🎓 FINAL EXAM SYLLABUS BREAKDOWN — Compiler Construction (CS4031)

**🧠 SEMANTIC ANALYSIS**

**🔹 Introduction**

* **Purpose:** Ensures that the parsed syntax has **meaning**.
* Detects **semantic errors** like:
  + Using undeclared variables
  + Type mismatches (int a = "text")
  + Function misuse
* Conducted after syntax analysis, before intermediate code generation.

**🔹 Attributes for Grammar Symbols**

* **Attributes:** Metadata added to grammar symbols (e.g., types, values)
* Two kinds:
  1. **Synthesized attributes** → Bottom-up (from children)
  2. **Inherited attributes** → Top-down (from parent/sibling)
* Syntax-directed definitions (SDDs) use these for rule evaluation.

**🔹 Writing Syntax-Directed Translation (SDT)**

* Combines parsing with semantic rules.
* Example:

plaintext

Copy code

E → E1 + T { E.val = E1.val + T.val }

T → digit { T.val = digit.lexval }

**🔹 Bottom-Up Evaluation of SDT**

* Used with **LR parsers**
* Semantic actions are evaluated when a **production is reduced**
* Only **synthesized attributes** work well here (S-attributed)

**🔹 Creation of Syntax Tree**

* Syntax tree is a compact version of the parse tree
* Shows **hierarchical structure** of code
* Used for:
  + Code generation
  + Optimization

**🔹 Directed Acyclic Graph (DAG)**

* Enhanced syntax tree that eliminates **common subexpressions**
* Avoids redundant computations
* Useful in **optimization phase**

**Example:**

c

Copy code

a = b + c;

d = b + c;

→ Only one node for b + c in DAG

**🔹 Types of SDTs**

1. **S-Attributed** (only synthesized attributes, bottom-up)
2. **L-Attributed** (includes inherited attributes, top-down)
3. **Translation Schemes** (actions embedded in productions)

**🔹 Synthesized Attributed Definition**

* **Attributes derived from child nodes**
* Example:

plaintext

Copy code

E → E1 + T → E.val = E1.val + T.val

**🔹 Top-Down Evaluation of S-Attributed Grammar**

* Applicable in **recursive descent parsers**
* Attributes are evaluated **on the way down**

**🔹 Inherited Attribute Grammar**

* Values are passed from **parent or left siblings**
* Requires **left-to-right traversal** (top-down)

**Example:**

plaintext

Copy code

T → int { T.type = "integer" }

D → T id { id.type = T.type }

**🧩 INTERMEDIATE CODE GENERATOR**

**🔹 Introduction**

* Converts annotated parse tree into **intermediate representation (IR)**
* IR is machine-independent
* Forms a bridge between semantic analysis and code generation

**🔹 Intermediate Languages**

Examples:

* **Three-Address Code (TAC)**
* **Quadruples / Triples**
* **Postfix notation**

**Requirements:**

* Easy to translate into machine code
* Compact and expressive
* Easy for optimization

**🔹 Types of Three-Address Statements**

1. **Assignment**  
   x = y op z
2. **Unary**  
   x = -y
3. **Copy**  
   x = y
4. **Conditional Jump**  
   if x < y goto L1
5. **Unconditional Jump**  
   goto L2
6. **Procedure Call / Return**  
   param x, call p, n, return y

**🔹 Representation of Three Address Code**

1. **Quadruples:**

css

Copy code

op arg1 arg2 result

+ a b t1

1. **Triples:**

css

Copy code

index op arg1 arg2

(0) + a b

1. **Indirect Triples (pointer to triples)**

**🔹 Syntax-Directed Translation into TAC**

* SDD rules generate TAC while parsing
* Example:

mathematica

Copy code

E → E1 + T → E.place = newtemp()

emit("+", E1.place, T.place, E.place)

**⚙️ CODE OPTIMIZATION**

**🔹 Introduction**

* **Goal:** Improve intermediate code without changing meaning
* Improves:
  + **Speed**
  + **Memory usage**
  + **Code size**

**🔹 Where and How to Optimize**

* **Local Optimization:** within a block (basic block)
* **Global Optimization:** across blocks/functions
* **Loop Optimization:** focuses on inner loops (critical for performance)

**🔹 Procedure to Identify Basic Blocks**

* **Basic block:** a sequence of instructions with:
  + One **entry point**
  + One **exit point**
* Steps:
  + Start with first instruction
  + Leaders = first instruction, target of jumps, next of jump
  + Group instructions between leaders into basic blocks

**🔹 Flow Graph**

* Control Flow Graph (CFG): shows flow of control between basic blocks
* Nodes = basic blocks
* Edges = possible transfer of control (jumps, branches)

**🔹 DAG Representation of Basic Block**

* DAG is used to:
  + **Detect common subexpressions**
  + **Eliminate redundant computations**
  + **Determine evaluation order**

**🔹 Construction of DAG**

Steps:

1. Traverse statements
2. For each operator, check if node already exists
3. Reuse node if subexpression already exists
4. Create new node otherwise

**🔹 Loop Optimization**

Techniques:

* **Invariant Code Motion**: move computations outside loop
* **Loop Unrolling**: reduce loop overhead
* **Induction Variable Elimination**

**🔹 Global Flow Analysis**

* Tracks values across basic blocks
* Used for:
  + Constant propagation
  + Dead code elimination
  + Reaching definitions

**🧾 TARGET CODE GENERATION**

**🔹 Introduction**

* Final phase of the compiler
* Converts IR into **assembly / machine code**

**🔹 Issues in the Design of Code Generator**

1. **Instruction Selection:** choosing optimal machine instructions
2. **Register Allocation:** deciding which values stay in registers
3. **Instruction Scheduling:** reordering to reduce pipeline stalls
4. **Target Machine Constraints:** architecture-specific details (x86, ARM, RISC)

**🔹 Instruction Costs**

* Some instructions are **faster** or **cheaper**
* Code generator must prefer:
  + **Fewer instructions**
  + **Cheaper addressing modes**
  + **Efficient usage of registers**

**Example:**

asm

Copy code

MOV R1, #0 ; Fast

MUL R1, R2, R3 ; Expensive

✅ CS4031 – Compiler Construction

Lecture 9: Semantic Analysis  
Instructor: Mahzaib Younas – FAST NUCES CFD

**🔷 Semantic Analysis**

Semantic analysis is the phase of a compiler that **adds meaning to the syntactic structure** derived during parsing. It ensures that the program has **logical sense** and is **meaningful** according to language rules.

**🔸 What is Semantics?**

* Semantics define the **meaning of syntactic elements** like variables, operators, and control structures.
* For example:

c

CopyEdit

int a = "book";

This is **syntactically valid**, but a **semantic error**: "book" is a string, not an integer.

**🔸 Key Equation:**

java

CopyEdit

CFG + Semantic Rules = Syntax Directed Definition (SDD)

**🔷 Role of Semantic Analysis**

Semantic Analysis processes the **parse tree** (from syntax analysis) and produces an **annotated parse tree** by attaching extra information (attributes).

**🔸 Core Tasks:**

1. **Attribute Association**: Attach data to grammar symbols.
2. **Evaluation of Semantic Rules**:
   * Generate intermediate code.
   * Insert into symbol table.
   * Type checking.
   * Error detection.
   * Semantic analysis execution.
3. **Attributes** may hold:
   * Strings, numbers, memory addresses, types, etc.

**🔷 Syntax Directed Translation (SDT)**

**🔸 Definition:**

Syntax-directed translation uses the parser to **drive semantic analysis and translation** based on grammar rules.

**🔸 Two Approaches:**

1. **Syntax Directed Definitions (SDD)**:
   * High-level specification.
   * Do **not specify order** of rule execution.
2. **Syntax Directed Translation Schemes**:
   * Specify **explicit order** of semantic actions within grammar.

**🔷 Attribute Grammar**

* Grammar symbols are assigned **attributes**.
* **Semantic rules** are attached to grammar productions.
* These rules define how to compute attribute values.

**🔸 Example Production with Rules:**

plaintext

CopyEdit

Production Semantic Rule

E → E1 + T E.val = E1.val + T.val

F → digit F.val = digit.lexval

**🔷 Synthesized vs Inherited Attributes**

**🔸 Synthesized Attribute:**

* Computed from **child nodes** up to **parent**.
* Bottom-up flow.
* Example:

kotlin

CopyEdit

A → B C D A.val = B.val + C.val + D.val

**🔸 Inherited Attribute:**

* Passed **from parent/siblings** to the node.
* Top-down flow.
* Example:

kotlin

CopyEdit

C.val = A.val + B.val

**🔷 Annotated Parse Tree**

* A parse tree **with attribute values** shown at each node.
* Values are computed by:
  + Semantic rules.
  + Constants or lexical inputs.
* This process is called **annotating** or **decorating** the parse tree.

**🔷 Dependency Graph**

* A graph showing **evaluation order** of attributes.
* Node M → Node N implies M’s attribute must be computed **before** N’s.

**🔸 Example:**

makefile

CopyEdit

Input: 5 + 3 \* 4

Dependency: Evaluate 3 and 4 before \*, then \*, then +.

**🔷 SDD Types**

**🔹 S-Attributed Grammar:**

* Uses only **synthesized attributes**.
* Works with **bottom-up parsers** (e.g., LR).

**🔹 L-Attributed Grammar:**

* Uses both **synthesized + inherited** attributes.
* Suitable for **top-down parsers** (e.g., LL(1)).
* Restrictions:
  + Inherited attributes come from parent or **left** sibling.

**🔷 Attribute Flow Summary**

| **Attribute Type** | **Source of Value** | **Flow Direction** |
| --- | --- | --- |
| Synthesized | Child nodes | Bottom-Up |
| Inherited | Parent and left sibling nodes | Top-Down |

**🔷 Practical Examples**

**🔸 Inherited Attribute Example**

plaintext

CopyEdit

Production: Semantic Rule:

D → T L L.in = T.type

T → int T.type = integer

L → id addtype(id.entry, L.in)

Explanation:

* Type int is assigned to T.
* This type is passed down as an **inherited attribute** to L.

**🔷 Syntax Directed Definition and Semantic Rules**

**🔸 SDD Sample:**

plaintext

CopyEdit

Production Semantic Rule

L → E n print(E.val)

E → E1 + T E.val = E1.val + T.val

T → T1 \* F T.val = T1.val \* F.val

F → digit F.val = digit.lexval

**🔷 Example 1: Annotated Parse Tree for 3 \* 5 + 4n**

Constructed using **bottom-up** approach:

kotlin

CopyEdit

F.val = 3

T1.val = 3

F.val = 5

T.val = 3 \* 5 = 15

E1.val = 15

F.val = 4

T.val = 4

E.val = 15 + 4 = 19

L → E n

print(E.val) → 19

**🔷 Example 2: (3 + 4) \* (5 + 6)n**

Constructing the syntax tree:

* Compute 3 + 4 = 7
* Compute 5 + 6 = 11
* Multiply 7 \* 11 = 77
* Print 77

**🔷 Example 3: Attributed Grammar for 3 \* 5**

**CFG:**

r

CopyEdit

T → F T'

T' → \* F T' | ε

F → digit

**Semantic Rules:**

ini

CopyEdit

T.val = F.val \* T'.val

T'.val = F.val \* T'.val (recursive)

T'.val = 1 (base case for ε)

F.val = digit.lexval

**Evaluation for 3 \* 5:**

kotlin

CopyEdit

F → 3 ⇒ F.val = 3

T' → \* F ⇒ F → 5 ⇒ F.val = 5, T'.val = 5 \* 1 = 5

T.val = 3 \* 5 = 15

✅ CS4031 – Compiler Construction

Lecture 10: Semantic Analysis – Part 2  
Instructor: Mahzaib Younas – FAST NUCES CFD

🔷 Types of Syntax Directed Translation (SDT)

**🔹 S-Attributed Grammar vs L-Attributed Grammar**

| **Feature** | **S-Attributed Grammar** | **L-Attributed Grammar** |
| --- | --- | --- |
| Attribute Type | Only **synthesized** attributes | Both **synthesized** and **inherited** |
| Placement of Semantic Actions | Only at the **end** of RHS | **Anywhere** in RHS |
| Evaluation Order | **Bottom-up parsing** | **Top-down**, **left-to-right**, **depth-first** traversal |
| Use in Parsers | Mostly with **LR parsers** | Mostly with **LL parsers** |

**🔹 L-Attributed Grammar (Expanded)**

* Allows both synthesized and inherited attributes.
* Restriction: An inherited attribute of a symbol **can only depend** on:
  + Its **parent’s** attributes, or
  + Its **left sibling’s** attributes (not right sibling).

**🔹 Example:**

**Production:**

css

CopyEdit

A → XYZPQ

**Valid L-Attributed Rule:**

ini

CopyEdit

Z.i = f(A.i, X.i, Y.i) ✅

**Invalid L-Attributed Rule:**

cpp

CopyEdit

Z.i = f(P.i, Q.i) ❌ // Violates L-attributed rule by accessing right siblings

**🔷 SDT Classification Example:**

**SDT:**

CopyEdit

A → LM { L.i = f(A.i); M.i = f(L.s); A.s = f(M.s); }

A → QR { R.i = f(A.i); Q.i = f(R.s); A.s = f(Q.s); }

**Solution:**

* First rule is **L-Attributed** ✅
* Second rule is **Not L-Attributed** ❌ (because Q inherits from **R**, which is a right sibling)

**🔷 Semantic Analyzer Tasks**

The semantic analyzer performs **static checks** during compilation. These ensure semantic correctness without executing the code.

**1. Uniqueness Checks**

* Verifies that variables or identifiers are **not redefined** in the same scope.
* Example:

c

CopyEdit

int x;

float x; // ❌ Redefinition error

**2. Flow of Control Checks**

* Ensures flow-altering statements like break, continue, goto are used **legally**.
* Example:

c

CopyEdit

break; // ❌ Error if not inside a loop

**3. Type Checks**

* Ensures operators are used with compatible data types.
* Example:

c

CopyEdit

int a = 5 + "abc"; // ❌ Incompatible types

**4. Name-Related Checks**

* Verifies matching names where required.
* Example (in ADA):

ada

CopyEdit

loop myloop

-- some statements

end myloop; // ✅ must match the start

**🔷 Type Checking**

**🔹 What is Type Checking?**

* Ensures that variables and expressions are used **with correct data types**.
* Prevents **type errors** at runtime or compile time.

**🔹 What is a Type?**

* A set of **values** and **operations** allowed on those values.
* Examples: Integer, Boolean, Real, Char, Pointer.

**🔷 Type System**

**🔹 Categories of Programming Languages:**

| **Type System** | **Description** | **Example Languages** |
| --- | --- | --- |
| **Untyped** | No type-checking done | Assembly |
| **Statically Typed** | All types are known at **compile time** (strongly typed) | C, Java |
| **Dynamically Typed** | Type checks done at **runtime** | Python, Lisp, JavaScript |

**🔹 Type System Definition:**

* A **set of rules** that assigns **type expressions** to program components.
* Different compilers may define **different type systems** for the same language.

**🔹 Pascal Example:**

* array[1..5] of int is **not same** as array[6..10] of int
* Type includes **index set**.

**🔷 Types of Expressions**

**1. Basic Type**

* Primitive types:
  + Boolean, Integer, Char, Real, Void

**2. Constructor Type**

* Constructs a new type from existing types.
* Example: array[1..20] of int = Array(1..20, int)

**3. Product Type**

* Represents tuple-like types:  
  int × char

**4. Record Type**

* Named fields like structures in C:

c

CopyEdit

struct { int age; char name[10]; }

**5. Pointer Type**

* Points to a value in memory.
* Example: pointer(int) or int\* in C

**6. Function Type**

* Type of input and output of a function.
* Example:

sql

CopyEdit

int × int → int // mod(int, int)

char × char → pointer(int)

**🔷 Example: Type Expression Problems**

**1. Pointer to Array of Real (Index 1 to 100)**

**Expression:**

plaintext

CopyEdit

pointer(array[1..100, real])

**2. Function: int → char, result is pointer(int)**

**Expression:**

plaintext

CopyEdit

(integer → char) → pointer(integer)

**🔷 Type Checking for Expressions**

* Compiler must **check the type of operands** and operators.
* Example:

c

CopyEdit

int x = 5 + "abc"; // ❌ Type mismatch

**🔷 Type Checking for Statements**

* Statements typically **do not produce values**, so they are assigned a special type: void.
* Example:

c

CopyEdit

return; // Has type void

* Type checking also applies to:
  + if conditions
  + Loop conditions
  + Assignment compatibility

**🔷 Rules for Symbol Table Entry**

The **symbol table** stores:

* Name of the variable/function
* Scope
* Type
* Memory address / offset
* Additional metadata

Compiler uses it for:

* Type checking
* Uniqueness check
* Error reporting

**🔶 1. Rules for Symbol Table Entry**

This table shows **semantic actions** used during **declarations** and **type assignments**. The purpose is to populate the **symbol table** with identifier types.

| **Production** | **Semantic Rule** | **Explanation** |
| --- | --- | --- |
| D → id : T | addtype(id.entry, T.type) | This means "add the identifier id into the symbol table and assign it the type T.type". Example: int x → x has type int. |
| T → char | T.type = char | When type is declared as char, assign T.type as char. |
| T → integer | T.type = int | When the declared type is integer, assign type as int. |
| T → T₁ \* | T.type = pointer(T₁.type) | If a pointer is declared (like int \*x), the type is derived by wrapping the inner type in a pointer. |
| T → T₁ [num] | T.type = array(0..num-1, T₁.type) | If an array is declared (e.g., int a[10]), the array's index range and base type (T₁.type) are stored as the final type. |

👉 **Goal:** Populate the symbol table during declarations with accurate type info using the production rules.

**🔶 2. Type Checking for Expression**

This table defines **how to assign a type to an expression (E)** and detect **type mismatches** using production rules.

| **Production** | **Type Rule** | **Explanation** |
| --- | --- | --- |
| E → literal | E.type = char | A literal character like 'a' has type char. |
| E → num | E.type = integer | A numeric constant like 5 has type integer. |
| E → id | E.type = lookup(id.entry) | Find id in the symbol table and get its declared type. |
| E → E₁ % E₂ | E.type = if E₁.type == E₂.type == integer then integer else type\_error | % (modulo) operator requires both operands to be integer. If they are, result is integer. |
| E → E₁ [E₂] | E.type = if E₂.type == integer && E₁.type == array(s,t) then t else type\_error | Array indexing: check if E₂ is an index (int) and E₁ is an array. Result type is the base type t. |
| E → \*E₁ | E.type = if E₁.type == pointer(t) then t else type\_error | Pointer dereference: if E₁ is of type pointer(t), dereferencing gives type t. Otherwise, error. |

👉 **Goal:** Ensure operators are used with compatible types. For example, % must have integer operands.

**🔶 3. Type Checking for Statement (Overview)**

This lists **common statements** where type checking will be applied:

| **Production** | **Meaning** |
| --- | --- |
| S → id := E | Assignment of expression E to variable id. Types of E and id must match. |
| S → if E then S₁ | Conditional statement. Type of E must be boolean. |
| S → while E do S₁ | Loop. The condition E must be boolean. |
| S → S₁ ; S₂ | Sequence of statements. Each must be valid. |

These are the **constructs** where type mismatches can be caught during **semantic analysis**.

**🔶 4. Type Checking for Statement (Detailed Semantic Rules)**

This defines **how to apply type checking** rules **in full detail** during semantic analysis:

| **Production** | **Semantic Rule** |
| --- | --- |
| S → id := E | S.Type = if id.type == E.type then void else type\_error 💡 **Explanation**: Assignment must match types. If not, return type error. |
| S → if E then S₁ | S.Type = if E.type == boolean then S₁.Type else type\_error 💡 **Explanation**: E must evaluate to boolean for if to make sense. |
| S → while E do S₁ | S.Type = if E.type == boolean then S₁.Type else type\_error 💡 **Explanation**: Similar to if. Loop condition must be boolean. |
| S → S₁ ; S₂ | S.Type = if S₁.Type == void && S₂.Type == void then void else type\_error 💡 **Explanation**: Both statements must be valid (of type void) to proceed. |

👉 **Key Idea:** Type of a statement is always void if it’s semantically correct. Otherwise, the compiler flags a type\_error.

**✅ Summary: How Type Checking Works in a Compiler**

1. **Declarations (Symbol Table Entry)**:
   * The compiler stores each identifier (id) with its data type (T.type) in the symbol table.
   * This is essential for verifying consistency of use.
2. **Expressions (Type Compatibility)**:
   * The type of every expression is computed based on its components (e.g., operands, operators).
   * Type rules are applied recursively (e.g., E1 + E2 must ensure both E1 and E2 are of type int).
3. **Statements (Control & Assignment)**:
   * Each control construct (if, while) checks the type of its condition.
   * Assignment statements (id := E) check for matching types.
   * Compound statements (S1; S2) ensure both sub-statements are valid.
4. **Type Errors**:
   * Any mismatch between expected and actual types triggers a semantic **type error**.
   * Type checking phase **ensures the program is meaningful**, not just syntactically correct.

**✅ How to Study Intermediate Representation (IR)**

**🧠 Core Idea:**

Intermediate representation (IR) is a data structure or code used internally by the compiler to represent source code in a way that is easier for analysis and optimization.

🎓 CS4031 – Compiler Construction

📘 Lecture 11 – Intermediate Representation (IR) & Three-Address Code (TAC)

**Instructor:** Mahzaib Younas  
**Department of Computer Science – FAST NUCES CFD**

**🧩 What is Intermediate Code?**

**❓Definition:**

An **Intermediate Representation (IR)** is an internal form of the source program used by the compiler. It connects the frontend (parsing, lexical analysis) with the backend (code generation).

**🔧 Why Use Intermediate Code?**

* Machine **independent**
* Enables **portability** between compilers
* Allows **optimizations**
* **Bridges** high-level languages and target machine code

**🧱 Types of Intermediate Representations**

**1️⃣ High-Level IR (HIR)**

* Keeps high-level features like loops, arrays
* Preserves structure (e.g., AST)
* Example:

c

CopyEdit

int f(int a, int b) {

int c;

c = a + 2;

print(b, c);

}

**2️⃣ Medium-Level IR (MIR)**

* Language-independent
* Close to machine but still abstract
* Best for **optimizations**
* Usually represented as **Three Address Code (TAC)**

**3️⃣ Low-Level IR (LIR)**

* Resembles machine instructions
* Architecture-specific
* Used just before generating actual code

**🔢 Types of Intermediate Representations**

| **#** | **IR Type** | **Examples** |
| --- | --- | --- |
| 1 | **Syntax Trees** | Tree-like AST |
| 2 | **DAGs (Directed Acyclic Graph)** | Optimized expression trees |
| 3 | **Postfix Notation** | abc\*+ |
| 4 | **Three Address Code (TAC)** | t1 = b \* c |

**⚙️ Three Address Code (TAC)**

**📌 What is TAC?**

A linearized form of syntax trees using **temporary variables**:

c

CopyEdit

a + b \* c

TAC form:

c

CopyEdit

T1 = b \* c

T2 = a + T1

**✅ General Format:**

ini

CopyEdit

A = B op C

* A: target (temp or variable)
* B, C: operands
* op: operation (e.g., +, \*, -)

**🧠 TAC: Complete Example**

c

CopyEdit

a = b \* -(c - d) + b \* -(c - d);

**TAC Steps:**

ini

CopyEdit

T1 = c - d

T2 = -T1

T3 = b \* T2

T4 = c - d

T5 = -T4

T6 = b \* T5

T7 = T3 + T6

a = T7

**🔍 Syntax Tree for TAC**

Expression: a = -(b - c)

**Rules:**

* E → id
* E → E1 - E2
* E → -E1
* S → id = E

**TAC:**

ini

CopyEdit

T1 = b - c

T2 = -T1

a = T2

**🧾 Types of TAC Statements**

| **#** | **Type** | **Format** | **Example** |
| --- | --- | --- | --- |
| 1 | Binary Assignment | A = B op C | t1 = a + b |
| 2 | Unary Assignment | A = op B | t2 = -a |
| 3 | Copy Statement | A = B | x = y |
| 4 | Unconditional Jump | goto L | goto L1 |
| 5 | Conditional Jump | if a > b goto L | if a > b goto L1 |
| 6 | Function Call | param x, call f, n | param x; call f |
| 7 | Indexed Assign | A[i] = B | \*t2 = x |
| 8 | Address Assign | A = &B | p = &x |
| 9 | Pointer Assign | A = \*B / \*A = B | t = \*p; \*x = y |

**🗃 Representation of TAC**

**✅ 1. Quadruple**

Structure: (op, arg1, arg2, result)

c

CopyEdit

T1 = a + b → (+, a, b, T1)

**✅ 2. Triple**

Structure: (op, arg1, arg2)  
→ No named temporaries, uses position index

css

CopyEdit

(+, a, b) // position 0

**✅ 3. Indirect Triple**

Structure: List of pointers to triples (used for better rearrangement and optimization)

**📐 Syntax-Directed Translation (SDT) for TAC**

**Attributes:**

* place: the name holding result
* code: TAC code list

**Rule Example for: E → E1 + E2**

c

CopyEdit

E.place = newtemp()

E.code = E1.code || E2.code || gen(E.place = E1.place + E2.place)

**✍️ Assignment Statement TAC Generation**

For: a = - (b - c)

**Steps:**

1. E → b, c → lookup symbol table → set E1.place = b, E2.place = c
2. E → E1 - E2 → T1 = b - c
3. E → -E1 → T2 = -T1
4. S → id = E → a = T2

**✅ Type Checking with TAC**

* Ensures that **operand types are compatible** in expressions and statements.

**Example:**

| **Expression** | **TAC** | **Type Rule** |
| --- | --- | --- |
| x = a + b | t1 = a + b; x = t1 | Both a and b must be of same type |
| \*x = y | \*x = y | x must be pointer to y's type |
| x = \*p | t1 = \*p; x = t1 | p must be pointer type |

**🧾 Summary Table: C Code to TAC Examples**

| **C Statement** | **TAC Code** |
| --- | --- |
| x = a + b | t1 = a + b, x = t1 |
| x = -a | t1 = -a, x = t1 |
| if (a > b) goto L1; | if a > b goto L1 |
| arr[i] = x; | t1 = i \* width, t2 = base + t1, \*t2 = x |
| x = \*c; | t1 = \*c, x = t1 |
| p = &x; | p = &x |
| y = call f(x); | param x, y = call f |

**📘 Possible Exam Questions from Lecture 11**

**🔹 Short Questions**

1. What is Intermediate Representation (IR)?
2. Differentiate between HIR, MIR, and LIR.
3. What is the need of generating intermediate code in compilers?
4. Define Three Address Code with example.
5. What is a Quadruple? How is it different from Triple?

**🔹 Medium-Length Questions**

1. Write the TAC for a = b \* -(c - d) + b \* -(c - d).
2. Explain the difference between syntax trees, DAGs, and postfix notation as IR.
3. Describe how assignment and pointer statements are handled in TAC.
4. Explain how function calls and parameter passing is handled in TAC.

**🔹 Long Questions**

1. Discuss the types of three-address code statements with examples.
2. Generate and explain the TAC and syntax tree for a = -(b - c).
3. Compare and contrast the three representations: Quadruples, Triples, and Indirect Triples.
4. Explain syntax-directed translation and how place and code attributes are used for TAC generation.
5. Show how indexed assignments, function calls, and pointer operations are translated to TAC.

🔹 TYPES OF QUESTIONS AND ANSWERS 🔹

**Q1: What is Intermediate Code and why is it needed in a compiler?**

**Answer:**  
Intermediate code is a **machine-independent representation** of source code generated between parsing and code generation.  
**Need:**

* Makes the compiler **portable** (reuse frontend for different machines).
* Allows **optimization** before final machine code is generated.
* Separates source language from target machine.

**Q2: What are the three levels of IR? Explain with examples.**

**Answer:**

| **Level** | **Name** | **Description** | **Example** |
| --- | --- | --- | --- |
| HIR | High-Level IR | Closer to source code | AST, symbol tables |
| MIR | Medium-Level IR | Language-independent, for optimization | Three Address Code |
| LIR | Low-Level IR | Closer to machine code | Assembly-like operations |

**Q3: Generate the Three-Address Code for a = b \* - (c - d) + b \* - (c - d);**

**Step-by-step solution:**

ini

CopyEdit

T1 = c - d

T2 = -T1

T3 = b \* T2

T4 = c - d

T5 = -T4

T6 = b \* T5

T7 = T3 + T6

a = T7

**Q4: Draw the Syntax Tree and TAC for a = -(b - c)**

**Syntax Rules Used:**

mathematica

CopyEdit

E → id

E → E1 - E2

E → -E1

S → id = E

**TAC:**

ini

CopyEdit

T1 = b - c

T2 = -T1

a = T2

**Q5: What are the different types of Three Address Code statements?**

**Answer:**

| **#** | **TAC Type** | **Format** | **Example** |
| --- | --- | --- | --- |
| 1 | Assignment w/ binary op | x = y op z | t1 = a + b |
| 2 | Assignment w/ unary op | x = op y | t1 = -a |
| 3 | Copy | x = y | x = a |
| 4 | Unconditional Jump | goto L | goto L1 |
| 5 | Conditional Jump | if x relop y goto L | if a > b goto L2 |
| 6 | Function Call | param x, call f, n | param a; call f |
| 7 | Indexed Assignment | A[i] = B | \*t2 = x |
| 8 | Address Assignment | x = &y | p = &x |
| 9 | Pointer Assignment | x = \*y, \*x = y | t1 = \*p; \*x = y |

**Q6: Write TAC for this C code: arr[i] = x;**

**Answer:**

makefile

CopyEdit

t1 = i \* width

t2 = base + t1

\*t2 = x

**Q7: Differentiate between Quadruples, Triples, and Indirect Triples**

| **Feature** | **Quadruples** | **Triples** | **Indirect Triples** |
| --- | --- | --- | --- |
| Fields | 4 (op, arg1, arg2, result) | 3 (op, arg1, arg2) | Index list pointing to triples |
| Temp variables | Needed explicitly | Position acts as result | Indexed reference |
| Example | T1 = a + b → (+, a, b, T1) | (+, a, b) → [0] | Index[0] → (+, a, b) |

**Q8: What is a syntax-directed translation (SDT)? How does it help generate IR?**

**Answer:**

* SDT uses grammar rules and semantic actions to generate IR during parsing.
* Each grammar rule has associated code for generating intermediate code.
* Helps build the **three-address code (TAC)** step-by-step during parsing.

**Q9: Explain the attributes place and code used in TAC generation**

**Answer:**

* place: stores variable name or temporary name used to represent result of an expression.
* code: stores list of TAC instructions generated so far.

**Q10: Write syntax-directed definitions (SDD) for this rule:**

E → E1 + E2

**Answer:**

ini

CopyEdit

E.place = newtemp()

E.code = E1.code || E2.code || gen(E.place = E1.place + E2.place)

**🔁 PRACTICE QUESTIONS**

1. Write TAC for: x = (a + b) - (c \* d)
2. Draw the syntax tree and DAG for: a + a \* b + b
3. Differentiate copy statement and unary operation statement in TAC.
4. Explain how indexed and pointer assignments are represented in TAC.
5. Generate TAC for: if (a > b) then x = y + z

**🔸Q1: What is Intermediate Code? Why is it used in a compiler?**

**✅ Answer:**

**Intermediate Code (IC)** is a representation of the source program between the high-level source language and the low-level machine code. It acts as a bridge between the front-end (parsing) and back-end (code generation) of a compiler.

**📌 Purpose:**

* Makes the compiler **portable** and **machine-independent**
* Facilitates **optimization**
* Helps in **error detection and debugging**
* Simplifies **code generation**

**🔸Q2: What are the types of Intermediate Representations (IRs) used in compilers? Explain with examples.**

**✅ Answer:**

**IR Types:**

1. **High-Level IR (HIR):**
   * Example: Abstract Syntax Tree (AST)
   * Retains loop structures and data types
   * Source: int c = a + 2;
2. **Medium-Level IR (MIR):**
   * Example: Three Address Code (TAC)
   * Independent of source & target language
   * TAC:

ini

CopyEdit

T1 = a + 2

c = T1

1. **Low-Level IR (LIR):**
   * Example: Assembly-like code
   * Directly maps to machine instructions

**🔸Q3: Write Three Address Code (TAC) for the expression a = b \* -(c - d) + b \* -(c - d)**

**✅ Answer:**

ini

CopyEdit

T1 = c - d

T2 = -T1

T3 = b \* T2

T4 = c - d

T5 = -T4

T6 = b \* T5

T7 = T3 + T6

a = T7

**🔸Q4: What are the types of Three Address Statements? List and give examples.**

**✅ Answer:**

| **Type** | **Form** | **Example (TAC)** |
| --- | --- | --- |
| 1. Binary Assignment | A = B op C | T1 = a + b |
| 2. Unary Assignment | A = op B | T2 = -a |
| 3. Copy | A = B | x = y |
| 4. Unconditional Jump | goto L | goto L1 |
| 5. Conditional Jump | if X relop Y goto L | if a > b goto L2 |
| 6. Function Call | param x, call f | param x; y = call f |
| 7. Indexed Assignment | A[i] = B / A = B[i] | \*t2 = x / x = \*t2 |
| 8. Address Assignment | A = &B | p = &x |
| 9. Pointer Assignment | A = \*B / \*A = B | x = \*p / \*x = y |

**🔸Q5: What is the difference between Quadruples, Triples, and Indirect Triples?**

**✅ Answer:**

| **Feature** | **Quadruple** | **Triple** | **Indirect Triple** |
| --- | --- | --- | --- |
| Structure | (op, arg1, arg2, result) | (op, arg1, arg2) | List of pointers to triples |
| Temp Variables | Named explicitly (T1, T2) | Refers to instruction positions | Uses array of pointers |
| Readability | Easy to read | Harder to manage for optimization | Better for rearrangement |

**🔸Q6: Explain how a function call is represented in Three Address Code.**

**✅ Answer:**

For: y = func(a, b)

wasm

CopyEdit

param a

param b

y = call func, 2

**🔸Q7: Translate the statement arr[i] = x; into TAC.**

**✅ Answer:**

Assume size of one element = width

makefile

CopyEdit

T1 = i \* width

T2 = base + T1

\*T2 = x

**🔸Q8: Generate TAC and show syntax tree for a = -(b - c)**

**✅ TAC:**

ini

CopyEdit

T1 = b - c

T2 = -T1

a = T2

**✅ Syntax Tree:**

markdown

CopyEdit

=

/ \

a -

|

-

/ \

b c

**🔸Q9: What is the role of place and code attributes in Syntax Directed Translation for TAC?**

**✅ Answer:**

* **place**: Stores the name of the variable or temporary holding the result of a nonterminal.
* **code**: Contains the list of TAC statements generated for a nonterminal.

Used in rules like:

mathematica

CopyEdit

E → E1 + E2

E.place = newtemp()

E.code = E1.code || E2.code || gen(E.place = E1.place + E2.place)

**🔸Q10: Exam MCQ/Short Theoretical Type Questions**

| **Question** | **Answer** |
| --- | --- |
| Q: Which IR is closest to source language? | High-Level IR |
| Q: What is the format of Three Address Code? | A = B op C |
| Q: Which IR is used just before machine code? | Low-Level IR |
| Q: What structure uses op, arg1, arg2, result? | Quadruple |
| Q: What is the purpose of TAC? | Platform-independent intermediate representation |

**📝 BONUS: Practice Question**

**Q:** Write TAC for this code:

c

CopyEdit

x = a \* b + c;

y = x - d;

**✅ Answer:**

ini

CopyEdit

T1 = a \* b

T2 = T1 + c

x = T2

T3 = x - d

y = T3

**✅ 📝 Question:**

Convert the following C-style program into Three Address Code (TAC). Make sure to identify all 9 types of TAC statements.

c

CopyEdit

int a, b, c, d, x, y, z, \*p;

int arr[10];

void main() {

a = b + c; // Binary Assignment

x = -a; // Unary Assignment

y = x; // Copy

if (a > b) goto L1; // Conditional Jump

goto L2; // Unconditional Jump

L1:

z = add(a); // Function Call

L2:

arr[d] = z; // Indexed Assignment

p = &x; // Address Assignment

y = \*p; // Pointer Read

\*p = y; // Pointer Write

}

**✅ 🧾 Answer (TAC with Comments):**

| **TAC Statement** | **Type** | **Explanation** |
| --- | --- | --- |
| T1 = b + c | 1. Binary Assignment | a = b + c step |
| a = T1 | - | Store result in a |
| T2 = -a | 2. Unary Assignment | x = -a step |
| x = T2 | - | Store in x |
| y = x | 3. Copy | Copy x to y |
| if a > b goto L1 | 5. Conditional Jump | Conditional branching |
| goto L2 | 4. Unconditional Jump | Unconditional jump |
| L1: | - | Label |
| param a | 6. Function Call | Pass param a to function add() |
| z = call add, 1 | 6. Function Call | Function call that returns to z |
| L2: | - | Label |
| T3 = d \* 4 | (Assume int width = 4) | Index offset |
| T4 = arr + T3 |  | Compute address |
| \*T4 = z | 7. Indexed Assignment | arr[d] = z |
| p = &x | 8. Address Assignment | p = &x |
| T5 = \*p | 9. Pointer Assignment (read) | y = \*p |
| y = T5 |  | Store into y |
| \*p = y | 9. Pointer Assignment (write) | \*p = y |

**✅ Final TAC (without comments):**

makefile

CopyEdit

T1 = b + c

a = T1

T2 = -a

x = T2

y = x

if a > b goto L1

goto L2

L1:

param a

z = call add, 1

L2:

T3 = d \* 4

T4 = arr + T3

\*T4 = z

p = &x

T5 = \*p

y = T5

\*p = y

✅ Lecture 13 – Code Optimization (FULL EXPLANATION)

**🔍 What is Code Optimization?**

* It is the **transformation of code** to make it **more efficient** in terms of:
  + **Execution time** (runs faster)
  + **Memory space** (uses fewer resources)
* It does **not** change the **semantic meaning** of the original program.

**🤔 Why Do We Need Optimization?**

1. **Inefficient programming**:
   * Example: a = a + 0 (no real effect – can be removed)
2. **Easy-to-write constructs**:
   * Loops or conditions may be written in simpler but inefficient ways.
3. **Temporary variables**:
   * Generated by compilers, may be redundant.

**⚠️ Constraints of Optimization**

Any optimization must:

* ✅ Preserve the **semantic meaning** (original logic)
* ✅ Provide **measurable improvement** in performance
* ✅ Be worth applying (i.e., don’t use costly analysis for rarely run programs)
* ✅ Be suitable only when the code format allows the transformation

**🔗 Classification of Optimizations**

**🔹 Based on Code Level**

| **Level** | **Description** |
| --- | --- |
| **Design Level** | Developer chooses best algorithm/data structure manually |
| **Source Code Level** | User rewrites code for better performance |
| **Compile Level** | Compiler optimizes using loops, procedure calls, address calculations (on 3AC) |
| **Assembly Level** | Compiler uses machine-specific tricks (registers, instruction sets) |

**🔹 Based on Programming Language**

| **Type** | **Examples** |
| --- | --- |
| **Machine-Independent** | Constant folding, constant propagation, dead code elimination |
| **Machine-Dependent** | Register allocation, strength reduction, peephole optimizations |

**🔹 Based on Scope**

| **Type** | **Description** |
| --- | --- |
| **Local Optimization** | Within a basic block (simple, no data/control flow analysis needed) |
| **Global Optimization** | Across basic blocks (needs **data-flow analysis**) |

**🛠 Where & How Optimization Happens**

**1. Control Flow Analysis**

* Builds a **Control Flow Graph (CFG)** from the code
* Helps understand all paths the program might follow

**2. Data Flow Analysis**

* Tracks **flow of variables and values**
* Builds a **Data Flow Graph (DFG)**
* Used for optimizations like live variable analysis, reaching definitions

**3. Code Transformations**

* Transformations applied:
  + Don't change meaning
  + Improve runtime or space
  + Are based on the CFG/DFG analysis

**🔄 Code Optimization Model**

**🔸 Flow Graph**

* A visual representation of the control flow of a program
* **Nodes** = Basic Blocks
* **Edges** = Control transfer between blocks (like if, goto, etc.)

**🔸 Basic Block**

Definition:

* A sequence of instructions **without any jump/branch** except at the end.
* If **control enters** the basic block, it will **execute entirely and sequentially**.

**Example:**

For:

c

CopyEdit

a = b + c \* d / e;

TAC:

ini

CopyEdit

t1 = c \* d

t2 = t1 / e

a = b + t2

This is a **basic block** because:

* No jump or branch inside it
* Straight-line code

**🔍 How to Identify Basic Blocks?**

1. **Rule 1 (First Statement)** is a leader
2. **Rule 2 (Target of GOTO)** is a leader
3. **Rule 3 (Statement after a jump)** is a leader

Then, each leader starts a **new basic block**.

**📘 Example:**

Suppose this TAC:

plaintext

CopyEdit

(1) a = 1

(2) b = 2

(3) if a < b goto (12)

(4) c = a + b

(5) ...

(11) d = 0

(12) return d

**Leaders:**

* (1) by Rule 1
* (3), (12) by Rule 2
* (4), (12) by Rule 3

**Basic Blocks:**

* **B1**: (1), (2)
* **B2**: (3), (4)
* **B3**: (5)–(11)
* **B4**: (12)

**🔷 DAG (Directed Acyclic Graph)**

* Represents **data dependencies** in a basic block
* Used to **eliminate redundant operations** and identify **common subexpressions**

**🔧 DAG Nodes**

| **Type** | **Meaning** |
| --- | --- |
| **Leaf Nodes** | Constants or variable names |
| **Interior Nodes** | Operators (add, mul, etc.), result of expressions |

**🏗 DAG Construction Rules**

Given code:

c

CopyEdit

t1 = 4 \* i

t2 = a[t1]

t3 = b[t1]

t4 = t2 \* t3

pr = pr + t4

i = i + 1

if i <= 20 goto (1)

**Steps:**

1. Create i, 4, then node t1 = 4 \* i
2. Use t1 as index for arrays a[t1], b[t1] (create t2, t3)
3. Compute t4 = t2 \* t3
4. Add to pr, increment i
5. Check loop condition

✔ This DAG can now be used to **optimize code** by avoiding recomputations.

✅ Lecture 14 – Techniques of Code Optimization

**🔹 Overview**

Code optimization techniques can be **local** (within a single basic block) or **global** (across basic blocks). The aim is to reduce **execution time**, **memory usage**, and remove **redundant** or **inefficient code**, **without changing the output**.

**🔸 LOCAL CODE OPTIMIZATION TECHNIQUES**

Local optimization means performing optimization **within one basic block**.

**🔽 Why Local Optimization?**

* Simple to implement
* Does **not require control/data-flow analysis**
* Targets immediately visible inefficiencies

**📍 1. Constant Folding**

**Definition**: Evaluating constant expressions at compile-time.

**Example**:

c

CopyEdit

Circumference = (22 / 7) \* d;

✔ Replace (22 / 7) with 3.14 at compile-time to save execution time.

**📍 2. Constant Propagation**

**Definition**: If a variable is assigned a constant value and its value doesn’t change, we replace that variable with its constant wherever it is used.

**Example**:

c

CopyEdit

pi = 3.14;

radius = 10;

area = pi \* radius \* radius;

➡ Compiler will compute:

c

CopyEdit

area = 3.14 \* 10 \* 10;

**📍 3. Algebraic Simplification**

Using algebraic rules to make expressions simpler.

**Examples**:

* x + 0 → x
* x \* 1 → x
* x \* 0 → 0
* x - 0 → x

**📍 4. Strength Reduction**

Replace **expensive operations** (like multiplication, division) with **cheaper ones** (like addition, shift).

**Example**:

c

CopyEdit

x = x \* 2; → x = x + x;

y = x / 4; → y = x >> 2; // Bitwise shift

**📍 5. Dead Code Elimination**

Remove statements that are never executed or whose result is unused.

**Example**:

c

CopyEdit

int x = 5;

x = 6; // x = 5 is dead

Another form:

c

CopyEdit

if (false) {

print("This will never run");

}

**📍 6. Common Subexpression Elimination**

If the same expression is computed more than once, compute it once and reuse the result.

**Original Code**:

c

CopyEdit

int c = a \* 2;

int d = a \* 2; // redundant

**Optimized**:

c

CopyEdit

int c = a \* 2;

int d = c;

**🔸 GLOBAL CODE OPTIMIZATION**

**🌐 What is Global Optimization?**

Optimization **across multiple basic blocks** in a function or procedure using **data-flow analysis**.

* Basic blocks = nodes in a **Control Flow Graph (CFG)**
* Global optimizations can:
  + Remove unnecessary computations
  + Improve loop execution
  + Eliminate unreachable branches

**🔸 PEEPHOLE OPTIMIZATION**

Local optimization on small **sets of instructions** (usually 1–3), like through a **peephole**.

**🔍 Characteristics:**

* Low-level, machine-dependent optimization
* Replaces inefficient instructions with efficient equivalents
* Typical optimizations:
  + Remove redundant loads/stores
  + Collapse multiple jumps
  + Remove unreachable code
  + Replace patterns with **machine idioms**

**🔸 LOOP OPTIMIZATION**

Critical optimization as **loops are repeated many times**.

**🎯 Goals:**

* Move **invariant code** outside loop
* Replace costly **induction variables**
* Avoid recomputing constants

**📍 1. Loop Invariant Code Motion**

Move calculations that do not change within the loop **outside** the loop.

**Before**:

c

CopyEdit

for (int i = 0; i < n; i++) {

a = 5;

sum += a \* i;

}

**After**:

c

CopyEdit

a = 5;

for (int i = 0; i < n; i++) {

sum += a \* i;

}

**📍 2. Induction Variable Elimination**

Replace multiplications with additions when dealing with induction variables.

**Before**:

c

CopyEdit

t1 = i \* 4;

**After**:

c

CopyEdit

t1 = t1 + 4; // Replaces multiplication with addition

**📍 3. Example:**

**Before Optimization**:

c

CopyEdit

int sum = 0;

int x = 10;

for (int i = 0; i < 1000; i++) {

sum += x \* 2;

}

**After Optimization**:

c

CopyEdit

int sum = 0;

int x = 10;

int temp = x \* 2;

for (int i = 0; i < 1000; i++) {

sum += temp;

}

✅ Lecture 15 – Code Generator

**🧠 What is a Code Generator?**

* The **code generator is the final phase** of the compiler.
* It **converts Intermediate Representation (IR)** into **target machine code**.
* It uses:
  + Intermediate Code (e.g., Three Address Code)
  + Symbol Table
  + Target architecture details

**🔹 Main Tasks of Code Generator**

1. **Instruction Selection**:
   * Choose the most suitable machine instruction for each IR statement.
2. **Register Allocation and Assignment**:
   * Decide which variable is stored in which CPU register.
   * Reuse registers efficiently.
3. **Instruction Ordering**:
   * Decide the **execution order** of instructions for better performance (reduce dependencies, pipeline stalls).

**🧾 Examples**

**✅ Instruction Selection**

**IR**:

ini

CopyEdit

t1 = a + b

**Machine Code**:

asm

CopyEdit

MOV EAX, a

ADD EAX, b

MOV t1, EAX

**✅ Register Allocation**

**IR**:

ini

CopyEdit

t1 = a + b

**Registers**:

asm

CopyEdit

MOV R1, a

MOV R2, b

ADD R1, R2

MOV t1, R1

**✅ Instruction Ordering**

c

CopyEdit

x = a + b;

y = x \* c;

**Optimized Assembly Order**:

asm

CopyEdit

MOV R1, a

MOV R2, b

ADD R1, R2 ; x = a + b

MOV R3, c

MUL R1, R3 ; y = x \* c

**🧱 Inputs to Code Generator**

1. **Intermediate Code (3AC, DAG, or syntax tree)**
2. **Symbol Table** (Variable names, types, scope info)
3. **Target Machine Details**

**🎯 Target Code Output**

The target program can be in:

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Absolute Machine Code** | Directly executable binary | Early microcontrollers |
| **Relocatable Object Code** | Requires linking | .obj in C/C++ |
| **Assembly Code** | Human-readable, for debugging | .s files from GCC/LLVM |

**🧰 Memory Management in Code Generation**

* **Mapping variables to memory addresses** (done by front-end + code generator)
* Use **backpatching** to handle forward jumps or unknown addresses.
* Translate labels in 3AC to real instruction addresses.

**🖥 Target Machine Assumptions**

* Byte-addressable
* 4-byte words
* General-purpose registers: R0, R1, ..., Rn-1
* Two-address format:  
  op source, destination

**🔘 Opcodes Examples**

| **Opcode** | **Meaning** |
| --- | --- |
| MOV | Copy data from source to destination |
| ADD | Add source to destination |
| SUB | Subtract source from destination |

**📌 Addressing Modes**

| **Mode** | **Description** | **Example** |
| --- | --- | --- |
| **Absolute** | Direct memory access | MOV A, B (A & B in memory) |
| **Register** | Operand in register | MOV R1, R2 |
| **Indexed** | Array access via base + offset | MOV R1, A[R2] |
| **Indirect Register** | Register points to memory | MOV R1, [R2] |
| **Indirect Indexed** | Computed address points to memory | MOV R1, [R2 + offset] |
| **Literal** | Constant operand | MOV R1, #5 (R1 ← 5) |

**🧮 Instruction Cost Model**

For a **simple processor**, instruction cost = memory accesses

**Example 1:**

asm

CopyEdit

MOV R0, R1

* Read R0 → 0
* Write R1 → 1  
  ✔ **Total Cost: 1**

**Example 2:**

asm

CopyEdit

MOV R0, M

* Read R0 → 0
* Find M → 1
* Store in M → 1  
  ✔ **Total Cost: 2**

**Goal: Minimize memory access ⇒ reduce cost**

**⚙ LLVM Compilation Procedure**

Step-by-step:

bash

CopyEdit

llvm-as input.ll -o input.bc

llc input.bc -filetype=obj -o output.o

clang output.o -o finalExecutable

**📚 Exam Questions + Solutions**

**✅ Short Theory Questions**

1. **What is the role of a code generator in a compiler?**

**Ans**: It converts intermediate code (IR) into target machine code while handling instruction selection, register allocation, and memory addressing.

1. **Name any two addressing modes with examples.**

**Ans**:

* + Register Mode: MOV R1, R2
  + Literal Mode: MOV R1, #10

1. **What is the difference between absolute and relocatable code?**

**Ans**:

* + Absolute code can be directly loaded and executed.
  + Relocatable code must be linked and can be loaded at different addresses.

**✅ Long/Descriptive Questions**

1. **Explain the main tasks of a code generator.**

**Ans**:

* + **Instruction Selection**: Choose matching machine instruction.
  + **Register Allocation**: Assign CPU registers to variables.
  + **Instruction Ordering**: Schedule instructions for better performance.

1. **Describe the input and output of a code generator.**

**Ans**:

* + **Input**: 3AC/DAG + symbol table.
  + **Output**: Target program (absolute/relocatable/assembly).

1. **Explain five addressing modes used in code generation with examples.**

**Ans**:

* + Absolute: MOV M, A
  + Register: MOV R1, R2
  + Indexed: MOV R1, A[R2]
  + Indirect Register: MOV R1, [R2]
  + Literal: MOV R1, #5

1. **Discuss the significance of instruction cost in code generation. Solve a given instruction cost.**

**Example**:

asm

CopyEdit

ADD R1, R2

* + Read R1: 0
  + Read R2: 0
  + Write to R2: 1  
    **Cost: 1**

**✅ Code-Based Questions**

1. **Translate 3AC t1 = a + b to assembly with register allocation.**

**Ans**:

asm

CopyEdit

MOV R1, a

MOV R2, b

ADD R1, R2

MOV t1, R1

1. **Write instruction sequence for x = a + b; y = x \* c;**

**Ans**:

asm

CopyEdit

MOV R1, a

MOV R2, b

ADD R1, R2 ; R1 now holds x

MOV R3, c

MUL R1, R3 ; y = x \* c in R1

1. **Calculate the cost of MOV R1, M followed by MOV M, R2**
   * MOV R1, M → Cost = 2
   * MOV M, R2 → Cost = 2  
     ✔ **Total Cost: 4**

**✅ MCQs**

1. **Which of these is not a code generator task?**  
   a) Lexical analysis  
   ✅ b) Syntax parsing  
   c) Register allocation  
   d) Instruction selection
2. **What is the output of a code generator?**  
   a) Intermediate code  
   ✅ b) Target machine code  
   c) Tokens  
   d) Syntax tree
3. **Which instruction has highest cost?**  
   a) MOV R1, R2  
   ✅ b) MOV R1, M  
   c) ADD R1, R2  
   d) MOV #5, R1
4. **Which addressing mode uses a pointer stored in a register?**  
   a) Absolute  
   b) Indexed  
   ✅ c) Indirect Register  
   d) Literal

**✅ Viva Questions**

1. What is instruction selection?
2. How does register allocation affect performance?
3. Explain the difference between absolute and relocatable code.
4. What is backpatching in code generation?
5. Name some assembly output formats produced by modern compilers.