UNIT-4

There are three parts in this unit, this first part asks 10 marks in exam

4.1: INTERMEDIATE CODE GENERATOR

⊛ Intermediate Code Generation:

The front end translates the source program into intermediate representation from which the backend generates target code. Intermediate codes are machine independent codes, but they are close to machine instructions.

Advantages of using intermediate code representation:

→ If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.

→ Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.

→ The second part of compiler, synthesis, is changed according to the target machine.

→ It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

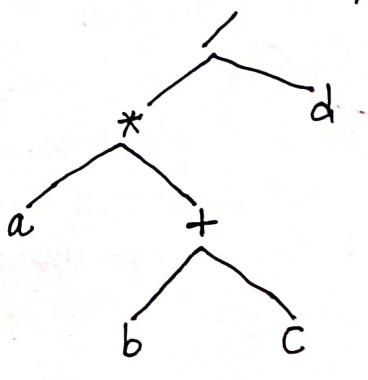
⊛. Intermediate Representations:

1). Graphical Representations:

[Imp]

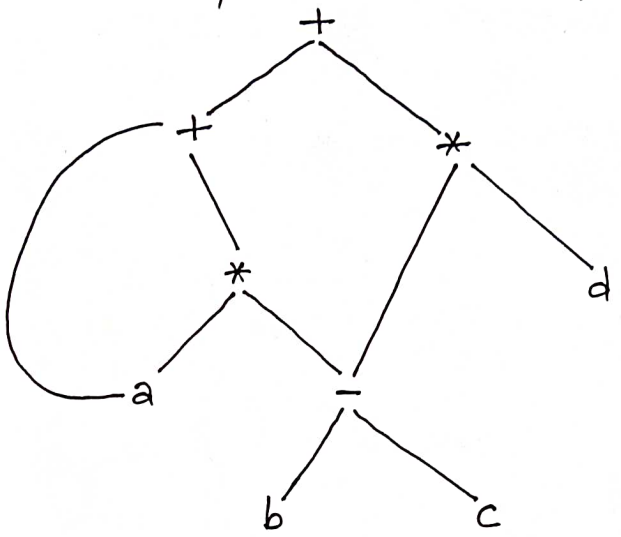
i) Syntax tree: Syntax tree is a graphical representation of given source program and it is also called variant of parse tree. A tree in which each leaf represents an operand and each interior node represents an operater is called syntax tree.

Example: Syntax tree for the expression a*(b+c)/d



1) Directed acyclic graph (DAG): A DAG for an expression identifies the [Imp] common sub expressions in the expression. It is similar to syntax tree, only difference is that a node in a DAG representing a common sub expression has more than one parent, but in syntax tree the common sub expression would be represented as a duplicate sub tree.

Example: DAG for the expression a+a*(b-c)+(b-c)*d.



2) Postfix notation: The representation of an expression in operators followed by operands is called postfix notation of that expression. In general if x and y be any two postfix expressions and OP is a binary operator then the result of applying OP to the x and y in postfix notation by "x y OP".

Examples:

1. (a+b)*c in postfix notation is: ab+c*
2. a*(b+c) in postfix notation is: abc+*

3) Three Address Code: The address code that uses at most three addresses, two for operands and one for result is called three address code. Each instruction in three address code can be described as a 4-tuple: (operator, operand1, operand2, result). A quadruple (Three address code) is of the form:

x=y op z where x, y and z are names, constants or compiler-generated temporaries and op is any operator.

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

Example1: Three address code for expression: (B+A)*(Y-(B+A))

Solution:

```
t1 = B+1
t2 = Y-t1
t3 = t1*t2
```

per line मा maximum 3 variable हुन्छ पर्ने 1,2,3 थपि गए पनि हुन्छ

Maths मा जस्तो जसको precedence बढि हुन्छ त्यसलाई पहिला convert गर्ने here bracket first.

Example2: Three address code for expression:

```
i = 2*n + k
while i do
  i = i-k
```

Solution:

```
t1 = 2
t2 = t1*n
t3 = t2+k
i = t3
L1: if i=0 goto L2
    t4 = i-k
    i = t4
    goto L1
L2: .....
```

This first four lines for i = 2*n+k

We use two variables L1 and L2 when loop comes. L1 for true condition and L2 for false condition. Here instead of i=0 we can also write i=false

यहाँ पनि 3 variable हुन् L1,L2, र i

These three lines are for i=i-k which is body of loop

goes out of body i.e. L2 when i=0

⊛. Implementation of Three-Address Statements:

Three-address statements are implemented or represented by Quadruples and Triples.

Quadruples: A quadruple is a record structure with four fields, [Imp] which are op, arg1, arg2 and result. The op field contains an internal code for the operator. The three-address statement x = y op z is represented by placing y in arg1, z in arg2 and x in result.

	op	arg1	arg2	result
(0)	−	c		t1
(1)	*	b	t1	t2
(2)	−	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

(0), (1), (2), ... numbering nothing else

first do three-address in rough as below then it is easy:

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

Fig: Quadruple representation of three-address statement a = b*-c+b*-c

Triples: A triple is a record structure with three fields, which are [Imp] op, arg1 and arg2. The result field of quadruples is absent here all other things are same, as quadruples. The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure.

	op	arg1	arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

rough: We use numbering here since result field having variable is absent there.

(0) −c
(1) b*(0)
(2) −c
(3) b*(2)
(4) (1)+(3)
(5) a=(4)

Fig: Triple representation of three-address statement a=b*−c+b*−c.

Example: Translate the expression x=(a+b)*(c+d)−(a+b+c)

a. Quadruples
b. Triples.

Solution: Three address code is:

t1=a+b
t2=c+d
t3=t1*t2
t4=t1+c
t5=t3−t4
x=t5

Quadruples:

	op	arg1	arg2	result
(0)	+	a	b	t1
(1)	+	c	d	t2
(2)	*	t1	t2	t3
(3)	+	t1	c	t4
(4)	−	t3	t4	t5
(5)	=	t5		x

Triple:

	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	(0)	c
(4)	−	(2)	(3)
(5)	=	x	(4)

Naming conventions for three address code:

S.code: three-address code for evaluating S.
S.begin: label to start of S.
S.after: label to end of S.
E.code: three-address code for evaluating E
E.place: a name that holds the value of E.

To represent three-address statement.

Gen (E.place '=' E1.place '+' E2.place)

code generation ⇓ t3 = t1+t2

अहिले सम्म हामिले expression लाई three-address code मा लैजाने, अब productions लाई 3-address code दिने।

Syntax-Directed Translation into Three-Address Code

1) Assignment statements:

Productions	Semantic rules
S → id = E	S.place = newtemp(); S.code = E.code || gen(id.place '=' E.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 → -E1	E.place = newtemp(); E.code = E1.code || gen(E.place '=' 'minus' E1.place)
E → (E1)	E.place = newtemp(); E.code = E1.code
E → id	E.place = id.name E.code = null

three address-code variable (ie, terminal को हुँदैन हामी further produce गर्न सकने non-terminal को मात्र निकाल्ने)

newtemp() creates temporary storage for S

|| is concatenation operator here

gen हामि expression को 3-address code निकाल्दा use गर्ने

right side three address निकाल्न मिल्ने non-terminal देख, so newtemp() गर्नु पर्दैन. id मा जे name आउछ त्यहि E को place मा जान्छ! so E.code निकाल्नु पर्दैन so E.code is null.

2) Boolean Expressions: [Imp]

Production	Semantic Rules
$B \rightarrow B_1 \| B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| label(B_1.false) \| B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| label(B_1.true) \| B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$E \rightarrow id_1$ relop id_2	$E.code = gen('if' id_1.place \text{ relop } id_2.place \text{ 'goto' } E.true) \| gen('goto' E.false)$
$B \rightarrow true$	$B.code = gen(goto\ B.true)$
$B \rightarrow false$	$B.code = gen(goto\ B.false)$

production मा || symbol आए logical OR हो/ Semantic rule मा आए मात्र concatenation

logical OR मा कुनै एउटा मात्र true भए पनि true हुन्छ। दुवै false हुँदा मात्र false हुन्छ। So, सुरुमा B_1 true हुँदा B true भयो। But if B_1 false भए अब B true हुन B_2 मा depend गर्छ we do not know B_2 so we give it newlabel(). Now if B_2 is true B is true otherwise false finally we write 3 address code.

logical AND. B true हुन दुवै B_1 र B_2 true हुनु पर्यो

B_1.true हुँदा मा B true हुन्छ भनि decide गर्न सक्नेछैनौं, logical AND मा B true हुन B_1 र B_2 दुवै true हुनु पर्यो। So, B to be true now it depends on value of B_2 so newlabel() is given

relation operator (can be <, >, =, <=, >=, != etc.

3) Flow of control statements: [Imp]

Production	Semantic Rules
$S \rightarrow if(B)\ S_1$	$B.true = newlabel()$ $B.false = S.next$ $S_1.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code$
$S \rightarrow if(B)\ S_1\ else\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S.next$ $S_2.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code \|$ $gen(goto\ S.next) \| label(B.false) \| S_2.code$

$S \rightarrow$ while $(B)\ S_1$

```
begin = newlabel()
B.true = newlabel()
B.false = S.next
S1.next = begin
S.code = label(begin) || B.code || label(B.true) ||
         S1.code || gen(goto begin).
```

Example : Generate three address code for the expression

```
if (x<5 || (x>10 && x==y)) x=3;
```

think of if condition in C program

This is similar to as;
if (x<5 || (x>10 && x==y))
{ x=3 } ;.....
i.e, if x<5 it goes to x=3
OR if x>10 and x==y then also it goes to x=3
otherwise goes out of if block .....

Logical OR

Solution:

```
L1: if x<5 goto L2
    else
        goto L3
L2: x=3 goto L5
L3: if x>10
      goto L4
    else
        goto L5
L4: if x==y
      goto L2
    else
        goto L5
L5: ........
```

4) Switch/case statements: [Imp]

Example: Convert the following switch statement into three address code:

```
Switch (i+j)
{
  Case1: x=y+z
  Case2: u=v+w
  Case3: p=q*w
  Default: s=u/v
}
```

Solution:

3 address code को use करके सीखना बहुत जरूरी है

```
t = i+j
L1: if t==1  goto L2
      else   goto L3
L2: x=y+z    goto L8
L3: if t==2  goto L4
      else   goto L5
L4: u=v+w    goto L8
L5: if t==3  goto L6
      else   goto L7
L6: p=q*w    goto L8
L7: s=u/v    goto L8
L8: ..........
```

5) Addressing array elements:

If the width of each array element is w, the the i^{th} element of array 'A' begins in location,

$$A[i] = base + (i - low) * w$$

where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is base is the relative address of A[low].

Example: Address of 15th element of array is calculated as below,
Suppose base address of array is 100 and type of array is integer of size 4 bytes and lower bound of an array is 10 then,

$$A[i] = base + (i - low) * w$$
$$\text{or, } A[15] = 100 + (15 - 10) * 4$$
$$= 100 + 20$$
$$= 120.$$

⇒ Similary for two dimensional array we have formula,

$$A[i_1, i_2] = base_A + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$$

where low_1, low_2 are the lower bounds on values i_1 and i_2, n_2 is the number of values that i_2 can take.

Example: Let A be a 10×20 array, there are 4 bytes per word, assume $low_1 = low_2 = 1$. Find addressing array elements $A[i_1, i_2]$ and three-address code, for 2D array addressing.

Solution:

$$A[i_1, i_2] = base_A + ((i_1 - 1) * 20 + i_2 - 1) * 4$$
$$= base_A + (20i_1 - 20 + i_2 - 1) * 4$$
$$= base_A + (20i_1 - 21 + i_2) * 4$$
$$= base_A + 80i_1 - 84 + 4i_2$$

($base_A$, i_1 and i_2 not given so we write them as it is)

Now converting into three-address code as follows:

$t_1 = 80$
$t_2 = t_1 * i_1$
$t_3 = 4$
$t_4 = t_3 * i_2$
$t_5 = baseA + t_2$
$t_6 = t_5 - 84$
$t_7 = t_6 + t_4$
$A[i_1, i_2] = t_7$

⊛ Procedure Calls:

```
param x call p return y
```

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

Procedure calls have the form:

```
param x1
param x2
. . . . . .
param xn
call p, n
```

corresponding to the procedure call $p(x1, x2, \ldots, xn)$.

return statement:

They have the form return y, representing a returned value is optional.

⊛ Back patching: The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. While generating code for boolean expressions and flow-of-control statements during one single pass we may not know the labels that control must go at the time the jump statements are generated. Hence a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

4.2 Code Generator

asks for 5 marks in exam normally.

⊛Factors Affecting code generator/code generator design issues: [Imp]

The code generator mainly concern with:

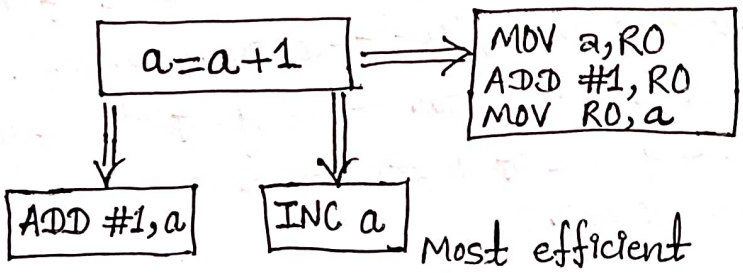
1) Input to the code generator: The input to the code generator is intermediate representation together with the information in the symbol table.

2) The Target Program: The output of the code generator is target code. Typically, the target code comes in three forms such as: absolute machine language, relocatable machine language and assembly language.

The advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

3) The target machine: Implementing code generation requires understanding of the target machine architecture and its instruction set.

4) Instruction Selection: Instruction selection is important to obtain efficient code. Suppose we translate three-address code,



5) Register Allocation: Since registers are the fastest memory in the computer, the ideal solution is to store all values in registers. We must choose which values are in the registers at any given time. Actually this problem has two parts:

i) Which values should be stored in registers?

ii) Which register should each select to store value?

The reason for the second problem is that often there are register requirements, e.g, floating-point values in floating-point registers and certain requirements for even-odd register pairs for multiplication/division.

6) Evaluation order: The order of the target code can be affected by the order in which the computations are performed.

⊛ Basic blocks:

A basic block is a sequence of consecutive instructions in which flow of control enters by one entry point and exit to another point without halt or branching except at the end.

Example:

```
    MOV 1, R0
    ADD n, R0
    MOV 2, R1
    MUL R0, R1
    JMP L2
L1: MUL 2, R0
    SUB 1, R1
L2: MUL 3, R1
    JMPNZ R1, L1
```

⟹

```
MOV 1, R0
ADD n, R0
MOV 2, R1
MUL R0, R1
JMP L2
```

```
L1: MUL 2, R0
    SUB 1, R1
```

```
L2: MUL 3, R1
    JMPNZ R1, L1
```

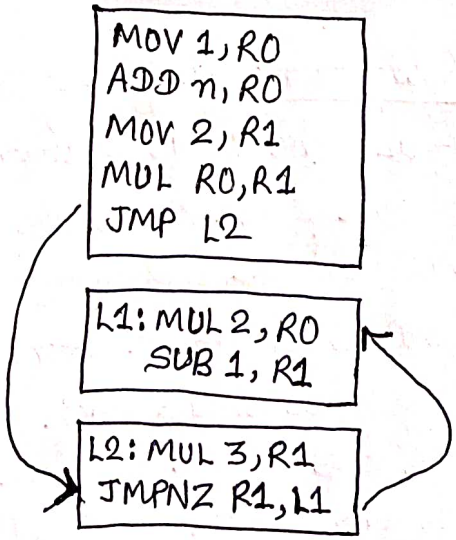
⊛ Flow Graphs:

A flow graph is a graphical representation of a sequence of instructions with control flow edges. A flow graph can be defined at the intermediate code level or target code level. The nodes of flow graphs are the basic blocks ~~of given sequence of instructions then such group of blocks~~ and flow-of-control to immediately follow node connected by directed arrow. Simply, if flow of control occurs in basic blocks of given sequence of instructions then such group of blocks is known as flow graphs.

Example:

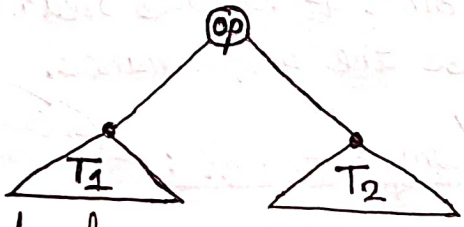
```
    MOV 1, R0
    ADD n, R0
    MOV 2, R1
    MUL R0, R1
    JMP L2
L1: MUL 2, R0
    SUB 1, R1
L2: MUL 3, R1
    JMPNZ R1, L1
```

⟹



⊛ Dynamic programming code-generation algorithm:

The dynamic programming algorithm can be used to generate code for any machine with r interchangable registers $R_0, R_1, \ldots, R_{r-1}$ and load, store, and add instructions. For simplicity, we assume every instruction costs one unit, although the dynamic programming algorithm can easily be modified to work even if each instruction has its own cost.



Contiguous evaluation: Compute the evaluation of T_1, T_2, then evaluate root.
Non-contiguous evaluation: First evaluate part of T_1 leaving the value in a register, next evaluate T_2, then return to evaluate the rest of T_1.

The dynamic programming algorithm uses contiguous evaluation and proceeds in three phases:

1). Compute bottom-up for each node n of the expression tree T, an array C of costs, in which the ith component C[i] is the optimal cost of computing the subtree S rooted at n into a register assuming i registers are available for the computation, for $1 <= i <= r$.

2). Traverse T, using the cost vectors to determine which subtrees of T must be computed in memory.

3). Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

4.3 Code Optimization

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e, CPU, Memory) so that faster-running machine code will result.

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Code optimization defn and need or importance is imp.

⊛. Need of code optimization / why optimize? :

Code optimization is needed because optimization helps to:

→ Reduce the space consumed and increases the speed of compilation.

→ Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.

→ An optimized code often promotes re-usability.

⊛. Criteria of code optimization:

→ The optimization must be correct, it must not, in any way, change the meaning of program.

→ Optimization should increase the speed and performance of the program.

→ The compilation time must be kept reasonable.

→ The optimization process should not delay the overall compiling process.

⊛Basic optimization techniques: [Imp]

Optimizations techniques are classified into two categories:
→Machine Independent optimization techniques
→ Machine dependent optimization techniques.

1) Machine Independent optimization techniques:

Machine independent optimization techniques are program transformations that improve the target code without taking into consideration any properties of the target machine.

a) Constant Folding: As its name suggests, it involves folding the constants. The expressions that contain the operands having constant values at compile time are evaluated. Those expressions are then replaced with their respective results.

Example: Circumference of Circle = (22/7) × Diameter
Here,
→This technique evaluates the expression 22/7 at compile time.
→The expression is then replaced with its result 3.14.
→This saves time at run time.

b) Constant Propagation: In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation. The condition is that the value of variable must not get alter in between.

Example: Pi = 3.14, Radius = 10, Area of circle = Pi × Radius × Radius.
Here,
→This technique substitutes the variables 'pi' and 'radius' at compile time.
→It then evaluates the expression 3.14×10×10
→The expression is then replaced with its ~~resu~~ result 314.
→This saves the time at run time.

c) Redundant Code Elimination: In computer programming, redundant code is source code or compiled code in a computer program that is unnecessary such as: code that is never executed (unreachable code).

Example: x = y+z
a = 2+y+b+z
That is reduced by,
a = 2+x+b.

d) **Variable Propagation:** Variable propagation means use of one variable instead of another.

Example: $x = r$

$A = pi \times x^2$

That is reduced by, $A = pi \times r^2$

e) **Strength reduction:** It involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example:

Code before optimization	Code after optimization
B = A × 2	B = A + A

→ This is because the cost of multiplication operator is higher than that of addition operator.

f) **Loop Optimization:** ← Defn: Loop optimization is the process of increasing execution speed and reducing the overheads associated with loops. Following are loop optimization techniques.

i) **Code Motion / Frequency Reduction:** It is a technique which moves the code outside the loop.

Example:
```
while (i<5000)
    x = i*sin(A)*cos(A);
```

This can be optimized as;
```
t = sin(A)*cos(A);
while (i<5000)
    x = i*t;
```

ii) **Loop jamming / Loop Fusion:** In loop fusion method several loops are merged to one loop.

Example:
```
for i=1 to n do
    for j=1 to m do
        a[i,j]=10.
```

It can be written as;
```
for i=1 to n*m do
    a[i]=10
```

iii) **Loop Unrolling:** The number of jumps and tests can be reduced by writing code two times.

Example:
```
int i=1;
while (i<=100)
{ a[i]=b[i];
  i++;
}
```

It can be written as;
```
int i=1;
while (i<=100)
{ a[i]=b[i];
  i++;
  a[i]=b[i];
  i++;
}
```

2) Machine Dependent Optimization Techniques:

Machine dependent optimization techniques are based on register allocation and utilization of special machine-instruction sequences. It is done after the target code has been generated and when the code is transformed according to the target machine architecture.

Peephole Optimization: Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve performance of the target program by examining the short sequence of target instructions (called the peephole) and replace these instructions replacing by shorter or faster sequence whenever possible. Peephole is a small, moving window on the target program. Peephole optimization techniques are as follows:

a) Redundant Load and Store Elimination: In this technique the redundancy is eliminated.

Example:

Initial code:
```
y=x+5;
i=y;
z=i;
w=z*3;
```

Optimized code
```
y=x+5;
i=y;
w=y*3;
```

b) The Flow of Control Optimization:

i) Dead Code Elimination: It involves eliminating the dead code i.e, statements of code which never executes or are unreachable.

Example:

Code before optimization
```
i=0;
if(i==1)
{
  a=x+5;
}
```

Code after optimization
```
i=0;
```

ii) Avoid jump on jump: The unnecessary jumps can eliminate in either the intermediate code or the target code.

We can replace the jump sequence:
```
Goto L1
.....
L1: goto L2
```
By the sequence:
```
Goto L2
.....
L1: goto L2.
```

c) Algebraic Simplification: Peephole optimization is an effective technique for algebraic simplification. The statements such as $x = x+0$ or $x = x*1$ can be eliminated by peephole optimization.

d) Machine Idioms: The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.

Example: Some machines have auto-increment or auto-decrement addressing modes.

e) Strength reduction: It involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example: Code before optimization:

$B = A \times 2$

Code after optimization:

$B = A + A$

Compiler Design and Construction (Marking Scheme) with Imp Topics:

Unit1: (3 hrs) : 5 marks (Analysis phase, Synthesis phase, compiler vs interpreter, one pass vs. Multipass compiler)

Unit2: (22 hrs):

- **2.1 Lexical Analysis:** 10 marks (lexemes, patterns, tokens, thomsons construction, subset construction)
- **2.2 Syntax Analysis:** 20 marks (role of syntax analyzer, Left recursion, left factoring, LL(1) parsing table, handle, LR(0) item, cannonical LR(0) collection, SLR parsing tables, cannonical LR(1) collection, LR(1), LALR(1), Top-down vs bottom-up parsing)
- **2.3 Semantic Analysis:** 10 marks (Type checking of expressions, Static vs. dynamic type checking, STD, STDS)

Unit3: (4 hrs):

- Symbol Table Design: 5 marks
- Run-time storage management: 5 marks

(very short unit, everything imp, read all)

Unit4: (16 hrs):

- **4.1 Immediate Code Generator:** 10 marks (Role of immediate code generator, Syntax tree, DAG, Postfix notation, finding 3 address code of expressions and statements)
- **4.2 Code Generator:** 5 marks (code generator design issues)
- **4.3 Code Optimization:** 5 marks (defn of code optimization with its need, basic optimization techniques)