Parameter Passing and AR

1. Discuss parameter passing techniques for procedure.
2. What is activation record? Explain stack allocation of activation records using example.

Based on the content of your uploaded file, here's a detailed answer to your questions:

---

1. Parameter Passing Techniques for Procedures

When a procedure (or function) is called, parameters are passed from the caller to the callee. There are several methods to handle this:

Types of Parameters:

- **Actual Parameters:** The values or variables provided by the caller.
- **Formal Parameters:** The variables defined by the function/procedure to receive the values.

Common Parameter Passing Methods:

1. **Call by Value:**
   - Copies the actual value into the formal parameter.
   - Changes made inside the procedure do **not** affect the actual parameter.
   - Safe but may require more memory.
2. **Call by Reference (Address/Location):**
   - Copies the address of the actual parameter.
   - Changes in the procedure **do affect** the original value.
   - Efficient but riskier due to side effects.
3. **Copy-Restore (Copy-in Copy-out or Value-Result):**
   - A hybrid method.
   - Value of actual parameter is copied into the formal one at the start.
   - When the procedure ends, the final value of the formal parameter is copied back to the actual one.
   - Useful when working with arrays or large structures.
4. **Call by Name:**
   - Works like macro expansion.
   - The actual expression is substituted directly into the function body.

- Less commonly used today.
- Allows late evaluation of parameters.

---

2. What is an Activation Record? Explain Stack Allocation with Example

Activation Record:

An **activation record** (or stack frame) is a block of memory used to manage information about a single execution of a procedure. It's stored on the call stack.

Fields of an Activation Record:

- **Temporary values:** For intermediate computations.
- **Local variables:** Declared within the procedure.
- **Machine status:** Save CPU state (e.g., registers).
- **Access link (optional):** Refers to non-local variables.
- **Control link (optional):** Points to the caller's activation record.
- **Actual parameters:** Values passed to the procedure.
- **Return value:** Result to return to the caller.

Stack Allocation of Activation Records (with Example):

Procedures are managed using a **stack-based memory allocation**:

Example:

Let's say we have:

```c
`void A() {
   B();  // Procedure B is called inside A
}

void B() {
   C();  // Procedure C is called inside B
}

void C() {
   // Execution
}
`
```

When the program runs:

1. **Call to A:** Stack frame (activation record) for A is created and pushed onto the stack.
2. **Call to B from A:** B's activation record is pushed on top of A's.
3. **Call to C from B:** C's activation record is pushed on top of B's.

Once `C()` finishes, its activation record is popped. Then `B()` continues, and when done, its record is removed. Finally, `A()` completes, and its frame is removed.

Diagrammatically:

```mathematica
`Top of Stack
---------------
| C's Record  |
|-------------|
| B's Record  |
|-------------|
| A's Record  |
---------------
Bottom of Stack
`
```

Calling Sequence:

- Caller evaluates parameters and saves return address.
- Callee saves machine state and initializes local variables.
- Upon return, callee stores return value, restores state, and control returns to caller.

Let me know if you'd like a visual diagram or simplified version for notes or presentations!

3. Describe Activation record and Activation tree.

Sure! Here's a clear explanation for both **Activation Record** and **Activation Tree**, based on your syllabus:

---

3. Activation Record and Activation Tree

✅ Activation Record:

An **activation record** is a data structure that stores information about a single execution of a procedure or function. It is used during **runtime** and placed on the **call stack**.

Fields of Activation Record:

1. **Temporary values** – Intermediate values during expression evaluation.
2. **Local variables** – Declared inside the procedure.
3. **Machine status** – Return address and saved registers.
4. **Access link (optional)** – Link to non-local variables (lexical nesting).
5. **Control link** – Points to the caller's activation record.

6. **Actual parameters** – Values/addresses passed to the procedure.
7. **Return value** – Space for returning result to caller.

📌 **Example Layout:**

```pgsql
`+-----------------+ ← Top of stack
| Return value     |
| Actual parameters|
| Control link     |
| Access link      |
| Machine status   |
| Local variables  |
| Temporaries      |
+-----------------+ ← Bottom of this record
`
```

✅ Activation Tree:

An **activation tree** is a graphical representation of how procedure calls are nested during program execution.

- Each **node** in the tree represents a **procedure activation**.
- The **edges** represent **calls** from one procedure to another.
- The **root** is the first function called (usually `main`).
- When a procedure calls another, the called one becomes a **child** node.

📌 **Example:**

Consider this code:

```c
`void main() {
   A();
}

void A() {
   B();
   C();
}

void B() { }

void C() {
   D();
}
`
```

🔽 **Activation Tree:**

```less
`        main
          |
```

```
       A
      / \
     B   C
         |
         D
`
```

📌 **Features:**

- Shows **calling relationships**, not call order.
- Helps visualize the **lifetime and scope** of each activation.
- Useful in compiler design and debugging.

Would you like a simple diagram image of this activation tree? I can generate one for you.

4. Explain static storage allocation technique.
5. Explain activation record organization.

Sure! Here's a clear and structured explanation of both topics from your syllabus:

---

4. Static Storage Allocation Technique

✅ **Definition:**

Static storage allocation is a memory allocation strategy where the memory for all variables and data objects is **allocated at compile time**, **before the program executes**.

✅ **Key Features:**

- Memory bindings are **fixed** and do not change during runtime.
- There is **no need for a runtime support system** to manage memory.
- All data objects (global variables, static variables, constants) have **predefined memory addresses**.

✅ **Advantages:**

- Very **fast access** to memory since locations are known.
- No overhead for memory allocation/deallocation at runtime.
- Simple to implement.

✅ **Disadvantages:**

- **Not flexible** – memory for unused variables is still allocated.
- Can't handle **recursive functions** or **dynamic data structures** like linked lists or trees.

✅ Use Cases:

- Used for:
  - Global variables
  - Constants
  - Static local variables
  - Fixed-size arrays

✅ Example:

```c
`int x = 5;
int arr[10];
`
```

- Here, `x` and `arr` will get memory allocated during compile time.

---

5. Activation Record Organization

✅ Definition:

An **activation record** is a block of memory created on the **call stack** whenever a function/procedure is invoked. It stores all information needed to manage the **execution** of that procedure.

✅ Fields of an Activation Record:

1. **Temporary values:** Intermediate results.
2. **Local variables:** Variables declared inside the procedure.
3. **Machine status:** Return address, old program counter, saved registers.
4. **Access link (optional):** Pointer to activation record of the lexically enclosing scope (non-local access).
5. **Control link:** Points to the **caller's** activation record.
6. **Actual parameters:** Values passed to the procedure.
7. **Return value:** Space for storing return data.

✅ Memory Layout (Top to Bottom):

```pgsql
`+----------------------+
| Return value         |
| Actual parameters    |
| Control link         |
```

```
| Access link          |
| Machine status       |
| Local variables      |
| Temporaries          |
+---------------------+
`
```

✅ Activation Record Lifecycle:

1. Procedure is called → Activation record is **pushed** to the stack.
2. Procedure returns → Activation record is **popped** from the stack.

✅ Example:

```c
`void A() {
    int x;
    B();
}

void B() {
    int y;
}
`
```

- Calling `` `A()` `` creates A's activation record.
- Inside `` `A()` ``, calling `` `B()` `` creates B's activation record on top of A's.

Would you like a visual diagram of this activation record layout? I can generate one if you'd like!

6. Explain any two parameter passing methods.
7. Discuss any two dynamic storage allocation techniques

Sure! Here's a concise and clear explanation for both questions:

---

6. Explain Any Two Parameter Passing Methods

✅ (i) Call by Value:

- The **actual value** of the argument is copied into the **formal parameter**.
- Changes made to the parameter **do not affect** the original argument.
- **Safe**, but can be **less efficient** for large data structures.

Example:

```c
`void change(int x) {
   x = x + 10;
}

int main() {
   int a = 5;
   change(a);
   // a is still 5
}
`
```

✅ (ii) Call by Reference:

- The **address** of the actual parameter is passed.
- Formal parameter becomes an **alias** of the actual parameter.
- Changes **directly affect** the original variable.
- **Efficient**, but **less safe** due to possible side effects.

Example:

```c++
`void change(int &x) {
   x = x + 10;
}

int main() {
   int a = 5;
   change(a);
   // a is now 15
}
`
```

---

7. Discuss Any Two Dynamic Storage Allocation Techniques

✅ (i) Explicit Allocation (Fixed or Variable Size Blocks):

- Programmer or system explicitly **allocates and frees memory**.
- Common functions: `` `malloc()` ``, `` `free()` `` in C, or `` `new` ``, `` `delete` `` in C++.
- Can lead to **fragmentation** if not managed carefully.

Example:

```c
`int *p = (int *)malloc(sizeof(int) * 10);
free(p);
`
```

✅ (ii) Implicit Allocation (Garbage Collection):

- Memory is automatically managed by the **runtime system**.
- Uses techniques like **Reference Counting** or **Mark and Sweep**.
- Programmer does **not manually deallocate** memory.
- Found in languages like **Java, Python**.

Reference Counting:

- Keeps a counter for how many references point to a memory block.
- When count = 0, block is **automatically deallocated**.

Let me know if you'd like visual diagrams for these concepts or a simplified version for quick revision!

8. Discuss how Activation Record are used to access local and global variables?
9. Write a short note on Stack allocation.

Sure! Here's a well-structured explanation for both questions:

---

8. How Activation Records Are Used to Access Local and Global Variables

✅ Accessing Local Variables:

- **Local variables** are declared within a function.
- When a function is called, an **activation record** is created and **pushed** onto the **call stack**.
- Local variables are stored **within** the activation record of that function.
- The function accesses its local variables using **offsets** from a base pointer (like `frame pointer` or `FP`).

Example:

```c
void func() {
    int a = 10;  // 'a' is stored in func's activation record
}
```

- `a` is accessed directly via the base pointer pointing to `func()`'s activation record.

✅ Accessing Global Variables:

- **Global variables** are not stored in activation records.
- They are stored in a **fixed memory area** (usually in the **data segment**).
- All functions can access them **directly using their name or symbol table entry**.

- Their address remains constant throughout program execution.

Example:

```c
`int g = 20; // Global variable

void func() {
    g = g + 1;  // Direct access, not via activation record
}
`
```

🔁 Summary:

Variable Type	Location	Access Method
Local Variable	Activation Record	Offset from Frame Pointer (FP)
Global Variable	Data Segment (Static)	Direct address via symbol table

---

9. Short Note on Stack Allocation

✅ Definition:

Stack allocation is a **run-time storage allocation** strategy where memory is managed using a **stack** data structure (LIFO – Last In First Out).

✅ Key Features:

- Memory is allocated when a **procedure is called** and **deallocated when it returns**.
- Activation records are **pushed** onto the stack during a call and **popped** on return.
- Efficient and simple to implement.

✅ Use in Function Calls:

- Each function call creates an **activation record**.
- This record contains local variables, parameters, return addresses, etc.
- As functions call each other, the stack grows.
- As they return, the stack shrinks.

✅ Advantages:

- Efficient memory management for function calls.
- Automatically handles **nested and recursive** calls.

✅ Disadvantages:

- Cannot be used for **dynamic memory** or **data that must persist** beyond a function's lifetime.
- Stack size is **limited**.

Would you like all 9 questions combined into a single document or notes file? I can prepare that for you!

10. How is task divided between calling & called program for stack updating?

Great question! Here's a clear and structured answer for:

---

10. How is Task Divided Between Calling and Called Program for Stack Updating?

In **stack-based allocation**, when a function (called program or callee) is invoked by another function (calling program or caller), both participate in **creating and updating the stack**. This is done through what’s called the **calling sequence** and the **return sequence**.

✅ Tasks of the Calling Program (Caller):

1. **Evaluate Actual Parameters:**
   - Calculates the values or addresses to pass to the called function.
2. **Create Space for Activation Record:**
   - Reserves space on the stack for the callee's activation record.
3. **Pass Parameters:**
   - Stores actual parameters in the activation record (by value or reference).
4. **Save Return Address:**
   - Stores the address of the instruction to return to after the callee finishes.
5. **Control Transfer:**
   - Transfers control to the called program (typically using a jump or call instruction).

✅ Tasks of the Called Program (Callee):

1. **Save Machine State:**
   - Saves important registers (like base pointer, return address) into the activation record.
2. **Set Up Environment:**
   - Creates space for local variables and temporaries.
3. **Initialize Local Variables:**
   - Initializes variables declared in the callee.
4. **Begin Execution:**
   - Starts executing the function’s code.

✅ Return Sequence (When Callee Finishes):

- **Callee:**
  - Places return value in a known location.
  - Restores machine status (registers, stack pointer).
  - Returns control to the caller using the saved return address.
- **Caller:**
  - Retrieves the return value.
  - Continues execution from where it left off.

📌 Diagram Summary:

```pgsql
`Caller Stack Frame
-------------------------
| Return Address        | ← Set by Caller
| Parameters            | ← Set by Caller
-------------------------
Callee Stack Frame
-------------------------
| Control Link          | ← Set by Caller
| Local Variables       | ← Set by Callee
| Temporaries           | ← Set by Callee
-------------------------
`
```

🔁 Summary Table:

Task	Caller	Callee
Evaluate parameters	✅	
Save return address	✅	
Allocate activation record	✅	✅ (fills in local data)
Save machine status		✅
Initialize local variables		✅
Restore state & return		✅
Resume execution	✅	

Let me know if you'd like a visual diagram or short notes PDF with this included!

1. Discuss any three methods for code optimization.

Sure! Here's a well-organized answer to:

---

1. Discuss Any Three Methods for Code Optimization

Code optimization improves the **performance and efficiency** of a program by modifying the code without changing its output or behavior. Here are three commonly used optimization techniques:

✅ 1. Common Sub-expression Elimination

📌 **Definition:**

If an **expression is computed more than once** and the values of variables involved don't change, it can be computed once and **reused**.

🔄 **Example:**

```c
`t1 := 4 * i
t2 := a[t1]
t3 := 4 * i   // repeated expression
`
```

✔️ After optimization:

```c
`t1 := 4 * i
t2 := a[t1]
// reuse t1 instead of recalculating
`
```

✅ **Benefit:**

- Reduces **redundant computations**
- Saves **time and memory**

✅ 2. Dead Code Elimination

📌 **Definition:**

Removes code that **does not affect** the program's output. A variable is said to be "dead" if it is **assigned a value but never used**.

🔄 **Example:**

```c
`a = 5;
if (a == 5) {
   c++;
}
else {
   k++;
   printf("Dead code");
```

```
}
`
```

✔️ The **else block** can be removed as it will **never be executed**.

✅ **Benefit:**

- Reduces **code size**
- Improves **execution speed**

✅ 3. Strength Reduction

📌 **Definition:**

Replaces **costly operations** (like multiplication/division) with **cheaper ones** (like addition/subtraction).

🔄 **Example:**

```c
`for(i = 1; i <= 50; i++) {
    count = i * 50;
}
`
```

✔️ After optimization:

```c
`temp = 0;
for(i = 1; i <= 50; i++) {
    temp = temp + 50;
    count = temp;
}
`
```

✅ **Benefit:**

- Improves performance, especially in **loops**
- Useful on systems where **multiplication is expensive**

✅ Summary Table:

Technique	Description	Benefit
Common Sub-expression Elimination	Avoid recomputing same expressions	Saves CPU time and memory
Dead Code Elimination	Removes code that doesn't affect program output	Reduces code size
Strength Reduction	Replace costly operations with simpler ones	Speeds up computation

---

Let me know if you'd like this formatted as class notes or turned into a diagram or PDF!

Code Optimization Techniques

Slight more descriptive but not too

Here's a slightly more descriptive version of your document with a bit more explanation while still keeping it concise and exam-friendly:

---

1. Discuss Any Three Methods for Code Optimization

Code optimization is the process of improving the **intermediate code** generated by a compiler to make it **faster**, **smaller**, or **more efficient**, without changing its behavior. Optimizations can reduce runtime, memory usage, or even power consumption.

Below are three important code optimization techniques:

✅ 1. Common Sub-expression Elimination

📌 Definition:

A **common sub-expression** is an expression that occurs **multiple times** in a program and computes the same value every time, provided the variables involved are unchanged.

Instead of recomputing it each time, we compute it **once** and reuse the result.

🔄 Example:

```c
`t1 := 4 * i
t2 := a[t1]
t3 := 4 * i   // repeated expression
`
```

✔️ Optimized version:

```c
`t1 := 4 * i
t2 := a[t1]
// Use t1 instead of recalculating
`
```

✅ Benefit:

- Eliminates **redundant computation**
- Saves **processor time** and **improves performance**

✅ 2. Dead Code Elimination

📌 **Definition:**

Dead code is code that **never affects** the program output. This includes variables assigned but never used, or unreachable blocks of code. Removing such code **reduces clutter** and makes the program leaner.

🔄 **Example:**

```c
`a = 5;
if (a == 5) {
   c++;
}
else {
   k++;
   printf("Dead code");
}
`
```

✔️ Optimized version:

```c
`a = 5;
c++;  // Removed the else block entirely
`
```

✅ **Benefit:**

- **Reduces code size**
- Improves **readability** and **execution speed**

✅ 3. Strength Reduction

📌 **Definition:**

Replaces **computationally expensive** operations (like multiplication or division) with **simpler ones** (like addition or subtraction), especially inside loops where such operations are repeated often.

🔄 **Example:**

```c
`for(i = 1; i <= 50; i++) {
   count = i * 50;
}
`
```

✔️ Optimized version:

```c
`temp = 0;
for(i = 1; i <= 50; i++) {
   temp = temp + 50;
   count = temp;
}
`
```

✅ Benefit:

- Reduces **CPU cycles**
- Ideal for **performance-critical code**

✅ Summary Table:

Technique	Description	Benefit
Common Sub-expression Elimination	Detects repeated expressions and computes once	Avoids redundant calculations
Dead Code Elimination	Removes code that never impacts program output	Smaller, faster code
Strength Reduction	Replaces heavy operations with cheaper equivalents	Optimized for speed and loops

---

Let me know if you'd like me to update this directly into your canvas!

2. Discuss any three code optimization techniques
3. Describe algorithm for global common subexpression elimination.

Sure! Here's an updated and slightly more descriptive version with answers for both questions:

---

2. Discuss Any Three Code Optimization Techniques

Code optimization improves the **performance**, **size**, or **efficiency** of a program without changing its behavior. Here are three widely used techniques:

✅ 1. Loop Invariant Code Motion (Code Movement)

📌 Definition:

Moves **invariant expressions** (those that don't change in each loop iteration) **outside the loop** to reduce repeated computation.

🔄 Example:

```c
for (i = 0; i < 100; i++) {
    x = a * b;
    y = x + i;
}
```

✔️ Optimized version:

```c
x = a * b;
for (i = 0; i < 100; i++) {
    y = x + i;
}
```

✅ **Benefit:**

- Reduces **frequency** of execution
- Saves **CPU time**

✅ 2. Copy Propagation

📌 **Definition:**

If a variable is **assigned to another**, this optimization replaces uses of that variable with the **original** one, reducing unnecessary references.

🔄 **Example:**

```c
y = x;
z = y + 2;
```

✔️ Optimized version:

```c
z = x + 2;
```

✅ **Benefit:**

- Simplifies expressions
- Enables further optimizations

✅ 3. Constant Folding

📌 **Definition:**

Evaluates **constant expressions** at compile time instead of runtime.

🔄 **Example:**

```c
area = 3.14 * 5 * 5;
```

✔️ Optimized version:

```c
area = 78.5;
```

✅ **Benefit:**

- Reduces runtime calculations
- Speeds up execution

---

3. Algorithm for Global Common Subexpression Elimination

This technique removes expressions that **repeat across basic blocks** and whose **operands do not change**.

✅ **Steps of the Algorithm:**

1. **Construct Flow Graph**:
    - Divide code into basic blocks and link them in a control flow graph.
2. **Compute Available Expressions**:
    - Use data-flow analysis to determine which expressions are **available** at the start of each block.
3. **Mark Redundant Expressions**:
    - Identify expressions that already exist and are not modified before reuse.
4. **Replace with Temporary Variables**:
    - Replace repeated expressions with references to a **temporary variable** that holds the value.
5. **Insert Computation at First Occurrence**:
    - Ensure the common expression is computed once and its result reused.

✅ **Example:**

Original Code:

```c
`a = b + c;
d = a * e;
x = b + c;
y = x * z;
`
```

Optimized:

```c
`t1 = b + c;
a = t1;
d = a * e;
x = t1;
y = x * z;
`
```

✔️ The subexpression `` `b + c` `` is computed once and reused.

Let me know if you want this added to your canvas document or formatted for print/PDF!
Thanks! Here's the updated content for your document with both questions answered slightly more descriptively:

---

2. Discuss Any Three Code Optimization Techniques

Code optimization helps to improve the **performance**, **speed**, and **efficiency** of a program without altering its intended output. Below are three popular techniques:

✅ 1. Loop Invariant Code Motion (Code Movement)

📌 Description:

Moves operations that **do not change within a loop** (loop-invariant statements) **outside** the loop, so they are not executed repeatedly.

🔄 Example:

```c
`for (i = 0; i < 100; i++) {
   x = a * b;
   y = x + i;
}
`
```

✔️ Optimized:

```c
`x = a * b;
for (i = 0; i < 100; i++) {
   y = x + i;
```

```
}
```

✅ **Benefit**: Reduces repetitive computation and enhances speed in loops.

✅ 2. Copy Propagation

📌 Description:

This replaces the use of a variable that was copied from another with the original variable, reducing unnecessary memory usage and simplifying expressions.

🔄 Example:

```c
y = x;
z = y + 5;
```

✔️ Optimized:

```c
z = x + 5;
```

✅ **Benefit**: Simplifies code and makes other optimizations more effective.

✅ 3. Constant Folding

📌 Description:

Evaluates **constant expressions** at compile-time rather than runtime to save execution effort.

🔄 Example:

```c
area = 3.14 * 5 * 5;
```

✔️ Optimized:

```c
area = 78.5;
```

✅ **Benefit**: Minimizes runtime computation and increases performance.

OpenAI

3. Algorithm for Global Common Subexpression Elimination

This technique removes **repeated expressions** that yield the same value, even across multiple basic blocks, and replaces them with a **single computed value**.

✅ **Steps in the Algorithm:**

1. **Build a Control Flow Graph (CFG)**:
   - Divide code into basic blocks and form connections.
2. **Perform Data Flow Analysis**:
   - Calculate **available expressions** at the start of each block.
3. **Detect Common Subexpressions**:
   - Look for repeated expressions whose operands have not changed.
4. **Introduce Temporaries**:
   - Store the expression's result in a **temporary variable** the first time it is computed.
5. **Replace All Subsequent Occurrences**:
   - Use the temporary instead of recomputing the expression.

✅ **Example:**

Original:

```c
`a = b + c;
d = a * e;
x = b + c;
y = x * z;
`
```

Optimized:

```c
`t1 = b + c;
a = t1;
d = a * e;
x = t1;
y = x * z;
`
```

✔️ The expression `` `b + c` `` is computed once and reused.

Let me know if you'd like this added to your canvas document or turned into a printable format!

4. Explain three loop optimization techniques with example.

5. Explain peephole optimization

Here's a clear and slightly descriptive explanation for both of your questions:

---

4. Explain Three Loop Optimization Techniques with Example

Loop optimization improves the performance of loops, which are frequently executed parts of programs. Here are three key loop optimization techniques:

✅ 1. Loop Invariant Code Motion

📌 **Description:**

Moves calculations **outside the loop** if their value **does not change** during iterations.

🔄 **Example:**

```c
for (i = 0; i < 100; i++) {
   x = a * b;   // Doesn't depend on i
   y = x + i;
}
```

✔️ Optimized:

```c
x = a * b;
for (i = 0; i < 100; i++) {
   y = x + i;
}
```

✅ **Benefit**: Reduces repeated calculations, improving efficiency.

✅ 2. Strength Reduction

📌 **Description:**

Replaces **expensive operations** (like multiplication) with **cheaper ones** (like addition) in loops.

🔄 **Example:**

```c
for (i = 1; i <= 50; i++) {
   count = i * 50;
}
```

✔️ Optimized:

```c
temp = 0;
for (i = 1; i <= 50; i++) {
   temp = temp + 50;
   count = temp;
}
```

✅ **Benefit**: Reduces computational cost.

✅ 3. Loop Unrolling

📌 Description:

Executes **multiple loop iterations in a single loop body** to reduce the overhead of loop control.

🔄 Example:

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

✔️ Optimized:

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

✅ **Benefit**: Reduces loop control instructions, speeding up execution.

---

5. Explain Peephole Optimization

✅ 📌 Definition:

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

It focuses on **short sequences** of instructions to perform small improvements.

✅ Types of Peephole Optimizations:

1. **Redundant Load/Store Elimination:**

```assembly
`MOV R1, A
MOV A, R1   // ← redundant
`
```

✔️ Optimized: Remove the second instruction.

---

2. **Jump-to-Jump Elimination:**

```assembly
`GOTO L1
...
L1: GOTO L2
`
```

✔️ Optimized:

```assembly
`GOTO L2
`
```

---

3. **Algebraic Simplification:**

```c
`x = x + 0;   // or x = x * 1;
`
```

✔️ Optimized: These can be **eliminated**.

---

4. **Strength Reduction (within peephole):**

```assembly
`x = x * 2;
`
```

✔️ Optimized:

```assembly
`x = x + x;
`
```

---

5. **Use of Machine Idioms:**

- Replace general code with a **special CPU instruction** (e.g., auto-increment).

✅ Benefit:

- Improves **execution speed**
- Reduces **instruction count**
- Easy to implement in **code generators**

Let me know if you'd like these added to your document or turned into a summarized revision sheet!

slight descriptive but not too much

Update the canvas document with the following content:

---

4. Explain Three Loop Optimization Techniques with Example

Loop optimization techniques improve the performance of loops, which are often executed many times. Here are three commonly used loop optimizations:

✅ 1. Loop Invariant Code Motion

Moves statements that produce the same result on every iteration **outside the loop**.

Example:

```c
`for (i = 0; i < 100; i++) {
    x = a * b;
    y = x + i;
}
`
```

Optimized:

```c
`x = a * b;
for (i = 0; i < 100; i++) {
    y = x + i;
}
`
```

✔️ Reduces repeated computation inside the loop.

✅ 2. Strength Reduction

Replaces **expensive operations** like multiplication with **cheaper ones** like addition.

Example:

```c
`for (i = 1; i <= 50; i++) {
    count = i * 50;
}
`
```

Optimized:

```c
`temp = 0;
for (i = 1; i <= 50; i++) {
    temp = temp + 50;
    count = temp;
}
`
```

✔️ Saves computation time in loops.

✅ 3. Loop Unrolling

Executes multiple iterations in one loop to reduce the overhead of loop control.

Example:

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

Optimized:

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

✔️ Reduces loop control instructions.

---

5. Explain Peephole Optimization

Peephole optimization is a **local optimization** technique where a small group of instructions is analyzed and replaced with a **shorter or faster sequence**.

✅ Common Types:

1. **Redundant Code Elimination**

```assembly
`MOV R1, A
MOV A, R1   // Redundant
`
```

✔️ Remove second instruction.

2. **Jump Optimization**

```assembly
`GOTO L1
...
L1: GOTO L2
`
```

✔️ Simplified to `GOTO L2`.

3. **Algebraic Simplification**

- Remove `x = x + 0` or `x = x * 1`.

4. **Strength Reduction**

- Replace `x = x * 2` with `x = x + x`.

✔️ Peephole optimization is simple but very effective in improving performance.
The canvas tool is currently unavailable, so I can't update the document directly. However, here is the slightly descriptive content you requested. You can copy and paste it into your notes or document:

---

4. Explain Three Loop Optimization Techniques with Example

Loop optimization improves execution speed by minimizing unnecessary computations in loops. Below are three key techniques:

✅ 1. Loop Invariant Code Motion

Moves calculations that don't change inside the loop to outside, so they are done only once.

Example:

```c
`for (i = 0; i < 100; i++) {
    x = a * b;
    y = x + i;
}
`
```

Optimized:

```c
x = a * b;
for (i = 0; i < 100; i++) {
    y = x + i;
}
```

✅ 2. Strength Reduction

Replaces expensive operations like multiplication with faster ones like addition.

Example:

```c
for (i = 1; i <= 50; i++) {
    count = i * 50;
}
```

Optimized:

```c
temp = 0;
for (i = 1; i <= 50; i++) {
    temp = temp + 50;
    count = temp;
}
```

✅ 3. Loop Unrolling

Executes multiple loop iterations in a single loop body to reduce loop overhead.

Example:

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

Optimized:

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

---

5. Explain Peephole Optimization

Peephole optimization examines a **small set of consecutive instructions** and replaces them with **simpler, faster, or shorter code**.

✅ Common Types:

1. **Redundant Load/Store Elimination**

```assembly
`MOV R1, A
MOV A, R1    // redundant
`
```

2. **Jump Optimization**

```assembly
`GOTO L1
...
L1: GOTO L2
`
```

→ Simplified to: `GOTO L2`

3. **Algebraic Simplification**

- Eliminates useless operations like `x = x + 0` or `x = x * 1`.

4. **Strength Reduction**

- Replace `x = x * 2` with `x = x + x`.

✔️ Peephole optimization is easy to apply and can significantly improve execution speed in local areas of code.

Let me know if you want all your answers compiled into a single file or notes PDF!

8. Define: DAG. Explain DAG representation of basic block with example

Here's a clear and concise answer for:

---

8. Define: DAG. Explain DAG Representation of Basic Block with Example

---

✅ Definition: DAG (Directed Acyclic Graph)

A **DAG (Directed Acyclic Graph)** is a data structure used in **compiler optimization** to represent expressions in a **basic block**.

- **Directed** means the edges have direction (from parent to child).
- **Acyclic** means there are **no cycles** (you can't return to the same node by following edges).
- In compiler design, DAG helps in identifying:
  - **Common sub-expressions**
  - **Redundant computations**
  - **Optimization opportunities**

✅ Purpose of DAG in Compiler Design:

- Detect and **eliminate repeated expressions**
- **Simplify** code by reusing already computed results
- **Improve** code generation efficiency

✅ DAG Representation of a Basic Block:

A **basic block** is a sequence of instructions without any jumps or labels inside. Let's take an example:

🔄 Example:

```c
`t1 = a + b
t2 = a + b
t3 = t1 - c
t4 = t2 - c
`
```

✅ Step-by-Step DAG Construction:

1. Create leaf nodes for `a`, `b`, and `c`.
2. Create a `+` node with children `a` and `b`, and label it for `t1` and `t2` (since it's common).
3. Create a `-` node with children `t1` and `c`, and label it for `t3` and `t4`.

✅ DAG Diagram (Text Form):

```css
`        (-)
       /     \
    (+)       c
   /   \
a       b

Labels:
(+) → t1, t2
(-) → t3, t4
`
```

✅ Benefits:

- **Avoids duplicate calculations** (e.g., `a + b` done once)
- Helps generate **optimized code**
- Simplifies **intermediate representation**

Let me know if you'd like a visual diagram or this formatted for a notes page or PDF!

Describe code generator design issues.

Here's a well-organized and slightly descriptive answer for:

---

9. Describe Code Generator Design Issues

A **code generator** is the part of a compiler that translates intermediate code into target machine code. Designing an efficient and correct code generator involves addressing several important issues:

✅ 1. Input to Code Generator

- The input is usually in the form of **intermediate representation** (IR) such as:
  - Three-address code
  - Syntax trees
  - DAGs
- The IR should be **error-free** and **semantically correct** before code generation begins.

✅ 2. Target Program Form

- The target code can be:
  - **Absolute code** (ready to run directly)
  - **Relocatable code** (can be linked and relocated later)
  - **Assembly code** (requires assembler to generate machine code)

✅ 3. Instruction Selection

- Choosing the most **appropriate machine instructions** for a given operation.
- Some instructions may perform combined operations (e.g., load and add), which are **more efficient**.

Example:

```c
`a = b + c
`
```

Could be translated into:

```assembly
`MOV b, R1
ADD c, R1
MOV R1, a
`
```

✅ 4. Register Allocation and Assignment

- Efficient use of CPU **registers** is crucial.
- It includes:
  - **Allocation**: Deciding which variables should be kept in registers.
  - **Assignment**: Assigning physical registers to those variables.
- Poor register use can lead to **more memory access** and **slower performance**.

✅ 5. Order of Evaluation

- The **order in which expressions are evaluated** affects:
  - Number of registers required
  - Execution speed
- The code generator must find an **evaluation order** that uses the **least resources**.

✅ 6. Instruction Cost and Addressing Modes

- Different instructions and addressing modes have different **execution costs**.
- The generator must consider:
  - Which instructions are **faster**
  - Which modes are **supported** by the hardware

✅ 7. Machine-Dependent Constraints

- Code generator must follow **target architecture** rules such as:
  - Number of registers
  - Instruction format
  - Supported operations and data types

✅ 8. Correctness and Simplicity

- The generator must produce **correct code** first and foremost.
- Design should be **modular**, **maintainable**, and **easy to test**.

✅ Summary Table:

Issue	Description
Input to code generator	IR like 3AC, syntax tree, or DAG
Target code form	Absolute, relocatable, or assembly
Instruction selection	Choose efficient machine instructions
Register allocation	Optimize register usage
Order of evaluation	Choose optimal computation sequence
Instruction cost	Use cheaper instructions and addressing modes
Machine constraints	Follow architecture rules and features
Correctness & simplicity	Ensure accurate and maintainable code

---

Let me know if you want this turned into a formatted study sheet or PDF!

Write syntax directed definition with inherited attributes for type declaration for list of identifiers. Show annotated parse tree for the sentence real id1, id2, id3.

Sure! Here's a clear explanation with a **Syntax Directed Definition (SDD)** using **inherited attributes** for a **type declaration** with a list of identifiers. We'll also create the **annotated parse tree** for the sentence:

```go
`real id1, id2, id3
`
```

✅ 1. Grammar for Type Declaration:

We'll use the following grammar to describe a type declaration for a list of identifiers:

```bash
`D → T L
T → real | integer
L → L , id
L → id
`
```

- `D` → declaration
- `T` → type (e.g., `real`, `integer`)

- `L` → list of identifiers

✅ 2. Syntax Directed Definition (SDD) with Inherited Attributes

Let's assume:

- `T.type` is a **synthesized attribute** (type from keyword like `real`)
- `L.inh` is an **inherited attribute** that passes the type from `T` to all identifiers
- `L.ids` is a **synthesized attribute** which accumulates the identifier-type pairs

```plaintext
`1. D → T L
   L.inh = T.type

2. T → real
   T.type = "real"

3. T → integer
   T.type = "integer"

4. L → L1 , id
   L1.inh = L.inh
   L.ids = L1.ids ∪ {(id.name, L.inh)}

5. L → id
   L.ids = {(id.name, L.inh)}
`
```

✅ 3. Annotated Parse Tree for: `real id1, id2, id3`

We apply:

```r
`D → T L
T → real
L → L , id3
L → L , id2
L → id1
`
```

Now, annotate each node with attributes:

```plaintext
`                   D
                 /   \
               T      L                        ← L.inh = "real"
              |      /   \
           real    L      id3                ← L.inh = "real"
                  /  \
                 L    id2                    ← L.inh = "real"
                 |
               id1

Annotations:
T.type = "real"
L.inh = "real" at each L node
```

```
L.ids (final) = { (id1, real), (id2, real), (id3, real) }
```

✅ Final Output:

- **Type** of `id1`, `id2`, and `id3` is **real**
- Using inherited attribute `L.inh`, type info is passed down the tree
- This design supports semantic analysis and symbol table entry creation

---

Let me know if you'd like this drawn as a tree diagram or saved as a printable note!

Write a syntax directed definition of a simple desk calculator and draw an annotated parse tree for 4*3 + 2*5 n

Sure! Here's a clear and well-structured answer for your question on the **Syntax Directed Definition (SDD)** of a **simple desk calculator**, along with the **annotated parse tree** for the expression:

```
4 * 3 + 2 * 5
```

(Note: I assume `n` at the end of your question is a typo or placeholder, as it doesn't fit in the expression. Let me know if it has a specific meaning.)

✅ 1. Grammar for a Simple Desk Calculator

We'll define the grammar for arithmetic expressions with addition and multiplication (no parentheses for now):

```r
E → E + T
E → T
T → T * F
T → F
F → num
```

- `E` → expression
- `T` → term
- `F` → factor
- `num` → a number (e.g., 4, 3, 2, 5)

✅ 2. Syntax Directed Definition (SDD)

We define an attribute `val` for each non-terminal that represents its **computed value**.

```plaintext
`1. E → E1 + T        { E.val = E1.val + T.val }
2. E → T             { E.val = T.val }
3. T → T1 * F        { T.val = T1.val * F.val }
4. T → F             { T.val = F.val }
5. F → num           { F.val = num.lexval }
`
```

Where `num.lexval` is the numeric value of the terminal `num`.

✅ 3. Parse Tree for the Expression `4 * 3 + 2 * 5`

Let's apply the grammar to parse this expression step-by-step:

```r
`E → E + T
E → T
T → T * F
T → F
F → 4

T → F
F → 3

T → T * F
T → F
F → 2

F → 5
`
```

✅ Annotated Parse Tree with Computed Values

```r
`                            E (val = 22)
                           /    |    \
                    E (14)   +    T (8)
                       /              \
                T (14)              T (8)
               /      \            /   \
          T (4)    * F (3)    T (2)   * F (5)
           |          |        |        |
         F (4)      num(3)   F (2)    num(5)
           |
         num(4)
`
```

✅ Value Calculation Trace:

- `F → num(4)` → F.val = 4
- `F → num(3)` → F.val = 3
- `T → T * F` → 4 * 3 = 12

- `F → num(2)` ⟶ F.val = 2
- `F → num(5)` ⟶ F.val = 5
- `T → T * F` ⟶ 2 * 5 = 10
- `E → E + T` ⟶ 12 + 10 = **22**

✅ Final Answer:

- **Expression:** `4 * 3 + 2 * 5`
- **Evaluated Value:** `22`
- **SDD and annotated parse tree** compute the result during parsing using syntax-directed evaluation.

Let me know if you'd like this rendered as a tree diagram or saved as a PDF!

Translate the expression – (a*b)+(c*d)+(a*b*c) into
1. Quadruples
2. Triples
3. Indirect triples.

Here's how to **translate the expression**

```css
`−(a * b) + (c * d) + (a * b * c)
`
```

into **1) Quadruples, 2) Triples, and 3) Indirect Triples** using standard intermediate code representations.

✅ Step 1: Break the expression into subparts

Expression:

```css
`−(a * b) + (c * d) + (a * b * c)
`
```

We'll compute:

- `t1 = a * b`
- `t2 = -t1`
- `t3 = c * d`
- `t4 = t2 + t3`

- `t5 = a * b` (again, reused)
- `t6 = t5 * c`
- `t7 = t4 + t6`

✅ 1. Quadruples

Each line is in the form:
(operator, arg1, arg2, result)

Index	Operator	Arg1	Arg2	Result
(1)	*	a	b	t1
(2)	uminus	t1	—	t2
(3)	*	c	d	t3
(4)	+	t2	t3	t4
(5)	*	a	b	t5
(6)	*	t5	c	t6
(7)	+	t4	t6	t7

✅ **Final Result = t7**

✅ 2. Triples

Each line is in the form:
(operator, arg1, arg2)
Arguments can reference previous instruction numbers instead of using variable names.

Index	Operator	Arg1	Arg2
(0)	*	a	b
(1)	uminus	(0)	—
(2)	*	c	d
(3)	+	(1)	(2)
(4)	*	a	b
(5)	*	(4)	c
(6)	+	(3)	(5)

✅ **Final Result = (6)**

✅ 3. Indirect Triples

Same as triples, but instead of using index numbers directly, we use a pointer table to reference instructions.

Pointer Table:

Index	Points to
0	(0)

Index	Points to
1	(1)
2	(2)
3	(3)
4	(4)
5	(5)
6	(6)

Triples:

Instruction	Operator	Arg1	Arg2
(0)	*	a	b
(1)	uminus	0	—
(2)	*	c	d
(3)	+	1	2
(4)	*	a	b
(5)	*	4	c
(6)	+	3	5

✅ **Final Result = pointer[6]**

Let me know if you want a visual table format or saved as a note sheet!

Construct a syntax directed translation scheme that translate arithmetic expression from infix to postfix notation. Show the application of your scheme to the string: 3 * 4 + 5 * 2.

Sure! Here's a clear and structured answer to your question on converting an infix expression to postfix using a **Syntax Directed Translation (SDT)** scheme.

✅ Goal:

Convert **infix expression** → **postfix notation**

✅ 1. Grammar for Arithmetic Expressions

We will use the following grammar for expressions involving `+` and `*`:

```r
`E → E + T
E → T
T → T * F
T → F
F → digit
`
```

Here:

- `E`, `T`, `F` are non-terminals for expression, term, and factor.
- `digit` is a terminal representing numbers.

✅ 2. Syntax Directed Translation Scheme (SDT)

We attach **semantic actions** (postfix output) to the grammar:

```r
`E → E1 + T     { print("+"); }
E → T
T → T1 * F     { print("*"); }
T → F
F → digit      { print(digit.lexval); }
`
```

✅ **Action:** Whenever a production rule is completed, the corresponding operator or operand is printed **after** its operands — this produces **postfix notation**.

✅ 3. Apply the Scheme to Expression:

Infix: `3 * 4 + 5 * 2`

🔁 Derivation:

```r
`E
→ E + T
→ T + T
→ T * F + T
→ F * F + T
→ 3 * 4 + T
→ 3 * 4 + T * F
→ 3 * 4 + F * F
→ 3 * 4 + 5 * 2
`
```

📌 Postfix Output Sequence (left to right evaluation):

- `F → 3` → print `3`
- `F → 4` → print `4`
- `T → T * F` → print `*`
- `F → 5` → print `5`
- `F → 2` → print `2`
- `T → T * F` → print `*`
- `E → E + T` → print `+`

✅ Final Postfix Output:

```
`3 4 * 5 2 * +
`
```

✔️ This is the postfix form of the infix expression `3 * 4 + 5 * 2`.

Let me know if you want this drawn into a syntax tree or explained step-by-step visually!

Draw syntax tree and DAG for the statement a = (a * b + c) ^ (b + c) * b + c. Write three address codes from both.

Great! Let's break this complex expression step by step.

✅ Given Expression:

```r
`a = (a * b + c) ^ (b + c) * b + c
`
```

✅ Step 1: Add Parentheses to Clarify Order of Operations

Based on standard operator precedence:

- `^` (exponentiation) has the highest precedence (assuming it's **right associative**)
- Then `*`
- Then `+`

So the expression becomes:

```r
`a = (((a * b) + c) ^ (b + c)) * b + c
`
```

✅ Step 2: Construct the Syntax Tree

```r
`
                 =
               /   \
             a       +
                   / \
                 *      c
```

```
          / \
         ^     b
        / \
       +     +
      / \   / \
     *   c  b   c
    / \
   a   b
```

✅ Step 3: DAG (Directed Acyclic Graph)

In DAG, we **eliminate common subexpressions**. Here, `b + c` appears twice.

Nodes:

- `a`, `b`, `c` — identifiers
- `t1 = a * b`
- `t2 = t1 + c`
- `t3 = b + c` (**shared**)
- `t4 = t2 ^ t3`
- `t5 = t4 * b`
- `t6 = t5 + c`
- `a = t6`

✅ Step 4: Three-Address Code from Syntax Tree (No common sub-expression elimination)

```ini
t1 = a * b
t2 = t1 + c
t3 = b + c
t4 = t2 ^ t3
t5 = t4 * b
t6 = t5 + c
a = t6
```

✅ Step 5: Three-Address Code from DAG (with Common Subexpression `b + c`)

Only difference is we avoid recomputing `b + c`.

```ini
t1 = a * b
t2 = t1 + c
```

```
t3 = b + c      // common subexpression
t4 = t2 ^ t3
t5 = t4 * b
t6 = t5 + c
a = t6
```

✔️ As you can see, both sets look the same **here** because we **manually reused** `b + c` — so DAG makes sure it's optimized by avoiding recomputation.

Would you like me to draw both the syntax tree and DAG as images for easier understanding?

Construct DAG for a + a * (b- c) + (b – c) * d. also generates three address code for same.

Great! Let's work step-by-step to construct the **DAG** for the expression and generate the **three-address code (TAC)**.

✅ Given Expression:

```css
a + a * (b - c) + (b - c) * d
```

✅ Step 1: Identify Subexpressions

We'll apply standard operator precedence:

1. Parentheses first
2. Multiplication
3. Addition

So we break it down as:

```css
a + (a * (b - c)) + ((b - c) * d)
```

We clearly see that **(b - c)** is a **common subexpression**.

✅ Step 2: DAG Construction

We'll represent the operations and reuse nodes when possible.

🎯 Operands:

- `a`, `b`, `c`, `d`

🎯 Subexpressions:

1. `t1 = b - c` ← used twice
2. `t2 = a * t1`
3. `t3 = t1 * d`
4. `t4 = a + t2`
5. `t5 = t4 + t3`

✅ DAG Structure (Textual View):

```markdown
`
        +
      /   \
    +       *
   / \     / \
  a   *   t1   d
     / \
    a   t1
         |
       b - c
`
```

✔️ `t1 = b - c` is reused in both multiplications — this is the key **optimization** the DAG enables.

✅ Three Address Code from DAG

```ini
`t1 = b - c
t2 = a * t1
t3 = t1 * d
t4 = a + t2
t5 = t4 + t3
`
```

✅ Final result is in `t5`

✅ Final Answer:

Three Address Code:

```c
`t1 = b - c
t2 = a * t1
t3 = t1 * d
t4 = a + t2
```

```
t5 = t4 + t3
`
```

✅ **Result:** Expression evaluates to `t5`

Would you like a visual diagram of the DAG or this formatted into a study note or printable sheet?

Define: DAG. Explain DAG representation of basic block with example

Sure! Here's a concise yet slightly descriptive answer to your question:

✅ Definition: DAG (Directed Acyclic Graph)

A **Directed Acyclic Graph (DAG)** is a data structure used in compiler design to represent the expressions and computations in a **basic block**. It is:

- **Directed**: Each edge has a direction (parent → child)
- **Acyclic**: No cycles exist in the graph
- Used to detect **common sub-expressions** and perform **code optimization**

✅ Purpose of DAG in Compiler Design

- Eliminates **redundant calculations**
- Helps in **register allocation**
- Simplifies **intermediate code generation**
- Ensures **efficient use** of computation and storage

✅ Basic Block

A **basic block** is a straight-line code sequence with:

- No branches in except to the entry
- No branches out except at the exit

✅ DAG Representation of a Basic Block – Example

Three-address code (TAC):

```
c

`t1 = a * b
t2 = a * b
t3 = t1 + c
```

```
t4 = t2 + c
`
```

✅ DAG Construction Steps

1. Create leaf nodes for operands `a`, `b`, and `c`
2. Create one node for `a * b` and label it with both `t1` and `t2` (common subexpression)
3. Create a `+` node with children `(a * b)` and `c`, and label it `t3` and `t4`

✅ DAG Diagram (Text View):

```markdown
`
           +
         /   \
       *       c
     /   \
    a     b

Labels:
* → t1, t2
+ → t3, t4
`
```

✅ Benefits of DAG:

- **Common Subexpression Elimination**: No need to compute `a * b` twice.
- **Efficient Code Generation**: Generates optimized three-address code.
- **Space & Time Saving**: Reuses computations wherever possible.

Would you like a visual diagram of this DAG? I can generate one for your notes or for a presentation!

Describe any three data structure for symbol table.

Sure! Here's a neat and slightly descriptive explanation of **three data structures used for implementing a Symbol Table**:

✅ What is a Symbol Table?

A **Symbol Table** is a **data structure** used by a compiler to keep track of **identifiers** (like variable names, function names, constants, etc.) along with their attributes such as:

- Type
- Scope
- Memory location

- Value (if constant)

✅ 1. Linear List (Array or Linked List)

📌 Description:

- The simplest way to store symbols.
- Each entry in the list contains the name and information (like type, scope, etc.).
- Search is done **linearly** (one by one).

✅ Example:

```text
`+--------+--------+--------+
| Name   | Type   | Scope  |
+--------+--------+--------+
|  x     | int    | local  |
|  y     | float  | global |
|  z     | char   | local  |
+--------+--------+--------+
`
```

⚠️ Drawback:

- **Slow lookup time** for large programs (O(n))
- Insertion is easy, but search is inefficient.

✅ 2. Self-Organizing List

📌 Description:

- It's a variation of a linear list.
- Frequently accessed names are moved **toward the front** of the list.
- Uses **linked list** with an added field to track references.

✅ Advantage:

- Reduces search time for frequently used identifiers.
- Good for languages where some symbols are used more often.

⚠️ Drawback:

- Still linear in worst case
- Rearranging on each access adds overhead

✅ 3. Hash Table

📌 Description:

- Uses a **hash function** to map identifier names to table indices.
- Offers **constant-time average case** for lookup and insertion.

✅ Structure:

- Maintains two tables:
  - **Hash table** (array of pointers or indices)
  - **Symbol table** (actual records)

✅ Example:

```text
`Hash(name) → index
index → [Name, Type, Scope, Address...]
`
```

✅ Advantage:

- **Fast lookup** and insertions (O(1) on average)
- Efficient for large-scale programs

⚠️ Drawback:

- Needs a good hash function to avoid **collisions**
- Handling collisions (e.g., chaining or open addressing) adds complexity

---

✅ Summary Table:

Data Structure	Pros	Cons
Linear List	Easy to implement	Slow search (O(n))
Self-Organizing List	Faster for frequent items	Still linear, extra overhead
Hash Table	Fast lookup (O(1) average)	Collisions & hash management

---

Let me know if you'd like diagrams or examples added to your notes!

Define: 1. Synthesized Attribute
2. Inherited attribute
3. Dependency graph
4.Translation Scheme

Sure! Here's a simple and clear explanation of each of the four definitions:

✅ 1. Synthesized Attribute

A **synthesized attribute** is an attribute whose value is **computed from the attributes of its children** in the parse tree.

- It flows **bottom-up** in the syntax tree.
- Mostly used in **bottom-up parsing**.

📌 Example:

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

Here, `E.val` is synthesized using `E1.val` and `T.val`.

✅ 2. Inherited Attribute

An **inherited attribute** is an attribute whose value is **computed from the attributes of its parent or siblings** in the parse tree.

- It flows **top-down** or **sideways**.
- Mostly used in **top-down parsing**.

📌 Example:

```plaintext
L → L1 , id     { L1.inh = L.inh }
```

Here, `L1` inherits `L.inh` from its parent node `L`.

✅ 3. Dependency Graph

A **dependency graph** shows the **flow of data (attribute values)** between grammar symbols in a parse tree.

- Each **node** in the graph is an attribute.
- Each **edge** indicates a dependency between attributes.
- It helps in determining the **evaluation order** for attributes.

📌 Purpose:

- To **check for circular dependencies**

- To **schedule attribute evaluation** properly

✅ 4. Translation Scheme

A **translation scheme** is a grammar that includes **semantic actions** embedded within the productions.

- These actions are typically written in `{ }` brackets.
- They are executed **during parsing** to generate intermediate code or perform computations.

📌 **Example:**

```plaintext
`E → E1 + T { print("+") }
`
```

This translation scheme converts an infix expression into postfix by printing the operator after evaluating operands.

Let me know if you'd like these formatted into a study sheet or visualized with diagrams!