Q. Explain the use of symbol table in compilation process. List out the various attributes for implementing the symbol table.

Use of Symbol Table in the Compilation Process

1. **Lexical Analysis**:
   - The lexical analyzer (scanner) identifies tokens and inserts identifiers into the symbol table.
2. **Syntax Analysis**:
   - The parser uses the symbol table to ensure that identifiers used in the code are correctly defined and to check for any syntax errors.
3. **Semantic Analysis**:
   - The compiler checks for semantic consistency using the symbol table, such as type checking and ensuring the correct scope of variables.
4. **Intermediate Code Generation**:
   - The symbol table provides necessary information about identifiers to generate intermediate representations.
5. **Optimization**:
   - Information from the symbol table is used to optimize code by understanding the properties and usage of variables.
6. **Code Generation**:
   - The symbol table helps map variable names to memory addresses or registers in the final target code.

Various Attributes for Implementing the Symbol Table

A symbol table typically includes the following attributes for each entry:

1. **Name**:
   - The identifier's name (e.g., variable, function).
2. **Type**:
   - Data type of the identifier (e.g., `int`, `float`, `char`).
3. **Scope**:
   - The scope in which the identifier is valid (e.g., local, global, function-level).
4. **Memory Location/Address**:
   - The memory location where the identifier's value is stored.
5. **Size**:
   - The size in bytes of the identifier (e.g., size of an array or structure).
6. **Value**:
   - The current value of constants or initialized variables.
7. **Line Number**:
   - The line number(s) where the identifier appears in the source code (useful for error messages and debugging).
8. **Attributes/Modifiers**:
   - Information such as `const`, `static`, `volatile`, etc., indicating how the variable can be used or modified.
9. **Function Parameters** (for function symbols):
   - List of parameter types and names, used for type checking function calls.
10. **Return Type** (for function symbols):
    - The return type of a function (e.g., `void`, `int`).
11. **Symbol Category**:
    - Specifies whether the symbol is a variable, function, class, or object.

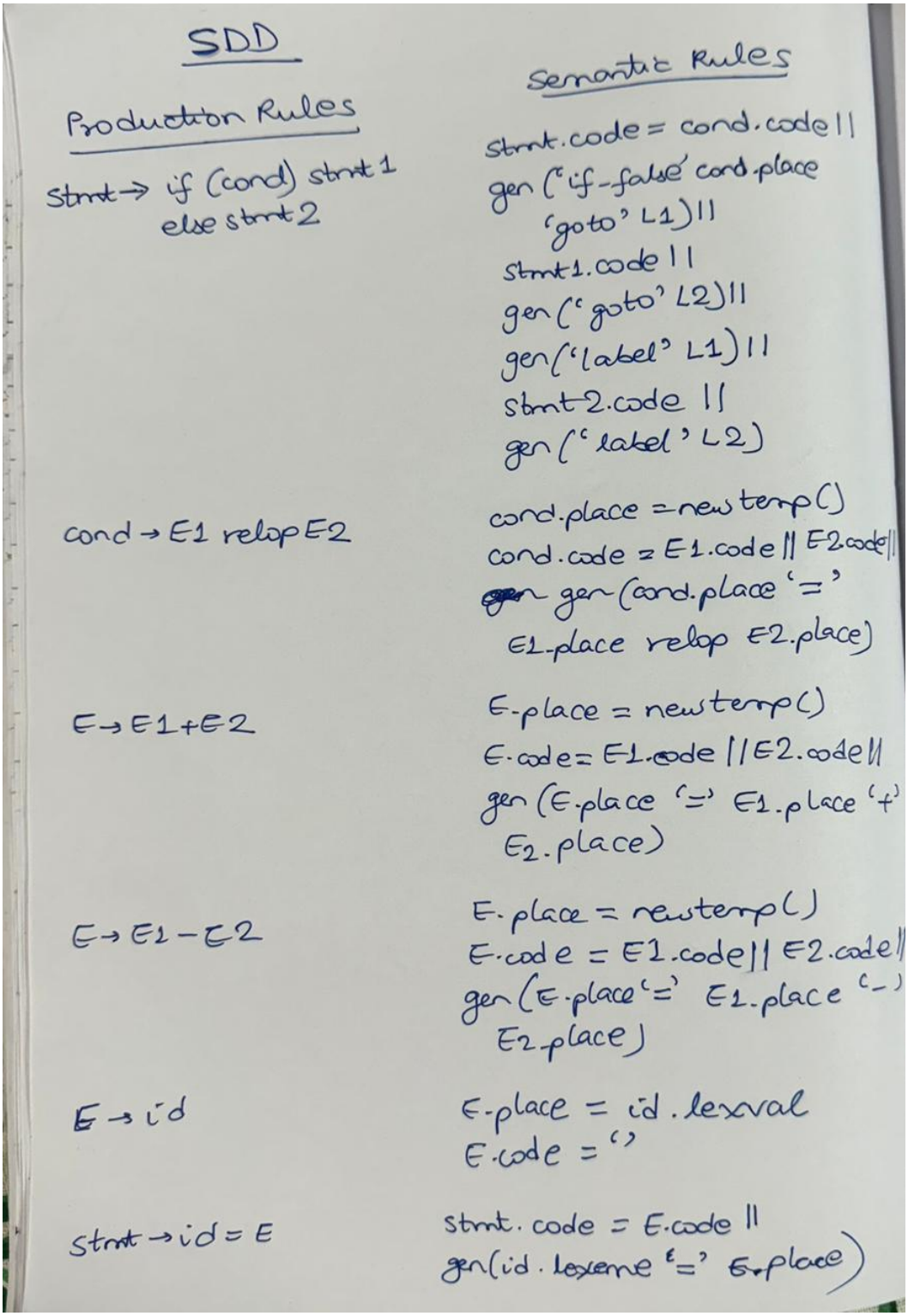
12. **Pointer Information**:
    - Indicates if the identifier is a pointer and what type of data it points to.
13. **Attributes for Array/Structure**:
    - Information such as dimensions of an array or fields of a structure.
14. **Access Control** (in OOP languages):
    - Access specifiers like `public`, `private`, and `protected` for classes and members.

Q.

Generate intermediate code for the following code segment along with the required syntax directed definition:

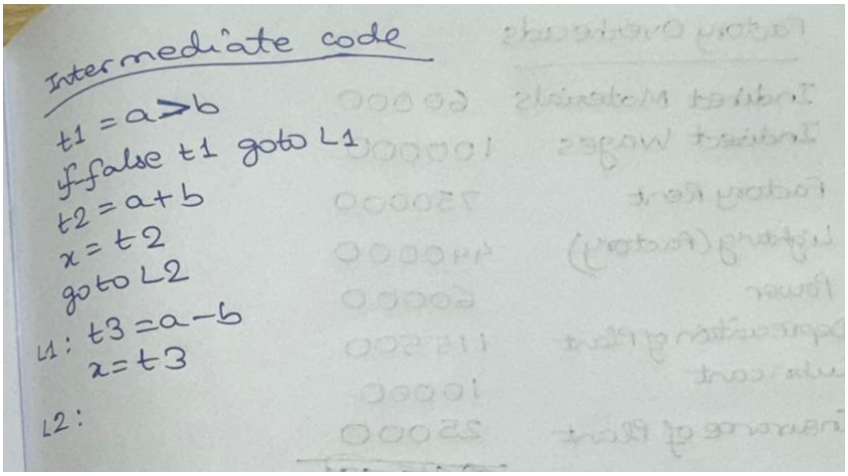
```
if (a > b)
    x = a+b;
else
    x = a-b;
```

Here datatype for x, a and b are int.



SDD

Production Rules	Semantic Rules
Stmt → if (cond) stmt1 else stmt2	Stmt.code = cond.code || gen('if-false' cond.place 'goto' L1) || Stmt1.code || gen('goto' L2) || gen('label' L1) || Stmt2.code || gen('label' L2)
cond → E1 relop E2	cond.place = newtemp() cond.code = E1.code || E2.code || gen(cond.place '=' E1.place relop E2.place)
E → E1+E2	E.place = newtemp() E.code = E1.code || E2.code || gen(E.place '=' E1.place '+' E2.place)
E → E1-E2	E.place = newtemp() E.code = E1.code || E2.code || gen(E.place '=' E1.place '-' E2.place)
E → id	E.place = id.lexval E.code = ''
Stmt → id = E	Stmt.code = E.code || gen(id.lexeme '=' E.place)



Q. Explain the algebraic translations of local machine-independent optimisations

Types of Local Machine-Independent Optimizations

1. **Constant Folding**:
   - **Explanation**: Compile-time evaluation of constant expressions.
   - **Example**: An expression like `3 * 4 + 5` is simplified to `17` during compilation.
   - **Algebraic Translation**: Replace `a + b` with `c` when `a` and `b` are constants and `c = a + b`.
2. **Strength Reduction**:
   - **Explanation**: Replace expensive operations with equivalent, cheaper ones.
   - **Example**: Multiplication by a power of 2, such as `x * 8`, can be replaced by `x << 3` (bit-shifting).
   - **Algebraic Translation**: Replace `x * 2^n` with `x << n`.
3. **Algebraic Simplification**:
   - **Explanation**: Use algebraic identities to simplify expressions.
   - **Examples**:
     - Replace `x * 1` with `x`.
     - Replace `x + 0` with `x`.
     - Replace `x - x` with `0`.
   - **Algebraic Translation**:
     - `x * 1 → x`
     - `x + 0 → x`
     - `x - x → 0`
4. **Common Subexpression Elimination (CSE)**:
   - **Explanation**: Identifies and eliminates duplicate computations.
   - **Example**: If `t1 = a + b` and `t2 = a + b` appear in the code, `t2` can be replaced by `t1` to avoid recomputation.

- **Algebraic Translation**: Identify equivalent expressions `E1` and `E2` such that `E1 = E2` and replace occurrences to reuse computed values.

5. **Strength Reduction of Division and Modulo**:
   - **Explanation**: Replace division or modulo operations with bit-shifting when applicable.
   - **Example**: Replace `x / 2` with `x >> 1` for unsigned integers.
   - **Algebraic Translation**: Replace `x / 2^n` with `x >> n`.
6. **Reassociation**:
   - **Explanation**: Change the grouping of operations to simplify computations and take advantage of constant folding.
   - **Example**: `a + (b + c)` can be rewritten as `(a + b) + c` if `b + c` is a known constant.
   - **Algebraic Translation**: Reassociative laws like `a + (b + c)` → `(a + b) + c` or `a * (b * c)` → `(a * b) * c`.
7. **Operator Strength Reduction**:
   - **Explanation**: Replace high-cost operations with lower-cost operations that achieve the same result.
   - **Example**: Replace `x * 0.5` with `x / 2` or replace division by a constant with multiplication by its reciprocal.
   - **Algebraic Translation**: Replace `x / c` (where `c` is a constant) with `x * (1/c)`.
8. **Code Motion (Loop Invariant Code Motion)**:
   - **Explanation**: Move calculations that do not change within a loop outside the loop to avoid repeated computation.
   - **Example**: In `for (i = 0; i < n; i++) { y = a + b; /* use y */ }`, `a + b` can be computed once before the loop.
   - **Algebraic Translation**: Identify an expression `E` in a loop where `E` does not change with each iteration and hoist it outside the loop.

Q. Discuss the following: i) Dead code elimination ii) copy propagation

i) Dead Code Elimination

Dead code elimination is a compilation optimization technique that removes code segments that do not affect the program's observable behavior. This type of code is known as "dead code" because it is never executed or its result is never used.

Purpose:

- **Optimize the code**: By removing unnecessary instructions, the compiler can produce a smaller, faster, and more efficient executable.
- **Improve readability and maintainability**: Removing dead code simplifies the intermediate representation, making it easier to optimize further.

How it Works:

- The compiler analyzes the program's control flow and data flow to detect code that does not influence the program's output or behavior.
- Examples include variables assigned but never used, or entire branches of code that are unreachable due to conditions that are always `false`.

Example:

```
int a = 5;
int b = 10; // Dead code: assignment to b is never used
int result = a * 2;
return result;
```

In this example, `b = 10` is never used or referenced and can be safely removed by the compiler.

Types of Dead Code:

1. **Unreachable Code**: Code that follows a return, break, or exit statement.
2. **Unused Variables and Assignments**: Assignments made to variables that are never used.

Process:

- **Mark and sweep**: The compiler marks code that is useful (i.e., code that has a visible effect or contributes to output) and sweeps away unmarked code.

ii) Copy Propagation

Copy propagation is an optimization that simplifies code by replacing occurrences of variables with their assigned values, where possible. This optimization helps remove redundant variables and assignments, leading to more concise code.

Purpose:

- **Reduce redundant variable usage**: Simplifies expressions and reduces the number of variables in use.
- **Enhance further optimizations**: By simplifying code, it enables other optimization techniques to be more effective, such as dead code elimination.

How it Works:

- If a variable `b` is assigned the value of another variable `a`, subsequent uses of `b` can be replaced with `a`.
- The compiler ensures that `a` has not changed since the assignment to `b` before performing this replacement.

Example:

```
int a = 5;
int b = a;  // Copy statement
int c = b + 2;
```

After **copy propagation**, the code would be:

```
int a = 5;
int c = a + 2;
```

In this case, the variable `b` is removed, and `c` is directly computed using `a`.

Key Considerations:

- **Side effects**: Copy propagation must ensure that replacing a variable does not change the program's behavior (e.g., through side effects or aliasing).
- **Liveness analysis**: The compiler needs to ensure that the variable being propagated remains unchanged between its assignment and its use.

Advantages:

- **Reduces variable clutter**: By reducing the number of temporary variables, the code becomes more efficient.
- **Enables further optimizations**: Simplifies code structure for subsequent optimizations like dead code elimination.

Limitations:

- Copy propagation may not be beneficial in cases where copying is necessary for readability or specific control flow, so it must be used judiciously.

Q. Explain loop optimisation in detail using a suitable example

Loop optimization refers to a set of techniques used to improve the performance of loops within a program by minimizing the computational overhead and maximizing execution efficiency. Loops are a crucial part of many programs, and optimizing them can lead to significant performance gains.

Types of Loop Optimization Techniques

1. **Loop Invariant Code Motion**:
   - Moves code that does not change within the loop (invariant code) outside the loop to avoid repeated evaluation.
2. **Loop Unrolling**:
   - Expands the loop body multiple times to reduce the overhead of loop control instructions.
3. **Loop Fusion**:
   - Combines two or more loops that iterate over the same range into a single loop to reduce loop overhead.
4. **Loop Fission (Loop Distribution)**:
   - Splits a loop into multiple smaller loops to improve parallel execution or reduce memory access conflicts.
5. **Strength Reduction**:
   - Replaces expensive operations within the loop with cheaper ones (e.g., replacing multiplication with addition).
6. **Induction Variable Elimination**:
   - Simplifies and reduces the number of induction variables used in the loop.
7. **Loop Blocking (Tiling)**:
   - Improves cache performance by breaking the loop into blocks to work on chunks of data that fit in the cache.

Detailed Explanation with an Example: Loop Invariant Code Motion

Loop invariant code motion is a technique where expressions that yield the same result each iteration of the loop are moved outside the loop. This helps avoid redundant calculations and improves performance.

Example: Consider the following simple C code:

```
for (int i = 0; i < n; i++) {
    int y = a * b;  // This calculation is loop-invariant
    arr[i] = y + i;
}
```

Explanation:

- The expression `a * b` does not change during each iteration of the loop; it is independent of the loop variable `i`.
- Calculating `a * b` inside the loop leads to repeated computations for every iteration, which is inefficient.

Optimization Process:

- Move the invariant code outside the loop:

```
int y = a * b;  // Move invariant computation outside the
loop
for (int i = 0; i < n; i++) {
    arr[i] = y + i;
}
```

Benefits:

- **Reduced computation**: By moving `int y = a * b;` outside the loop, the multiplication operation is performed only once, not `n` times.
- **Improved performance**: Fewer operations inside the loop body lead to faster execution, especially for large values of `n`.

Other Loop Optimization Techniques Explained with Examples

1. **Loop Unrolling**:
    - This technique reduces loop control overhead by repeating the loop body multiple times per iteration.
    -

```
// Original loop
for (int i = 0; i < n; i++) {
    arr[i] = arr[i] * 2;
}

// Unrolled version (with a factor of 2)
```

```
for (int i = 0; i < n; i += 2) {
    arr[i] = arr[i] * 2;
    if (i + 1 < n) {
        arr[i + 1] = arr[i + 1] * 2;
    }
}
```

2. **Strength Reduction**:

- Replaces costly operations with equivalent, simpler ones.

```
// Original code
for (int i = 0; i < n; i++) {
    arr[i] = i * 5;  // Multiplication inside the loop
}

// Strength reduced version
int multiplier = 0;
for (int i = 0; i < n; i++) {
    arr[i] = multiplier;  // Use addition instead
    multiplier += 5;
}
```

3. **Loop Fusion**:

- Combines two loops that iterate over the same range.

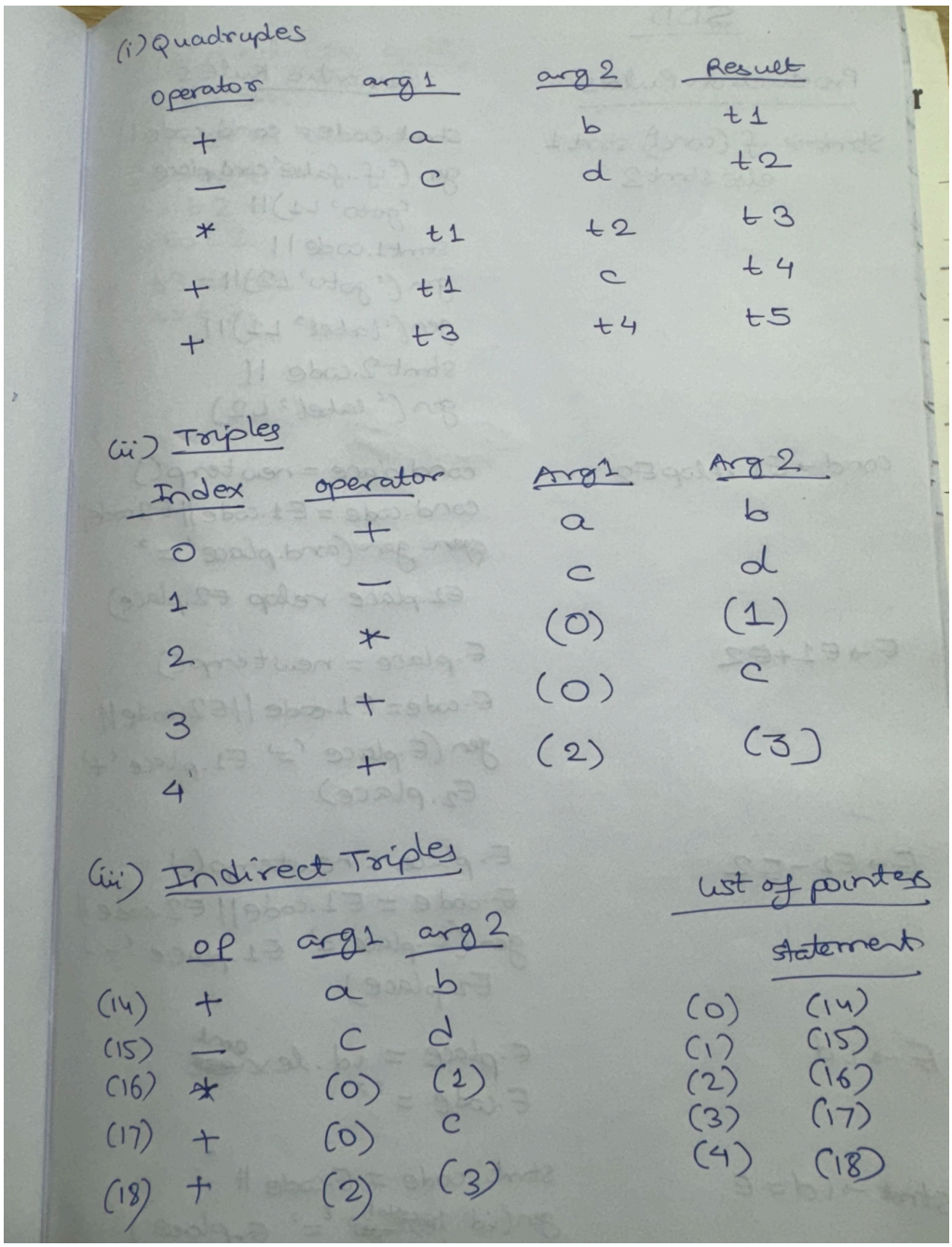
```
// Original code
for (int i = 0; i < n; i++) {
    arr1[i] = arr1[i] * 2;
}
for (int i = 0; i < n; i++) {
    arr2[i] = arr2[i] + 3;
}

// Fused version
for (int i = 0; i < n; i++) {
    arr1[i] = arr1[i] * 2;
    arr2[i] = arr2[i] + 3;
```

- }

General Impact of Loop Optimization

- **Reduced execution time**: Loop optimizations minimize the number of instructions executed during each iteration.
- **Improved memory access patterns**: Techniques like loop blocking can help ensure data fits better in the CPU cache, leading to faster memory access.
- **Increased parallelism**: Optimizations such as loop fission can facilitate parallel execution by splitting loops into smaller independent loops.

Q. Translate the expression: (a+b)*(c-d)+(a+b+c) into: (i) quadruples, (ii) triples and (iii) indirect triples



Q. List the fields in an activation record. Write down the purpose of each of these fields in an activation record.

1. Actual Parameters (Arguments)

- **Purpose**: Stores the values or references passed to the function by the caller. These values are essential for the function's execution as they are the data it operates on.
- **Usage**: The actual parameters are used within the function to perform computations or control logic. They are typically placed in the activation record so that the function can easily access them throughout its execution.

2. Return Values

- **Purpose**: Holds the result of the function that is passed back to the calling function. This field is necessary for functions that have a return type and need to communicate a result.
- **Usage**: After the function completes its execution, the return value is set, allowing the caller to receive and use the output.

3. Control Link (Dynamic Link)

- **Purpose**: Points to the activation record of the caller (the function that called the current function). This helps in returning control to the appropriate context after the function ends.
- **Usage**: Ensures that when the function completes, the program can continue execution from where the call was made. It supports stack-based function call management and nested function calls.

4. Access Link (Static Link)

- **Purpose**: Points to the activation record of the nearest lexically enclosing function. It facilitates the access of non-local variables that belong to an enclosing scope.
- **Usage**: Particularly important in languages that support nested functions, as it allows a function to access variables in its parent functions' scopes.

5. Saved Memory State (Saved Registers)

- **Purpose**: Stores the values of CPU registers or memory states that need to be preserved across function calls. This ensures that the calling function's context (register values, program counters, etc.) is intact after the call returns.
- **Usage**: This field allows the program to use registers temporarily during function execution without affecting the caller's state.

6. Local Variables

- **Purpose**: Allocates space for the function's own local variables, which are only accessible within the function's scope.
- **Usage**: These variables are used to store intermediate results, function-specific data, or other necessary information during the function's execution.

7. Temporary Data (Temporary Variables)

- **Purpose**: Holds intermediate values during the execution of the function. These can be results of sub-expressions or temporary storage required for calculations.
- **Usage**: Temporary data is often used when evaluating complex expressions or during the execution of intermediate steps within the function.

Q. Explain the sequence of stack allocation process for a function call using suitable example

The **stack allocation process** for a function call involves a series of steps that the compiler and runtime system execute to set up and manage function calls. The process ensures that each function invocation has its own **activation record (stack frame)**, which stores information specific to that call.

Sequence of Stack Allocation Process

1. **Caller Sets Up the Actual Parameters**:
    - The caller places the values of the actual parameters onto the stack.
    - **Example**: If `foo(x, y)` is called from `main()`, `main()` will push `x` and `y` onto the stack.
2. **Caller Saves the Return Address**:
    - The address of the instruction to return to after the function call is saved in the stack.
    - This allows the program to continue execution at the correct location after the function completes.
3. **Caller Saves the Control Link (Dynamic Link)**:
    - A pointer to the current activation record of the caller is saved. This links the new activation record to the previous one, supporting stack-based management.
4. **Callee Allocates Space for Local Variables**:
    - When the called function begins execution, space for its local variables is allocated on the stack.
    - **Example**: If `foo()` has local variables `int a` and `int b`, space for these variables will be reserved in `foo()`'s activation record.
5. **Callee Sets Up the Access Link (Static Link)**:
    - The callee sets up the access link if needed. This points to the activation record of the nearest lexically enclosing function, enabling access to non-local variables.
6. **Saved Registers and State**:
    - Any registers or memory state that might be modified during the function's execution are saved in the activation record to preserve the caller's state.
7. **Execution of the Function**:
    - The function executes its code, utilizing the local variables, temporary data, and accessing parameters from the activation record.
8. **Storing the Return Value**:
    - If the function has a return value, it is stored in a designated area of the activation record to be passed back to the caller.
9. **Callee Cleans Up and Returns Control**:
    - Before returning, the callee may deallocate the space used for local variables and temporary data.
    - The return address stored in the stack is used to transfer control back to the caller.
10. **Caller Cleans Up the Stack**:

- The caller removes the actual parameters from the stack and restores the saved state, including registers and program counter.

Example of Stack Allocation Process

Consider the following code:

```
void foo(int a, int b) {
    int sum = a + b;
    // Other operations...
}

int main() {
    int x = 5;
    int y = 10;
    foo(x, y);
    // Rest of the code...
}
```

Step-by-Step Stack Allocation

1. **`main()` sets up the actual parameters**:
   - `x` and `y` (values `5` and `10`) are pushed onto the stack.
2. **`main()` saves the return address**:
   - The address of the instruction in `main()` that comes after the call to `foo()` is pushed onto the stack.
3. **`main()` saves the control link**:
   - A pointer to `main()`'s activation record is pushed onto the stack to maintain the call chain.
4. **`foo()` allocates space for local variables**:
   - `foo()` pushes space for `sum` onto the stack.
5. **`foo()` sets up the access link (if needed)**:
   - In cases of nested functions, the access link would point to `main()`'s activation record.
6. **Saved registers**:
   - If `foo()` modifies registers used by `main()`, their original values are saved.
7. **Execution of `foo()`**:
   - `foo()` executes, using the local variable `sum` and accessing `a` and `b` from the stack.
8. **Storing the return value**:
   - If `foo()` had a return value (e.g., `return sum;`), it would be stored in a designated place in the activation record or in a specific register.

9. **`foo()` cleans up and returns control**:
    - The return address is used to transfer control back to `main()`.
10. **`main()` cleans up**:
    - The actual parameters `x` and `y` are removed from the stack, and any saved registers are restored.

Visual Representation of Stack During `foo(x, y)`

```
|         ...          |
|----------------------|
| Actual Parameters    | <-- x, y pushed by `main()`
|----------------------|
| Return Address       | <-- Address of `main()` after `foo()`
|----------------------|
| Control Link         | <-- Pointer to `main()`'s activation
record
|----------------------|
| Local Variables      | <-- Space for `int sum`
|----------------------|
| Temporary Data       |
|----------------------|
```

Summary

The stack allocation process manages function calls by setting up and tearing down activation records. It ensures that data, control flow, and state are maintained properly between function invocations, allowing the program to execute correctly and efficiently.

Q. Explain the simplification of simple type checker for statements, expressions and functions.

Type Checking of Expressions

Table: Associated rules for Type Checking

Productions	Associated rules for type
$E \rightarrow literal$	E.type = char
$E \rightarrow num$	E.type = integer
$E \rightarrow id$	E.type = lookup(id.entry)
$E \rightarrow E_1 mod E_2$	E.type = **if** $E_1.type = integer$ **and** $E_2.type = integer$ **then** *integer* **else** *type_error*
$E \rightarrow E_1[E_2]$	E.type = **if** $E_{2.type} = integer$ **and** $E_{1.type} = array(s,t)$ **then** t **else** *type_error*
$E \rightarrow E_1 \uparrow$	E.type = if $E_{1.type}$=pointer(t) then t **else** *type_error*

Type Checking of Statements
The state statements we consider are assignment, conditional, and while statements.

Table: default

Productions	Associated rules for type
$S \rightarrow id = E$	{S.type = **if** id.type == E.type **then** void **else** *type_error* }
$S \rightarrow$ *if* E then S_1	{S.type = **if** E.type == Boolean **then** S_1.*type* **else** *type_error* }
$S \rightarrow$ *while* E do S_1	{S.type = **if** E.type == Boolean **then** S_1.*type* **else** *type_error* }
$S \rightarrow S_1 ; S_2$	{S.type = **if** S_1.type == void **and** S_2.type == void **then** void **else** *type_error* }

Type Checking of Functions

Productions	Associated actions
$T \rightarrow T_1 \ ' \rightarrow ' \ T_2$	$\{T.type = T_1.type \rightarrow T_2.type\}$
$E \rightarrow E_1(E_2)$	{E.type = if $E_2.type$ == s and E_1.type == $s \rightarrow t$ **then** t **else** *type_error* }

Q. Write down the three-address code and construct the basic blocks for the following program segment:

```
sum = 0; i=0;
while (i <=10)
{
sum = sum + a[i]
i++;
}
```

where the datatype for a, b and x are integer

Algorithm: Partition into Basic Block

Input: A sequence of Three Address Statement.
Output: A list of Basic Blocks with each TAC in exactly one block.

Method:

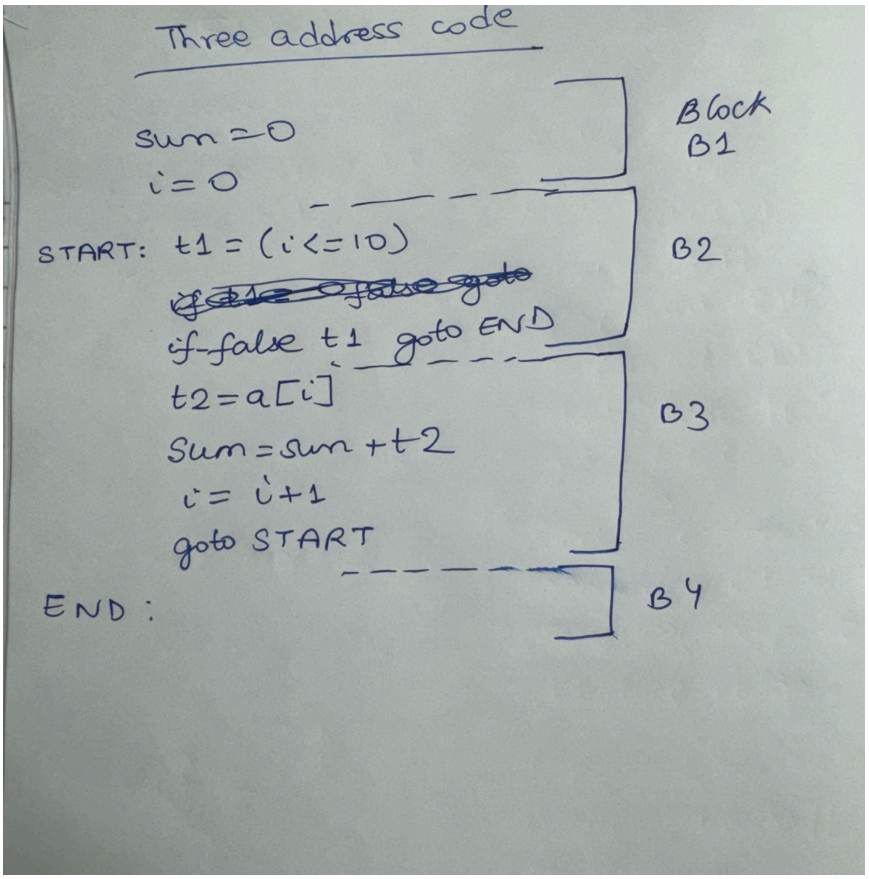
1. Determine the leaders, the first statements of Basic blocks.

   i) The first statement is a leader.

   ii) Any statement that is the target of a conditional or unconditional goto is a leader.

   iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.



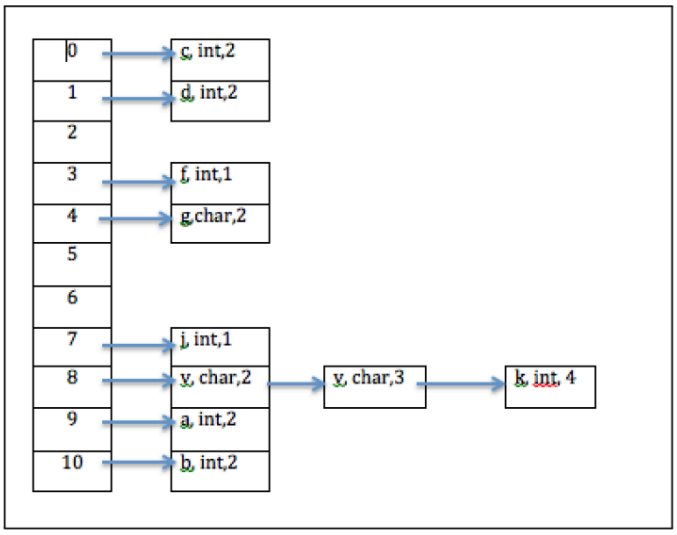
Q. How can hashing be used to design symbol table?

Hash Table

- Table of k pointers numbered from *zero* to $k-1$ that points to the symbol table and a record within the symbol table.
- To enter a name in to the symbol table we found out the hash value of the name by applying a suitable hash function.
- The hash function maps the name into an integer between *zero* and $k-1$ and using this value as an index in the hash table.

$$H_{Key} = ASCII_{id} \ mod \ 11$$

a	b	c	d	e	f	g	h	i	j
97	98	99	100	101	102	103	104	105	106



Q. Explain the characteristics of peephole optimisation technique

Peephole optimization is a **local optimization technique** used in compilers to improve the performance of the generated code by making small, localized changes to a sequence of instructions. This process typically operates on a window or "peephole" of instructions within the intermediate or machine-level code and looks for patterns that can be simplified or replaced with more efficient alternatives.

Here are the key characteristics of the **peephole optimization** technique:

1. Local Optimizations:

- **Scope**: Peephole optimization works on a small subset of instructions, often within a small window (usually 2 to 5 instructions). It does not look at the entire program or global code structure but focuses only on optimizing the sequence of consecutive instructions.
- **Optimization Focus**: The primary goal is to find small opportunities for improving performance or reducing code size within a local context.

2. Pattern Matching:

- **Searching for Patterns**: The optimizer searches for specific instruction patterns in the code that are suboptimal or redundant. These patterns are replaced with more efficient ones. The idea is to match known inefficient sequences and replace them with better alternatives.
- **Examples of Patterns**: Redundant operations, instruction combinations, or opportunities to eliminate unnecessary computations.

3. Efficiency:

- **Low Cost**: Peephole optimizations are generally low-cost because they only require examining a small portion of the code and applying relatively simple transformations. As a result, these optimizations are fast and do not require a significant amount of additional computation.
- **Incremental Improvement**: Although each individual peephole optimization may result in only a small improvement, when applied across many sections of the program, they can lead to noticeable performance gains.

4. Reduced Code Size or Improved Performance:

- **Code Size Reduction**: Peephole optimization can sometimes reduce the size of the generated code by eliminating unnecessary instructions.
- **Performance Gains**: By replacing inefficient instruction sequences with more optimized ones, the overall performance of the program can improve. This includes reducing the number of instructions executed or optimizing resource usage (such as memory or CPU cycles).

Below are specific techniques that are often applied during peephole optimization:

1. Flow of Code:

- **Description**: This technique aims to optimize the control flow of a program, ensuring that the flow of execution is efficient and does not involve unnecessary operations.

- **Example**: In the case of conditional jumps or unnecessary instructions, the flow of code can be optimized by removing redundant checks or jump instructions.

Example:

```
IF (A == B) GOTO L1
IF (A == B) GOTO L1
```

After optimization:

```
IF (A == B) GOTO L1
```

- **Explanation**: The second check is redundant and can be removed, simplifying the flow of control.

2. Unreachable Code:

- **Description**: Unreachable code is code that cannot be executed because the program will never reach it, often due to a previous jump or a logic error. Removing such code reduces the program size and eliminates unnecessary operations.
- **Example**: Code that is after a return statement or an unconditional jump is unreachable and should be eliminated.

Example:

```
IF (A == B) GOTO L1
PRINT("Hello")
L1: RETURN
```

After optimization:

```
IF (A == B) GOTO L1
L1: RETURN
```

- **Explanation**: The instruction `PRINT("Hello")` is unreachable because the flow of control always jumps to `L1` before this line is reached.

3. Copy Folding:

- **Description**: This technique aims to eliminate unnecessary assignments where a value is copied from one register to another without any modification. Such copies can be replaced by directly using the original value.
- **Example**: If a value is copied from one register to another and no further modifications occur, the copy can be eliminated.

Example:

```
MOV R1, R2   ; Copy R2 to R1
ADD R3, R1, R4
```

After optimization:

```
ADD R3, R2, R4   ; Directly use R2 instead of copying it to R1
```

- **Explanation**: The `MOV R1, R2` is redundant because `R1` is immediately used in the `ADD` instruction without any change.

4. Arithmetic Substitution:

- **Description**: This optimization identifies opportunities where arithmetic operations can be simplified. For example, multiplication by 1, addition of 0, etc., can be eliminated to simplify the code.
- **Example**: A multiplication operation by 1 or addition of 0 is unnecessary and can be removed or replaced.

Example:

```
MOV R1, 5
MUL R2, R1, 1   ; Multiply by 1 (no effect)
ADD R3, R2, 0   ; Add 0 (no effect)
```

After optimization:

```
MOV R2, R1   ; No need to multiply by 1 or add 0
```

- **Explanation**: The multiplication by 1 and addition of 0 are unnecessary, and the instruction sequence is simplified.

5. Dead Code Elimination:

- **Description**: Dead code refers to code that does not affect the program's output. This typically includes computations whose results are never used. Removing dead code reduces code size and improves performance.
- **Example**: If a variable is calculated but never used, the corresponding code can be safely removed.

Example:

```
MOV R1, 10
MOV R2, 20
ADD R3, R1, R2  ; Result is not used
```

After optimization:

```
MOV R1, 10
MOV R2, 20
```

- **Explanation**: The instruction `ADD R3, R1, R2` is dead code because the value of `R3` is never used in the program.

6. Reduction in Strength:

- **Description**: This technique replaces expensive operations with cheaper ones. It typically involves transforming high-cost operations (such as multiplication or division) into lower-cost operations (such as addition or bitwise shifts).
- **Example**: Multiplying by powers of 2 can be replaced by bit-shifting, which is generally faster.

Example:

```
MUL R1, R2, 8   ; Multiply by 8
```

After optimization:

```
SHL R1, R2, 3   ; Shift left by 3 (equivalent to multiplying by 8)
```

- **Explanation**: Instead of performing multiplication by 8, we can shift the value left by 3 bits, which is a more efficient operation in most architectures.

Q. List the various error recovery strategies for a lexical analysis.

1. Panic Mode Recovery:

- **Description**: Panic mode is a simple but effective error recovery strategy where the lexical analyzer (lexer) discards input characters until it reaches a known synchronization point. This synchronization point could be a semicolon, a newline, or a predefined keyword, which signifies the end of a statement or a logical block. Once the lexer reaches this point, it can continue lexical analysis from that position.
- **Purpose**: Panic mode allows the lexer to skip over large portions of the code, ignoring potentially malformed sections, so that it can continue analyzing valid portions of the code.
- **Advantages**: Simple to implement and ensures that the parser does not get stuck due to a single error.
- **Disadvantages**: It can skip over large portions of code, making error recovery less precise and potentially missing important context.
- **Example**: Consider the code:

```
int x = 5;
```

- `int y = 10`
- `int z = x + y;`

`In this example, there is a missing semicolon after int y = 10. When the lexer encounters this error, it may panic and discard characters until it finds a semicolon (;), like this:`

  - The lexer would discard everything between `int y = 10` and `int z = x + y;`.
  - Once the lexer encounters `int z = x + y;`, it synchronizes and continues lexical analysis from there.

2. Phrase-Level Recovery:

- **Description**: Phrase-level recovery (also known as syntactic or local recovery) focuses on trying to recover from lexical errors by correcting the phrase or token at the level of individual statements or expressions. Instead of discarding input, the lexer tries to modify the erroneous token to make it valid, and then continue analyzing.

- **Purpose**: This strategy attempts to fix small lexical errors in place, often by inserting or deleting characters. It works under the assumption that errors are isolated to a single token or a small section of code.
- **Advantages**: More precise than panic mode, as it doesn't discard large portions of the input and tries to correct the immediate error.
- **Disadvantages**: Less robust if errors propagate or if the recovery attempt makes a wrong assumption about the nature of the error.
- **Example**: Consider the code:

```
int x = 10
```

In this case, the lexer encounters an error due to the missing semicolon. Using phrase-level recovery, the lexer might:

  - Insert a semicolon (`;`) after the `10` to recover the error.
  - The lexer would then continue analyzing the rest of the code, which could be:

```
int x = 10;
```

3. Error Productions:

- **Description**: Error productions are special rules in the grammar that allow the lexical analyzer to handle errors by generating an explicit error token or production. These error productions are designed to match invalid tokens or sequences, and when such a sequence is detected, the lexer can produce an error token and continue the analysis.
- **Purpose**: This strategy allows the lexer to recognize specific types of errors, provide a meaningful error token, and ensure that the parser can handle the error gracefully.
- **Advantages**: Error productions can catch specific error cases and help the compiler give more meaningful error messages.
- **Disadvantages**: It requires careful design of the grammar, and there might still be cases where the error cannot be detected or recovered from.
- **Example**: Imagine a situation where a variable name is expected, but a number is encountered:

```
int x = 10;
```

- `x = 20;`

If the lexer encounters a number where an identifier is expected (e.g., int x = 10; 20 = x;), the lexer might use an error production to produce an error token. In the grammar, an error production rule might look like this:

<expr> → ERROR

The lexer would generate an ERROR token, which the parser could then handle accordingly, possibly reporting an error like "invalid assignment."

4. Global Correction:

- **Description**: Global correction involves making changes to the source code as a whole in order to correct multiple errors at once. This strategy focuses on finding a minimal set of changes that could resolve multiple errors and allow the program to continue its analysis. Global corrections are generally more complex and might involve analyzing the source code more holistically.
- **Purpose**: The aim is to make broad corrections that address multiple errors simultaneously. This may include inserting missing keywords, fixing mismatched parentheses, or correcting multiple tokens across the code.
- **Advantages**: It allows for fixing multiple errors in a single pass and can be more effective for complex error scenarios.
- **Disadvantages**: More computationally expensive and may require more advanced analysis and heuristics to determine the correct fixes.
- **Example**: Consider the following code snippet with multiple issues:

```
int x = 5 + ;
```

- `y = x * 10;`
    - In this case, there's a missing operand after + and also an undeclared variable `y`.
    - Using global correction, the lexer or parser might correct these issues by inserting a valid operand after + and declaring y

```
int x = 5 + 3;  // Fixed the missing operand
```

- `int y = x * 10; // Declared 'y' before using it`

Grammar Example:

```
E → E + T | T
T → T * F | F
F → (E) | id
```

Here, `E` represents an expression, `T` represents a term, and `F` represents a factor. The non-terminal `id` represents an identifier, and parentheses are used for grouping.

We will now illustrate how each error recovery strategy can be applied during lexical or syntactic analysis for errors in this grammar.

1. Panic Mode Recovery

Description: The idea behind Panic Mode recovery is that when an error occurs, the parser discards tokens until it finds a known synchronization point, which is usually a symbol that the grammar allows (like a semicolon or parentheses).

Example with Grammar:

Consider the input:

```
id + * id
```

In this case, there's an error after the `+` because the grammar expects a term (`T`) after `+`, but it encounters `*`, which is not valid in this context.

- **Panic Mode Recovery**: When the parser encounters the invalid `*` after `+`, it discards tokens until it reaches a valid synchronization point, such as `id` or `)`, or the end of the statement. So the parser might discard `*` and continue parsing from `id`.
  Steps:

  1. After encountering the `*`, the parser "panics."
  2. It discards the `*` and tries to continue parsing from the next valid token (`id` in this case).
- **Recovered Input**:

  ```
  id + id
  ```

- Now, the parser can successfully apply the rule `E → E + T` and continue parsing.

2. Phrase-Level Recovery

Description: In phrase-level recovery, the parser attempts to correct the current phrase, or token, and continues parsing. This strategy often involves inserting or deleting tokens to restore syntactic correctness.

Example with Grammar:

Consider the input:

`id + id *`

Here, there's an error after `id + id` because the grammar expects a `T` after the `+`, and the `*` does not fit as a valid continuation.

- **Phrase-Level Recovery**: The parser detects that `id + id` is valid as an expression but then encounters a `*`, which is misplaced. It could insert a missing `T` to continue parsing.
  Steps:

  3. The parser encounters the invalid `*` token after `id + id`.
  4. It inserts a missing `T` after `id + id` to correct the phrase.
- **Recovered Input**:

  `id + id * id`

  1. Now, the parser can successfully apply the rule `T → T * F` for `id * id` and continue parsing.

3. Error Productions

Description: In error productions, the grammar explicitly includes rules for handling errors. These error rules allow the parser to generate an "error" token when it encounters a syntactic issue and continue parsing.

Example with Grammar:

Consider the input:

`id + ( id id )`

Here, there's an error in the parentheses: `id id` is not a valid sequence, as the grammar expects an operator like `+` or `*` between the identifiers.

- **Error Productions**: We can introduce an error production to handle such situations. For example, we could have a production like this:

  `E → ERROR`

- When the parser encounters `id id` inside the parentheses, it will trigger the `ERROR` token, indicating that this sequence is not valid.
- The parser would then backtrack to recover from the error by attempting other valid alternatives.

- **Steps**:
  - The parser encounters `id id`, which does not match the expected grammar.
  - The error production rule is invoked to generate an `ERROR` token.
  - The parser continues by handling the error token appropriately.
- **Recovered Input**:

```
id + ( ERROR )
```

The parser might proceed by attempting to continue parsing after `ERROR`, such as by closing the parentheses and continuing with the rest of the input.

4. Global Correction

Description: Global correction involves making changes to the input program to fix errors across a larger scope, such as inserting missing operators, parentheses, or other necessary constructs.

Example with Grammar:

Consider the input:

```
id + ( id * id
```

In this case, there's a missing closing parenthesis for the expression `( id * id`.

- **Global Correction**: The parser might recognize the missing parenthesis as a global error and insert the missing closing parenthesis to correct the input.
  Steps:

  2. The parser reaches the end of the input and detects that a parenthesis is unbalanced (i.e., there's no closing `)` for the opening `(`).
  3. It inserts the missing closing parenthesis `)` to correct the input.
- **Recovered Input**:

```
id + ( id * id )
```

1. Now, the parser can successfully apply the rule `E → E + T` and continue parsing.

Q. Discuss how does register allocation and evaluation order play an important role in a target code generation

Register Allocation and Evaluation Order play crucial roles in **target code generation** in the context of a compiler. These two factors influence the efficiency and correctness of the final generated machine code. Let's break down their importance in detail.

1. Register Allocation

Register allocation refers to the process of assigning variables to machine registers during the code generation phase. Since registers are much faster to access than memory, an efficient allocation of registers can significantly improve the performance of the generated code. However, there are typically fewer registers available than the number of variables used in the program, which leads to the need for careful allocation strategies.

Importance in Target Code Generation:

1. **Performance Optimization**:
    - Registers are the fastest form of storage, so using them to store frequently accessed variables can reduce memory access times.
    - Efficient register allocation helps in minimizing memory loads and stores (which are slower than register accesses), thus improving the overall performance of the generated code.
2. **Minimizing Spills**:
    - When there are more live variables than available registers, some variables must be spilled to memory. Spilling refers to saving the values of registers to memory and reloading them when needed.
    - Spilling introduces overhead because memory access is slower than register access. Effective register allocation tries to minimize spills and favors keeping frequently used variables in registers.
3. **Reducing Instruction Count**:
    - By using registers efficiently, fewer instructions are needed to move data in and out of memory. This directly reduces the number of instructions in the target code, improving execution speed.
4. **Interference Graphs**:
    - The process of register allocation often involves creating **interference graphs**, which model which variables interfere with each other (i.e., cannot share the same register because they are live at the same time).
    - A register allocator will try to assign registers to variables in a way that minimizes conflicts in the interference graph, leading to better register usage.
5. **Live-Range Splitting**:
    - For variables with long live ranges (i.e., they are used throughout a function or program), the allocator might split their live range into separate periods where they can be assigned to different registers.
    - This allows for more efficient register usage by assigning registers dynamically when the variables are active.

Example:

Consider the expression:

```
a = b + c * d;
```

In an assembly-like language, a naïve approach might require several memory accesses like:

```
LOAD R1, b      ; load b into register R1
LOAD R2, c      ; load c into register R2
MUL R2, d       ; multiply c and d, store result in R2
ADD R1, R2      ; add b and the result of c * d
STORE a, R1     ; store the result into a memory location for
a
```

However, an optimized register allocation might try to use fewer memory accesses:

```
MOV R1, b       ; move b to register R1
MOV R2, c       ; move c to register R2
MUL R2, d       ; multiply c and d in R2
ADD R1, R2      ; add b and the result in R2, store in R1
MOV a, R1       ; store the result in a
```

Here, registers `R1` and `R2` are efficiently used, and memory access is reduced.

2. Evaluation Order

Evaluation order refers to the sequence in which operands in an expression are evaluated. This has a significant impact on both the performance and correctness of the generated code, especially when optimizing for specific machine architectures.

Importance in Target Code Generation:

1. **Avoiding Unnecessary Computations**:
    - **Expression reordering** (based on commutative and associative properties) can avoid redundant computations. For instance, for the expression `a + (b + c)`, reordering it as `(a + b) + c` might be beneficial depending on the target architecture and register allocation.
2. **Dependency and Data Flow**:
    - The order of evaluation in an expression affects the dependency between operands. For instance, if `a = b + c * d`, the order of evaluating `b + c * d` matters. The order affects which intermediate results are computed first and whether temporary values must be stored.
    - If `c * d` is computed first, then `b` can be added to that result in one step, reducing the number of intermediate values that need to be stored in memory.
3. **Minimizing Register Usage**:

- By choosing the correct evaluation order, the compiler can reduce the number of registers used for temporary results. For example, by evaluating the operands of a multiplication before performing an addition, temporary values can be reused without spilling to memory.

4. **Precedence and Associativity**:
   - Some expressions can be evaluated in different orders based on their precedence or associativity, and the best choice of order depends on the machine's architecture.
   - For instance, in an expression `a * (b + c)`, it may be beneficial to compute `b + c` first if multiplication is costly, as it allows for better register reuse.
5. **Instruction Scheduling**:
   - The order of evaluation can affect how instructions are scheduled on the target machine. If multiple expressions can be evaluated in parallel or if certain operations (like addition) are faster than others (like multiplication), the evaluation order can impact the efficiency of instruction scheduling.
   - For example, if `a = b + c * d` is evaluated as `a = (b + c) * d`, we might have to compute `b + c` first and then multiply by `d`. However, if we change the order of evaluation and compute `c * d` first, it may save time and reduce register usage.

Example:

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

- **Evaluation order 1**:
  2. First, `b + c` is computed.
  3. Then, the result is multiplied by `d`.

This evaluation order requires storing the result of `b + c` in a temporary register or memory location before multiplying by `d`.

- **Evaluation order 2**:
  4. First, `c * d` is computed.
  5. Then, `b` is added to the result of `c * d`.

This order may reduce temporary storage requirements because it may reuse registers more efficiently and avoid an extra intermediate value.

In **target code generation**, evaluating expressions in the optimal order can reduce the number of required instructions and improve the overall execution time.