1. (i) Code motion

* It is also known as loop-invariant code motion, involves moving computations out of loops if they do not depend on loop variables and remain constant across iterations. This optimization reduces redundant computations and improves performance.

Eg:
```
for(int i = 0; i<n; i++){
    int result = a[i] + b[i];
    c[i] = result + 1;
}
```
⇒
```
int result;
for(int i=0; i<n; i++){
    result = a[i] + b[i];
    c[i] = result + 1
}
```

(ii) Induction variable Elimination

* Induction variables are loop variables that are incremented or decremented by a constant amount in each iteration. Induction variable elimination replaces such variables with constants, eliminating the need for updates and reducing loop overhead.

Eg:
```
int sum = 0;
for(int i=1; i<=n; i++){
    sum += i;
}
```
⇒ `int sum = n*(n+1)/2;`

(iii) Strength Reduction

* It replaces expensive operations with cheaper alternative. For example, replacing multiplication with addition or shifting or replacing division with multiplication or bit manipulation.

Eg:
```
for(int i = 0; i<n; i++){
    a[i] = i * 4;
}
```
⇒
```
int value = 0;
for(int i=0; i<n; i++){
    a[i] = value;
    value += 4;
}
```

2. Basic Blocks are straight-line sequences of code with no branches except at the entry and exit points. Optimizing basic blocks involves analyzing and transforming code within these blocks to improve performance, reduce code size, or enhance other desired properties.

1) Identifying Basic Blocks

* The 1st step in optimizing basic blocks is to identify them within the code. This typically involves dividing the code into contiguous sequences of instructions that start with a single entry point and end with single exit point. usually a branch or return statements.

2) Analysis

* Various analyses, such as data flow and control flow analysis, are performed to gather insights into how data and control flow through the block, aiding optimization techniques.

3) Optimization

* Techniques like constant folding, dead code elimination and loop unrolling are applied to improve code efficiency and reduce resource-consumption within each basic blocks.

4) Transformation

* After optimization, transformations may be applied to simplify or restructure blocks while preserving their semantics, optimizing further for performance or code reliability.

5) Validation

* Optimized block undergo testing and verification to ensure that they maintain correctness and do not introduce errors or unintended behaviours into the program.

3. Three address code

$t_1 := A - B$
$t_2 := C * D$
$t_3 := t_1 + t_2$
$t_4 := t_3 - E$
$S := t_4 + F$

Quadruples

	Op	arg1	arg2	result
(0)	-	A	B	t_1
(1)	*	C	D	t_2
(2)	+	t_1	t_2	t_3
(3)	-	t_3	E	t_4
(4)	+	t_4	F	S

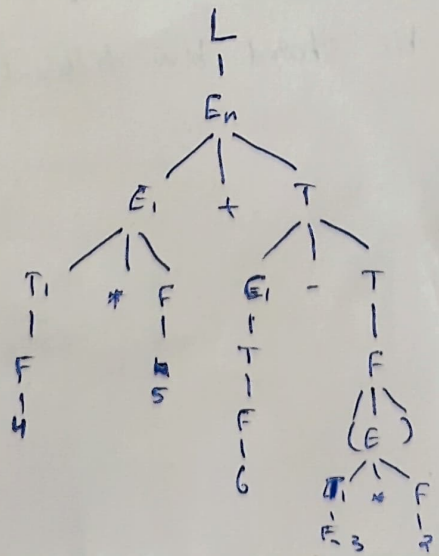
Machine code

```
LOAD  A, R1
LOAD  B, R2
SUB   R1, R2, R3
LOAD  C, R4
LOAD  D, R5
MUL   R4, R5, R6
ADD   R3, R6, R7
LOAD  E, R8
SUB   R7, R8, R9
LOAD  F, R10
ADD   R9, R10, S
```

4. SDD of Desk Calculator

Production	Semantic Rule
$L \rightarrow E_n$	Print (E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow E_1 - T$	
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
F → digit	F.val := digit.lexval

annotated parse tree for 4*5+6-(3*2)



5. Static Allocation and Heap allocation are two common memory allocation strategies used in programming. (4)

Static Allocation

* In static allocation, memory is allocated at compile time, and the size of memory needed is known beforehand. This memory allocation is fixed and remains constant throughout the program's execution. Static allocation is used for:

1) Global Variables: Variables declared outside of any function are allocated statically. They have a fixed memory location throughout the program's execution.

2) Static variables: Variables declared with the static keyword inside a function are allocated statically. They retain their values b/w function calls and have a fixed memory location.

3) Constants: Memory for constants, such as string literals, is allocated statically.

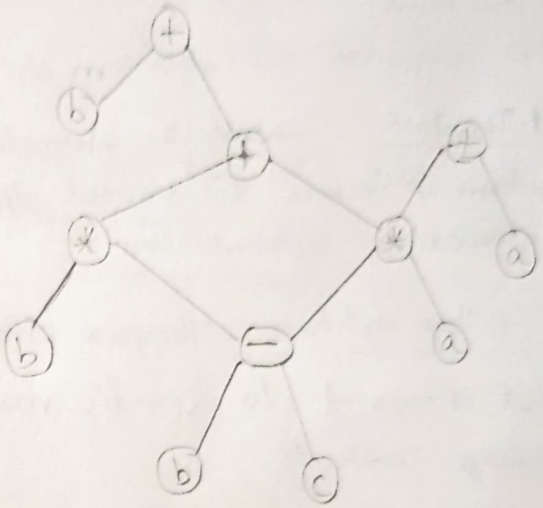
Heap Allocation

* Also known as dynamic memory allocation, involves allocating memory during program execution from a region of memory called the heap. Memory allocated on the heap can be resized and deallocated during runtime. Heap allocation is used when:-

1) The size of memory needed is not known at compile time.
2) When memory needs to be shared b/w different parts of the program.

6. Three address code

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = b * t_1$
$t_4 = a + t_2$
$t_5 = t_4 + t_3$
$t_6 = t_5 + b$



7. Original 3 address code

$t_1 = a + b$
$x = t_1$
$t_2 = a + b$
$t_3 = t_2 + c$
$b = t_2$
$t_4 = a + b$
$y = t_4$

After elimination

$t_1 = a + b$
$x = t_1$
~~$t_2 = t_1$~~
$t_3 = t_1 + c$
$b = t_1$
~~$t_4 = t_1$~~
~~$y = t_4$~~ $y = t_1$

Quadruple form

	Op	arg1	arg2	Result
(0)	+	a	b	t_1
(1)	assign	t_1		x
	~~assign~~	~~t_1~~		~~t_2~~
(2)	+	t_1	c	t_3
(3)	assign	t_1		b
(4)	assign	t_1		y

8. Code generation Algorithm is a process used by compilers to translate intermediate representations into executable machine code or assembly language instructions.

The steps in the algorithm are:

1. Traversal: Traverse the intermediate representation in a top-down or bottom up manner. This traversal depends on the structure of the intermediate representation.

2. Pattern matching: Recognize patterns in the intermediate representation that correspond to specific machine instructions or assembly language constructs.

3. Instruction Selection: Select appropriate machine instructions or assembly language constructs based on the recognized patterns. This involves mapping high-level language constructs to low-level machine operations.

4. operand Selection: Determine the operands for each instruction. This involves mapping high-level language variables, constants and expressions to memory locations or CPU registers.

5. Code Emission: Emit the selected instructions along with their operands to generate the final machine code or assembly language o/p.

6. Optimization: Optionally, perform optimization techniques to improve the efficiency of the generated code, such as constant folding, common subexpression elimination and register allocation.

getReg Function

* It is a helper function used in code generation to allocate and manage registers for storing intermediate values during compilation.

- Register Pools: maintain a pool of available CPU registers. The size of this pool depends on the target architecture and the number of available registers.

- Allocation: when a new intermediate value needs to be stored in a register, the getreg function is called to allocate a register from the register pool. The register is then used to hold the value temporarily during code generation.

- Register Spilling: If there are no available registers in the pool, the getreg function may perform register spilling, where it temporarily stores the value in memory and then frees up a register for allocation.

9.

three address code

$t_1 := a / b$

$t_2 := c - d$

$t_3 := t_1 * t_2$

$t_4 := t_1 + t_3$

$x := t_4$

machine-code

```
LOAD  a, R1
LOAD  b, R2
DIV   R1, R2, R3
LOAD  c, R4
LOAD  d, R5
SUB   R4, R5, R6
MUL   R3, R6, R7
ADD   R3, R7, R6
STORE R3, x
```

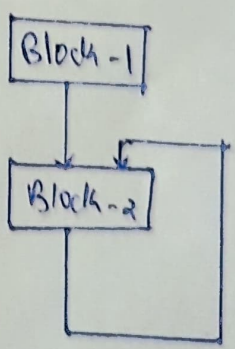
10.

Block-1

```
Prod = 0
i = 1
```

Block-2

```
T1 = 4 * i
T2 = a[T1]
T3 = 4 * i
T4 = b[T3]
T5 = T2 * T4
T6 = T5 * Prod
Prod = T6
T7 = i + 1
i = T7
if (i <= 10) goto B2
```



Flow graph