USN | | | | | | | | | | |

43

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU
ACCREDITED WITH A++ GRADE BY NAAC

Internal Assessment Test 2 – May 2025

Sub:	Compiler Design					Sub Code:	BCS613C	Branch:	CSE
Date:	24.05.2025	Duration:	90 mins	Max Marks:	50	Sem / Sec:	VI/ A, B, C	OBE	

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	a) Summarize the steps that are involved in the concept of LL(1) grammar.	[5]	CO3	L2
	b) Explain FIRST() and FOLLOW() sets to be constructed with the given grammar. $E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow (E) \mid id$	[5]	CO3	L2
2	a)Explain bottom-up parsing and the concept of shift-reduce parsing	[5]	CO3	L2
	b) Explain handle and handle pruning in bottom up parsing	[5]	CO3	L2
3	Illustrate algorithm used for eliminating left recursion. Eliminate left recursion from the grammar: $S \rightarrow Aa \mid b \quad A \rightarrow Ac \mid Sd \mid a.$	[10]	CO3	L3

Eliminate Left Recursion

- LL(1) grammar should be free from **left recursion**.
- Example:
- $E \rightarrow E + T \mid T$ (left recursive)

is rewritten as:

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

2. Left Factoring

- If a non-terminal has two or more productions beginning with the same prefix, factor them to remove ambiguity.
- Example:
- $A \rightarrow \alpha\beta1 \mid \alpha\beta2$

becomes:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta1 \mid \beta2$

3. Compute FIRST sets

- FIRST(X) = set of terminals that can appear first in some string derived from X.
- Rules:
  - If X is a terminal → FIRST(X) = {X}
  - If $X \rightarrow \varepsilon$ → include ε in FIRST(X)
  - If $X \rightarrow Y1Y2\ldots Yn$ → add FIRST(Y1). If Y1 can derive ε, then also add FIRST(Y2), and so on.

4. Compute FOLLOW sets

- FOLLOW(A) = set of terminals that can appear immediately to the right of A in some sentential form.

- Rules:
    - Place $ (end marker) in FOLLOW(S) for the start symbol S.
    - If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in FOLLOW(B).
    - If there is a production A → αB or A → αBβ where ε ∈ FIRST(β), then everything in FOLLOW(A) is placed in FOLLOW(B).

5. Construct the LL(1) Parsing Table

- For each production A → α:
    1. For each terminal **a ∈ FIRST(α)**, put A → α in M[A, a].
    2. If ε ∈ FIRST(α), then for each **b ∈ FOLLOW(A)**, put A → α in M[A, b].

6. Check for Conflicts

- If **any cell** of the parsing table contains more than one production, the grammar is **not LL(1)**.
- A grammar is LL(1) **iff**:
    - It is **unambiguous**,
    - **No left recursion**,
    - **Left factored**,
    - Parsing table has **no multiple entries** in any cell.

---

1. FIRST

👉 **Definition**:
FIRST(X) = the set of terminals that can appear first in some string derived from X.

- If **X is a terminal** → FIRST(X) = {X}
- If **X → ε** → add ε to FIRST(X)
- If **X is a non-terminal**:

- For a production X → Y1Y2…Yn:
  - Add FIRST(Y1) to FIRST(X).
  - If Y1 can produce ε, then also add FIRST(Y2), and so on.
  - If all Y1...Yn can derive ε, then add ε to FIRST(X).

🔷 **Example:**
Grammar:

E → T E'

E' → + T E' | ε

T → F T'

T' → * F T' | ε

F → (E) | id

- FIRST(F) = { (, id }
- FIRST(T) = { (, id } (because T → F …)
- FIRST(E) = { (, id }
- FIRST(E') = { + , ε }
- FIRST(T') = { * , ε }

---

2. FOLLOW

👉 **Definition**:
FOLLOW(A) = the set of terminals that can appear **immediately to the right of A** in some sentential form.

- If S is the **start symbol** → put $ in FOLLOW(S) (end of input).
- If there is a production A → αBβ:
  - Everything in FIRST(β) (except ε) goes to FOLLOW(B).
- If there is a production A → αB **or** A → αBβ where ε ∈ FIRST(β):
  - Everything in FOLLOW(A) goes to FOLLOW(B).

🔷 **Example:** (same grammar)

Start symbol = E, so $ ∈ FOLLOW(E).

- From E → T E' → FOLLOW(T) includes FIRST(E') = { + , ε }.
    - So FOLLOW(T) = { + ,) , $ } (because ε ∈ FIRST(E'), so FOLLOW(E) ⊆ FOLLOW(T)).
- From E' → + T E' → FOLLOW(T) includes FIRST(E') = { + , ε }.
- From F → (E) → FOLLOW(E) includes).

Final sets:

- FOLLOW(E) = {) , $ }
- FOLLOW(E') = {) , $ }
- FOLLOW(T) = { + ,) , $ }
- FOLLOW(T') = { + ,) , $ }
- FOLLOW(F) = { * , + ,) , $ }

Bottom-Up Parsing (Shift–Reduce Parsing)

✅ **In short**:
Bottom-up parsing starts from the **input string (tokens)** and works **backwards toward the start symbol** by **reducing** substrings to non-terminals until the start symbol is reached.

Steps in Bottom-Up Parsing

1. **Input**: Sequence of tokens from the lexical analyzer.
2. **Shift**: Move (shift) the next input symbol onto the parsing stack.
3. **Reduce**: If the symbols on top of the stack match the **right-hand side (RHS)** of a production, replace them with the corresponding **non-terminal (LHS)**.
4. **Accept**: If the stack contains only the start symbol and the input is fully read, the string is **accepted**.
5. **Error**: If no valid shift/reduce is possible → parsing **fails**.

Example

Grammar:

E → E + T | T

T → T * F | F

F → (E) | id

Input:

id + id * id

Parsing Process (Shift–Reduce):

- Stack: [ ] , Input: id + id * id $
- **Shift**: [id]
- **Reduce**: [F] (since F → id)
- **Reduce**: [T] (since T → F)
- **Reduce**: [E] (since E → T)
- **Shift**: [E, +]
- **Shift**: [E, +, id]
- **Reduce**: [E, +, F]
- **Reduce**: [E, +, T]
- **Shift**: [E, +, T, *]
- **Shift**: [E, +, T, *, id]
- **Reduce**: [E, +, T, *, F]
- **Reduce**: [E, +, T] (since T → T * F)
- **Reduce**: [E] (since E → E + T)
- **Accept** ✅

Types of Bottom-Up Parsers

1. **Operator-Precedence Parser**
2. **LR Parsers** (most powerful & widely used)
    - **SLR(1)** (Simple LR)
    - **Canonical LR(1)**

- **LALR(1)** (Lookahead LR, most common in compilers like YACC/Bison)

Comparison with Top-Down Parsing

- **Top-Down (LL)**: starts from start symbol → derives string.
- **Bottom-Up (LR)**: starts from input string → reduces to start symbol.

👉 Do you want me to also **draw a shift–reduce parsing table** for this example so you see the moves step by step (like in exam-style solutions)?

Definition:
Pruning means **cutting off unnecessary parts of a search tree or grammar derivation** to reduce complexity, memory, and time.
It's like trimming branches of a tree that will never lead to the desired solution.

---

1. Pruning in Parsing / Compilers

- In **parsing**, pruning refers to **ignoring invalid or redundant parse trees**.
- Example: In **ambiguous grammar**, multiple parse trees may be generated → pruning removes the ones that **don't fit the parsing strategy** (e.g., LL(1) or LR(1) table conflicts).
- **Error pruning** is used to discard impossible parsing paths early.

---

2. Pruning in AI (Search Algorithms)

- In **search trees** (like minimax in game playing):
  - **Alpha-Beta Pruning** removes branches that will **never affect the final decision**, so fewer nodes are explored.
  - Example: In chess AI, it avoids checking useless moves.

---

3. Pruning in Machine Learning

- In **decision trees** (like ID3, C4.5, CART):

- Pruning removes branches that give **little or no improvement** in prediction accuracy.
- Types:
  - **Pre-pruning (early stopping):** Stop splitting when further splits are not significant.
  - **Post-pruning:** Build the full tree, then remove weak branches.

Left Recursion in Grammar

👉 A grammar is **left recursive** if it has a non-terminal A such that:

$A \Rightarrow^+ A\alpha$

That means, A can derive itself as the **leftmost symbol**.

Example:

$E \rightarrow E + T \mid T$

Here $E \rightarrow E + T$ is **left recursive**.

Algorithm to Eliminate Immediate Left Recursion

For each non-terminal A with productions:

$A \rightarrow A\alpha1 \mid A\alpha2 \mid \ldots \mid A\alpha m \mid \beta1 \mid \beta2 \mid \ldots \mid \beta n$

- Where:
  - Aαi are **left recursive** (start with A)
  - βj are **non-left recursive** (don't start with A)

We replace them with:

$A \rightarrow \beta1A' \mid \beta2A' \mid \ldots \mid \beta nA'$

$A' \rightarrow \alpha1A' \mid \alpha2A' \mid \ldots \mid \alpha mA' \mid \varepsilon$

Step-by-Step Example

Grammar:

E → E + T | T

- Here:
    - Left recursive part = E + T (Aα form)
    - Non-left recursive part = T (β form)

Apply algorithm:

E → T E'

E' → + T E' | ε

✅ Now grammar is **free from left recursion**.

Algorithm to Eliminate General Left Recursion (Multiple Non-terminals)

Sometimes indirect left recursion exists, like:

A → Bα

B → Aβ

General Algorithm (order the non-terminals: A1, A2, …, An)

For i = 1 to n:

For each production of Ai → Ajγ where j < i:

- Replace Aj with its productions.

Eliminate **immediate left recursion** on

LR(0) Items

👉 An **LR(0) item** is just a grammar production with a **dot (•)** somewhere in the RHS,showing how much of the input has been "seen" so far.

Example: For production A → XYZ, the LR(0) items are:

When a compiler translates source code → machine code, it often uses Intermediate Representations (IR) like:

Quadruples

Triples

Indirect Triples

These are mainly used for representing three-address code (TAC) operations.

---

1. Quadruple (Quad) Representation

👉 A quadruple has 4 fields:

(op, arg1, arg2, result)

op → operator (e.g., +, -, *, /)

arg1 → first operand

arg2 → second operand

result → location (temporary variable) where result is stored

✅ Example

For expression: x = a + b * c

Three-address code (TAC):

t1 = b * c

t2 = a + t1

x = t2

Quadruples:

(*, b, c, t1)

(+, a, t1, t2)

(=, t2, -, x)

---

2. Triple Representation

👉 A triple has 3 fields:

(op, arg1, arg2)

Result is not stored in a separate temporary variable.

Instead, it is referred to by its position (index) in the table.

✅ Example (same expression x = a + b * c)

Triples:

0: (*, b, c)

1: (+, a, (0)) ← refers to result of instruction 0

2: (=, (1), x)

Here:

Instruction 0 = b * c

Instruction 1 = a + result_of(0)

Instruction 2 = assign result_of(1) to x

---

3. Indirect Triples

👉 In indirect triples, we maintain a pointer table (or index table) that refers to triples.

This allows reordering of instructions easily (helpful in optimization).

✅ Example

Pointer Table:

0 → statement 2

1 → statement 0

2 → statement 1

Triples (same as before):

0: (*, b, c)

(+, a, (0))

(=, (1), x)

ut execution order is given by the pointer table, not the fixed order.

A •XYZ

A → X•YZ

A → XY•Z

A → XYZ•

If the dot is **before a non-terminal**, then closure rules apply (we expand).

If the dot is at the **end**, it means a **reduction** can be applied.

Steps to Find LR(0) Items

Augment the grammar

Add a new start symbol S' → S.

This helps mark acceptance.

Construct items

For each production, place the dot • in all possible positions.

Compute Closure

If an item has a dot before a non-terminal, say A → α • B βthen add **all productions of B** with the dot at the beginning (B → •γ)
to the closure set.

Repeat until no new items can be added.

Compute GOTO

From a set of items I, if the dot is before a symbol X,
then GOTO(I, X) gives another set of items where the dot is moved past X.

Essentially, it's like shifting the dot across X.

Example

Grammar:

E → E + T | T

T → T * F | F

F → (E) | id

Step 1: Augment

E' → E

Step 2: List LR(0) items

For each production, put the dot in all positions:

E' → •E

E' → E•

E → •E + T

E → E • + T

E → E + •T

E → E + T•

E → •T

E → T•

T → •T * F

T → T • * F

T → T * •F

T → T * F•

T → •F

T → F•

F → •(E)

F → (•E)

F → (E•)

F → (E)•

F → •id

F → id•

✅ These are the **LR(0) items** for the grammar.

Next steps (if we want the full **canonical collection of LR(0) sets**):

Start with I0 = closure({E' → •E})

Apply closure and GOTO repeatedly to construct **all item sets (states)** for the DFA usd in LR parsing.

Whn a compiler translates source code → machine code, it often use**sIntermediate Representations (IR)** like:

Quadruples

Triples

Indirect Triples

These are mainly used for representing **three-address code (TAC)** operations.

---

1. Quadruple (Quad) Representation

👉 A quadruple has **4 fields**:

(op, arg1, arg2, result)

op → operator (e.g., +, -, *, /)

arg1 → first operand

arg2 → second operand

result → locaion (temporary variable) where result is stored

✅ **Example**For expression: x = a + b * c

Three-address code (TAC):

t1 = b * c

t2 = a + t1

x = t2

Quadruples:

(*, b, c, t1)

(+, a, t1, t2)

(=, t2, -, x)

---

2. Triple Representation

👉 A triple has **3 fields**:

(op, arg1, arg2)

Result is **not stored in a separate temporary variable**.

Instead, it is referred to by its **position (index)** in the table.

✅ **Example** (same expression x = a + b * c)

Triples:

0: (*, b, c)

1: (+, a, (0)) ← refers to result of instruction 0

2: (=, (1), x)

Here:

Instruction 0 = b * c

Instruction 1 = a + result_of(0)

Instruction 2 = assign result_of(1) to x

---

3. Indirect Triples

👉 In indirect triples, we maintain a **pointer table** (or index table) that refers to triples.

This allows **reordering of instructions** easily (helpful in optimization).

✅ **Example**

Pointer Table:

0 → statement 2

→ statement 0

→ statement 1

Triples (same as before):

: (*, b, c)

(+, a, (0))

: (=, (1), x)

execution order is given by the **pointer table**, not the fixed order.

ariants of Syntax Tree

When we parse a program, the compiler generates different tree structures as intermediate representations. These include:

1. Concrete Syntax Tree (Parse Tree)

Also called derivation tree.

Represents the complete structure of a program according to the grammar.

Contains all grammar symbols (both terminals and non-terminals).

Shows step-by-step derivation.

✅ Example for expression: a + b * c

Root → E

Expands according to grammar: E → E + T, T → T * F, etc.

Very detailed, but often too big for optimization.

2. Abstract Syntax Tree (AST)

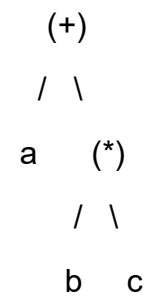
A simplified version of the parse tree.

Keeps only the essential structure (operators & operands).

Grammar details (like E, T, F, parenthesis rules) are removed.

✅ Example for a + b * c

AST would look like:



Much more compact → used for semantic analysis & optimization.

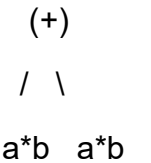
3. Directed Acyclic Graph (DAG) for Expressions

Variant of AST where common sub-expressions are shared.

Avoids duplication of repeated expressions → helps in optimization.

✅ Example: a * b + a * b

AST:



DAG:

(+)

/ \

(a*b) same node

Only one node for a * b, both branches refer to it.

4. Syntax Directed Translation Trees

Extended AST with annotations or attributes (like type info, intermediate code).

Each node may store extra info needed for code generation.

📌 Summary

Variant	Key Feature
Parse Tree	Shows complete grammar derivation (too detailed).
Abstract Syntax Tree (AST)	Simplified version, only essential structure.
DAG	AST with common sub-expression elimination.
SDT Tree	AST with attributes for translation/code generation.

DAG for Expressions

👉 A DAG is a compact representation of an expression, similar to an AST, but with **sharing of common sub-expressions**.

- **AST** → duplicates sub-expressions.
- **DAG** → merges identical sub-expressions.

Steps to Construct DAG

Given an expression:

Example: a + a * (b - c) + (b - c) * d

Step 1: Identify Sub-Expressions

- (b - c) occurs twice.
- a * (b - c) and (b - c) * d are distinct.

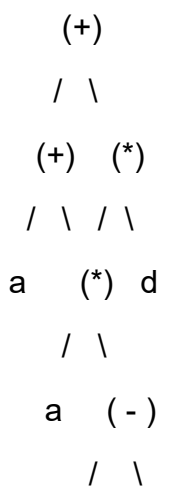
Step 2: Build Nodes

- Leaf nodes → operands (a, b, c, d).
- Interior nodes → operators (+, -, *).

Step 3: Merge Common Sub-Expressions

- For repeated (b - c), create **only one node** and share it.

Step 4: Construct DAG



b c

- (b - c) appears **once** and is reused.
- This makes the representation **compact**.

Another Simple Example

Expression: a * b + a * b

AST:

(+)

/ \

a*b a*b

DAG:

(+)

/ \

(a*b) same node

Advantages of DAG

1. **Eliminates redundant computations** (common sub-expressions).
2. **Saves memory** by not duplicating sub-trees.
3. Helps in **optimization** (e.g., register allocation, code generation).

✅ **In short:**

- Construct AST.
- Identify and merge common sub-expressions.
- Result = DAG.

Three-Address Code (TAC)

Definition

Three-Address Code is an **Intermediate Representation (IR)** of a program where **each instruction contains at most three addresses** (operands).

- Typically used in **code generation and optimization**.
- Helps to simplify translation from high-level language to machine code.

Structure

A TAC instruction generally looks like:

x = y op z

- x → result (temporary variable or memory location)
- y → first operand
- op → operator (+, -, *, /, etc.)
- z → second operand

Note: Some instructions may have fewer operands:

- Unary operators: x = -y
- Assignment: x = y
- Control transfer: if x < y goto L1

Characteristics

1. **Three addresses** max per instruction.
2. Uses **temporary variables** (t1, t2…) to hold intermediate results.
3. **Linear sequence** – easy for optimization and code generation.
4. Can represent **arithmetic, assignments, conditional/unconditional jumps, array accesses, and function calls**.

Examples

Example 1: Arithmetic Expression

Expression:

x = a + b * c

Step 1: Break into TAC using temporaries

t1 = b * c

t2 = a + t1

x = t2

Example 2: Conditional Statement

High-level code:

if a < b then x = a + b

TAC:

if a < b goto L1

goto L2

L1: t1 = a + b

x = t1

L2:

Example 3: Array Access

High-level code:

a[i] = b[j] + c

TAC:

t1 = b[j]

t2 = t1 + c

a[i] = t2

Types of TAC Representation

1. **Quadruples**: (op, arg1, arg2, result)
2. **Triples**: (op, arg1, arg2) – result referenced by index
3. **Indirect Triples**: Uses pointer table to refer triples

Advantages

- Simple and uniform representation.
- Easier for **code optimization** (common sub-expression, constant folding).
- Helps in **translation to machine code**.

✅ **Summary:**

- TAC breaks complex expressions/statements into **simple 3-address instructions**.
- Uses **temporary variables** for intermediate results.
- Forms the **basis for code optimization and generation**.

If you want, I can **draw a diagram showing a high-level expression → TAC → quadruples → triples** to visualize it in one shot.

Do you want me to do that?

“t2 = t1 + c”

Would you like me to **draw DAG construction step-by-step** for a smaller expression (like a + a * b + b * c) with **tables and node numbering** (the way it appears in compiler design exam answers)?

You’ve hit the Free plan limit for GPT-5.

Responses will use another model until your limit resets after 5:03 PM.

Upgrade to Go