**1. What is code generation? Explain basic blocks and flow graphs?**

Code generation is the process of translating the intermediate representation (IR) of a source program into an equivalent sequence of low-level machine instructions or another target language that can be executed on a specific hardware platform. This process involves transforming the abstract representation of the program into concrete instructions that can be understood and executed by the target machine.

* **Intermediate Representation (IR):** Before code generation, the compiler typically performs lexical analysis, parsing, semantic analysis, and optimization phases to convert the source code into an intermediate representation (IR). The IR is a higher-level, platform-independent representation of the program that retains the essential structure and semantics of the source code.
* **Target Machine:** The target machine refers to the specific hardware platform or execution environment for which the compiler is generating code. This could be a particular CPU architecture, a virtual machine, or another execution environment with its own instruction set and memory model.
* **Translation Process:** During code generation, the compiler translates the IR into machine code or another target language suitable for the target machine. This translation process involves several key steps:

1. Instruction Selection
2. Register Allocation
3. Addressing Modes
4. Control Flow
5. Optimization

* **Output:** The output of the code generation phase is the generated machine code or target language code, which is typically represented as a sequence of binary instructions or text-based representations suitable for execution on the target machine. This generated code can then be further processed, assembled, linked, and executed to produce the final executable program.

**Basic blocks:**

A basic block is a simple combination of statements.

Except for entry and exit, the basic blocks do not have any branches like in and out. It means that the flow of control enters at the beginning and it always leaves at the end without any halt.

The execution of a set of instructions of a basic block always takes place in the form of a sequence.

The first step is to divide a group of three-address codes into the basic block. The new basic block always begins with the first instruction and continues to add instructions until it reaches a jump or a label. If no jumps or labels are identified, the control will flow from one instruction to the next in sequential order.

**The algorithm for the construction of the basic block is described below step by step:**

1. **Algorithm:** The algorithm used here is partitioning the three-address code into basic blocks.
2. **Input:** A sequence of three-address codes will be the input for the basic blocks.
3. **Output:** A list of basic blocks with each three address statements, in exactly one block, is considered as the output.
4. **Method:** We’ll start by identifying the intermediate code’s leaders. The following are some guidelines for identifying leaders:

* The first instruction in the intermediate code is generally considered as a leader.
* The instructions that target a conditional or unconditional jump statement can be considered as a leader.
* Any instructions that are just after a conditional or unconditional jump statement can be considered as a leader.
* Each leader’s basic block will contain all of the instructions from the leader until the instruction right before the following leader’s start.

**Example of basic block:**

Consider the following source code for dot product of two vectors a and b of length 10:

begin

prod :=0;

i:=1;

do begin

prod :=prod+ a[i] \* b[i];

i :=i+1;

end

while i <= 10

end

The three address code for the above source program is given below:

B1

(1) prod := 0

(2) i := 1

B2

(3) t1 := 4\* i

(4) t2 := a[t1]

(5) t3 := 4\* i

(6) t4 := b[t3]

(7) t5 := t2\*t4

(8) t6 := prod+t5

(9) prod := t6

(10) t7 := i+1

(11) i := t7

(12) if i<=10 goto (3)

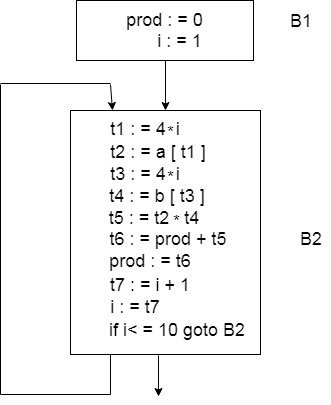
Basic block B1 contains the statement (1) to (2)

Basic block B2 contains the statement (3) to (12)

**Flow Graph**

* Flow graph is a directed graph.
* It contains the flow of control information for the set of basic block.
* A control flow graph is used to depict that how the program control is being parsed among the blocks.
* It is useful in the loop optimization.

Flow graph for the vector dot product is given as follows:



* Block B1 is the initial node. Block B2 immediately follows B1, so from B2 to B1 there is an edge.
* The target of jump from last statement of B1 is the first statement B2, so from B1 to B2 there is an edge.
* B2 is a successor of B1 and B1 is the predecessor of B2.

**2. Explain three address codes with example. 363 388**

Three-address codes are an intermediate representation of code that simplifies the process of converting high-level programming languages into machine code. They are designed to have at most three operands per instruction, making them easier to translate into assembly language.

* Three-address code is an intermediate code. It is used by the optimizing compilers.
* In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
* Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

**Three-Address Instructions**

Three-address code is a sequence of instructions of the form

**x = y op z**

Here,

* x, y and z are the operands.
* Operands may be constants, names, or compiler generated temporaries.
* op represents the operator.

**Common Three Address Instruction Forms-**

The common forms of Three Address instructions are-

**1. Assignment Statement-**

x = y op z and x = op y

Here,

* x, y and z are the operands.
* op represents the operator.

It assigns the result obtained after solving the right side expression of the assignment operator to the left side operand.

**2. Copy Statement-**

x = y

Here,

* x and y are the operands.
* = is an assignment operator.

It copies and assigns the value of operand y to operand x.

**3. Conditional Jump-**

If x relop y goto X

Here,

* x & y are the operands.
* X is the tag or label of the target statement.
* relop is a relational operator.

If the condition “x relop y” gets satisfied, then-

* The control is sent directly to the location specified by label X.
* All the statements in between are skipped.

If the condition “x relop y” fails, then-

* The control is not sent to the location specified by label X.
* The next statement appearing in the usual sequence is executed.

**4. Unconditional Jump-**

goto X

Here, X is the tag or label of the target statement.

On executing the statement,

* The control is sent directly to the location specified by label X.
* All the statements in between are skipped.

**5. Procedure Call-**

param x call p return y

Here, p is a function which takes x as a parameter and returns y.

**Three Address Code is Used in Compiler Applications**

* **Optimization:** Three address code is often used as an intermediate representation of code during optimization phases of the compilation process. The three address code allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.
* **Code generation:** Three address code can also be used as an intermediate representation of code during the code generation phase of the compilation process. The three address code allows the compiler to generate code that is specific to the target platform, while also ensuring that the generated code is correct and efficient.
* **Debugging:** Three address code can be helpful in debugging the code generated by the compiler. Since three address code is a low-level language, it is often easier to read and understand than the final generated code. Developers can use the three address code to trace the execution of the program and identify errors or issues that may be present.
* **Language translation:** Three address code can also be used to translate code from one programming language to another. By translating code to a common intermediate representation, it becomes easier to translate the code to multiple target languages.

**Implementation of Three Address Code**

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

**1. Quadruple** – It is a structure which consists of 4 fields namely op, arg1, arg2 and result.

In quadruples representation, each instruction is splitted into the following 4 different fields-

**op, arg1, arg2, result**

Here-

* The op field is used for storing the internal code of the operator.
* The arg1 and arg2 fields are used for storing the two operands used.
* The result field is used for storing the result of the expression.

**Advantage –**

* Easy to rearrange code for global optimization.
* One can quickly access value of temporary variables using symbol table.

**Disadvantage –**

* Contain lot of temporaries.
* Temporary variable creation increases time and space complexity.

**2. Triples –** This representation doesn’t make use of extra temporary variable to represent a single operation instead when a reference to another triple’s value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.

**Advantages -**

* More compact than quadruples, requiring less space.
* Can potentially improve code optimization by implicit temporary variables.

**Disadvantage –**

* Temporaries are implicit and difficult to rearrange code.
* It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

**3. Indirect Triples –** This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

* This representation is an enhancement over triples representation.
* It uses an additional instruction array to list the pointers to the triples in the desired order.
* Thus, instead of position, pointers are used to store the results.
* It allows the optimizers to easily re-position the sub-expression for producing the optimized code.

**Advantages:**

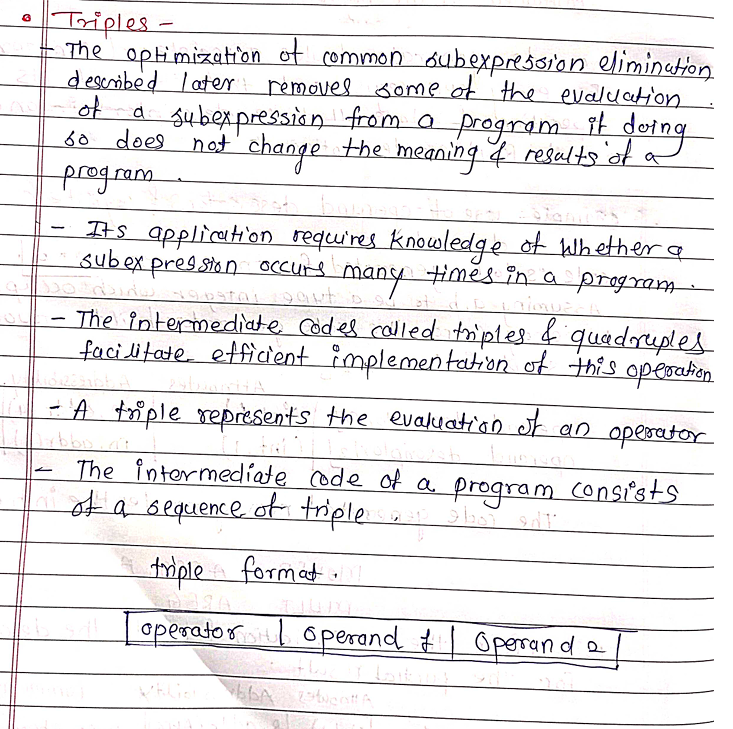
* Combines the benefits of both quadruples and triples: space efficiency from triples and explicit temporary variables for optimization.

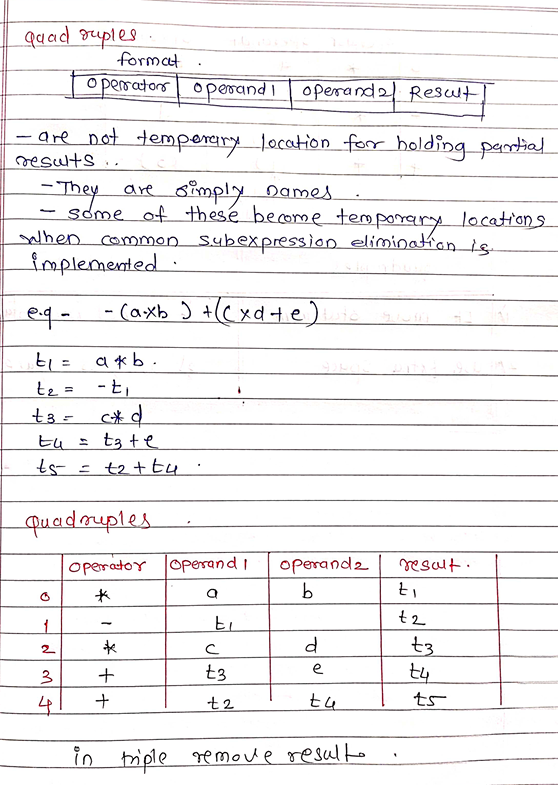
**Disadvantages:**

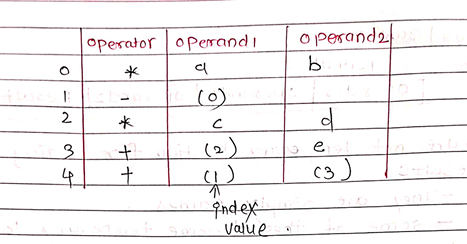
* Increased complexity in implementation due to the separate list of computations.

**3. Explain quadruples and triples with example.**

*(Refer the above answer)*







**4. What are issues in design of a code generator? 505**

**Design Issues**

In the code generation phase, various issues can arise:

1. Input to the code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

**1. Input to the code generator**

* The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
* Intermediate representation has the several choices:

a) Postfix notation

b) Syntax tree

c) Three address code

* We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
* The code generation phase needs complete error-free intermediate code as an input requires.

OR

* The input to the code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data objects denoted by the names in the intermediate representation.
* Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAGs, etc.
* The code generation phase just proceeds on an assumption that the input is free from all syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

**2. Target program:**

* The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, or assembly language.
* Absolute machine language as output has the advantages that it can be placed in a fixed memory location and can be immediately executed. For example, WATFIV is a compiler that produces the absolute machine code as output.
* Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by a linking loader. But there is added expense of linking and loading.
* Assembly language as output makes the code generation easier. We can generate symbolic instructions and use the macro-facilities of assemblers in generating code. And we need an additional assembly step after code generation.

**3. Memory management**

* During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
* Mapping name in the source program to address of data is co-operating done by the front end and code generator.
* Local variables are stack allocation in the activation record while global variables are in static area.

**4. Instruction selection:**

* Selecting the best instructions will improve the efficiency of the program.
* It includes the instructions that should be complete and uniform.
* Instruction speeds and machine idioms also play a major role when efficiency is considered.
* But if we do not care about the efficiency of the target program then instruction selection is straightforward.
* For example, the respective three-address statements would be translated into the latter code sequence as shown below:

Example:

The Three address code is:

a:= b + c

d:= a + e

Inefficient assembly code is:

MOV b, R0 R0→b

ADD c, R0 R0 c + R0

MOV R0, a a → R0

MOV a, R0 R0→ a

ADD e, R0 R0 → e + R0

MOV R0, d d → R0

**5. Register allocation**

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

* Register allocation: In register allocation, we select the set of variables that will reside in register.
* Register assignment: In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

**For example:**

Consider the following division instruction of the form:

**D x, y**

Where,

x is the dividend even register in even/odd register pair

y is the divisor

Even register is used to hold the reminder.

Old register is used to hold the quotient.

**6. Evaluation order**

* The code generator decides the order in which the instruction will be executed.
* The order of computations affects the efficiency of the target code.
* Among many computational orders, some will require only fewer registers to hold the intermediate results.
* However, picking the best order in the general case is a difficult NP-complete problem.

**5. Explain run time storage management and next use information.525**

Runtime storage management refers to the mechanisms employed by a compiler to allocate and deallocate memory dynamically during the execution of a program generated by the compiler. Runtime storage management is a crucial aspect of compiler design, as it directly impacts the efficiency and correctness of the generated code. Here's a breakdown of runtime storage management in compiler construction:

**Stack Allocation:**

* Local variables and function parameters are typically allocated on the stack.
* The stack is a region of memory that is managed in a last-in, first-out (LIFO) fashion.
* When a function is called, space for its local variables and parameters is allocated on the stack, and when the function returns, that space is deallocated.
* The stack pointer (SP) is used to keep track of the current top of the stack.

**Heap Allocation:**

* Dynamic memory allocation using functions like malloc() (in C) or new (in C++) allocates memory from the heap.
* The heap is a region of memory that is managed dynamically at runtime.
* Memory allocated on the heap persists until explicitly deallocated.
* Heap memory management involves tracking allocated and deallocated memory blocks to prevent memory leaks and fragmentation.

**Garbage Collection:**

* Some programming languages, such as Java and C#, use automatic garbage collection to manage heap-allocated memory.
* Garbage collection automatically identifies and reclaims memory that is no longer in use, reducing the need for manual memory management and helping to prevent memory leaks.
* Garbage collection algorithms vary, but they typically involve identifying and reclaiming memory that is no longer reachable by the program.

**Memory Management Unit (MMU) Interaction:**

* In systems with virtual memory management, the compiler may interact with the Memory Management Unit (MMU) to manage memory allocation and protection.
* The compiler may generate code that interacts with the MMU to allocate and deallocate memory, map virtual addresses to physical addresses, and enforce memory protection policies.
* Optimizations and Analysis:
* Compiler optimizations may include techniques to minimize runtime memory usage, such as stack allocation optimization, dead variable elimination, and memory reuse analysis.
* Runtime memory usage analysis may be performed statically or dynamically to detect potential memory leaks, buffer overflows, and other memory-related issues.

**Next-Use Information:**

Next-use information refers to data that helps the compiler make informed decisions during code generation and optimization. It primarily focuses on predicting the next time a specific variable or data item will be used in the program. This information helps in making decisions such as:

* **Register Allocation:** If a variable is not used after its current use within a certain scope, its register can be reused for another variable to minimize register pressure.
* **Instruction Scheduling:** Instructions can be reordered to overlap the execution of independent instructions, taking into account the next use of variables. This can improve pipeline utilization and reduce execution time.
* **Memory Access Optimization:** In the context of cache optimization, knowledge of the next use of variables can be used to prefetch data into the cache before it is needed, reducing cache misses and improving memory access performance.
* **Loop Optimization:** In loop optimizations like loop unrolling or software pipelining, next use information can help in determining the optimal number of loop iterations or the scheduling of loop operations to maximize performance.

By effectively managing runtime storage and utilizing next use information, compilers can generate more efficient code with better memory utilization and reduced execution time. This contributes to overall performance improvements in software applications.

**6. Explain issues in register allocation.**

Register allocation is a critical optimization phase in compiler construction where the compiler assigns variables to processor registers to minimize memory accesses and improve performance. However, several issues arise during register allocation due to constraints imposed by the target architecture and the complexity of the optimization process. Here are some key issues in register allocation in the context of compiler construction:

**Limited Number of Registers:**

* Most processors have a limited number of general-purpose registers available for allocation.
* The scarcity of registers poses a challenge for the compiler to efficiently allocate registers to variables, especially in programs with many variables or complex control flow.

**Register Pressure:**

* Register pressure refers to the situation where the compiler needs more registers than are available to efficiently map variables.
* High register pressure can lead to register spills, where variables need to be stored in memory temporarily due to insufficient register availability, resulting in slower code execution.

**Interference and Lifetime Analysis:**

* Interference occurs when the same register is required for multiple variables simultaneously or when the lifetime of different variables overlaps.
* Lifetime analysis is the process of determining the lifespan of variables to identify opportunities for register reuse.
* Resolving interference and performing accurate lifetime analysis are crucial for effective register allocation.

**Global vs. Local Allocation:**

* Register allocation can be performed globally across the entire program or locally within individual functions or basic blocks.
* Global register allocation can achieve better register utilization but may be computationally expensive and complex.
* Local register allocation is simpler but may miss optimization opportunities across function boundaries.

**Spilling and Reload Costs:**

* Register spills occur when the compiler cannot allocate sufficient registers and must spill variables to memory.
* Spilling incurs additional memory accesses and instructions to store and reload spilled variables, which can degrade performance.
* Minimizing spilling and optimizing spill code are essential for mitigating the impact of register spills on program performance.

**Instruction Scheduling and Register Usage:**

* Register allocation interacts with instruction scheduling, as the availability of registers affects the scheduling of instructions.
* Optimizing instruction scheduling while considering register usage is crucial for maximizing parallelism and minimizing pipeline stalls.

**Target Architecture Constraints:**

* Different target architectures have specific constraints and limitations on register usage, such as calling conventions, reserved registers, and register renaming capabilities.
* The compiler must adhere to these constraints while performing register allocation to ensure compatibility and optimal performance on the target platform.

**7. Explain code generation form DAGs.**

Code generation from Directed Acyclic Graphs (DAGs) is a process used in compiler construction to translate high-level intermediate representations (IRs) into machine code or assembly language. DAGs are often employed as an intermediate representation because they efficiently capture common subexpressions and dependencies, making them suitable for optimization and code generation. Here's an overview of the process of code generation from DAGs:

1. **Constructing the DAG:** Before code generation can occur, the compiler constructs a DAG representation of the intermediate code, typically after performing optimizations such as common subexpression elimination, constant folding, and dead code elimination. Each node in the DAG represents an operation, and edges represent data dependencies between operations.
2. Traversing the DAG:The compiler traverses the DAG in a topological order, ensuring that nodes are visited in an order that respects their dependencies. This ensures that code generation proceeds in a manner that correctly reflects the data flow of the program.
3. Generating Code for Nodes:As the compiler traverses the DAG, it generates machine code or assembly language instructions for each node in the DAG. The code generation process for each node depends on the type of operation represented by the node and the operands involved. For example:

* Arithmetic Operations: Nodes representing arithmetic operations such as addition, subtraction, multiplication, and division are translated into corresponding machine instructions.
* Memory Access Operations: Nodes representing memory accesses, such as loading from or storing to memory, are translated into load and store instructions with appropriate memory addresses.
* Control Flow Operations: Nodes representing control flow operations, such as conditional branches and loops, are translated into branch instructions that modify the program counter based on runtime conditions.
* Function Calls: Nodes representing function calls are translated into instructions that push parameters onto the stack, transfer control to the called function, and handle return values.

1. Handling Register Allocation:During code generation, the compiler performs register allocation to assign variables to processor registers efficiently. Register allocation aims to minimize the number of memory accesses and reduce register pressure. Techniques such as graph coloring, linear scan, and interference graph analysis are used to allocate registers while avoiding data dependencies and register spills.
2. Optimizing Code Generation:Compiler designers often incorporate optimizations into the code generation process to improve the efficiency and performance of the generated code. These optimizations may include instruction scheduling, loop unrolling, function inlining, and peephole optimization. Optimizations aim to reduce execution time, minimize code size, and improve the overall quality of the generated code.
3. Outputting Machine Code or Assembly Language:Finally, the compiler outputs the generated machine code or assembly language, which can then be assembled or linked to produce the final executable program.

Let's consider a simple example expression and walk through the process of generating code from its corresponding DAG representation:

Example Expression:

Consider the following expression:

(a \* b) + (c \* d) - (a \* b)

**Step 1: Constructing the DAG:**

We construct the DAG for the expression:

+

/ \

\* \*

/ \ / \

a b c d

**Step 2: Traversing the DAG:**

We traverse the DAG in a topological order, visiting nodes in a sequence that respects their dependencies. In this case, we might start from the bottom nodes (a, b, c, d) and work our way up to the root node (+), and then to the root node of the whole expression (-).

**Step 3: Generating Code for Nodes:**

As we traverse the DAG, we generate code for each node based on its operation and operands.

For the multiplication nodes (\*), we generate machine code or assembly instructions to perform the multiplication operation.

For the addition node (+), we generate instructions to add the results of the two multiplications.

For the subtraction node (-), we generate instructions to subtract one of the multiplication results from the sum.

**Step 4: Handling Register Allocation:**

During code generation, we perform register allocation to efficiently use processor registers. We allocate registers to hold the values of variables a, b, c, and d, as well as temporary values for intermediate results.

**Step 5: Optimizing Code Generation:**

We incorporate optimizations into the code generation process to improve the efficiency and performance of the generated code. For example, we might apply common subexpression elimination to avoid redundant computations of (a \* b).

**Step 6: Outputting Machine Code or Assembly Language:**

Finally, we output the generated machine code or assembly language, which might look something like this (assuming a simple hypothetical assembly language):

; Load values of variables a, b, c, and d into registers

LOAD a, R1

LOAD b, R2

LOAD c, R3

LOAD d, R4

; Multiply a and b, store result in R5

MUL R1, R2, R5

; Multiply c and d, store result in R6

MUL R3, R4, R6

; Add the results of the two multiplications, store result in R7

ADD R5, R6, R7

; Subtract the result of the first multiplication from the sum, store result in R8

SUB R7, R5, R8

; Result stored in R8

This assembly code reflects the structure and semantics of the original expression (a \* b) + (c \* d) - (a \* b) and can be further optimized and translated into machine code to produce the final executable program.

**8. Construct DAG (Direct Acyclic Graph) for following Expression ((a\*b)+(a\*b))+((c\*d)+(c\*d))**

1. **Explain in detail Source of optimization.**
2. **Explain in detail peephole of optimization.**

*Q1 and Q2 are there in the pdf previous*

1. **What are basic blocks?**

*Refer it from Q1 from this doc only*

1. **What are loops in flow graphs?**  
   *Refer this:* https://www.brainkart.com/article/Loops-in-Flow-Graphs\_8196/