**CS3501-COMPILER DESIGN**

**PART-C(15 Marks)**

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

## ****1. Explain the Phases of a Compiler with a Neat Diagram.****

A **compiler** translates high-level source code into machine code. It consists of several phases:

### ****1. Lexical Analysis (Scanner)****

* Converts the **source code** into **tokens**.
* Example: int x = 10; → <int, identifier, =, number, ;>

### ****2. Syntax Analysis (Parser)****

* **Checks grammar correctness** using **context-free grammars (CFG)**.
* Example: if (x > 0 { print(x); } → **Error (missing ))**.

### ****3. Semantic Analysis****

* Checks **meaning** (type checking, scope resolution).
* Example:

c

CopyEdit

int x;

x = "hello"; // Type mismatch error

### ****4. Intermediate Code Generation****

* Converts code into **three-address code (TAC)**.
* Example:

a = b + c \* d;

**TAC Representation:**

t1 = c \* d;

t2 = b + t1;

a = t2;

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

* **Removes redundant operations** to improve performance.
* Example: Strength reduction:

ini

CopyEdit

x = a \* 8;

Optimized:

ini

CopyEdit

x = a << 3;

### ****6. Code Generation****

* Converts TAC to **machine code**.

### ****7. Symbol Table and Error Handling****

* Keeps track of **variable names, types, scopes**.
* Handles **errors at various stages**.

### ****Diagram of Compiler Phases:****

css

CopyEdit

Source Code → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Optimizer → Code Generator → Target Code

**2.Explain in detail the phases of a compiler with a neat diagram. Discuss the role of each phase in the compilation process.**

A **compiler** is a program that converts high-level code into machine code. The compilation process consists of several phases:

**Phases of a Compiler**

1. **Lexical Analysis**
   * Converts source code into **tokens**.
   * Removes whitespaces, comments, and identifies **keywords, identifiers, and symbols**.
   * Example: int a = 10; → Tokens: int, a, =, 10, ;
2. **Syntax Analysis (Parsing)**
   * Checks the **grammatical structure** using **parsing techniques (LL, LR, etc.)**.
   * Constructs a **parse tree**.
   * Example: Ensures if (x > 10) { y = 20; } follows correct syntax.
3. **Semantic Analysis**
   * Ensures **meaning correctness** of statements.
   * Example: If int x; x = 5.5; → Gives a **type mismatch error**.
4. **Intermediate Code Generation**
   * Converts code into **Intermediate Representation (IR)**.
   * Example:

ini

CopyEdit

t1 = c \* d

t2 = b + t1

a = t2

1. **Code Optimization**
   * Improves performance by removing redundant computations.
   * Example:

ini

CopyEdit

x = y \* 2;

z = y \* 2;

→ Optimized as

ini

CopyEdit

t = y \* 2;

x = t;

z = t;

1. **Code Generation**
   * Converts IR into **machine code** (assembly code).
   * Example:

nginx

CopyEdit

MOV R1, y

MUL R1, 2

MOV x, R1

1. **Symbol Table & Error Handling**
   * **Symbol Table** stores variable names, types, memory locations.
   * **Error Handling** reports lexical, syntax, and semantic errors.

**Diagram of Compiler Phases:**

css

CopyEdit

Source Code

↓

Lexical Analyzer → Tokens

↓

Syntax Analyzer → Parse Tree

↓

Semantic Analyzer → AST

↓

Intermediate Code Generator → IR

↓

Code Optimizer → Optimized IR

↓

Code Generator → Machine Code

↓

Target Program

**3.Explain Lexical Analysis and its role in a compiler. Discuss input buffering techniques and how tokens are recognized using finite automata.**

**Lexical Analysis**

* The **first phase** of a compiler.
* Converts **source code** into a sequence of **tokens**.
* Removes **whitespaces, comments** and identifies **keywords, identifiers, operators**.
* Uses **Finite Automata** for pattern recognition.

**Role of Lexical Analyzer**

1. Reads input **character-by-character**.
2. Groups characters into **lexemes** and converts them into **tokens**.
3. Ignores **whitespaces and comments**.
4. Passes tokens to the **syntax analyzer**.

Example:  
For int a = 10;, tokens generated:

* int → **Keyword**
* a → **Identifier**
* = → **Assignment Operator**
* 10 → **Constant**
* ; → **Delimiter**

**Input Buffering in Lexical Analysis**

* Used to handle **large source code efficiently**.
* Avoids **backtracking** using two buffering techniques:

1. **Two-Buffer Scheme**
   * Uses **two buffers** to read input.
   * One buffer is processed while the other is being filled.
   * Helps in recognizing **multi-character tokens**.
2. **Sentinel Method**
   * Uses a **special end marker** (EOF) instead of checking buffer limits repeatedly.

**Token Recognition Using Finite Automata**

**Finite Automata**

* **DFA (Deterministic Finite Automata)** and **NFA (Non-Deterministic Finite Automata)** are used for **pattern matching** in lexical analysis.

**Example: Recognizing an Identifier ([a-zA-Z][a-zA-Z0-9]\*)**

pgsql

CopyEdit

State 0 → Start

State 1 → Letter found (Valid Identifier Start)

State 2 → Letter/Digit found (Continue Identifier)

Final State → Accept Identifier

**Transition Table for DFA:**

| **State** | **Input (Letter)** | **Input (Digit)** | **Other Input** |
| --- | --- | --- | --- |
| 0 | 1 (Move) | - (Reject) | - (Reject) |
| 1 | 1 (Stay) | 2 (Move) | - (Reject) |
| 2 | 2 (Stay) | 2 (Stay) | - (Accept) |

**Lex Tool**

* A **tool for specifying lexical analyzers**.
* Generates a **C-based lexical analyzer** from regular expressions.

**Structure of a Lex Program**

1. **Definition Section**

cpp

CopyEdit

%{

#include <stdio.h>

%}

1. **Rules Section**

scss

CopyEdit

[0-9]+ printf("NUMBER");

[a-zA-Z][a-zA-Z0-9]\* printf("IDENTIFIER");

1. **Main Function**

csharp

CopyEdit

int main() {

yylex();

return 0;

}

**Example Lex Program to Identify Numbers and Identifiers**

lex

CopyEdit

%{

#include <stdio.h>

%}

%%

[0-9]+ printf("NUMBER\n");

[a-zA-Z][a-zA-Z0-9]\* printf("IDENTIFIER\n");

. ;

%%

int main() {

yylex();

return 0;

}

**Output:**

makefile

CopyEdit

Input: 123 abc

Output:

NUMBER

IDENTIFIER

**Conclusion**

* Lexical Analysis **breaks source code into tokens**.
* Uses **Finite Automata** for pattern recognition.
* **Input Buffering** helps handle large code efficiently.
* **Lex Tool** automates lexical analysis for compiler designBottom of Form

# ****Unit II: Syntax Analysis****

## ****1. Explain Top-Down Parsing and Bottom-Up Parsing with Examples.****

Parsing is the process of **analyzing the syntax** of a program.

### ****1. Top-Down Parsing****

* **Starts from the root** of the parse tree and derives the input.
* **Types:**
  + **Recursive Descent Parsing**
  + **LL(1) Parsing**

#### ****Example: Recursive Descent Parsing****

Grammar:

E → T + E | T

T → F \* T | F

F → (E) | id

String: id + id \* id

1. Match id with F → id
2. Match \* using T → F \* T
3. Match id with F → id
4. Match + using E → T + E

### ****2. Bottom-Up Parsing****

* **Starts from the input** and reduces it **to the start symbol**.
* **Types:**
  + **Shift-Reduce Parsing**
  + **LR Parsing (SLR, LALR, CLR)**

#### ****Example: Shift-Reduce Parsing****

Grammar:

r

CopyEdit

E → E + T | T

T → T \* F | F

F → id

Input: id + id \* id

| **Step** | **Stack** | **Input** | **Action** |
| --- | --- | --- | --- |
| 1 | id | + id \* id | Shift |
| 2 | F | + id \* id | Reduce F → id |
| 3 | T | + id \* id | Reduce T → F |
| 4 | E | + id \* id | Reduce E → T |
| 5 | E + | id \* id | Shift |
| 6 | E + id | \* id | Shift |
| 7 | E + F | \* id | Reduce F → id |
| 8 | E + T | \* id | Reduce T → F |
| 9 | E + T \* | id | Shift |
| 10 | E + T \* id |  | Shift |
| 11 | E + T \* F |  | Reduce F → id |
| 12 | E + T |  | Reduce T → T \* F |
| 13 | E |  | Reduce E → E + T |

**Final Output:** **Accepted Parse Tree**

### 2.Explain the role of a parser in a compiler. What is a context-free grammar (CFG), and how do we write grammars? Discuss top-down parsing techniques with examples.

**Role of a Parser**

A **parser** is a phase in the compiler that checks the **syntax** of a given program based on predefined rules (grammar). It takes tokens from the **lexical analyzer** and ensures the **source code follows the grammar rules** of the programming language.

**Functions of a Parser:**

1. **Syntax Checking:** Ensures the program follows grammatical rules.
2. **Parse Tree Construction:** Generates a **hierarchical structure** of input.
3. **Error Detection & Recovery:** Reports and corrects **syntax errors**.
4. **Semantic Processing:** Passes structure to the **semantic analyzer**.

**Context-Free Grammar (CFG)**

A **CFG** consists of:

1. **Terminals (T)** → Actual symbols (e.g., a, b, +, \*, id).
2. **Non-terminals (N)** → Variables that define the structure.
3. **Start Symbol (S)** → The starting point of derivations.
4. **Production Rules (P)** → Rules defining replacements.

**Example CFG:**

r

CopyEdit

E → E + T | T

T → T \* F | F

F → (E) | id

This grammar generates arithmetic expressions.

**Top-Down Parsing Techniques**

**Top-down parsing** starts from the start symbol and derives the input string.

**1. Recursive Descent Parsing**

* Uses **recursion** to match input with grammar rules.
* Works well for **LL(1) grammars** (single-symbol lookahead).

**Example: Parsing E → E + T | T**  
If input is id + id, it recursively applies:

1. E → E + T
2. E → T
3. T → id
4. T → id

**2. Predictive Parsing (LL(1))**

* Uses a **parsing table** to predict the next rule.
* Eliminates **backtracking**.
* Requires **left-factored** and **non-left-recursive** grammar.

**Example LL(1) Table for E → E + T | T**

|  | **id** | **+** | **\*** | **(** | **)** | **$** |
| --- | --- | --- | --- | --- | --- | --- |
| **E** | E → T |  |  | E → T |  |  |
| **T** | T → F |  |  | T → F |  |  |
| **F** | F → id |  |  | F → (E) |  |  |

**3.Explain Shift-Reduce Parsing and LR parsing techniques. How is an SLR parsing table constructed? Discuss error handling in syntax analysis.**

**Shift-Reduce Parsing**

* A **bottom-up** parsing technique.
* Uses **a stack** and input buffer.
* **Shift:** Moves input symbol onto the stack.
* **Reduce:** Applies grammar rules when a valid pattern appears.

**Example:**  
For id \* id + id, the stack operations are:

shell

CopyEdit

Stack | Input | Action

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

$ | id \* id + id$ | Shift id

$ id | \* id + id$ | Reduce F → id

$ F | \* id + id$ | Reduce T → F

$ T | \* id + id$ | Shift \*

$ T \* | id + id$ | Shift id

$ T \* id | + id$ | Reduce F → id

$ T \* F | + id$ | Reduce T → T \* F

$ T | + id$ | Shift +

$ T + | id$ | Shift id

$ T + id | $ | Reduce F → id

$ T + F | $ | Reduce E → T + F

$ E | $ | Accept

**LR Parsing Techniques**

LR parsers **(Left-to-right scanning, Rightmost derivation)** are efficient **bottom-up** parsers.

**1. LR(0) Parser**

* Uses **LR(0) items** (A → α•β).
* Simple but cannot handle **many real-world grammars**.

**2. SLR (Simple LR) Parser**

* Uses **LR(0) items** and **Follow sets** for reduction.
* Easier to implement but **less powerful** than LALR or CLR.

**3. LALR (Lookahead LR) Parser**

* Combines **states of SLR parser** to optimize memory usage.
* Used in **YACC parser generators**.

**4. CLR (Canonical LR) Parser**

* Most powerful but has **complex parsing tables**.
* Uses **LR(1) items** (with lookahead symbols).

**SLR Parsing Table Construction**

1. **Compute LR(0) items** for grammar rules.
2. **Construct a DFA** for shift operations.
3. **Use Follow sets** to determine reductions.

**Example Grammar:**

less

CopyEdit

S → AA

A → aA | b

**Step 1: LR(0) Items**

vbnet

CopyEdit

S' → •S

S → •AA

A → •aA

A → •b

**Step 2: DFA Construction**

| **State** | **Input a** | **Input b** | **Input $** |
| --- | --- | --- | --- |
| I0 | Shift I1 | Shift I2 |  |
| I1 | Shift I3 | Reduce A → b |  |
| I2 | Reduce A → b |  |  |

**Step 3: SLR Parsing Table**

| **State** | **a** | **b** | **$** | **A** |
| --- | --- | --- | --- | --- |
| I0 | S1 | S2 |  | 3 |
| I1 | S4 | R(A→b) |  | 5 |

**Error Handling in Syntax Analysis**

1. **Panic Mode Recovery**
   * Skip symbols until a **synchronization token** (e.g., ; or {) is found.
2. **Phrase-Level Recovery**
   * Modify the erroneous token and continue parsing.
3. **Error Productions**
   * Extend grammar to handle common errors.
4. **Global Correction**
   * Suggest minimum changes for a valid input.

**YACC Tool (Yet Another Compiler Compiler)**

* A tool to **generate parsers** from **CFGs**.
* Works with **Lex** to generate full-fledged **syntax analyzers**.

**Example YACC Program**

yacc

CopyEdit

%token NUMBER

%%

expr : expr '+' term

| term

;

term : term '\*' factor

| factor

;

factor : NUMBER

| '(' expr ')'

;

%%

int main() {

yyparse();

}

* Parses arithmetic expressions like 5 + 2 \* 3.

# ****Unit III: Syntax-Directed Translation & Intermediate Code Generation****

## ****1. Explain Syntax Directed Translation with an Example.****

Syntax Directed Translation (SDT) **associates attributes** with grammar symbols and uses **semantic rules** to compute values.

### ****Types of Attributes:****

1. **Synthesized Attributes** - Computed from **child nodes**.
2. **Inherited Attributes** - Computed from **parent or siblings**.

### ****Example: Syntax-Directed Definition (SDD)****

Grammar:

r

CopyEdit

E → E1 + T { E.val = E1.val + T.val }

E → T { E.val = T.val }

T → T1 \* F { T.val = T1.val \* F.val }

T → F { T.val = F.val }

F → (E) { F.val = E.val }

F → num { F.val = num.lexval }

For input: **2 + 3 \* 4**

1. 3 \* 4 = 12
2. 2 + 12 = 14
3. **Output: 14**

### ****Intermediate Code Generation****

Converts the syntax tree into **Three-Address Code (TAC).**

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

c

CopyEdit

a = b + c \* d;

**TAC Representation:**

ini

CopyEdit

t1 = c \* d;

t2 = b + t1;

a = t2;

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

SDT is crucial for **semantic analysis and intermediate code generation.**

**2.Explain Syntax-Directed Definitions (SDDs) and their role in compiler design. How is a syntax tree constructed, and how does bottom-up evaluation of S-attributed definitions work?**

**Syntax-Directed Definitions (SDDs)**

* **SDD** associates **semantic rules** with grammar productions.
* It helps in **semantic analysis, type checking, and intermediate code generation**.

**Components of SDD:**

1. **Syntax Rules:** CFG rules that define grammar structure.
2. **Attributes:**
   * **Synthesized Attributes (S-attributes)** → Computed from **children nodes**.
   * **Inherited Attributes** → Computed from **parent or siblings**.
3. **Semantic Rules:** Define computations for attributes.

**Example SDD for Arithmetic Expressions**

r

CopyEdit

E → E1 + T { E.val = E1.val + T.val }

E → T { E.val = T.val }

T → T1 \* F { T.val = T1.val \* F.val }

T → F { T.val = F.val }

F → (E) { F.val = E.val }

F → num { F.val = num.lexval }

* Here, .val is a synthesized attribute storing the value of an expression.

**Construction of Syntax Tree**

A **syntax tree** represents the hierarchical structure of an expression.

**Example: Expression a + b \* c**

1. Grammar:

r

CopyEdit

E → E + T

T → T \* F

F → id

1. Syntax Tree:

css

CopyEdit

+

/ \

a \*

/ \

b c

* Operators are **internal nodes**, operands are **leaves**.

**Bottom-up Evaluation of S-attributed Definitions**

* **S-attributed definitions** use **only synthesized attributes**.
* Evaluated **during parsing (postfix evaluation).**
* **Bottom-up Parsing (Shift-Reduce) builds the tree from leaves to root.**

**Example: Computing the value of 3 + 5 \* 2**

| **Stack** | **Input** | **Action** | **Computed Attributes** |
| --- | --- | --- | --- |
| 3 | + 5 \* 2 | Shift | F.val = 3 |
| 3 + | 5 \* 2 | Shift |  |
| 3 + 5 | \* 2 | Shift | F.val = 5 |
| 3 + 5 \* | 2 | Shift |  |
| 3 + 5 \* 2 |  | Reduce T → F \* F | T.val = 5 \* 2 = 10 |
| 3 + T |  | Reduce E → E + T | E.val = 3 + 10 = 13 |

Final result: 13.

**3.Explain Intermediate Languages and discuss Three-Address Code (TAC). How is backpatching used in translation of expressions?**

**Intermediate Languages**

* **Intermediate Representation (IR)** is between **source code and machine code**.
* Used for **optimization** and **target-independent analysis**.
* Common forms:
  + **Syntax Tree**
  + **Three-Address Code (TAC)**
  + **Postfix Notation**

**Three-Address Code (TAC)**

* **TAC** breaks complex expressions into **simple three-operand statements.**
* Each statement follows:

ini

CopyEdit

x = y op z

* Uses **temporary variables (t1, t2 etc.)**.

**Example: Convert a + b \* c to TAC**

ini

CopyEdit

t1 = b \* c

t2 = a + t1

* Operators are computed **step by step**.

**TAC for if (a < b) x = y + z; else x = y - z;**

vbnet

CopyEdit

if a < b goto L1

x = y - z

goto L2

L1: x = y + z

L2:

* Uses **labels (L1, L2)** for conditional jumps.

**Type Checking**

* Ensures **type compatibility** in expressions.
* **Types:**
  + **Basic types:** int, float, char
  + **Composite types:** arrays, structures
* **Example:** int x; float y; x = y;
  + Requires **type conversion (float → int)**.

**Backpatching**

* Used for **filling in jump addresses** in **TAC**.
* Needed for **Boolean expressions and loops**.

**Example: if (a < b) x = 1; else x = 0;**

1. **Generate TAC with missing jumps**

vbnet

CopyEdit

if a < b goto \_ (L1)

x = 0

goto \_ (L2)

L1: x = 1

L2:

1. **Backpatch by filling \_ with correct labels**

vbnet

CopyEdit

if a < b goto L1

x = 0

goto L2

L1: x = 1

L2:

* **L1 and L2 are filled later during code generation.**

# ****Unit IV: Run-Time Environment & Code Generation****

## ****1. Explain Storage Allocation Strategies in Run-Time Environment.****

Memory management is essential in a compiler’s **run-time environment**.

### ****1. Static Allocation****

* **Memory is fixed at compile-time.**
* Used for **global variables**.
* Example:

c

CopyEdit

int a, b;

### ****2. Stack Allocation****

* **Memory is allocated at runtime using a stack.**
* Used for **local variables and function calls**.
* Example:

c

CopyEdit

void func() {

int x; // Allocated on the stack

}

### ****3. Heap Allocation****

* **Memory is allocated dynamically at runtime.**
* Used for **objects, dynamic arrays**.
* Example:

c

CopyEdit

int \*p = (int\*) malloc(sizeof(int));

### ****Parameter Passing Methods****

1. **Call by Value** - Function receives a **copy**.
2. **Call by Reference** - Function receives **address**.
3. **Call by Name** - **Lazy evaluation** of arguments.

**2.Explain different Storage Allocation Strategies in runtime environments. How are parameters passed in function calls?**

**Storage Organization in Runtime Environment**

A **runtime environment** manages memory allocation during program execution. Memory is divided into:

1. **Code Segment** → Stores compiled program code.
2. **Static Data Segment** → Stores global and static variables.
3. **Stack** → Stores local variables and function call data.
4. **Heap** → Stores dynamically allocated memory.

**Storage Allocation Strategies**

**1. Static Allocation**

* Memory is assigned **at compile-time** and does not change.
* Used for **global and static variables**.
* No memory reuse.
* **Example:**

c

CopyEdit

int a; // Allocated at compile-time

* **Advantage:** Simple and fast.
* **Disadvantage:** No recursion or dynamic memory usage.

**2. Stack Allocation**

* Memory is allocated/deallocated **during function calls (LIFO order)**.
* Used for **local variables and function parameters**.
* Supports **recursion**.
* **Example:**

c

CopyEdit

void fun() {

int x; // Allocated on stack

}

* **Advantage:** Fast and efficient.
* **Disadvantage:** Limited size; variables disappear after function returns.

**3. Heap Allocation**

* Memory is allocated **dynamically** during execution.
* Used for **data structures (linked lists, trees, etc.).**
* **Example:**

c

CopyEdit

int \*ptr = (int \*)malloc(sizeof(int)); // Heap allocation

* **Advantage:** Flexible, can grow/shrink.
* **Disadvantage:** Slow due to fragmentation and garbage collection.

**Parameter Passing in Function Calls**

**1. Call by Value**

* Copies **actual argument** into function parameter.
* Changes inside function **do not affect** original variable.
* **Example:**

c

CopyEdit

void fun(int x) { x = x + 5; } // Only modifies local copy

**2. Call by Reference**

* Passes **address of argument** to the function.
* Changes **reflect in original variable**.
* **Example:**

c

CopyEdit

void fun(int \*x) { \*x = \*x + 5; } // Modifies original variable

**3. Call by Name (Lazy Evaluation)**

* Parameters are **not evaluated until used**.
* Used in **functional languages (e.g., Haskell).**

**4. Call by Need**

* Like **Call by Name**, but caches the result after first evaluation.

**3.Explain the design of a simple code generator and discuss basic blocks, flow graphs, and optimal code generation using dynamic programming.**

**Code Generation in Compiler**

A **Code Generator** translates **intermediate representation (IR)** into **machine code** or assembly.

**Challenges in Code Generation**

1. **Efficiency** → Generate optimized code.
2. **Correctness** → Preserve program semantics.
3. **Register Allocation** → Minimize memory usage.
4. **Instruction Selection** → Choose best CPU instructions.

**Basic Blocks and Flow Graphs**

**1. Basic Block**

* A **sequence of statements** where:
  + **Control enters at the beginning**.
  + **Control leaves at the end**.
  + **No branching inside**.
* Example Basic Block:

c

CopyEdit

t1 = a + b;

t2 = t1 \* c;

d = t2 + e;

* **Uses**: Optimization, Dead Code Elimination.

**2. Flow Graph**

* A **directed graph** where:
  + **Nodes** = Basic Blocks.
  + **Edges** = Control flow (jumps, branches).
* **Example Flow Graph:**

markdown

CopyEdit

B1

/ \

B2 B3

\ /

B4

* **Uses**: Loop detection, Optimization.

**Design of a Simple Code Generator**

A simple code generator follows these steps:

1. **Intermediate Representation (IR)**
   * Uses **Three-Address Code (TAC)**.
   * Example: t1 = a + b.
2. **Instruction Selection**
   * Choose **efficient CPU instructions**.
   * Example: ADD R1, R2, R3 (Assembly for t1 = a + b).
3. **Register Allocation**
   * Allocate **registers efficiently** to minimize memory access.
4. **Code Emission**
   * Generate final machine code.

**Optimal Code Generation for Expressions**

* **Goal:** Generate efficient assembly for arithmetic expressions.
* **Approach:** Convert **expressions into Directed Acyclic Graphs (DAGs)**.

**Example: Expression a + (b \* c) + d**

1. **Convert to DAG:**

css

CopyEdit

+

/ \

+ d

/ \

a \*

/ \

b c

1. **Optimal Code Generation:**

pgsql

CopyEdit

LOAD R1, b

MUL R1, c

ADD R1, a

ADD R1, d

**Dynamic Programming for Code Generation**

* **Uses a table to find the best instruction sequence**.
* Example: For a \* (b + c), store optimal solutions for subproblems.

**Steps in Dynamic Programming Code Generation**

1. **Break down expression into subproblems.**
2. **Solve smaller expressions first.**
3. **Reuse computed results for efficiency.**

**Example: a \* (b + c)**

1. Compute t1 = b + c
2. Compute t2 = a \* t1
3. Store t1, t2 results to avoid recomputation.

**UNIT V CODE OPTIMISATION**

**1. Explain the Principal Sources of Code Optimization with Suitable Examples.**

Code optimization **improves execution speed and reduces memory usage**. The major sources of optimization include:

**1. Common Subexpression Elimination (CSE)**

* If an expression is **computed multiple times**, store it once and reuse it.
* Example:

c

CopyEdit

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

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

**Optimization:**

c

CopyEdit

int temp = x + y;

int a = temp \* z;

int b = temp \* w;

**2. Dead Code Elimination**

* Removes **unused** variables or statements.
* Example:

c

CopyEdit

int x = 10;

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

**3. Loop Optimization**

* **Loop Invariant Code Motion:** Move calculations **outside** the loop.
* **Loop Unrolling:** Reduce loop control overhead.

**Example (Before Optimization):**

c

CopyEdit

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

int temp = a \* b; // This does not change

arr[i] = temp + i;

}

**Optimized Code:**

int temp = a \* b;

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

arr[i] = temp + i;

}

**4. Strength Reduction**

* Replace **expensive operations** with cheaper alternatives.
* Example:

c

CopyEdit

int x = a \* 8; // Multiplication is expensive

**Optimization:**

c

CopyEdit

int x = a << 3; // Bitwise left shift is faster

**5. Constant Folding**

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

c

CopyEdit

int a = 5 \* 10; // Replaced with int a = 50;

**6. Code Motion**

* Move code **out of loops** if possible.
* Example:

c

CopyEdit

if (x > 0) {

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

}

**Conclusion**

Using **these optimizations**, compilers generate **faster and memory-efficient** machine code.

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

Peep-hole optimization is a **local optimization technique** that scans **a small set of instructions** (window) and replaces them with optimized instructions.

**Peep-hole Optimization Techniques:**

**1. Redundant Load and Store Elimination**

* Remove unnecessary memory operations.
* Example:

assembly

CopyEdit

MOV R1, X

MOV X, R1 // Unnecessary store

**Optimized:**

assembly

CopyEdit

MOV R1, X

**2. Constant Folding**

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

c

CopyEdit

int x = 3 \* 4; // Replaced with int 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. Jump Optimization**

* Remove unnecessary jumps.
* Example:

assembly

CopyEdit

JUMP L1

L1: ADD R1, R2

**Optimized:**

assembly

CopyEdit

ADD R1, R2

**6. Unreachable Code Elimination**

* Remove code that **never executes**.
* Example:

c

CopyEdit

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

x = 10;

}

**7. Null Sequence Removal**

* Remove **unnecessary operations**.
* Example:

assembly

CopyEdit

MOV R1, R1 // This does nothing

**Conclusion**

Peep-hole optimization **enhances performance** by refining **small sections** of code.

**3. Discuss Global Data Flow Analysis and Explain an Efficient Data Flow Algorithm with an Example.**

Global Data Flow Analysis **examines the flow of data across basic blocks** to improve optimization.

**1. Steps in Global Data Flow Analysis**

1. **Define Data Flow Equations**
   * **GEN[B]**: What a block generates.
   * **KILL[B]**: What a block removes.
   * **IN[B]** and **OUT[B]**: Define data flow.
2. **Solve Equations**
   * Use **iterative** methods to compute data flow.
3. **Apply Optimizations**
   * **Dead code elimination**
   * **Constant propagation**
   * **Loop invariant code motion**

**2. Efficient Data Flow Algorithm**

A widely used algorithm is the **Worklist Algorithm**.

**Example: Live Variable Analysis**

* Determines which **variables are used after a statement**.

**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;

}

**Step 1: Construct Basic Blocks**

vbnet

CopyEdit

B1: x = 10;

B2: y = x + 5;

B3: if (y > 20) goto B4;

B4: z = y + 2;

**Step 2: Compute GEN and KILL Sets**

| **Basic Block** | **GEN** | **KILL** |
| --- | --- | --- |
| B1 | x | - |
| B2 | y | x |
| B3 | - | - |
| B4 | z | - |

**Step 3: Compute LIVE\_IN and LIVE\_OUT**

Using **iteration**:

CopyEdit

LIVE\_IN[B4] = { y }

LIVE\_OUT[B3] = { y }

LIVE\_OUT[B2] = { y }

LIVE\_OUT[B1] = { y }

**Step 4: Apply Dead Code Elimination**

* If z is not used later, remove z = y + 2;

**Conclusion**

Global Data Flow Analysis helps in **constant propagation, dead code removal, and redundancy elimination**.