COMPTLER

MODEL ANSWER IN-SEM EXAM-19

Name: Prof. Gharu Anand
Dept: Computer Dept
Class: BE.
Subject: Compiler.
College: PVGCOE, Nasik

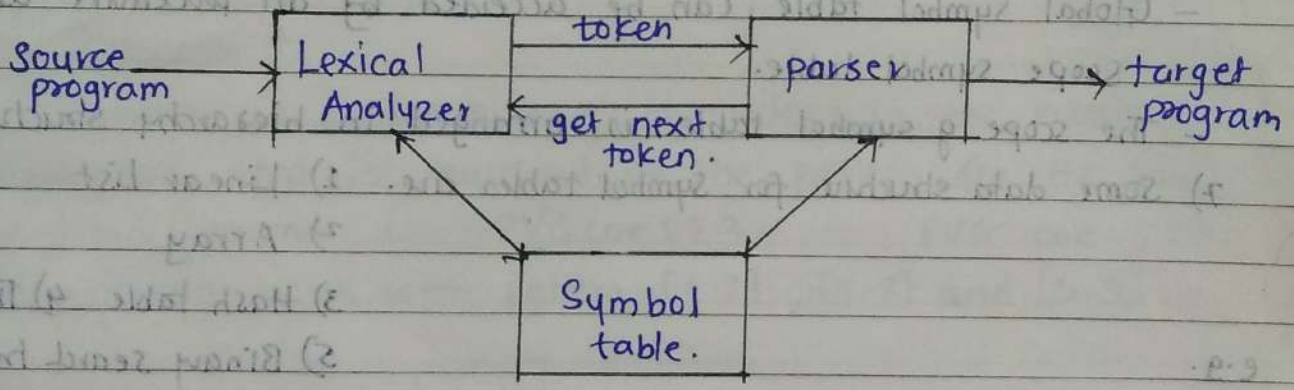
Q.1 4-Marks.

a) Briefly describe role of lexical analyzer with diagram.

→ 1] Lexical Analyzer is the first phase of the compiler.

2] The main purpose of the lexical analyzer is to read input character and produce as o/p a sequence of tokens that parser uses for syntax anlysis.

3] Lexical analyzer scan the program from left to right, and generates the token by seperating the identifier, number, keyword & alphabets etc.



Interaction of Lexical Analyzer With Parser.

4] Lexical Analyzer is also called as Scanning.

5] Above fig. shows the interaction between Lexical analyzer and parser.

6] As lexical analyzer generates token. and it passes the token to the parser for creating syntax tree.

7] symbol table is used to keep the information or token which are generated by Lexical analyzer.

8] parser demands the token to the lexical analyzer for genelating syntax tree.

Q.1.

2] What are the data structure used for Symbol table? 4 Marks.

→ "Symbol table is an imp data structure created & maintained by compiler in order to store infor abaut variable name, constant, number, keyword for future reference"

1) Symbol table is used by Analysis & synthesis phase.
2) it stores name & all entities in a structured form
3) It uses to verify if variable declared
4) It uses for type checking & conversion.
5) Symbol table is one of the part of Compiler.
6) Compiler Contains 2 type of symbol table.
   i) Global symbol table
   ii) Scope Symbol table.
   - Global Symbol table can be accessed by all procedure and scope symbol table.
   - The scope of symbol table is arranged in hierarchy structure
7) Some data structure for Symbol tables are. 1) Linear List
   2) Array
   3) Hash table 4) Tree etc.
   5) Binary search tree.

e.g.

```
# include <stdio.h>
{
void main ()
  int a, b;
  float c = 1.2;
}.
```

Symbol table ⟹

Token	meaning
#	preprocessor
include	keyword
<stdio.h>	header file.
{	braces
void	keyword
main()	function
int	data type
a	variable
b	variable
float	data type
c	variable
}	braces.

Q.1.

c) Define Token, pattern and lexemes with example ? 2-Marks

→ 1) Token :

Token is a sequence of characters that can be treated as single logical entities.

e.g. identifier, keyword, operator, constant, Special symbol etc

2) ~~Lex~~ Pattern :

pattern is set of rules which describe the structure or behaviour of program.

e.g. [0-9] detect only number 0-9

[a-z] detect only small letter.

[\n] if new line.

3) Lexemes :

Lexemes is a sequence of characters in the source program that is matched by the pattern for the token.

e.g. lexemes is PVGcoe123. PVG coe 123.

it will match with pattern [A-Z], [a-z] and [0-9].

OR

Q.2 4-Marks

a) What is regular expression for identifier declaration, string literal, comments, floating point no, white space for C lang

→ "Regular expression is an imp notation for specifying patterns. each pattern matches set of strings so regular expression serve as a set of strings"

1) Regular Expression for Identifier :- (declaration)

Identifier = (letter) (letter | digit)*.

2) string Literal =

Literal = Integer | Boolean | float | char.

3) Comment =

// single line comment
/* */ multi-line comment. } for progr

& using Lex = comment = ?#

4) Floating-point no =

float = integer.integer.

5) White spaces =

white spaces = [' '] or [\s].

Q.2.

b) Explain front End & Back-End of Compiler with e.g. 4-Marks

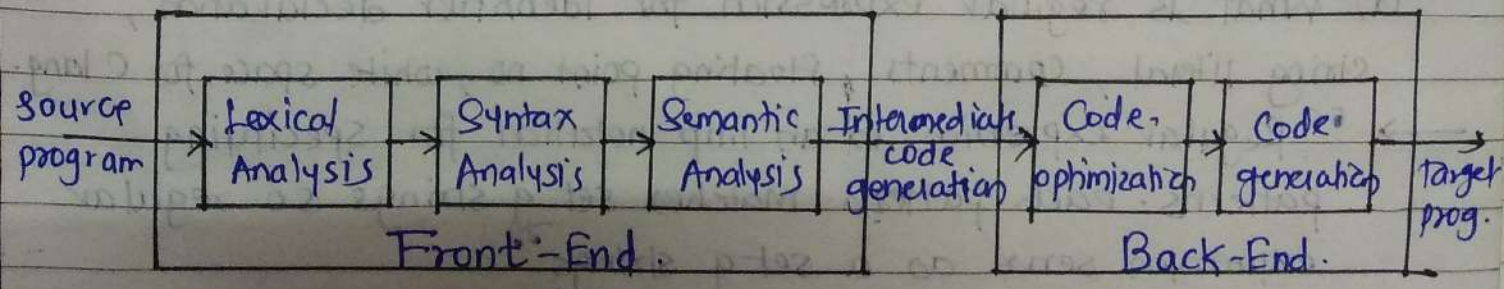
→ 1) Front End - Back-End is also called as Analysis - synthesis phase (phases of compiler)

2) The phases of compiler is collected into Front-End & Back end.

3) Front-end consist of those phases which are depend on Source program i.e. Lexical anlysis, Syntax & Semantic.

4) Back-end Consist of those phases which are depend on target program or synthesis phase

i.e. Intermediate code generation, Code optimization, Code generation



FRONT-END & BACK-END OF COMPILER.

5) Front-End analysis source program and produce intermediate Code while the back-end synthesizes the target program from intermediate code.

6) Front End analyse the code by scanning & checking syntax whereas Back-end synthesize intermediate code to code optimizer & Code generator.

Q.2.

c) Discuss the various Garbage collection techniqus 2-Marks

→ * Garbage Collection Techniques : " Garbage collection is the automatic storage collechian g comp. storage."

1) Reference Counting
2) Mark and Sweep.
3) Copying Collection.

1) Reference Counting :

→ - Each object has an associated count g reference (pointers) to it.
- Each time reference to the object is created, its reference count is increased by one or vice-versa
- When reference count reaches to zero, object spaces may be reclaimed.

2) Mark-Sweep collechian :

- Done by tracing - starting at root set and usually traversing the graph g pointer relship. The reached object are marked
- Once all live object are marked, memory is exhaustively examined to find all the unmark (garbage) object and reclaim their space (sweep)

3) Copying - Garbage collechian :

- The algo. moves all the live objects into one area and the rest g heaps become available.
- There are several copying garbage collection, one g which is "stop & copy" garbage collechian.
- In this scheme, heap is divided into two contiguous semispace. During normal program execution, only one g them is in use.

Q.3. a) Construct a SLR(1) parsing table for the given grammar 6-Marks

$S \rightarrow 0S0 \mid 1S1 \mid 10$

I_0 : $S' \rightarrow .S$
$S \rightarrow .0S0$
$S \rightarrow .1S1$
$S \rightarrow .10$

I_1 : goto (I_0, S)
$S' \rightarrow S.$

I_2 : goto $(I_0, 0)$
$S \rightarrow 0.S0$
$S \rightarrow .0S0$
$S \rightarrow .1S1$
$S \rightarrow .10$

I_3 : goto $(I_0, 1)$
$S \rightarrow 1.S1$
$S \rightarrow 1.0$
$S \rightarrow .0S0$
$S \rightarrow .1S1$
$S \rightarrow .10$

I_4 : goto (I_2, S)
$S \rightarrow 0S.0$

I_5 : goto (I_3, S)
$S \rightarrow 1S.1$

I_6 : goto $(I_3, 0)$
$S \rightarrow 10.$

I_7 : goto $(I_4, 0)$
$S \rightarrow 0S0.$

I_8 : goto $(I_5, 1)$
$S \rightarrow 1S1.$

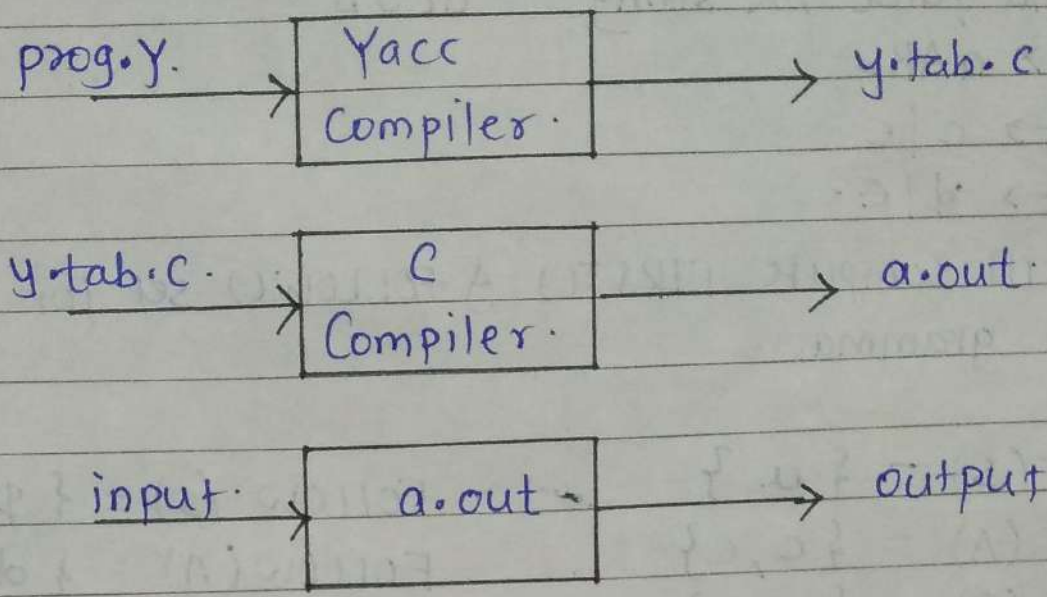
SLR parsing table

State	Action			Goto	
	1	0	$	S	0
0	S_3	-	Accept	-	2
1	-	-	-	-	-
2	-	-	-	4	-
3	-	S_6	-	5	-
4	-	-	-	-	7
5	-	-	-	-	-
6	r_3	-	r_3	-	-
7	r_1	-	r_1	-	-
8	r_2	-	r_2	-	-

Q.3. 4-Marks.

b) Explain Automatic Construction of parser using YACC.

→



YACC Automatic Parser Generator.

- YACC stands for yet Another Compiler - Compiler.
- Yacc is automatic tool that generates parser program.
- Yacc is available in Unix OS.
- Basically YACC is LALR parser generator that reports conflicts or uncertainties in the form of error menage.
- YACC translator can be represented as shown in figure.
- YACC compiler Accept input file as prg.y and it is compiled by Yacc compiler.
- YACC compiler generates y.tab.c file.
- y.tab.c file is passed to the C-compiler, C-compiler generates a.out file or lex.yy.c file.
- a.out block takes i/p file and generates executable file.
- Yacc compiler uses yyparse() to parse the program.
- Yacc compiler demands token to the lex program while compiling yacc program to generate syntax tree or for checking or validation of expression.

OR

Q.4 a) Construct Predictive Parser for following grammar and parse the string "acdb" 6-Marks.

$S \rightarrow aABb$

$A \rightarrow c \mid \epsilon$

$B \rightarrow d \mid \epsilon$

→ we will compute FIRST() & FOLLOW() set for given grammar.

FIRST(S) = {a}
FIRST(A) = {c, ϵ}
FIRST(B) = {d, ϵ}

FOLLOW(S) = {$}
FOLLOW(A) = {d, b}
FOLLOW(B) = {b}.

Follow(A) = $S \rightarrow aABb$
$A \rightarrow \alpha B B$
$A=S, \alpha=a, B=A, B=Bb$.
FIRST(B) = Bb
FIRST(B) = {d}
Follow(A) = {d, b}.

Follow(A) = $S \rightarrow aABb$.
$A \rightarrow \alpha BB$
$A=S, \alpha=a, B=AB, B=b$.
FIRST(B) = {b}.
Note :- Not mandatory.

* LL parsing table will be :-

	a	b	c	d	$
S	S→aABb				
A		A→ϵ	A→c	A→ϵ	
B		B→ϵ		B→d	

According to Rule → $A \rightarrow \alpha$.

e-g.
1) $S \rightarrow aABb$
$A \rightarrow \alpha$
$A=S, \alpha=aABb$.
FIRST(aABb)
FIRST(a) = M[S, a]

2) $A \rightarrow \epsilon$ Follow(A) = {d, b}
$A \rightarrow \alpha$ M[A, d]
$A=A, \alpha=\epsilon$ M[A, b]

parse the string "acdb"

stack	input	Action
$	acdb	S→aABb.
$bBAa	acdb	—
$bBA	cdb	A→c
$bBc	cdb	—
$bB	db	B→d
$bd	db	—
$b	b	—
$	$	Accept

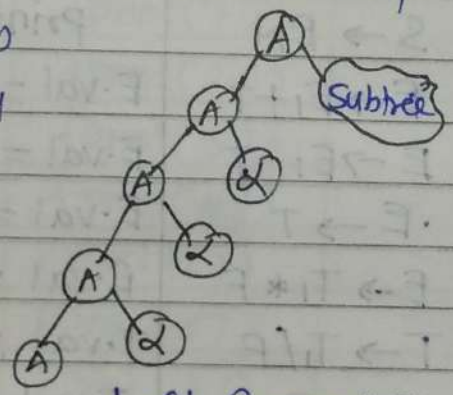
Q.4.

b) Explain what is elimination of left recursion and left factoring in predictive parsing 4-Marks

→ The left recursive grammar is a grammar which is as given below.

$$A \overset{+}{\Rightarrow} A\cdot\alpha$$

- Here, $\overset{+}{\Rightarrow}$ means deriving the i/p in one or more steps. The A can be non-terminal & α denotes some i/p string.
- If left recursion is present in the grammar then it creates problem. Because of left recursion the top-down parser can either in infinite loop as shown in fig.
- Thus, expansion of A causes further expansion of A only and due to generation of A, $A\alpha$, $A\alpha\alpha$ --- $A\alpha\alpha\alpha$
- To eliminate left recursion we need to modify grammar.



Left Recursion.

Let G be a CFG having a produ rule

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

$$\Downarrow$$

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

an equivalent grammar.

In general,

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \beta_n$$

where β_1-β_n do not start with A.

$$\Downarrow$$

eliminate left recursion

$$A \rightarrow \beta_1 A \mid \cdots \mid \beta_n A$$
$$A' \rightarrow \alpha_1 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

2) Left factoring:-

- In left factoring, it is not clear which of 2 alternative productions to use to expand non-terminal A. i.e. if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$.
- we don't know whether to expand A or $\alpha\beta_1$ or to $\alpha\beta_2$
- To eliminate left factoring for this grammar, replace all productions containing α as prefix by $A \rightarrow \alpha A'$ then $A' \rightarrow \beta_1 \mid \beta_2$

Q.5 a) Write form of Syntax Directed definition & Syntax directed translation. 4-Marks.

→ "Syntax directed definition is a generalization of CFG in which each grammar production $X \to \alpha$ is associated with it a set of semantic rules in the form of $a := f(b_1, b_2 \ldots b_k)$, where a is an attribute obtained from the function f."

- SDD can be written as :

production Rule	Semantic Rule/action
$S \to EN$	Print (E.val)
$E \to E_1 + T$	E.val = E_1.val + T.val
$E \to E_1 - T$	E.val = E_1.val − T.val
$E \to T$	E.val = T.val
$E \to T_1 * F$	E.val = T_1.val * F.val
$T \to T_1 / F$	T.val = T_1.val / F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
F = digit	F.val = digit.lexval
$N \to ;$	can be ignored by lex. analyzer as ; it is terminating symbol

* Syntax Directed Translation :

"During process of parsing evaluation of attribute take place by consulting the semantic action enclosed in { }. At the right of the grammar symbol. This process of exe. of code fragment Semantic action from the SDD is called Syntax Directed Translation."

Syntax Directed Translation (SDT)

→ E → TP

T → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

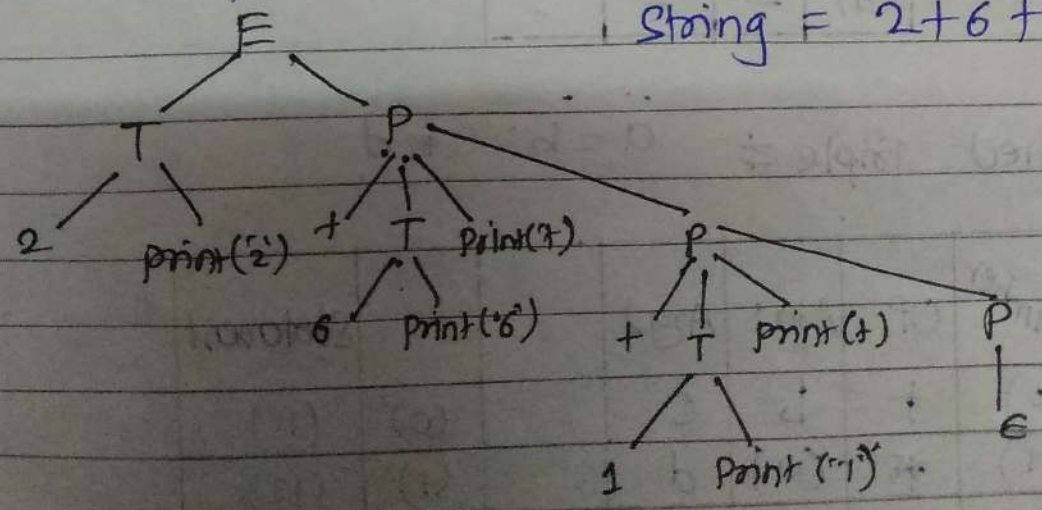
P → +TP | ∈

The translation scheme for this grammar is as given below

production	Semantic Action
E → TP	
T → 0	{print('0')}
T → 1	{print('1')}
T → 2	{print('2')}
T → 3	{print('3')}
T → 4	{print('4')}
T → 5	{print('5')}
T → 6	{print('6')}
T → 7	{print('7')}
T → 8	{print('8')}
T → 9	{print('9')}
P → +TP | ∈	{print('+')}P | ∈

It's annoted pase tree.

String = 2+6+1



Annoted Parse Tree

Q.5.

b) Generate three Addres code., quadruple, Triple, Indirect Triple form for foll. example.

$a = b + c * d$.

6=Marks.

→ 1) Three Addres code :

$t1 = b + c$

$t2 = t1 * d$

$a = t3$.

2) Quadruple = $a = b + c * d$

	OP	Arg1	Arg2	Result
(0)	+	b	c	t1
(1)	*	t1	d	t2
(2)	=	t3	-	a
(3)				
(4)				

$a = b + c * d$

3) Triple =

Number	OP	Arg1	Arg2
(0)	+	b	c
(1)	*	(0)	d
(2)	=	a	(2)
(3)	-	-	-

4) Indirect Triple = $a = b + c * d$

Number	OP	Arg1	Arg2
(0)	+	b	c
(1)	*	(11)	d
(2)	=	a	(13)

	Statement
(0)	(11)
(1)	(12)
(2)	(13)

OR.

Q.6 a) Explain S-attribute & L-attribute with e.g. 4 Marks

→ * Synthesized Attribute :-

The attribute a is called synthesized attribute of X and $b_1, b_2 \ldots b_k$ are attribute belonging to the production symbols.

The value of synthesize ~~node~~ attribute at a node is computed from the value of attribut at the children of that node in the parse tree.

- In SDD, terminal have synthesized attribute only. Thus, there is no definition of terminal. The synthesized attribute can be quite often used in SDD. The SDD that uses only synthesized attribute is called as S-attributed definition.

* L-attributed Definition :-

The SDD can be defined as the L-attributed for the production rule $A \rightarrow X_1 X_2 \ldots X_K$ where the inherited attribute X_K is such that $1 \leq K \leq n$.

The Production $A \rightarrow X_1 X_2 \ldots X_n$ is such that:

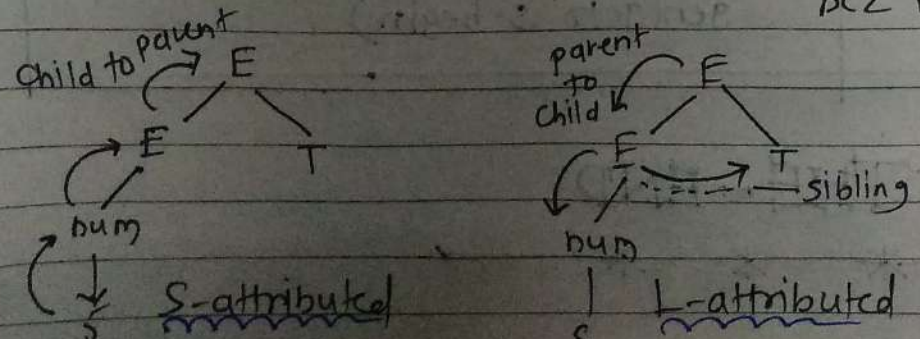
1) It depends upon the attributes of the symbol $X_1, X_2 \ldots X_{j-1}$ to the left of X_j
2) It also depends upon the inherited attribut of A.

e.g.

Check whether given SDD is L-attributed or not.

$A \rightarrow PQ \quad P.in := P(A.in) \longrightarrow$ L-attributed

$A \rightarrow XY \quad X.in := X(Y.sY) \longrightarrow$ Not L-attributed
bcz value depend on right symbol



Q.6.

b) Give SDD for if...else, if...else then and while...Do statement. 6-Marks.

→ * SDD for If...else statement :-

1)	S → if E then S1	E.true = newlable E.false = S.next S1.next = S.next S.code = E.code || gen(E.true) || S.co

* SDD for If — then ... else statement.

	Production	Semantic Action.
2)	S → if E then S1 else S2	E.true = newlable E.false = newlable S1.false = S.next S2.next = S.next S.code = E.code || gen(E.true) || S1.code.gen(goto S1.next) || gen('E.false':') || S2.code

* SDD for while...Do statement

	Production	Semantic Action.
3)	S → while E do S1	S.begin = newlabel E.true = newlabel E.false = S.next S1.next = S.begin S.code = gen(S.begin':' || E.true':' gen(E.true';') || S1.code || ~~gen~~. gen(goto' S.begin)

THE - END