Intermediate code generation + Three-address code

ICG → platform Independant

purpose of ICG is to improve the efficiency of the code generation.

For Evaluating an arithmatic Expression, we are having 4-different strategies. one of them is the Syntax tree.

Syntax trees(SI) Abstract Syntax trees are one of the ways to represent the ICG.

Second one is Directed Acyclic Graphs.

Ex: (a+b) * (a+b+c)

① Syntax tree

② DAG

Another form is postfix Notation.

③ ab+ ab+c+ *

④ Another form of representing the Intermediate code is the Three-address code.

IGG (Intermediate code generator) will take the output of Semantic Analyser and produces the output in any of the mentioned ways.

The most popular way of representing the Intermediate code is 3-address code format.

$(a+b) * (a+b+c)$

$t_1 = a+b$

$t_2 = a+b$

$t_3 = t_2+c$

$t_4 = t_1 * t_3$

In 3-address code format we have atmost three addresses in any Expression. We will introduce the temporary variables to store the intermediate results.

$(a+b) * (a+b+c)$

t_1 t_2 t_3 t_4

Representations of 3-Address code:

① $x = a+b*c$

Temporary Variable t_1 Address ← $t_1 = b*c$ → Address$_1$ Address$_2$

$t_2 = a+t_1$

$x = t_2$

↓

Atmost 3-addresses

We can have 2-addresses also, as in 3-address code format we have at most 3-addresses in an Expression. Assignment operations also represented in 3-address format.

We can represent the three-address code in 3-ways.

① Quadraply

② Triply

③ Indirect Triply.

① **Quadraply:** Every operation will be represented with 4 values.

t_1	b	c	*
t_2	a	t_1	+
x	-	t_2	=

R A_1 A_2 op

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

As we have only one address in the operation A_1 is Empty.

② **Triply:** No temporary variables in case of Triples.

(1)	b	c	*
(2)	a	(1)	+
(3)	x	(2)	=

The result of operation b*c will be stored in the memory location (1).

Instead of the temporary variable we will directly use the memory location (01) address.

③ **Indirect Triply:**

(101)	b	c	*
(102)	a	(101)	+
(103)	x	(102)	=

pointers

We are storing the result of Computation Explicitly in the memory locations. We can reuse the same code when ever required at different location.

Reuse Computations.

(104) y (101) +

Types of 3-Address Code:

① $x = y \overset{A_1}{op} z \overset{A_2}{}$ ✓

② $x = op\ y$

③ $x = y$

④ $x = *y$

⑤ $x = \&y$

⑥ $x = a[i]$

⑦ $a[i] = x$

⑧ Goto L

⑨ if (expr) Goto L

⑩ $y = f(x_1, x_2 \cdots x_n)$

① $x = y \text{ op } z$
$A_3 \; A_1 \; \downarrow \; A_2$
operator
we can represent with 3-address format

② $x = \text{op } y \Rightarrow x = -z$ unary operation also represented with 3-address code.

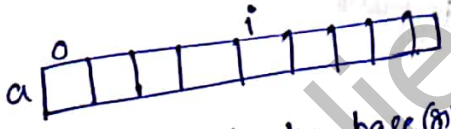
③ $x = y$ Simple Assignment operation. It is also represented.

④ $x = *y$ ✓ unary operation

⑤ $x = \&y$ Address of y. unary operation.
$A_1 \quad A_2$

⑥ $x = a[i]$ ✓ 3-Addresses
$A_1 \; A_2 \; A_3$
value at index i will be assigned to variable 'x'.

⑦ $a[i] = x$
$\downarrow \; \downarrow \; \downarrow$
$A_1 \; A_2$



a represents the base (or) starting address of the array.
We can represent with 3-address code.

⑧ Goto L → unconditional jump
↓ ↓
operation Address

⑨ If (expr) Goto L
↓ ↓
Address
Expression needs to be evaluated and stored in the memory location
↓
Address

⑩ $y = f(x_1, x_2, x_3 \cdots x_n)$ → NO
$\downarrow \; \downarrow \; \downarrow \quad \downarrow$
$A_1 \; A_2 \; A_3 \quad A_n$
More than 3-addresses are in the expression. We may not represent with 3-addresses.

Example of the 3-address Code:

① $x = \overbrace{\underbrace{a-b}_{t_1} - c + \underbrace{d/e}_{t_3}}^{t_4}$

$t_1 = a - b$

$t_2 = t_1 - c$

$t_3 = d / e$

$t_4 = t_2 + t_3$

$x = t_4$

All are valid 3-address format instructions (or) operations.

②
```
int x=1;
int y=2;
int z=x+y;
print z;
```
↓

$x=1$ → valid 3-address code.

$y=2$

$z=x+y$

goto L;

L: print(z)

We can represent in different ways. one of the possible ways we have written the 3-address code.

③
```
if (x>y)
    print("TOC")
else
    print("CD")
```

① if (x>y) goto L5 — Conditional Jump

② L1: print("CD")

③ goto ⑤ → unconditional Jump.

④ L5: print("TOC")

⑤ Exit/END

④

```
for(int i=1; i<=10; i++)
{
    int x=2*i;
    print(x);
}
```

We can represent the for loop with the valid 3-address code format.

Initialization ① i=1

Checking the condition → ② if (i>10) goto ⑨ : Conditional jump if condition is True then goto ⑨ END/EXIT program.

Perform the operation → ③ x=2*i → valid 3-address code.

④ Goto L → unconditional jump

⑤ L: print(x)

Increment i value → ⑥ t=i+1 } → valid 3-address code.

⑦ i=t }

⑧ goto ② → unconditional Jump
↳ Loop need to be iterated for the second time followed by 3rd time...

⑨ END/EXIT

Code optimizer and code generator takes the 3-address code and check for the possible optimizations and then generate machine code.

Runtime Environment:

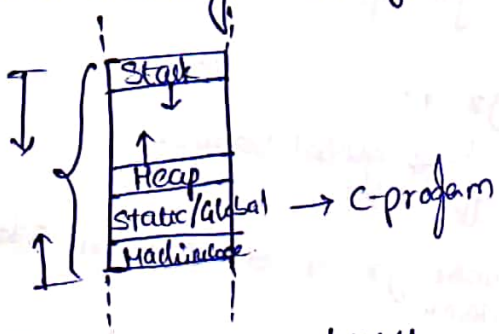
We can think of Runtime Environment as the suppat we get from operating system while executing the pacfam.

OS

Code → CD → MLL

↓

The generated MLL code will be stored in machine code Area.



When stack size increas, Heap will allocated in the Bottom of the memory.

operating system needs to allocate the Memory in Main memory to our pacfam/code and Static/Global variaby will also be stored in the code Area (or) static Area.

External Libraries (or) DLLs may not be available through out the pacfam execution.

operating System will not allows the whole memory to a single process. Whenever we call a function the operating system will helps to bring it available in the memory with the help of Linker.

Stack area will help us to store the information related to function calls.

Heap is used for Dynamic memory allocation. In C-programming we will use the functions like calloc and Malloc to dynamically allocate the memory.

Static Memory:

- variable bound to an address @ load-time.
- cannot free up in between.

```
int gx=1;
```

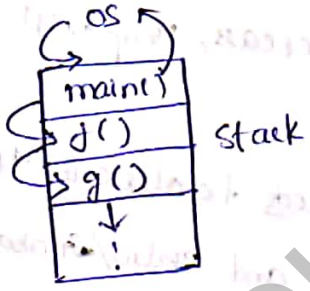
↳ global variable.

⇓

The variable name gx is bound to an address @ Loadtime

We cannot free up & remove the memory in between

Stack:



- One Entry (activation record) per function call.
- contains local variable in a function.

Stack is used to store (or) track function calls. Each Entry in the Stack is called as an activation Record.

OS itself calls main() to begin the Execution of the program.

All the variables (or) local variables are also stored in the Stack.

Recursion-stack

Heap:

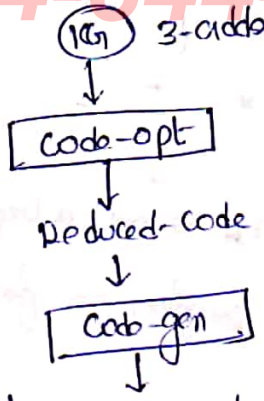
Dynamically allocate memory (calloc, Malloc, Free)

Operating System provides the memory for the programs which are currently in Execution.

Introduction to Code-optimization:

Reduces the number of lines
makes code run-faster

→ Machine Indep & Dependant



The purpose of code-optimizer is to take the 3-address code as an output of the ICG (Intermediate Code Generator) and generates the optimized (or) Reduced code.

The Reduced code can be given input to Code-generator will generate MLL code (or) Assembly Language code. If it generates ASL code will be given to Assembler and it will produce the MLL code.

Two kinds of optimizations

→ Machine dependant optimization. → [Intel, ARM]

→ Machine Independant optimizations.

⇓

We can perform (or) optimize irrespective of micro processor, Hardware configurations.

{ ARM
INTEL .. }

Based on microprocessor architecture, there can be some optimizations knowing machine dependant optimizations.

Loop-optimization (Machine-Independant-optimization)

In most of the programming languages we spent most of the time in loops. (80% Approximately)

→ First, detects loops in code [program Flowgraph]

```
While(0)
{
x  =
}
```

It is not a loop.

In a program flow graph the nodes are nothing but a basic blocks.

Basic-Blocks:

Sequence of 3-addr codes with no jumps.
- ↳ unconditional
- ↳ conditional

Eg:

```
f = t1
t2 = i+1
i = t2
```

No jumps in the code. only one block exists.

finding Basic Blocks:

Leaders: { 1st statement, target of a goto, stmt after a goto }

→ First statement of a program is always a leader

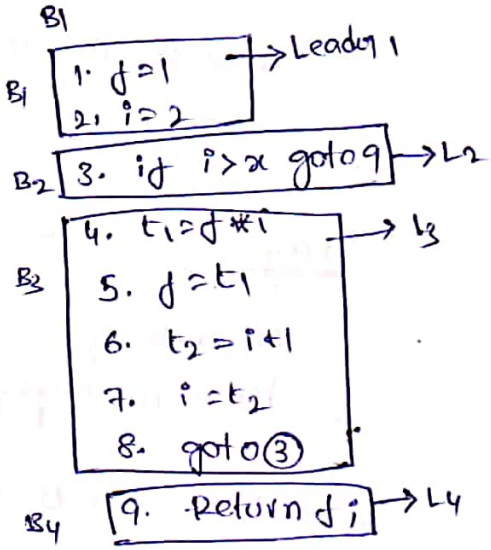
→ Target of a goto (conditional or unconditional) is also a leader.

→ Statement after a goto statement is also leader.

Eg:

```
fact(int x)
{
  int f=1;
  for(i=2; i<=x; i++)
    f=f*i
  return f;
}
```

ICG ⟹



① L_1 → First statement is a Leader.

② L_2 = goto ③ → Target of goto is a Leader

③ L_3 = ④ → Statement after a goto is a Leader,

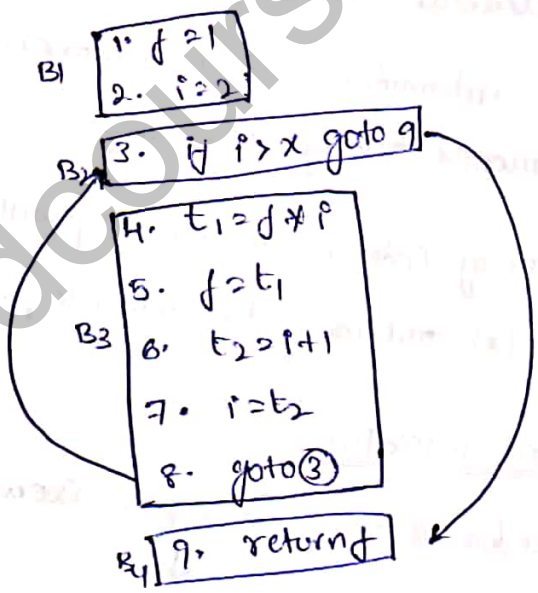
④ L_4 = ⑨ → Statement after a goto is a Leader.

The set of statements between the Leaders are known as a basic Block.

Program flow graph:

All the possible flow paths in the program.

cycle → Loop exists in our code [Loop-detection]



Loop-detection Algorithm:

① Find Leaders

② Basic Blocks construction.

③ Draw program flow graph.

④ If cycle exists in program flow graph, then loop will be there in the code.

⑤ Once we identify the loops, then we can proceed with loop-optimizations.

Loop optimization:

→ Reduce # lines of code in a loop.

Frequency Reduction:

```
i=0, x=0.1;
While(i<5000)
{
  A=sin(x)*i
  i++
}
```

We have to call the function to Evaluate sin(x)

```
i=0, x=0.1, t=sin(x);
while(i<5000)
{
  A=t+i;
  i++;
}
```

The intermediate code corresponding to the function call sin(x) will be moved out of while loop.

Lines of code inside the loop will be reduced, after moving the function sin(x) outside of the while loop.

Loop unrolling:

Reduce # times a loop is Executed. (JCG)

⇓

Looping is leading to more jump statements in the code.

```
While(i<10)
{
  a[i]=0
  i++
}
```

10 times

⇒

```
While(i<10)
{
  x[i]=0;
  i++;
  x[i]=0;
  i++;
}
```

Loop will run for 5 times

x[0] }
x[1] }

In the same loop two initializations are there.

 Loop unrolling will reduce the number of jump statements.

Loop Jamming:

* ⇒ Reduce Number of loops

Eg:

```
for(i=0; i<10; i++)
{
    for(j=0; j<10; j++)
        x[i,j]=10
}
for(i=0; i<10; i++)
    x[i,i]=0
```

⟹

```
for(i=0; i<10; i++)
{
    for(j=0; j<10; j++)
        x[i,j]=10
    x[i,i]=0
}
```

Some how we reduce the number of loops.

Introduction to Code optimization - II

Other Machine Independant optimization:

① **Folding:** $x = 2+3+C+B$ → Fold the Code. → Compile Time → $x = 5+C+B$

As the Compiler knows it as an Integer Constant, might be able to fold the Constants.

② **Redundancy Elimination (DAG):**

$a = b+c$

$d = 2+b+3+c$ ⇒ Redundancy Elimination → $a = b+c$, $d = 2+3+a$ → Fold the Constants. → $d = 5+a$

If the Compiler can able to identify the Expression which is Evaluated earlier then we can reduce the redundancy.

③ Strength Reduction:

B = A * 4 → B = A << 2

A * 4 : costly operation

A << 2 : cheap

④ Algebraic Simplification:

Eliminate the Expressions like

a = a + 0 ;
b = b * 1 ; } Not much significance of code in the program.

Machine Dependant optimization:

① Register Allocation
② peephole optimization

Some of the architectures are having more number of Registers and some of them are having less number of registers. As we are aware Register memory acess faster, we can assign the Registers optimally.

Peephole optimizations:

ⓐ Redundant Load & Store:

a = b + c
d = a + e

```
mov b,R0
add c,R0
MOV R0,a      STORE (Redundant)
MOV a,R0      LOAD
add c,R0
MOV R0,d
```

We can Eliminate this kind of Redundant Load and Store Instructions.

Flow-Control-optimizations:

① Avoid jump on jump

② Eliminate dead code

L1: jump L2
L2: jump L3
L3: jump L4

→ L1: JMP L4

We can directly move onto L4 instead of L3

```
#define x 0
if (x)
{
==
==
}
```

→ dead code.

⇒ x=0

```
if(0)
{
==
}
```

} we will never reach to code.

We can Eliminate or Ignore the dead code Instead of converting it into machine code.

③ Use machine code more appropriately:

i = i+1 →

```
MOV i, R0
add 1, R0
MOV R0, i
```

⇒ INC i

Instead of writing all the instructions, if the micro processor supports instruction formats like increment (INC) we can write the code with one instruction.

Constant propagation & Common SubExpression Elimination:

Data flow Analysis

Constant propagation:

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

propagate x →

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

propagate y →

```
int x=14
int y=0
return 0;
```

dead code Elimination ←

```
return 0;
```

propagate x ⇒ substitute x, where ever x in the code.

Now propagate y ⇒ int y = 7-14/2 = 0

⇒ y*(28/14+2) ⇒ 0*(28/14+2) ⇒ 0

Let f() {

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

After Constant propagation the function f() returns '0'.

Common Subexpression Elimination (CSE):

→ Redundancy Elimination is a form of CSE.

Eg:
```
a = b*c + g ;   -(1)
d = b*c*e ;     -(2)
```

b*c is the common subexpression in both the statements.

⇒
```
tmp = b*c
a = tmp + g;
d = tmp * e;
```
→ Space will also increase due to the additional temporary variables.

Trade off: if the computational cost > cost (temp creation & fetching) then it is useful.

Local Common Subexpression Elimination → With in a single block

Global Common Subexpression Elimination → Entire function/procedure

Data flow Analysis → structured graph-based analysis of the program.

Flow Graph (or) CFG (Control flow Graph)

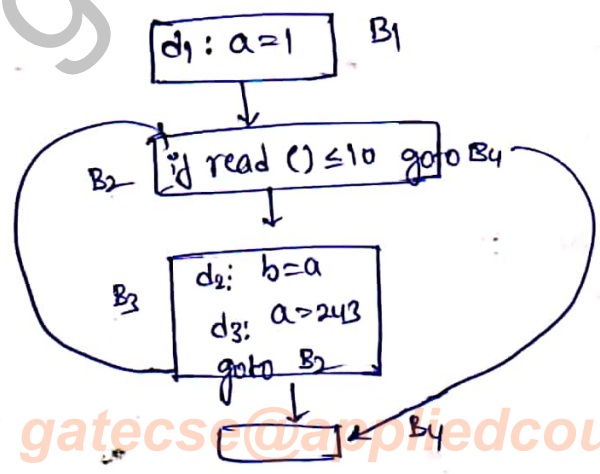
d_i : Definition-i

B_j : Basic Block-j

Block : Vertex

Transition : Edges

Flow (or)

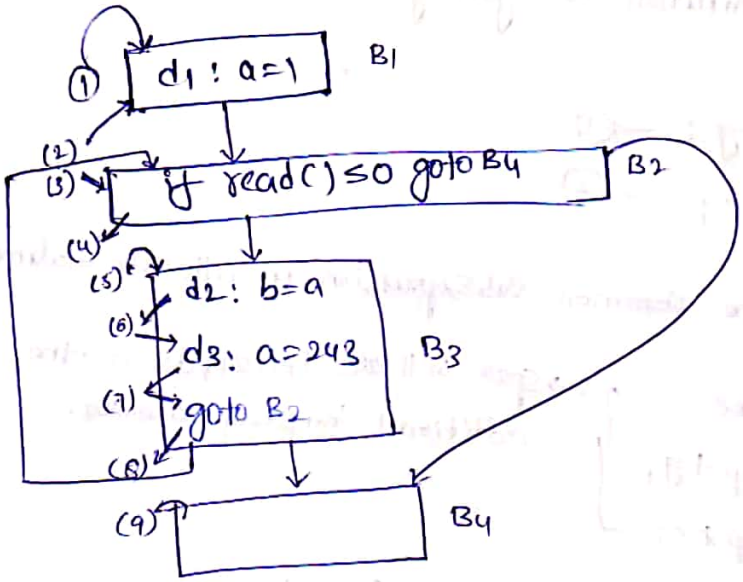


In the given code, we have a total of 3 definitions

d_1 : a=1
d_2 : b=a
d_3 : a=243

4 Basic Blocks.

Reaching Definition Analysis:



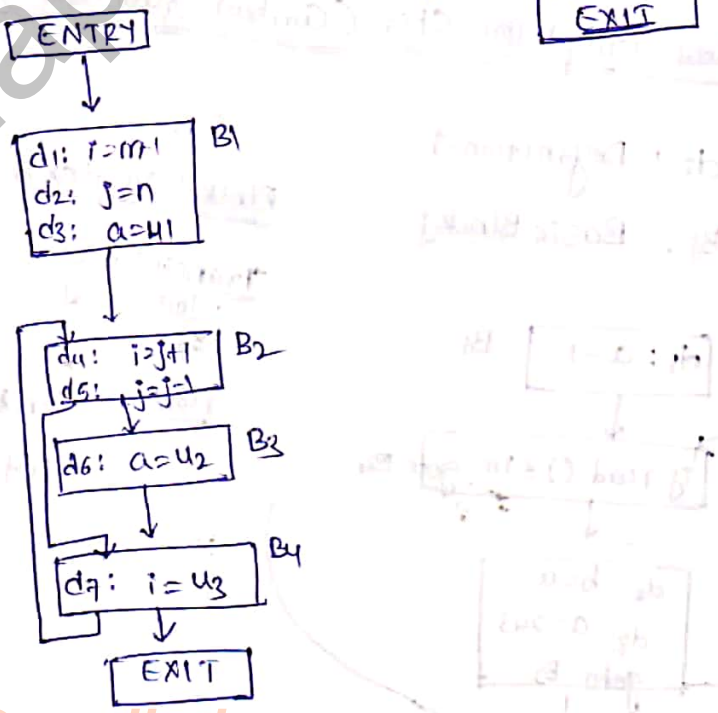
d_1 : "reaching point (5)"

We assigned a value of 1 to 'a'

definition d_1 reaching program point "5"

If there exist atleast one path to reach a program point then we say d_1 reaches point '5'

Flow-Graph: We are going to introduce two more blocks in the flow graph.



Reaching Definition Analysis:

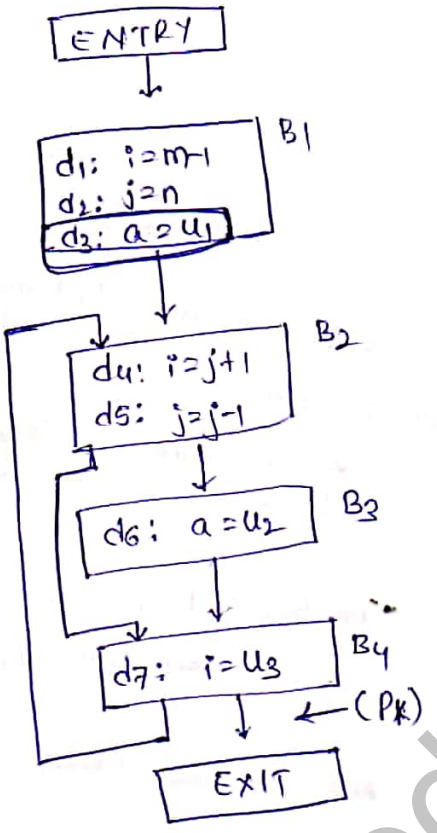
Question: Does d_3 reach end of B_4?

↓

Conservative Analysis

↓

Need to find atleast one path to reach point P_k.



If we reach B_3 via B_2 in B_3 the definition d_3 is overwritten by d_6.

$B_1 \rightarrow B_2 \rightarrow B_3$

If we follow this path definition d_3 is not reachable at Point P_k.

$B_1 \rightarrow B_2 \rightarrow B_4$

If we follow this path $B_1 \rightarrow B_2 \rightarrow B_4$ then the definition d_3 is reachable at point P_k.

GEN(B), KILL(B):

↳ Generates in Block B → sets.

↳ kills in Block B

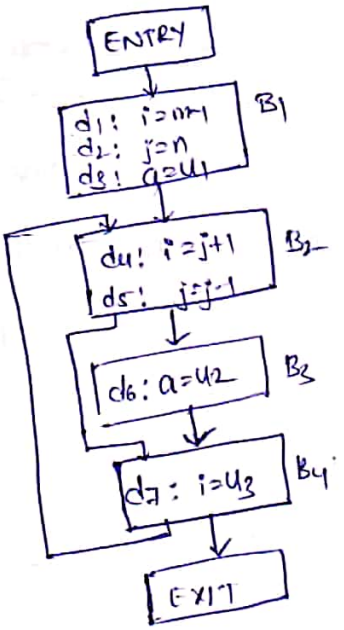
bit-vector

GEN(B1) = {d1, d2, d3} → 1110000

kill(B1) = {d4, d5, d6, d7} → 0001111

GEN(B2) = {d4, d5} 0001100

kill(B2) = {d1, d2, d3, d7} 1100001



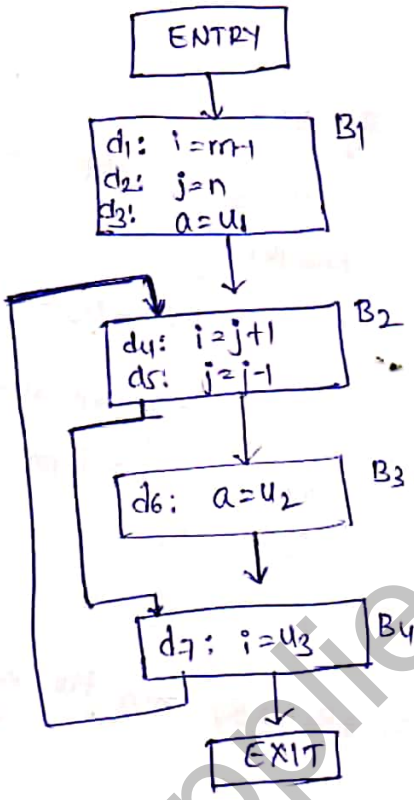
$gen(B_3) = \{d_6\}$ → only one definition d_6.

$kill(B_3) = \{d_3\}$ → The value of 'a' killed by d_6.

$gen(B_4) = \{d_7\}$

$kill(B_4) = \{d_1, d_4\}$

IN & OUT : sets of definitions



$IN(B_1) = \phi$ point of Entering into the Block.

$OUT(B_1) = \{d_1, d_2, d_3\}$

at the time of exiting from block B_1, the definitions that we left with.

$IN(B_2) = 1110\,1\,LL$

$\{d_1, d_2, d_3, d_5, d_6, d_7\}$

We can also enter into B_2 via B_4.

If we consider the path

$B_2 - B_4 - B_2$

The definition d_4 is killed by d_7. So on Entering into B_2 d_4 is not available.

$IN[B_2] = 1110111$

$OUT[B_2] = 0011\,1$ ($d_3\ d_4\ d_5$)

d_1 is killed by d_4

d_2 is killed by d_5

$IN[B_3] = 0011110$

$OUT[B_3] = 0001110$

$IN[B_4] = 0011110$

$OUT[B_4] = 0010111$

Mathematically define IN and OUT

$$IN(B) = \bigcup_{P \text{ predecessor of } B} OUT(P)$$

predecessor successor

Eg:- $IN(B_4) = OUT(B_3) \cup OUT(B_2)$

$= 0001110 \cup 0011110$

$= 0011110$

$OUT(B) = GEN(B) \cup [IN[B] - kill(B)]$

We can also write it as

$OUT_B = f_B(IN(B))$

↳ some function

Reaching definition Algorithm:

```
1  OUT[ENTRY] = ϕ ;
2  for (each basic block B other than ENTRY) OUT[B] = ϕ
3  while (changes to any OUT occur)
4      for (each basic block B other than Entry) {
5          IN[B] = ∪ OUT[P];
                   P a predecessor of B
6          OUT[B] = gen_B ∪ (IN[B] - kill_B);
       }
```

∀ B gen(B) and kill(B)

We have to run this while loop untill changes to any out occurs.

Reaching Definition Analysis:- Analysis using Dataflow Analysis / control flow Analysis.

① domain ⟶ definitions

② Transfer fn ⟶ forward : $OUT(B) = f_B(in(B))$

$f_B(x) = Gen_B \cup (x - kill_B)$

③ Meet operation ⟶ $in(b) = \bigcup_p out(p)$

④ Boundary condition ⟶ $OUT[Entry] = \phi$

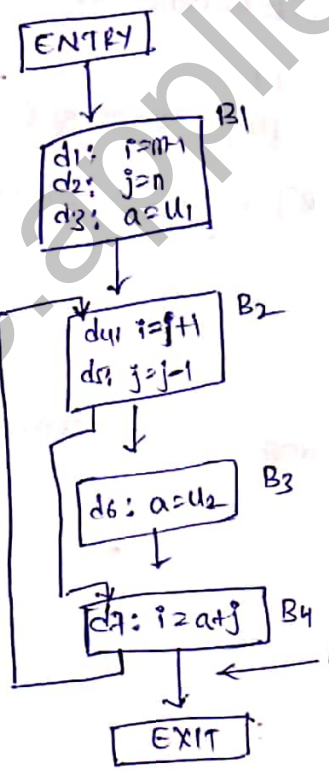
⑤ Initial Interior points ⟶ $OUT[B] = \phi$

Live Variable Analysis: DFA & CFG

variable 'v' is live @ point P if value of V is used in some path in flowgraph starting at P.

Else it's dead.

Eg:

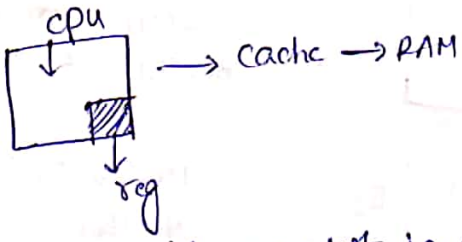


@ this point a, i, j & u_2 are live

There exist some path in which the variables are used.

Application: Register Allocation

Live variable Analysis



cpu can access the data, very fast in comparision with cache, RAM.

part of code-optimization.

variably that are used very often, that will be stored in Registers.

If the variables a, i, j & u_2 are preferred to store in the Registers.

DEF[B] & USE[B]:

def(B) is the set of variably defined in block B, before using them.

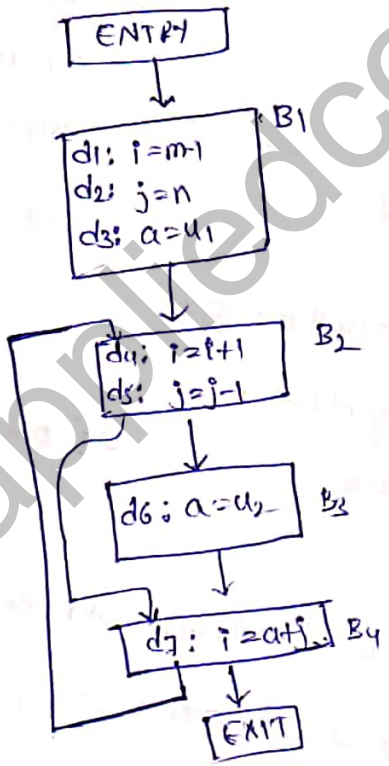
sets of variably

def(B1) = {i, j, a}
use(B1) = {m, n, u}

def(B2) = { }
use(B2) = {i, j}

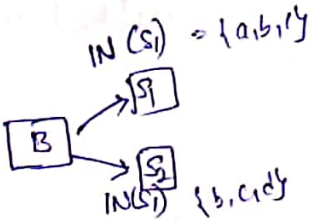
def(B3) = {a}
use[B3] = {u_2}

def(B4) = {i}
use(B4) = {a, j}



def(B2) = { } Null set

Before defining i and j, we are using it.

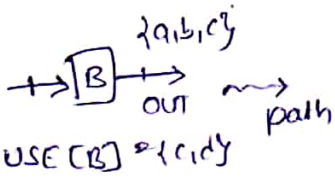


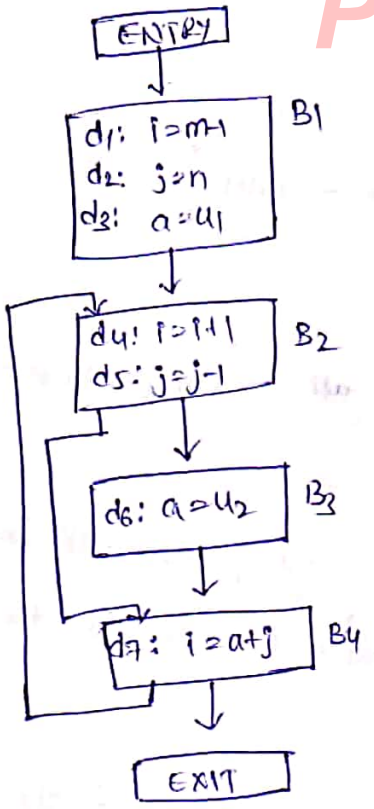
① $OUT[B] = \bigcup_{\text{Successor of } B} IN[S]$

② $IN[B] = USE[B] \cup (OUT[B] - DEF(B))$

$= \{c, d\} \cup \{a, b, c\} - \{a\}$

$= \{c, d\} \cup \{b, c\} = \{a, b, c, d\}$





$IN(B1) = \{m, n, u_1, u_2\}$
$def(B1) = \{i, j, a\}$
$use(B1) = \{m, n, u_1\}$
$OUT(B1) = \{i, j, u_2, a\}$

$IN(B_2) = \{i, j, u_2, a\}$
$def(B_2) = \{\}$
$use(B_2) = \{i, j\}$
$OUT(B_2) = \{u_2, a, i, j\}$

$IN(B_3) = \{j, u_2\}$
$def(B_3) = \{a\}$
$use(B_3) = \{u_2\}$
$OUT(B_3) = \{a, j, u_2\}$

$IN(B_4) = \{a, j, u_2\}$
$def(B_4) = \{i\}$
$use(B_4) = \{a, j\}$
$OUT(B_4) = \{a, i, j, u_2\}$

$IN(B_1) = \{m, n, u_1, u_2\}$

The variable m, n, u_1 and u_2 are used somewhere in the path / graph.

all variable are live

In $def(B_1)$ a is there, it will not present

$IN(B_1)$ as per the definition

$IN(B_1) = USE(B_1) \cup (OUT[B_1] - DEF[B_1])$

$IN[B_2] = USE[B_2] \cup (OUT[B_2] - DEF[B_2])$

$= \{i, j\} \cup \{OUT[B_2] - \{\}\}$

$= \{i, j\} \cup \{u_2, a\} = \{i, j, a, u_2\}$

if we follow the path $B_2 - B_4 - B_2$

'a' is used.

'a' will be added to $IN[B_2]$ also

IN[B3] = {j, u2}

def[B3] = {a}

use[B3] = {u1}

OUT[B3] = {a, j, u2}

IN[B4] = {a, j, u2}

Def[B4] = {i}

use[B4] = {a, j}

OUT[B4] = {a, i, j, u2}

Live-variable Analysis:

Input: A flowgraph with def and use computed for each block.

OUTPUT: IN[B] and OUT[B] the set of variables live on entry and exit of each blockB of the flowgraph.

Method:

IN[EXIT] = ϕ

for (each basic block B other than EXIT) IN[B] = ϕ

While (changes to any IN occur)

for (each basic block B otherthan EXIT)

{

$$OUT[B] = \bigcup_{S \text{ a succ of } B} IN[S];$$

$$IN[B] = use_B \cup (OUT[B] - def_B)$$

}

Iterative algorithm to Compute live variables.

Live-variable Analysis:

① domain ⟶ variables

② Transfer fun ⟶ Backward : $IN[B] = f_B(OUT[B])$

$$f_B(x) = USE_B \cup (x - DEF_B)$$

Meet-operation ⟶ $OUT[B] = \bigcup_S IN(S)$

Boundary Condition ⟶ $IN[EXIT] = \phi$

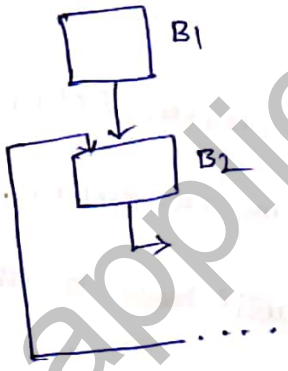
Initial Interior points ⟶ $IN[B] = \phi$

Applications of Data flow Analysis:

It is one of the widely used Application in code-optimization.

One of the applications

Available Expression Computation: ↘ Useful in Global Common SubExpression Elimination.



Define:

IN[Bi]

OUT[Bi]

Iterative Algorithm

Set of Expressions {i*j, k+4, i+k}

Set of expressions which are available at a in and out of particular block.

Similar to live variable analysis, here also we need to define multiple definitions like In and out for each block.

Available Expression Computation is a type of data flow Analysis will be useful to Eliminate global Common SubExpressions.

$IN[B_3] = \{j, u_2\}$

$def[B_3] = \{a\}$

$use[B_3] = \{u_1\}$

$OUT[B_3] = \{a, j, u_2\}$

$IN[B_4] = \{a, j, u_2\}$

$Def[B_4] = \{i\}$

$use[B_4] = \{a, j\}$

$OUT[B_4] = \{a, i, j, u_2\}$

Live-variable Analysis:

Input: A flowgraph with def and use computed for each block.

OUTPUT: IN[B] and OUT[B] the set of variables live on entry and exit of each block B of the flowgraph.

Method:

```
IN[EXIT] = ϕ
for (each basic block B other than EXIT) IN[B] = ϕ
    While (changes to any IN occur)
        for (each basic block B otherthan EXIT)
        {
            OUT[B] = ∪ IN[S];
                     S a succ of B
            IN[B] = use_B ∪ (OUT[B] - def_B)
        }
```

Iterative algorithm to compute live variables.

Constant propagation: can also be done using dataflow Analysis

↳ More Complex but doable.

It is also have IN and OUT and Iterative algorithm to perform Constant propagation, similar to Live variable Analysis, Available Expressions.

Solved problems: DFA and Code-optimization GATE-2015 Set-1

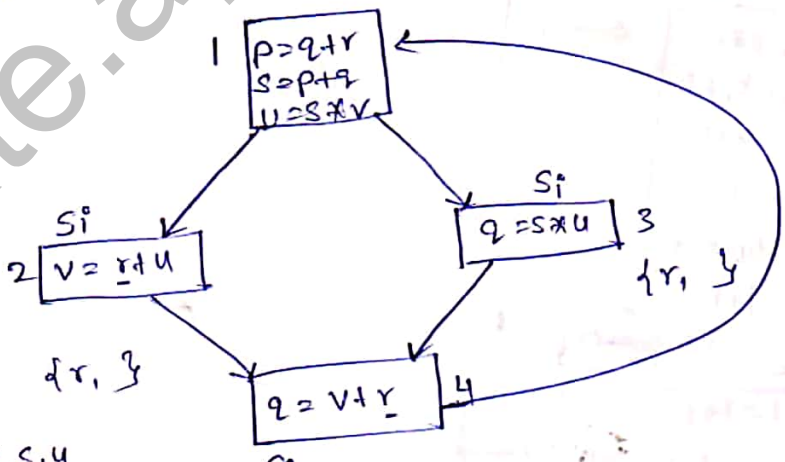
(1) A variable x is said to be live at a statement S_i in a program if the following three conditions hold simultaneously

(1) There Exist a statement S_j that uses x

(2) There is a path from S_i to S_j in the flow graph corresponding to the program.

(3) The path has no intervening assignment to x including at S_i and S_j

The variables which are live both at the statement in basic blocks and at the statement in basic blocks of the given control flow graph are



The given variables are

p, q, r, s, u, v

Let 2 is S_i

There is a path from S_i to S_j in which 'r' live on B_2 to B_4.

A: p, s, u

B: r, s, u

C: r, u

D: q, v

Similarly there is a path from B_3 to B_4 in which 'r' is live at B_4.

So, 'r' will be live on Both $B_2 - B_3$, $B_3 - B_4$

'r' must be there in the Solution.

option 'A' and 'D' Does not Contains 'r' as a live variable. we can eliminate them.

Let 'x' = 's' , $S_i = B_3$ and $S_j = B_1$

There is a path from B_3 to B_1 via B_4

But the variable 's' is redefined at B_1, before use. So s is not live from S_i to S_j ($B_3 - B_4 - B_1$).

option 'B' is also Eliminated.

The most suitable or appropriate answer is 'c' for the given problem.

(2) Consider the Intermediate code given below. GATE 2015 Set 2

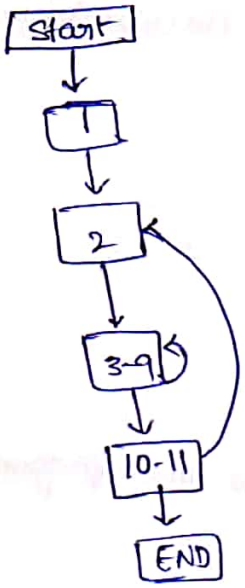
```
L4  1. i=1                    B1
L4  2. j=1
L2  3. t1=5*i
    4. t2=t1+j
    5. t3=4*t2
    6. t4=t3
    7. a[t4]=-1
    8. j=j+1
    9. if j<=5 goto (3)
L3  10. i=i+1
    11. if i<5 goto (2)
```

The number of nodes and Edges in the Control-flow-graph Constructed for the above code respectively are

① First Statement is always a Leader.

② Target of any Conditional (or) unconditional goto is a Leader.

③ The next Statement that immediately follows Conditional (or) unconditional goto is a leader.



Control-flow graph

Total number of nody are 6 and the Edges are '7'.

option 'B' is the most suitable Answer.

③ Which of the following is FALSE ? GATE 2014 set-1

(A) A basic block is a sequence of instructions where control Enters the sequence at the begining and exits at the end.

(B) Available Expression analysis can be used for common subexpression elimination.

(C) Live variable analysis can be used for dead code Elimination.

(D) $x = 4 * 5 \Rightarrow x = 20$ is an Example of Common Subexpression elimination.

(A): True

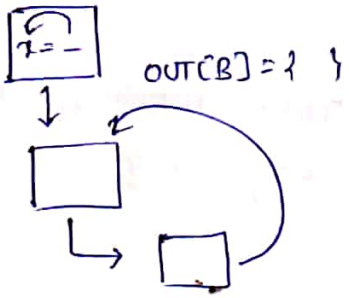
It is the definition of a Basic Block.

(B): True

It is used for Common SubExpression Elimination.

(C): True.

We can identify and eliminate the dead code using live variable Analysis.



The variable x is not used in the program, then it results in a dead code.

(D) FALSE

$x = 4 * 5$ → Constant

$\Rightarrow x = 20$ it is not the example of Common SubExpression.

$x = 4 * y$

$z = 4 * y + 5$ $\Rightarrow$ $z = x + 5$

Common SubExpression.

(4) GATE-2006

Consider the following code segment

```
for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
     {
       if (i%2)
        {
          x += (4*j + 5*i);
          y += (7 + 4*j);
        }
     }
```

Which one of the following is False?

(A) The code contains loop invariant computation.

(B) There is scope of Common Sub Expression Elimination in this code.

(C) There is scope of strength Reduction in this code.

(D) There is scope of dead code Elimination in this code.

A. True
$(5*i)$ will not impact inner loop. we can move to outside of the loop.

B. True
$4*j$ is the common Sub Expression. We can also Eliminate it from the loop.

C. $4*j == j<<2$ True
↓
costly operation
Left shifting by '2' will also done the same job.

D. False
NO scope of dead code elimination for the given code.

GATE-2008

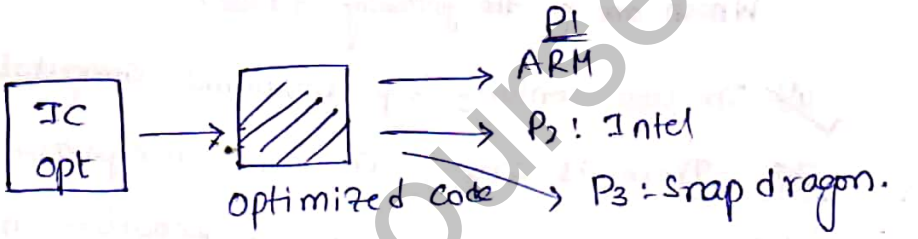
(5) Some code optimizations are carried out on the Intermediate code because

(A). They enhance the portability of the compiler to other target processor.

✗ (B). Program Analysis is more accurate on intermediate code than on machine code.

✗ (C). The information from data flow Analysis cannot otherwise be used for optimization.

✗ (D). The information from the front end cannot otherwise be used for optimization.



A seems to be suitable Answer.

(B): Not necessarily true always.

(C): The information from data flow Analysis will be used for optimization.

(D): The information from the front will also be used for optimization.

A is the right answer for this given problem.