UNIT-IV INTERMEDIATE CODE GENERATION

Translation of different language features:

A translator is a programming language processor that modifies a computer program from one language to another. It takes a program written in the source program and modifies it into a machine program. It can find and detect the error during translation.

There are various types of a translator which are as follows −

- **Compiler** − A compiler is a program that translates a high-level language (for example, C, C++, and Java) into a low-level language (object program or machine program). The compiler converts high-level language into the low-level language using various phases. A character stream inputted by the customer goes through multiple stages of compilation which at last will provide target language.
- **Pre-Processor** − Pre-Processor is a program that processes the source code before it passes through the compiler. It can perform under the control of what is referred to as pre-processor command lines or directives.
- **Assembler** − An assembler is a translator which translates an assembly language program into an equivalent machine language program of the computer. Assembler provides a friendlier representation than a computer 0's and 1's that simplifies writing and reading programs.

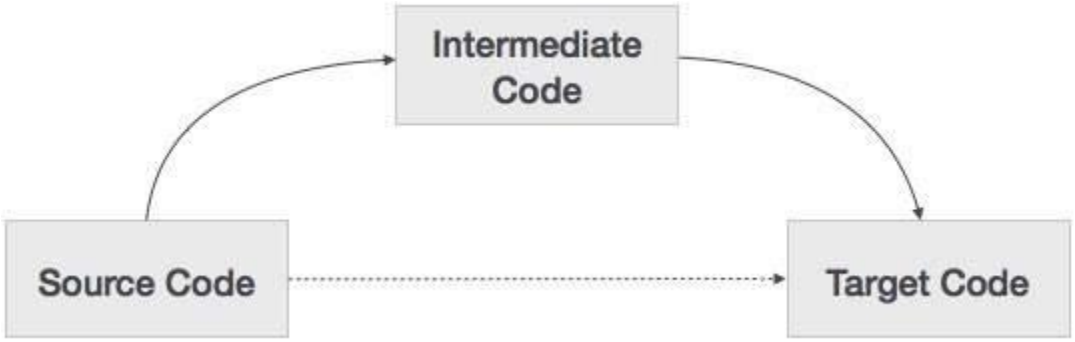
An assembler reads a single assembly language source document and creates an object document including machine instructions and bookkeeping data that supports to merge of various object files into a program.

- **Interpreter** − An interpreter is a program that executes the programming code directly rather than only translating it into another format. It translates and executes programming language statements one by one.
- **Macros** − Many assembly languages support a "macro" facility whereby a macro statement will translate into a sequence of assembly language statements and possibly other macro statements before being translated into machine code. Therefore, a macro facility is a text replacement efficiency.
- **Linker** − Linker is a computer program that connects and combines multiple object files to create an executable file. All these files might have been compiled by a separate assembler. The function of a linker is to inspect and find referenced module/routines in a program and to decide the memory location where these codes will be loaded creating the program instruction have an absolute reference.
- **Loader** − The loader is an element of the operating framework and is liable for loading executable files into memory and implement them. It can compute the size of a program (instructions and data) and generate memory space for it. It can initialize several registers to start execution.

It creates a new address space for the program. This address space is huge to influence the text and data segments, along with a stack segment. It can repeat instructions and data from the executable file into the new address space.

Intermediate Code Generation

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2

*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;
float b;

Allocation process:
{offset = 0}

  int a;
  id.type = int
  id.width = 2
```

```
offset = offset + id.width
{offset = 2}

  float b;
  id.type = float
  id.width = 4

offset = offset + id.width
{offset = 6}
```

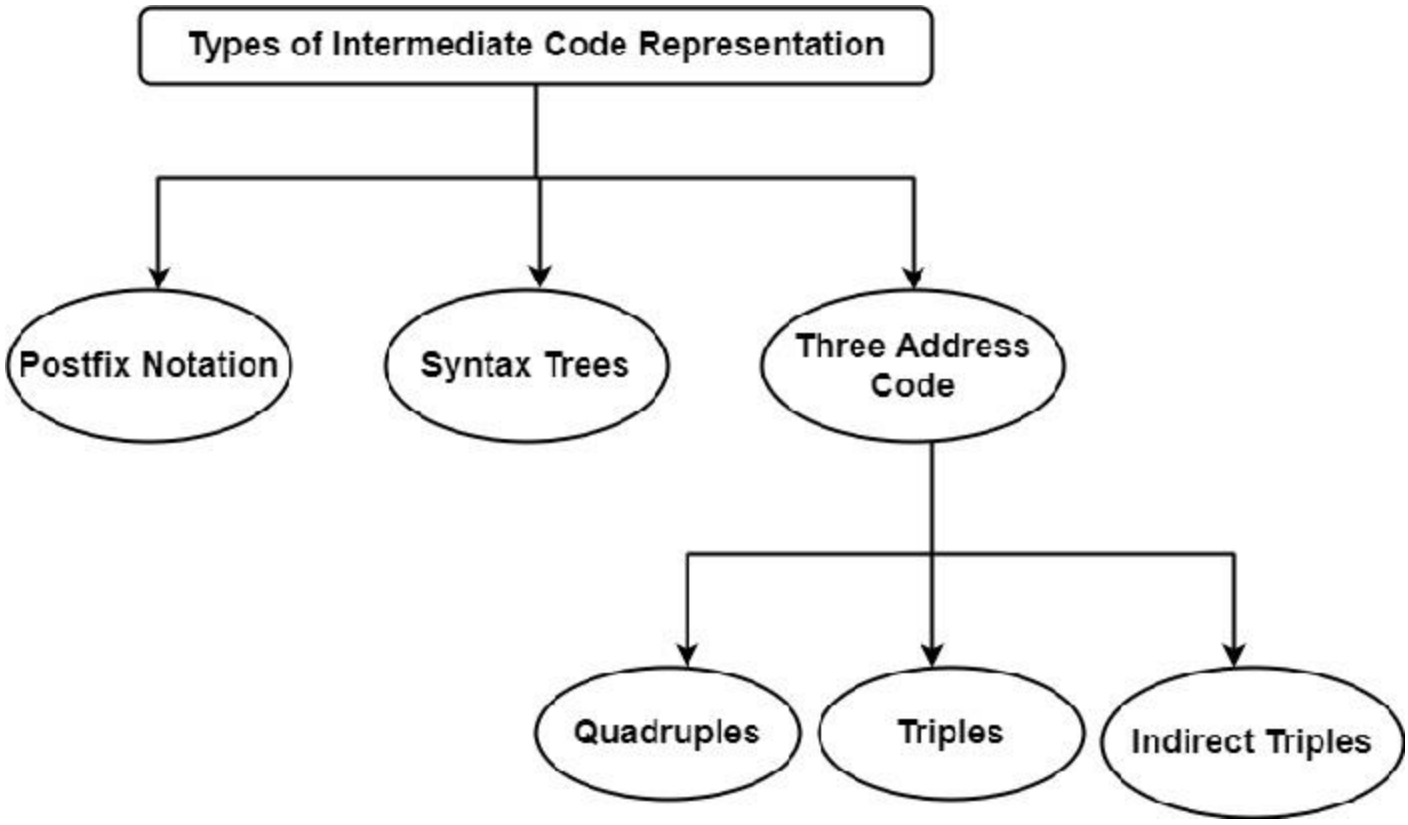
To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

```
enter(name, type, offset)
```

This procedure should create an entry in the symbol table, for variable *name*, having its type set to type and relative address *offset* in its data area.

Different types of intermediate forms:

The translation of the source code into the object code for the target machine, a compiler can produce a middle-level language code, which is referred to as intermediate code or intermediate text. There are three types of intermediate code representation are as follows −



Postfix Notation

In postfix notation, the operator comes after an operand, i.e., the operator follows an operand.

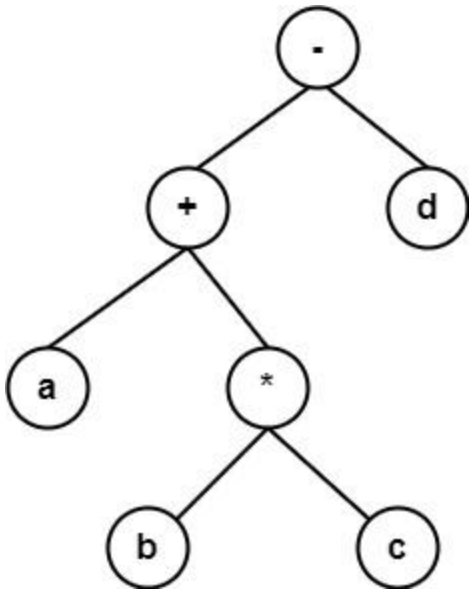
Example

- Postfix Notation for the expression (a+b) * (c+d) is ab + cd +*
- Postfix Notation for the expression (a*b) - (c+d) is ab* + cd + - .

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.

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



Three-Address Code

The three-address code is a sequence of statements of the form A−=B op C, where A, B, C are either programmer-defined names, constants, or compiler-generated temporary names, the op represents for an operator that can be fixed or floatingpoint arithmetic operators or a Boolean valued data or a logical operator. The reason for the name "three address code" is that each statement generally includes three addresses, two for the operands, and one for the result.

There are three types of three address code statements which are as follows −

Quadruples representation − Records with fields for the operators and operands can be define three address statements. It is possible to use a record structure with fields, first hold the operator 'op', next two hold operands 1 and 2 respectively, and the last one holds the result. This representation of three addresses is called a quadruple representation.

Triples representation − The contents of operand 1, operand 2, and result fields are generally pointer to symbol records for the names described by these fields. Therefore, it is important to introduce temporary names into the symbol table as they are generated.

This can be prevented by using the position of statement defines temporary values. If this is completed then, a record structure with three fields is enough to define the three address statements− The first holds the operator and the next two holds the values of operand 1 and operand 2 respectively. Such representation is known as triple representation.

Indirect Triples Representation − The indirect triple representation uses an extra array to list the pointer to the triples in the desired sequence. This is known as indirect triple representation.

The triple representation for the statement x− = (a + b)*-c is as follows −

Statement	Statement	Location	Operator	Operand 1	Operand 2
(0)	(1)	(1)	+	a	B

Statement	Statement	Location	Operator	Operand 1	Operand 2
(1)	(2)	(2)	-	c	
(2)	(3)	(3)	*	(0)	(1)
(3)	(4)	(4)	/	(2)	d
(4)	(5)	(5)	:=)	(3)	

Code Improvement (optimization):

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 less 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.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
```

```
  item = 10;
  value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
  value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

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 memory hierarchy.

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.

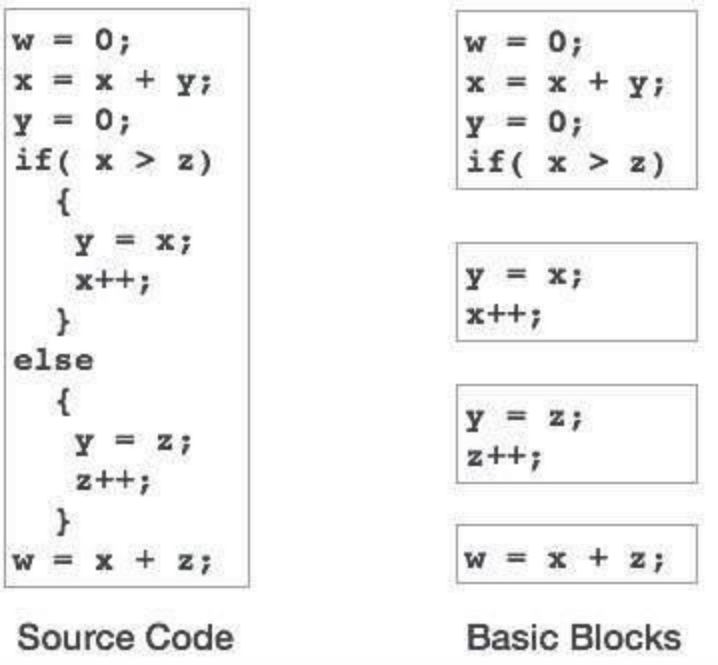
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

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.

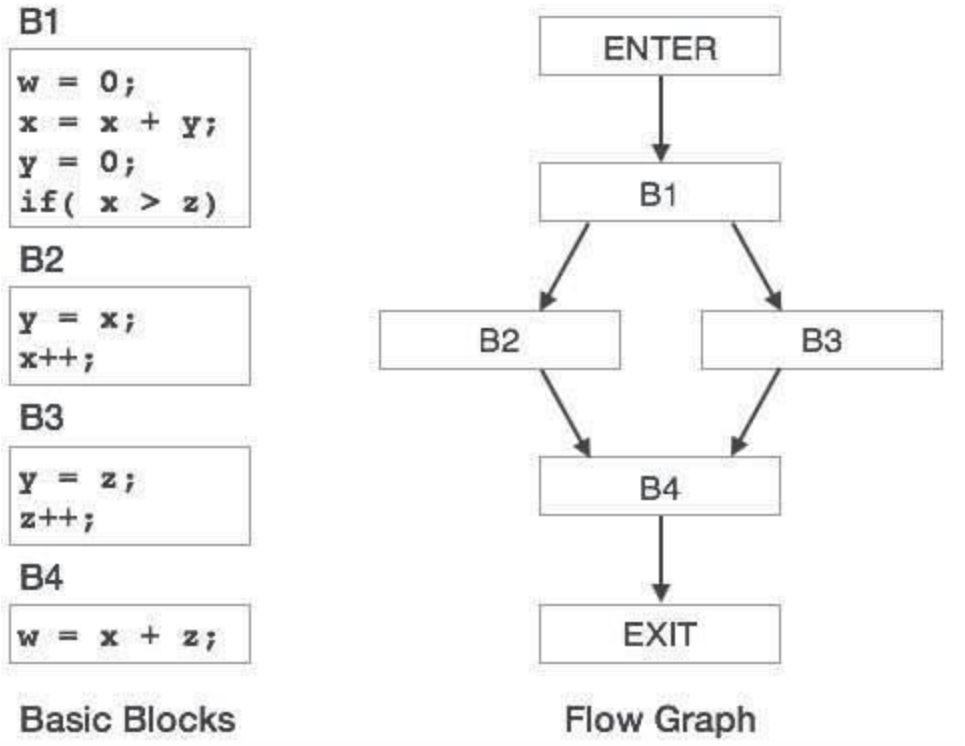
Basic blocks are important concepts from both code generation and optimization point of view.



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.



Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Dead-code Elimination

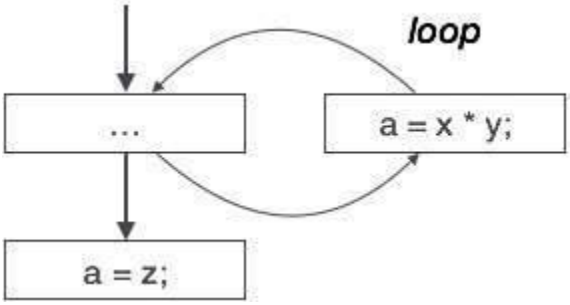
Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

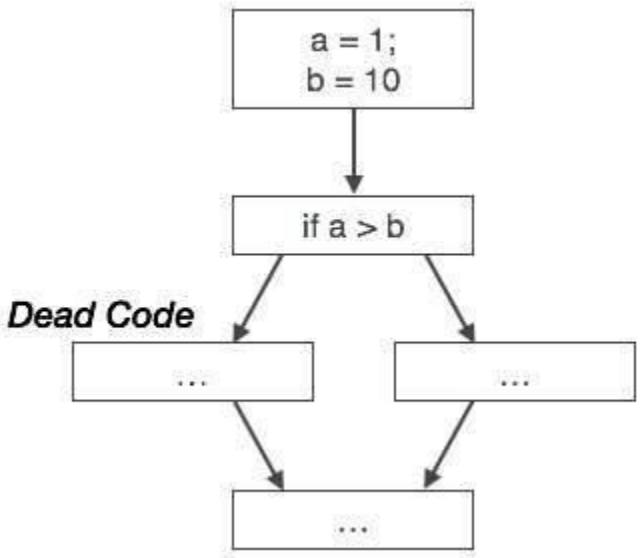
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



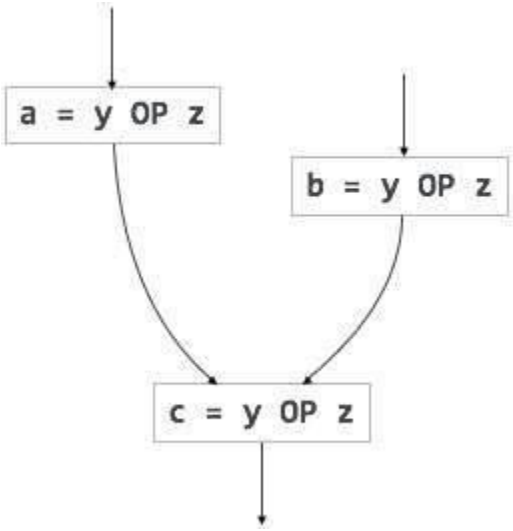
The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x * y’. Let us assume that the value assigned to ‘a’ is never used inside the loop.Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program. We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.



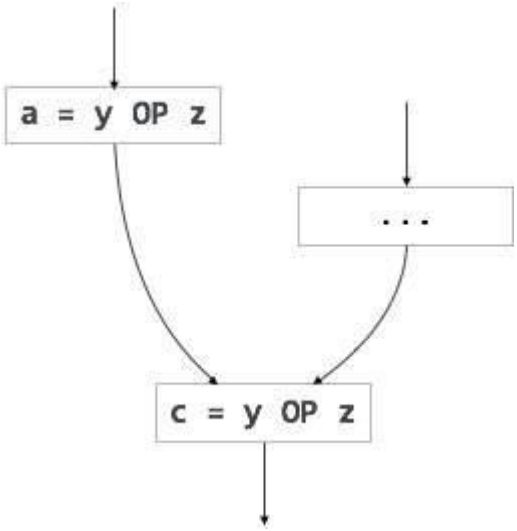
Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands.whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



[redundant expression]



[partially redundant expression]

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
{
  a = y OP z;
}
else
{
  ...
}
c = y OP z;
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
```

```
else
{
  ...
  tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

Control-flow, Data-flow dependence :

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.

Data flow analysis is a technique used in compiler design to analyze how data flows through a program. It involves tracking the values of variables and expressions as they are computed and used throughout the program, with the goal of identifying opportunities for optimization and identifying potential errors.

The basic idea behind data flow analysis is to model the program as a graph, where the nodes represent program statements and the edges represent data flow dependencies between the statements. The data flow information is then propagated through the graph, using a set of rules and equations to compute the values of variables and expressions at each point in the program.

Some of the common types of data flow analysis performed by compilers include:

1. Reaching Definitions Analysis: This analysis tracks the definition of a variable or expression and determines the points in the program where the definition “reaches” a particular use of the variable or expression. This information can be used to identify variables that can be safely optimized or eliminated.
2. Live Variable Analysis: This analysis determines the points in the program where a variable or expression is “live”, meaning that its value is still needed for some future computation. This information can be used to identify variables that can be safely removed or optimized.

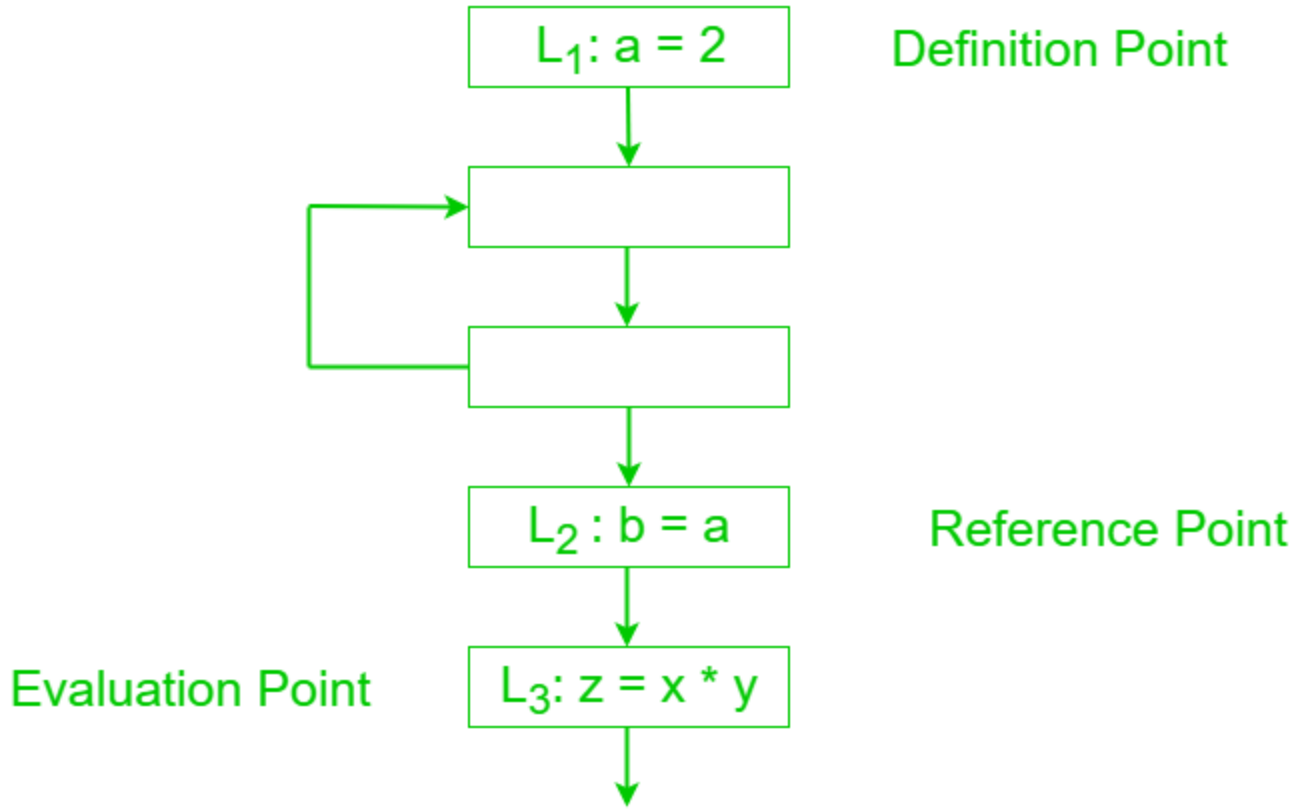
3. Available Expressions Analysis: This analysis determines the points in the program where a particular expression is “available”, meaning that its value has already been computed and can be reused. This information can be used to identify opportunities for common subexpression elimination and other optimization techniques.
4. Constant Propagation Analysis: This analysis tracks the values of constants and determines the points in the program where a particular constant value is used. This information can be used to identify opportunities for constant folding and other optimization techniques.

Data flow analysis can have a number of advantages in compiler design, including:

1. Improved code quality: By identifying opportunities for optimization and eliminating potential errors, data flow analysis can help improve the quality and efficiency of the compiled code.
2. Better error detection: By tracking the flow of data through the program, data flow analysis can help identify potential errors and bugs that might otherwise go unnoticed.
3. Increased understanding of program behavior: By modeling the program as a graph and tracking the flow of data, data flow analysis can help programmers better understand how the program works and how it can be improved.

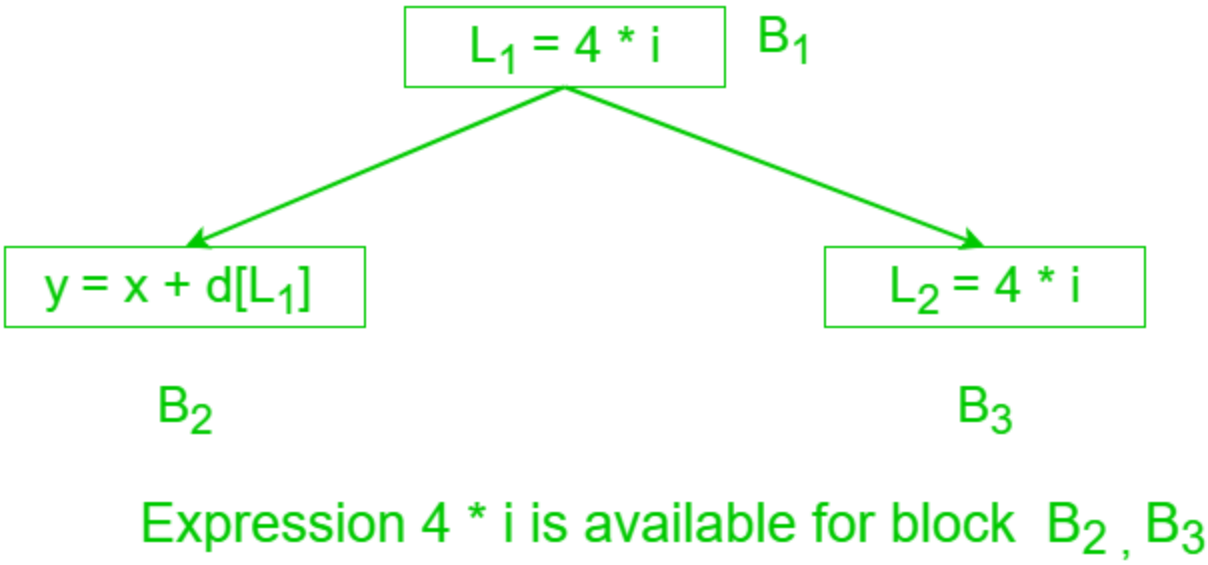
Basic Terminologies –

- **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.



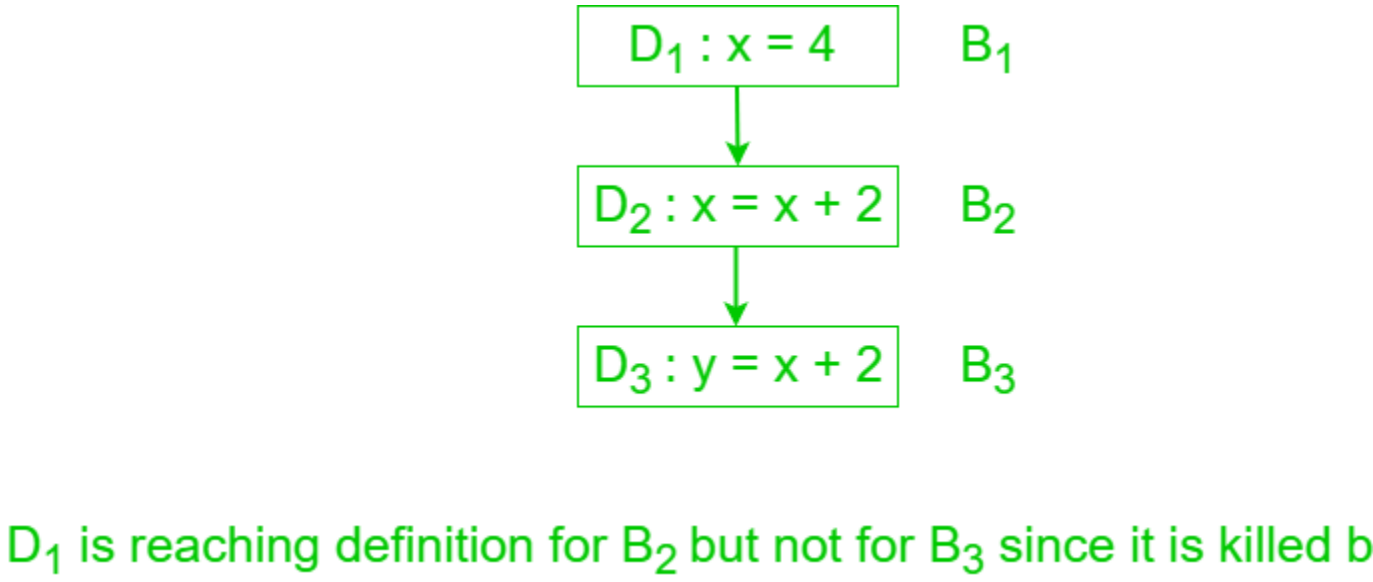
Data Flow Properties –

- **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 –



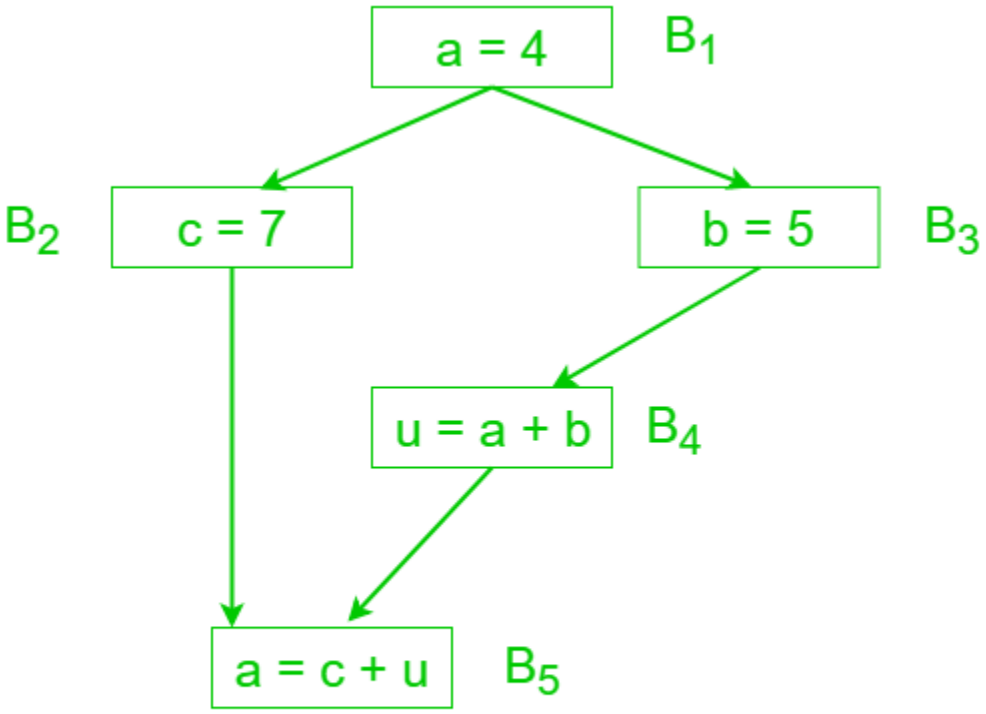
- **Advantage –**
  It is used to eliminate common sub expressions.

- **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 –



- **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.

Features :

Identifying dependencies: Data flow analysis can identify dependencies between different parts of a program, such as variables that are read or modified by multiple statements.

Detecting dead code: By tracking how variables are used, data flow analysis can detect code that is never executed, such as statements that assign values to variables that are never used.
Optimizing code: Data flow analysis can be used to optimize code by identifying opportunities for common subexpression elimination, constant folding, and other optimization techniques.
Detecting errors: Data flow analysis can detect errors in a program, such as uninitialized variables, by tracking how variables are used throughout the program.
Handling complex control flow: Data flow analysis can handle complex control flow structures, such as loops and conditionals, by tracking how data is used within those structures.
Interprocedural analysis: Data flow analysis can be performed across multiple functions in a program, allowing it to analyze how data flows between different parts of the program.
Scalability: Data flow analysis can be scaled to large programs, allowing it to analyze programs with many thousands or even millions of lines of code.

Local Optimization

- The simplest form of optimization
- Optimize one basic block. (No need to analyze the whole procedure body).

Some statements can be deleted

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

Eliminating these instructions are great optimizations as we are removing an entire instruction.

Some statements can be simplified

```
x := x * 0  =>  x := 0
```

Maybe the instruction on the right is faster than the instruction on the right. More importantly, assigning to a constant allows more cascading optimizations, as we will see later.

Another example:

```
y := y ** 2  => y := y * y
```

Typically, an architecture will not have the exponentiation instruction and so this operator may have to be implemented by an exponentiation loop in software. For the special cases (e.g., exponent = 2), we can simply replace by multiplication.

Another example:

```
x := x * 8  => x := x << 3
x := x * 15  => t := x << 4; x := t - x
```

Replace multiplication by a power-of-two to a shift-left operation. Can also do this for non powers-of-two. On some machines, left-shift is faster than multiply, but not on all! Modern machines have their own "rewriting logic" in hardware that can deal with these special cases really fast. In other words, some of these compiler local optimizations may become less relevant on modern hardware.

All these transformations are examples of *algebraic simplifications*.

Operations on constants can be computed at compile time

- If there is a statement `x := y op z`
- And `y` and `z` are constants.
- Then `y op z` can be computed at compile time.

Examples

- `x:= 2 + 2` can be changed to `x := 4`
- `if 2 < 0 jump L` can be deleted.
- `if 2 > 0 jump L` can be replaced by `jump L`

This class of optimizations is called *constant folding*. One of the most consequential and most common optimizations performed by compilers.

Another important optimization: eliminate unreachable basic blocks (dead code):

- Code that is unreachable from the initial block
  - e.g., basic blocks that are not the target of any jump or "fall through" from a conditional
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects
    - Increased spatial locality

Called dead-code elimination.

Why would unreachable basic blocks occur?

- e.g., `if (DEBUG) then { .... }`
  - Need constant propagation followed with dead-code elimination to remove this whole code.
- Libraries: a library may supply a 100 methods but we are using only a 3 of those methods, the other 97 are dead-code.
- Usually other optimizations may result in more dead code

Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment.

- Rewrite intermediate code in *single assignment* form
- `x := z + y`
- `a := x`
- `z := 2 * x`

Change to

```
b := z + y
a := b
x := 2 * b
```

More complicated in general, due to loops

If

- Basic block is in single assignment form
- A definition `x :=` is the first use of `x` in a block

Then

- When two assignments have the same rhs, they compute the same value.

Example

```
x := y + z
....
....
w := y + z
```

We can be sure that the values of `x`, `y`, and `z` do not change in the code. Hence, we can replace the assignment to `w` as follows:

```
x := y + z
....
....
w := x
```

This optimization is called *common-subexpression elimination*. This is also another very common and consequential compiler optimization.

If we see `w := x` appears in a block, replace subsequent uses of `w` with uses of `x`

- Assumes single assignment form

Example:
```
b := z + y
a := b
x := 2 * a
```
This can be replaced with
```
b := z + y
a := b
x := 2 * b
```
This is called *copy propagation*. This is useful for enabling other optimizations

- Constant folding. e.g., if the propagated value is a constant
- Dead code elimination. e.g., the copy statement can be eliminated as it is inconsequential.

Example:

```
a := 5
x := 2 * a
y := x + 6
t := x * y
```
gets transformed to
```
a := 5
x := 10
y := 16
t := x >> 4
```
We could have also assigned 160 to `t`. That would be even better.

If `w := rhs` appears in a basic block and `w` does not appear anywhere else in the program, THEN the statement `w := rhs` is dead and can be eliminated.

- Dead: does not contribute to the program's result

In our copy-propagation example, one of the assignments is dead and can be eliminated.

- Each local optimization does little by itself
- Typical optimizations interact
    - Performing one optimization enables another
- Optimizing compilers can be thought of as a *big bag of tricks*.

- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time
  - Certain properties on these transformations (tricks) ensure that we converge (and not oscillate) in this fixed point procedure. e.g., we always *improve* for some notion of improvement. This means that we are uni-directional and can never oscillate.

Example

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e + f
```

Final form

```
a := x * x
f := a + a
g := 6 * f
```

Possible to simplify further, but the compiler may get stuck in "local minima". e.g., should it have changed a + a to 2 * a, it would have had a better optimization opportunity (replacing g = 12*f and eliminating computation of f).

Peephole optimization

- Optimizations can be directly applied to assembly code. Typically for optimizations that got missed at IR stage (perhaps due to local minima problem).

  . At the assembly-level, we have greater visibility into instruction costs and instruction opcodes. E.g., hardware opcodes may be higher-level than IR opcodes.

- Peephole optimization is effective for improving assembly code
  - The "peephole" is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)

Peephole optimizations are usually written as replacement rules

```
i1; i2; i3; ..; in --> j1; j2; ..; jm
```

Example:

```
move $a $b; move $b $a --> move $a $b
```
Works if the second instruction is not the target of a jump (i.e., both instructions belong to a basic block).

Another example:

```
addiu $a $a i; addiu $a $a j --> addiu $a $a (i+j)
```

Many (but not all) of the basic block optimizations can be cast as peephole optimizations

- Example: `addiu $a $b 0` --> `move $a $b`
- Example: `move $a $a` -->
- These two together eliminate `addiu $a $a 0`.

As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect

- Need to ensure that the replacement rules cannot cause oscillations. e.g., each replacement rule can only "improve" the code.

Many simple optimizations can still be applied on assembly language.

"Program optimiztion" is grossly misnamed

- Code "optimizers" have no intention or even pretense of working towards the optimal program.
- Code "improvers" is a better term.

LOOP and GLOBAL 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:

- C

```
(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:

- C

```
//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:

- C

```
(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:

- C

```
#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 Folding, 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 :

- C

```
//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

```
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.

- C++

```
#include <iostream>

using namespace std;



int main() {

  int num;

  num=10;

    cout << "GFG!";

    return 0;

  cout << num; //unreachable code
```

```
}

//after elimination of unreachable code

int main() {

  int num;

  num=10;

    cout << "GFG!";

    return 0;

 }
```

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:

- C

Example 1 :

Multiplication with powers of 2 can be replaced by shift left 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:

- C

```
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:

- C

```
// 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:

- C

```
//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.
- **Interprocedural Optimization:** As the name indicates, the optimizations are applied inter procedurally. Techniques followed are Inline Substitution and Procedure Placement.

Advantages of Code Optimization:

Improved performance: Code optimization can result in code that executes faster and uses fewer resources, leading to improved performance.
Reduction in code size: Code optimization can help reduce the size of the generated code, making it easier to distribute and deploy.

Increased portability: Code optimization can result in code that is more portable across different platforms, making it easier to target a wider range of hardware and software.
Reduced power consumption: Code optimization can lead to code that consumes less power, making it more energy-efficient.
Improved maintainability: Code optimization can result in code that is easier to understand and maintain, reducing the cost of software maintenance.

Disadvantages of Code Optimization:

Increased compilation time: Code optimization can significantly increase the compilation time, which can be a significant drawback when developing large software systems.
Increased complexity: Code optimization can result in more complex code, making it harder to understand and debug.
Potential for introducing bugs: Code optimization can introduce bugs into the code if not done carefully, leading to unexpected behavior and errors.
Difficulty in assessing the effectiveness: It can be difficult to determine the effectiveness of code optimization, making it hard to justify the time and resources spent on the process.