UNIT-IV

Run time storage: Storage organization, storage allocation strategies scope access to now local names, parameters, language facilities for dynamics storage allocation.
Code optimization: Principal sources of optimization, optimization of basic blocks, peephole optimization, flow graphs, Data flow analysis of flow graphs.

Run time storage

The run-time environment is the structure of the target computers registers and memory that serves to manage memory and maintain information needed to guide a programs execution process.

Types of Runtime Environments –

Fully Static:

Fully static runtime environment may be useful for the languages in which pointers or dynamic allocation is not possible in addition to no support for recursive function calls.

- Every procedure will have only one activation record which is allocated before execution.
- Variables are accessed directly via fixed address.
- Little bookkeeping overhead; i.e., at most return address may have to be stored in activation record.
- The calling sequence involves the calculation of each argument address and storing into its appropriate parameter location and saving the return address and then a jump is made.

Stack Based:

In this, activation records are allocated (push of the activation record) whenever a function call is made. The necessary memory is taken from the stack portion of the program. When program execution return from the function the memory used by the activation record is deallocated (pop of the activation record). Thus, the stack grows and shrinks with the chain of function calls.

Fully Dynamic:

Functional language such as Lisp, ML, etc. use this style of call stack management. Silently, here activation record is deallocated only when all references to them have disappeared, and this requires the activation records to dynamically freed at arbitrary times during execution. Memory manager (garbage collector) is needed.

The data structure that handles such management is heap an this is also called as Heap Management.

Activation Record –

Information needed by a single execution of a procedure is managed using a contiguous block of storage called “activation record”.

An activation record is allocated when a procedure is entered and it is deallocated when that procedure is exited. It contain temporary data, local data, machine status, optional access link, optional control link, actual parameters and returned values.

- **Program Counter (PC) –** whose value is the address of the next instruction to be executed.
- **Stack Pointer (SP) –** whose value is the top of the (top of the stack, ToS).
- **Frame pointer (FP) –** which points to the current activation.

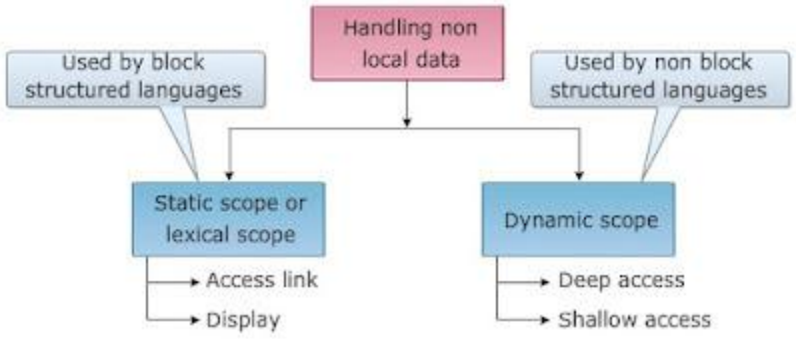
Storage allocation strategies scope access to now local names

In some cases, when a procedure refer to variables that are not local to it, then such variables are called nonlocal variables

There are two types of scope rules, for the non-local names. They are

Static scope

Dynamic scope


Handling non local data
Used by block structured languages
Used by non block structured languages
Static scope or lexical scope
Dynamic scope
Access link
Display
Deep access
Shallow access

Static Scope or Lexical Scope

Lexical scope is also called static scope. In this type of scope, the scope is verified by examining the text of the program.

Examples: PASCAL, C and ADA are the languages that use the static scope rule.

These languages are also called block structured languages

Block

A block defines a new scope with a sequence of statements that contains the local data declarations. It is enclosed within the delimiters.

Example:

{

Declaration statements

……….

}

The beginning and end of the block are specified by the delimiter. The blocks can be in nesting fashion that means block $_{B2}$ completely can be inside the block B_1

In a block structured language, scope declaration is given by static rule or most closely nested loop

At a program point, declarations are visible

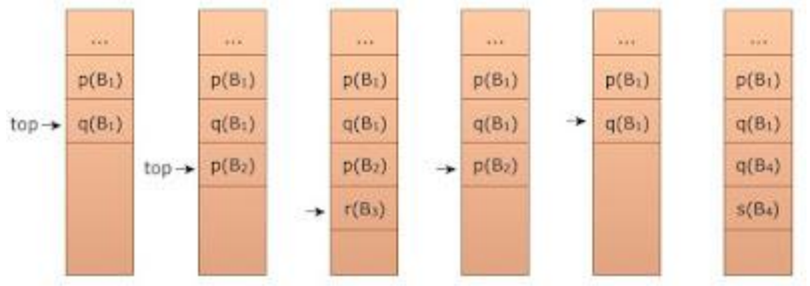
The declarations that are made inside the procedure

The names of all enclosing procedures

The declarations of names made immediately within such procedures

The displayed image on the screen shows the storage for the names corresponding to particular block

Thus, block structure storage allocation can be done by stack

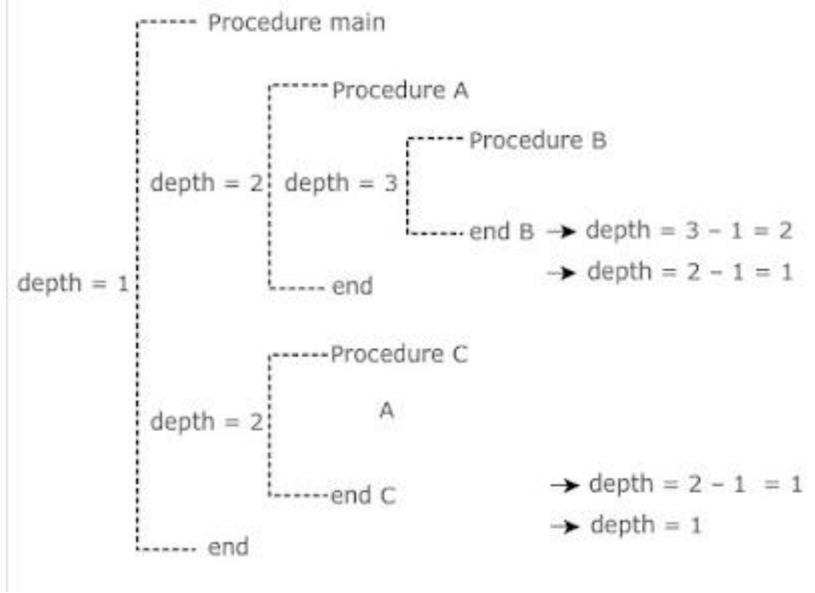


Lexical Scope for Nested Procedure

If a procedure is declared inside another procedure then that procedure is known as nested procedure

A procedure pi, can call any procedure, i.e., its direct ancestor or older siblings of its direct ancestor

Procedure main
Procedure P1
Procedure P2
Procedure P3
Procedure P4



Nesting Depth:

Lexical scope can be implemented by using nesting depth of a procedure. The procedure of calculating nesting depth is as follows:

The main programs nesting depth is ‘1’
When a new procedure begins, add ‘1’ to nesting depth each time
When you exit from a nested procedure, subtract ‘1’ from depth each time
The variable declared in specific procedure is associated with nesting depth

Static Scope or Lexical Scope

The lexical scope can be implemented using access link and displays.

Access Link:

Access links are the pointers used in the implementation of lexical scope which is obtained by using pointer to each activation record

If procedure p is nested within a procedure q then access link of p points to access link or most recent activation record of procedure q

Example: Consider the following piece of code and the runtime stack during execution of the program

```
program test;
var a: int;
procedure A;
var d: int;
{
        a := 1,
}
procedure B(i: int);
var b : int;
procedure C;
var k : int;
{
        A;
}
{
        if(i<>0) then B(i-1)
        else C;
}
{
        B(1);
}
```

Displays:

If access links are used in the search, then the search can be slow

So, optimization is used to access an activation record from the direct location of the variable without any search

Display is a global array d of pointers to activation records, indexed by lexical nesting depth. The number of display elements can be known at compiler time

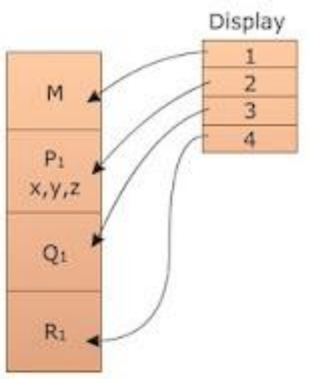
d[i] is an array element which points to the most recent activation of the block at nesting depth (or lexical level)

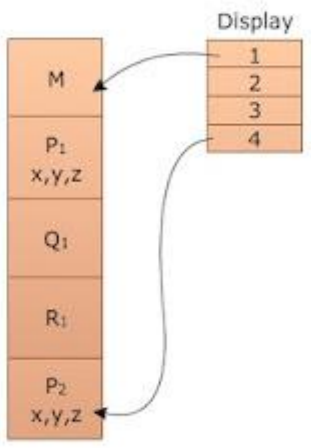
A nonlocal X is found in the following manner:

Use one array access to find the activation record containing X. if the most-closely nested declaration of X is at nesting depth I, the d[i] points to the activation record containing the location for X

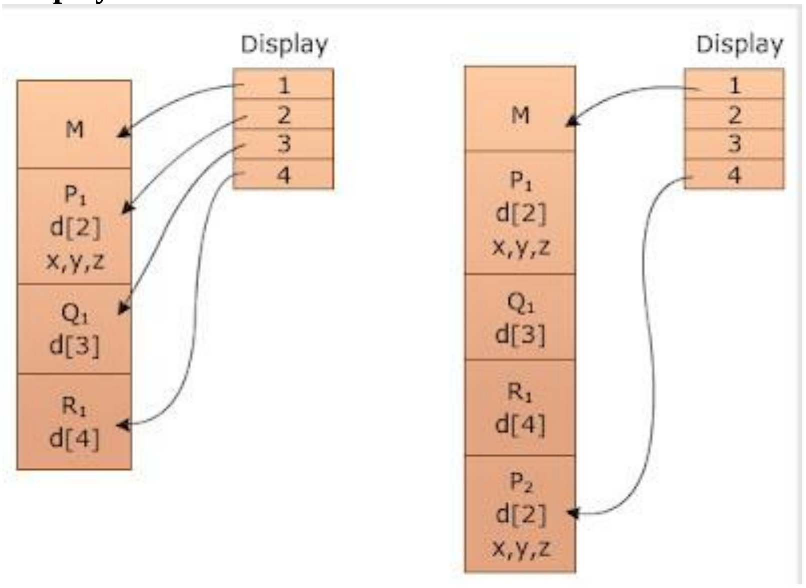
Use relative address within the activation record

Example:





How to maintain display information?



When a procedure is called, a procedure ‘p’ at nesting depth ‘i’ is setup:
    Save value of d[i] in activation record for ‘p’
    ‘I’ set d[i] to point to new activation record
When a ‘p’ returns:
    Reset d[i] to display value stored

Parameters

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.

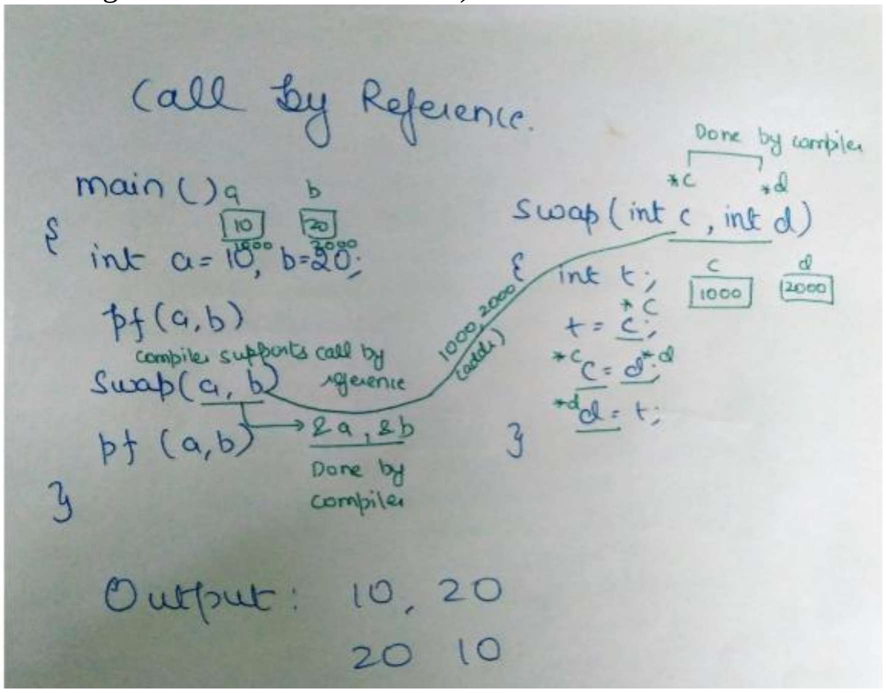
Different ways of passing the parameters to the procedure

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.

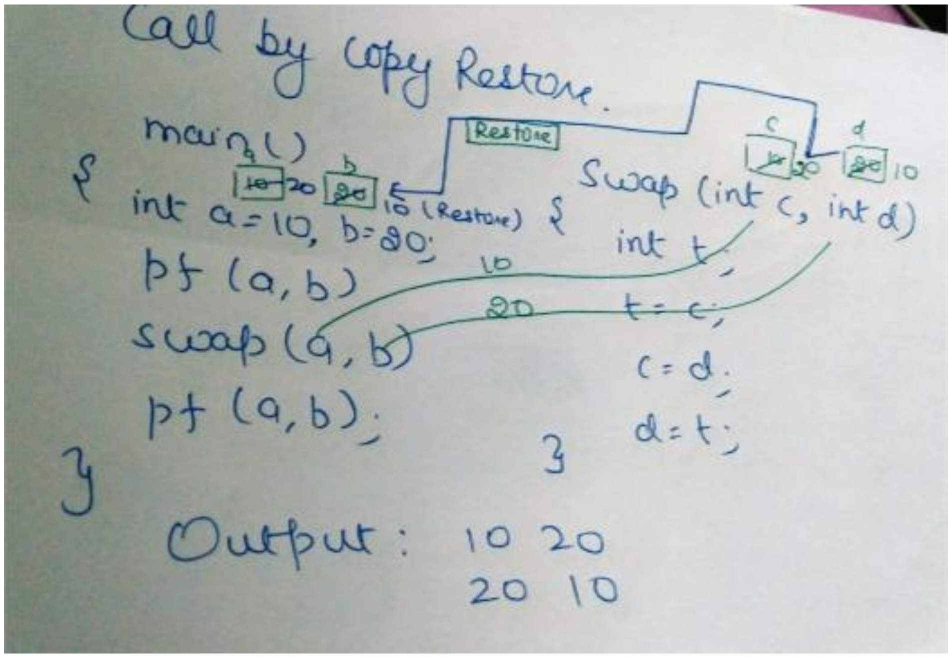
Call by Reference

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 does 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.



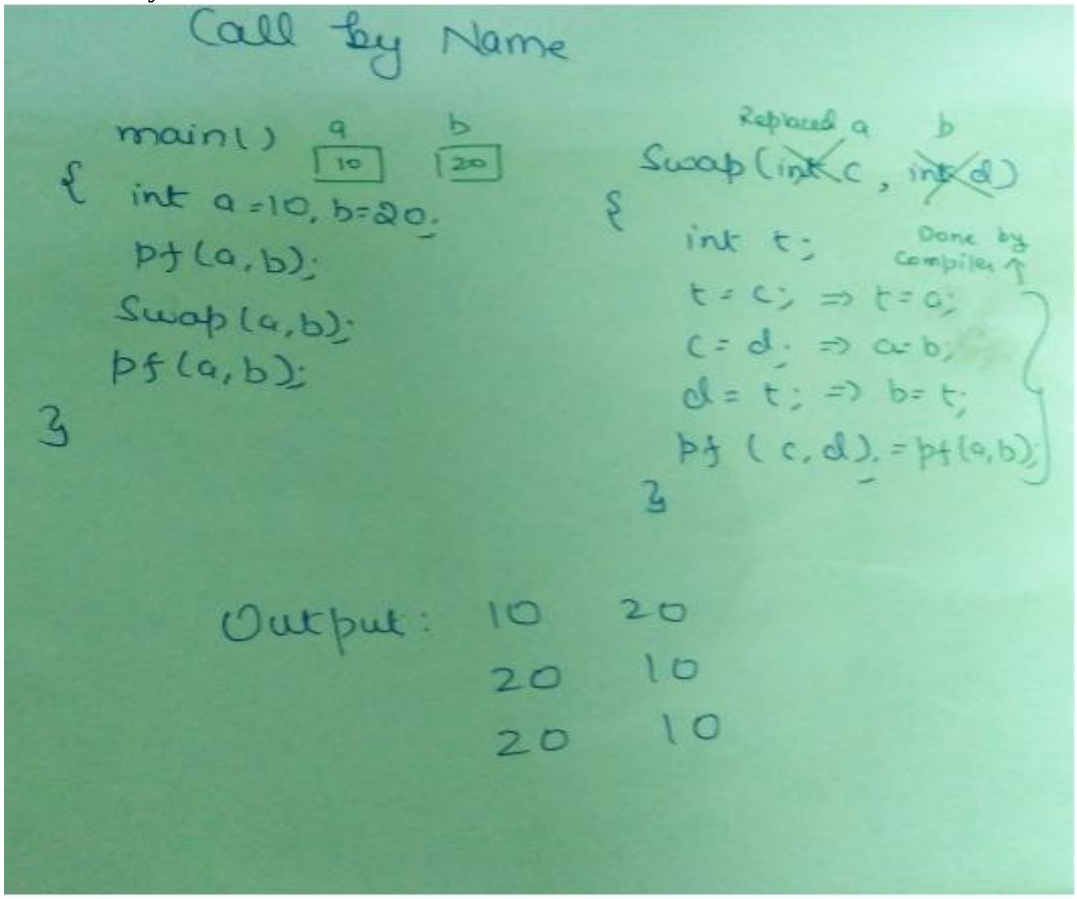
Call by Copy Restore.

```
main()                          Swap (int c, int d)
{                               {
int a=10, b=20;                   int t;
pf (a,b)                          t=c;
swap(a,b)                         c=d;
pf (a,b);                         d=t;
}                               }

Output: 10 20
        20 10
```

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.



Call by Name

```
main()                     Swap(int c, int d)
{                          {
 int a=10, b=20;             int t;
 pf(a,b);                    t=c;  => t=a;
 Swap(a,b);                  c=d;  => a=b;
 pf(a,b);                    d=t;  => b=t;
}                            pf (c,d); = pf(a,b);
                           }

Output: 10 20
        20 10
        20 10
```

Language facilities for dynamics storage allocation

a. Explicit and Implicit allocation of memory
b. Garbage Collection

- **Explicit and Implicit allocation of memory to variables**
  - Most languages support dynamic allocation of memory.
  - Pascal supports new(p) and dispose(p) for pointer types.
  - C provides malloc() and free() in the standard library.
  - C++ provides the new and free operators.
  - These are all examples of EXPLICIT allocation.
  - Other languages like Python and Lisp have IMPLICIT allocation.
- **Garbage** – Finding variables that are not referred by the program any more.
  - In languages with explicit deallocation, the programmer must be careful to free every dynamically allocated variable, or GARBAGE will accumulate.
  - Garbage is dynamically allocated memory that is no longer accessible because no pointers are pointing to it.
  - In some languages with implicit deallocation, GARBAGE COLLECTION is occasionally necessary.
  - Other languages with implicit deallocation carefully track references to allocated memory and automatically free memory when nobody refers to it any longer.

Principal sources of 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. 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.

When to Optimize?

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.

Why Optimize?

Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So 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.

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.

Code Optimization is done in the following different ways:

1. Compile Time Evaluation:

(i) A = 2*(22.0/7.0)*r
Perform 2*(22.0/7.0)*r at compile time.

(ii) x = 12.4
y = x/2.3
Evaluate x/2.3 as 12.4/2.3 at compile time.

2. Variable Propagation:

```
//Before Optimization
c = a * b
x = a
till
d = x * b + 4

//After Optimization
c = a * b
x = a
till
d = a * b + 4
```

3. Constant Propagation:

If the value of a variable is a constant, then replace the variable with the constant. The variable may not always be a constant.

Example:

(i) A = 2*(22.0/7.0)*r
Performs 2*(22.0/7.0)*r at compile time.

(ii) x = 12.4
y = x/2.3
Evaluates x/2.3 as 12.4/2.3 at compile time.

(iii) int k=2;
if(k) go to L3;
It is evaluated as :
go to L3 (Because k = 2 which implies condition is always true)

4. Constant Folding:

Consider an expression: a = b op c and the values b and c are constants, then the value of a can be computed at compile time.

Example:

#define k 5

x = 2 * k
y = k + 5

This can be computed at compile time and the values of x and y are :
x = 10
y = 10

Note: Difference between Constant Propagation and Constant Folding:

- In Constant Propagation, the variable is substituted with its assigned constant where as in Constant Propagation; the variables whose values can be computed at compile time are considered and computed.

5. Copy Propagation:

- It is extension of constant propagation.
- After a is assigned to x, use a to replace x till a is assigned again to another variable or value or expression.
- It helps in reducing the compile time as it reduces copying.

Example :

```
//Before Optimization
c = a * b
x = a
till
d = x * b + 4

//After Optimization
c = a * b
x = a
till
d = a * b + 4
```

6. Common Sub Expression Elimination:

- In the above example, a*b and x*b is a common sub expression.

7. Dead Code Elimination:

- Copy propagation often leads to making assignment statements into dead code.
- A variable is said to be dead if it is never used after its last definition.
- In order to find the dead variables, a data flow analysis should be done.

Example:

```
c = a * b
x = a
till
d = a * b + 4

//After elimination :
c = a * b
till
d = a * b + 4
```

8. Unreachable Code Elimination:

- First, Control Flow Graph should be constructed.
- The block which does not have an incoming edge is an Unreachable code block.

- After constant propagation and constant folding, the unreachable branches can be eliminated.

9. Function Inlining:

- Here, a function call is replaced by the body of the function itself.
- This saves a lot of time in copying all the parameters, storing the return address, etc.

10. Function Cloning:

- Here, specialized codes for a function are created for different calling parameters.
- **Example:** Function Overloading

11. Induction Variable and Strength Reduction:

- An induction variable is used in the loop for the following kind of assignment i = i + constant. It is a kind of Loop Optimization Technique.
- Strength reduction means replacing the high strength operator with a low strength.

Examples:

Example 1:

Multiplication with powers of 2 can be replaced by shift right operator which is less expensive than multiplication

```
a=a*16
// Can be modified as :
a = a<<4
```

Example 2:

```
i = 1;
while (i<10)
{
   y = i * 4;
}
//After Reduction
i = 1
t = 4
{
  while( t<40)
  y = t;
  t = t + 4;
}
```

Loop Optimization Techniques:

1. Code Motion or Frequency Reduction:

- The evaluation frequency of expression is reduced.
- The loop invariant statements are brought out of the loop.

Example:

```
a = 200;
 while(a>0)
 {
     b = x + y;
     if (a % b == 0}
     printf("%d", a);
    }
```

```
//This code can be further optimized as
a = 200;
b = x + y;
while(a>0)
 {
    if (a % b == 0}
    printf("%d", a);
  }
```

2. Loop Jamming:

- Two or more loops are combined in a single loop. It helps in reducing the compile time.

Example:

```
// Before loop jamming
for(int k=0;k<10;k++)
{
 x = k*2;
}

for(int k=0;k<10;k++)
{
 y = k+3;
}

//After loop jamming
for(int k=0;k<10;k++)
{
 x = k*2;
 y = k+3;
}
```

3. Loop Unrolling:

- It helps in optimizing the execution time of the program by reducing the iterations.
- It increases the program's speed by eliminating the loop control and test instructions.

Example:

```
//Before Loop Unrolling

for(int i=0;i<2;i++)
{
  printf("Hello");
}

//After Loop Unrolling

printf("Hello");
printf("Hello");
```

Where to apply Optimization?

Now that we learned the need for optimization and its two types,now let’s see where to apply these optimization.

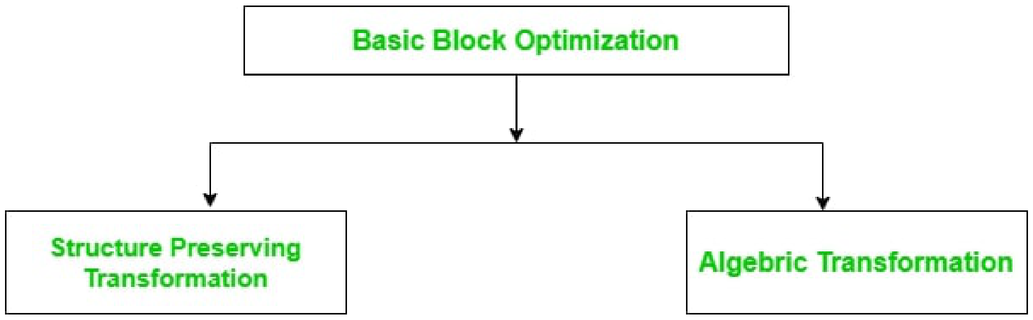
- **Source program:** Optimizing the source program involves making changes to the algorithm or changing the loop structures. The user is the actor here.
- **Intermediate Code:** Optimizing the intermediate code involves changing the address calculations and transforming the procedure calls involved. Here compiler is the actor.
- **Target Code:** Optimizing the target code is done by the compiler. Usage of registers, and select and move instructions are part of the optimization involved in the target code.
- **Local Optimization:** Transformations are applied to small basic blocks of statements. Techniques followed are Local Value Numbering and Tree Height Balancing.
- **Regional Optimization:** Transformations are applied to Extended Basic Blocks. Techniques followed are Super Local Value Numbering and Loop Unrolling.
- **Global Optimization:** Transformations are applied to large program segments that include functions, procedures, and loops. Techniques followed are Live Variable Analysis and Global Code Replacement.
- **Inter-procedural Optimization:** As the name indicates, the optimizations are applied inter procedurally. Techniques followed are Inline Substitution and Procedure Placement.

Optimization of basic blocks

Optimization is applied to the basic blocks after the intermediate code generation phase of the compiler. Optimization is the process of transforming a program that improves the code by consuming fewer resources and delivering high speed. In optimization, high-level codes are replaced by their equivalent efficient low-level codes. Optimization of basic blocks can be machine-dependent or machine-independent. These transformations are useful for improving the quality of code that will be ultimately generated from basic block.

There are two types of basic block optimizations:

1. Structure preserving transformations
2. Algebraic transformations



Structure-Preserving Transformations:

The structure-preserving transformation on basic blocks includes:

1. Dead Code Elimination
2. Common Sub-expression Elimination
3. Renaming of Temporary variables
4. Interchange of two independent adjacent statements

1. Dead Code Elimination:

Dead code is defined as that part of the code that never executes during the program execution. So, for optimization, such code or dead code is eliminated. The code which is never executed during the program (Dead code) takes time so, for optimization and speed, it is eliminated from the code. Eliminating the dead code increases the speed of the program as the compiler does not have to translate the dead code.

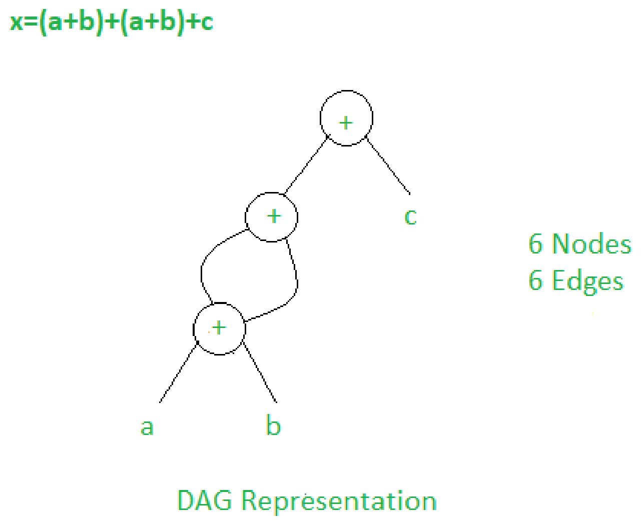
Example:

```
// Program with Dead code
int main()
{
   x = 2
   if (x > 2)
     cout << "code"; // Dead code
   else
     cout << "Optimization";
   return 0;
}
// Optimized Program without dead code
int main()
{
   x = 2;
   cout << "Optimization"; // Dead Code Eliminated
   return 0;
}
```

2. Common Sub-expression Elimination

In this technique, the sub-expression which are common are used frequently are calculated only once and reused when needed. DAG (Directed Acyclic Graph) is used to eliminate common subexpressions.

Example:



3. Renaming of Temporary Variables:

Statements containing instances of a temporary variable can be changed to instances of a new temporary variable without changing the basic block value.

Example: Statement t = a + b can be changed to x = a + b where t is a temporary variable and x is a new temporary variable without changing the value of the basic block.

4. Interchange of Two Independent Adjacent Statements:
If a block has two adjacent statements, which are independent can be interchanged without affecting the basic block value.
Example:
t1 = a + b
t2 = c + d
These two independent statements of a block can be interchanged without affecting the value of the block.
Algebraic Transformation:
Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set. Some of the algebraic transformation on basic blocks includes:

1. Constant Folding
2. Copy Propagation
3. Strength Reduction

1. Constant Folding:
Solve the constant terms, which are continuous so that compiler does not need to solve this expression.
Example:
x = 2 * 3 + y ⇒ x = 6 + y (Optimized code)
2. Copy Propagation:
It is of two types, Variable Propagation, and Constant Propagation.
Variable Propagation:
x = y ⇒ z = y + 2 (Optimized code)
z = x + 2
Constant Propagation:
x = 3 ⇒ z = 3 + a (Optimized code)
z = x + a
3. Strength Reduction:
Replace expensive statement/ instruction with cheaper ones.
x = 2 * y (costly) ⇒ x = y + y (cheaper)
x = 2 * y (costly) ⇒ x = y << 1 (cheaper)
Loop Optimization:
Loop optimization includes the following strategies:

1. Code motion & Frequency Reduction
2. Induction variable elimination
3. Loop merging/combining
4. Loop Unrolling

1. Code Motion & Frequency Reduction
Move loop invariant code outside of the loop.

```
// Program with loop variant inside loop
int main()
{
   for (i = 0; i < n; i++) {
      x = 10;
      y = y + i;
```

```
    }
    return 0;
}
// Program with loop variant outside loop
int main()
{
    x = 10;
    for (i = 0; i < n; i++)
        y = y + i;
    return 0;
}
```

2. Induction Variable Elimination:

Eliminate various unnecessary induction variables used in the loop.

```
// Program with multiple induction variables
int main()
{
    i1 = 0;
    i2 = 0;
    for (i = 0; i < n; i++) {
        A[i1++] = B[i2++];
    }
    return 0;
}
// Program with one induction variable
int main()
{
    for (i = 0; i < n; i++) {
        A[i++] = B[i++]; // Only one induction variable
    }
    return 0;
}
```

3. Loop Merging/Combining:

If the operations performed can be done in a single loop then, merge or combine the loops.

```
// Program with multiple loops
int main()
{
    for (i = 0; i < n; i++)
        A[i] = i + 1;
    for (j = 0; j < n; j++)
        B[j] = j - 1;
}
return 0;
}
// Program with one loop when multiple loops are merged
int main()
{
    for (i = 0; i < n; i++) {
        A[i] = i + 1;
        B[i] = i - 1;
    }
    return 0;
}
```

4. Loop Unrolling:

If there exists simple code which can reduce the number of times the loop executes then, the loop can be replaced with these codes.

```
// Program with loops
int main()
{
    for (i = 0; i < 3; i++)
        cout << "Cd";
    return 0;
}
// Program with simple code without loops
int main()
{
    cout << "Cd";
    cout << "Cd";
    cout << "Cd";
    return 0;
}
```

Peephole optimization

Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the 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.

Characteristics of Peephole Optimization So The peephole optimization can be applied to the target code using the following characteristic.

1. Redundant instruction elimination

Especially the redundant loads and stores can be eliminated in this type of transformations.

Example: MOV R0,x MOV x,R0

We can eliminate the second instruction since x is in already R0. But if MOV x, R0 is a label statement then we can not remove it.

2. Unreachable code

Especially the redundant loads and stores can be eliminated in this type of transformations.

An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate the sequence of instructions.

Example:

```
# define debug 0
```

```
if(debug)
{
   print debugging information
}
```

In the intermediate representation the if statement may be translated as if-

if debug=1 goto L1 goto L2 L1: print debugging information L2:

Additionally, One obvious peephole optimization is to eliminate jumps over jumps. Thus no matter what the value of debugging, can replaced by: If debug goto L2debug≠1

Print debugging information

L2: - Now, since debug set to 0 at the beginning of the program, constant propagation program, should replace by

If 0≠1 goto L2≠1

Print debugging information

L2: - As the argument of the first statement of evaluates to a constant true, it can replace by goto L2. - Then all the statement that prints debugging aids are manifestly unreachable and can manifestly eliminate one at a time.

3. The flow of control optimization

The unnecessary jumps can eliminate 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: goto L2 By the sequence Goto L2 …….

L1: goto L2

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

If a

……

L1: goto L2

Can replaced by

If a

……

L1: goto L2

4. Algebraic simplification

So Peephole optimisation is an effective technique for algebraic simplification.

The statements such as x = x + 0 or x := x* 1 can eliminated by peephole optimisation.

5. Reduction in strength

Certain machine instructions are cheaper than the other.

In order to improve the performance of the intermediate code, we can replace these instructions by equivalent cheaper instruction.

So For example, x2 is cheaper than x * x. Similarly, addition and subtraction are cheaper than multiplication and division. So we can add an effectively equivalent addition and subtraction for multiplication and division.

6. Machine idioms

So the target instructions have equivalent machine instructions for performing some have operations.

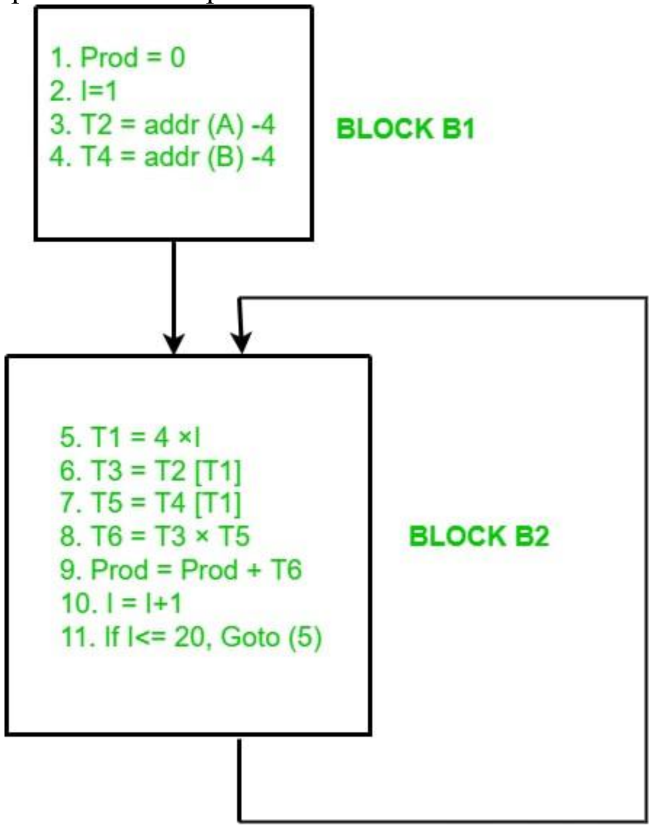
Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.

Example: Some machines have auto-increment or auto-decrement addressing modes. decrement These modes can use in a code for a statement like i=i+1.

Flow graphs

A flow graph is simply a directed graph. For the set of basic blocks, a flow graph shows the flow of control information. A control flow graph is used to depict how the program control is being parsed among the blocks. A flow graph is used to illustrate the flow of control between basic blocks once an intermediate code has been partitioned into basic blocks. When the beginning instruction of the Y block follows the last instruction of the X block, an edge might flow from one block X to another block Y.

Let’s make the flow graph of the example that we used for basic block formation:



Flow Graph for above Example

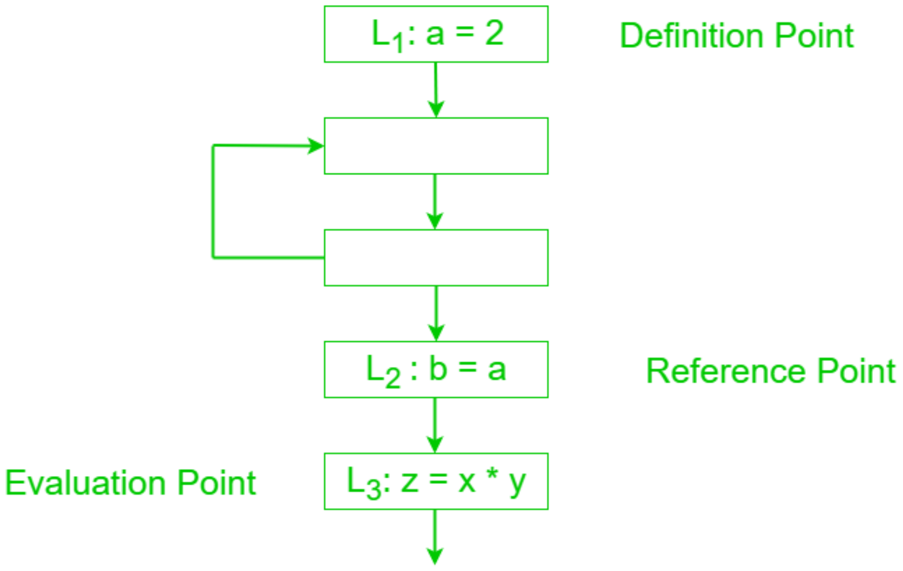
- Block B1 is the initial node. Block B2 immediately follows B1, so from B2 to B1 there is an edge.
- The target of jump from last statement of B1 is the first statement B2, so from B1 to B2 there is an edge.
- B2 is a successor of B1 and B1 is the predecessor of B2.

Data flow analysis of flow graphs.

It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis, optimization can be done. In general, its process in which values are computed using data flow analysis. The data flow property represents information that can be used for optimization.

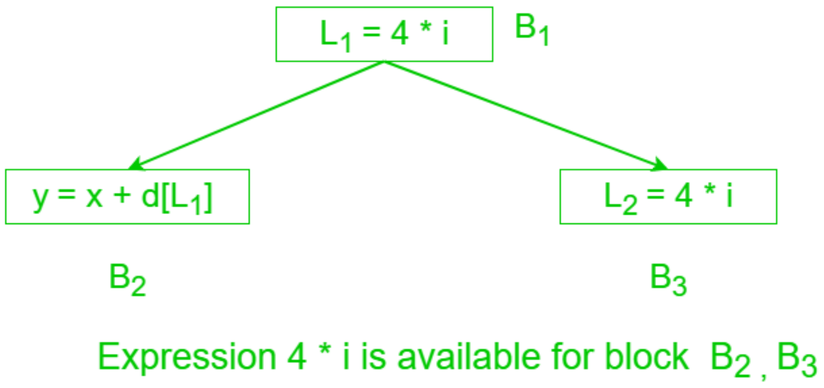
Basic Terminologies –

- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.



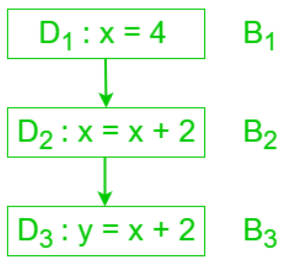
Data Flow Properties –

- **Available Expression** – A expression is said to be available at a program point x if along paths its reaching to x. A Expression is available at its evaluation point. An expression a+b is said to be available if none of the operands gets modified before their use.
  Example



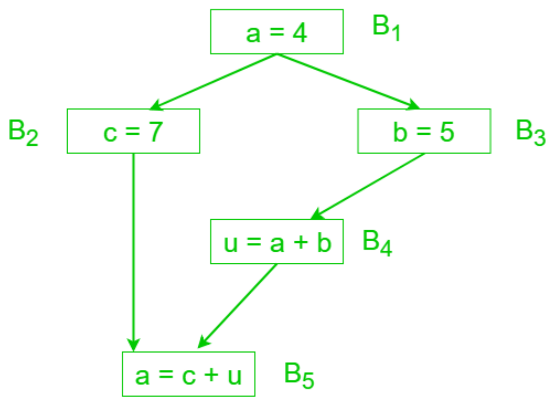
- **Advantage**
  It is used to eliminate common sub expressions.
- **Reaching Definition** – A definition D is reaches a point x if there is path from D to x in which D is not killed, i.e., not redefined.

Example



D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2

- **Advantage**
  It is used in constant and variable propagation.
- **Live variable –** A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.
  Example



a is live at block B_1 , B_3 , B_4 but killed at B_5

- **Advantage**
  1. It is useful for register allocation.
  2. It is used in dead code elimination.
- **Busy Expression –** An expression is busy along a path if its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.
  Advantage –
  It is used for performing code movement optimization.
  - To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
  - Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
  - For this kind of optimization user defined chaining is one particular problem.

- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

```
x = a + b;
x = 6 * 3
```

- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;

Some optimization needs information that is more global. For example, consider the following code:

```
a = 1;
   b = 2;
   c = 3;
   if (....) x = a + 5;
   else x = b + 4;
   c = x + 1;
```

In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

- Which variables are guaranteed to have constant values
- Which variables will be used before being redefined

Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.