Basic blocks and flow graphs

Basic Block is a set of statements that always executes one after other, in a sequence.
Basic Block is a straight line code sequence that has no branches in and out except to the entry and at the end respectively.

Flow graph is the graph representation of intermediate code.

NOTE:-A compiler first converts the source code of any programming language into an intermediate code. It is then converted into basic blocks. After partitioning an intermediate code into basic blocks, the flow of control among basic blocks is represented by a flow graph.

Properties of Basic Block:-

- Contains one entry and one exit.
- Middle of block should not contain conditional statements (if , else) or unconditional control statements (break, continue, exit , goto).
- Both conditional statements & unconditional statements can appear at beginning & end of block

Partitioning three-address codes into basic blocks

- The task is to partition a sequence of three-address codes into the basic block.
- The new basic block always begins with the first instruction and continues to add instructions until a jump or a label is reached.
- If no jumps or labels are identified, control will flow sequentially from one instruction to another.

Rule1:- Determine the leader

a. First statement is a leader
b. The target of the conditional or unconditional statement is a leader.
c. The statement following conditional or unconditional statement is a leader.

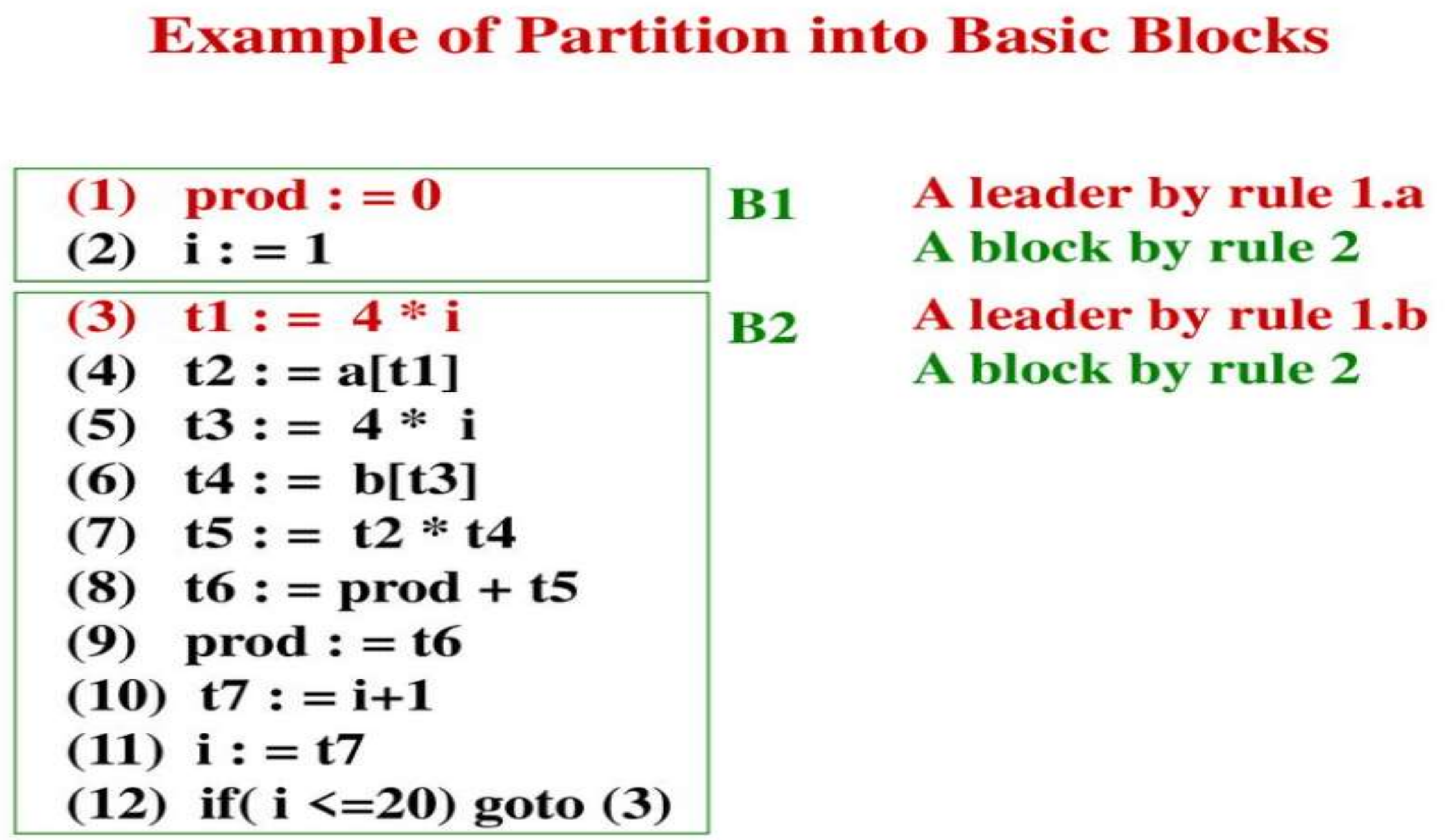
Rule2:- Determine the Basic blocks.

a. Begins of leader statement
b. Ends at the statement before the next leader.

Algorithm

1. First, find the set of leaders from intermediate code, the first statements of basic blocks. The following are the steps for finding leaders:
   a. The first instruction of the three-address code is a leader.
   b. Instructions that are the target of conditional/unconditional goto are leaders.
   c. Instructions that immediately follow any conditional/unconditional goto/jump statements are leaders.
2. For each leader found, its basic block contains itself and all instructions up to the next leader.

Partitioning three-address codes into basic blocks



Example of Partition into Basic Blocks

B1

```
(1)  prod : = 0
(2)  i : = 1
```

A leader by rule 1.a

A block by rule 2

B2

```
(3)  t1 : =  4 * i
(4)  t2 : = a[t1]
(5)  t3 : =  4 *  i
(6)  t4 : =  b[t3]
(7)  t5 : =  t2 * t4
(8)  t6 : = prod + t5
(9)  prod : = t6
(10)  t7 : = i+1
(11)  i : = t7
(12)  if( i <=20) goto (3)
```

A leader by rule 1.b

A block by rule 2

Flow graph

Flow graph is a directed graph in which flow control is added.
In flow graph

Properties of Flow Graphs
The control flow graph is process-oriented.
A control flow graph shows how program control is parsed among the blocks.
The control flow graph depicts all of the paths that can be traversed during the execution of a program.
It can be used in software optimization to find unwanted loops.
s

Optimization of basic blocks

The two different scenarios of Block Optimization are as given below

1. Structure-Preserving Transformations

a) Elimination of common sub-expression

b) Elimination of Dead Code

c) Renaming the temporary variables s

d) Interchanging two adjacent independent statements

2. Algebraic Transformations

Optimization of basic blocks

1. Structure-Preserving Transformations

a)Elimination of Common Sub-Expression

In this elimination technique, there is no need to repeatedly compute the same sequence expression. When the same expression is repeated, the previous computation evaluates the result.

Example:-

w: x+y

x: w-z //

y: x+y

z: w-z //

s

After optimization

w: x+y

x: w-z

y: x+y

z: x // since same expression (w-z) is already evaluated hence replace the expression with the LHS (x)

Optimization of basic blocks

1. Structure-Preserving Transformations
b)Elimination of Dead code

Dead code is a code that never gets executed or not used.

Example:-

```
a=0
If(a==1)
   a++;
```

After optimization

```
a=0
```

Optimization of basic blocks

1. Structure-Preserving Transformations

c) Renaming the temporary variables

Rename the temporary variable if more than one LHS are same to avoid overwriting problem.

Example:-

```
T1 =b + c    //
T2=a - T1
T1=T1 *d    //
D=T2 + T1
```

s

Bothe 1 & 2 are having same LHS , so get ride of overwriting we use different temporary variables

After optimization

```
T1 =b + c
T2=a - T1
T3=T1 *d
D=T2 + T3
```

Optimization of basic blocks

1. Structure-Preserving Transformations

d) Interchanging two adjacent independent statements

<u>Example:-</u>

```
T1 =b + c
T2=a - T1   //
T3=T1 *d   //
D=T2 + T3
```

s

here T2 & T3 are independent statements so they can be interchanged.

<u>After optimization</u>

```
T1 =b + c
T3=T1 *d
T2=a - T1
D=T2 + T3
```

Optimization of basic blocks

2. Algebraic Transformations, useless operations are deleted

Example:-

```
T1=a-a      //  a-a =0 , no need of performing minus operation
T2=b-T1    //  b-0 = b , no need of performing minus operation
T3=T2*2   // * is a costly operation, so use left shift operator instead of multiplication operator.
```

After optimization

```
T1=0
T2=b
T3=T2 << 2
```

Principle Sources of Optimization

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following **objectives :-**

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

S

1. Compile Time Evaluation
   a) Folding
   b) Propagation
2. Common sub Expression Elimination
3. Variable Propagation
4. Code movement
5. Strength Reduction
6. Dead Code Elimination

Principle Sources of Optimization

1. **Compile Time Evaluation**
   a) **Folding**, technique of computing constants at compile time instead of execution time.
      Ex:- length=(22/7) &d // here constant calculation is 22/7 , computed at compile time.
   a) **Propagation**, technique of computing expressions at compile time.
      Ex:- Area= pi * r * r; // here expression the pi*r*r, computed at compile time.

2. Common Sub Expression Elimination, if a common sub expression is appearing repeatedly in program then its result is used instead of recompiling it each time.

3. Variable Propagation , is a technique of using one variable instead of another.

```
Ex:- x= pi;
     area=x *r*r; // here instead of pi we are using x variable.
```

4. Code Movement, it is a technique of reducing size of code i.e, to obtain space complexity
reducing the frequency of execution i.e, to obtain time complexity.

Ex:-

```
x=y+5;                    temp=y+5;
k=(y*5)+50;      →        x=temp;
                          k=(temp)+50;
```

Principle Sources of Optimization

5. Strength Reduction, technique in which higher strength operator can be replaced by lower strength operator.

Ex:-

```
for(i=1;i<=50;i++)
{
count=i*7;
}
```

Here we get the count values 7, 14, 21 ….. upto 50

```
temp=7;
for(i=1;i<=50;i++)
{
count=temp;
temp=temp+7;
}
```

s

6. Dead Code Elimination

Dead code is a code that never gets executed or not used.

Example:-

```
a=0
If(a==1)
   a++;
```

After optimization

```
a=0
```

Loop Optimization

Loop Optimization is a code optimization technique that focuses on improving the performance of loops in computer programs. It aims to reduce the number of operations or iterations required to complete a loop.

NOTE:- *Loop Optimization is a machine independent optimization.*

1. **Code Motion**
2. **Induction Variable Elimination & Strength Reduction**

S

3. **Loop Invariant Method**
4. **Loop Unrolling**
5. **Loop Fission**

Loop Optimization

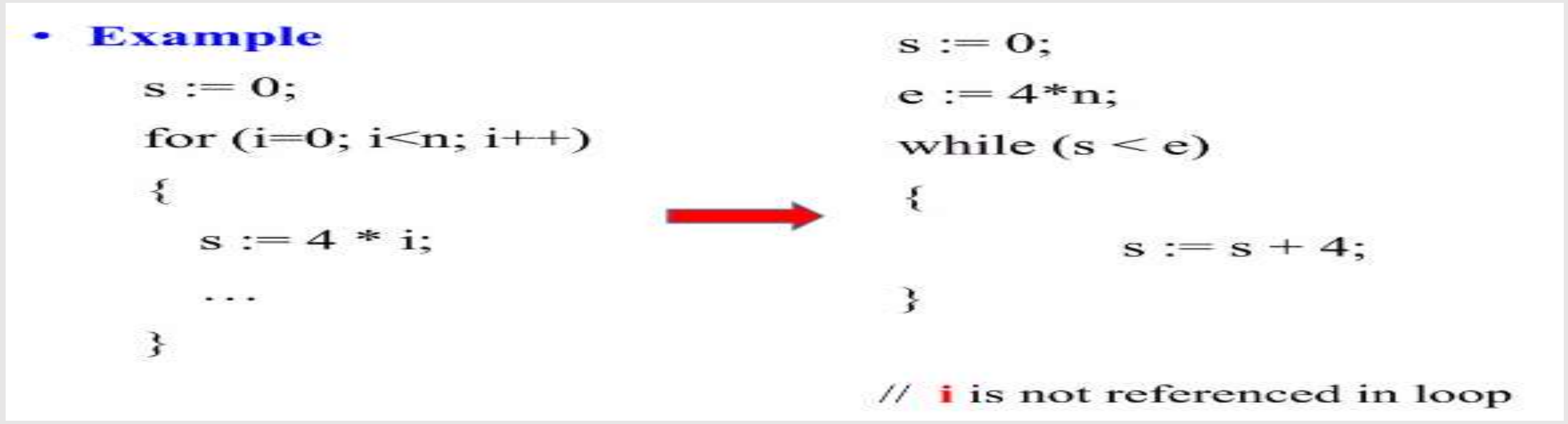
1. **Code Motion** , technique which moves the code outside the loop.

```
Ex:-while(i<=Max-1)
{
    sum=sum + a[i];
}
```

```
n=Max-1;
 while(i<=n)
{
    sum=sum + a[i];
}
```

2. Induction Variable Elimination & Strength Reduction

A variable x is called an induction variable of loop L if the value of the variable gets changed every time. It is either incremented or decremented by some constant.



- **Example**

```
s := 0;
for (i=0; i<n; i++)
{
    s := 4 * i;
    ...
}
```

```
s := 0;
e := 4*n;
while (s < e)
{
        s := s + 4;
}

// i is not referenced in loop
```

Loop Optimization

Strength Reduction, technique in which higher strength operator can be replaced by lower strength operator.

Ex:-

```
for(i=1;i<=50;i++)
{
count=i*7;
}
```

Here we get the count values 7, 14, 21 ….. upto 50

higher strength operator (*) is replaced by lower strength operator (+)

```
temp=7;
for(i=1;i<=50;i++)
{
count=temp;
temp=temp+7;
}
```

3. Loop Invariant Method, it's a optimization technique to avoid overhead of computation inside the loop.

Ex:-

```
void action(int[] arr, int b, int c) {

for (int i = 0; i < arr.length; i++) {
        int a = b + c; // overhead of computation
        int result = arr[i] * a;
}
}
```

```
void action(int[] arr, int b, int c) {
int a = b + c;
for (int i = 0; i < arr.length; i++) {

int result = arr[i] * a;
}
}
```

Loop Optimization

4. Loop Unrolling, technique used to reduce the number of iterations of a loop.

Ex:-

```
for(int i=1; i<=10; i++)
{
  print(“Hello”);
}
```

Here number of iterations are reduce to 5 from 10 iterations

```
for(int i=1; i<=10; i+2)
{
  print(“Hello”);
  print(“Hello”);
}
```

5. Loop Fission, technique in which several loops are merged into one.

Ex:-

```
for (i = 0; i < 300; i++)
  a[i] = a[i] + 3;

for (i = 0; i < 300; i++)
  b[i] = b[i] + 4;
```

Here two adjacent loops are fused into one loop.

```
for (i = 0; i < 300; i++)
 {
   a[i] = a[i] + 3;
   b[i] = b[i] + 4;
 }
```

DAG representation of basic blocks

Basic block is a set of statements that execute one after another in sequence.

DAG representation or Directed Acyclic Graph representation is used to represent the structure of basic blocks.

- It displays the flow of values between basic blocks and provides basic block optimization algorithms.
- It is an efficient way of identifying common subexpressions(the size of tree can be reduced by finding the common subexpressions).
- Like syntax tree DAG also contains internal nodes and leaf nodes.
- **Internal nodes** represents operators or results of expressions
- **Leaf nodes** represents identifiers or constants.

NOTE:- syntax tree and DAG both are graphical representations. Syntax tree does not find common subexpressions where as DAG can.

DAG representation of basic blocks

Applications of DAG:-

1. Determines the common subexpressions
2. Determines which names are used inside the block and computed outside the block
3. Determines which statements of the block could have their computed value outside the block.
4. Simplify the list of quadruples by eliminating common subexpressions.
5. Dag is used to apply optimization technique on basic block. For this purpose DAG constructs 3address code for intermediate code generation

3 ADDRESS CODE:- It's a type of intermediate code which is easy to generate and can easily be converted to machine code.

- 3 address code makes use of at most 3 addresses & 1 operator to represent expression.
- And value computed is stored in temporary variable

Ex:- T1 = a + b // here 3 addresses T1, a, b and 1 operator + is used in the code

DAG representation of basic blocks

Algorithm for constructing DAG

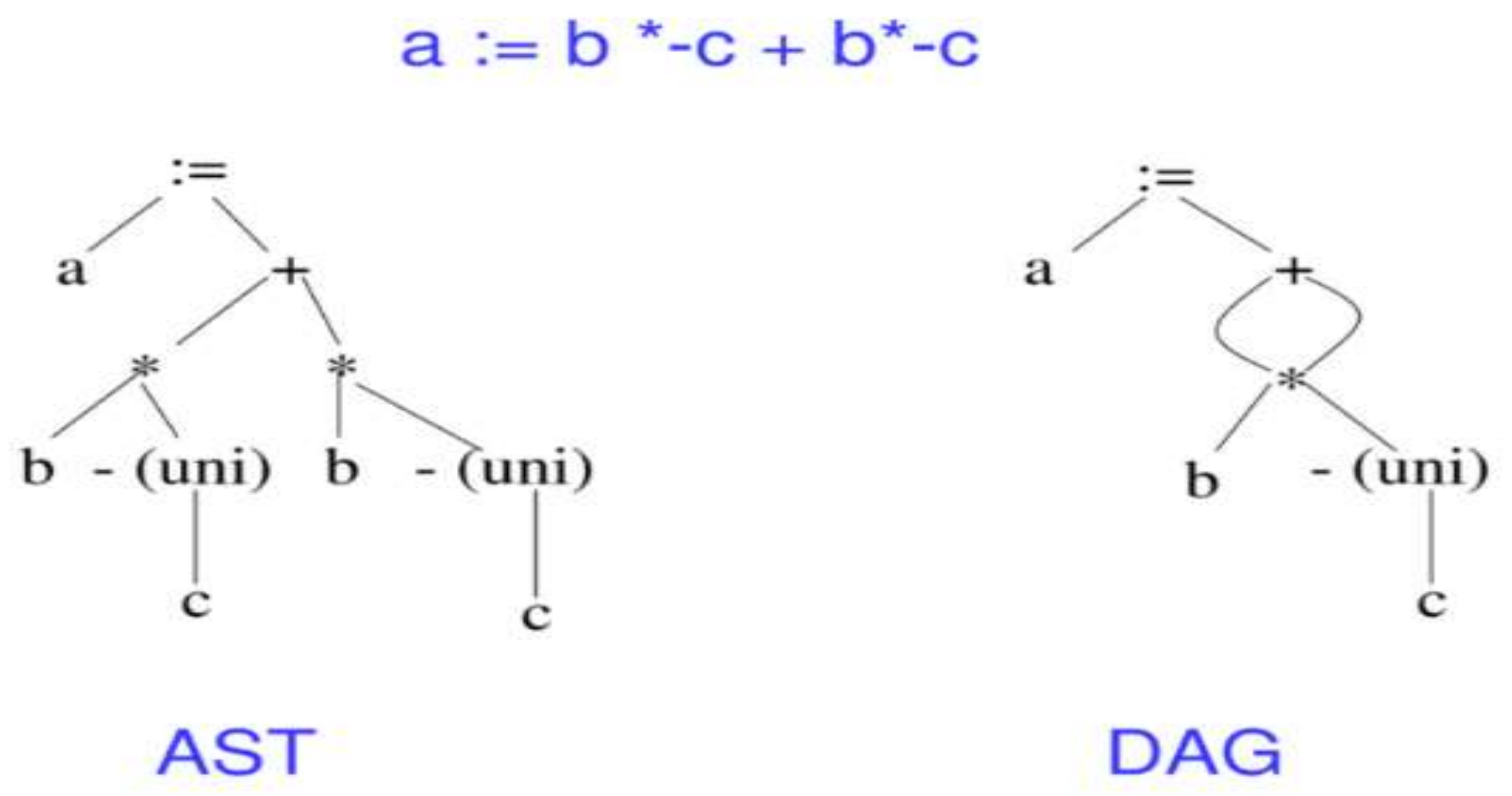
Input- The input will contain a basic block.

Output- The output will contain the following information-

1. Each node of the graph represents a label.
    1. If the node is a leaf node, the label represents an identifier.
    2. If the node is a non-leaf node, the label represents an operator.
2. Each node contains a list of attached identifiers to hold the computed value.

DAG representation of basic blocks

Example:-



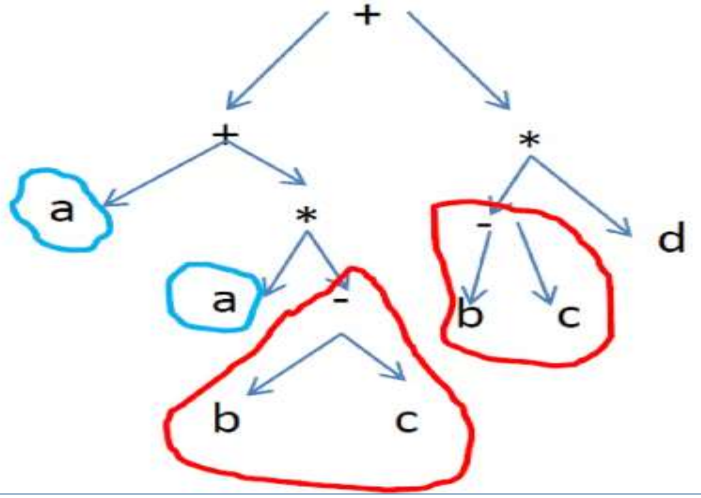
DAG representation of basic blocks

Syntax tree vs. DAG vs. Three address code

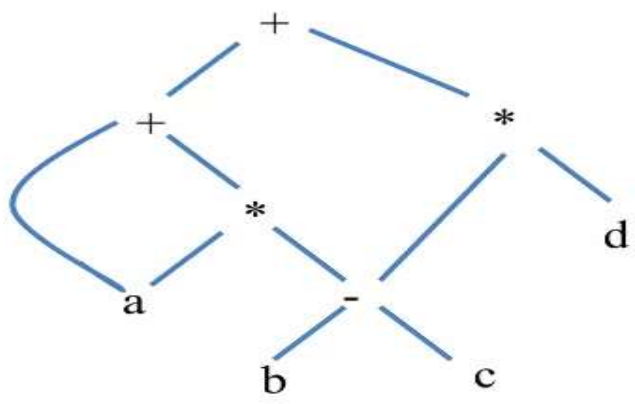
- AST is the procedure's parse tree with the nodes for most non-terminal symbols removed.
- Directed Acyclic Graph is an AST with a unique node for each value.
- Three address code is a sequence of statements of the general form **x := y op z**
- In a TAC there is at most one operator at the right side of an instruction.
- Example:

a+a*(b-c)+(b-c)*d

AST



DAG



TAC

t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

Simple Code Generator

A code generator is a compiler that translates the intermediate representation of the source program into the target program. In other words, a code generator translates an abstract syntax tree into machine-dependent executable code.

The process of generating machine-dependent output from an abstract syntax tree involves two steps:

1. one for constructing the abstract syntax tree and
2. second for generating its corresponding machine code.

It uses getReg() function to assign registers to variables.

s

It uses two data structures

1. **Register Descriptor,** used to track which variable is stored in a register.

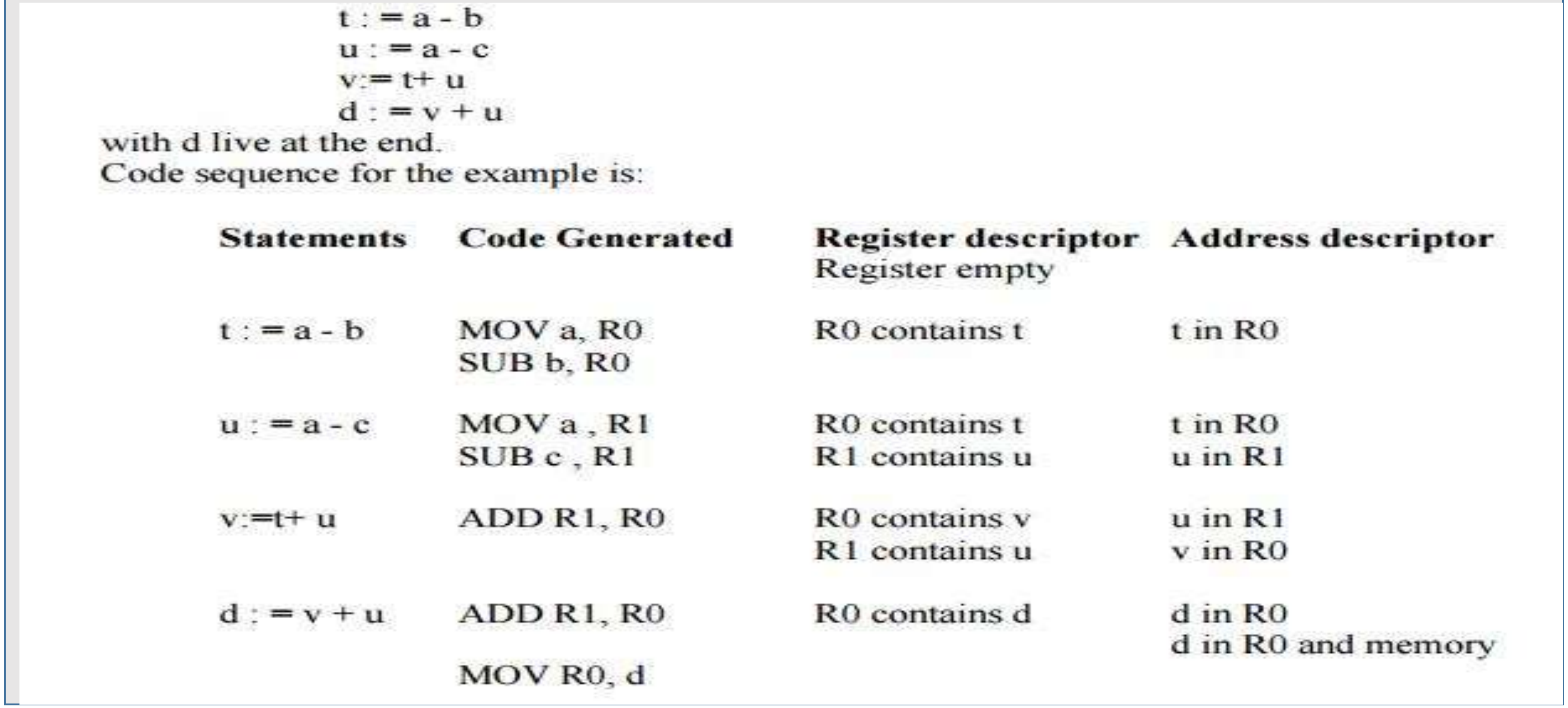
2. **Address Descriptor,** used to keep track of location where variable is stored.
   location can be a register or memory address stack.

Simple Code Generator

Example:-

d : = (a-b) + (a-c) + (a-c)

First translate the expression in 3 address code sequence



```
t : = a - b
u : = a - c
v:= t+ u
d : = v + u
```

with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0	R0 contains d	d in R0
	MOV R0, d		d in R0 and memory

Register Allocation and Assignment

- While programming for store data purpose memory like RAM or registers can be used.
- But accessing RAM is significantly slower than accessing registers and slows down the execution speed of the compiled program, so an optimizing compiler aims to assign as many variables to registers as possible.
- But during compilation, the compiler must decide how to allocate these variables to a small, finite set of registers.
- Not all variables are in use (or "live") at the same time, so some registers may be assigned to more than one variable.
- Variables which cannot be assigned to some register must be kept in RAM and loaded in/out for every read/write, a process called spilling, whereas the reverse procedure of moving a variable from memory to a register is known as filling.

NOTE:-Common approach of register allocation is to **assign specific values to specific registers**.
Ex:- separate set of registers allocated for base registers, stack pointers, arithmetic computations ect.

--**Advantage** is simplified design of compiler for code generation is.
--**Disadvantage** is complicated design of compiler for restrictive use of registers.

Register Allocation and Assignment

<u>Strategies for Register Allocation and Assignment</u>

1. Global register allocation
2. Usage count
3. Register assignment for outer loop
4. Graph coloring for register assignment

<u>1. Global register allocation</u>

s

There are two types of register allocation:
a) Local register allocation: This is a process of allocating one basic block (or hyper block or super block) at a time. Local register allocation boosts speed.
b) Global register allocation: If the register utilization is poor using local allocation, it is important to make use of global register allocation. In simple global register allocation, the most active values are allocated in every inner loop. Full global register allocation uses a procedure to identify live ranges in a control flow graph, assign live ranges and also split ranges as required.

Register Allocation and Assignment

1. Global register allocation

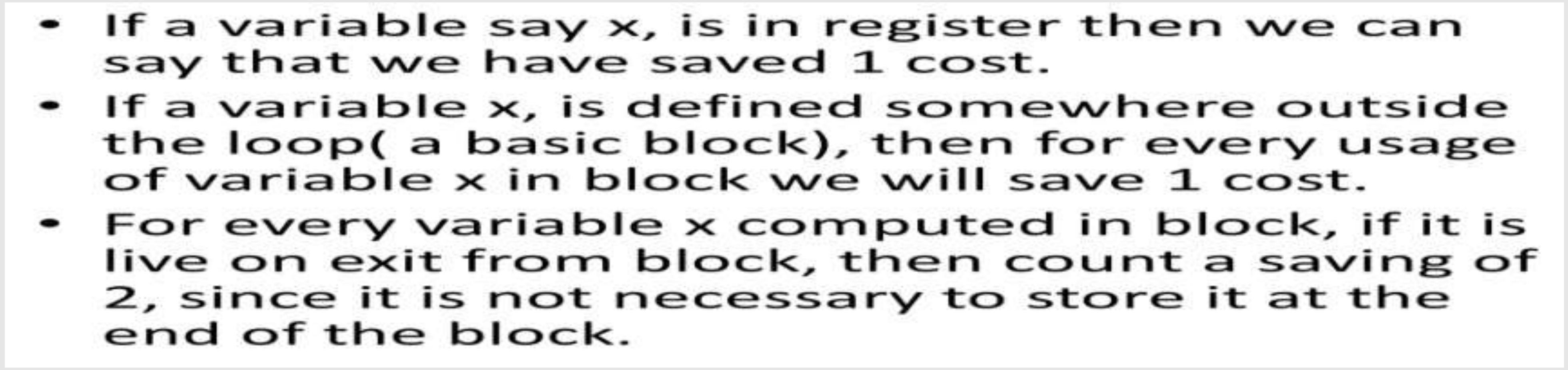
In simple global register allocation, the most active values are allocated in every inner loop.
Full global register allocation uses a procedure to identify live ranges in a control flow graph, assign live ranges and also split ranges as required.

Following are strategies adopted while doing global register allocation:-

- Allocate most frequently used variables in fixed registers throughout the loop.
- Assign some fixed number of global registers to hold the most active values.
- Registers not yet allocated can be used to hold values local to one block
- For certain languages like C , register allocation can be done by using register declaration

Register Allocation and Assignment

2. Usage Count:- , it refers to the unit of saving by register allocation



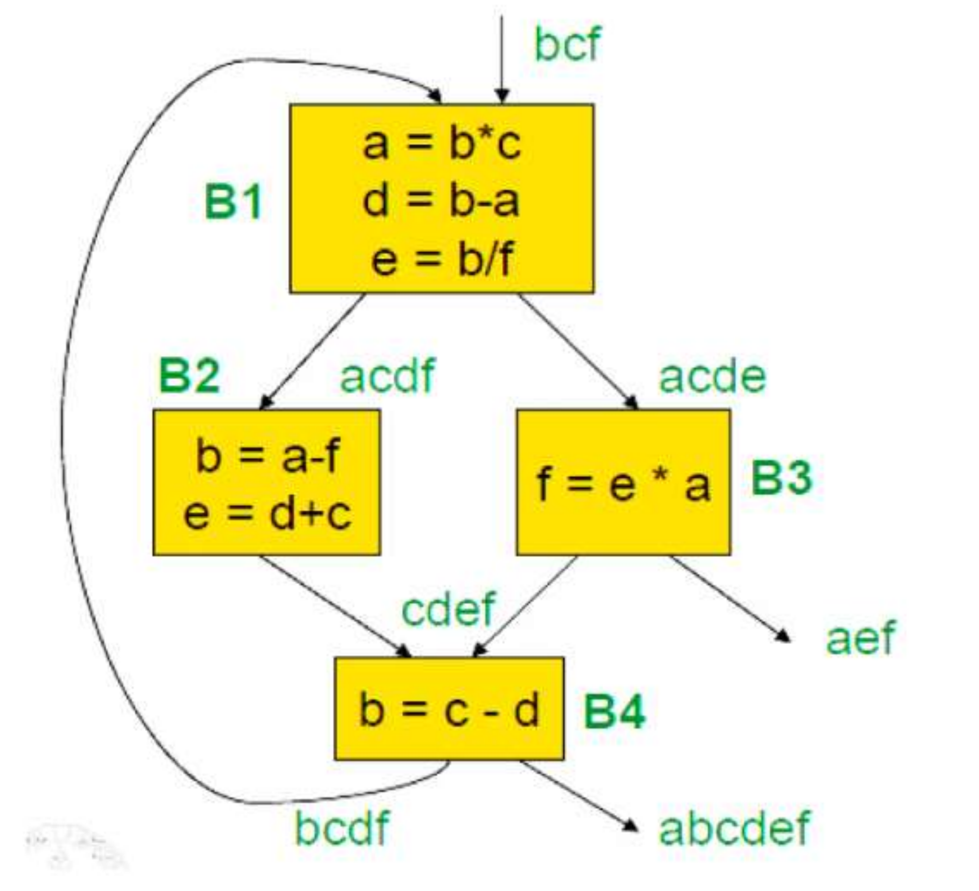
- If a variable say x, is in register then we can say that we have saved 1 cost.
- If a variable x, is defined somewhere outside the loop(a basic block), then for every usage of variable x in block we will save 1 cost.
- For every variable x computed in block, if it is live on exit from block, then count a saving of 2, since it is not necessary to store it at the end of the block.

An approximate formula for the benefit to be realized from allocating a register to x within a loop L is:

$$\boxed{\sum_{blocks\ B\ in\ L} (use(x, B) + 2 * live(x, B))}$$

where,
-use(x, B) is the number of times x is used in B prior to any definition of x;
-live(x, B) is 1 if x is live on exit from B and is assigned a value in B and 0 otherwise.

Register Allocation and Assignment

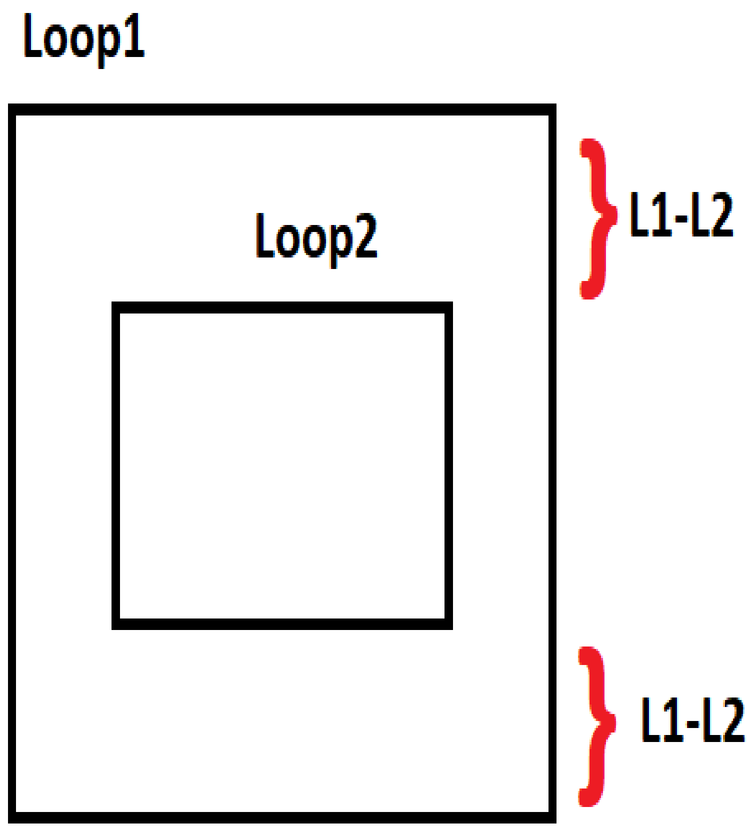


	B1		B2		B3		B4	
a:	(0+2)	+	(1+0)	+	(1+0)	+	(0+0)	= 4
b:	(3+0)	+	(0+0)	+	(0+0)	+	(0+2)	= 5
c:	(1+0)	+	(1+0)	+	(0+0)	+	(1+0)	= 3
d:	(0+2)	+	(1+0)	+	(0+0)	+	(1+0)	= 4
e:	(0+2)	+	(0+2)	+	(1+0)	+	(0+0)	= 5
f:	(1+0)	+	(1+0)	+	(0+2)	+	(0+0)	= 4

If there are 3 registers available R1, R2 & R3.
Then the highest usage count variables are allocated first i.e, variables b & e are allotted with R1 & R2.
And variables a ,d & f are allotted with r3s

Register Allocation and Assignment

3. Register assignment for outer loop



- If an outer loop L1 contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1-L2.
- However if name x is allocated a register in loop L1 but not in L2, we must store x on entrance to L2 and load x on exit from L2.
- Similarly, if we choose to allocate x a register in L2 but not L1, we must load x on entrance to *L2* and store x on exit from L2.

Register Allocation and Assignment

4. Graph coloring for register assignment
when a register is needed for construction but all registers are in use , the contents of one register must be stored in memory location.

Two passes are used

1. In first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.
2. In the second pass the register inference graph is prepared. In register inference graph each node is a symbolic register and an **edge** connects two nodes where one is live at a point where other is defined.

.

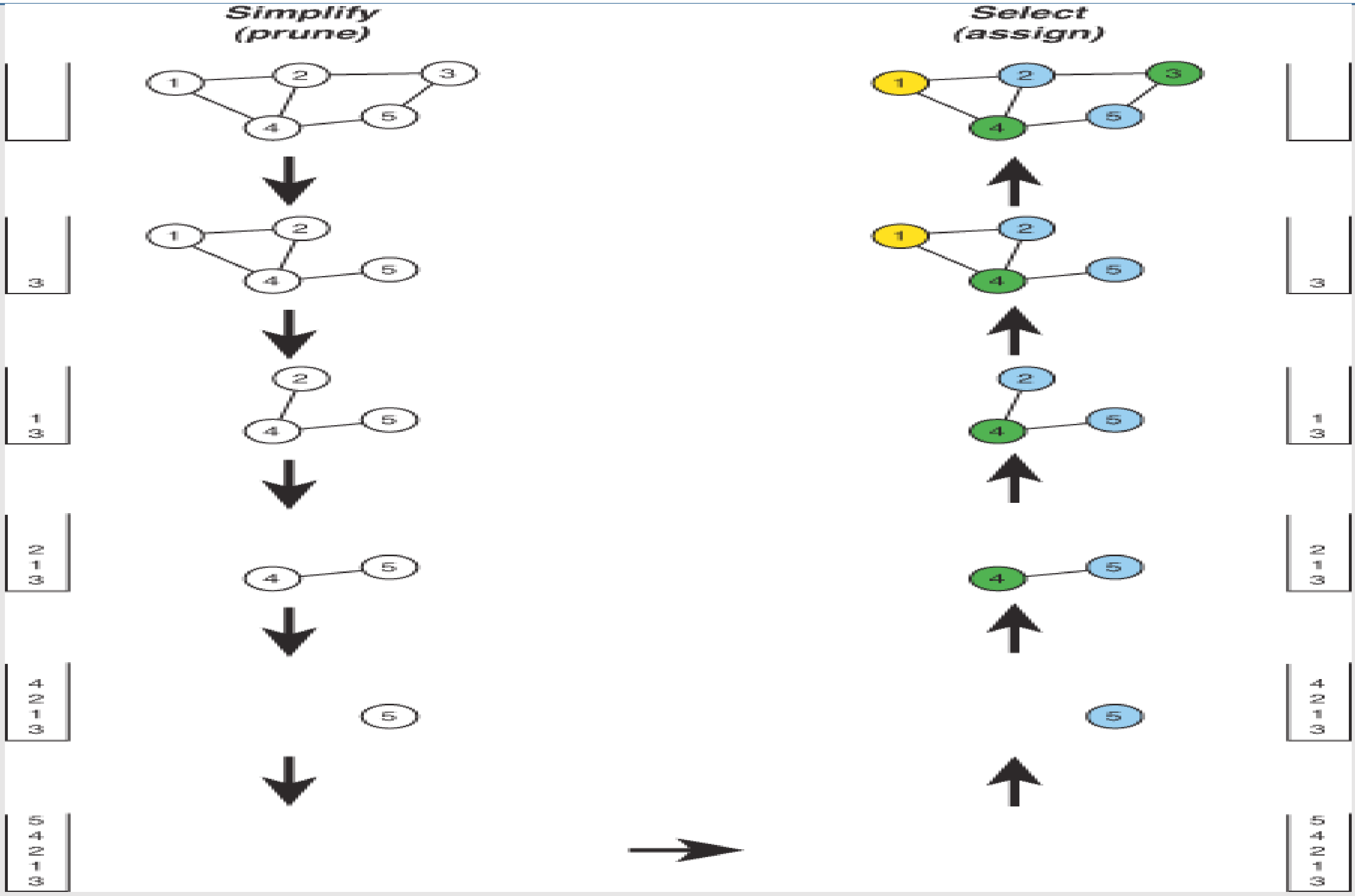
Register Allocation and Assignment

The problem of Register Allocation can be viewed as a graph coloring problem. Suppose that we have ‘n’ available registers: r1, r2... rn; if we view each register as a different color, then the register allocation problem is equivalent to the graph coloring problem where we try to assign one of the ‘n’ different colors to the graph nodes. This way, no two adjacent nodes will have the same color.

The core of the coloring process itself starts with the *simplify* phase, sometimes called *pruning*. Here, the graph is repeatedly examined and nodes with fewer than *k* neighbors are removed (where *k* is the number of colors we have to offer). As each node is removed it is placed on a stack and its edges are removed from the graph, thereby decreasing the degree of interference of its neighbors.

Finally, the *select* phase actually selects a color (register) for each node. This is done by repeatedly popping a node off the stack, re-inserting it into the graph, and assigning it a color different from all of its neighbors.

Register Allocation and Assignment



Peephole Optimization

Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.

The small set of instructions or small part of code on which peephole optimization is performed is known as **peephole or window**.

It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output.

The peephole is **machine-dependent optimization** s

Peephole optimization is to improve performance, reduce memory footprint and to reduce code size

Types:-

A. Redundant load and store elimination:
B. Flow of control optimization
C. Algebraic simplification
D. Reduction in strength
E. Machine idioms

Peephole Optimization

A. Redundant load and store elimination:-

In this technique the redundancy is eliminated. For e.g.

Ex:-

```
y = x + 5;
i = y;
z = i;
w = z * 3;
```

$\longrightarrow$

```
y = x + 5;

i = y;
w = y * 3;
```

B. **Flow of control optimization**

Optimizing unnecessary jumps on jumps are eliminated.

Ex:-

```
goto L1
L1: goto L2
……..
L2: goto L3
………..
L3: Mov a, R3
```

As multiple jumps make the code inefficient

$\longrightarrow$

```
goto L3
L1: goto L3
……..
L2: goto L3
………..
L3: Mov a, R3
```

Peephole Optimization

C. Algebraic simplification
useless operations are deleted

Example:-

```
T1=a-a      //  a-a =0 , no need of performing minus operation
T2=b-T1    //  b-0 = b , no need of performing minus operation
```

After optimization

s

```
T1=0
T2=b
```

Peephole Optimization

D. Reduction in strength

Strength Reduction, technique in which higher strength operator can be replaced by lower strength operator.

Ex:-

```
for(i=1;i<=50;i++)
{
count=i*7;
}
```

Here we get the count values 7, 14, 21 ….. upto 50

higher strength operator (*) is replaced by lower strength operator (+)

```
temp=7;
for(i=1;i<=50;i++)
{
count=temp;
temp=temp+7;
}
```

E. Machine idioms

The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.

Ex:- some machine have auto-increment or auto-decrement addressing modes that are used to perform add or subtract operations.

Instead of writing a= a + 1; $\longrightarrow$ we can write inc a;