Code Optimization

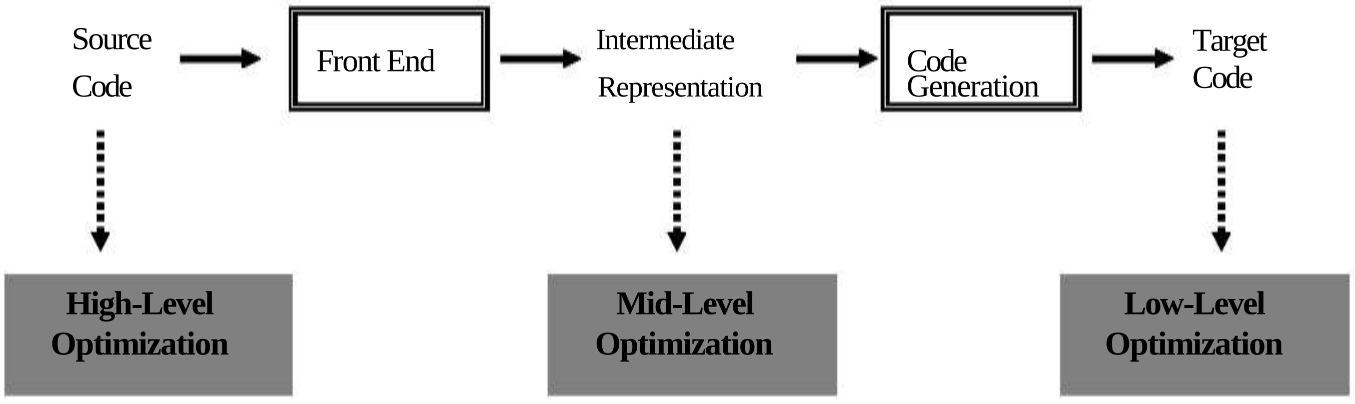
Compilers should produce target code that is as good as can be written by hand. This goal is achieved by program transformations that are called " Optimization " . Compilers that apply code improving transformations are called " Optimizing Compilers ".

Code optimization attempts to increase program efficiency by restructuring code to simplify instruction sequences and take advantage of machine specific features:-

- Run Faster , or
- Less Space , or
- Both (Run Faster & Less Space).

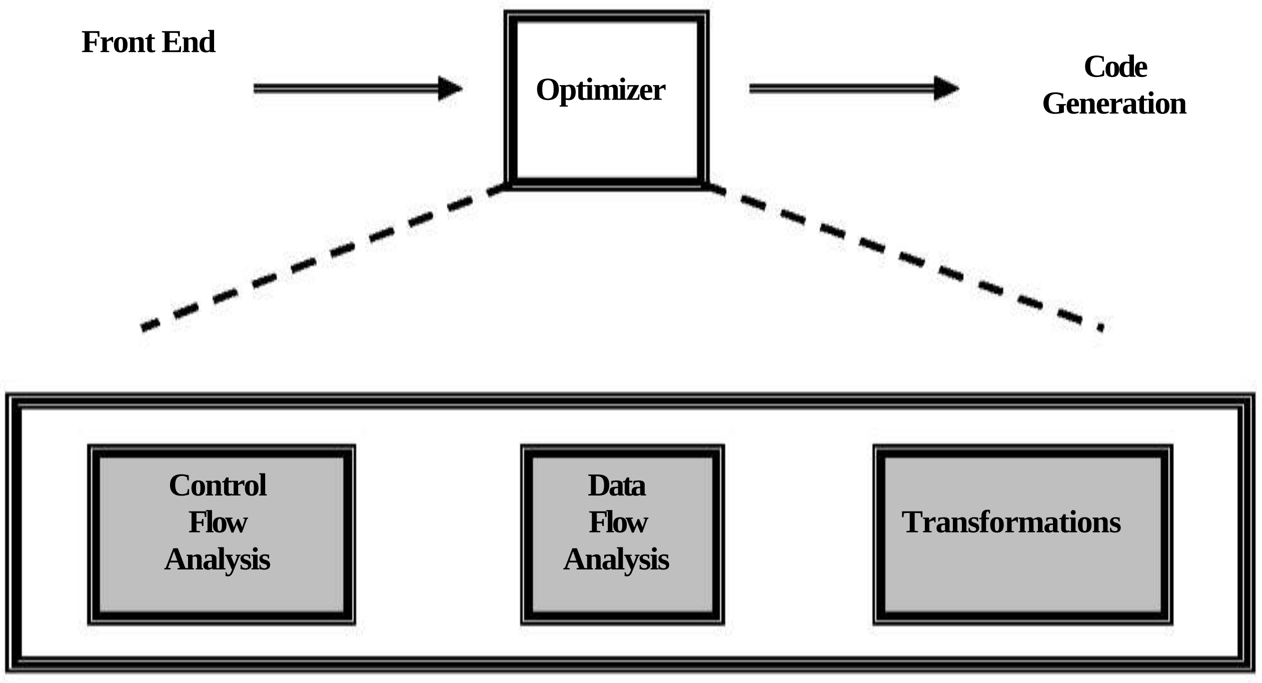
The transformations that are provided by an optimizing compiler should have several properties:-

1. A transformation must preserve the meaning of program. That is , an optimizer must not change the output produce by program for an given input, such as **division by zero.**
2. A transformation must speed up programs by a measurable amount.



Places for Optimization

This lecture concentrates on the transformation of intermediate code (Mid-Optimization or Independent Optimization),this optimization using the following organization:-



Organization of the Optimizer

This organization has the following advantages :-

1. The operations needed to implement high-level constructs are made explicit in the intermediate code.
2. The intermediate code can be independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for different machine

Basic Blocks:-

The code is typically divided into a sequence of "Basic Blocks". A Basic Block is a sequence of straight-line code, with no branches " In " or " Out " except a branch "In" at the top of block and a branch "Out" at the bottom of block.

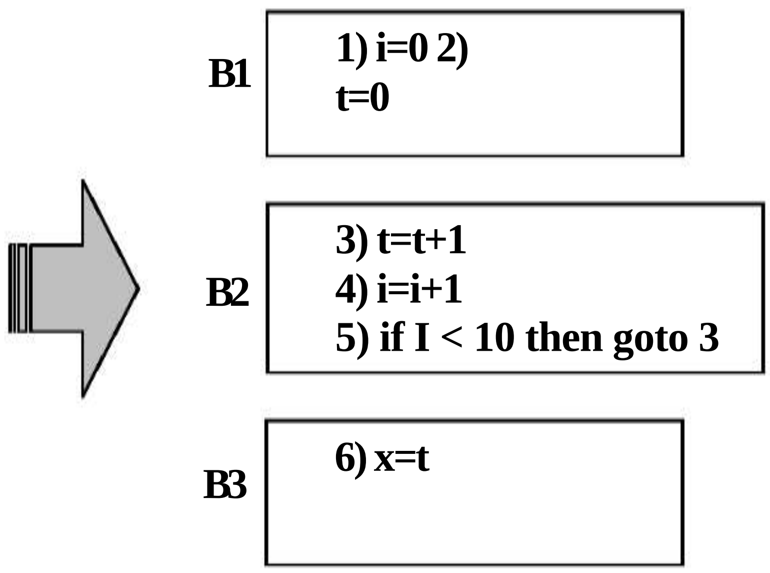
- **Set of Basic Block :** The following steps are used to set the Basic Block:
  1. Determine the Block beginning:
  i- The First instruction
  ii- Target of conditional & unconditional Jumps.
  iii- Instruction follow Jumps.

2. **Determine the Basic Blocks:**

i-There is Basic Block for each Block beginning.
ii-The Basic Block consist of the Block beginning and runs until the next Block beginning or program end.

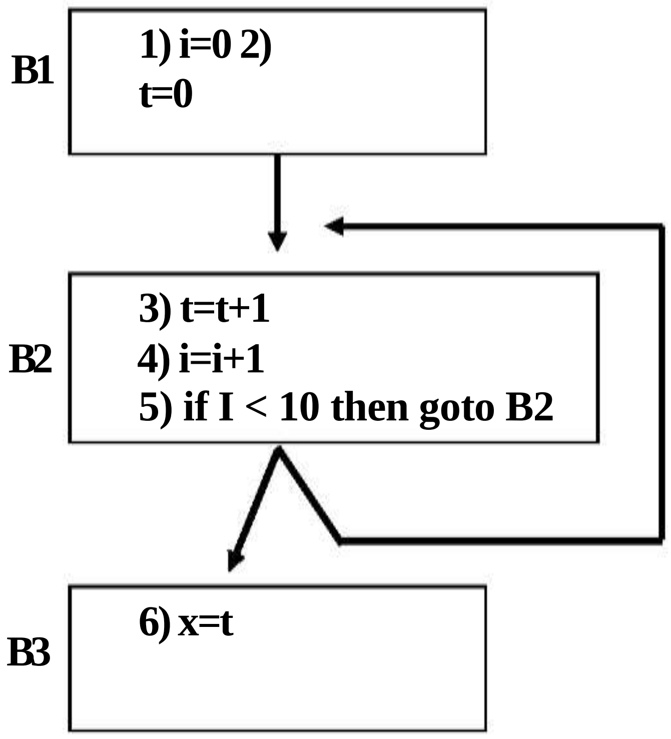
Example\\

```
1) i=0 2)
t=0
3) t=t+1
4) i=i+1
5) if I < 10 then goto 3
6) x=t
```



B1
```
1) i=0 2)
t=0
```
B2
```
3) t=t+1
4) i=i+1
5) if I < 10 then goto 3
```
B3
```
6) x=t
```

Basic Blocks



B1
```
1) i=0 2)
t=0
```
B2
```
3) t=t+1
4) i=i+1
5) if I < 10 then goto B2
```
B3
```
6) x=t
```

Control Flow

Data - Flow Analysis (DFA)

In order to do code optimization a compiler needs to collect information about program as a whole and to distribute this information to each block in the flow graph. DFA provides information about how the execution of a program may manipulate its data , and it provides information for *global optimization* .

There are many DFA that can provide useful information for optimizing transformations. One data-flow analysis determines how definitions and uses are related to each other, another estimates what value variables might have at a given point, and so on. Most of these DFAs can be described by data flow equations derived from nodes in the flow graph.

Reaching Definitions Analysis: All definitions of that variable, which reach the beginning of the block, as follow:

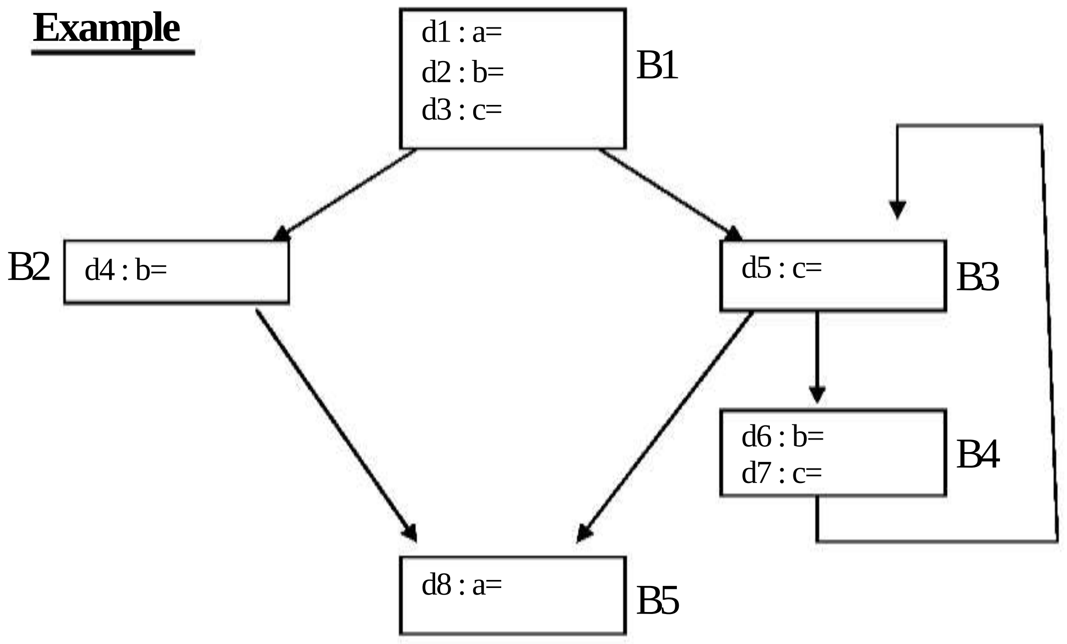
1. **Gen**[B] : contains all definitions *d:v= e* , in block B that *v* is not defined after *d* in B.
2. **Kill**[B] : if *v* is assigned in B , then Kill[B] contains all definitions *d:v= e*,in block different from B.
3. **In**[B] : the set of definitions reaching the beginning of B.

$$\textbf{In[B]} = \cup\ \textbf{Out[H]} \text{ where } H \in \text{Pred[B]}$$

4. **Out[B]** : the set of definitions reaching the end of B.

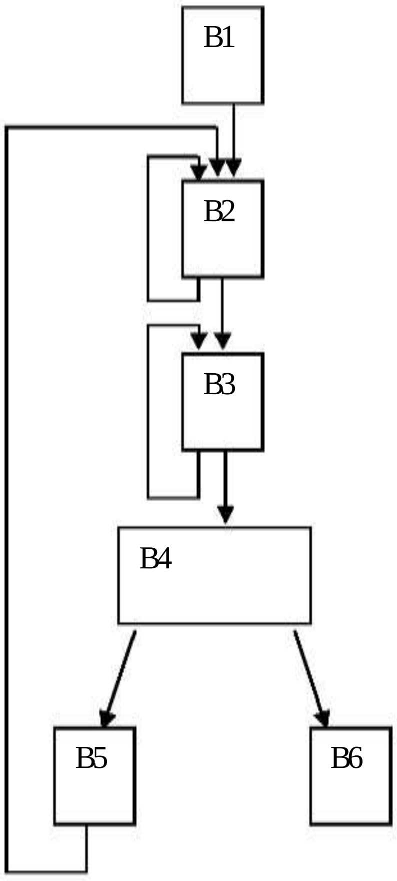
$$\textbf{Out[B]} = \textbf{Gen[B]} \cup (\textbf{In[B]} - \textbf{Kill[B]})$$

Example



Block	Gen	Kill	In	Out
B1	d1d2d3	d4d5d6d7d8	•	d1d2d3
B2	d4	d2d6	d1d2d3	d1d3d4
B3	d5	d3d7	d1d2d3d6d7	d1d2d5d6
B4	d6d7	d2d3d4d5	d1d2d5d6	d1d6d7
B5	d8	d1	d1d2d3d4d5d6	d2d3d4d5d6d8

Loop Information: The simple iterative loop which causes the repetitive execution of one or more *basic blocks* becomes the prime area in which optimization will be considered .Here we determine all the loops in program and limit *headers* & *preheaders* for every loop, for example:



Flow Graph

Loop No.	Header	Preheader	Blocks
1	B2	B1	2-3-4-5-2
2	B2	B1	2-2
3	B3	B2	3-3

Loop Information