Department of Computer Science & Engineering

PUNJABI UNIVERSITY, PATIALA

Compiler Design

Sec-B

Part-2

Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).
The benefits of using machine independent intermediate code are:

- Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, 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. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- Retargeting is facilitated
- It is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.

If we generate machine code directly from source code then for n target machine, we will have n optimisers and n code generators but if we will have a machine independent intermediate code, we will have only one optimiser. Intermediate code can be either language specific (e.g., Bytecode for Java) or language. independent (three-address code).

The following are commonly used intermediate code representation:

- **Three-Address Code**
- **Postfix Notation**
- **Syntax Tree**
- **Directed acyclic graph (DAG)**

Three-Address Code

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b * c + d :
T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d
T 1, T 2 , T 3 are temporary variables.

Or

- 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.

Example

GivenExpression:
a := (-c * b) + (-c * d)
Three-address code is as follows:

t_1 := -c
t_2 := b*t_1
t_3 := -c
t_4 := d * t_3
t_5 := t_2 + t_4
a := t_5

t is used as registers in the target program.

Common Three-Address Instruction Forms

Assignments.
Two possible forms: • x= y op z where op is a binary arithmetic or logical operation, • x= op y where op is a unary operation (minus, negation, conversion operator)
Copy statements.
They have the form x = y.
Inconditional jumps. They have the form goto L where L is a symbolic label of a statement.
Conditional jumps.
They have the form if x relop y goto L where statement L is executed if x and y are in relation relop.
Procedure calls.
They have the form param x1 param x2 param xn call p, n corresponding to the procdure call p(x1, x2, …, xn)

Return statement.
They have the form return y where y representing a returned value is optional.
Indexed assignments.
They have the form x = y[i] or x[i] = y.
Address assignments.
They have the form x = &y which sets x to the location of y.
Pointer assignments.
They have the form • x = *y where y is a pointer and which sets x to the value pointed to by y • *x = y which changes the location of the value pointed to by x.

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

Quadruples

The quadruples have four fields to implement the three-address code. The field of quadruples contains the name of the operator, the first source operand, the second source operand and the result respectively.

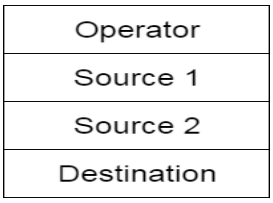


Fig: Quadruples field

Example-1

a := -b * c + d

Three-address code is as follows:

t_1 := -b
t_2 := c + d
t_3 := t_1 * t_2
a := t_3

These statements are represented by quadruples as follows:

	Operator	Source 1	Source 2	Destination
(0)	uminus	b	-	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3	-	a

Example – 2
Consider expression a = b * – c + b * – c.
The three-address code is:
t1 = uminus c
t2 = t1 * b
t3 = uminus c
t4 = t3 * b
t5 = t2 + t4
a = t5

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

Quadruple representation

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.

Triples

The triples have three fields to implement the three-address code. The field of triples contains the name of the operator, the first source operand and the second source operand.
In triples, the results of respective sub-expressions are denoted by the position of expression. Triple is equivalent to DAG while representing expressions.

Operator
Source 1
Source 2

Fig: Triples field

Example-1

a := -b * c + d
Three address code is as follows:
t_1 := -b t_2 := c + dM t_3 := t_1 * t_2 a := t_3
These statements are represented by triples as follows:

	Operator	Source 1	Source 2
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

Example-2 – 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)	=	(4)	a

Triples representation

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.

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.

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)	=	[illegible]	[illegible]

#	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)

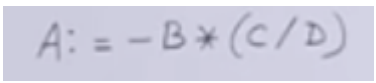
Triples representation

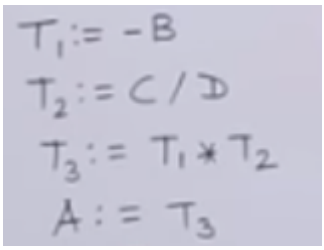
#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

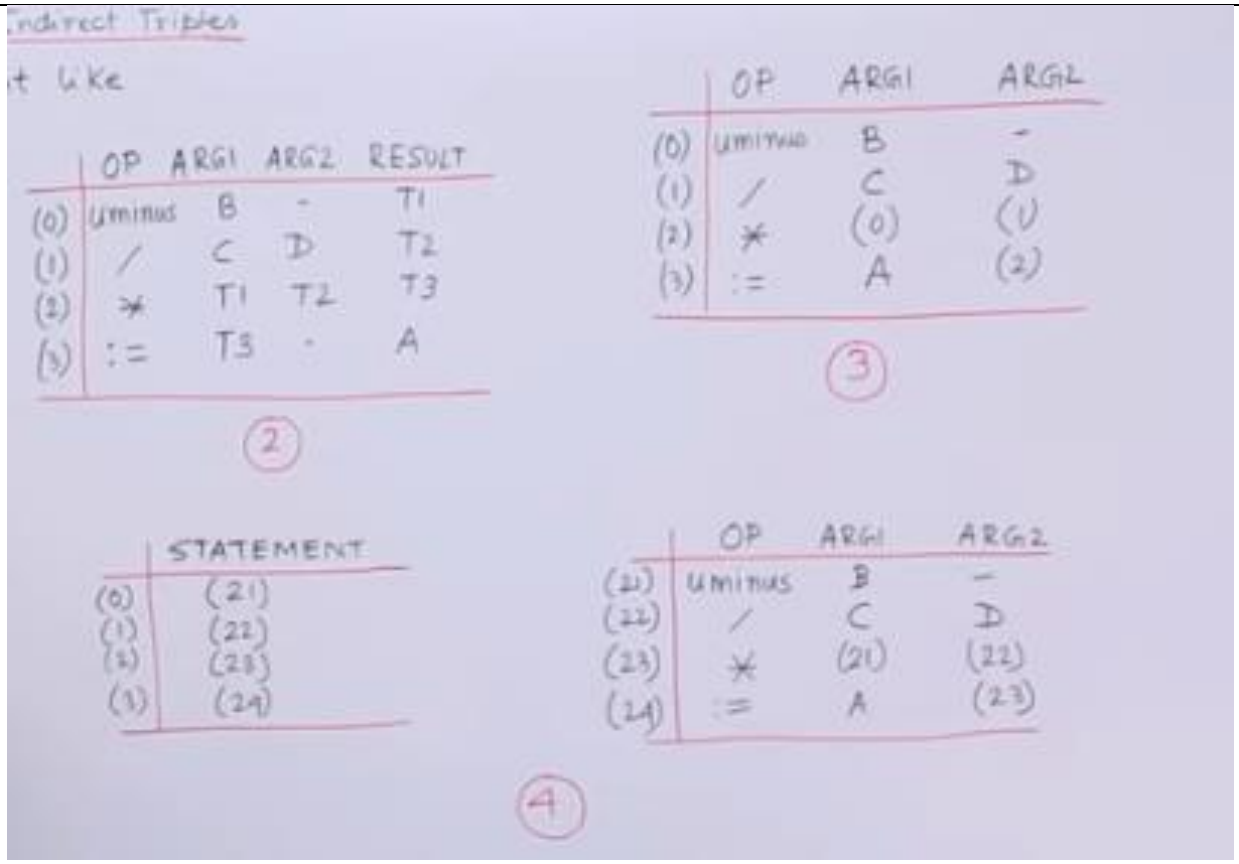
List of pointers to table

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

Indirect Triples representation





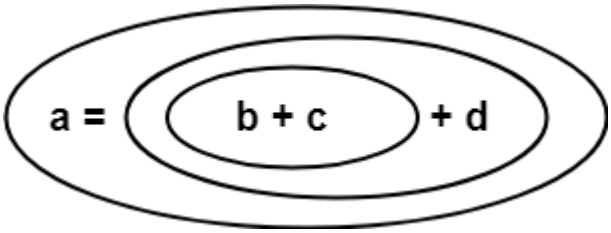


Problem-01:

Write Three Address Code for the expression given below-
a = b + c + d

Solution-

The given expression will be solved as-



So, We can write Three Address Code for the expression given as-
(1) T1 = b + c
(2) T2 = T1 + d
(3) a = T2

Problem-02:

Write Three Address Code for the expression given below-
-(a x b) + (c + d) – (a + b + c + d)

Solution-

We can write Three Address Code for the expression given as-
(1) T1 = a x b
(2) T2 = uminus T1
(3) T3 = c + d

(4) T4 = T2 + T3
(5) T5 = a + b
(6) T6 = T3 + T5
(7) T7 = T4 – T6

Problem-03:

Convert the expression a * – (b + c) into three address code.

$t_1 = b + c$
$t_2 = uminus\ t_1$
$t_3 = a * t_2$

Postfix Notation –

The ordinary (infix) way of writing the sum of a and b is with 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. In postfix notation the operator follows the operand.

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

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

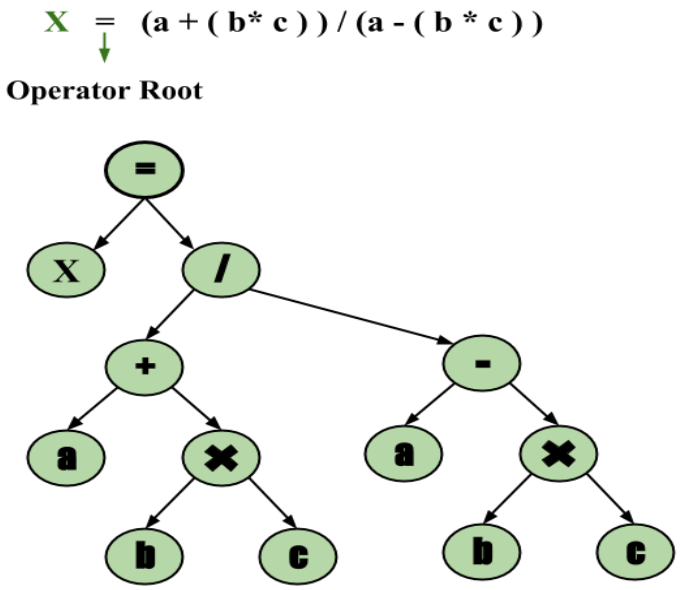
Syntax Tree

Abstract Syntax Tree

• Condensed form of a parse tree
• useful for representing language constructs

Syntax tree is nothing more than 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 single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

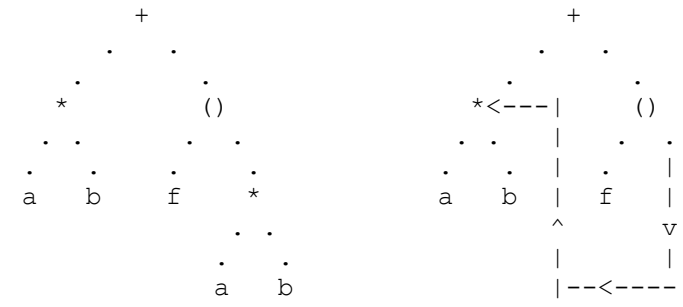
Example –
x = (a + b * c) / (a – b * c)



Directed acyclic graph (DAG)

A *directed acyclic graph* (DAG!) is a directed graph that contains no cycles. A rooted tree is a special kind of DAG and a DAG is a special kind of directed graph. For example, a DAG may be used to represent common subexpressions in an optimising compiler.

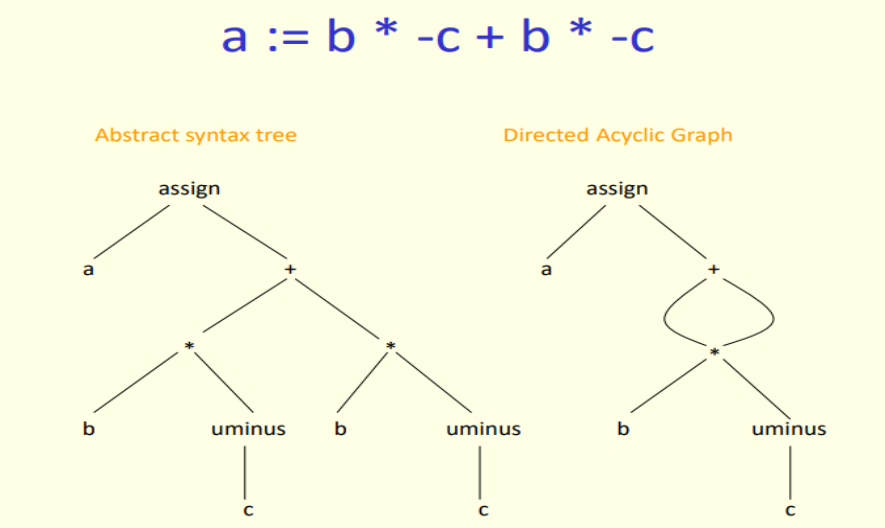
DAG is more compact than abstract syntax tree because common sub expressions are eliminated



expression: a*b+f(a*b)

Example of Common Subexpression.

The common subexpression a*b need only be compiled once but its value can be used twice.



Backpatching

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example, the translation of the boolean expression *B* in if (*B*) *S* contains a jump, for when *B* is false, to the instruction following the code for *S*. In a one-pass translation, *B* must be translated before *S* is examined. Labels can be passed as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses.

In ***backpatching,*** lists of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

Problem-01:

Write Three Address Code for the expression given below-
If A < B then 1 else 0

Solution-

We can write Three Address Code for the expression given as-
(1) If (A < B) goto (4)
(2) T1 = 0
(3) goto (5)
(4) T1 = 1
(5)

Problem-02:

Write Three Address Code for the expression given below-
If A < B and C < D then t = 1 else t = 0

Solution-

We can write Three Address Code for the expression given as-
(1) If (A < B) goto (3)
(2) goto (4)
(3) If (C < D) goto (6)

(4) t = 0
(5) goto (7)
(6) t = 1
(7)

Control-Flow Realization of Boolean Expressions

if ((a+b < c+d) || ((e==f) && (g > h-k))) A1; else A2; A3;

```
100:          T1 = a+b
101:          T2 = c+d
103:          if T1 < T2 goto L1
104:          goto L2
105:L2:       if e==f goto L3
106:          goto L4
107:L3:       T3 = h-k
108:          if g > T3 goto L5
109:          goto L6
110:L1:L5:    code for A1
111:          goto L7
112:L4:L6:    code for A2
113:L7:       code for A3
```

Code for a < b or c < d and e < f

100: if a < b goto 103
101: $t_1 = 0$
102: goto 104
103: $t_1 = 1$
104:
if c < d goto 107
105: $t_2 = 0$
106: goto 108
107: $t_2 = 1$
108:
if e < f goto 111
109: $t_3 = 0$
110: goto 112
111: $t_3 = 1$
112:
$t_4 = t_2$ and t_3
113: $t_5 = t_1$ or t_4

Code Generation Template for *C For-Loop*

for (E_1; E_2; E_3) S

```
        code for E1
L1:     code for E2 (result in T)
        goto L4
L2:     code for E3
        goto L1
L3:     code for S /* all jumps out of S goto L2 */
        goto L2
L4:     if  T == 0  goto  L5 /* if T is zero, jump to exit */
        goto L3
L5:     /* exit */
```

Example ...

Code for

```
while a < b do
        if c<d then
                x=y+z
        else
                x=y-z
```

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 = Y + Z
        X= t1
        goto L1
L4:     t1 = Y - Z
        X= t1
        goto Ll
Lnext:
```

Code Optimization

Code Optimization is an approach for enhancing the performance of the code by improving it through the elimination of unwanted code lines and by rearranging the statements of the code in a manner that increases the execution speed of the code without any wastage of the computer resources.

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. Compiler optimizing process should meet the following objectives:

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

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand a smaller number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

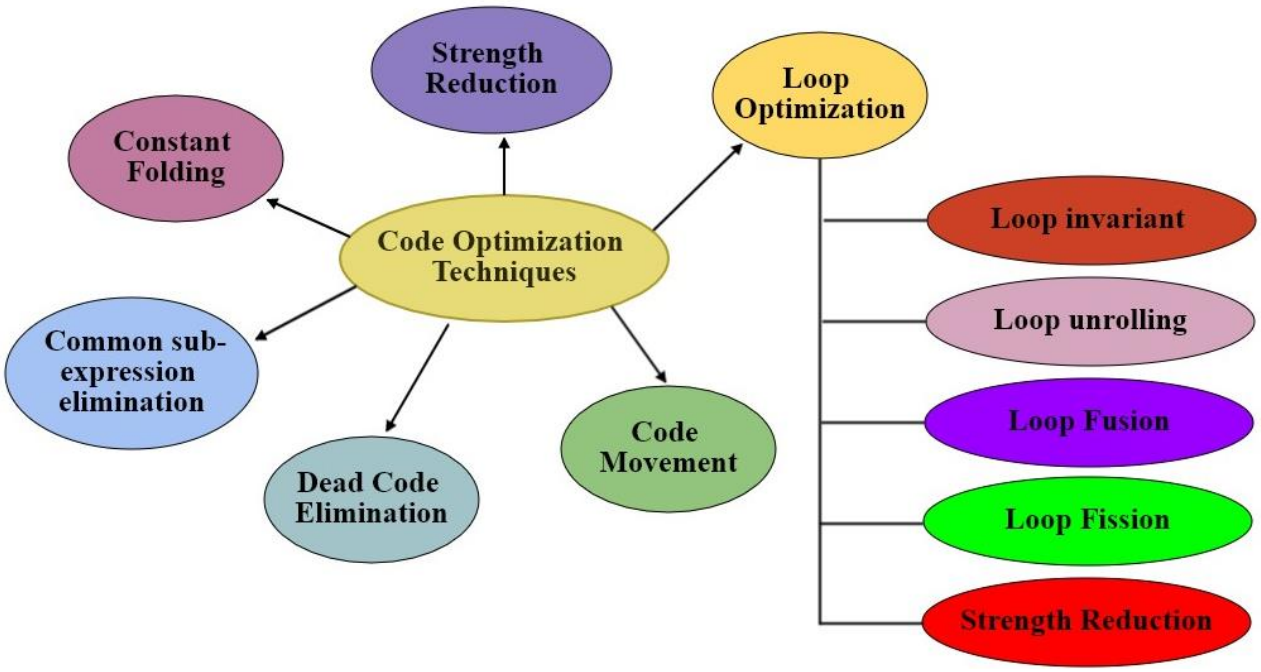
Types of Code Optimization –The optimization process can be broadly classified into two types:

1. **Machine Independent Optimization** – This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization** – Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Advantages of Code Optimization-

- Optimized code has faster execution speed
- Optimized code utilizes the memory efficiently
- Optimized code gives better performance

Techniques for Code Optimization-



- Constant folding
- Common sub-expression elimination
- Dead Code Elimination
- Code Movement
- Strength Reduction
- Loop Optimization

Constant folding

As the name suggests, this technique involves folding the constants by evaluating the expressions that involves the operands having constant values at the compile time.

Example-
Circumference of circle = (22/7) x Diameter
Here, this technique will evaluate the expression 22/7 and will replace it with its result 3.14 at the compile time which will save the time during the program execution.

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 wherever it has been used during compilation, provided that its value does not get alter in between.

Example-
pi = 3.14
radius = 10
Area of circle = pi x radius x radius
Here, this technique will substitute the value of the variable's 'pi' and 'radius' at the compile time and then it will evaluate the expression 3.14 x 10 x 10 at the compile time which will save the time during the program execution.

Common sub-expression elimination

The expression which has been already computed before and appears again and again in the code for computation is known as a common sub-expression.
As the name suggests, this technique involves eliminating the redundant expressions to avoid their computation again and again. The already computed result is used in the further program wherever its required.

Example-

Code before Optimization	**Code after Optimization**
S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i **// Redundant Expression** S5 = n S6 = b[S4] + S5	S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

Code Movement

As the name suggests, this technique involves the movement of the code where the code is moved out of the loop if it does not matter whether it is present inside the loop or it is present outside the loop.
Such a code unnecessarily gets executed again and again with each iteration of the loop, thus wasting the time during the program execution.

Example-

Code before Optimization	Code after Optimization

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

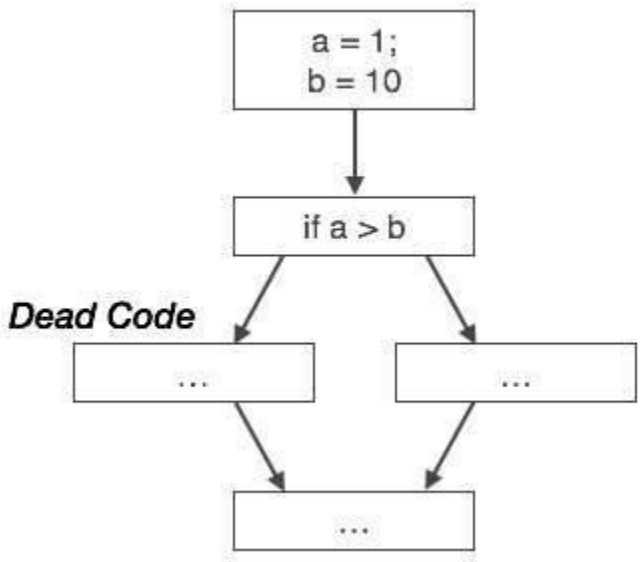
Dead code elimination

As the name suggests, this technique involves eliminating the dead code where those statements from the code are eliminated which either never executes or are not reachable or even if they get execute, their output is never utilized.

Example-

Code before Optimization	Code after Optimization

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



Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

Loop Optimization

- **Loop invariant**
- **Loop unrolling/Loop unwinding**
- **Loop fusion/Loop jamming/Loop ramming/Loop combining**
- **Loop fission/Loop Distribution**
- **Strength reduction**

Loop-invariant

In computer programming, loop-invariant code consists of statements or expressions (in an imperative programming language) which can be moved outside the body of a loop without affecting the semantics of the program. **Loop-invariant code motion** (also called **hoisting** or **scalar promotion**) is a compiler optimization which performs this movement automatically.

`for (int i = 0; i < n; ++i) {`
`  x = y + z;`
`  a[i] = 6 * i + x * x;`
`}`
	`x = y + z;`
	`t1 = x * x;`
	`for (int i = 0; i < n; ++i) {`
	`  a[i] = 6 * i + t1;`
	`}`

The calculation `x = y + z` and `x * x` can be moved outside the loop since within they are loop-invariant code This code can be optimized further. For example, strength reduction could remove the two multiplications inside the loop (`6*i` and `a[i]`), and induction variable elimination could then elide `i` completely. Since `6 * i` must be in lock step with `i` itself, there is no need to have both.	

Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Code Before Optimization	Code After Optimization

#include<stdio.h>
int main(void)
{
for (int i=0; i<5; i++)
printf("Hello\n"); //print hello 5 times
 	
return 0;
}
	#include<stdio.h>
	
	int main(void)
	{
	// unrolled the for loop in program 1
	printf("Hello\n");
	printf("Hello\n");
	printf("Hello\n");
	printf("Hello\n");
	printf("Hello\n");
	
	return 0;
	}

Advantages:

- Increases program efficiency.
- Reduces loop overhead.
- If statements in loop are not dependent on each other, they can be executed in parallel.

Disadvantages:

- Increased program code size, which can be undesirable.
- Possible increased usage of register in a single iteration to store temporary variables which may reduce performance.
- Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

Loop Fusion

Loop fusion (or loop jamming) is a compiler optimization and loop transformation which replaces multiple loops with a single one. It is possible when two loops iterate over the same range and do not reference each other's data. Loop fusion does not always improve run-time speed. On some architectures, two loops may actually perform better than one loop because, for example, there is increased data locality within each loop.

Code Before Optimization	Code After Optimization

int i, a[100], b[100];
for (i = 0; i < 100; i++)
a[i] = 1;
for (i = 0; i < 100; i++)
b[i] = 2;
	int i, a[100], b[100];
	for (i = 0; i < 100; i++)
	{
	a[i] = 1;
	b[i] = 2;
	}

Loop Fission

In computer science, **loop fission** (or **loop distribution**) is a compiler optimization in which a loop is broken into multiple loops over the same index range with each taking only a part of the original loop's body. The goal is to break down a large loop body into smaller ones to achieve better utilization of locality of reference. This optimization is most efficient in multi-core processors that can split a task into multiple tasks for each processor.

Code Before Optimization	Code After Optimization

int i, a[100], b[100];
for (i = 0; i < 100; i++) {
a[i] = 1;
b[i] = 2;
}
	int i, a[100], b[100];
	for (i = 0; i < 100; i++) {
	a[i] = 1;
	}
	for (i = 0; i < 100; i++) {
	b[i] = 2;
	}

Strength reduction

Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

Example:

Code Before Optimization	Code After Optimization

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

In the above code, it is cheaper to compute s=s+6 than j=3 *i	

Example-

Code before Optimization	Code after Optimization
B = A x 2	B = A + A

Here, the expression "A x 2" has been replaced with the expression "A + A" because the cost of multiplication operator is higher than the cost of addition operator.

Peephole Optimization

A statement-by-statement code-generations strategy often produces 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 be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristics of peephole optimizations:

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

Redundant Loads and Stores:
If we see the instructions sequence
(1) MOV R0,a
(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.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).

Or
This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination
At source code level, the following can be done by the user:

```
int add_ten(int x)
  {
  int y, z;
  y = 10;
  z = x + y;
  return z;
  }
```

```
int add_ten(int x)
  {
  int y;
  y = 10;
  y = x + y;
  return y;
  }
```

```
int add_ten(int x)
  {
  int y = 10;
  return x + y;
  }
```

```
int add_ten(int x)
  {
  return x + 10;
  }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

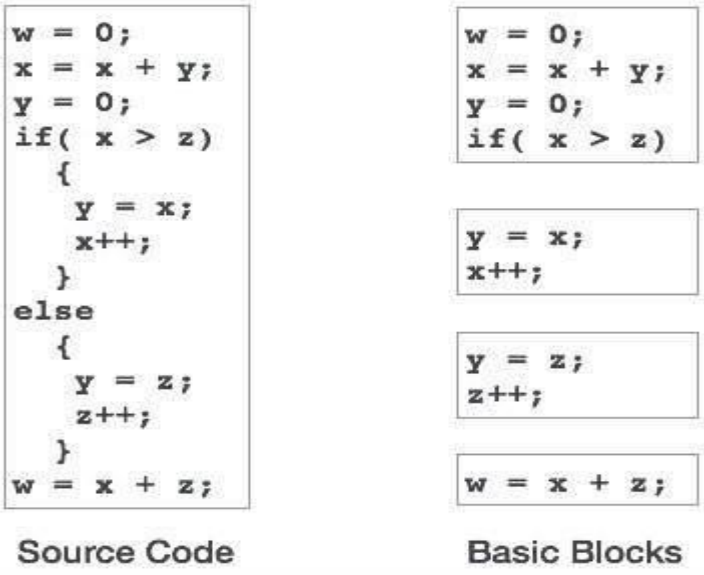
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
  - First statement of a program.
  - Statements that are target of any branch (conditional/unconditional).
  - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

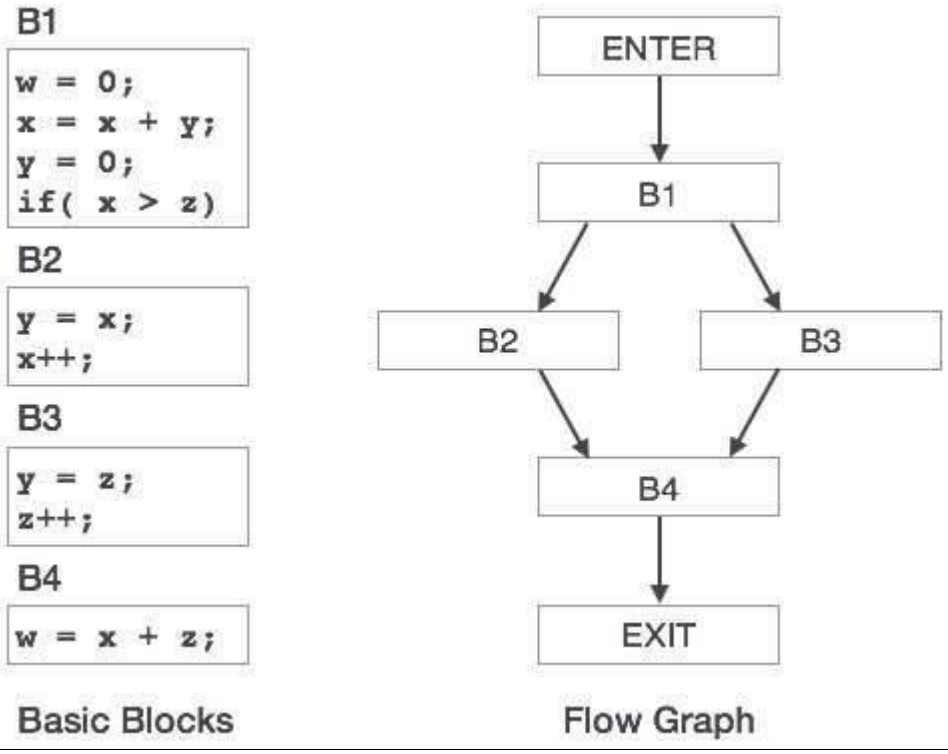
Basic blocks are important concepts from both code generation and optimization point of view.



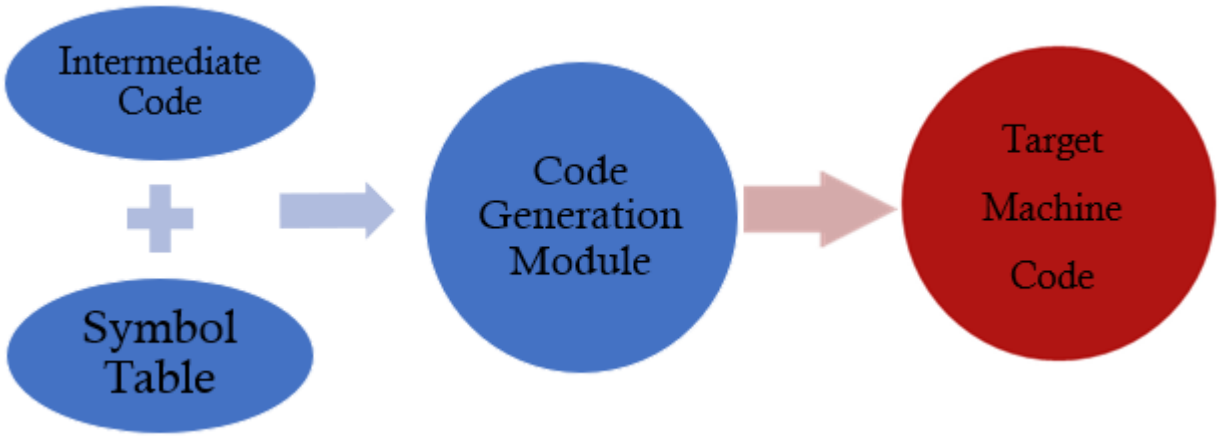
Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



Target Code Generation



A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Input to code generator –**
- **Target program –**
- **Memory Management –**
- **Instruction selection –**
- **Register allocation issues –**
- **Evaluation order –**

Input to code generator –
The input to 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, DAG's etc. Assume that they are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

Target program –
Target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

1. Absolute machine language as an output has advantages that it can be placed in a fixed memory location and can be immediately executed.
2. Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader.
3. Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.

Memory Management

- Mapping names in the source program to addresses of data objects in run-time memory is done cooperatively by the front end and the code generator.
- A name in a three- address statement refers to a symbol-table entry for the name.

- From the symbol-table information, a relative address can be determined for the name in a data area for the procedure.

Instruction selection –

- The factors to be considered during instruction selection are:
  - The uniformity and completeness of the instruction set.
  - Instruction speed and machine idioms.
  - Size of the instruction set.
- Eg., for the following address code is:

  a := b + c
  d := a + e

  inefficient assembly code is:

  MOV b, R_0 $R_0 \leftarrow b$
  ADD c, R_0 $R_0 \leftarrow c + R_0$
  MOV R_0, a a $\leftarrow R_0$
  MOV a, R_0 $R_0 \leftarrow a$
  ADD e, R_0 $R_0 \leftarrow e + R_0$
  MOV R_0 , d d $\leftarrow R_0$

  Here the fourth statement is redundant, and so is the third statement if 'a' is not subsequently used.

Register allocation issues –

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

1. During **Register allocation** – we select only those set of variables that will reside in the registers at each point in the program.
2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increase, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

M a, b

These types of multiplicative instruction involve register pairs where a, the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

Evaluation order –

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.

Approaches to code generation issues: 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
- Maintainable

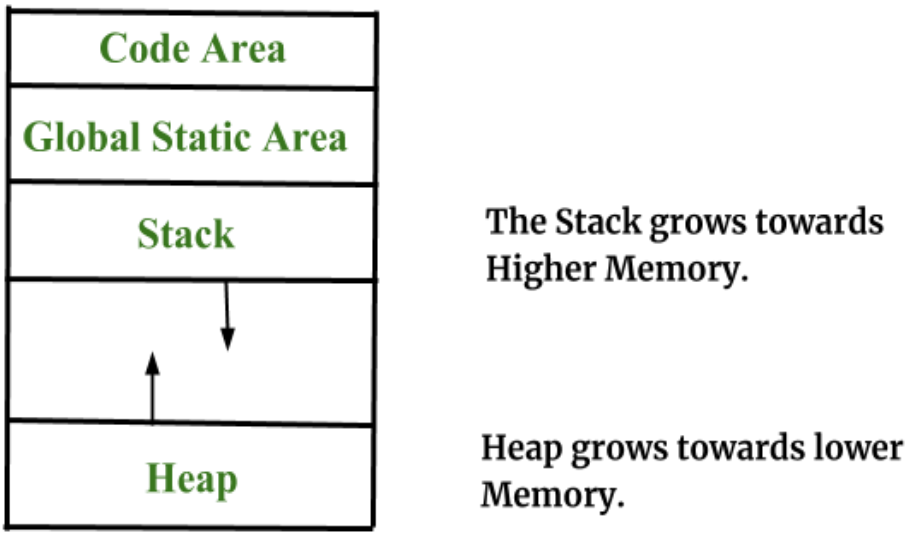
Run-time Environment

Compiler must cooperate with OS and other system software to support implementation of different abstractions (names, scopes, bindings, data types, operators, procedures, parameters, flow-of-control) on the target machine

Compiler does this by Run-Time Environment in which it assumes its target programs are being executed

Storage Allocation Strategies:

- Static data objects
- Dynamic data objects- heap
- Automatic data objects- stack



STORAGE ALLOCATION TECHNIQUES

I. Static Storage Allocation

- For any program if we create memory at compile time, memory will be created in the static area.
- For any program if we create memory at compile time only, memory is created only once.
- It don't support dynamic data structure i.e memory is created at compile time and deallocated after program completion.
- The drawback with static storage allocation is recursion is not supported.
- Another drawback is size of data should be known at compile time

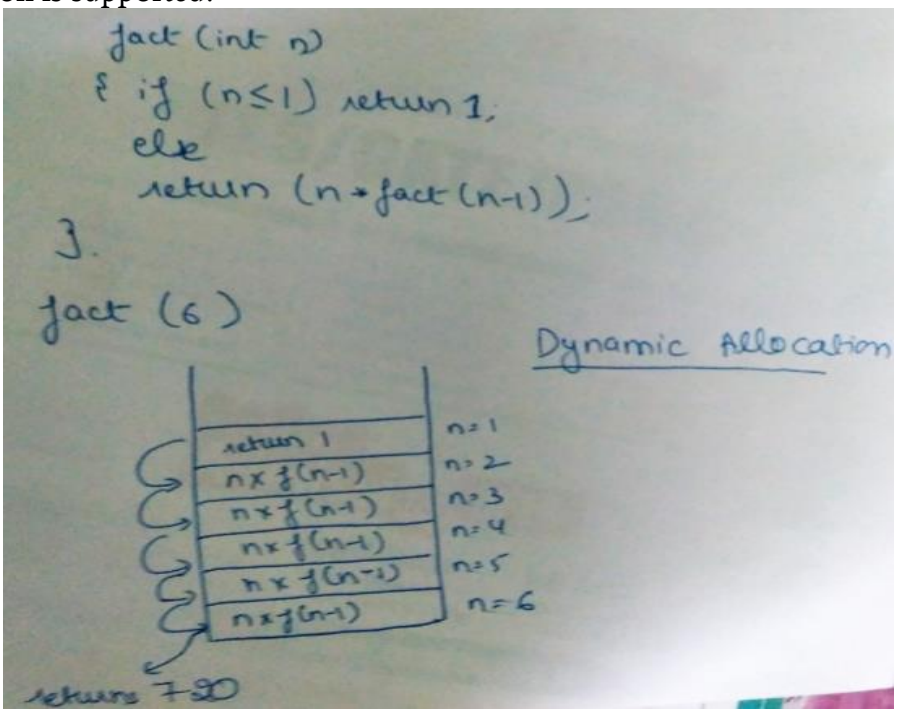
Eg- FORTRAN was designed to permit static storage allocation.

II. Stack Storage Allocation

- Storage is organised as a stack and activation records are pushed and popped as activation begin and end respectively. Locals are contained in activation records so they are bound to fresh storage in each activation.
- Recursion is supported in stack allocation

III. Heap Storage Allocation

- Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Recursion is supported.

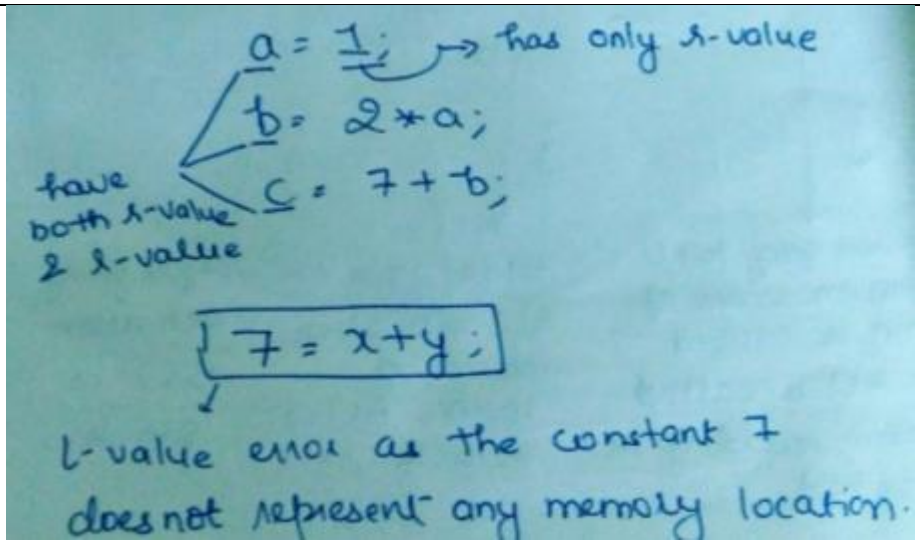


Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

Basic terminology:

- **R- value:** The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if its appear on the right side of the assignment operator. R-value can always be assigned to some other variable.
- **L-value:** The location of the memory(address) where the expression is stored is known as the l-value of that expression. It always appears on the left side if the assignment operator.



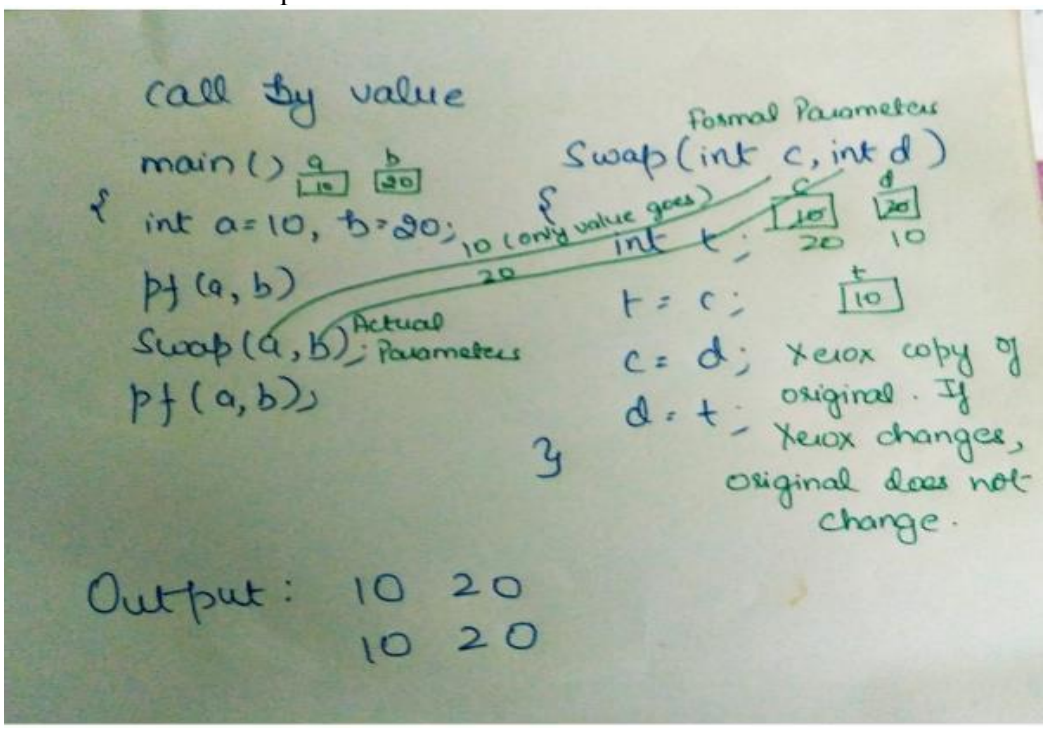
i. Formal Parameter: Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

ii. Actual Parameter: Variables whose values and functions are passed to the called function are called actual parameters. These variables are specified in the function call as arguments.

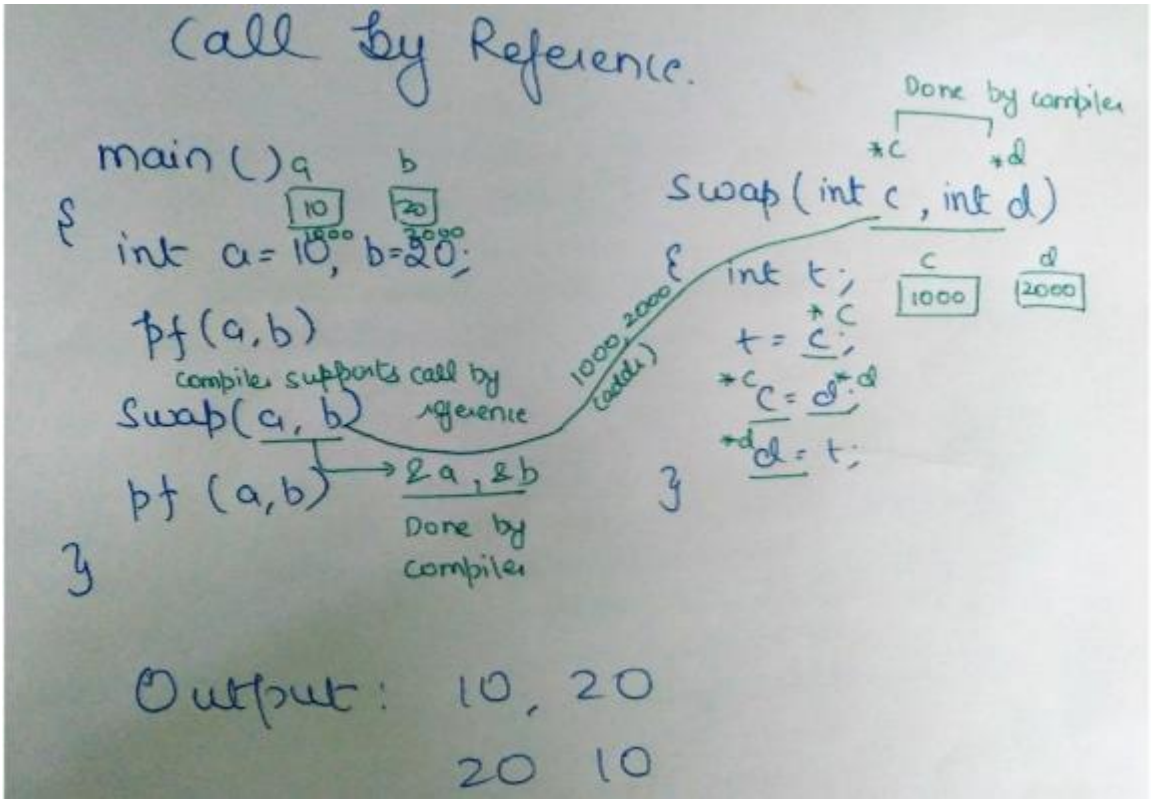
Different ways of passing the parameters to the procedure

- **Call by Value**

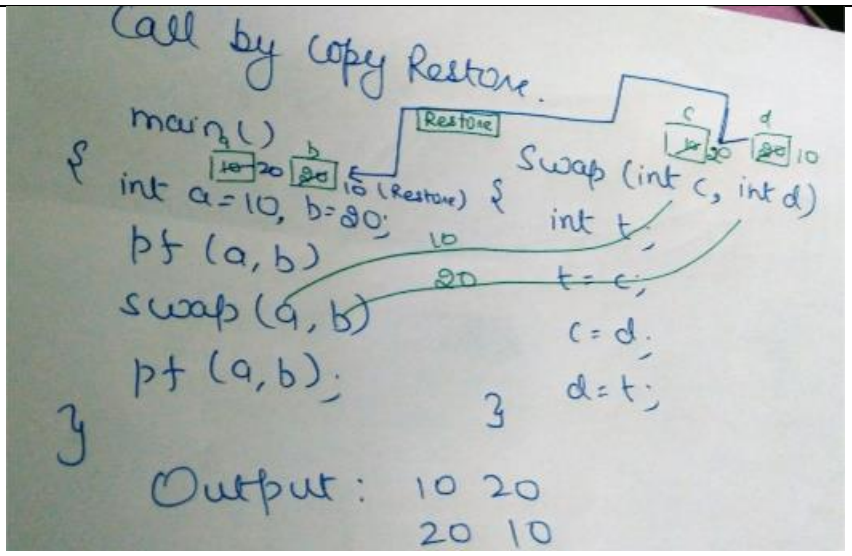
  In call by value the calling procedure pass the r-value of the actual parameters and the compiler puts that into called procedure's activation record. Formal parameters hold the values passed by the calling procedure, thus any changes made in the formal parameters does not affect the actual parameters.



- **Call by Reference** In call by reference the formal and actual parameters refers to same memory location. The l-value of actual parameters is copied to the activation record of the called function. Thus, the called function has the address of the actual parameters. If the actual parameters do not have a l-value (eg- i+3) then it is evaluated in a new temporary location and the address of the location is passed. Any changes made in the formal parameter is reflected in the actual parameters (because changes are made at the address).



- **Call by Copy Restore**
  In call by copy restore compiler copies the value in formal parameters when the procedure is called and copy them back in actual parameters when control returns to the called function. The r-values are passed and on return r-value of formals are copied into l-value of actuals.



- **Call by Name**
  In call by name the actual parameters are substituted for formals in all the places formals occur in the procedure. It is also referred as lazy evaluation because evaluation is done on parameters only when needed.