20CB601/20CS601/20DS601/20IT601

Hall Ticket Number:

III/IV B. Tech (Regular) DEGREE EXAMINATION

May, 2026 — **Common to CB, CS, DS & IT**

Sixth Semester — **Compiler Design**

Time: Three Hours — **Maximum:** 70 Marks

Answer question 1 compulsorily. **(14X1 = 14 Marks)**

Answer one question from each unit. **(4X14 = 56 Marks)**

			CO	BL	M
1	a)	Define compiler.	CO 1	L1	1 M
	b)	What is a language processor?	CO 1	L1	1 M
	c)	Why should left recursion be eliminated?	CO 1	L1	1 M
	d)	Why is LL(1) grammar important?	CO 1	L1	1 M
	e)	Differentiate SLR and LALR parsing.	CO 2	L2	1 M
	f)	Analyze the need for lookahead in LR parsing.	CO 2	L1	1 M
	g)	What is a syntax-directed definition (SDD)?	CO 2	L1	1 M
	h)	What are the components of three-address code?	CO 3	L1	1 M
	i)	How are basic blocks formed from code?	CO 3	L1	1 M
	j)	Differentiate syntax tree and DAG.	CO 3	L1	1 M
	k)	What is the purpose of static allocation strategy?	CO 4	L1	1 M
	l)	Differentiate static and stack allocation.	CO 4	L1	1 M
	m)	What is a symbol table?	CO 4	L1	1 M
	n)	What are the contents of activation record?	CO 4	L1	1 M

Unit-I

2	a)	Explain the structure of a compiler with a neat diagram. Describe each phase briefly.	CO 1	L1	7 M
	b)	Explain left recursion. Eliminate left recursion from the following grammar: $E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow (E) \mid id$ Explain each step.	CO 1	L3	7 M

(OR)

3	a)	Consider the Grammar S->(L)|a L->SL' L' ->)SL' | ε Check whether the given grammar is LL(1) or not.	CO 1	L3	7 M
	b)	Describe the working procedure of recursive descent parsing and explain how grammar rules are implemented using functions.	CO 1	L2	7 M

Unit-II

4	a)	Consider the grammar. S-->S+S S-->S*S S --> id Perform Shift Reduce parsing for input string "id + id + id".	CO 2	L3	7 M
	b)	Explain YACC (Yet Another Compiler Compiler) and describe its role in generating parsers for a compiler.	CO 2	L2	7 M

(OR)

5	a)	Construct CLR parsing table for the given context free grammar S-->AA A-->aA|b	CO 2	L4	10 M
	b)	Explain Syntax Directed Definitions (SDD) and describe how they are used to associate semantic rules with grammar productions.	CO 2	L2	4 M

P.T.O

Unit-III

6	a)	Write quadruple, triples and indirect triples for following expression: (x + y) * (y + z) + (x + y + z)	CO 3	L3	7 M
	b)	Explain simple code generation algorithm for the following. t = a-b u = a-c v = t+u	CO 3	L3	7 M

(OR)

7	a)	Give SDT scheme for Boolean expression using backpatching. Generate three address code for the Boolean expression a>b && c<d || e>f.	CO 3	L2	7 M
	b)	Compute the basic blocks for the given three address statements- (1) PROD = 0 (2) I = 1 (3) T2 = addr(A) – 4 (4) T4 = addr(B) – 4 (5) T1 = 4 x I (6) T3 = T2[T1] (7) T5 = T4[T1] (8) T6 = T3 x T5 (9) PROD = PROD + T6 (10) I = I + 1 (11) IF I <=20 GOTO (5)	CO 3	L3	7 M

Unit-IV

8	a)	How does runtime environment manage memory during runtime?	CO 4	L2	4 M
	b)	Describe the data structures used for symbol tables. How do different structures affect search and insertion operations?	CO 4	L2	10 M

(OR)

9	a)	A program contains recursive and non-recursive functions. Explain how the runtime environment distinguishes between static and stack allocation for these cases. Why is stack allocation preferred for recursion?	CO 4	L2	7 M
	b)	A compiler needs fast access to identifiers during parsing. Compare how different data structures for symbol tables (like hash tables and trees) handle insertion and lookup. Which is more efficient in this scenario?	CO 4	L4	7 M

1(a) Define compiler. **[1M]**

A compiler is a system software that translates a high-level language program into equivalent machine language or assembly language program. It also detects and reports errors during translation.

1(b) What is a language processor? **[1M]**

A language processor is software that converts a source program into machine understandable form. Examples include compiler, interpreter, and assembler.

1(c) Why should left recursion be eliminated? **[1M]**

Left recursion must be eliminated because recursive descent parsers and LL(1) parsers cannot handle left recursive grammars. Left recursion causes infinite recursion during parsing.

1(d) Why is LL(1) grammar important? **[1M]**

LL(1) grammar allows top-down parsing using one lookahead symbol. It helps in constructing efficient predictive parsers without backtracking.

1(e) Differentiate SLR and LALR parsing. **[1M]**

SLR uses FOLLOW sets for reductions and is simpler but less powerful. LALR combines LR(1) states having same cores, producing compact parsing tables with better power than SLR.

1(f) Analyze the need for lookahead in LR parsing. **[1M]**

Lookahead symbols help LR parsers decide correct reductions and avoid ambiguity. They improve parsing accuracy and reduce parsing conflicts.

1(g) What is a syntax-directed definition (SDD)? **[1M]**

An SDD is a context free grammar augmented with attributes and semantic rules used to define semantics of language constructs.

1(h) What are the components of three-address code? **[1M]**

Three-address code contains operators, operands, temporaries, assignments, labels, and jump instructions.

1(i) How are basic blocks formed from code? **[1M]**

Basic blocks are formed by identifying leaders. The first statement, targets of jumps, and statements following jumps become leaders. Statements between leaders form a basic block.

1(j) Differentiate syntax tree and DAG. **[1M]**

A syntax tree represents hierarchical structure of expressions. A DAG eliminates common subexpressions and avoids duplication of nodes.

1(k) What is the purpose of static allocation strategy? [1M]

Static allocation reserves memory during compile time. It is useful when sizes and lifetimes of data objects are known before execution.

1(l) Differentiate static and stack allocation. [1M]

Static allocation allocates memory permanently during execution. Stack allocation allocates memory dynamically during function calls and deallocates after return.

1(m) What is a symbol table? [1M]

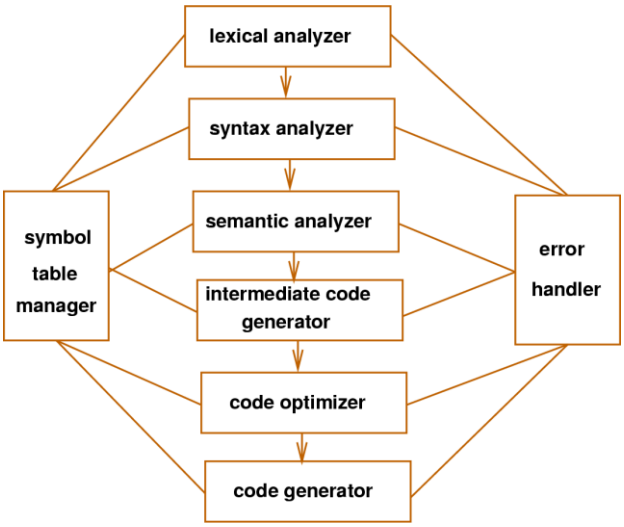
A symbol table is a data structure used by a compiler to store information about identifiers such as names, types, scope, and addresses.

1(n) What are the contents of activation record? [1M]

Activation record contains parameters, return address, local variables, temporary variables, saved registers, and control links.

2(a) Explain the structure of a compiler with a neat diagram. Describe each phase briefly. **[7M]**

Ans:



A compiler consists of several phases:

1. Lexical Analysis:

   Converts source code into tokens.

   Removes spaces and comments.

2. Syntax Analysis:

   Checks grammatical correctness using parsing techniques.

3. Semantic Analysis:

   Verifies meaning of statements, type checking, and scope checking.

4. Intermediate Code Generation:

   Generates intermediate representation such as three-address code.

5. Code Optimization:

   Improves efficiency of generated code.

6. Code Generation:

   Produces target machine code.

7. Symbol Table Management:

   Stores identifier information.

8. Error Handling:

Detects and reports errors in all phases.

2(b) Explain left recursion. Eliminate left recursion from the following grammar:

E → E + T | T

T → T * F | F

F → (E) | id

Explain each step. **[7M]**

Ans:

Given grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Step 1: Eliminate left recursion in E

$E \rightarrow TE'$

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

Step 2: Eliminate left recursion in T

$T \rightarrow FT'$

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

Step 3: F remains same

$F \rightarrow (E) \mid id$

Final grammar:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

3(a) Consider the Grammar

S->(L)|a

L->SL'

L' ->)SL' | ε

Check whether the given grammar is LL(1) or not. [7M]

Ans:

Given Grammar:

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow)SL' \mid \varepsilon$

Procedure:

1. Compute FIRST sets.

2. Compute FOLLOW sets.

3. Ensure no conflicts between productions.

First Set

FIRST(S) = { (, a }

FIRST(L) = {) , a }

FIRST(L′) = {) , ε }

Follow Set

FOLLOW(S) = {) , $ }

FOLLOW(L) = {) }

FOLLOW(L′) = {) }

Predictive Parsing Table

Non-Terminal	**(**	**a**	**,**	**)**	**$**
S	S→(L)	S→a	—	—	—
L	L→SL′	L→SL′	—	—	—
L'	—	—	L′→,SL′	L′→,SL′	—

The predictive parsing table contains no conflicts. Hence the grammar is LL(1).

3(b) Describe the working procedure of recursive descent parsing and explain how grammar rules are implemented using functions. [7M]

Ans:

Recursive descent parsing is a top-down parsing technique in which every non-terminal is represented by a function.

Working:

1. Start symbol function is called.

2. Functions recursively call other functions based on grammar rules.

3. Input tokens are matched step-by-step.

Advantages:

- Simple implementation

- Easy to understand

Disadvantages:

- Cannot handle left recursive grammars directly.

Example:

For grammar:

E → T E'

E' → + T E' | ε

Recursive Descent Parser

```
void E()
{
   T();
   Eprime();
}

void Eprime()
{
   if(input == '+')
   {
      match('+');
      T();
      Eprime();
   }
   else
   {
      return;
   }
}
void T()
{
   if(input == id)
   {
      match(id);
   }
   else
   {
      error();
   }
```

```
}

void match(token)
{
   if(input == token)
      input = nextToken();
   else
      error();
}
```

4(a) Consider the grammar.

S-->S+S

S-->S*S

S --> id

Perform Shift Reduce parsing for input string "id + id + id". [7M]

Ans:

Step	Stack	Input	Action
1	$	id + id + id $	Shift id
2	$ id	+ id + id $	Reduce id → S
3	$ S	+ id + id $	Shift +
4	$ S +	id + id $	Shift id
5	$ S + id	+ id $	Reduce id → S
6	$ S + S	+ id $	Reduce S+S → S
7	$ S	+ id $	Shift +
8	$ S +	id $	Shift id
9	$ S + id	$	Reduce id → S
10	$ S + S	$	Reduce S+S → S
11	$ S	$	Accept

4(b) Explain YACC (Yet Another Compiler Compiler) and describe its role in generating parsers for a compiler. [7M]

Ans:

YACC stands for Yet Another Compiler Compiler.

Role:

Reads grammar rules written by the programmer.

Constructs parsing tables automatically.

Generates parser code in C language.

Detects syntax errors in source programs.

Structure of YACC Program

A YACC program has three sections:

1. Declarations Section

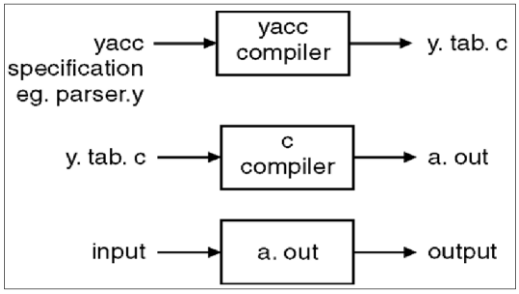
Contains token declarations and header files.

2. Rules Section

Contains grammar productions and semantic actions.

3. User Programs Section

Contains supporting C functions like main() and yyerror().



5(a) Construct CLR parsing table for the given context free grammar

S-->AA

A-->aA|b **[10M]**

Ans:

Grammar:

S → AA

A → aA | b

Steps:

1. Augment grammar.

2. Construct LR(1) items.

3. Compute closures and goto transitions.

4. Build ACTION and GOTO tables.

CLR parsing provides accurate parsing with minimal conflicts because it uses complete

LR(1) lookahead information.

Step 1: Augmented Grammar

Add new start symbol:

S′→SS'

Augmented grammar:

1. S′→S
2. S→AA
3. A→aA
4. A→b

Step 2: Canonical LR(1) Items

I_0

S′→.S, $

S→.AA, $

A→.aA, a/b

A→.b, a/b

Transitions:

- goto(I_0,S)=I_1
- goto(I_0,A)=I_2
- goto(I_0,a)=I_3
- goto(I_0,b)=I_4

I_1

S′→S., $ Accept state.

I_2

S→A.A, $

A→.aA, $

A→.b, $

Transitions:

- goto(I_2,A)=I_5
- goto(I_2,a)=I_6
- goto(I_2,b)=I_7

I_3

A→a.A, a/b

A→.aA, a/b

A→.b, a/b

Transitions:

- goto(I_3,A)=I_8
- goto(I_3,a)=I_3
- goto(I_3,b)=I_4

I_4

A→b., a/b

Reduce by:

A→b

I_5

S→AA., $

Reduce by:

S→AA

I_6

A→a.A, $

A→.aA, $

A→.b, $

Transitions:

- goto(I_6,A)=I_9
- goto(I_6,a)=I_6
- goto(I_6,b)=I_7

I_7

A→b., $

Reduce by:

A→b

I_8

A→aA., a/b

Reduce by:

A→aA

I_9

A→aA., $

Reduce by:

A→aA

Step 3: CLR Parsing Table

Productions:

1. S→AA
2. A→aA
3. A→b

State	a	b	$	S	A
I0	S3	S4		1	2
I1			Accept		
I2	S6	S7			5
I3	S3	S4			8
I4	R3	R3			
I5			R1		
I6	S6	S7			9
I7			R3		
I8	R2	R2			
I9			R2		

5(b) Explain Syntax Directed Definitions (SDD) and describe how they are used to associate semantic rules with grammar productions. [4M]

Ans:

Definition

An SDD is a grammar in which every grammar symbol has attributes and every production has semantic rules for computing attribute values.

Components of SDD

1. Attributes

Attributes store semantic information.

Two types:

(a) Synthesized Attributes

- Computed from child nodes.
- Information flows from bottom to top.

Example:

Expression value computation.

(b) Inherited Attributes

- Obtained from parent or sibling nodes.
- Information flows from top to bottom or sideways.

Example:

Type information passing.

2. Semantic Rules

Semantic rules define how attribute values are calculated for each production.

Example of SDD

Grammar:

E→E1+T

E→T

T→digit

Semantic Rules:

E.val=E1.val+T.val

E.val=T.val

T.val=digit.lexval

Here:

- val is a synthesized attribute.
- lexval is lexical value of digit.

How SDD Associates Semantic Rules with Productions

In SDD, each grammar production is attached with semantic rules.

Example:

A→B C

Semantic Rule:

A.val=B.val+C.val

During parsing:

1. Parser recognizes production.
2. Semantic rules are executed.
3. Attribute values are computed.
4. Semantic information is generated.

Thus syntax and semantics are processed together.

6(a) Write quadruple, triples and indirect triples for following expression:

(x + y) * (y + z) + (x + y + z) **[7M]**

Ans:

Expression:

(x + y) * (y + z) + (x + y + z)

Three Address Code:

t1=x+y

t2=y+z

t3=t1∗t2

t4=x+y

t5=t4+z

t6=t3+t5

Quadruples:

No	Operator	Argument1	Argument2	Result
1	+	x	y	t1
2	+	y	z	t2
3	*	t1	t2	t3
4	+	x	y	t4
5	+	t4	z	t5
6	+	t3	t5	t6

Triples:

No	Operator	Argument1	Argument2

No	Operator	Argument1	Argument2
0	+	x	y
1	+	y	z
2	*	(0)	(1)
3	+	x	y
4	+	(3)	z
5	+	(2)	(4)

Indirect Triples:

Pointer	Statement Number
P0	0
P1	1
P2	2
P3	3
P4	4
P5	5

6(b) Explain simple code generation algorithm for the following.

t = a-b

u = a-c

v = t+u **[7M]**

Ans:

Statements:

t = a - b

u = a - c

v = t + u

Statements	Code generated	Register Descriptors	Address Descriptors
		Registers empty	a:M; b:M; c:M; t:M; u:M; v:M
1. t = a - b	MOV a, R0 SUB b, R0	R0:t	a:M; b:M; c:M; t:R0; u:M; v:M
2. u = a - c	MOV a, R1	R0:t	a:M; b:M; c:M;

	SUB c, R1	R1:u	t:R0; u:R1; v:M
3. v = t + u	MOV R0, R2 ADD R1, R2	R0:t R1:u R2:v	a:M; b:M; c:M; t:R0; u:R1; v:R2

	MOV R0, t
	MOV R1, u
	MOV R2, v		

7(a) Give SDT scheme for Boolean expression using backpatching. Generate three address code for the Boolean expression a>b && c<d || e>f. [7M]

Ans:

Backpatching is used for generating code for Boolean expressions.

Expression:

a>b && c<d || e>f

Generated three-address code:

if a>b goto L1

goto L2

L1: if c<d goto L3

goto L2

L2: if e>f goto L3

goto L4

L3: TRUE

L4: FALSE

Backpatching fills jump addresses later when labels become known.

7(b) Compute the basic blocks for the given three address statements-

(1) PROD = 0

(2) I = 1

(3) T2 = addr(A) – 4

(4) T4 = addr(B) – 4

(5) T1 = 4 x I

(6) T3 = T2[T1]

(7) T5 = T4[T1]

(8) T6 = T3 x T5

(9) PROD = PROD + T6

(10) I = I + 1

(11) IF I <=20 GOTO (5) **[7M]**

Ans:

Basic block formation:

Leader statements:

1, 5

Basic Block B1:

(1) PROD = 0

(2) I = 1

(3) T2 = addr(A)-4

(4) T4 = addr(B)-4

Basic Block B2:

(5) T1 = 4 x I

(6) T3 = T2[T1]

(7) T5 = T4[T1]

(8) T6 = T3 x T5

(9) PROD = PROD + T6

(10) I = I + 1

(11) IF I<=20 GOTO(5)

B2 forms a loop block.

8(a) How does runtime environment manage memory during runtime? **[4M]**

Ans:

Runtime Environment and Memory Management

The runtime environment is responsible for managing memory and resources during program execution. It allocates memory to variables, functions, objects, and dynamically created data.

The runtime system ensures efficient execution of programs by organizing memory into different areas.

Memory Organization During Runtime

The memory of a program is generally divided into three main sections:

1. Static Area

- Stores global variables and static variables.
- Memory is allocated at compile time.
- Allocation remains throughout program execution.

Features

- Fixed size
- Faster access
- Used for permanent data

Example:

```
static int x;
global int y;
```

2. Stack Area

The stack stores activation records for function calls.

Each function call creates an activation record containing:

- Parameters
- Local variables
- Return address
- Temporary variables

When a function is called:

- Activation record is pushed onto stack.

When function returns:

- Activation record is popped from stack.

Features

- Dynamic allocation
- Supports recursion
- Follows LIFO order

3. Heap Area

Heap is used for dynamic memory allocation during execution.

Memory is allocated and deallocated explicitly by the programmer.

Example:

```
malloc(), calloc(), free()
```

Features

- Flexible memory usage
- Used for linked lists, trees, objects
- Slower than stack allocation

8(b) Describe the data structures used for symbol tables. How do different structures affect search and insertion operations? [10M]

Ans:

Common data structures:

1. Linear List:

   Simple but slow searching.

2. Linked List:

   Dynamic but sequential access.

3. Hash Table:

   Fast insertion and lookup using hashing.

   Average complexity: O(1)

4. Binary Search Tree:

   Ordered storage.

   Average complexity: O(log n)

Data Structure	Search Time	Use Case
Linear List	O(n)	Small/simple compilers
Hash Table	O(1)	Most widely used
BST	O(log n)	Ordered data

Hash tables are most commonly used in compilers because of fast access speed.

9(a) A program contains recursive and non-recursive functions. Explain how the runtime environment distinguishes between static and stack allocation for these cases. Why is stack allocation preferred for recursion? [7M]

Ans:

Static and Stack Allocation in Runtime Environment

The runtime environment manages memory allocation for functions during program execution. Depending on whether functions are recursive or non-recursive, it uses static allocation or stack allocation.

1. Static Allocation

In static allocation, memory for variables and functions is allocated at compile time and remains fixed throughout program execution.

Used For

- Non-recursive functions
- Global variables
- Static variables

Features

1. Memory size is fixed.
2. Allocation occurs only once.
3. Faster access.
4. Does not support multiple simultaneous function activations.

Limitation

If a function calls itself recursively, static allocation cannot create separate copies of local variables for each function call.

2. Stack Allocation

In stack allocation, memory is allocated dynamically during function calls using a stack.

Each function call creates a separate activation record containing:

- Parameters
- Local variables
- Return address
- Temporary variables

When the function returns, the activation record is removed from the stack.

Runtime Environment Distinction

For Non-Recursive Functions

The runtime environment may use static allocation because:

- Only one activation of the function exists at a time.
- Memory requirements are fixed.
- Separate copies of variables are not needed.

For Recursive Functions

The runtime environment uses stack allocation because:

- Multiple function calls can exist simultaneously.
- Each recursive call needs its own local variables and return address.
- Separate activation records must be maintained.

Why Stack Allocation is Preferred for Recursion

1. Supports Multiple Activations

Recursive calls require separate memory for each invocation.

Stack allocation creates separate activation records automatically.

2. Automatic Memory Management

Memory is allocated when function is called and deallocated when function returns.

3. Preserves Function State

Each recursive call stores:

- Local variables
- Parameters
- Return address

independently.

4. Efficient Execution

Stack operations (push and pop) are very fast and efficient.

Example

Consider recursive factorial function:

```
fact(n)
{
  if(n==1) return 1;
  return n * fact(n-1);
}
```

Each recursive call creates a new activation record in the stack until the base condition is reached.

9(b) A compiler needs fast access to identifiers during parsing. Compare how different data structures for symbol tables (like hash tables and trees) handle insertion and lookup. Which is more efficient in this scenario? [7M]

Ans:

Hash Tables:

- Fast insertion and lookup
- Average time complexity O(1)
- Best for compiler symbol tables

Trees:

- Maintain sorted order
- Lookup complexity O(log n)
- Useful when ordered traversal is needed

Feature	Hash Table	Tree
Insertion	O(1) average	O(log n) average
Lookup	O(1) average	O(log n) average
Ordering	Not maintained	Maintained
Speed	Faster	Slower
Collision Handling	Required	Not required

For compiler symbol tables, hash tables are more efficient because:

1. Compilers require very frequent lookup operations.
2. Hash tables provide constant-time access.
3. Parsing speed becomes faster.
4. Insertion and retrieval are efficient even for large programs.

Therefore, most compilers use **hash tables** for symbol table implementation.

Signature of the Faculty, CSE(CB)	**Signature of the HoD, CSE(CB)**