q

I6 = goto (I5,y)
S → x A y •

I3 = goto(I2, A)
S → X A.Y
S → X A.Z

I7 = goto (I3,z)
S → x A z •

Step 3. Computation of FOLLOW

1. S → x A y

Follow (S) = {$}

Applying Rules (2a) of FOLLOW

(Comparing S → x a y with A → α B β

∴ FIRST (β) = FIRST (y) = {y}

∴ FOLLOW (A) = {y}

Rule (3) cannot be applied.

As, S → x A y cannot be compared with A → α B

2. S → x B Y

Applying Rule (2a) of FOLLOW

comparing S → x By with A → α B β

∴ FIRST (β) = {y}

Rule (3) ca
3. S → x A y
Applying R

Rule (3) can
4. A → qS
Rule (2a) ca
Appling Rule
Comparing A
∴

Rule (2a) and
q. Combining stat

Step 4: Constr

States	x	
0	s2	
1		
2		
3		
4		
5	s2	
6		
7		
8		
9		

Q.3. (b) Discuss the grammar ? State

Ans. If a context w∈ L(G), it is called a most derivations for s

Problem

To check whether X → X+X | X*X | X

$\therefore$ FOLLOW(B) = {y}

Rule (3) cannot be applied.

3. $S \rightarrow xAy$

Applying Rule (2a) of Follow

First (β) = {z}

$\therefore$ FOLLOW(A) = {z}

Rule (3) cannot be applied.

4. $A \rightarrow qS$

Rule (2a) cannot be applied. As $A \rightarrow qS$ cannot be compared with $A \rightarrow \alpha B \beta$

Appling Rule (3)

Comparing A – qS with $A \rightarrow \alpha\beta$

$\therefore$ $A = A, \alpha = q, B = S$

$\therefore$ FOLLOW (S) = {FOLLOW (A)}

Rule (2a) and Rule (3) of FOLLOW cannot be applied on production $A \rightarrow q$ and $B \rightarrow$

Combining statements 1 to 5

FOLLOW (A) = {y, z}
FOLLOW (s) = {$, y, z}
FOLLW (B) = {y}

Step 4: Construction of SLR (1) Parsing Table:

tates	Action					goto		
	x	y	z	q	$	S	A	B
	s2	r3/r4				1		
					accept			
				s5			3	4
		s6	s7					
		58						
	s2	r5/r6	r5			9		
		r1	r1		r1			
		r3	r3		r3			
		r2	r2		r2			
		r4	r4					

Q.3. (b) Discuss ambiguity of a CFG.How Ambiguity can be removed from grammar ? State suitable example. (5)

Ans. If a context free grammar G has more than one derivation tree for string w L(G), it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem

To check whether the grammar G with production rules–

$X \rightarrow X+X \mid X^*X \mid X^*a$ is ambiguous or not.

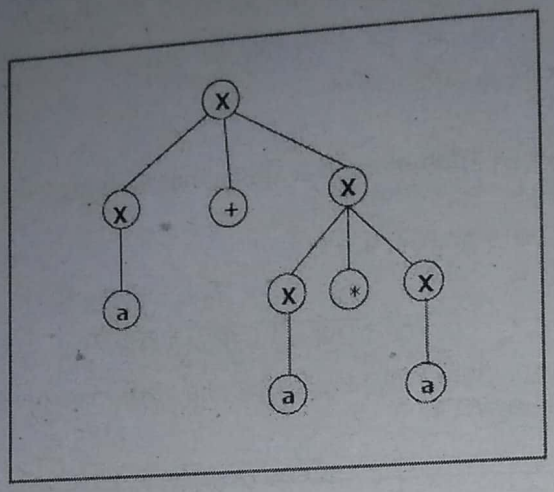
Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

Derivation 1 –

$X \rightarrow X+X \rightarrow a+X \rightarrow a+X*X \rightarrow a+a*X \rightarrow a+a*a$

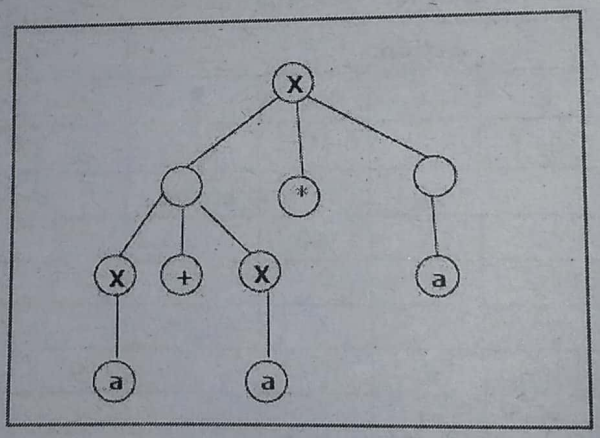
Parse tree 1 –



Derivation 2–

$X \bullet \rightarrow X*X \bullet \rightarrow X+X*X \bullet \rightarrow a+X*X \bullet \rightarrow a+a*X \rightarrow a+a*a$

Parse tree 2 –



As there are two parse trees for a single string "a+a*a", the grammar G is ambiguous.

Removal of Ambiguity:

1. Precedence of Operators is not respected.

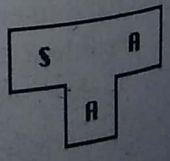
2.Sequence of identical operators can group either from left or from right .We would see two different parse tree for the above expression. Since addition is associative, it doesn't matter w

Q.4.Write short notes on following:

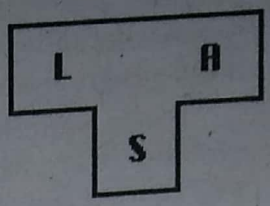
(a) Bootstrapping and Cross Compiler (2.5)

Ans: A compiler is characterized by three languages:

1. Source Language
2. Target Language
3. Implementation Language .



Notation: ${}^{S}C_{I}^{T}$ represents a compiler for Source S, Target T, mented in I. The *T-diagram* shown above is also used to t the same compiler.



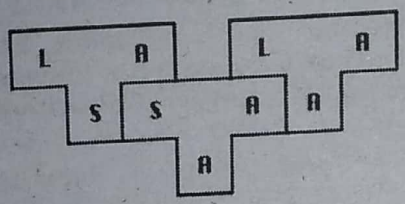
To create a new language, L, for machine A:

Create ${}^{S}C_{A}^{A}$, a compiler for a subset, S, of the desired language, L, using language hich runs on machine A. (Language A may be assembly language.)

Create ${}^{L}C_{S}^{A}$, a compiler for language L written in a subset of L.

Compile ${}^{L}C_{S}^{A}$ using ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$, a compiler for language L, which runs achine A and produces code for machine A.

$$ {}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A} $$



he process illustrated by the T-diagrams is called *bootstrapping* and can be arized by the equation:

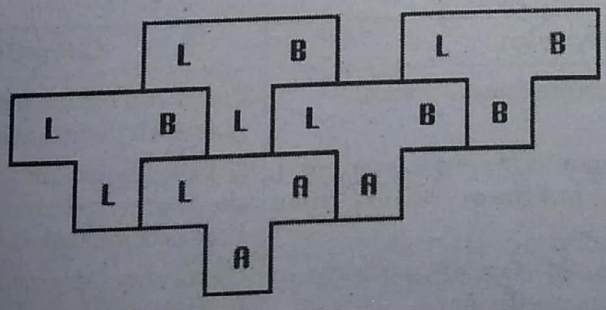
$$L_S\,\hat{}\,+S_A\,\hat{}\,=L_A\,\hat{}$$

produce a compiler for a different machine B:

Convert ${}^{L}C_{S}^{A}$ into ${}^{L}C_{L}^{B}$ (by hand, if necessary). Recall that language S is a subset guage L.

Compile ${}^{L}C_{L}^{B}$ to produce ${}^{L}C_{A}^{B}$, a *cross-compiler* for L which runs on machine A oduces code for machine B.

Compile ${}^{L}C_{L}^{B}$ with the cross-compiler to produce ${}^{L}C_{B}^{B}$, a compiler for language ch runs on machine B.



Cross Compiler is a compiler capable of creating executable code for a platform other the one on which the compiler is running. For example, a compiler that runs on dows 7 PC but generates code that runs on Android smartphone is a cross compiler.

A cross compiler is necessary to compile for multiple platforms from one machine. atform could be infeasible for a compiler to run on, such as for the microcontroller of mbedded system because those systems contain no operating system.

In paravirtualizationone machine runs many operating systems, and a cross compiler could generate an executable for each of them from one main source.

Cross compilers are not to be confused with source-to-source compilers. A cross compiler is for cross-platform software development of binary code, while a source-to-source compiler translates from one programming language to another in text code. Both are programming tools.

Uses of cross compilers

The fundamental use of a cross compiler is to separate the build environment from target environment. This is useful in a number of situations:

- Embedded computers where a device has extremely limited resources. For example, a microwave oven will have an extremely small computer to read its touchpad and door sensor, provide output to a digital display and speaker, and to control the machinery for cooking food. This computer will not be powerful enough to run a compiler, a file system, or a development environment. Since debugging and testing may also require more resources than are available on an embedded system, cross-compilation can be less involved and less prone to errors than native compilation.
- Compiling for multiple machines. For example, a company may wish to support several different versions of an operating system or to support several different operating systems. By using a cross compiler, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm. Similar to compiling for multiple machines, a complicated build that involves many compile operations can be executed across any machine that is free, regardless of its underlying hardware or the operating system version that it is running.
- Bootstrapping to a new platform. When developing software for a new platform, or the emulator of a future platform, one uses a cross compiler to compile necessary tools such as the operating system and a native compiler.
- Compiling native code for emulators for older now-obsolete platforms like the Commodore 64 or Apple II by enthusiasts who use cross compilers that run on a current platform (such as Aztec C's MS-DOS 6502 cross compilers running under Windows XP).

Use of virtual machines (such as Java's JVM) resolves some of the reasons for which cross compilers were developed. The virtual machine paradigm allows the same compiler output to be used across multiple target systems, although this is not always ideal because virtual machines are often slower and the compiled program can only be run on computers with that virtual machine.

Q.4. (b) Difference between compiler and interpreter **(2.5)**

Ans.

	Interpreter	Compiler
1.	Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
2.	It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
3.	No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
4.	Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
5.	Programming language like Python,	Programming language like C, C++ use compilers.

Q.4. (c) Handle and Handle Prunning

Ans: *Handles:*

A *handle* of a string is a substring that matches the right side of a production, ose reduction to the nonterminal on the left side of the production represents one ing the reverse of a rightmost derivation.

Precise definition of a handle:

A *handle* of a right-sentential form γ is a production $A \to \beta$ and a position of γ the string β may be found and replaced by A to produce the previous right- ial form in a rightmost derivation of γ.

i.e., if $S \Rightarrow \alpha Aw \Rightarrow \alpha\beta w$, then $A \to \beta$ in the position following α is a handle of $\alpha\beta w$.

The string w to the right of the handle contains only terminal symbols.

In the example above, abbcde is a right sentential form whose handle is $A \to b$ at n 2. Likewise, aAbcde is a right sentential form whose handle is $A \to Abc$ at n 2.

Handle Pruning:

A rightmost derivation in reverse can be obtained by *handle pruning*.

i.e., start with a string of terminals w that is to parse. If w is a sentence of the mar at hand, then $w = \to\gamma_n$, where $\to\gamma_n$ is the nth right sentential form of some as known rightmost derivation.

$$S = \gamma_0 \gamma_1 \gamma_2 \cdots \gamma_{n-1} \gamma_n = w.$$

xample for right sentential form and handle for grammar

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

ight Sentental orm	Handle	Reduction Production
d1 + id2 * id3	id1	$E \to id$
+ id2 * id3	id2	$E \to id$
+ E * id3	id3	$E \to id$
+ E • E	E * E	$E \to E * E$
+ E	E + E	$E \to E + E$

Q.4. (d) LEX Tool (2.5)

Ans: **Lex** is a computer program that generates lexical analyzers ("scanners" or ers").

Lex is commonly used with the yacc parser generator. Lex, originally written by Mike and Eric Schmidt and described in 1975, is the standard lexical analyzer generator many Unix systems, and an equivalent tool is specified as part of the POSIX standard.

Lex reads an input stream specifying the lexical analyzer and outputs source implementing the lexer in the C programming language

pen source

Though originally distributed as proprietary software, some versions of Lex are open source. Open source versions of Lex, based on the original AT&T code are now tributed as open source systems such as OpenSolaris. One popular open

source version of Lex, called flex, or the "fast lexical analyzer", is not derived from proprietary coding.

Structure of a Lex file

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows

- The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
- The **C code** section contains C statements and functions that are copied to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Example of a Lex file

The following is an example Lex file for the flex version of Lex. It recognizes strings of numbers (positive integers) in the input, and simply prints them out.

```
/*** Definition section ***/
%{
/* C code to be copied verbatim */
#include <stdio.h>
%}
/* This tells flex to read only one input file */
%option noyywrap
%%
/*** Rules section ***/
/* [0-9]+ matches a string of one or more digits */
[0-9]+ {
/* yytext is a string containing the matched text. */
printf("Saw an integer: %s\n", yytext);
}
|\n { /* Ignore all other characters. */ }
%%
/*** C Code section ***/
int main(void)
{
/* Call the lexer, then quit. */
yylex();
return 0;
}
```

If this input is given to flex, it will be converted into a C file, lex.yy.c. This can be compiled into an executable which matches and outputs strings of integers. For example, given the input:

```
abc123z. ! &*2gj6
```

the program will print:

```
Saw an integer: 123
Saw an integer: 2
Saw an integer: 6
```