SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-IV Compiler Design – SCS1303

IV. CODE OPTIMIZATION

Optimization –Issues related to optimization –Basic Block – Conversion from basic block to flow graph – loop optimization & its types - DAG – peephole optimization – Dominators – Data Flow Optimization

Optimization:

Principles of source optimization:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (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.

Types of optimization:

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.

Organization of the code optimizer:

The techniques used are a combination of Control-Flow and Data-Flow analysis as shown in Fig 4.1.

Control-Flow Analysis: Identifies loops in the flow graph of a program since such loops are usually good candidates for improvement.

Data-Flow Analysis: Collects information about the way variables are used in a program.

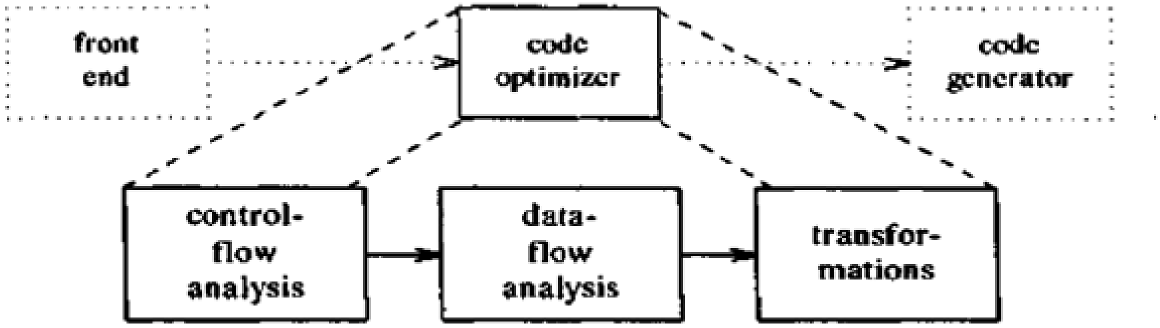


Fig 4.1. Organization of the code optimizer

Steps before optimization:

1) Source program should be converted to Intermediate code

2) Basic blocks construction

3) Generating flow graph

4) Apply optimization

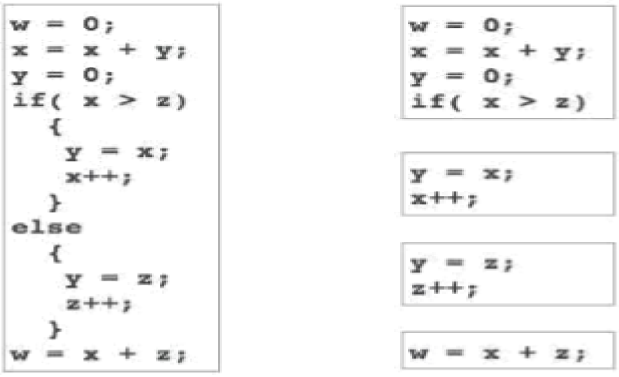
Basic Block and Flowgraphs:

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, SWITCHCASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Characteristics of Basic Blocks:

1. They do not contain any kind of jump statements in them.
2. There is no possibility of branching or getting halt in the middle.
3. All the statements execute in the same order they appear.
4. They do not lose the flow control of the program.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Program

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

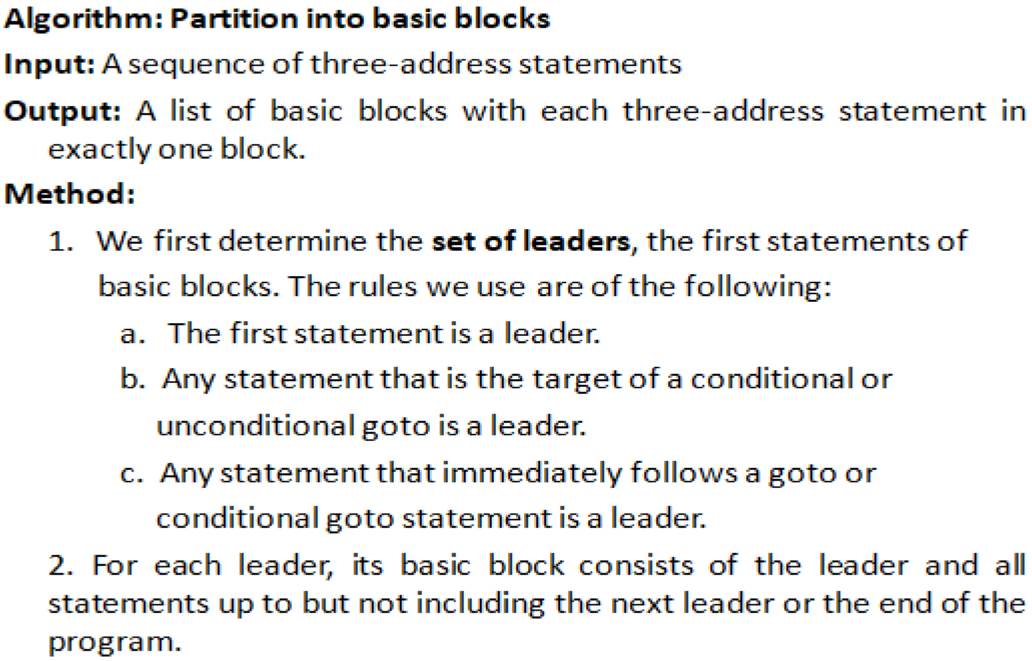
```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks

Fig 4.2. Basic Blocks for the sample source program

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



Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

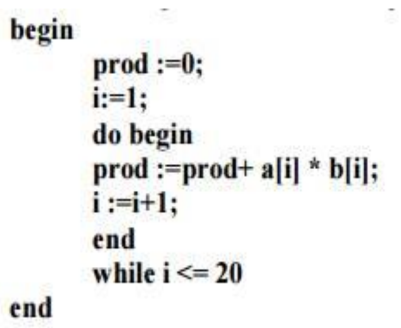
Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the **set of leaders,** the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

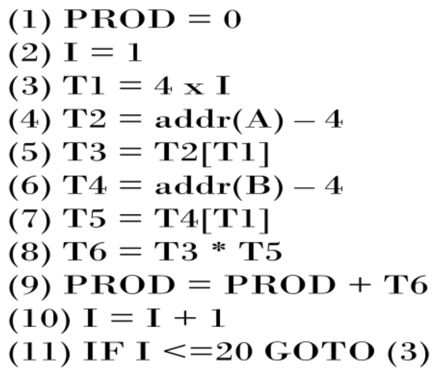
Example:

Consider the following source code for dot product of two vectors:



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

The three address code sequence for the above source is given as follows:

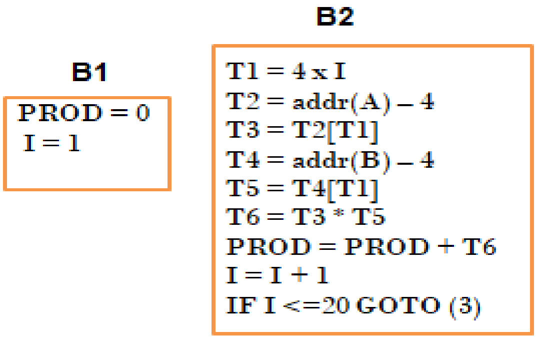


```
(1) PROD = 0
(2) I = 1
(3) T1 = 4 x I
(4) T2 = addr(A) – 4
(5) T3 = T2[T1]
(6) T4 = addr(B) – 4
(7) T5 = T4[T1]
(8) T6 = T3 * T5
(9) PROD = PROD + T6
(10) I = I + 1
(11) IF I <=20 GOTO (3)
```

Solution: For partitioning the three address code to basic blocks.

- PROD = 0 is a leader since first statement of the code is a leader.
- T1 = 4 * I is a leader since target of the conditional goto statement is a leader.
- Any statement that immediately follows a goto or conditional goto statement is a leader.

Now, the given code can be partitioned into two basic blocks as:



```
PROD = 0
I = 1
```

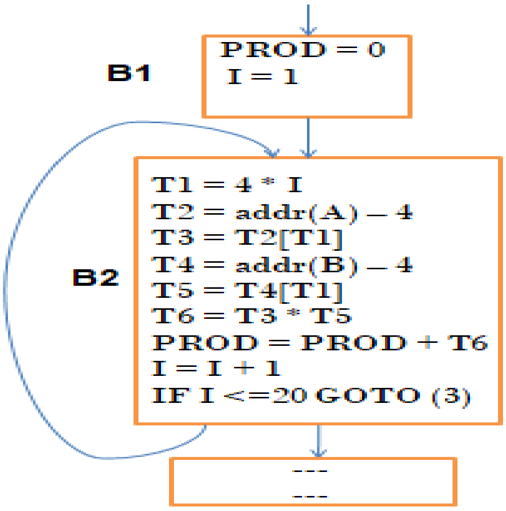
```
T1 = 4 x I
T2 = addr(A) – 4
T3 = T2[T1]
T4 = addr(B) – 4
T5 = T4[T1]
T6 = T3 * T5
PROD = PROD + T6
I = I + 1
IF I <=20 GOTO (3)
```

Flow graphs:

A flow graph is a directed graph with flow control information added to the basic blocks.The basic blocks serves as nodes of the flow graph. The nodes of the flow graph are basic blocks. It has a distinguished initial node.

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

E.g. Flow graph for the vector dot product is shown in Fig 4.3:



```
PROD = 0
I = 1
```

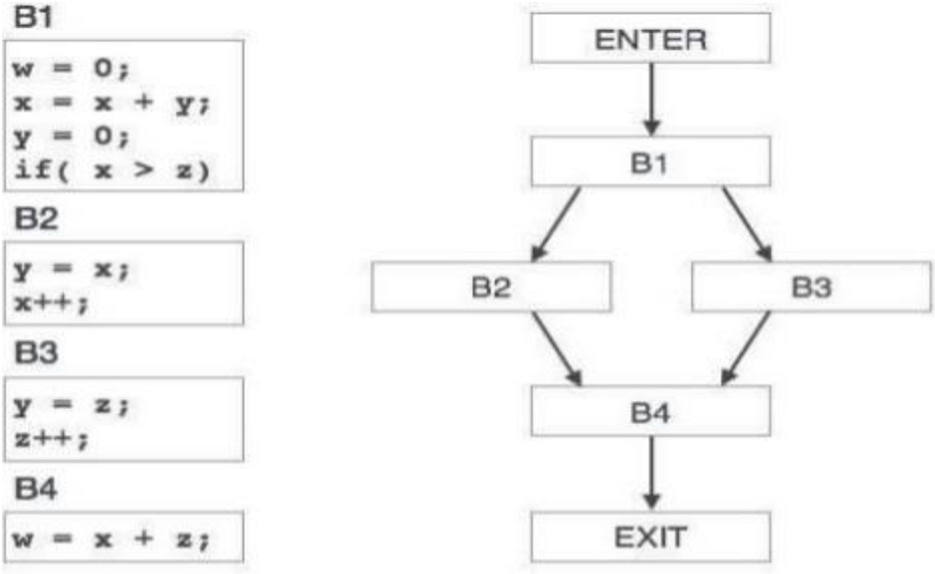
```
T1 = 4 * I
T2 = addr(A) – 4
T3 = T2[T1]
T4 = addr(B) – 4
T5 = T4[T1]
T6 = T3 * T5
PROD = PROD + T6
I = I + 1
IF I <=20 GOTO (3)
```

```
---
---
```

Fig 4.3. Flow graph

In Fig 4.3 B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B2 is the first statement B2, so there is an edge from B2(last statement) to B2 (first statement).B1 is the predecessor of B2, and B2 is a successor of B1.

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. Another example is shown in Fig 4.4.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

Fig 4.4. Flow graph for the basic blocks

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without modifying expressions computed by the block. Two important classes of transformation are :

1. Structure Preserving Transformations
    - Common sub-expression elimination
    - Dead code elimination
    - Renaming of temporary variables
    - Interchange of two independent adjacent statements
2. Algebraic transformation

Structure preserving transformations:

a)Common sub-expression elimination: Consider the sequence of statements:

1) a = b + c
2) b = a - d
3) c = b + c
4) d = a - d

Since the second and fourth expressions compute the same expression, the code can be transformed as:

1) a = b + c
2) b = a - d
3) c = b + c
4) d = b

b) Dead-code elimination: Suppose x is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables: A statement t : = b + c (t is a temporary) can be changed to u : = b + c (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block. Such a block is called a normal-form block.

d) Interchange of statements: Suppose a block has the following two adjacent statements:

t1 : = b + c

t2 : = x + y

We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2.

2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.

ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

iii) x/1 = x , x/2=x*0.5

Loop optimization & its types :

A loop is a collection of nodes in a flow graph such that:

1. All nodes in the collection are strongly connected**,** i.e, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and
2. The collection of nodes has a unique entry, i.e, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is through the entry point.

A loop that contains no other loops is called an **inner loop**.

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

(a) Invariant code :

- If a computation produces the same value in every loop iteration, move it out of the loop.
- An expression can be moved out of the loop if all its operands are invariant in the loop
- Constants are loop invariants.

(b) Induction analysis:

A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.

Eg:

extern int sum;

int foo(int n)

```
{
int i, j; j = 5;
for (i = 0; i < n; ++i)
{
j += 2; sum += j;
}
return sum;
}
```

This can be re-written as,

```
extern int sum;
int foo(int n)
{
int i; sum = 5;
for (i = 0; i < n; ++i)
{
sum += 2 * (i + 1);
}
return sum;
}
```

(c) Reduction in Strength:

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.

```
c = 7;
for (i = 0; i < N; i++)
{
y[i] = c * i;
}
```

can be replaced with successive weaker additions

```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
```

```
y[i] = k;
k = k + c;
}
```

(d) Constant folding and constant propagation

Constant folding:

It is the process of recognizing and evaluating statements with constant expressions (i=22+222+2222 to i=2466), string concatenation (“abc”+”def” to “abcdef”) and expressions with arithmetic identities (x=0; z=x*y[i]+x*2 to x=0; z=0;) at compile time rather than at execution time.

Constant propagation:

It is the process of substituting values of known constants in an expression at compile time.

Eg:

```
int x=14;
int y=7+x/2;
return y*(28/x+2);
```

Applying constant folding and constant propagation,

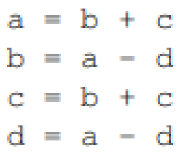
```
int x=14;
int y=14;
return 56;
```

Direct Acyclic Graph (DAG):

DAG is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:



```
a = b + c
b = a - d
c = b + c
d = a - d
```

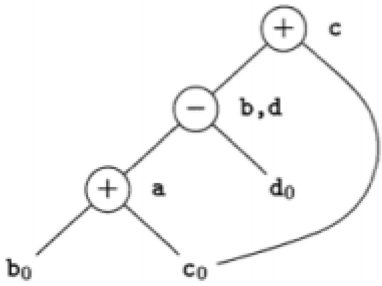


Fig. 4.5. Directed Acyclic Graph

Peephole Optimization

- Optimizing a small portion of the code.
- 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

(1) Redundant instruction elimination

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

First instruction can be rewritten as

MOV x,R1

(2) Unreachable Code

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

```
#define debug 0
….
If ( debug )
{
Print debugging information
```

}

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

If debug =1 goto L1 goto L2

L1: print debugging information L2: …………………….................................…… (a)

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

If debug ≠1 goto L2

Print debugging information L2: …………………………................................... (b)

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

As the argument of the statement of (c) evaluates to a constant true it can be replaced

By goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

(3) Flow of control optimization

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

We can replace the jump sequence

goto L1

….

L1: gotoL2 (d)

by the sequence

goto L2

….

L1: goto L2

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

if a < b goto L1

….

L1: goto L2 (e)

can be replaced by

If a < b goto L2

….

L1: goto L2

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

```
goto L1
L1: if a < b goto L2 (f)
L3:
```

may be replaced by

```
If a < b goto L2 goto L3
…….
L3:
```

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

(4) Algebraic Simplification

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

x := x+0

or

x := x * 1

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

(5) Reduction in Strength

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

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

X2 → X*X

(6) Use of Machine Idioms

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

i:=i+1 → i++

i:=i-1 → i- -

Dominators

Loop:

Loop is any cycle in the flow graph of a program

Properties of a loop:

1. It should have a single entry node, such that al paths from outside the loop to any node in the loop go through the entry.
2. It should be strongly connected (ie) it should be possible to go from any node of the loop to any other, staying within the loop.

Dominators

The method of detecting loops in the flow graphs is described using the notion of Dominators.

In order to deduct the control flow within basic blocks, it’s required to compute dominators. In the flow graph consisting a node D and E, if path leaves from D to E, then node D is said as dominating E. Dominator Tree: Dominator tree represents the hierarchy of the blocks and its execution flow. Here, the initial node is taken as the root and every further parent is said to be intermediate dominator of child node.

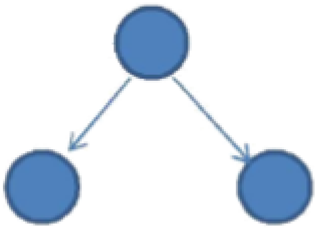


Fig. 4.6. Dominator Tree

In above tree, node 1 dominates 2 and 3.

Dominator Tree:

A useful way of presenting dominator information is in a tree called dominator tree, in which the initial node is the root and the parent of each other node is its immediate dominator

Immediate Dominator

The immediate dominator of ‘n’ is the closest dominator of ‘n’ on any path from the initial node to ‘n’.

Properties of Dominators

The algebraic properties regarding the dominance relation are:

(1) Dominance is a reflexive partial order. It is

Reflexive { a DOM [a for all a]}

Antisymmetric { a DOM b & b DOM a}

Transitive { a DOM b & b DOM c => a DOM c}

(2) The dominators of each node ‘n’ are linearly ordered by the DOM relation

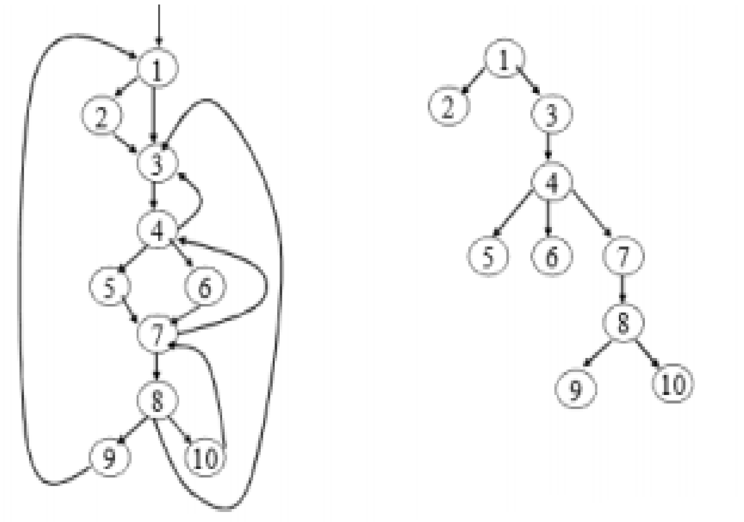
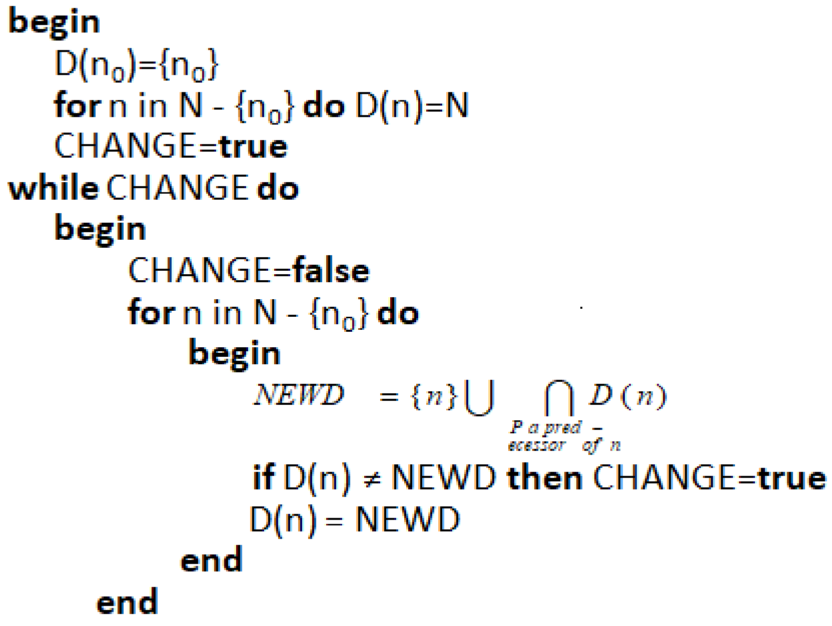
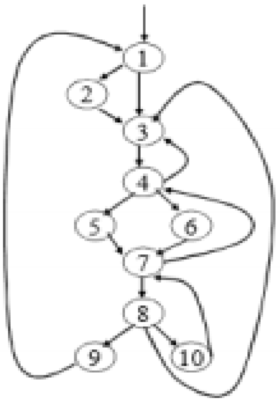


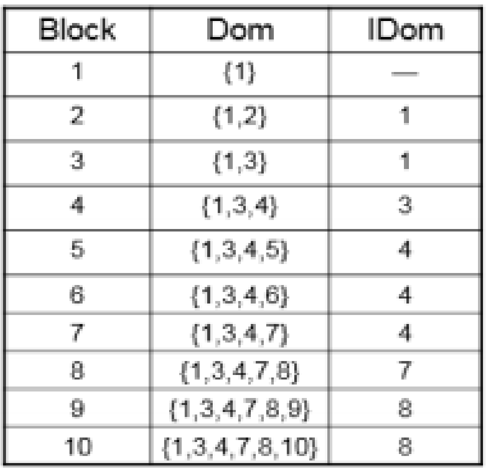
Fig. 4.7. Dominator Tree for the given flow graph

Dominator Computing Algorithm



```
begin
    $D(n_0)=\{n_0\}$
    for n in N - $\{n_0\}$ do D(n)=N
    CHANGE=true
while CHANGE do
    begin
        CHANGE=false
        for n in N - $\{n_0\}$ do
            begin
                $NEWD = \{n\} \bigcup \bigcap_{P\ a\ pred-ecessor\ of\ n} D(n)$
                if D(n) ≠ NEWD then CHANGE=true
                D(n) = NEWD
            end
    end
```





Block	Dom	IDom
1	{1}	—
2	{1,2}	1
3	{1,3}	1
4	{1,3,4}	3
5	{1,3,4,5}	4
6	{1,3,4,6}	4
7	{1,3,4,7}	4
8	{1,3,4,7,8}	7
9	{1,3,4,7,8,9}	8
10	{1,3,4,7,8,10}	8

Fig.4.8 Example for Dominators

Data flow analysis:

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.In order to find out the value of any variable (A) of a program at any point (P) of the program, Global data flow analysis is required.

use-definition (or ud-) chaining is one particular problem.

- Given that variable A is used at point p,at which points could the value of A used at p have been defined?
- By a use of variable A means any occurrence of A as an operand.
- By a definition of A means either an assignment to A or the reading of a value for a.
- By a point in a program means the position before or after any intermediate language statement.
  - Control reaches a point just before a statement when that statement is about to be executed.
  - Control reaches a point after a statement when that statement has just been executed.

A definition of a variable **A** reaches a point **p ,**if there is a path in the flow graph from that definition to **p**, such that no other definitions of **A** appear on the path.To determine the definitions that can reach a given point in a program requires assigning distinct number to each definition.To compute two sets for each basic block **B:**

- **GEN[B]-**set of generated definitions within block **B.**
- **KILL[B]-** set of definitions outside of **B** that define identifiers that also have definition within **B**.

To compute the sets IN[B] and OUT[B]:

IN[B]-set of all definition reaching the point just before the first statement of block B.

OUT[B]-set of definitions reaching the point just after the last statement of B

Data Flow equations:

1. $\mathrm{OUT}[B] = \mathrm{IN}[B] - \mathrm{KILL}[B] \bigcup \mathrm{GEN}[B]$
2. $IN[B] = \bigcup_{\substack{P\ a\ pred-\\ ecessor\ \ of\ B}} OUT\ [P]$

The rule (1) says that a definition d reaches the end of the block B if and only if either

i. d is in IN[B], i.e d reaches the beginning of B and is not killed by B ,or

ii. d is generated within B i.e., it appears in B and its identifier is not subsequently redefined within B.

The rule (2) says that a definition reaches the beginning of the block B if and only if it reaches the end of one of its predecessors.

Algorithm for reaching definition:

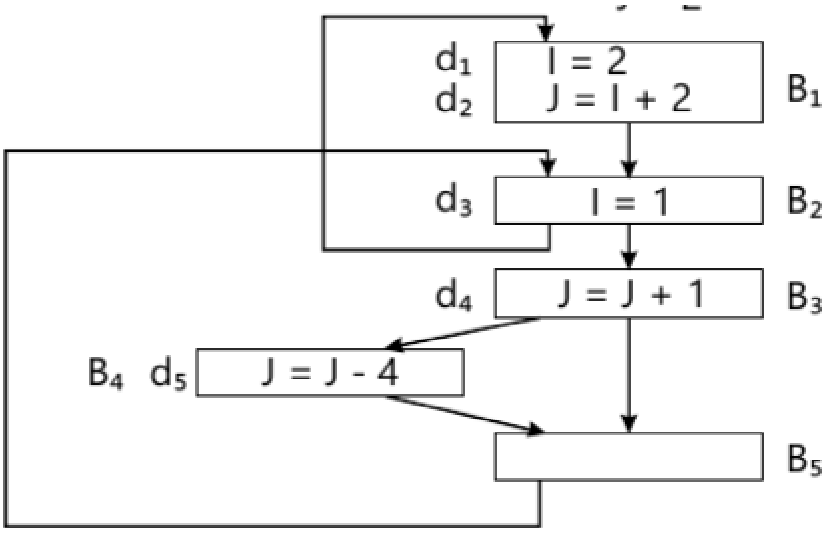
Input: A flow graph for which GEN[B] and KILL[B] have been computed for each block B.

Output: IN[B] and OUT[B] for each block B.

Method: An iterative approach, initializing with IN[B]=Ø for all B and converging to the desired values of IN and OUT.

```
begin
    for each block B do
    begin
        IN[B]=Ø
        OUT[B]=GEN[B]
    end
CHANGE=true
while CHANGE do
    begin
        CHANGE=false
        for each block B do
            begin
            $NEWIN = \bigcup_{P \text{ a predecessor of } B} OUT[P]$
            if NEWIN≠IN[B] then CHANGE=true
            IN[B]=NEWIN
            $OUT[B] = IN[B] - KILL[B] \bigcup GEN[B]$
            end
```

Example: Consider the Flow graph:



Solution:

i) Generate GEN(B) and KILL(B) for all blocks:

Block B	**GEN[B]**	**Bit vector**	**KILL[B]**	**Bit vector**
B1	$\{d_1,d_2\}$	11000	$\{d_3,d_4\ ,d_5\}$	00111
B2	$\{d_3\}$	00100	$\{d_1\}$	10000
B3	$\{d_4\}$	00010	$\{d_2,d_5\}$	01001
B4	$\{d_5\}$	00001	$\{d_2,d_4\}$	01010
B5	Ø	00000	Ø	00000

Computation For block B1 :

NEWIN=OUT[B2] , since B2 is the only predecessor of B1

NEWIN=GEN[B2]=00100 = IN[B1]

OUT[B1] = IN[B1] – KILL[B1] + GEN[B1]

= 00100 – 00111 + 11000

= 11000

For block B2 :

NEWIN=OUT[B1] + OUT[B5] , since B1 and B5 are the predecessor of B2

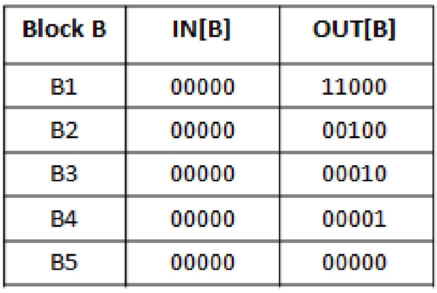
IN[B2] = 11000 + 00000 = 11000

OUT[B2] = IN[B2] – KILL[B2] + GEN[B2]

= 11000 – 10000 + 00100

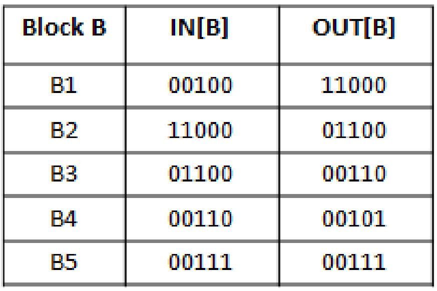
= 01100

Initial pass



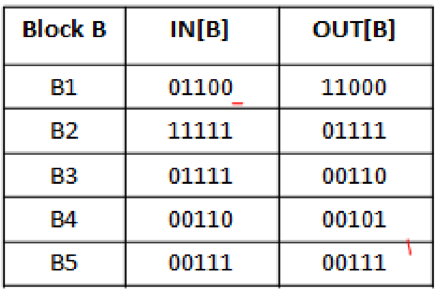
Block B	IN[B]	OUT[B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Pass-1



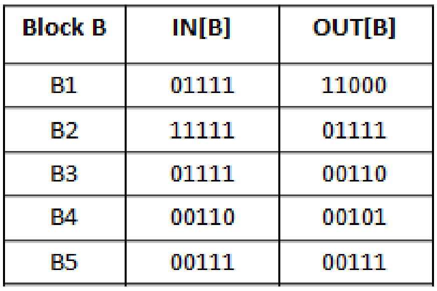
Block B	IN[B]	OUT[B]
B1	00100	11000
B2	11000	01100
B3	01100	00110
B4	00110	00101
B5	00111	00111

Pass-2



Block B	IN[B]	OUT[B]
B1	01100	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Pass-3



Block B	IN[B]	OUT[B]
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

As iteration 4 and 5 produces same values, the process ends. Hence the value of definition anywhere at any point of time is deduced.

Computing ud-chains:

- From reaching definitions information we can compute ud-chains.
- If a use of variable A is preceded in its block by a definition of A, then only the last definition of A in the block prior to this use reaches the use.
  - i.e. ud-chain for this use contains only one definition.
  - If a use of A is preceded in its block B by no definition of A, then the ud-chain for this use consists of all definitions of A in IN[B].

- Since ud-chain take up much space, it is important for an optimizing compiler to format them compactly.

Applications:

- Knowing the use-def and def-use chains are helpful in compiler optimizations including constant propagation and common sub-expression elimination.
- Helpful in dead-code elimination.
- Instruction reordering.
- (Implementation of) scoping/shadowing.