*"I am a GTU student (CSE, Semester 7). My subject is Artificial Intelligence (3170716) and my mid-sem exam syllabus is
Topics
1
Overview of the Compiler and its Structure:
Language processor, Applications of language processors, Definition-Structure-Working of compiler, the science of building compilers, Basic understanding of interpreter and assembler. Difference between interpreter and compiler. Compilation of source code into target language, Cousins of compiler, Types of compile

I will give you important questions from this syllabus. You have to give me answers in simple Indian English, written in a way that is easy to understand and good for writing in exams. The answer should be correct, clear, and to the point (not too short, not too long). Where needed, add examples or diagrams (in text form). Avoid very hard English words."*

wait not yet answer

Got it 👍
I will wait for you to share the important questions. Once you give them, I'll write the answers in **simple Indian English**, suitable for GTU exams, with clear points, short examples, and neat formatting (good for writing in exam papers).

Would you like me to also prepare **short notes style + full answer style** for each question? That way you can revise quickly before exam and also write proper answers.

1 What is the difference between compiler and interpreter? 3
2 Explain analysis phase of the source program with example 4
3 What is a pass in a compiler? What is the effect of reducing the number of passes?
3
4 Explain buffer pairs and sentinels. 4
5 Translate the arithmetic expression a*-(b+c) into 1. Syntax tree 2. Postfix notation 3. Three address code
3
6 List the cousins of compiler and explain the role of any one of them. 3
7 Write a brief note on input buffering techniques. 4
8 Explain front end and back end of compiler in detail. 7
9 Define cross compiler, token and handle. 3
10 Explain phases of compiler with example. 7

11 What is symbol table? For what purpose , compiler uses symbol table? 3
12 Explain various data structures used in symbol table management. 7

Perfect 👍 Thanks for sharing the important questions.
I'll now prepare answers for you in **simple Indian English**, exam-friendly style (GTU CSE, Semester 7, AI subject – Compiler overview unit).
Each answer will be clear, to the point, and easy to write in exam.

Q1) Difference between Compiler and Interpreter (3 marks)

Compiler

- Translates the whole source program into machine code at once.
- Output is an **object code** or **executable file**.
- Errors are shown **after full compilation**.
- Execution is faster because program is already converted into machine code.
- Example: C, C++ use compiler.

Interpreter

- Translates program **line by line** and executes immediately.
- No separate object code is created.
- Errors are shown **immediately after the faulty line**.
- Execution is slower because translation happens every time.
- Example: Python, JavaScript use interpreter.

Diagram (in text form):

```mathematica
Compiler: Source Code → Compiler → Machine Code → Run
Interpreter: Source Code → Interpreter → Run (line by line)
```

👉 Do you want me to continue and prepare **all answers (Q2–Q12)** in this same exam-ready format, or should I give you answers one by one so you can revise step by step?

yes

Great 👍 Let's do it step by step. I'll continue with each question in **GTU exam-friendly format**.

Q2) Explain analysis phase of the source program with example (4 marks)

The **analysis phase** of a compiler reads the source program and breaks it into meaningful parts for translation.
It checks the correctness of the program.
This phase is also called the **front-end** of compiler.

The analysis phase is divided into 3 main parts:

1. **Lexical Analysis (Scanning)**
    - Reads characters and groups them into **tokens** (like keywords, identifiers, operators).
    - Example: For `a = b + 5;`
      Tokens: `id(a), =, id(b), +, number(5), ;`
2. **Syntax Analysis (Parsing)**
    - Checks grammar rules (structure of program).
    - Builds a **parse tree**.
    - Example: `a = b + 5;` → Parser checks that it follows grammar of assignment.
3. **Semantic Analysis**
    - Checks meaning (types, declarations, compatibility).
    - Example: If `b` is declared as float and `a` as int, it checks type conversion.

Example (step by step for `a = b + 5;`):

```python
Source code: a = b + 5;

Lexical Analysis → Tokens: id(a), =, id(b), +, number(5), ;
Syntax Analysis → Parse tree for assignment
Semantic Analysis → Type checking (int = int + int)
```

✅ This completes Q2.

Do you want me to continue with **Q3 (What is a pass in compiler?)** now?

next 3

Alright 👍 Let's move to the next one.

Q3) What is a pass in a compiler? What is the effect of reducing the number of passes? (3 marks)

Pass in Compiler:

- A **pass** means **one complete scan/processing of the source program (or intermediate code) by the compiler**.

- In each pass, the compiler performs a specific set of tasks.
- Example:
    - **First pass** → builds symbol table, checks syntax.
    - **Second pass** → generates code.

Effect of reducing number of passes:

- **Advantages:**
    - Less memory use.
    - Faster compilation (because fewer scans).
- **Disadvantages:**
    - Compiler design becomes more **complex**.
    - Optimization quality may be reduced.

Summary:

- Pass = one complete traversal of source program.
- Reducing passes saves **time** and **space**, but makes compiler design harder.

✅ Q3 done.

Shall I continue with **Q4 (Buffer pairs and Sentinels)?**

Q4

Good 👍 Let's write Q4 in exam-ready format.

Q4) Explain buffer pairs and sentinels. (4 marks)

When compiler reads the **source code** from disk, it must bring it into **memory** in blocks for processing. To make input reading **fast and efficient**, two techniques are used: **buffer pairs** and **sentinels**.

---

1. Buffer Pairs (Double Buffering)

- Here, the input is divided into **two buffers** of equal size.
- While the compiler is processing one buffer, the other buffer is filled with new input from disk.
- This avoids delay in reading characters.
- Used in **lexical analysis**.

Diagram (text form):

```lua
 -------------------------------
 | Buffer 1 | Buffer 2 |
 -------------------------------
 | being used | being filled |
```

---

2. Sentinels

- A **sentinel** is a special character (like `EOF` or `#`) placed at the end of each buffer.
- It tells the compiler that the buffer has ended, so no need to check for boundary each time.
- This reduces **extra comparisons** and makes scanning faster.

Example:

```css
Buffer: [a][=][b][+][5][;][EOF]
```

✅ Summary:

- **Buffer pairs** → use two buffers to avoid delay.
- **Sentinels** → use a special end marker to avoid extra checking.

Do you want me to move to *Q5 (Translate expression a-(b+c))** next?

Q5

Perfect 👍 Let's do Q5 carefully, step by step.

Q5) Translate the arithmetic expression `a * -(b + c)` into:

1. **Syntax Tree**
2. **Postfix Notation**
3. **Three-Address Code**

---

1. Syntax Tree

The syntax tree shows the hierarchical structure:

```markdown
          (*)
         /   \
       a     ( - )
              |
             ( + )
            /     \
           b       c
```

---

2. Postfix Notation (Reverse Polish Notation)

- In postfix, operators come **after operands**.
- Expression: `a * -(b + c)`
- Postfix:

```
css

b c + - a *
```

---

3. Three-Address Code (TAC)

- Break the expression into small steps with **temporary variables**.

Steps:

```
ini

t1 = b + c
t2 = - t1
t3 = a * t2
```

✅ Final Answer:

- Syntax tree: (as shown)
- Postfix: `b c + - a *`
- TAC:

```
ini

t1 = b + c
t2 = - t1
t3 = a * t2
```

Do you want me to continue with **Q6 (Cousins of Compiler)**?

Q6

Alright 👍 Let's write Q6 in a neat exam format.

Q6) List the cousins of compiler and explain the role of any one of them. (3 marks)

Cousins of Compiler:
Other language processors related to compiler are called **cousins of compiler**. They are:

1. **Preprocessor**
2. **Assembler**
3. **Linker**
4. **Loader**

Roles (choose any one in exam):

1. **Preprocessor**
    - Works before compilation.
    - Handles macros, header files, conditional compilation.
    - Example: In C, `#include <stdio.h>` is managed by preprocessor.
2. **Assembler**
    - Converts **assembly code** into **machine code**.
    - Input: Assembly language program
    - Output: Object code
3. **Linker**
    - Combines object code with required **library files**.
    - Produces a single executable file.
4. **Loader**
    - Loads the executable file into **main memory** for execution.

✅ **Summary Answer:**
Cousins of compiler are: **Preprocessor, Assembler, Linker, Loader**.
For example, **Linker** combines object code with libraries to make a single executable program.

Shall I move to **Q7 (Input buffering techniques)?**

Q7

Good 👍 Let's write Q7 in simple exam style.

Q7) Write a brief note on input buffering techniques. (4 marks)

Need of Input Buffering:

- Compiler reads source code character by character.
- Reading directly from disk is slow.
- To make it fast, **input buffering** is used.

Techniques of Input Buffering:

1. **Single Buffering**
    - Only one buffer is used.
    - While processing, if buffer becomes empty, compiler waits for refill.
    - Simple but **slow**.
2. **Double Buffering (Buffer Pairs)**
    - Two buffers of equal size are used.
    - While one buffer is processed, the other is filled with input.
    - No waiting time → **faster**.
3. **Sentinel Technique**
    - A special marker (like `EOF`) is placed at the end of buffer.
    - This avoids checking for buffer boundary at each step.
    - Improves efficiency of lexical analysis.

Diagram (text form for Double Buffering):

```lua
 -------------------------------
 | Buffer 1 | Buffer 2 |
 -------------------------------
 | in use   | being filled |
```

✅ **Summary:**
Input buffering reduces **I/O time** and makes source program scanning **fast and efficient**.
Main techniques: **Single buffer, Double buffer, Sentinel method**.

Do you want me to continue with **Q8 (Front end and Back end of compiler)?**

Q8

Alright 👍 Here's Q8 in exam-ready format.

Q8) Explain front end and back end of compiler in detail. (7 marks)

A compiler is mainly divided into two parts: **Front End** and **Back End**.

---

1. Front End of Compiler

- Deals with **analysis** of source program.
- It is **machine independent** (not dependent on target machine).
- Tasks:
  1. **Lexical Analysis** → breaks program into tokens.
  2. **Syntax Analysis (Parsing)** → checks grammar, builds parse tree.
  3. **Semantic Analysis** → checks meaning (type checking, declarations).
  4. **Intermediate Code Generation** → generates an intermediate representation.

👉 Example: For `a = b + 5;`

- Tokens: `id(a), =, id(b), +, number(5)`
- Parse tree is made
- Types checked (`int = int + int`)
- Intermediate code: `t1 = b + 5`, `a = t1`

---

2. Back End of Compiler

- Deals with **synthesis** of target code.
- It is **machine dependent**.
- Tasks:
  1. **Code Optimization** → improves efficiency (reduce instructions, save memory).
  2. **Target Code Generation** → converts intermediate code to machine code.
  3. **Register Allocation** → assigns CPU registers.

👉 Example: From intermediate code →

```ini
t1 = b + 5
a = t1
```

The back end produces **machine instructions** like:

```css
LOAD R1, b
ADD R1, 5
STORE a, R1
```

Diagram (Text form):

```mathematica
 Source Program
      |
   Front End  (Analysis)
      |
Intermediate Code
      |
   Back End  (Synthesis)
      |
 Target Code
```

✅ **Summary:**

- **Front End** = Analysis phase (machine independent).
- **Back End** = Code generation phase (machine dependent).

Shall I move to **Q9 (Define cross compiler, token and handle)?**

Q9

Good 👍 Let's write Q9 in a crisp exam format.

Q9) Define cross compiler, token and handle. (3 marks)

1. Cross Compiler

- A compiler that runs on one machine but produces code for another machine.
- Example: A cross compiler running on **Windows** but generating code for **embedded ARM processor**.

---

2. Token

- Smallest unit of a program that has **meaning** for the compiler.
- Types: keywords, identifiers, operators, constants, punctuation.
- Example: For `a = b + 5;` → Tokens: `id(a), =, id(b), +, number(5), ;`

---

3. Handle

- A **substring of the right-hand side** of a production in a grammar which matches a part of the sentential form and whose reduction represents one step in a **bottom-up parsing**.
- In short → Handle is the part of the string to be **reduced** during parsing.

Example:
Grammar: `E → E + T | T`
String: `id + id`
Here, `id` is a handle for reduction to `T`.

✅ **Summary:**

- Cross compiler → generates code for another machine.
- Token → smallest meaningful unit.
- Handle → substring reduced in bottom-up parsing.

Shall I continue with **Q10 (Phases of compiler with example)?**

Q10

Perfect 👍 This one is an important 7-mark question, so let's write it neatly.

Q10) Explain phases of compiler with example. (7 marks)

A compiler works in different **phases** to convert **source code** into **target code**.
These phases are usually grouped into **front end** and **back end**.

Phases of Compiler:

1. **Lexical Analysis (Scanning)**
    - Breaks source code into **tokens**.
    - Example: `a = b + 5;` → Tokens: `id(a), =, id(b), +, number(5), ;`
2. **Syntax Analysis (Parsing)**
    - Checks grammar and builds **parse tree**.
    - Example: Verifies `a = b + 5;` follows assignment grammar.
3. **Semantic Analysis**
    - Checks **meaning** (types, declarations, scope).
    - Example: Ensures `a` and `b` are declared and types are compatible.
4. **Intermediate Code Generation**
    - Produces a machine-independent code.
    - Example:

```ini
t1 = b + 5
a = t1
```

5. **Code Optimization**
    - Improves efficiency of intermediate code (speed, memory).
    - Example: If `5` is constant, reuse it instead of recomputing.
6. **Code Generation**
    - Converts optimized code into **target machine code**.
    - Example:

```css
LOAD R1, b
ADD R1, 5
STORE a, R1
```

7. **Symbol Table Management** (works throughout)
    - Stores identifiers, data types, scope, etc.
8. **Error Handling** (works throughout)
    - Detects and reports lexical, syntax, and semantic errors.

Diagram (Text form):

```css
Source Code
   ↓
Lexical Analysis → Syntax Analysis → Semantic Analysis
   ↓                  ↓                  ↓
Tokens             Parse Tree       Annotated Tree
   ↓
Intermediate Code Generation
   ↓
Code Optimization
   ↓
Code Generation
   ↓
Target Machine Code
```

✅ **Summary:**
The compiler works in **phases**: lexical, syntax, semantic, intermediate code, optimization, and code generation.
Symbol table and error handling support all phases.

Do you want me to continue with **Q11 (Symbol Table)**?

Q11

Alright 👍 Here's Q11 in neat exam style.

Q11) What is symbol table? For what purpose, compiler uses symbol table? (3 marks)

Symbol Table:

- A **data structure** used by compiler to store information about the **identifiers** (variables, functions, objects, etc.) in a program.
- It works like a **dictionary** for the compiler.

Contents of Symbol Table:

- Name of identifier (variable/function/class).
- Data type (int, float, etc.).
- Scope (local/global).
- Memory location / address.
- Other attributes (size, parameters for functions).

Purpose of Symbol Table in Compiler:

1. Helps in **semantic analysis** (type checking, declaration checking).
2. Supports **intermediate code generation** (mapping variables to addresses).
3. Useful for **optimization** (reuse variables, avoid duplicates).
4. Needed in **code generation** (deciding memory layout).

✅ **Summary:**
Symbol table is a data structure that stores all information about identifiers.
Compiler uses it for **declarations, type checking, memory allocation, and code generation**.

Do you want me to move to the last one, **Q12 (Data structures used in symbol table management)?**

Q12

Great 👍 Let's complete the last one.

Q12) Explain various data structures used in symbol table management. (7 marks)

The **symbol table** is a data structure used by the compiler to store information about identifiers. To implement it efficiently, different data structures are used.

---

1. Linear List

- Simple array or linked list.
- Each entry stores identifier name + attributes.
- **Advantage:** Easy to implement.
- **Disadvantage:** Searching takes **O(n)** time (slow for large programs).

---

2. Hash Table

- Uses a **hash function** to map identifier names to table index.
- Provides **fast searching and insertion** (average O(1)).
- **Advantage:** Very efficient.
- **Disadvantage:** Collisions may occur, need extra handling.

---

3. Binary Search Tree (BST)

- Identifiers are stored in a tree structure.
- Searching, insertion, deletion can be done in **O(log n)** time (if balanced).
- **Advantage:** Ordered structure, useful for range queries.
- **Disadvantage:** Can become unbalanced (O(n) worst case).

---

4. Trie (Prefix Tree)

- Characters of identifiers are stored as paths in a tree.
- Efficient for storing large set of identifiers with common prefixes.
- **Advantage:** Fast lookup when many identifiers share prefixes.
- **Disadvantage:** More memory usage.

---

5. Self-Organizing List

- Frequently used identifiers are moved closer to the front.
- Improves average search time.

Diagram (text form for Hash Table):

```pgsql
Hash Function → Index
 ---------------------------
 | 0 | id1 → info          |
 | 1 | id2 → info          |
 | 2 | id3 → info          |
 ---------------------------
```

✅ **Summary:**
Common data structures for symbol table are:

- **Linear List, Hash Table, Binary Search Tree, Trie, Self-Organizing List**.
- **Hash Table** is the most widely used in compilers because it provides **fast lookup and insertion**.

✨ Now all **Q1 to Q12** are covered in exam-friendly style.

Do you want me to also prepare a **1-page quick revision sheet (only key points)** for last-minute reading before exam?

yes

Perfect 👍 Let me prepare a **Quick Revision Sheet** for all Q1–Q12.
This will be in **short notes style**, so you can revise in **5–10 minutes** before exam.

📘 Compiler Overview – Quick Revision (GTU CSE, Sem 7)

Q1) Compiler vs Interpreter

- **Compiler** → Translates whole program → object code → faster execution.
- **Interpreter** → Line by line → no object code → slower execution.
- Example: C (compiler), Python (interpreter).

Q2) Analysis Phase

- **Lexical Analysis** → tokens.
- **Syntax Analysis** → grammar check, parse tree.
- **Semantic Analysis** → meaning, type check.
  Example: `a = b + 5;` → tokens → parse tree → type check.

Q3) Pass in Compiler

- **Pass** = one complete scan of program.
- Fewer passes → faster, less memory.
- But design becomes complex.

Q4) Buffer Pairs & Sentinels

- **Buffer pairs** → double buffer, one in use, one being filled.
- **Sentinel** → special marker (EOF) to avoid boundary check.

Q5) Expression `a * -(b + c)`

- **Syntax Tree:**

```
r

   (*)
  /   \
 a    (-)
        |
       (+)
      /   \
     b     c
```

- **Postfix:** `b c + - a *`
- **TAC:**

```
ini

t1 = b + c
t2 = -t1
t3 = a * t2
```

Q6) Cousins of Compiler

- **Preprocessor** → handles macros, includes.
- **Assembler** → converts assembly → machine code.
- **Linker** → combines object + libraries.
- **Loader** → loads program into memory.

Q7) Input Buffering

- **Single buffer** → slow.
- **Double buffer** → fast (overlap read & process).
- **Sentinel** → special marker avoids extra checks.

Q8) Front End & Back End

- **Front End (machine independent)**: Lexical → Syntax → Semantic → Intermediate code.
- **Back End (machine dependent)**: Optimization → Code generation → Register allocation.

Q9) Definitions

- **Cross Compiler** → makes code for other machine.
- **Token** → smallest unit (id, keyword, operator).
- **Handle** → substring reduced in bottom-up parsing.

Q10) Phases of Compiler

1. Lexical analysis → tokens
2. Syntax analysis → parse tree
3. Semantic analysis → meaning check
4. Intermediate code generation
5. Code optimization
6. Code generation
   Supporting: Symbol table + Error handling

Q11) Symbol Table

- Stores info of identifiers: name, type, scope, address.
- Purpose: type checking, memory allocation, optimization, code generation.

Q12) Data Structures for Symbol Table

- **Linear List** → simple, slow.
- **Hash Table** → fast lookup (most used).
- **BST** → ordered, log n search.
- **Trie** → prefix tree, efficient for many identifiers.
- **Self-organizing list** → frequently used in front.

✅ **Tip:** For long answers (7 marks), write phases in order + draw diagram.
✅ For short answers (3–4 marks), write **definition + 2–3 points + example**.

Would you like me to also make a **diagram-only cheat sheet** (all compiler phases + front/back end + cousins + buffers) in simple text diagrams for quick last-minute memory boost?