**CS3501-COMPILER DESIGN**

**PART-B(13 Marks)**

**UNIT 1 INTRODUCTION TO COMPILERS& LEXICAL ANALYSIS**

**1.Describe the various phases of compiler with suitable example**

**1. Phases of a Compiler with an Example**

A compiler translates a high-level program into machine code through multiple phases. The phases are:

**(i) Lexical Analysis:**

* Converts source code into tokens.
* Example: int x = 10; → Tokens: int, x, =, 10, ;

**(ii) Syntax Analysis (Parsing):**

* Checks grammar rules (structure of the program).
* Example: int x = 10; → Verifies correct variable declaration.

**(iii) Semantic Analysis:**

* Ensures logical correctness (e.g., type checking).
* Example: x = "hello"; → Error (integer cannot store string).

**(iv) Intermediate Code Generation:**

* Converts code into an intermediate representation (IR).
* Example: Three-address code:

ini

CopyEdit

T1 = 10

x = T1

**(v) Code Optimization:**

* Improves performance by eliminating redundancies.
* Example: x = 5 \* 2; → Optimized to x = 10;

**(vi) Code Generation:**

* Generates machine code.
* Example: Assembly code:

nginx

CopyEdit

MOV R1, #10

MOV x, R1

**(vii) Symbol Table & Error Handling (Throughout Phases):**

* Maintains variable information.
* Handles errors in syntax, semantics, etc.

**2. Analyze structure of compiler with an assignment statement.**

A compiler consists of:

**(i) Front End:**

* **Lexical Analyzer:** Converts input into tokens.
* **Syntax Analyzer:** Parses tokens into syntax tree.
* **Semantic Analyzer:** Checks for logical correctness.
* **Intermediate Code Generator:** Produces IR.

**(ii) Middle End:**

* **Optimizer:** Improves IR performance.

**(iii) Back End:**

* **Code Generator:** Converts IR to machine code.

**Example with a = b + c \* d;**

1. **Lexical Analysis:** Tokens: a, =, b, +, c, \*, d, ;
2. **Syntax Analysis:** Parses into an expression tree.
3. **Semantic Analysis:** Checks types of b, c, and d.
4. **Intermediate Code Generation:**

ini

CopyEdit

T1 = c \* d

T2 = b + T1

a = T2

1. **Optimization:** Removes redundant calculations.
2. **Code Generation:**

css

CopyEdit

LOAD R1, c

MUL R1, d

ADD R1, b

STORE R1, a

**3. Discuss in detail about the role of Lexical analyzer with the possible error recovery actions.**

The **Lexical Analyzer** is the first phase of a compiler. It scans the source code and:

* Converts character streams into tokens.
* Removes comments & whitespace.
* Identifies keywords, identifiers, literals, etc.

**Error Recovery Techniques:**

1. **Panic Mode Recovery:** Skips characters until a delimiter (e.g., ;) is found.
2. **Error Productions:** Uses predefined rules to handle invalid tokens.
3. **Symbol Table Repair:** Corrects misspelled keywords (e.g., fload → float).
4. **Token Re-insertion:** Replaces missing tokens (e.g., int x 10; → int x = 10;).

**4. Describe in detail about issues in lexical analysis.**

1. **Token Ambiguity:**
   * Example: if(condition)else\_stmt; (if else is part of if or separate).
2. **Handling Reserved Keywords & Identifiers:**
   * int class = 5; (Error: class is a keyword).
3. **Dealing with Whitespace & Comments:**
   * int x = 10; // assign 10 (Ignored by lexical analyzer).
4. **Character Encoding Issues:**
   * UTF-8, ASCII, etc., may cause errors in different environments.
5. **Efficient Scanning:**
   * Requires optimal DFA (Deterministic Finite Automata) to avoid backtracking.

**5. Describe the Input buffering techniques in detail**

**Input buffering** helps optimize lexical analysis by reducing character-by-character reading overhead.

**(i) Naïve Approach (One Character at a Time)**

* **Inefficient** as it requires multiple disk accesses.

**(ii) Buffering with Two Buffers (Lookahead Buffering)**

* Uses **two buffers** of equal size (N).
* First buffer reads input, second buffer provides lookahead.
* **Example:**

lua

CopyEdit

| H | e | l | l | o | | W | o | r | l | d |

|---|---|---|---|---|---|---|---|---|---|---|

^ ^ ^

lexeme\_start forward buffer\_end

* **Advantages:** Reduces frequent I/O calls.

**(iii) Sentinels in Buffering**

* Uses **sentinels (EOF markers)** to avoid additional checks.
* Example: Instead of checking end-of-buffer repeatedly, a **special marker ($)** signals the end.

**(iv) Scheme for Handling Large Tokens**

* If a token is too large to fit in one buffer, **lexeme linking** stores parts separately.

**UNIT-II SYNTAX ANALYSIS**

**1.Explain left recursion and Left Factoring.**

**1. Left Recursion and Left Factoring**

**(i) Left Recursion:**

* A grammar is **left-recursive** if a **non-terminal** appears as the first symbol in its production, leading to infinite recursion.
* Example:

less

CopyEdit

A → A α | β

This causes **infinite recursion**, making parsing difficult.

* **Eliminating Left Recursion:**

vbnet

CopyEdit

A → β A'

A' → α A' | ε

Here, A' is introduced to handle recursion.

* **Example:**

r

CopyEdit

E → E + T | T

Eliminating left recursion:

pgsql

CopyEdit

E → T E'

E' → + T E' | ε

**(ii) Left Factoring:**

* If two or more productions have a **common prefix**, left factoring is used.
* Example:

less

CopyEdit

A → α β1 | α β2

Factoring out α:

vbnet

CopyEdit

A → α A'

A' → β1 | β2

* **Example:**

mathematica

CopyEdit

S → if E then S else S | if E then S

Left Factored:

bash

CopyEdit

S → if E then S S'

S' → else S | ε

**2. Eliminate left recursion and left factoring for the following grammar. E → E + T | E - T | T T → a | b | ( E ).**

Given Grammar:

r

CopyEdit

E → E + T | E - T | T

T → a | b | (E)

* **Removing Left Recursion:**

pgsql

CopyEdit

E → T E'

E' → + T E' | - T E' | ε

(Since E starts with E, we replace it with T E'.)

* **No left factoring needed** in T → a | b | (E), since it has distinct productions.

**3. What is an ambiguous and un ambiguous grammar? Identify the following grammar is ambiguous or not.**

**E→E+E | E\*E | (E)|-E |id for the sentence id+id\*id**

**(i) Ambiguous Grammar:**

A grammar is **ambiguous** if it allows **multiple parse trees** for the same string.

Example:

mathematica

CopyEdit

E → E + E | E \* E | (E) | id

For id + id \* id, there are **two parse trees:**

1. id + (id \* id)
2. (id + id) \* id  
   Since **operator precedence** is unclear, the grammar is **ambiguous**.

**(ii) Unambiguous Grammar:**

A grammar is **unambiguous** if it has **one unique parse tree** for every valid string.

* **Disambiguation by setting precedence:**

pgsql

CopyEdit

E → T E'

E' → + T E' | ε

T → F T'

T' → \* F T' | ε

F → (E) | id

* + \* has higher precedence than +.
  + Parsing id + id \* id will now correctly **group multiplication first**.

Thus, the given grammar is **ambiguous**.

**4.Illustrate the predictive parser for the following grammar.**

**S→ (L) | a L→ L, S | S**

Given Grammar:

less

CopyEdit

S → (L) | a

L → L, S | S

**FIRST and FOLLOW Sets:**

* FIRST(S) = { ( , a }
* FIRST(L) = { ( , a }
* FOLLOW(S) = { $, ) , , }
* FOLLOW(L) = { ) }

**Predictive Parsing Table:**

| **Non-Terminal** | **(** | **a** | **)** | **,** | **$** |
| --- | --- | --- | --- | --- | --- |
| S | (L) | a | - | - | - |
| L | L, S | S | - | - | - |

**Parsing Example ((a, a))**

1. S → (L)
2. L → L, S
3. L → S
4. S → a
5. L → L, S
6. S → a

Thus, **string (a, a) is accepted**.

**6.Evaluate predictive parsing table and parse the string id+id\*id. find FIRST and FOLLOW. E→E+T | T T→T\*F | F F→(E) | id**

Given Grammar:

E → E + T | T

T → T \* F | F

F → (E) | id

**(i) FIRST and FOLLOW Sets**

* FIRST(E) = { id, ( }
* FIRST(T) = { id, ( }
* FIRST(F) = { id, ( }
* FOLLOW(E) = { $, ) }
* FOLLOW(T) = { +, $, ) }
* FOLLOW(F) = { \*, +, $, ) }

**(ii) Predictive Parsing Table:**

| **Non-Terminal** | **id** | **+** | **\*** | **(** | **)** | **$** |
| --- | --- | --- | --- | --- | --- | --- |
| E | E → T E' | - | - | E → T E' | - | - |
| E' | - | E' → + T E' | - | - | E' → ε | E' → ε |
| T | T → F T' | - | - | T → F T' | - | - |
| T' | - | T' → ε | T' → \* F T' | - | T' → ε | T' → ε |
| F | F → id | - | - | F → (E) | - | - |

**(iii) Parsing id + id \* id**

**Steps:**

1. E → T E'
2. T → F T'
3. F → id
4. T' → ε
5. E' → + T E'
6. T → F T'
7. F → id
8. T' → \* F T'
9. F → id
10. T' → ε
11. E' → ε

✔ **String is accepted.**

**UNIT-III SYNTAX DIRECTED TRANSLATIONS AND INTERMEDIATE CODE GENERATION**

**1. Discuss the following in detail about the Syntax Directed Definitions. (i)Inherited Atrributes and Synthesized attributes. (ii) Evaluate SDD of a parse tree.**

### ****1. Syntax Directed Definitions (SDD)****

Syntax Directed Definitions (SDD) are used in compiler design to associate computations with syntax rules. They involve attributes that hold information and rules (semantic rules) that define how attributes are computed.

#### ****(i) Inherited Attributes and Synthesized Attributes****

1. **Inherited Attributes**:
   * These are attributes of a non-terminal that get their values from its parent or siblings in the parse tree.
   * Typically used in top-down parsing.
   * Used to enforce context-sensitive constraints.
   * Example: If a language has a type system where the type of an expression depends on its context, the type could be passed down as an inherited attribute.
2. **Synthesized Attributes**:
   * These attributes get their values from the attributes of their children in the parse tree.
   * Typically used in bottom-up parsing.
   * Used to propagate information upwards in a parse tree.
   * Example: The value of an arithmetic expression (like 3+5) is a synthesized attribute calculated from its operands.

#### ****(ii) Evaluating SDD on a Parse Tree****

The process involves:

1. **Constructing the parse tree**: The tree follows the grammar rules.
2. **Assigning attributes**: Each node in the parse tree has attributes (synthesized or inherited).
3. **Applying semantic rules**: These rules define how the attributes are computed.
4. **Traversing the tree**:
   * If synthesized attributes are used → bottom-up traversal is used.
   * If inherited attributes are used → top-down traversal is used.

Example:  
For an arithmetic expression like E → E1 + T, the synthesized attribute E.val can be computed as E.val = E1.val + T.val.

### 2. Explain the steps for constructing a DAG. Construct the DAG for the following expression ((x+y)-((x+y)\*(x-y)))+((x+y)\*(x-y))

A **DAG (Directed Acyclic Graph)** represents expressions to eliminate redundant computations.

#### ****Steps to Construct a DAG****

1. **Identify Unique Subexpressions**: Each unique subexpression appears only once in the DAG.
2. **Create Nodes for Operands**: Each variable or constant gets a separate node.
3. **Create Nodes for Operators**: Create nodes for operations, ensuring operands are reused.
4. **Maintain Pointers**: Use pointers to connect operators with their operands.
5. **Eliminate Common Subexpressions**: If the same operation is encountered, reuse the existing node instead of creating a new one.

#### ****DAG for the expression****

((x+y)−((x+y)∗(x−y)))+((x+y)∗(x−y))((x+y)-((x+y)\*(x-y)))+((x+y)\*(x-y))((x+y)−((x+y)∗(x−y)))+((x+y)∗(x−y))

Step-wise DAG construction:

1. Compute x + y → Store result in a node N1.
2. Compute x - y → Store result in a node N2.
3. Compute (x+y) \* (x-y) → Use nodes N1 and N2 to create N3.
4. Compute (x+y) - N3 → Use N1 and N3 to create N4.
5. Compute N3 + N4 → The final node represents the result.

DAG removes redundant calculations by reusing N1 and N2.

**3.What is Type conversion? What are the two types of type conversion? Evaluate the rules for the type conversion.**

**Type conversion** is the process of converting one data type into another in programming languages.

#### ****Types of Type Conversion****

1. **Implicit Type Conversion (Type Promotion)**:
   * Automatically performed by the compiler.
   * Happens when assigning a smaller data type to a larger one.
   * Example: int to float.
2. **Explicit Type Conversion (Type Casting)**:
   * Manually performed using type casting.
   * Example: (float) a / b.

#### ****Rules for Type Conversion****

1. Lower-rank types are promoted to higher-rank types (e.g., int → float).
2. Conversion between compatible types is allowed (e.g., char to int).
3. Loss of precision is possible when converting from high-precision to low-precision types.
4. Explicit type conversion should be used when required to avoid ambiguity.

**4.Generate an intermediate code for the following code segment with the required syntax- directed translation scheme. if ( a > b) x = a + b else x = a – b**

Intermediate code is a low-level representation between high-level code and machine code.

#### ****Syntax-Directed Translation Scheme****

nginx

CopyEdit

if ( a > b )

x = a + b;

else

x = a - b;

##### **Three-Address Code (TAC)**

makefile

CopyEdit

t1 = a > b

if t1 goto L1

t2 = a - b

x = t2

goto L2

L1: t3 = a + b

x = t3

L2:

* **Conditional jump (if t1 goto L1)** directs execution flow.
* **Temporary variables (t1, t2, t3)** store intermediate results.

**5.Create the following for the arithmetic expression a+- (b+c)\* into (i)Syntax tree (ii)Quadruples (iii)Triples (iv)Indirect Triples**

We will create:

1. **Syntax Tree**
2. **Quadruples**
3. **Triples**
4. **Indirect Triples**

#### ****(i) Syntax Tree****

markdown

CopyEdit

+

/ \

a \*

/ \

- d

|

+

/ \

b c

#### ****(ii) Quadruples****

| **Op** | **Arg1** | **Arg2** | **Result** |
| --- | --- | --- | --- |
| + | b | c | t1 |
| - | t1 |  | t2 |
| \* | t2 | d | t3 |
| + | a | t3 | t4 |

#### ****(iii) Triples****

| **Index** | **Op** | **Arg1** | **Arg2** |
| --- | --- | --- | --- |
| (0) | + | b | c |
| (1) | - | (0) |  |
| (2) | \* | (1) | d |
| (3) | + | a | (2) |

#### ****(iv) Indirect Triples****

| **Index** | **Instruction** |
| --- | --- |
| (0) | + (b, c) |
| (1) | - (0, -) |
| (2) | \* (1, d) |
| (3) | + (a, 2) |

Bottom of Form

**UNIT-IV RUN TIME ENVIRONMENT AND CODE GENERATON**

**1.Compare static versus dynamic memory allocation.**

## ****1. Static vs. Dynamic Memory Allocation****

Memory allocation is a crucial aspect of program execution. It can be classified into **static** and **dynamic** allocation.

| **Feature** | **Static Memory Allocation** | **Dynamic Memory Allocation** |
| --- | --- | --- |
| **Definition** | Memory is allocated at compile-time. | Memory is allocated at runtime. |
| **Flexibility** | Fixed memory size, cannot be changed during execution. | Can allocate or deallocate memory as needed. |
| **Efficiency** | Faster since memory is pre-allocated. | Slower due to runtime allocation overhead. |
| **Example** | Arrays, global variables, static variables. | Heap memory, linked lists, dynamic arrays (malloc(), new). |
| **Memory Management** | Managed by compiler, no need for explicit deallocation. | Requires manual deallocation (free(), delete). |
| **Usage** | Used when memory requirements are known beforehand. | Used when memory size is unpredictable. |

**Example:**

c

CopyEdit

// Static allocation

int arr[10]; // Fixed size

// Dynamic allocation

int \*ptr = (int \*)malloc(10 \* sizeof(int)); // Allocated at runtime

## ****2.**** Explain in detail about the various issues in code generation with examples

Code generation involves translating intermediate code into machine code. However, several **issues** arise:

### ****1. Instruction Selection****

* Choosing the best machine instructions to execute a given operation.
* Example:
  + a = b + c;
  + On x86: ADD b, c
  + On ARM: ADD R1, R2, R3

### ****2. Register Allocation****

* Limited number of CPU registers.
* Variables must be stored in registers or spilled to memory.
* Example:
  + If registers are full, some values are saved to RAM, increasing execution time.

### ****3. Order of Evaluation****

* Expression evaluation order impacts efficiency.
* Example:
  + (a + b) \* (c + d)
  + Computing a + b first may reduce memory accesses.

### ****4. Handling Control Flow****

* **Jump optimizations** in loops, if-else, and switch statements.
* Example:

c

CopyEdit

if (x > 0) goto L1;

* + Can be optimized using conditional instructions.

### ****5. Code Optimization****

* Removing redundant computations and reusing subexpressions.
* Example:
  + Instead of computing x + y twice, store it in a temporary variable.

### 3. Discuss in detail about the activation tree and activation record with suitable example

### ****(i) Activation Tree****

An **activation tree** represents function calls in a program's execution.

**Example:**

c

CopyEdit

void A() {

B();

C();

}

void B() {

D();

}

void C() {}

void D() {}

**Activation Tree:**

css

CopyEdit

A

/ \

B C

/

D

* The tree shows the order in which functions are called.
* A calls B and C, while B calls D.

### ****(ii) Activation Record****

An **activation record (stack frame)** stores information about function calls.

| **Field** | **Purpose** |
| --- | --- |
| Return Address | Stores the address to return after function execution. |
| Saved Registers | Stores register values before function execution. |
| Parameters | Stores function arguments. |
| Local Variables | Stores local variables of the function. |
| Temporary Variables | Stores intermediate results. |

**Example (C Code):**

c

CopyEdit

int add(int x, int y) {

int sum = x + y;

return sum;

}

Activation Record for add():

markdown

CopyEdit

-----------------

| Return Addr |

| Saved Regs |

| x |

| y |

| sum |

-----------------

### 4. Explain in detail about instruction selection and register allocation of code generation.

### ****(i) Instruction Selection****

* The process of choosing the most efficient machine instructions for a given operation.
* Example:

c

CopyEdit

a = b + c;

* + **Possible Instructions:**
    - ADD R1, R2, R3 (RISC architecture)
    - MOV R1, b; ADD R1, c; MOV a, R1; (CISC architecture)

### ****(ii) Register Allocation****

* Assigning variables to a **limited number of CPU registers**.
* If registers are full, values are stored in **memory (spilling)**.

#### ****Techniques for Register Allocation****

1. **Graph Coloring Algorithm**: Treats registers as colors and minimizes register spills.
2. **Linear Scan Allocation**: Faster but less optimal.
3. **Greedy Allocation**: Allocates registers dynamically.

**Example (Register Allocation for Expression)**

c

CopyEdit

t1 = a + b

t2 = t1 \* c

t3 = t2 - d

Possible allocation:

* t1 → R1
* t2 → R2
* t3 → R3 If registers are full, t1 might be spilled to memory.

### 4.Illustrate in detail about the code generation algorithm with an example

### ****Steps for Code Generation****

1. **Input:** Intermediate representation (IR) of the program.
2. **Instruction Selection:** Choose machine instructions for operations.
3. **Register Allocation:** Assign registers to variables.
4. **Instruction Ordering:** Optimize instruction execution order.
5. **Generate Machine Code:** Convert IR to assembly.

### ****Example****

#### ****Given Expression:****

c

CopyEdit

a = b + c \* d;

#### ****Step 1: Convert to Three-Address Code****

ini

CopyEdit

t1 = c \* d

t2 = b + t1

a = t2

#### ****Step 2: Register Allocation****

* t1 → R1
* t2 → R2
* a → R3

#### ****Step 3: Generate Machine Code****

css

CopyEdit

MOV R1, c

MUL R1, d

MOV R2, b

ADD R2, R1

MOV a, R2

* MOV moves values to registers.
* MUL multiplies values.
* ADD adds values.
* MOV a, R2 stores result in a.

### ****Final Output: Optimized Assembly Code****

css

CopyEdit

MUL R1, c, d

ADD R2, b, R1

MOV a, R2

* Redundant moves are eliminated.
* Code is optimized for efficiency.

**UNIT-V CODE OPTIMISATION**

## ****1. Explain the Principal Sources of Optimization in Compiler Design.****

Code optimization improves program performance by reducing execution time and memory usage. The **principal sources of optimization** include:

### ****1. Redundant Code Elimination****

* Removing unnecessary computations.
* Example:

c

CopyEdit

int x = 5;

int y = 10;

int z = x + y; // z is never used

* + The computation of z can be eliminated.

### ****2. Loop Optimization****

* **Loop Invariant Code Motion**: Move computations **outside** loops if they don’t change inside.
* **Loop Unrolling**: Reduce loop overhead by executing multiple iterations at once.

**Example (Loop Invariant Code Motion)**:

c

CopyEdit

for (int i = 0; i < 100; i++) {

int x = a + b; // Invariant (does not change)

arr[i] = x \* i;

}

* Optimization: Move int x = a + b; **outside** the loop.

### ****3. Common Subexpression Elimination (CSE)****

* If an expression is computed multiple times, replace it with a single computation.
* Example:

c

CopyEdit

int a = (x + y) \* z;

int b = (x + y) \* w;

* + Optimization: Compute (x + y) once and reuse.

### ****4. Dead Code Elimination****

* Removing unused code.
* Example:

c

CopyEdit

int x = 10;

x = 20; // The first assignment is dead code.

### ****5. Strength Reduction****

* Replace expensive operations with cheaper ones.
* Example:

c

CopyEdit

int x = a \* 2; // Multiplication is expensive

* + Optimization: int x = a << 1; (Bitwise shift)

### ****6. Code Motion****

* Move code to reduce execution time.
* Example:

c

CopyEdit

if (x > 0) {

int y = 10; // This is always executed, move it outside

}

### ****Conclusion****

Using these optimizations, a compiler **reduces execution time, improves efficiency, and minimizes memory usage.**

## ****2. Explain Peep-hole Optimization with Examples.****

**Peep-hole optimization** is a **local optimization technique** that improves performance by examining **a small section of code (window) at a time** and making improvements.

### ****Techniques of Peep-hole Optimization:****

1. **Redundant Load and Store Elimination**
2. **Constant Folding**
3. **Strength Reduction**
4. **Algebraic Simplifications**
5. **Unreachable Code Elimination**

### ****1. Redundant Load and Store Elimination****

* Remove unnecessary memory operations.
* Example:

assembly

CopyEdit

MOV R1, X

MOV X, R1 // Unnecessary store

* + Optimization: Remove MOV X, R1.

### ****2. Constant Folding****

* Compute constant expressions at compile-time.
* Example:

c

CopyEdit

int x = 3 \* 4; // Replaced with x = 12;

### ****3. Strength Reduction****

* Replace expensive operations with simpler ones.
* Example:

assembly

CopyEdit

MUL R1, 2 → SHL R1, 1 // Multiplication replaced with bit shift

### ****4. Algebraic Simplifications****

* Simplify arithmetic expressions.
* Example:

c

CopyEdit

x = x \* 1; // x = x;

x = x + 0; // x = x;

### ****5. Unreachable Code Elimination****

* Remove dead code.
* Example:

c

CopyEdit

if (0) { // This block will never execute

x = 10;

}

### ****Conclusion****

Peep-hole optimization makes small but **effective** improvements in performance.

## ****3. Discuss the Optimization of Basic Blocks using DAG Representation.****

**Basic blocks** are **sequences of instructions** with:

* **No branching (except at the end).**
* **No entry except at the beginning.**

### ****Directed Acyclic Graph (DAG) Representation****

A **DAG (Directed Acyclic Graph)** is used to **optimize basic blocks** by:

1. **Eliminating common subexpressions**
2. **Reducing redundant calculations**
3. **Improving register allocation**

### ****Example:****

Given the expression:

ini

CopyEdit

a = b + c;

d = a - e;

f = b + c;

g = f - e;

* b + c is repeated.
* Optimization: Compute it **once** and reuse.

**DAG Representation:**

markdown

CopyEdit

+

/ \

b c

| |

------

|

a, f

|

-

/ \

e d, g

* Common subexpression (b + c) is **computed once**.
* a and f **share the same value**.

### ****Conclusion****

Using DAGs, **redundant computations are eliminated**, improving efficiency.

## ****4. Explain Global Data Flow Analysis and Efficient Data Flow Algorithm.****

### ****1. Global Data Flow Analysis****

Global Data Flow Analysis collects **information across basic blocks** to optimize code.

### ****Steps:****

1. **Define Data Flow Equations**:
   * Define **gen** (what is generated).
   * Define **kill** (what is removed).
2. **Solve Equations**:
   * Use iterative methods to compute data flow.
3. **Apply Optimizations**:
   * **Constant propagation**
   * **Dead code elimination**
   * **Loop invariant code motion**

### ****2. Efficient Data Flow Algorithm****

A **widely used algorithm** for data flow analysis is the **Worklist Algorithm**.

**Example:** **Live Variable Analysis**

* Determines which variables are **live** at a point.

**Data Flow Equation for Liveness:**

CopyEdit

LIVE\_OUT[B] = (LIVE\_IN[successors]) - KILL[B] + GEN[B]

### ****Example Code:****

c

CopyEdit

x = 10;

y = x + 5;

if (y > 20) {

z = y + 2;

}

* **Dead Code Elimination**: If z is not used later, remove it.

### ****Conclusion****

Data flow analysis **improves global optimizations** like **dead code elimination and constant propagation.**

## ****5. Describe the Recent Trends in Compiler Design.****

Compiler design is **evolving** with new techniques:

### ****1. Just-In-Time (JIT) Compilation****

* Translates code **at runtime** instead of compile-time.
* Used in Java and Python **(JVM, PyPy).**

### ****2. Machine Learning in Compilation****

* AI models predict **optimal code transformations**.

### ****3. Parallel and Multi-core Compilation****

* Optimizes **code execution for multi-core processors**.

### ****4. Dynamic Binary Translation****

* Converts **one machine code** to **another** at runtime.
* Example: **Apple Rosetta 2 (x86 → ARM)**.

### ****5. Cloud-based Compilation****

* Compilers are **hosted in the cloud** for faster execution.

### ****Conclusion****

Modern compilers **optimize performance, memory, and execution speed** using AI, parallelization, and JIT techniques.