UNIT-5
Machine Dependent Phases

Syllabus:

UNIT-V: Machine Dependent Phases

Intermediate Code Generation: Intermediate code, three address code, quadruples, triples, directed acyclic graph.

Code Optimization: Common sub expression elimination, copy propagation, dead code elimination, constant folding, strength reduction, loop optimization.

Code Generation: Basic blocks & flow graphs, Peephole optimization, Register allocation and assignment.

Intermediate Code Generation

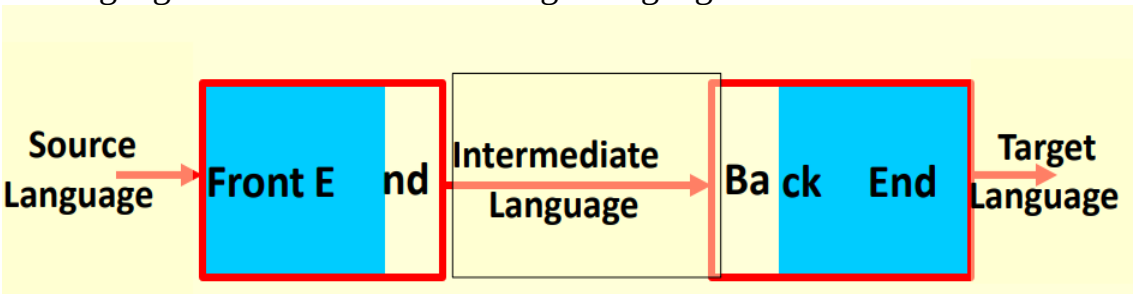
→ An internal form of a program created by the compiler while translating the program from a HLL to an assembly code or object code.

→ Intermediate code is more attractive form of target code than does the assembly or machine code.

→ It is intermediate in complexity between a HLL and MLL

→ In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an intermediate code, and then the back end of the compiler uses this intermediate code to generate the target code.

→ An language between source and target language



→ Provides an intermediate level of abstraction

- More details than the source
- Fewer details than the target

→ The intermediate code must be:

- Easy to produce.
- Easy to translate into target code.

Benefits of intermediate code generation

→ A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.

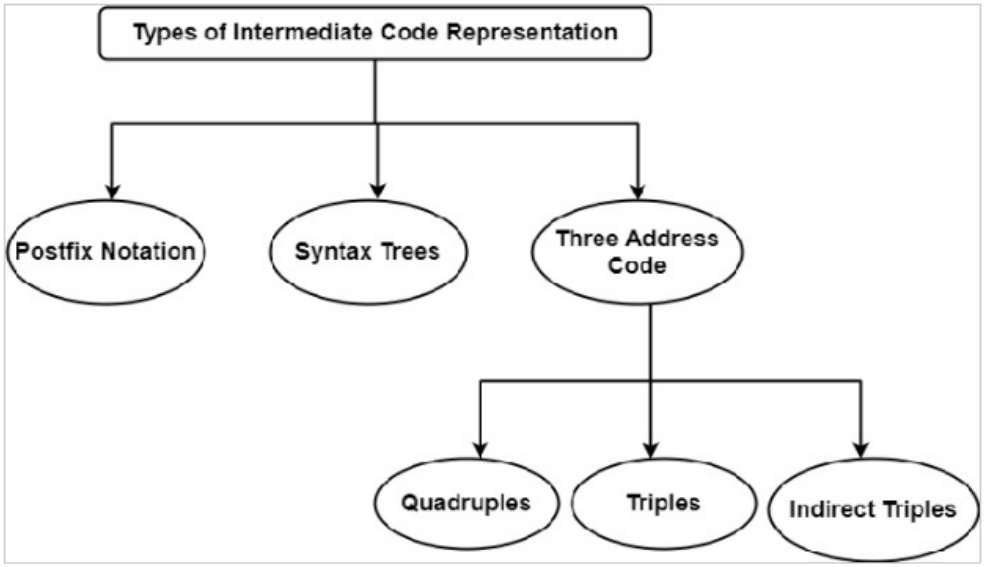
→ A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source language to existing back end.

→ A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation

Types of Intermediate Codes

→ The following are commonly used intermediate code representation :

- Abstract Syntax tree(AST) and Directed Acyclic Graphs(DAG)
- Postfix notation or Reverse Polish Notation or suffix notation
- Three-address code(TAC)



Abstract Syntax Tree:

→ A tree in which each leaf node describes an operand & each interior node an operator.

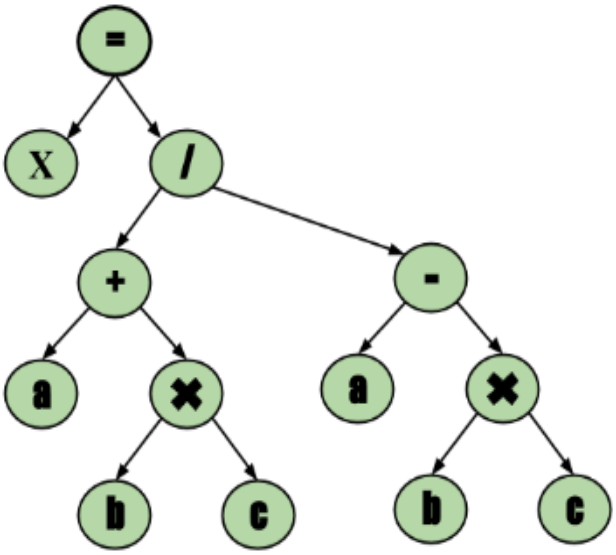
→ The syntax tree is shortened form of the Parse Tree.
i.e., a syntax tree is nothing more than a condensed form of a parse tree.

→ The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by the single link

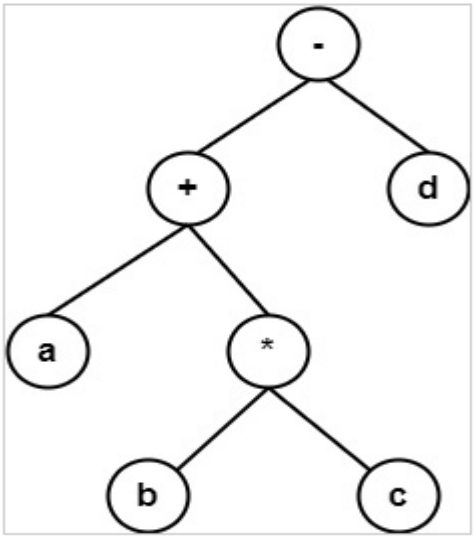
→ In the syntax tree the internal nodes are operators and child nodes are operands.

→ To form a syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

→ **Example:** x = (a + b * c) / (a – b * c)



→ **Example** : Draw Syntax Tree for the string a + b ∗ c − d.



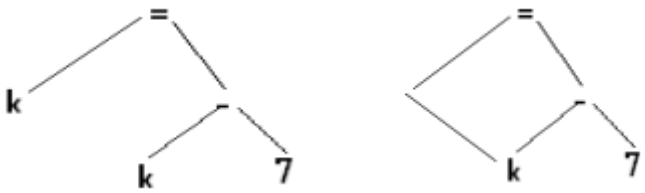
Directed Acyclic Graph (DAG):

→ Directed Acyclic Graph (DAG) is a special kind of Abstract Syntax Tree.
→ A DAG (Directed Acyclic Graph) gives the same information as Syntax Tree but in a more compact way.
→ In DAG , there are more than one path from start symbol to terminals
→ Each node of it contains a unique value.
→ It does not contain any cycles in it, hence called Acyclic.
→ Leaf nodes represent identifiers, names or constants.
→ Interior nodes represent operators.

→ An important derivative of abstract syntax tree is known as Directed Acyclic Graph. It is used to reduce the amount of memory used for storing the Abstract Syntax Tree data structure.

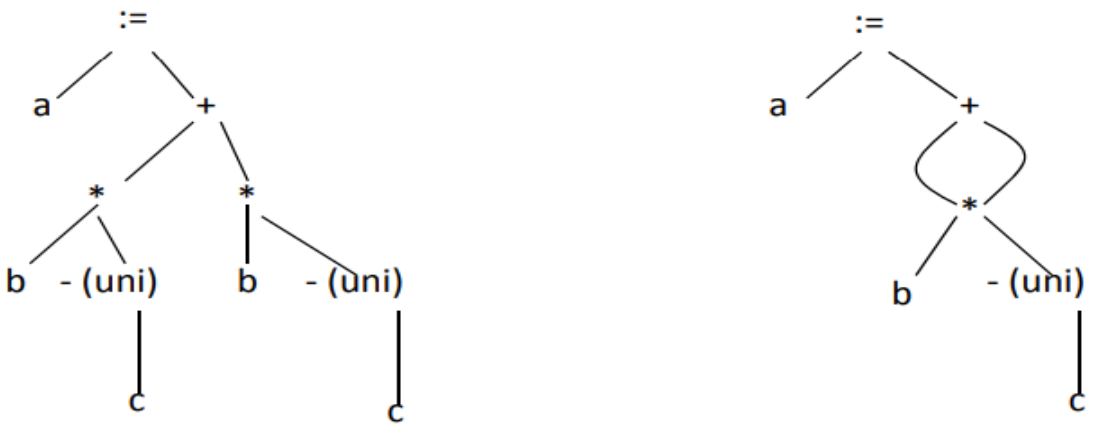
→ Consider an expression: k=k-7; The AST and DAG is shown in the fig below

AST **DAG**



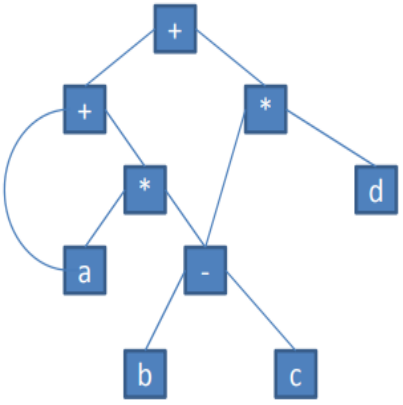
→ For a := b *-c + b*-c

AST **DAG**

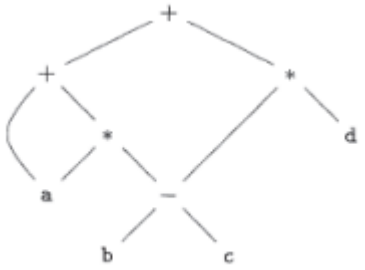


→ A node in a Directed Acyclic Graph (DAG) may have more than one parent.

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



→ Example of DAG for the expression a + a * (b-c) + (b-c) * d

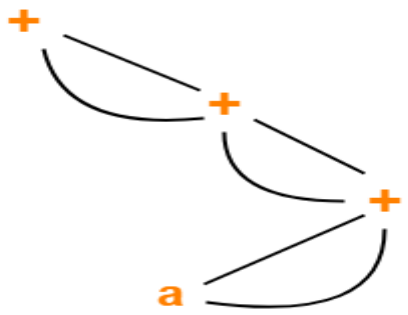


Problem: Consider the following expression and construct a DAG for it-

(((a + a) + (a + a)) + ((a + a) + (a + a)))

Solution-

Directed Acyclic Graph for the given expression is:



Directed Acyclic Graph

Postfix Notation:

→ Also known as reverse Polish notation or suffix notation.

→ The ordinary (infix) way of writing the sum of a and b is with an operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as **ab+**

→ In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +

→ No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression.

→ Unlike infix notation, postfix notation places the operator after the two operands. And, it doesn't use any parenthesis to group expressions.

→ In postfix notation, the operator follows the operand.

Example 1: The postfix representation of the expression (a + b) * c is : **ab + c ***

Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is : **ab – cd + *ab -+**

Example 3: (l + m) * n is an infix expression, the postfix notation will be **l m + n ***

Example 4: p * (q + r) is an infix expression, the postfix expression will be **p q r + ***

Example 5: (p - q) * (r + s) + (p - q) is an infix expression, the postfix expression will be **p q - r s + * p q - +**

Problem: Convert the following expression to the postfix notation and evaluate it. P + (–Q + R * S)

Ans: The postfix notation for the given expression is: PQ - RS * ++

Problems:

Infix Expression: (A + (B * C))
Postfix Expression: A B C * +

Infix Expression: ((A + (B * C)) / (D + E))
Postfix Expression: A B C* + D E + /

Infix Expression: ((A + B) * (C + E))
Postfix Expression: A B + C E + *

Infix Expression: (A * (B * (((C + A) + B) * C)))
Postfix Expression: ABC A + B + C * * *

Infix Expression: ((H * ((((A + ((B + C) * D)) * F) * G) * E)) + J)
Postfix Expression: H A B C + D * + F * G * E * * J +

Three-Address Code (TAC)

→ A statement involving no more than three addresses (two for operands and one for result) is known as a three address statement. A sequence of three address statements is known as a three address code.

→ Three address statement is of form x = y op z, where x, y, and z will have address (memory location). Sometimes a statement might contain less than three references but it is still called a three address statement.

→ Three address codes is a type of intermediate code which is easy to generate and can be easily converted to machine code.

→ It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.

→ The compiler decides the order of operation given by three address code.

→ **General representation:**

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and **op** represents the operator

→ **Example:** The three address code for the expression a + b * c + d is

t_1 = b * c

t_2 = a + t_1

t_3 = t_2 + d

Where t_1 , t_2 , t_3 are temporary variables.

Implementation of Three Address Codes:

(Data structures for three address codes)

→ There are 3 representations of three address code namely

1. Quadruples
2. Triples
3. Indirect Triples

Quadruples:

→ Quadruple is defined as a record structure used to represent a three-address statement.

→ It consists of four fields. The first field contains the operator, the second and third fields contain the operand 1 and operand 2, respectively, and the last field contains the result of that three-address statement.

→ **Advantages:**

- 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.

→ **Example:** Consider expression a = b * – c + b * – c. The three address code is:

t1 = uminus c

t2 = b * t1

t3 = uminus c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

Triples

→ A triple is also defined as a record structure that is used to represent a three address statement.

→ In triples, for representing any three-address statement three fields are used, namely, operator, operand 1 and operand 2, where operand 1 and operand 2 are pointers to either symbol table or they are pointers to the records (for temporary variables) within the triple representation itself.

→ In this representation, the result field is removed to eliminate the use of temporary names referring to symbol table entries. Instead, we refer the results by their positions.

→ The pointers to the triple structure are represented by parenthesized numbers, whereas the symbol-table pointers are represented by the names themselves.

→ Note that instead of referring the temporary t by its name, we refer it by its position in the triple.

→ **Disadvantages:**

- 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.

→ **Example:**Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

Indirect Triples:

→ An indirect triple representation consists of an additional array that contains the pointers to the triples in the desired order

→ Its similar in utility as compared to quadruple representation but requires less space than it.

→ Temporaries are implicit and easier to rearrange code.

→ **Example –** Consider expression a = b * – c + b * – c

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Question: Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

Explanation: The three address code is:

```
t1 = x + y
t2 = y + z
t3 = t1 * t2
t4 = t1 + z
t5 = t3 + t4
```

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

Problem:

Convert S = -z/a * (x + y) into three address code,where Unary minus is represented by -z.

The following is the three-address code:

t1 = x + y
t2 = a * t1
t3 = - z
t4 = t3/t2
S = t4

Quadruple Representation

#	Operator	Operand1	Operand2	Result
0	+	x	y	t_1
1	*	a	t_1	t_2
2	-	z		t_3
3	/	t_3	t_2	t_4
4	=	t_4		S

Triple Representation

#	Operator	Operand1	Operand2
0	+	x	y
1	*	a	(0)
2	-	z	
3	/	(2)	(1)
4	=	S	(3)

Indirect Triple Representation

#	A
101	(0)
102	(1)
103	(2)
104	(3)
105	(4)

#	Operator	Operand1	Operand2
0	+	x	y
1	*	a	(0)
2	-	z	
3	/	(2)	(1)
4	=	S	(3)

Problem: Construct Quadruples, Triples, and Indirect Triples for the expression **-(a + b) * (c + d) - (a + b + c)**

Solution

First of all this statement will be converted into Three Address Code as:

t1 = a + b
t2 = −t1
t3 = c + d
t4 = t2 ∗ t3
t5 = t1 + c
t6 = t4 − t5

Quadruple

Location	Operator	arg 1	arg 2	Result
(0)	+	a	b	t1
(1)	–	t1		t2
(2)	+	c	d	t3
(3)	*	t2	t3	t4
(4)	+	t1	c	t5
(5)	–	t4	t5	t6

Triple

Location	Operator	arg 1	arg 2
(0)	+	a	b
(1)	–	(0)	
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	(0)	c
(5)	–	(3)	(4)

Indirect Triple

Statement
(0) (11)
(1) (12)
(2) (13)
(3) (14)
(4) (15)
(5) (16)

Location	Operator	arg 1	arg 2
(11)	+	a	b
(12)	-	(11)	
(13)	+	c	d
(14)	*	(12)	(13)
(15)	+	(11)	c
(16)	-	(14)	(15)

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Code Optimization

→ Code Optimization is an approach to improve the code by eliminating unnecessary code lines and arranging the statement in such a sequence that speed up the program execution without changing the meaning of the program

→ The code produced by the straight forward compiling algorithms can often be faster or take less space, or both. This improvement is achieved by program t that are traditionally called optimizations. Compilers that apply code-improvi transformations are called optimizing compilers.

→ The process of code optimization involves:
- Eliminating the unwanted code lines
- Rearranging the statements of the code

→ **The criteria for code Optimization:**
- The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input.
- A transformation must, on the average, speed up programs by a measurable amount.
- The transformation must be worth the effort. i.e., yield the most benefit for the least effort.

→ **Advantages of Code optimization:**
- Optimized code has faster execution speed.
- Optimized code utilizes the memory efficiently.
- Optimized code gives better performance.

→ Optimizations are classified into two categories. They are
- Machine independent optimizations
- Machine dependant optimizations

→ Machine independent optimizations: Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

→ Machine dependant optimizations: Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences

Common Types of Code Optimization Techniques

(Principle Sources of Optimization (or) Structure-Preserving Transformation)

Local Optimization:

1. Common sub-expression elimination
2. Dead Code Elimination
3. Compile Time Evaluation

Loop Optimization:

4. Code Movement
5. Strength Reduction
6. Induction Variable elimination

Common Sub-Expression Elimination:

→ An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.

→ We can avoid re-computing the expression if we can use the previously computed value.

→ In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

→ **Example 1:**

a:= b+c
b:=a-d
c:=b+c
d:=a-d

in the above code a-d is common sub expression but b+c is not common sub expression

The above code can be optimized using the common sub-expression elimination as

a:= b+c
b:=a-d
c:=b+c
d:=b

→ **Example 2:**

	Code After Optimization
t1: = 4*i t2: = a [t1] t3: = 4*j t4: = 4*i t5: = n t6: = b[t4] +t5	t1: = 4*i t2: = a [t1] t3: = 4*j t5: = n t6: = b[t1] +t5

The above code can be optimized using the common sub-expression elimination as
The common sub expression t4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

Dead Code Elimination:

→ A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
→ A piece of code is said to be dead, whose computes values are never used anywhere in the program.
→ In this technique,
- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

→ A related idea is dead or useless code, statements that compute values that never get used. An optimization can be done by eliminating dead code.
→ Example:

	Code After Optimization

i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
	i = 0 ;

Here, 'if' statement is dead code because this condition will never get satisfied

Compile Time Evaluation:

Two techniques that fall under compile time evaluation are-

i) Constant Folding:

In this technique,
→ As the 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 1:

Circumference of Circle = (22/7) x Diameter

Here,

- This technique evaluates the expression 22/7 at compile time.
- The expression is then replaced with its result 3.14.
- This saves the time at run time.

Example 2:

For example, a=3.14157/2 can be replaced by a=1.570 thereby eliminating a division operation.

ii) 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:

radius = 10

Area = pi x radius x radius

Here,

- This technique substitutes the value of variables 'pi' and 'radius' at compile time.
- It then evaluates the expression 3.14 x 10 x 10.
- The expression is then replaced with its result 314.
- This saves the time at run time.

Code Movement or Frequency Reduction

(Moving loop-invariant computation)

→ An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.

→ This technique reduces the execution frequency of expression by moving the code.

→ In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Example1:

	Code After Optimization

for (j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 * j;
}
	x = y + z ;
	for (j = 0 ; j < n ; j ++)
	{
	a[j] = 6 * j;
	}

Example 2:

For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /* statement does not change limit*/

Code motion will result in the equivalent of

t= limit-2;

while (i<=t) /* statement does not change limit or t */

Strength Reduction:

In this technique,

→ As the name suggests, it involves reducing the strength of expressions.

→ This technique replaces the expensive and costly operators with the simple and cheaper ones.

→ For example, x*x cheaper to implement as x+ x. x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

Example:

	Code After Optimization
B = A x 2	B = A + A

Here,

- The expression "A x 2" is replaced with the expression "A + A".
- This is because the cost of multiplication operator is higher than that of addition operator.

Induction variable Elimination:

→ A basic induction variable is a variable X whose only definitions within the loop are assignments of the form: X = X+c or X = X-c, where c is either a constant or a loop-invariant variable. (e.g., i)

→ An induction variable is a basic induction variable B, or a variable defined once within the loop, whose value is a linear function of some basic induction variable at the time of the definition: A = c1 * B + c2

→ When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination.

→ Example:

Code before optimization

```
s=3*i + 1;
While(i<10)
{
a[s] = a[s]-2;
i= i+2;
s=s+6;
}
```

Code after optimization

```
s=3*i + 1;
While(s<31)
{
a[s] = a[s]-2;
s=s+6;
}
```

Here s and i are induction variables. We can eliminate i from the loop.

Code Generation

→ The final phase in compiler model is the code generator.

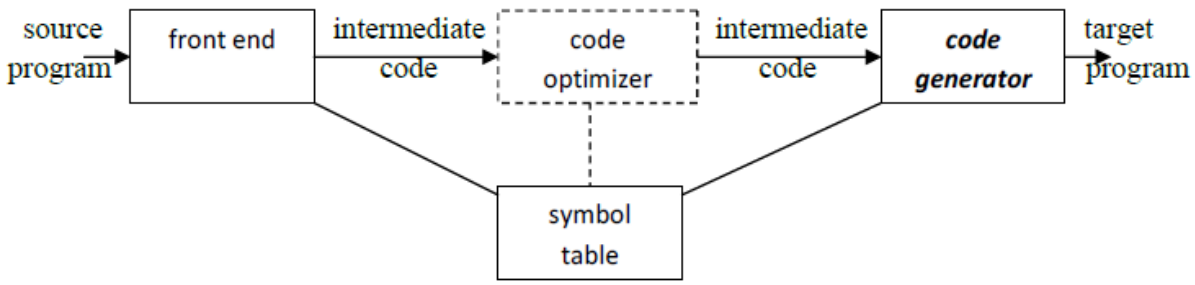
→ It is machine dependent phase.

→ It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

→ It is the most complex phase of a compiler, since it depends not only on the characteristics of the source language but also on detailed information about the target machine architecture, the structure of the run time environment and the operating system running on the target machine.

→ The position of code generator in compilation process is illustrated by following figure.

Position of code generator



Issues In The Design Of A Code Generator

→ The following issues arise during the code generation phase:

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

1. Input to code generator:

→ The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

→ Intermediate representation can be :

- Linear representation such as postfix notation
- Three address representation such as quadruples
- Virtual machine representation such as stack machine code
- Graphical representations such as syntax trees and DAGs.

→ Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

→ The output of the code generator is the target program. The output may be:

- Absolute machine language
  - It can be placed in a fixed memory location and can be executed immediately.
- Relocatable machine language
  - It allows subprograms to be compiled separately.
- Assembly language
  - Code generation is made easier.

3. Memory management:

→ Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

→ It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

→ Labels in three-address statements have to be converted to addresses of instructions.

→ For example,

j : goto i generates jump instruction as follows :

- if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.
- if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

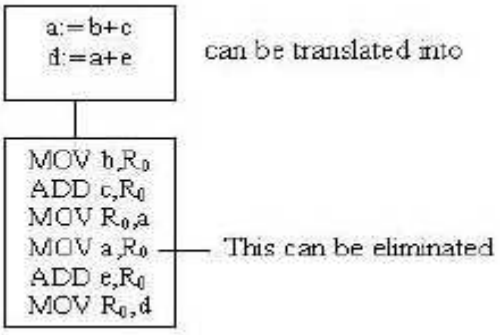
4. Instruction selection:

→ The instructions of target machine should be complete and uniform.

→ Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

→ The quality of the generated code is determined by its speed and size.

→ The former statement can be translated into the latter statement as shown below:



5. Register allocation:

→ Instructions involving register operands are shorter and faster than those involving operands in memory.

→ The use of registers is subdivided into two sub problems :

- Register allocation – the set of variables that will reside in registers at a point in the program is selected.
- Register assignment – the specific register that a variable will reside in is picked.

→ Certain machine requires even-odd register pairs for some operands and results. For example , consider the division instruction of the form :

D x, y

Where, x – dividend even register in even/odd register pair

y – divisor

even register holds the remainder

odd register holds the quotient

6. Evaluation order:

→ The order in which the computations are performed can affect the efficiency of the target code.

→ Some computation orders require fewer registers to hold intermediate results than others.

7. Approaches to code generation:

→ Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face.

→ Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

Basic Blocks And Flow Graphs

Basic Blocks

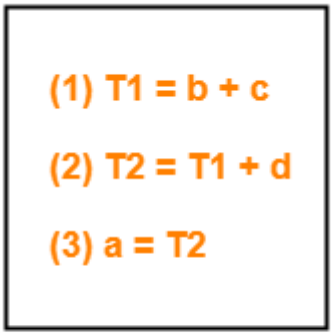
→ A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

→ The characteristics of basic blocks are-

- They do not contain any kind of jump statements in them.
- There is no possibility of branching or getting halt in the middle.
- All the statements execute in the same order they appear.
- They do not lose the flow control of the program.
- They executes in a sequence one after the other.

→ Example Of Basic Block:

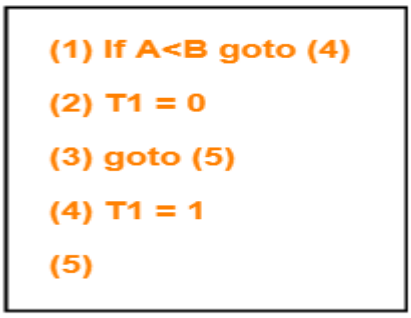
Three Address Code for the expression **a = b + c + d** is:



Basic Block

→ Example Of Not A Basic Block:

Three Address Code for the expression **If A<B then 1 else 0** is:



Not a basic Block

→ Example2: The following sequence of three-address statements forms a basic block:

t1 : = a * a
t2 : = a * b
t3 : = 2 * t2
t4 : = t1 + t3
t5 : = b * b
t6 : = t4 + t5

Basic Block Construction:

(Algorithm to Partition into basic blocks)

→ Algorithm:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:
   - The first statement is a leader.
   - Any statement that is the target of a conditional or unconditional goto is a leader.
   - 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.

→ Consider the following source code for dot product of two vectors a and b of length 20

```
begin
  prod :=0;
   i:=1;
  do begin
       prod :=prod+ a[i] * b[i];
       i :=i+1;
  end
 while i <= 20
end
```

→ The three-address code for the above source program is given as :

(1) prod := 0
(2) i := 1
(3) t1 := 4* i
(4) t2 := a[t1] /*compute a[i] */
(5) t3 := 4* i
(6) t4 := b[t3] /*compute b[i] */
(7) t5 := t2*t4
(8) t6 := prod+t5
(9) prod := t6
(10) t7 := i+1
(11) i := t7
(12) if i<=20 goto (3)

Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

Flow Graphs

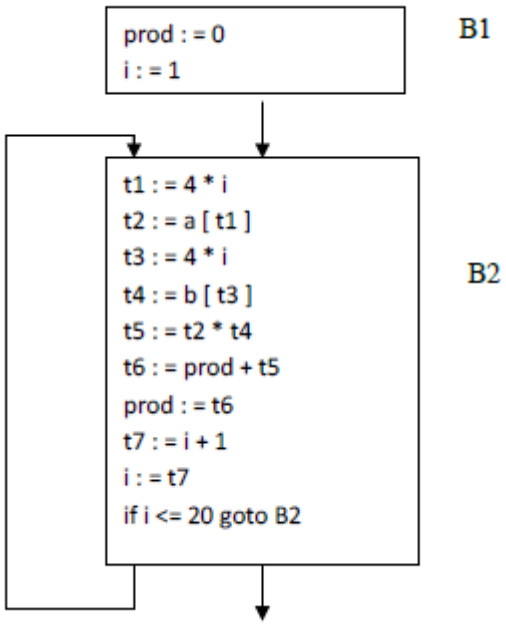
→ Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.

→ The nodes of the flow graph are basic blocks. It has a distinguished initial node.

→ There is a directed edge from block B1 to block B2 if B2 appears immediately after B1 in the code.

→ 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.

→ E.g.: Flow graph for the vector dot product is given as follows:



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

Example 2:

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

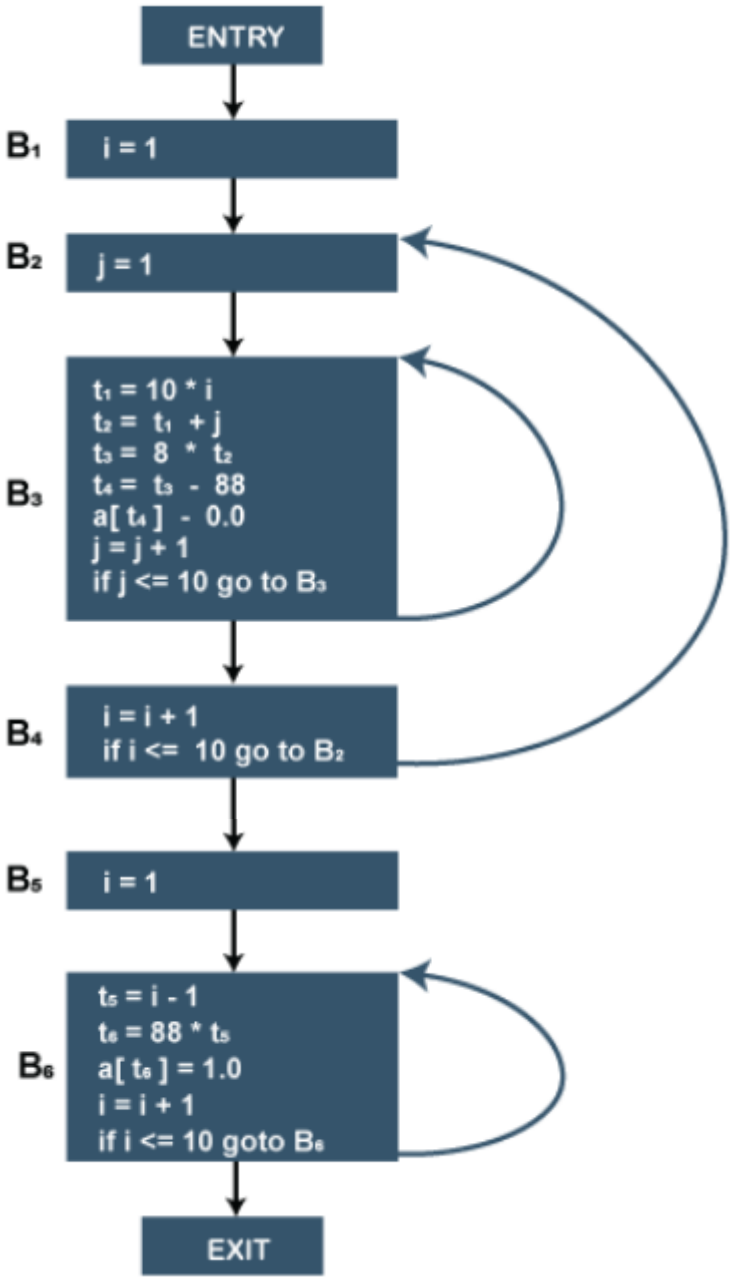
→ According to the given algorithm, instruction 1 is a leader.

- Instruction 2 is also a leader because this instruction is the target for instruction 11.
- Instruction 3 is also a leader because this instruction is the target for instruction 9.
- Instruction 10 is also a leader because it immediately follows the conditional goto statement.
- Similar to step 4, instruction 12 is also a leader.
- Instruction 13 is also a leader because this instruction is the target for instruction 17.

→ So there are six basic blocks for the above code, which are given below:

- **B1 for statement 1**
- **B2 for statement 2**
- **B3 for statement 3-9**
- **B4 for statement 10-11**

- **B5 for statement 12**
- **B6 for statement 13-17.**



PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:

  - ✓ Redundant-instructions elimination
  - ✓ Flow-of-control optimizations
  - ✓ Algebraic simplifications
  - ✓ Use of machine idioms
  - ✓ Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R_0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0

....

If ( debug ) {

Print debugging information

}
```

- In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2
goto L2
L1: print debugging information
L2:                                   ……(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2
Print debugging information
L2:                                   ……(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

```
If debug ≠0 goto L2
Print debugging information
L2:                                   ……(c)
```

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
goto L1
….
L1: gotoL2
```

by the sequence

```
goto L2
….
L1: goto L2
```

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

```
if a < b goto L1
....
L1: goto L2
```

can be replaced by

```
If a < b  goto  L2
....
L1: goto L2
```

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

```
goto L1
……..
L1: if a < b goto L2
L3:                    ……..(1)
```

- May be replaced by

```
If a < b goto L2
goto L3
…….
L3:                    …….(2)
```

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

```
x := x+0
Or
x := x * 1
```

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 ⟶ i++

i:=i-1 ⟶ i- -

Register Allocation and Assignment

→ Register allocation is the process of assigning variables to registers and managing data transfer in and out of registers.

→ Various Approaches:

- Local register allocation
- Global register Allocation
- Register Allocation by Usage Counts
- Register assignment for outer loop
- Register allocation by graph coloring

Local register allocation

→ Register allocation is only within a basic block. It follows top-down approach.

→ Assign registers to the most heavily used variables

Traverse the block

Count uses

Use count as a priority function

Assign registers to higher priority variables first

→ Advantage
Heavily used values reside in registers
→ Disadvantage
Does not consider non-uniform distribution of uses

Global register allocation

→ Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.
→ To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.
→ Register allocation depends on:
- Size of live range
- Number of uses/definitions
- Frequency of execution
- Number of loads/stores needed.
- Cost of loads/stores needed.

Register Allocation by Usage Counts

→ A slightly more sophisticated method for global register allocation is called usage counts.
→ In this method, registers are allocated first to the variables that are used the most.
→ **Example :**

Consider the following loop:

```
LOOP: X = 2 * E
        Z = Y + X + 1
         IF some condition THEN
        Y = Z + Y
         D = Y - 1
        ELSE Z = X - Y
         D = 2
         ENDIF
         X = Z + D
         Z = X
   ENDLOOP
```

Here, there are five references to *X*, and *Z*, five references to *Y*, three references to *D*, and one to *E*. Thus, if there are three registers, a reasonable approach would be to allocate *X* and *Z* to two of them, saving the third for local computations.

Register assignment for outer loop

→ Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger enclosing loops. If an outer loop *L1* contains an inner loop L_2, the names allocated registers in L_2 need not be allocated registers in L1 — L2. Similarly, if we choose to allocate x a register in L2 but not Li, we must load x on entrance to L2 and store x on exit from L2.

Register allocation by graph coloring

→ Global register allocation can be seen as a graph coloring problem.

→ Basic idea:

1. Identify the live range of each variabl
2. Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
3. Try to assign as many colors to the nodes of the graph as there are registers so that two neighbors have different colors

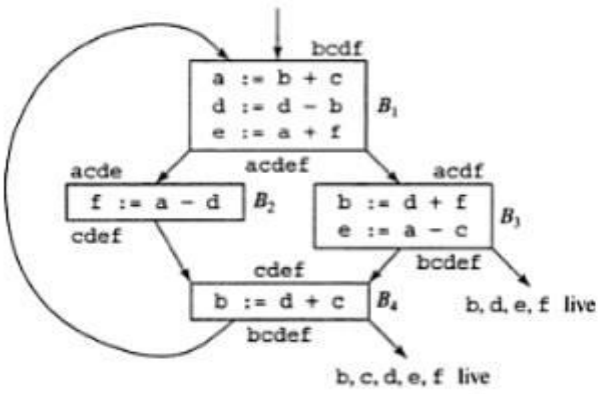


Fig 4.3 Flow graph of an inner loop

Tutorial Questions:

1. What is role of intermediate Code generator in compilation process? Explain Various Forms Of Intermediate Codes Used By Compiler.
2. Explain various methods of implementing three address statements with suitable examples.
   (or) Explain about quadruples, triples and indirect triples of three-address statements of intermediate code
3. Write quadruples, triples and indirect triples for the expression:
   -(a*b)+(c+d)-(a+b+c+d)
   (or) For the given expression generate different kinds of three-address codes -(a*b)+(c+d)-(a+b+c+d)
4. Write the short note on: (i) Abstract syntax tree (ii) Polish notation (iii) Three address code

5. Construct abstract syntax tree & DAG for the assignment statement
   x:=a*b+c-a*b+d
6. Write the quadruple, triple, indirect triples for the statement
   a:= b * -c + b * -c
7. Translate the assignment A:= -B*(C+D) into following
   i) Quadruple ii) Triples iii) Indirect triples
8. Translate the expression -(a+b)*(c+d)+(a+b+c) into the following
   i) Quadruples ii) Triples iii) Indirect triples
9. Convert the following arithmetic expressions into Abstract syntax tree, DAG, postfix notation and three-address code:
   i) b*-(a+b)
   ii) a+b*(a+b)+c+d
10. What is code optimization? Compare machine dependent and independent code optimization techniques.
11. What is code optimization? Explain about various levels and types of optimizations
    (or) Explain different principle sources of optimization techniques with suitable examples
12. Discuss about principal sources of optimization.
13. Write short note on
    a. Constant Folding
    b. Dead Code Elimination
    c. Code Motion
    d. Induction Variable Elimination
14. Discuss briefly various loop optimization techniques.
15. Define flow graph. Explain the optimization of Basic Blocks.
16. Write about all issues in code generation. Describe it.
    (or) Explain the different issues in the design of a code generator
17. Explain the peephole optimization Techniques?
    (or) Discuss the transformations that are characteristic of peephole optimizations.
    (or) What kinds of peephole techniques can be used to perform machine-dependent optimizations?+
18. What is a basic block and flow graph? Explain how flow graph can be constructed for a given program.
19. What is peephole optimization? How can it be performed? Give its role in code generation.
20. Discuss about register allocation and assignment in target code generation.