◆ 1. Loop Jamming (also called *Loop Fusion*)

Definition:
Loop jamming combines two or more loops that iterate over the same range into a single loop. It helps reduce loop overhead and can improve cache performance.

Example:
Before Loop Jamming:

```c
CopyEdit
for (int i = 0; i < n; i++)
    a[i] = b[i] + 1;

for (int i = 0; i < n; i++)
    c[i] = d[i] - 1;
```

After Loop Jamming:

```c
CopyEdit
for (int i = 0; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = d[i] - 1;
}
```

Advantage: Fewer loop control instructions and better data locality.

◆ 2. Loop Unrolling

Definition:
Loop unrolling replicates the loop body multiple times to decrease the overhead of loop control (i.e., `i++`, `i<n`, etc.) and increase speed.

Example:
Before Unrolling:

```c
CopyEdit
for (int i = 0; i < 4; i++)
    a[i] = a[i] + 1;
```

After Unrolling:

```c
CopyEdit
a[0] = a[0] + 1;
a[1] = a[1] + 1;
a[2] = a[2] + 1;
a[3] = a[3] + 1;
```

Advantage: Reduces loop control overhead and can increase performance, especially in small loops.

◆ 3. Loop Optimization

Definition:
Loop optimization techniques are used to **improve the efficiency** of loops in programs by **reducing the number of iterations** or **moving invariant code** outside the loop.

Common Loop Optimization Techniques:

- Loop jamming
- Loop unrolling
- Code motion
- Strength reduction
- Loop invariant code motion

Example:

```
c
CopyEdit
for (int i = 0; i < n; i++)
    y = x * 5;
```

After loop invariant code motion:

```
c
CopyEdit
y = x * 5;
for (int i = 0; i < n; i++);
```

◆ 4. Code Motion

Definition:
Code motion is an optimization technique that moves **invariant expressions** (i.e., computations whose result doesn’t change during loop iterations) **outside the loop**.

Example:
Before Code Motion:

```
c
CopyEdit
for (int i = 0; i < n; i++)
    y = a * b;
```

After Code Motion:

```
c
CopyEdit
```

```
y = a * b;
for (int i = 0; i < n; i++);
```

Advantage: Avoids recomputing the same value in each iteration, thus saving time.

◆ 5. Difference Between Common Subexpression Elimination (CSE) and Dead Code Elimination (DCE)

Feature	Common Subexpression Elimination (CSE)	Dead Code Elimination (DCE)
Definition	Removes repeated expressions by reusing already computed values.	Removes code that does not affect program output.
Purpose	Avoid recomputing expressions whose values are already known.	Remove unnecessary code to reduce program size and time.
Example	`a = b + c; ... d = b + c;` → reuse `a` for `d`.	`int x = 5;` (if `x` is never used again, it's removed).
Benefit	Saves computation time by avoiding redundant work.	Saves memory and execution time by removing unused code.

Let me know if you'd like diagrams, MCQs, or a short handwritten-style PDF summary for revision.

4o

You said:

Diff with example

ChatGPT said:

Here is a **clear difference between Common Subexpression Elimination (CSE)** and **Dead Code Elimination (DCE)** along with **examples**, suitable for your **semester exam**:

✅ Difference between Common Subexpression Elimination and Dead Code Elimination

Aspect	Common Subexpression Elimination (CSE)	Dead Code Elimination (DCE)
Definition	Eliminates expressions that are **computed multiple times** with **same operands and values**.	Eliminates code that **does not affect** the **program's output**.
Goal	Avoid recomputation of identical expressions.	Remove unnecessary or unreachable code.

Aspect	Common Subexpression Elimination (CSE)	Dead Code Elimination (DCE)
Focus	Efficiency in computation.	Code size reduction and cleanup.
Type of Optimization	**Redundancy removal**	**Useless code removal**
When Applied	When a subexpression is **repeated** and its variables haven't changed.	When a **variable is assigned** a value but **never used**.

📘 Example of CSE (Common Subexpression Elimination)

Before:

```
c
CopyEdit
a = b + c;
...
d = b + c;
```

After:

```
c
CopyEdit
a = b + c;
...
d = a;   // reused the earlier result
```

✅ **Explanation**:
The expression `b + c` is computed twice. Since `b` and `c` haven't changed, the second calculation is redundant.

📘 Example of DCE (Dead Code Elimination)

Before:

```
c
CopyEdit
int a = 10;
int b = 20;
a = b + 5;
```

After:

```
c
CopyEdit
int b = 20;
```

✅ **Explanation**:
Variable `a` is never used after being assigned. So, assigning `a` is **dead code** and can be eliminated.

Let me know if you'd like a visual diagram or handwritten-style summary for this!

4o

You said:

What is DAG Advantages of DAG Peephole optimization Define basic blocks Write down the steps involved in dividing the code into basic blocks Explain how commutativity and associativity can be used to generate more efficient code from the DAG by taking suitable source code

ChatGPT said:

Here are clear, **exam-ready answers** for each of the compiler design terms you mentioned:

✅ 1. What is DAG (Directed Acyclic Graph)?

A **DAG** is a graphical representation of expressions where:

- **Nodes** represent **operators** or **operands**.
- It shows how expressions are **computed** without **redundancy**.
- It contains **no cycles**, hence "acyclic".

✏️ Example:

For the expression:
`a = (b + c) * (b + c)`

DAG Representation:

```
r
CopyEdit
       (*)
      /   \
    (+)    (+)
   /   \ /    \
  b     c     b    c   <- (redundant reused)
```

☞ Both `b + c` are **common subexpressions**, shown only **once** in the DAG.

✅ 2. Advantages of DAG

- **Eliminates redundant computations** (common subexpressions).
- **Optimizes code** by reusing values.
- **Helps in register allocation**.
- **Enables better code generation**.
- **Improves performance** by reducing the number of instructions.

✅ 3. Peephole Optimization

A **local optimization** technique where a **small set of instructions** (a "peephole") is analyzed and replaced with a **more efficient sequence**.

✏️ Example:

Before:

```
asm
CopyEdit
MOV R0, A
MOV A, R0
```

After:

```
asm
CopyEdit
// Removed redundant move
// No code needed
```

✔ Common peephole optimizations:

- Redundant instruction elimination
- Constant folding
- Strength reduction (e.g., `x * 2 → x + x`)
- Algebraic simplification

✅ 4. Define Basic Blocks

A **basic block** is a **sequence of statements** in a program that:

- **Has only one entry point** (first statement)
- **Has only one exit point** (last statement)
- **No branching** except at the end

✔ All control flows enter at the **start** and leave at the **end**.

✅ 5. Steps to Divide Code into Basic Blocks

Given a list of statements:

1. **Identify Leaders**:
   - First statement is always a leader.
   - Any target of a jump is a leader.
   - Any statement following a jump or branch is a leader.
2. **Group into Basic Blocks**:
   - Start from each leader and include all statements up to **(but not including)** the next leader or end of program.

✅ 6. Using Commutativity and Associativity in DAG

🔄 Commutativity:

- `a + b` = `b + a`
- `a * b` = `b * a`

✔ So, `x = a + b` and `y = b + a` are **same** → combine.

🔄 Associativity:

- `(a + b) + c = a + (b + c)`

✔ Useful for **reordering** computations for optimal code generation.

✏ Example Code (with DAG Optimization using commutativity & associativity)

```c
c
CopyEdit
x = a + b + c;
y = b + a + c;
```

Using:

- Commutativity: `a + b` = `b + a`
- Associativity: `(a + b) + c` = `a + (b + c)`

☞ Both `x` and `y` have **same computation** → can reuse result.

1. Symbol Table Organization

A **symbol table** is a data structure used by a compiler to store information about identifiers (e.g., variable names, function names, classes) such as type, scope, memory location, etc.

Approaches for Symbol Table Organization:

1. **Linear List (Sequential Search Table):**
   - A simple list where entries are stored one after the other.
   - **Search Time**: O(n) in the worst case.
   - **Example**:

     ```
     plaintext
     CopyEdit
     Symbols: [a, b, c, d]
     Search "c" → Compare a, b, then c (3 comparisons)
     ```

   - **Use case**: Simple compilers or when the number of symbols is small.
2. **Binary Search Tree (BST):**
   - Stores symbols in a sorted binary tree.
   - **Search Time**: O(log n) on average.
   - Maintains order, supports efficient search, insert, delete.
   - **Example**:

     ```
     plaintext
     CopyEdit
         b
        / \
       a   d
          /
         c
     Search "c" → b → d → c (3 steps)
     ```

3. **Hash Table:**
   - Uses a **hash function** to map identifiers to indices in a table.
   - **Search Time**: O(1) on average.
   - Handles collisions using techniques like chaining or open addressing.
   - **Example**:
     - Symbols: a, b, c
     - Hash(a) = 1, Hash(b) = 3, Hash(c) = 1 (collision handled)
4. **Trie (Prefix Tree):**
   - A tree where each node represents a character of a string.
   - Efficient for identifiers with common prefixes.
   - **Example**:

     ```
     plaintext
     CopyEdit
     For "int", "input", "index":
     root → i → n → t
                  → d → e → x
               → p → u → t
     ```

---

2. Hash Functions and Their Criteria

Principal Criteria to Choose a Hash Function:

- Should distribute identifiers uniformly across the hash table.
- Should be fast to compute.
- Should minimize collisions.
- Deterministic (always gives the same output for same input).

Three Hash Functions:

1. **Mid-Square Method:**
   - Square the key, extract a portion from the middle of the result.
   - **Example**: Key = 123, square = 15129 → middle 3 digits = 151 → Hash = 151 % table size.
   - Advantage: Uses all parts of the key, good distribution.
2. **Folding Method:**
   - Divide the key into parts, sum them.
   - **Example**: Key = 123456 → divide into 123 and 456 → 123 + 456 = 579 → Hash = 579 % table size.
   - Can also reverse some parts for better distribution.
3. **Division Method:**
   - Simple and commonly used: `Hash = key % table_size`
   - **Example**: Key = 12345, table size = 100 → Hash = 12345 % 100 = 45
   - Choose table size as a prime number for better spread.

---

3. Storage Allocation Techniques

Storage allocation refers to how memory is assigned to programs and data during compilation and execution.

Three Techniques:

1. **Static Allocation:**
   - Memory is allocated at compile time.
   - Fixed size and location.
   - Used for global variables and constants.
   - **Example**: `int x = 5;` (x has a fixed address)
2. **Stack Allocation:**
   - Memory allocated and deallocated in a last-in, first-out (LIFO) manner.
   - Used for local variables, function parameters.
   - Each function call gets a **stack frame**.
   - Efficient but cannot handle dynamic structures well.
3. **Heap Allocation:**
   - Memory allocated at runtime.
   - Supports dynamic data structures like linked lists, trees.
   - Requires manual allocation/deallocation (`malloc`/`free` in C).
   - More flexible, but less efficient than stack.

Summary Table:

Component	Description
Symbol Table Approaches	Linear List, BST, Hash Table, Trie
Hash Functions	Mid-Square, Folding, Division
Storage Allocation	Static, Stack, Heap

✅ 1. Desirable Features of Error Reporter (with Examples)

The **error reporter** in a compiler is responsible for detecting and reporting errors in the source code. Good error reporting improves developer productivity and makes debugging easier.

◆ Desirable Features:

1. **Accuracy**:
   - Clearly identify the exact location and type of error.
   - Example:

     ```
     c
     CopyEdit
     int a = 5
     ```

     Error: `Missing semicolon at line 1`

2. **Clarity**:
   - The error message should be understandable and helpful.
   - Example:

     ```
     pgsql
     CopyEdit
     Error: '}' expected before 'else' at line 10
     ```

3. **Non-redundancy**:
   - Should not report the same error multiple times.
   - Avoid cascading errors (caused by earlier ones).
4. **Recovery Assistance**:
   - Provide suggestions when possible.
   - Example:

     ```
     arduino
     CopyEdit
     Error: Undeclared variable 'x' at line 8. Did you mean 'y'?
     ```

5. **Multiple Errors**:
   - Should detect and report multiple errors in one compilation pass, not just the first one.
6. **Categorization**:

- Clearly distinguish between **lexical**, **syntax**, **semantic**, and **runtime** errors.

✅ 2. Error Recovery Strategies in Compiler

When a compiler encounters an error, it needs to recover and continue parsing the remaining code.

◆ Common Strategies:

1. **Panic Mode Recovery**:
    - Skip input symbols until a synchronizing token (like `;`, `{`, `}`) is found.
    - Simple and guarantees termination.
    - Example:

      ```
      c
      CopyEdit
      int a = ; int b = 10;
      ```

      Skips until `;`, continues parsing from `int b`.

2. **Phrase-Level Recovery**:
    - Replaces or inserts tokens to correct the error and continue.
    - Example:

      ```
      c
      CopyEdit
      if (a > b) printf("Hi")
      ```

      Recovery: insert semicolon after `printf("Hi")`

3. **Error Productions**:
    - Grammar is extended to include common mistakes.
    - Useful in specific language constructs.
    - Example:

      ```
      pgsql
      CopyEdit
      if-statement → if (condition) statement
      error → if (condition statement
      ```

4. **Global Correction**:
    - Tries to make minimal changes to the program for it to be correct.
    - Expensive; not commonly used in practice.

✅ 3. What is LEX and YACC? (with Examples)

◆ LEX (Lexical Analyzer Generator)

- Tool used to generate **lexical analyzers (scanners)**.

- It identifies **tokens** from source code (keywords, identifiers, numbers).
- Converts source code into a stream of tokens.

Example (LEX code):

```
lex
CopyEdit
%%
[0-9]+      { printf("NUMBER "); }
[a-zA-Z]+   { printf("IDENTIFIER "); }
%%

int main() {
  yylex();  // start scanning
  return 0;
}
```

Input:

```
ini
CopyEdit
x = 123;
```

Output:

```
nginx
CopyEdit
IDENTIFIER NUMBER
```

◆ YACC (Yet Another Compiler Compiler)

- Tool to generate **parsers**.
- Works with **grammar rules** (usually context-free grammars).
- Accepts tokens from LEX and builds a **syntax tree** or performs **semantic actions**.

Example (YACC code):

```
yacc
CopyEdit
%token NUMBER

%%
expr: expr '+' expr { printf("Add\n"); }
    | NUMBER         { printf("Number\n"); }
    ;
%%

int main() {
  yyparse();  // start parsing
  return 0;
}
```

✅ 4. Difference between LEX and YACC

Feature	LEX	YACC
Role	Lexical analysis	Syntax analysis (parsing)
Input	Regular expressions	Context-free grammar rules
Output	Tokens	Parse tree or actions
Dependency	Works independently or with YACC	Depends on LEX for tokens
Use	Tokenizing source code	Building and analyzing grammar
Example Input	`if`, `123`, `x`	`if (x > 0) x = x + 1;`

✅ Summary

Topic	Key Points
Error Reporter	Accurate, clear, multiple errors, helpful messages
Error Recovery	Panic mode, phrase-level, error productions, global correction
LEX	Token generator using regex
YACC	Parser generator using grammar
Difference (LEX vs YACC)	LEX = lexical, YACC = syntax/parser

✅ 1. SDD (Syntax Directed Definition) and SDT (Syntax Directed Translation)

Both are methods of **attaching semantics** to grammar rules in a compiler.

◆ SDD – Syntax Directed Definition

- An SDD is a **context-free grammar** with **attributes** and **semantic rules**.
- Attributes are associated with grammar symbols:
  - **Synthesized attributes**: passed up the parse tree.
  - **Inherited attributes**: passed down or across.

Example (SDD for arithmetic expression):

```
plaintext
CopyEdit
E → E1 + T    { E.val = E1.val + T.val }
E → T         { E.val = T.val }
T → num       { T.val = num.lexval }
```

Input: `3 + 4`

Evaluation:

```
r
CopyEdit
T → num            (num.lexval = 3)
E1 → T             (E1.val = 3)
T → num            (num.lexval = 4)
E → E1 + T         (E.val = 3 + 4 = 7)
```

◆ SDT – Syntax Directed Translation

- An SDT is like an SDD but allows **actions (code snippets)** to be embedded **within productions**.
- These actions can be inlined, enabling actual translation during parsing.

Example (SDT for printing result):

```
plaintext
CopyEdit
E → E1 '+' T { print('+') }
E → T
T → num { print(num.lexval) }
```

Input: `3 + 4`
Output: `3 4 +` (postfix notation)

✅ Difference between SDD and SDT

Feature	SDD (Syntax Directed Definition)	SDT (Syntax Directed Translation)
Focus	Attribute evaluation	Embedding actions during parsing
Use	Attribute grammar specification	Code generation or translation
Actions	Separate from grammar	Inlined in grammar productions
Evaluation Order	Based on attribute dependencies	Based on parsing strategy

---

✅ 2. Why Every S-attributed Definition is L-attributed

◆ Definitions:

- **S-attributed**: Only synthesized attributes are used.

- **L-attributed**: Allows both synthesized and inherited attributes, but inherited attributes must be evaluated using only left-to-right (top-down) information.

◆ Explanation:

- Since **S-attributed** uses only **synthesized** attributes, which are computed from **child nodes** (bottom-up), they naturally fit into **L-attributed** evaluation (which is more general).
- Hence, every **S-attributed** grammar is also **L-attributed**, because it satisfies all L-attributed constraints (and doesn't even use inherited attributes).

◆ Example:

S-attributed (also L-attributed):

```
plaintext
CopyEdit
E → E1 + T   { E.val = E1.val + T.val }
E → T        { E.val = T.val }
T → num      { T.val = num.lexval }
```

No inherited attributes → S-attributed → Also satisfies L-attributed.

L-attributed (not necessarily S-attributed):

```
plaintext
CopyEdit
A → B C
   B.in = A.in
   C.in = B.out
   A.out = C.out
```

Here, `in` and `out` are inherited and synthesized attributes. This is L-attributed, but **not** S-attributed.

✓ 3. Code Optimization Techniques

◆ (A) Eliminating Induction Variables

- **Induction variables** are variables that change linearly with each loop iteration.
- We can eliminate **redundant induction variables** to improve performance.

Example:

```
c
CopyEdit
for (i = 0; i < n; i++) {
    a = 4 * i;
    b = a + 3;
}
```

Optimization:

```c
CopyEdit
for (i = 0, a = 0; i < n; i++, a += 4) {
    b = a + 3;
}
```

- `4 * i` is replaced by an incremental update of `a`, avoiding multiplication.

◆ (B) Code Motion (Loop-Invariant Code Motion)

- Moves **invariant code** outside loops to avoid redundant computation.

Example (Before Optimization):

```c
CopyEdit
for (i = 0; i < n; i++) {
    y = x * 10;
    a[i] = y + i;
}
```

Here, `x * 10` is invariant (doesn't depend on loop), so move it out.

After Code Motion:

```c
CopyEdit
y = x * 10;
for (i = 0; i < n; i++) {
    a[i] = y + i;
}
```

✅ Summary Table

Concept	Explanation / Key Point
SDD	Attribute grammar using semantic rules
SDT	Grammar with embedded actions for translation
S-attributed	Only synthesized attributes
L-attributed	Synthesized + inherited attributes (left to right)
Induction Var Elimination	Replace redundant calculations with incremental vars
Code Motion	Move loop-invariant expressions outside loops