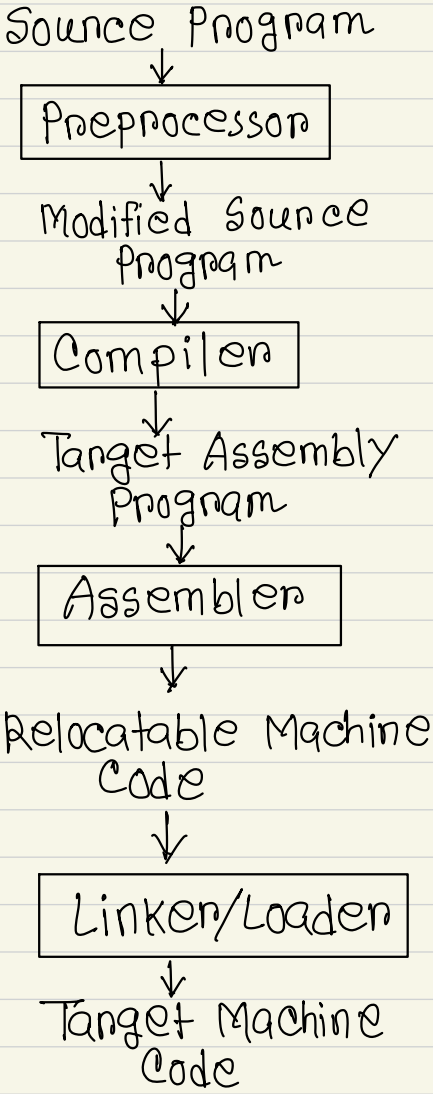
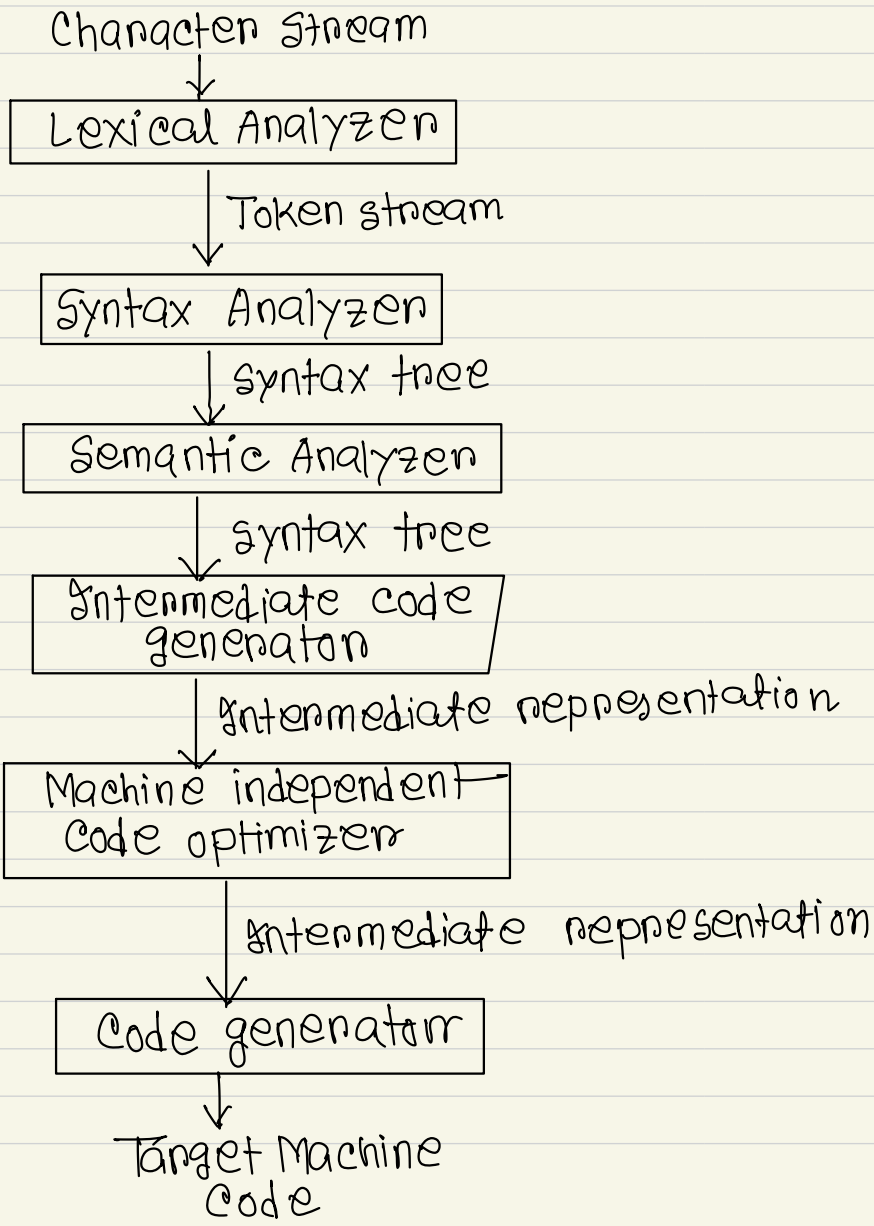
Mid 21-22

a) A computer program is a set of instructions understandable by human beings. Discuss the steps to convert a computer program into computer understandable format in brief.



b) The process of constructing a compiler can be modularized to improve efficiency. Draw a block diagram showing various construction phases and modules of a compiler and discuss in brief.



c) Discuss the strategies to recover from lexical errors.

i) Panic mode error recovery

ii) Deleting an extraneous character

iii) Replacing an incorrect character

iv) Inserting a missing character

v) Transposing 2 adjacent characters

a) Consider the following grammar:

$A \rightarrow aB \mid b$

$B \rightarrow cC \mid d$

$C \rightarrow a \mid c$

What type of phrase-structure grammar can accurately describe the grammar? Justify your answer.

Type 3

w1 = A and w2 = either aB or a

b) You need to construct a predictive parser for the following grammar:

$A \rightarrow AB \mid AC \mid b$ (α α β)

$B \rightarrow Bd \mid e \mid d$ (α β β)

$C \rightarrow a \mid d$

Preprocess the grammar to make it ready to work with predictive parser.

Left recursion

$A \rightarrow bA'$

$A' \rightarrow BA' \mid CA' \mid \varepsilon$

$B \rightarrow eB' \mid dB'$

$B' \rightarrow dB' \mid \varepsilon$

$C \rightarrow a \mid d$

c) Show the steps to prove that the string *baedd* is a valid sentence for the grammar given in Question 2.b) using recursive descent parsing method.

$A \rightarrow bA'$

$A' \rightarrow BA' \mid CA' \mid \varepsilon$

$B \rightarrow eB' \mid dB'$

$B' \rightarrow dB' \mid \varepsilon$

$C \rightarrow a \mid d$

$A \rightarrow bA'$

$\rightarrow bCA'$

$\rightarrow bCBA'$

$\rightarrow bceB'A'$

$\rightarrow baeB'A'$

$\rightarrow baedB'A'$

$\rightarrow baeddB'A'$

$\rightarrow baeddA'$

$\rightarrow baedd$

[proved]

a) A grammar is given below:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

i. Find the set of FIRST(X) and FOLLOW(X) for each non terminal X.

$E \rightarrow TE'$

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

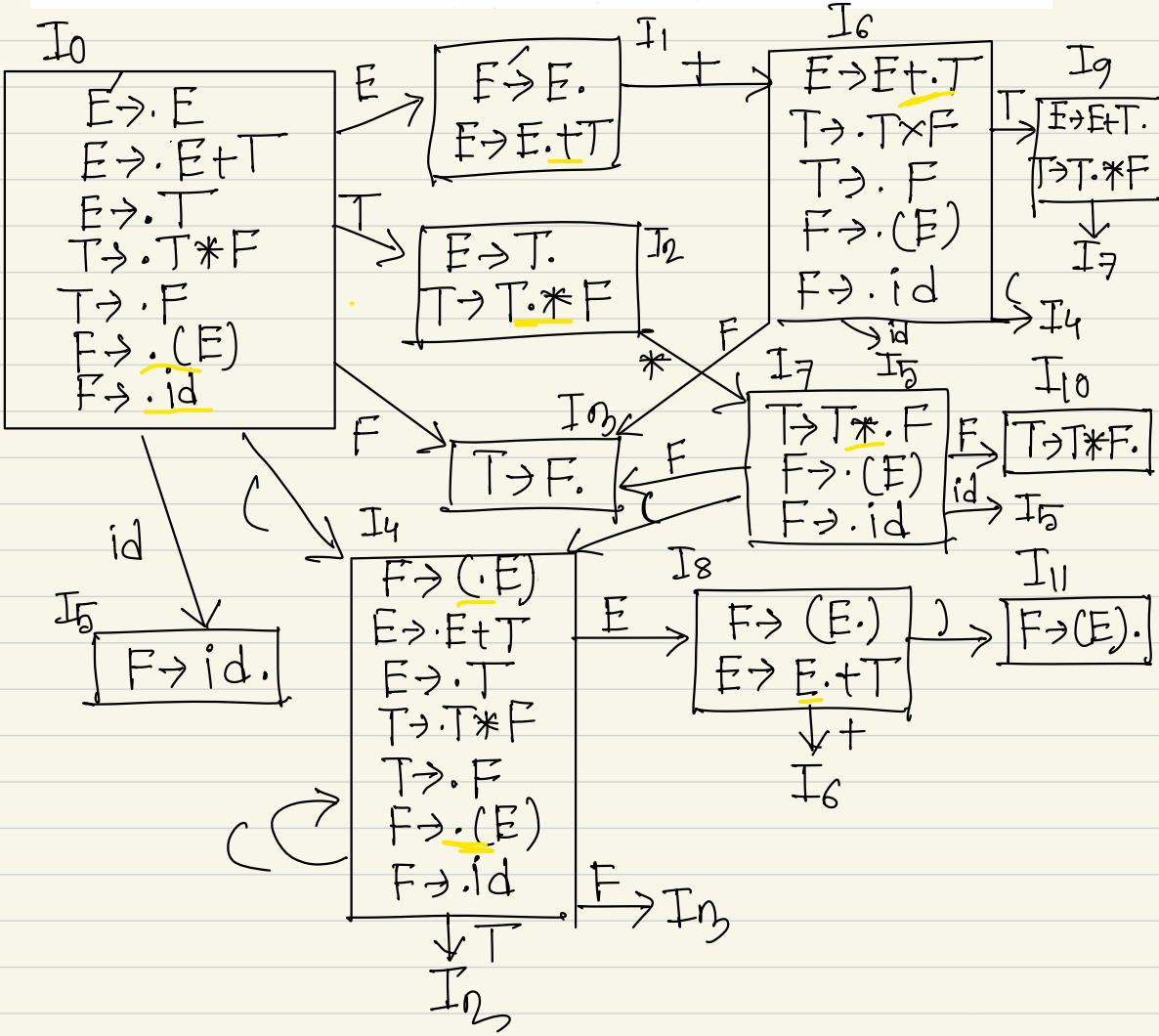
$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

	FIRST	FOLLOW
$E \rightarrow TE'$	{id, (}	{), $}
$E' \rightarrow TE' \mid \varepsilon$	{+, ε}	{), $}
$T \rightarrow FT'$	{id, (}	{+,), $}
$T' \rightarrow *FT' \mid \varepsilon$	{*, ε}	{+,), $}
$F \rightarrow id \mid (E)$	{id, (}	{+, *,), $}

ii. Generate set of states, i.e., LR(0) items for the grammar to construct SLR parser.



STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

	St
(1)	0

b) Explain how the set of FIRST(X) and FOLLOW(X) helps to take decisions during SLR parsing.

$A \rightarrow A\beta$

FIRST

Fill in the action table

If next input is in the FIRST set of production's right side, parser may use it

FOLLOW

Conflicts between shifting and reducing occurs.

FOLLOW sets help to minimize such conflicts

Final 21-22

Consider the context fee grammar shown in Figure 1.

$$
\begin{aligned}
G &\rightarrow L \\
L &\rightarrow E\,;L \\
L &\rightarrow E \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow \textbf{id} \\
T &\rightarrow \textbf{id}\,(L)
\end{aligned}
$$

Figure 1: A context-free grammar for Question 1.

a) Construct a predictive parse table for the grammar.
b) Show the parsing steps for the input text ***id(id+id);id*** using parse table generated from the answer of Question 1.a)

$G \rightarrow L$
$L \rightarrow EL'$
$L' \rightarrow ;L \mid \varepsilon$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow id\,T'$
$T' \rightarrow (L) \mid \varepsilon$

State	FIRST	FOLLOW
G	{id}	{$}
L	{id}	{), $}
E	{id}	{;,), $}
L′	{;, ε}	{), $}
T	{id}	{+, ;,), $}
E′	{+, ε}	{;,), $}
T′	{(, ε}	{+, ;,), $}

	id	;	+	(	)	$
G	G→L					
L	L→EL′					
L′		L′→;L			L′→ε	L′→ε
E	E→TE′					
E′		E′→ε	E′→+TE′		E′→ε	E′→ε
T	T→idT′					
T′		T′→ε	T′→ε	T′→(L)	T′→ε	T′→ε

STACK	INPUT	ACTION
G$	id(id+id);id	
L$	id(id+id);id	G→L
EL'$	id(id+id);id	L→EL'
TE'L'$	id(id+id);id	E→TE'
idT'E'L'$	id(id+id);id	T→idT'
T'EL'	(id+id);id	match id
(L)EL'	(id+id);id	T→(L)
L)EL'	id+id);id	match (
EL')EL'	id+id);id	L→EL'
TE'L')EL'	id+id);id	E→TE'
idT'E'L')EL'	u	T→idT'
T'E'L')EL'	+id);id	match id
E'L')EL'	+id);id	T→ε

a) What are the different storage allocation strategies for function implementation during compiler design? Discuss their pros and cons.

Focus	Static	Stack	Heap
Memory allocation	Compile time	run time	run time (on demand)
Recursion	No	Yes	Yes
Dynamic data structure	No	No	Yes
Usage	Global variables, constants	Local variables, function call	Dynamic data structures

b) You need to design a compiler which will support recursive call of functions. Which storage allocation strategy should you implement? Justify your answer.

Stack/heap

a) State how static checking and dynamic checking can be handled during compiler construction time.

Static

- Check for correctness at compile time
- Syntax errors, type errors, scope violation
- Static semantics

Dynamic

- Verifies at runtime
- Memory access errors, runtime errors, logical errors
- Dynamic semantics

a) Explain implementation of different types of static checking with examples.

Type check

```
int a=5;
float b=3.5;
int c= a+b;      //error
```

Flow-of-control check

```
myfunc(){

  break;     //error
}
```

Uniqueness check

```
myfunc(){
  int i,j,i;     //error
}
```

Name related check

```
myfunc(){
  int result= x+5;  //error
}
```

b) Write down the semantic rules to assign types to the nodes representing left side of the following productions:

- $E \rightarrow E_1 + E_2$
- $E \rightarrow E_1$ **and** E_2
- $E \rightarrow E_1[E_2]$

$E \rightarrow E_1 + E_2$ { E.type := if (E1.type = integer and E2.type = integer) then integer

else if (E1.type = integer and E2.type = float) then float

else type_error }

$E \rightarrow E_1$ and E_2 { E.type := if (E1.type = boolean and E2.type = u) then boolean

else type_error }

$E \rightarrow E_1[E_2]$ { E.type := if (E2.type = integer and E1.type = array (s,t)) then t

else type_error }

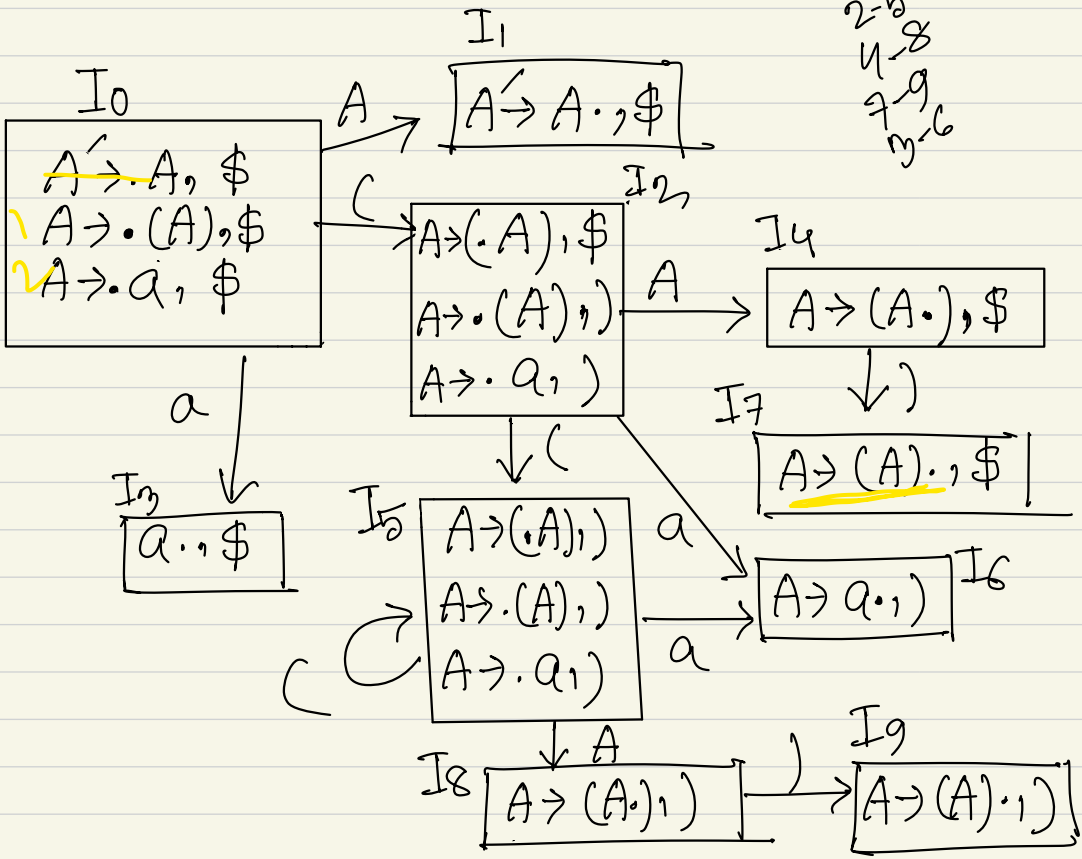
Consider the context-free grammar shown below:

$$A \rightarrow (A)$$
$$A \rightarrow a$$

Figure 2: A context-free grammar for Question 4.

a) Construct the set of LR(1) items and draw the respective transition diagram.

b) Construct a CLR parse table for the grammar using the data generated by the answer of Question 4.a)



State	ACTION				GOTO
	(	)	a	$	A
0	s_2		s_3		1
1				acc	
2	s_5		s_6		4
3				r_2	
4		s_7			
5	s_5		s_6		8
6		r_2			
7				r_1	
8		s_9			
9		r_1			

c) Construct an LALR parse table for the grammar.

$I_2, I_5 \rightarrow I_{25}$
$I_3, I_6 \rightarrow I_{36}$
$I_4, I_8 \rightarrow I_{48}$
$I_7, I_9 \rightarrow I_{79}$

} merged

State	Action				GOTO
	(	)	a	$	A
0	S_2		S_3		1
1			acc		
25	S_5		S_6		48
36		r_2		r_2	
48		S_{79}			
79		r_1		r_1	

a) Discuss Syntax-Directed Definitions in brief.

→ CFG + Attributes & rules

→ Attributes : values, types

→ 2 types of attributes
  i) Synthesized
  ii) Inherited

→ 2 types of SDD

  i) L-attributed : Synthesized / Inherited
     Prefix + Postfix
     top-down

  ii) S-attributed : Synthesized
     Postfix
     bottom-up

b) Assume that the maximum length of the right side of the productions of a grammar is n. How many passes may be required at most to evaluate the inherited attributes of a syntax tree generated for an input for the grammar? Discuss in detail along with definition of inherited attributes.

2 pass

→ First pass (Pre-order) ✓

Evaluate inherited attributes

→ 2nd pass (Post-order) ✓

Evaluate synthesized attributes

c) Provide an efficient tree traversal method to evaluate L-attributed definitions.

Depth-first P-L-R-P

Pre-order → left → right → Post-order

↓ ↓ ↓ ↓

Inherited left subtree right subtree Synthesized

c) Considering the grammar given in Figure 3, design the SDDs to construct syntax tree for a given input using bottom-up approach.

$E \rightarrow E+T$
$E \rightarrow E-T$
$E \rightarrow T$
$T \rightarrow (E)$
$T \rightarrow \textbf{id}$
$T \rightarrow \textbf{num}$

Figure 3: A context-free grammar for Question 6.c)

E → E+T	E.node = new_node('+', E.node, T.node)
E → E-T	E.node = new_node('-', E.node, T.node)
E → T	E.node = T.node
T → (E)	T.node = E.node
T → id	T.node = new_leaf(id, id.entry)
T → num	T.node = new_leaf(num, num.val)

Final 20-21

Consider the context-free grammar shown below and answer the following questions:

$$
\begin{aligned}
G &\rightarrow L \\
L &\rightarrow E\,;\,L \\
L &\rightarrow E \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow \textbf{id} \\
T &\rightarrow \textbf{id}\ (\ ) \\
T &\rightarrow \textbf{id}\ (\ L\ )
\end{aligned}
$$

a) Find the set of FIRST(x) and FOLLOW(x), where x is a non-terminal.

b) Generate canonical LR(0) collection of items for the grammar.

c) Generate the SLR parse table

d) Is the grammar SLR(1)? Justify your answer.

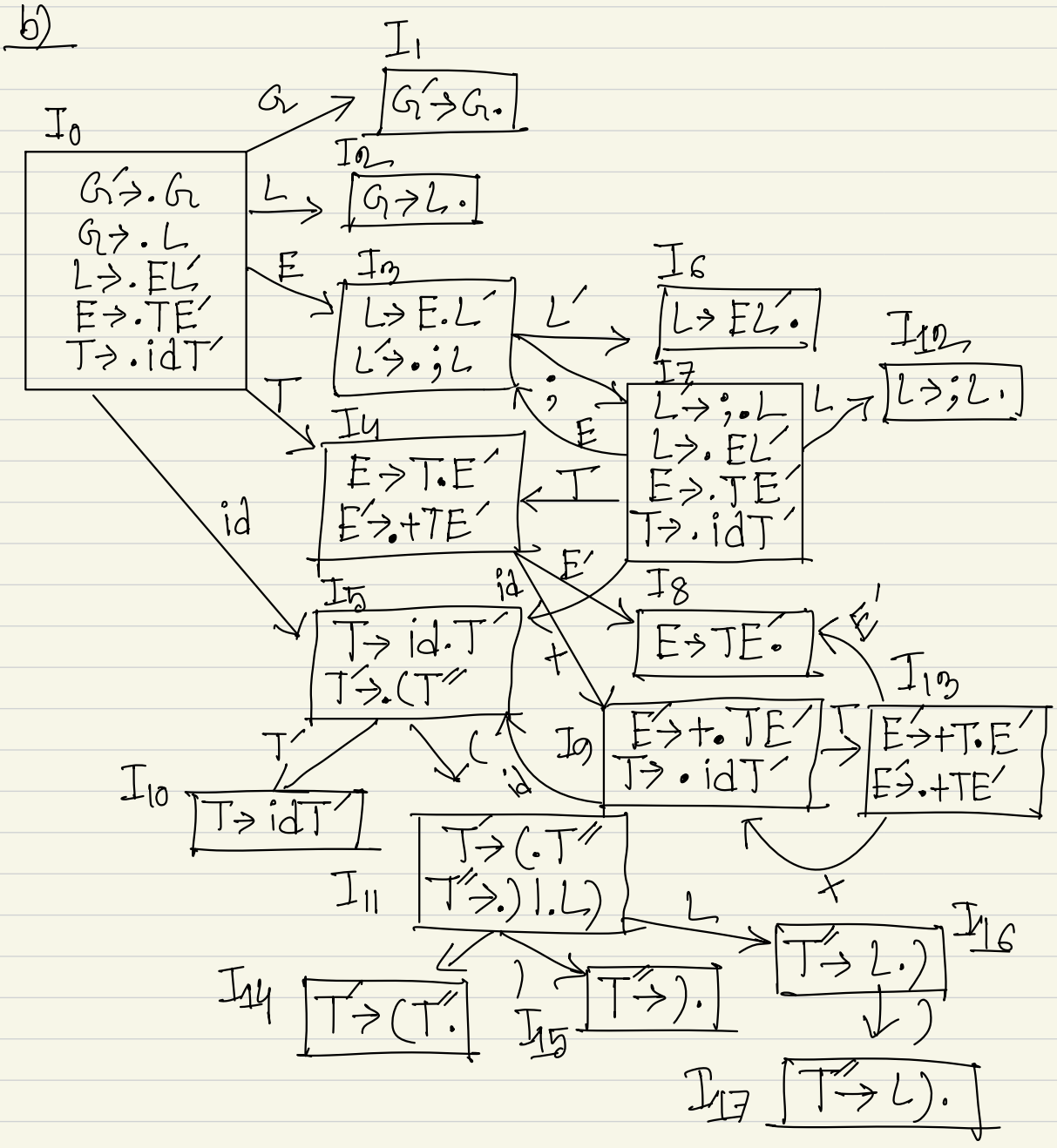
Augmented grammar

a)

$G \rightarrow L$
$L \rightarrow EL'$
$L' \rightarrow ;L \mid \varepsilon$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow \text{id}T'$
$T' \rightarrow (T'' \mid \varepsilon$
$T'' \rightarrow) \mid L)$

	FIRST	FOLLOW
G	{id}	{$}
L	{id}	{$,)}
L'	{;, ε}	{$,)}
E	{id}	{;, $,)}
E'	{+, ε}	{;, $,)}
T	{id}	{+, ;,), $}
T'	{(, ε}	{+, ;,), $}
T''	{), L}	{+, ;,), $}

b)


I0
G'→.G
G→.L
L→.EL'
E→.TE'
T→.idT'
G
I1
G'→G.
L
I2
G→L.
E
I3
L→E.L'
L'→.;L
L'
I6
L→EL'.
;
I7
L→;.L
L→.EL'
E→.TE'
T→.idT'
L
I12
L→;L.
T
I4
E→T.E'
E'→.+TE'
E
T
id
I5
T→id.T'
T'→.(T'')
id
E'
I8
E→TE'.
+
I13
E'→+T.E'
E'→.+TE'
T'
I10
T→idT'
(
id
I9
E'→+.TE'
T→.idT'
T
+
I11
T→(.T''
T''→.)|.L)
L
I16
T''→L.)
)
I14
T→(T''.
)
I15
T''→).
)
I17
T''→L).

a) List the contents of an activation record (for a procedure call) along with brief description.

Local variable
Data local to the function

Temporary values
Stores intermediate values while evaluating an expression

Machine status
Holds status of machine before the func. call

Access link
To access data in other activation record

Control link
Points to the activation record of caller function

Actual parameters
Parameters used to send input to the called func.

Return values
Store result of function call

b) Design syntax-directed definitions to generate intermediate codes for the following statements:

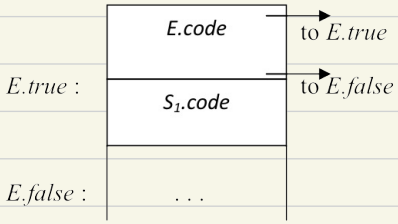
$S \rightarrow$ **if** E **then** $S1$

$S \rightarrow$ **if** E **then** $S1$ **else** $S2$

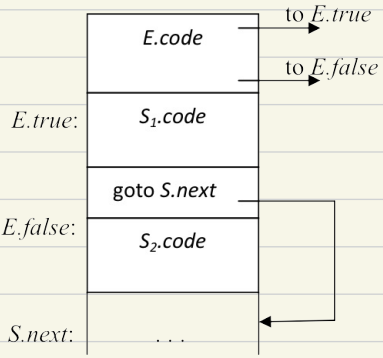
$S \rightarrow$ **do** $S1$ **while** E

$S \rightarrow$ **while** E **do** $S1$

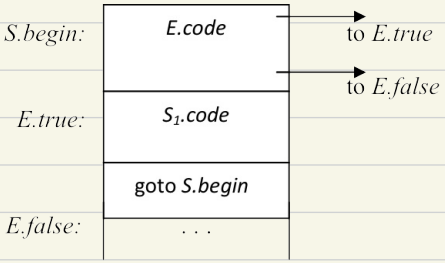
PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen('\textbf{goto}'\ S.next) \mid\mid$ $gen(E.false\ ':') \mid\mid S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \mid\mid E.code \mid\mid$ $gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen('\textbf{goto}'\ S.begin)$



(a) if-then



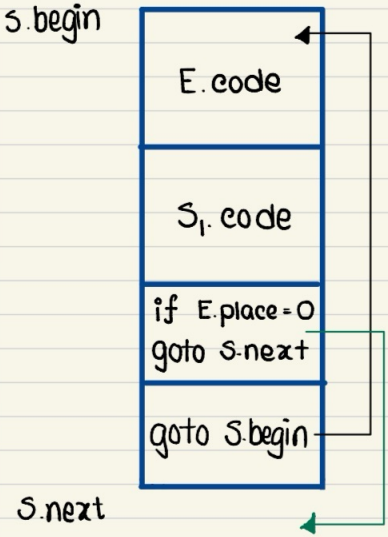
(b) if-then-else



(c) while-do

S → Do S1 while E

Solution :



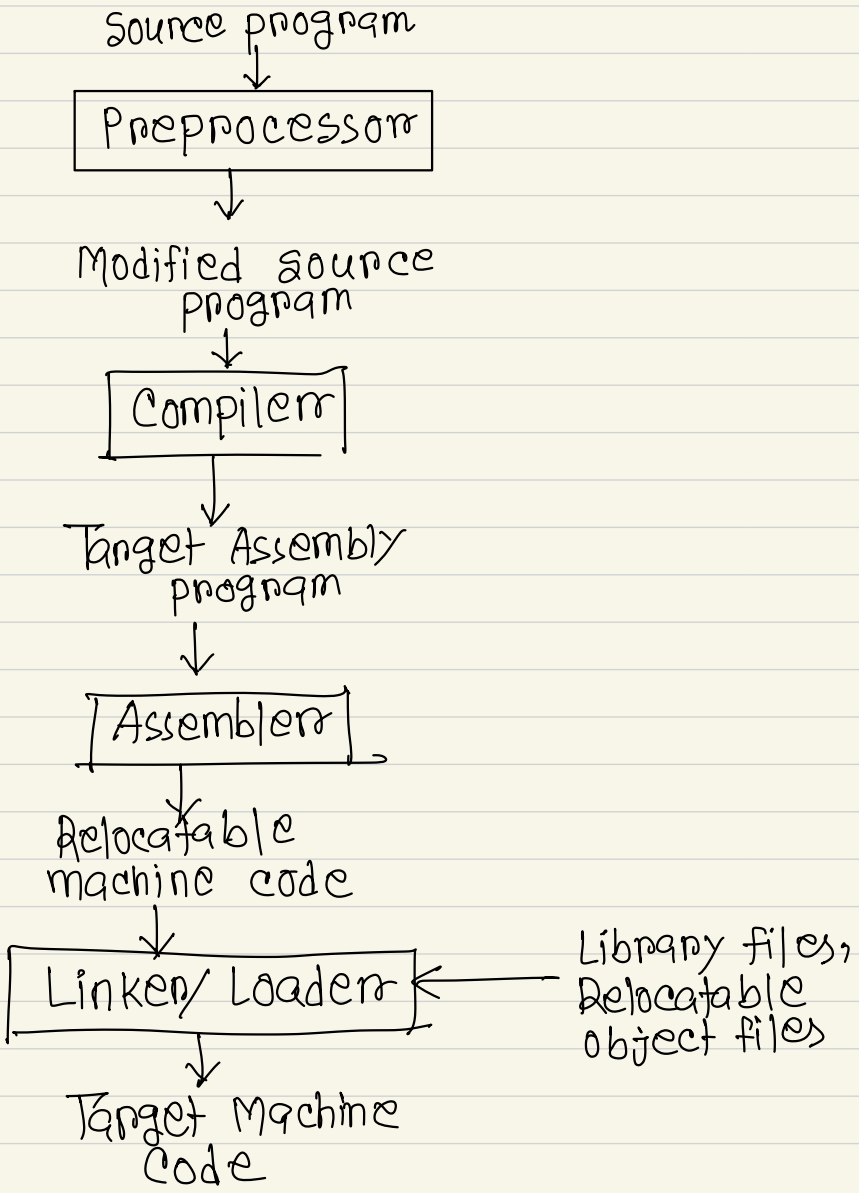
Semantic Rules

S.begin := newlabel ;

S.next := newlabel ;

S.code := gen (S.begin ':') ||
E.code ||
S_1.code ||
gen ('if' E.place '=' '0'
'goto' S.next) ||
gen ('goto' S.begin) ||
gen (S.next ':')

a) Draw the block diagram of a language processing system and briefly discuss each of its components.



b) As a member of a compiler construction team you are asked to implement a *symbol table* along with *symbol table manager*. Discuss the implementation strategy you would follow to complete the task with fast access time and efficient memory uses.

Linear list
- Array
-Linked list

Hash table

c) Discuss the transformation of a grammar which are needed to apply top-down parsing.

1) Eliminate left recursion

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

$\downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

2) Do left-factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$\downarrow$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$

a) Consider the context-free grammar shown below and respective parse table shown in Table 1:

$G \rightarrow L$
$L \rightarrow L\ P$
$L \rightarrow P$
$P \rightarrow (\ P\ )$
$P \rightarrow (\ )$

Table 1: Parsetable

state	action			goto	
	(	)	$	L	P
0	s3			1	2
1	s3		accpt		4
2	r3		r3		
3	s6	s7			5
4	r2		r2		
5		s8			
6	s6	s10			9
7	r5		r5		
8	r4		r4		
9		s11			
10		r5			
11		r4			

Show in full detail, the steps that an LR(1) parser would follow to parse the string (( )) ( ) using the above grammar. For each step of the parsing, show the contents of the stack, present input symbol and the action taken.

	Stack	Symbol	Input	Action
1	0	$	(())() $	shift
2	03	$(	())() $	shift
3	036	$((	))() $	shift
4	03610	$(()	)() $	reduce 5
5	035	$(P	)() $	shift
6	038	$(P)	() $	Reduce 4
7	02	$P	() $	Reduce 3
8	01	$L	() $	shift
9	013	$L(	) $	shift
10	0137	$L()	$	reduce 5
11	014	$LP	$	reduce 2
12	01	$L	$	accept

b) A compiler designer writes following grammar to support *if-then-else* statement:

stmt → *if expr then stmt*
| *if expr then stmt else stmt*
| *other*

Then he realizes that the grammar is ambiguous. So he rewrites the grammar as follows to remedy the dangling-else ambiguity:

stmt → *if expr then stmt*
| *matched_stmt*
matched_stmt -> *if expr then matched_stmt else stmt*
| *other*

Show that the grammar is still ambiguous.

```
if expr
then
   if expr
   then matched-stmt
   else
      if expr
      then matched-stmt
else stmt
```

```
if expr
then
   if expr
   then matched-stmt
   else
      if expr
      then matched-stmt
      else stmt
```

Last "else stmt" creates the ambiguity

b) A compiler is needed to provide recursive call for functions. The compiler designer chose *static allocation* strategy for run-time memory allocation for functions. Explain why the selected run-time memory allocation strategy will fail to support the required recursive function call.

Static storage has,

→ Fixed memory locations

→ Compile time allocation of variables

Static storage fails because,

→ Overwriting variables

↳ recursive calls overwrites each other's variables

→ Fixed return address

↳ can't maintain multiple return addresses for different recursion calls

→ No independent instances

↳ recursion calls can't have a separate context.

c) Design a tree traversal algorithm to evaluate L-Attributed definitions. Write down the pseudocode to implement the algorithm.

Input : A node n in a parse tree T

Output: Evaluation order for the attributes of the subtree of T rooted at n

```
procedure depth_first(n)
begin
   for every child m of n from L→R do begin
      evaluate inherited attributes of m
      depth_first(m)
   end
   evaluate synthesized attributes of n
end
```