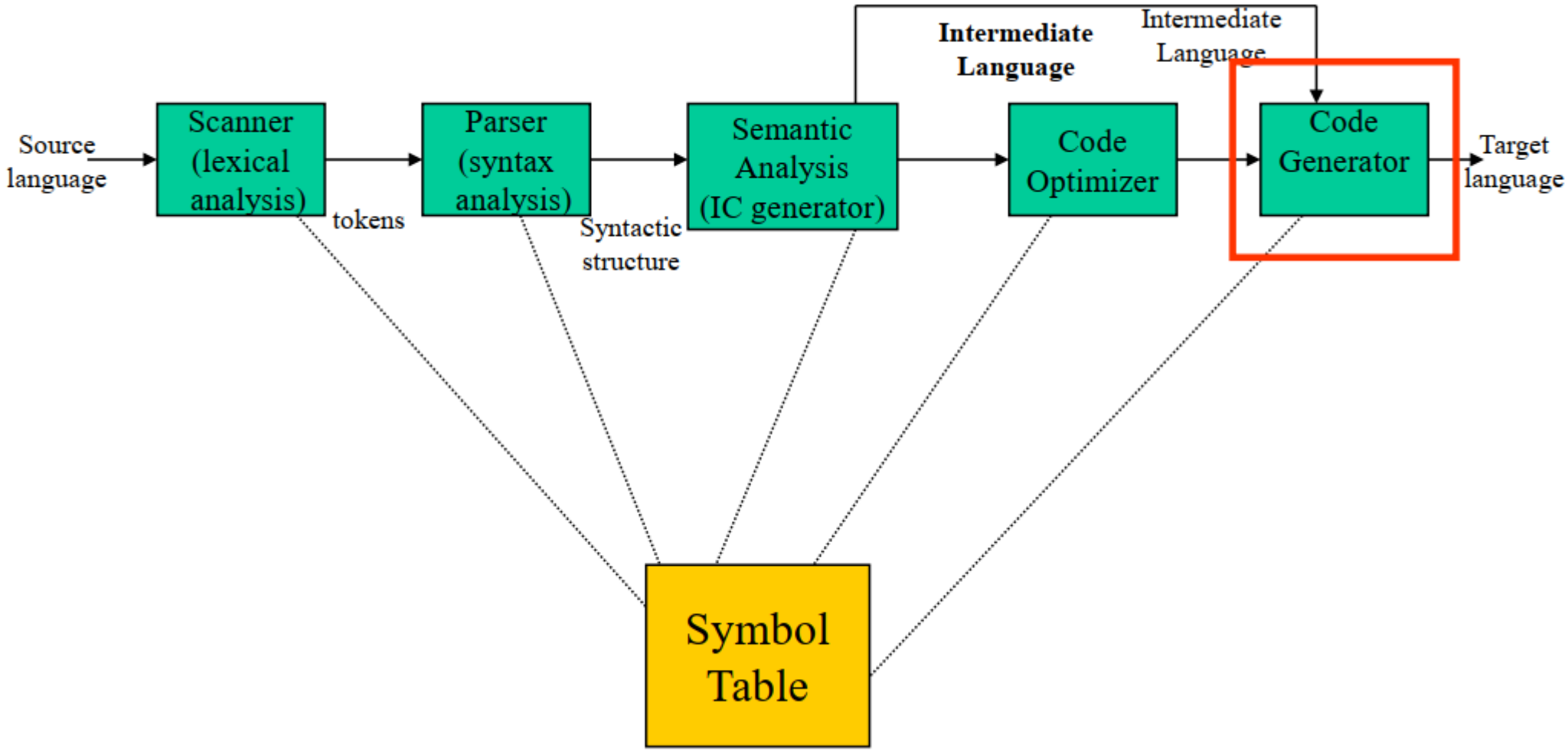
Code Optimization and Generation


Source language
Scanner (lexical analysis)
tokens
Parser (syntax analysis)
Syntactic structure
Semantic Analysis (IC generator)
Intermediate Language
Intermediate Language
Code Optimizer
Code Generator
Target language
Symbol Table

Code 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. A code optimizing process must satisfies the three main criterion:
  - Must preserve 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.

Types of Code Optimization

- The optimization process can be broadly classified into two types:
  - 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.
  - 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.

Terms used in Code Optimization

- Basic Block
  - Basic Block is a set of statements which always executes one after other, in a sequence.
  - 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.
- Flow Graph
  - Flow graph is a directed graph. It contains the flow of control information for the set of basic block.
  - A control flow graph is used to depict that how the program control is being parsed among the blocks. It is useful in the loop optimization.

Terms used in Code Optimization

- Steps for finding Basic Block
  - Determine the leader statements. Rules for finding the leader statements are as follows:
    - First statement of the code will always be leader.
    - Statement that is a target of the conditional or unconditional goto statement will always be leader.
    - Statement that appears immediately after a goto statement will always be leader.
  - All the statements that follow the leader (including the leader) till the next leader appears form one basic block.
  - Block having first statement as leader is called initial block.

Terms used in Code Optimization

1. Prod=0;
2. i=1;
3. t_1=4*I;
4. t_2=a[t_1]
5. t_3=4*I;
6. t_4=a[t_3]
7. t_5=t_3*t_4
8. Prod=Prod+t_5
9. i=i+1
10. If(i<=20) goto (3)

Terms used in Code Optimization

1. Prod=0;
2. i=1;
3. t_1=4*I;
4. t_2=a[t_1]
5. t_3=4*I;
6. t_4=a[t_3]
7. t_5=t_3*t_4
8. Prod=Prod+t_5
9. i=i+1
10. If(i<=20) goto (3)

In this code, statement 1 and 3 are leaders.

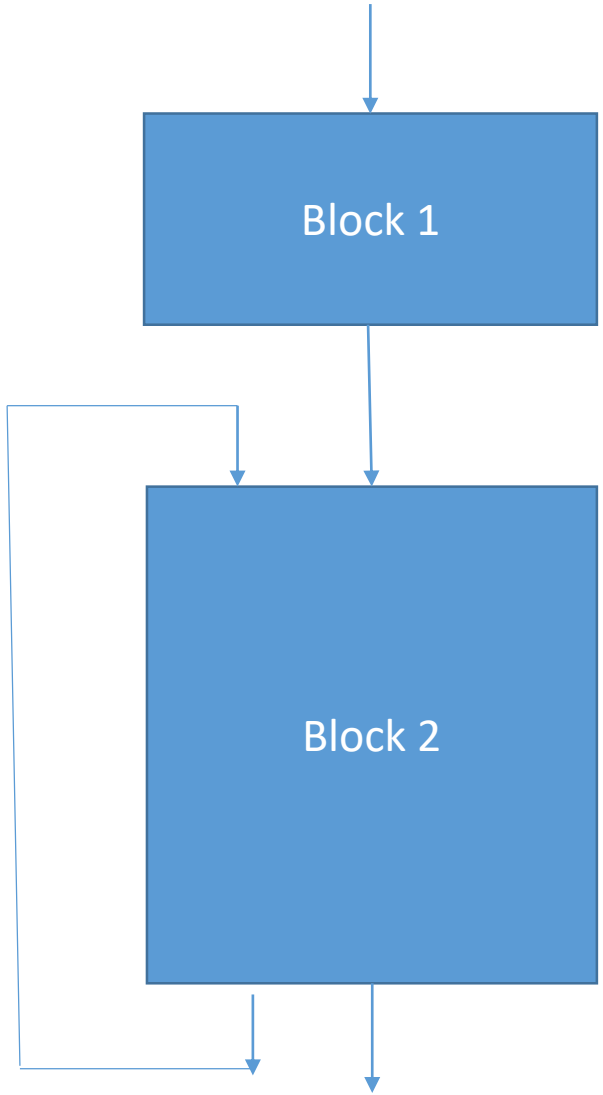
So in Block 1, statement 1,2 while in Block 2, statement 3 to 10 will be there

Terms used in Code Optimization

1. Prod=0;
2. i=1;
3. t_1=4*I;
4. t_2=a[t_1]
5. t_3=4*I;
6. t_4=a[t_3]
7. t_5=t_3*t_4
8. Prod=Prod+t_5
9. i=i+1
10. If(i<=20) goto (3)

In this code, statement 1 and 3 are leaders.

So in Block 1, statement 1,2 while in Block 2, statement 3 to 10 will be there



Terms used in Code Optimization

```
1.   i=1
2.   j=1
3.   t1 = 10 * i
4.   t2 = t1 + j
5.   t3 = 8 * t2
6.   t4 = t3 - 88
7.   a[t4] = 0.0
8.   j = j + 1
9.   if j <= goto (3)
10.  i = i + 1
11.  if i <= 10 goto (2)
12.  i = 1
13.  t5 = i - 1
14.  t6 = 88 * t5
15.  a[t6] = 1.0
16.  i = i + 1
17.  if i <= 10 goto (13)
```

Terms used in Code Optimization

1. i=1
2. j=1
3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 0.0
8. j = j + 1
9. if j <= goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. t5 = i - 1
14. t6 = 88 * t5
15. a[t6] = 1.0
16. i = i + 1
17. if i <= 10 goto (13)

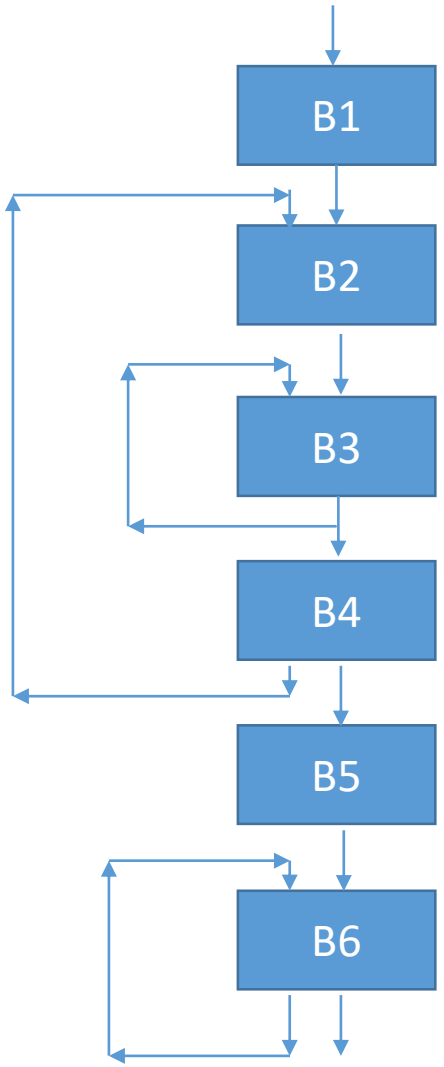
Leaders: 1,2, 3, 10,12,13

Block1 has Statement 1, Block 2 has statement 2, Block 3 has statement 3-9, Block 4 has statement 10-11, Block 5 has statement 12, Block 6 has statement 13-17

Terms used in Code Optimization

1. i=1
2. j=1
3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 0.0
8. j = j + 1
9. if j <= goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. t5 = i - 1
14. t6 = 88 * t5
15. a[t6] = 1.0
16. i = i + 1
17. if i <= 10 goto (13)

Leaders: 1,2, 3, 10,12,13



Block1 has Statement 1, Block 2 has statement 2, Block 3 has statement 3-9, Block 4 has statement 10-11, Block 5 has statement 12, Block 6 has statement 13-17

Terms used in Code Optimization

```
 (1)     i = m-1
 (2)     j = n
 (3)     t1 = 4*n
 (4)     v = a[t1]
 (5)     i = i+1
 (6)     t2 = 4*i
 (7)     t3 = a[t2]
 (8)     if t3<v goto (5)
 (9)     j = j-1
(10)     t4 = 4*j
(11)     t5 = a[t4]
(12)     if t5>v goto (9)
(13)     if i>=j goto (23)
(14)     t6 = 4*i
(15)     x = a[t6]
(16)     t7 = 4*i
(17)     t8 = 4*j
(18)     t9 = a[t8]
(19)     a[t7] = t9
(20)     t10 = 4*j
(21)     a[t10] = x
(22)     goto (5)
(23)     t11 = 4*i
(24)     x  = a[t11]
(25)     t12 = 4*i
(26)     t13 = 4*n
(27)     t14 = a[t13]
(28)     a[t12] = t14
(29)     t15 = 4*n
(30)     a[t15] = x
```

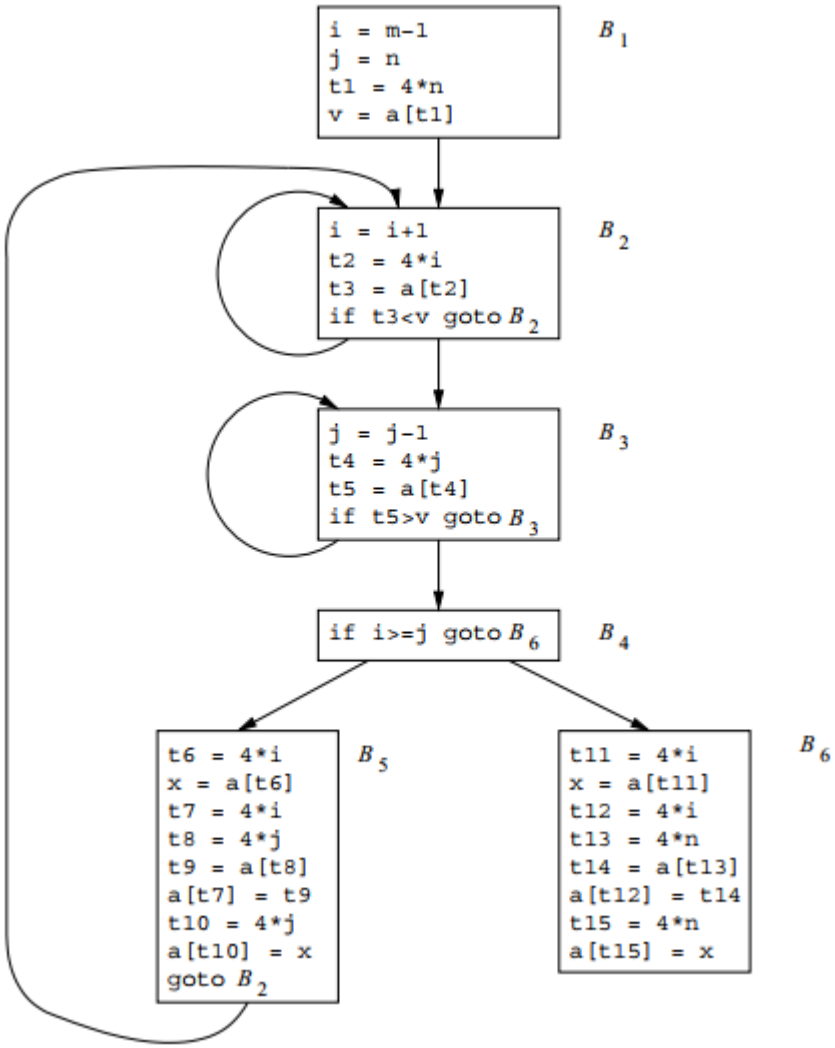
Terms used in Code Optimization

```
(1)     i = m-1
(2)     j = n
(3)     t1 = 4*n
(4)     v = a[t1]
(5)     i = i+1
(6)     t2 = 4*i
(7)     t3 = a[t2]
(8)     if t3<v goto (5)
(9)     j = j-1
(10)    t4 = 4*j
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
(13)    if i>=j goto (23)
(14)    t6 = 4*i
(15)    x = a[t6]
(16)    t7 = 4*i
(17)    t8 = 4*j
(18)    t9 = a[t8]
(19)    a[t7] = t9
(20)    t10 = 4*j
(21)    a[t10] = x
(22)    goto (5)
(23)    t11 = 4*i
(24)    x  = a[t11]
(25)    t12 = 4*i
(26)    t13 = 4*n
(27)    t14 = a[t13]
(28)    a[t12] = t14
(29)    t15 = 4*n
(30)    a[t15] = x
```

- Leaders are statement 1, 5, 9, 13, 14, 23
- Block 1 will have 1-4 statements
- Block 2 will have 5-8 statements
- Block 3 will have 9-12 statements
- Block 4 will have statements no 13
- Block 5 will have 14-22 statements
- Block 6 will have 23-30 statements

Terms used in Code Optimization

```
(1)     i = m-1
(2)     j = n
(3)     t1 = 4*n
(4)     v = a[t1]
(5)     i = i+1
(6)     t2 = 4*i
(7)     t3 = a[t2]
(8)     if t3<v goto (5)
(9)     j = j-1
(10)    t4 = 4*j
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
(13)    if i>=j goto (23)
(14)    t6 = 4*i
(15)    x = a[t6]
(16)    t7 = 4*i
(17)    t8 = 4*j
(18)    t9 = a[t8]
(19)    a[t7] = t9
(20)    t10 = 4*j
(21)    a[t10] = x
(22)    goto (5)
(23)    t11 = 4*i
(24)    x  = a[t11]
(25)    t12 = 4*i
(26)    t13 = 4*n
(27)    t14 = a[t13]
(28)    a[t12] = t14
(29)    t15 = 4*n
(30)    a[t15] = x
```

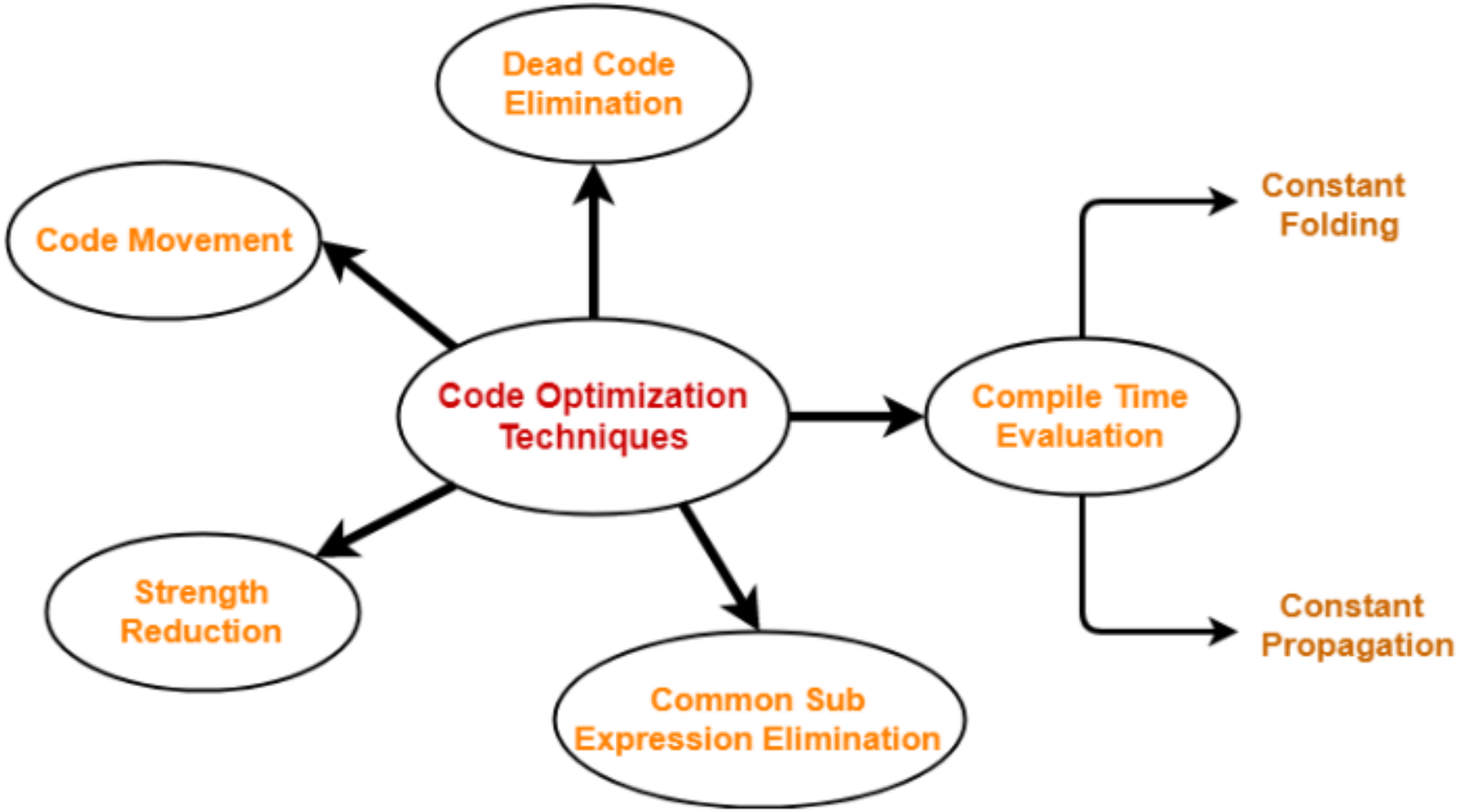


Leaders are statement 1, 5, 9, 13, 14, 23

Block 1 will have 1-4 statements, Block 2 will have 5-8 statements, Block 3 will have 9-12 statements

Block 4 will have statements no 13, Block 5 will have 14-22 statements, Block 6 will have 23-30 statements

Machine Independent Code Optimization



Machine Independent Code Optimization

- Constant Folding/Code Motion
  - This optimization is a type of Loop Optimization.
  - The expressions that contain the operands having constant values at compile time are evaluated. Those expressions are then replaced with their respective results.
  - E.g. Circumference of Circle = (22/7) x Diameter
  - This technique evaluates the expression 22/7 at compile time. The expression is then replaced with its result 3.14. This saves the time at run time.

```
While(i<=max-1)
    sum=sum+a[i]


n=max-1
While(i<=n)
    sum=sum+a[i]
```

Machine Independent Code Optimization

- Code Movement/removal of induction variable
  - This optimization is a type of Loop Optimization.
  - An induction variable is any variable whose value can be represented as a function of loop invariants i.e. which does not change with execution of loop.
  - It involves movement of the code. The code present inside the loop is moved out if it does not matter whether it is present inside or outside. Such a code unnecessarily gets execute again and again with each iteration of the loop. This leads to the wastage of time at run time.

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

Machine Independent Code Optimization

- Strength Reduction
  - This optimization is a type of Loop Optimization.
  - It involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.
  - E.g. B=A*2 will be replaced by B=A+A

Machine Independent Code Optimization

- Constant Propagation
  - If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation. The condition is that the value of variable must not get alter in between.
  - pi=3.142, radius=10;
  - Area of circle = pi x radius x radius
  - This technique substitutes the value of variables ‘pi’ and ‘radius’ at compile time. It then evaluates the expression 3.14 x 10 x 10. The expression is then replaced with its result 314. This saves the time at run time.

Machine Independent Code Optimization

- Dead Code Elimination
  - It involves eliminating the dead code.
  - The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Code Before Optimization	Code After Optimization
i = 0 ; **if (i == 1)** **{** **a = x + 5 ;** **}**	i = 0 ;

Machine Independent Code Optimization

- Common Subexpression Elimination
  - The expression that has been already computed before and appears again in the code for computation is called as Common Sub-Expression.
  - It involves eliminating the common sub expressions. The redundant expressions are eliminated to avoid their re-computation. The already computed result is used in the further program when required.

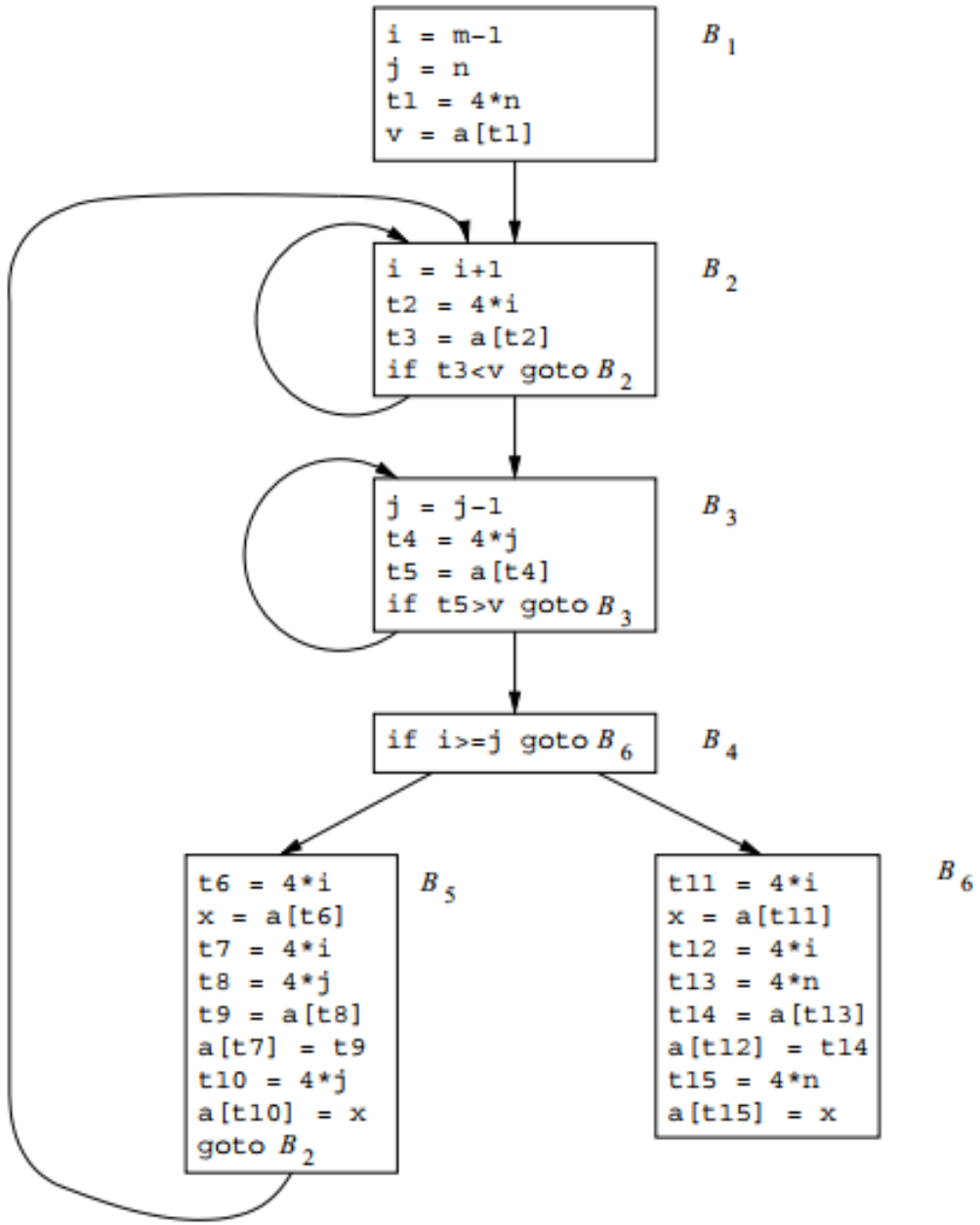
Code Before Optimization	Code After Optimization
S1 = 4 * i S2 = a[S1] S3 = 4 * j S4 = 4 * i **// Redundant Expression** S5 = n S6 = b[S4] + S5	S1 = 4 * i S2 = a[S1] S3 = 4 * j S5 = n S6 = b[S1] + S5

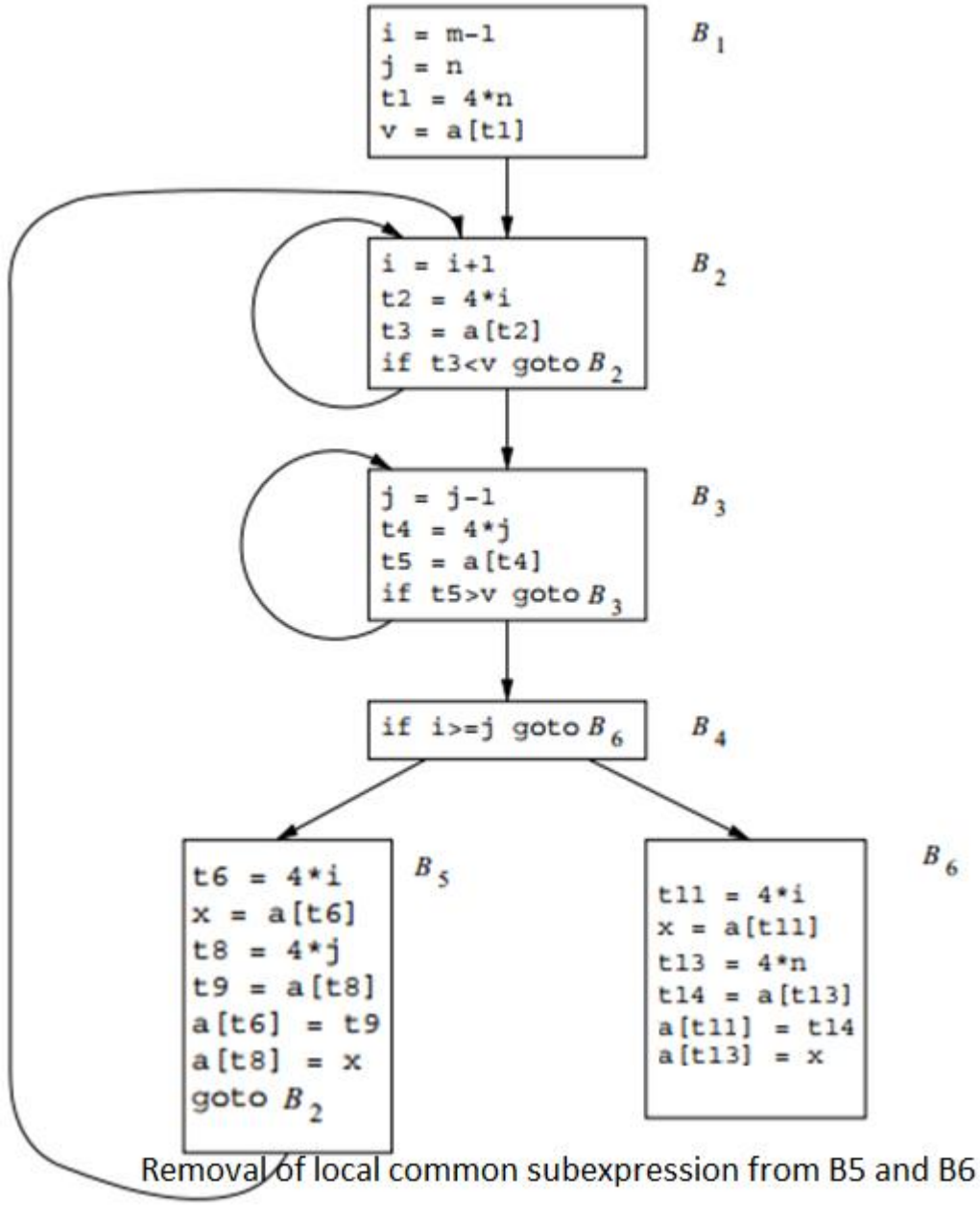
Machine Independent Code Optimization

- Local Common Subexpression Elimination
  - When the removal of subexpression effect only the block in which it was written then it is called local common subexpression elimination.

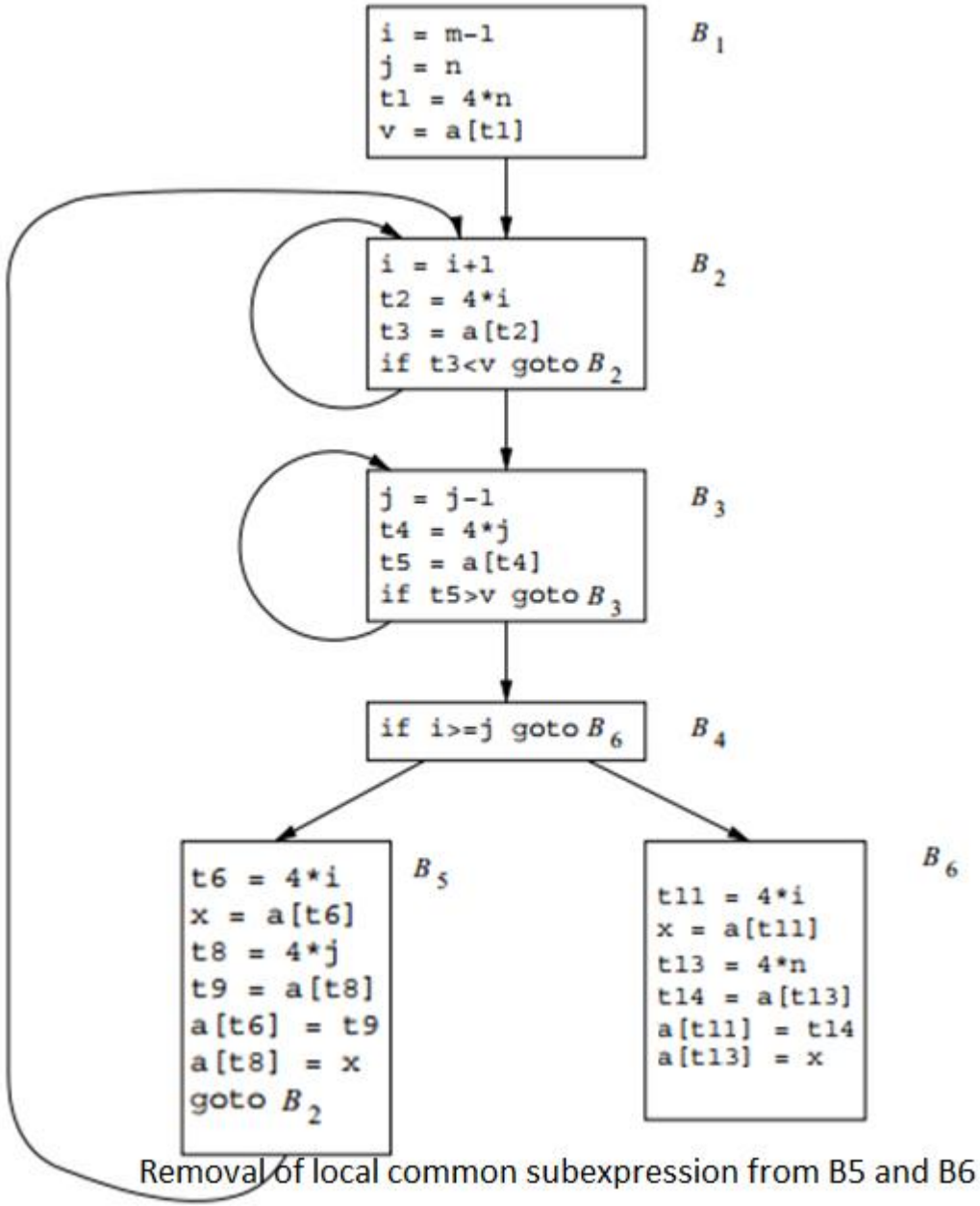
Machine Independent Code Optimization

- Global Common Subexpression Elimination
  - When the removal of subexpression effect multiple blocks then it is called global common subexpression elimination.

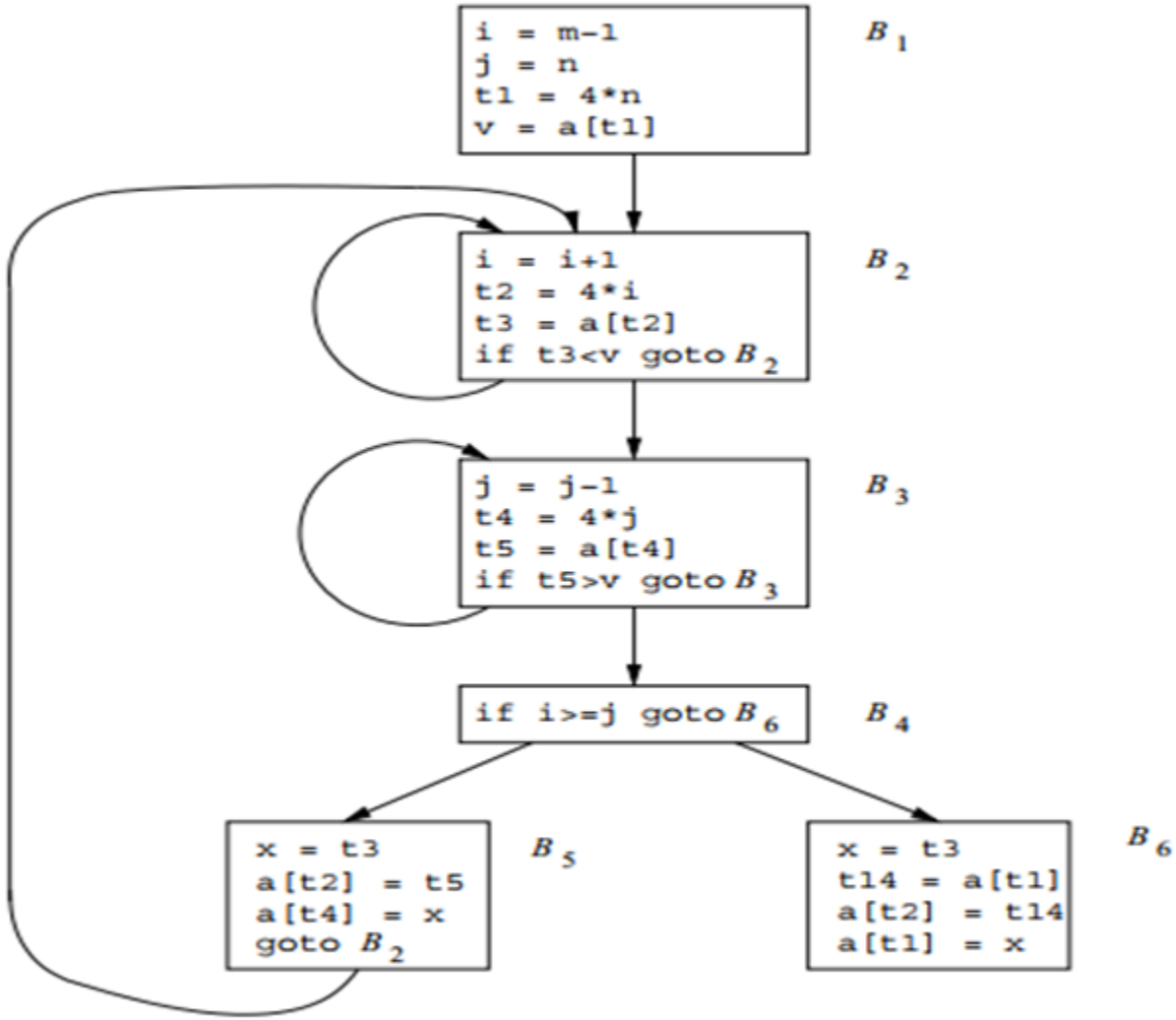




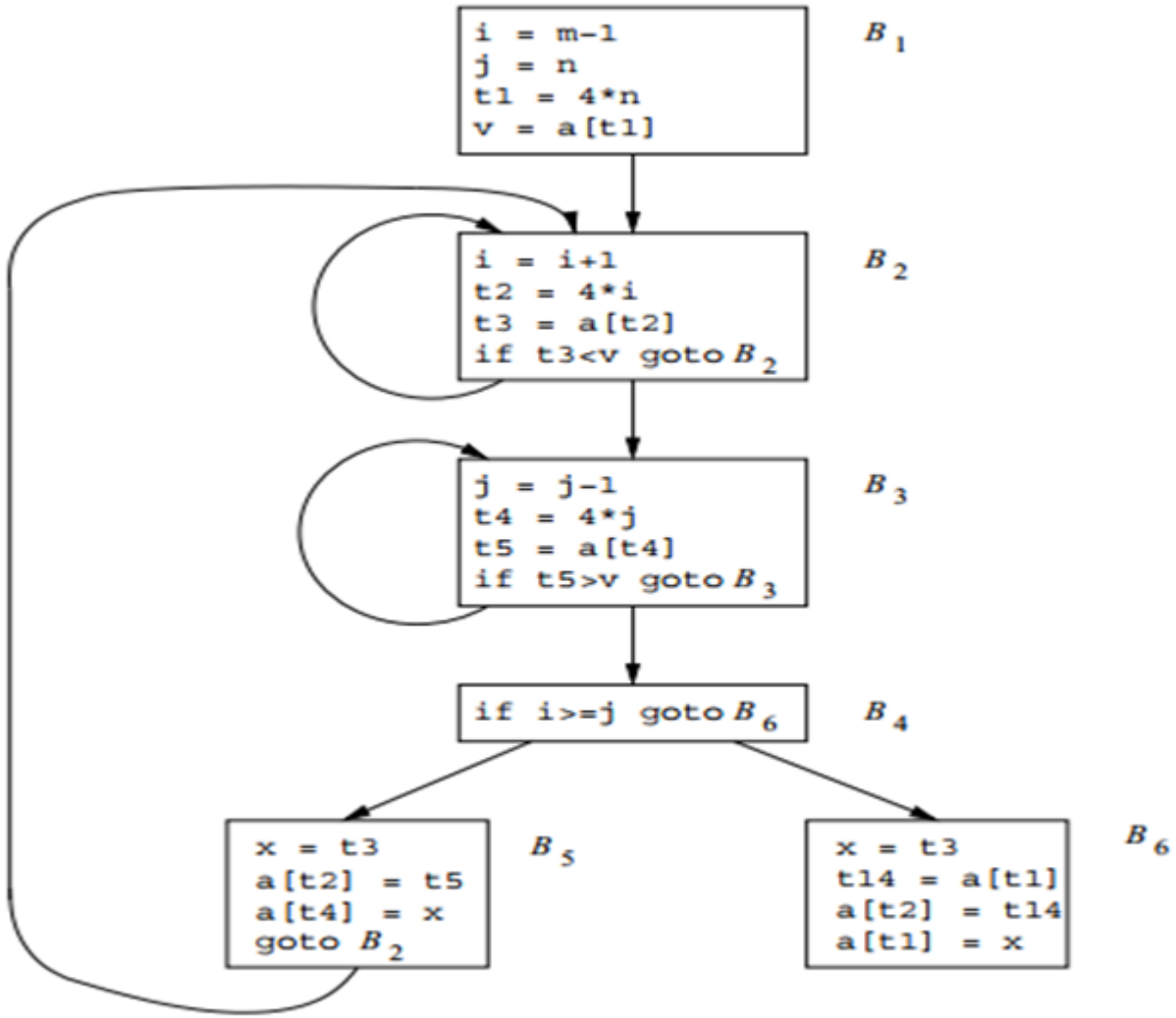
Removal of local common subexpression from B5 and B6



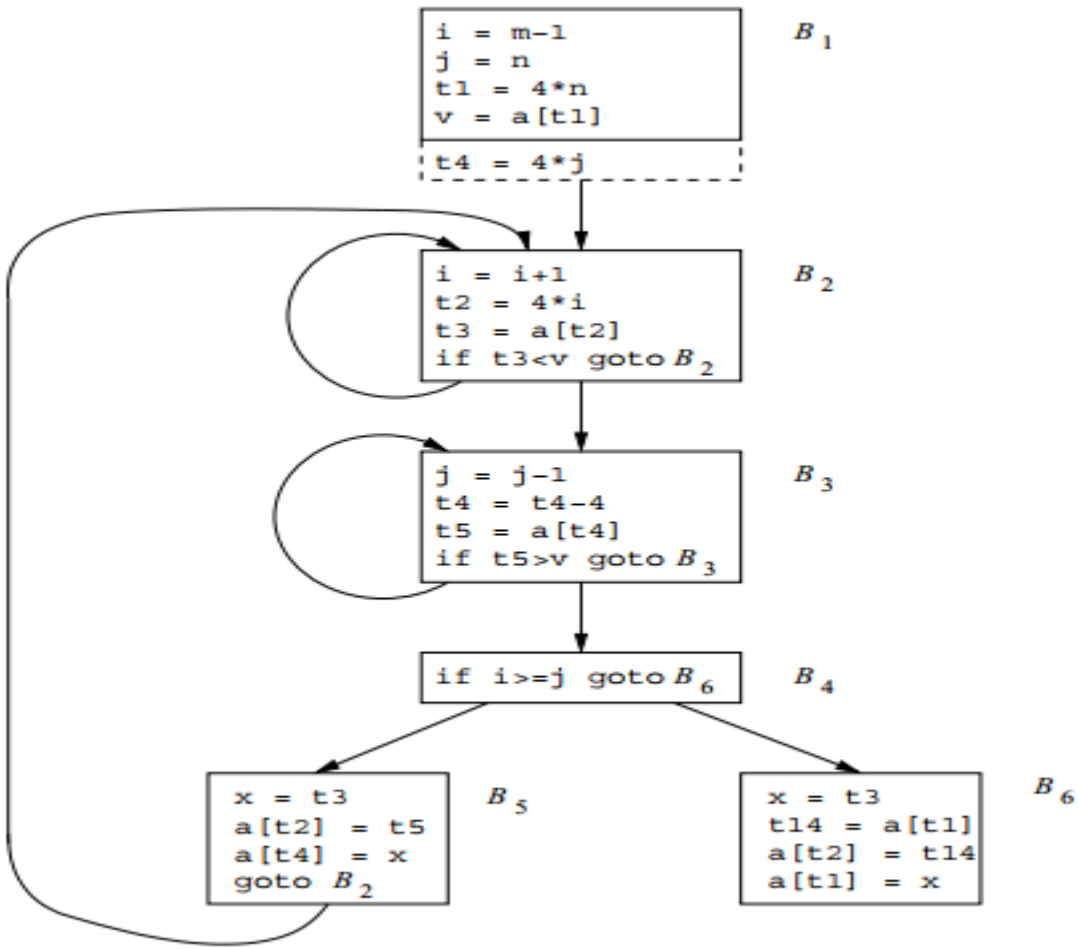
Removal of local common subexpression from B5 and B6



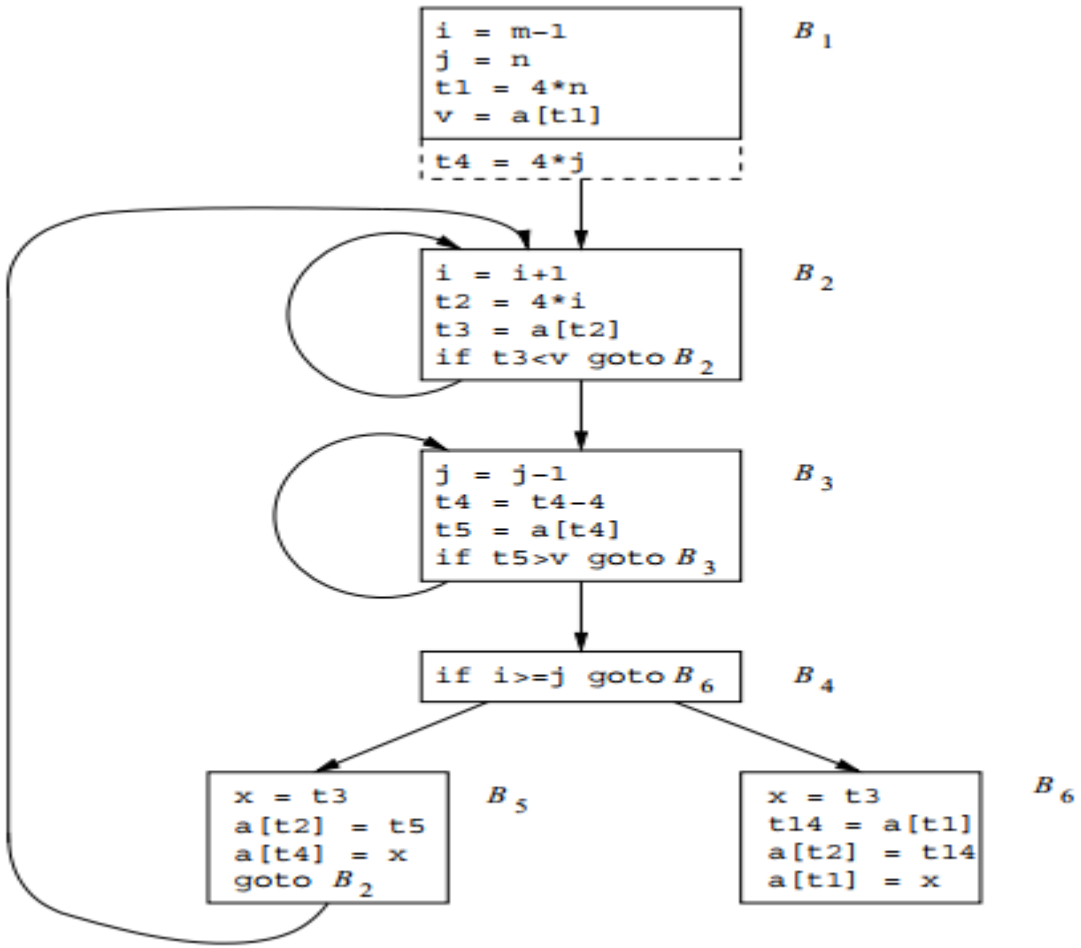
Removal of Global common subexpression from B5 and B6



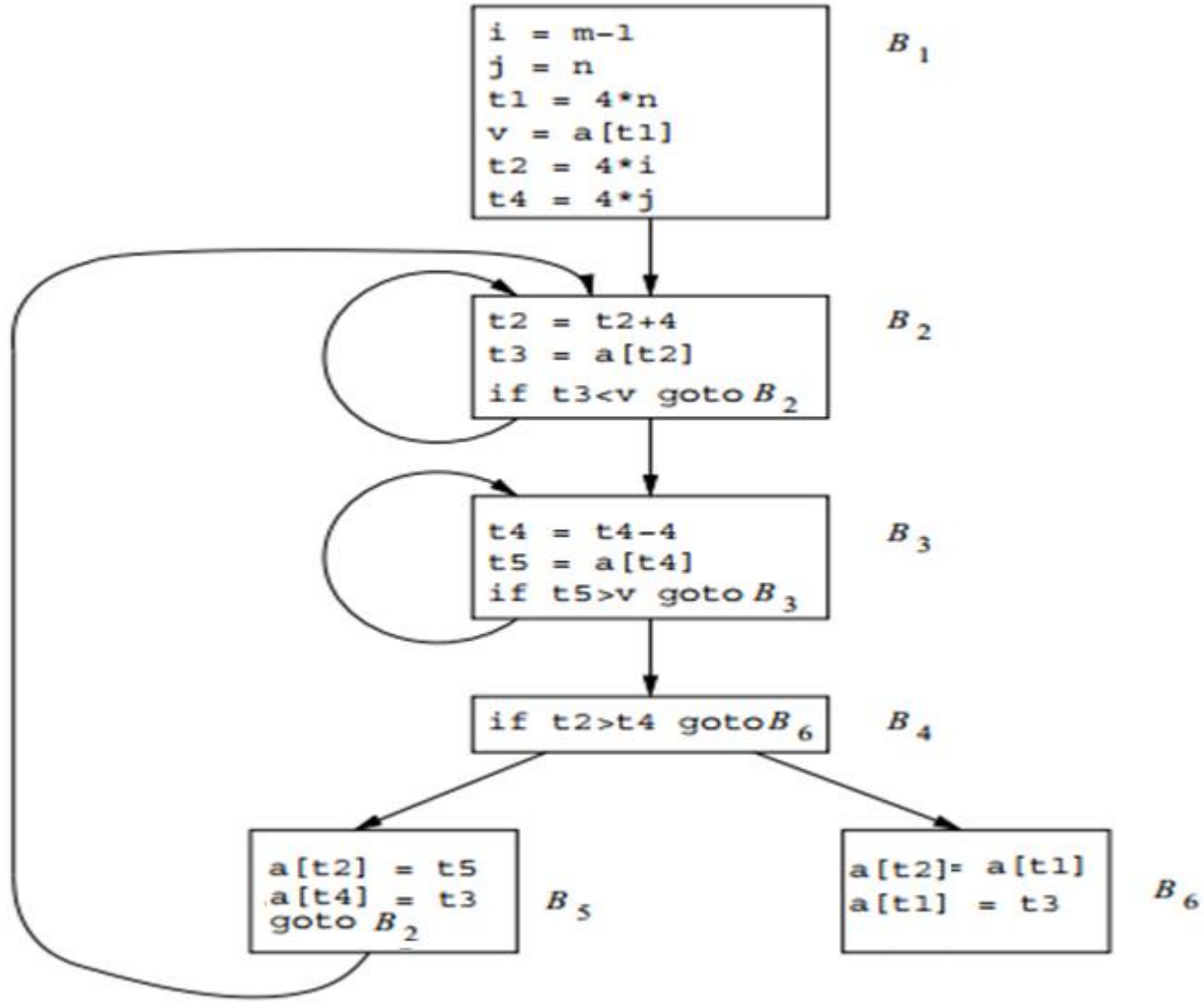
Removal of Global common subexpression from B5 and B6



Strength reduction applied to $4 * j$ in block B_3



Strength reduction applied to $4 * j$ in block B_3



Flow graph after induction-variable elimination

DAG representation of Basic Blocks

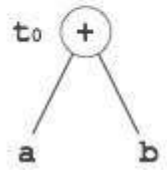
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.
- A DAG for a basic block is a directed acyclic graph with the following labels on nodes:
  - Leaves are labeled by unique identifiers, either variable names or constants.
  - Interior nodes are labeled by an operator symbol.
  - Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

Steps for construction of DAG

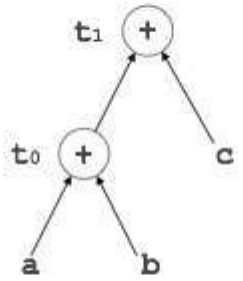
- Construct a label for each node. For leaves, the label is an identifier and for interior nodes it is an operator symbol.
- Each node contains a list of attached identifiers to hold the computed values. Case (i) x : = y OP z Case (ii) x : = OP y Case (iii) x : = y
  - Step 1:
    - If y is undefined then create node(y).
    - If z is undefined, create node(z) for case(i).
  - Step 2:
    - For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.
    - For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.
    - For case(iii), node n will be node(y).
  - Step 3:
    - Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Example of DAG

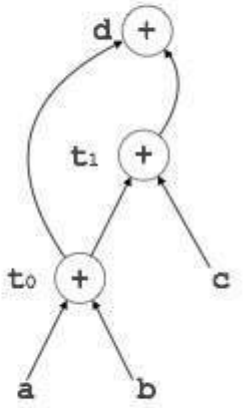
```
t0 = a + b
t1 = t0 + c
d = t0 + t1
```



[$t_0 = a + b$]



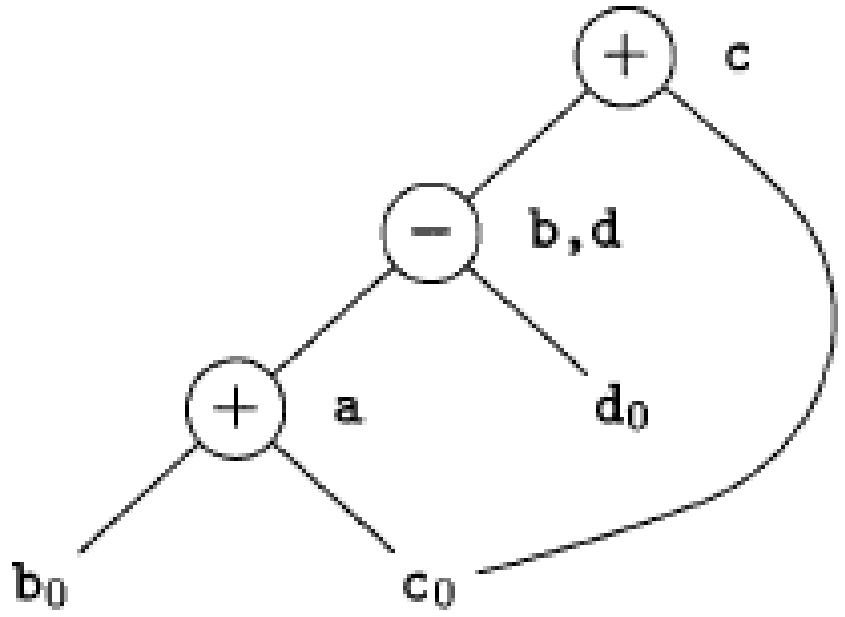
[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

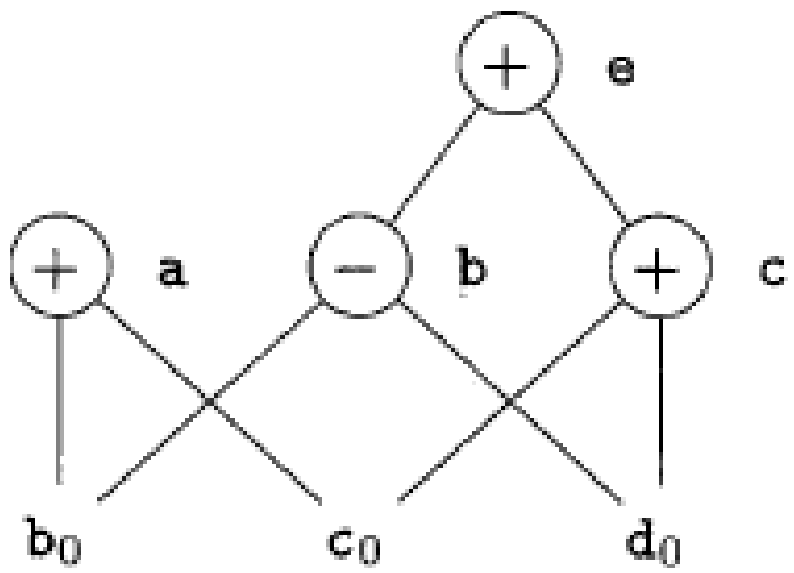
Example of DAG

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



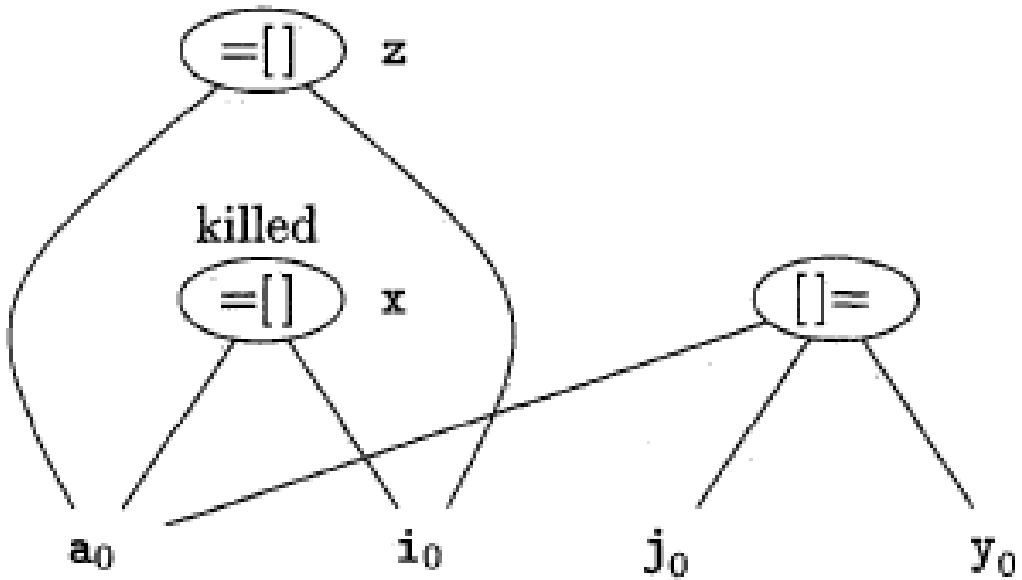
Example of DAG

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



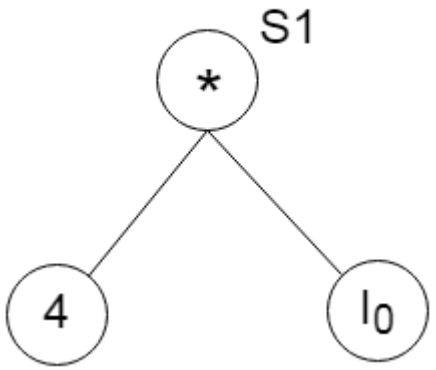
Example of DAG

```
x = a[i]
a[j] = y
z = a[i]
```

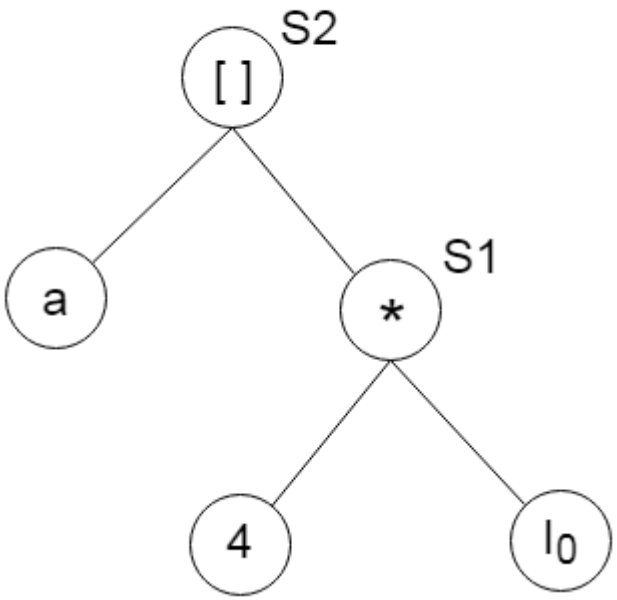


1. S1:= 4 * i
2. S2:= a[S1]
3. S3:= 4 * i
4. S4:= b[S3]
5. S5:= s2 * S4
6. S6:= prod + S5
7. Prod:= s6
8. S7:= i+1
9. i := S7
10. if i<= 20 goto (1)

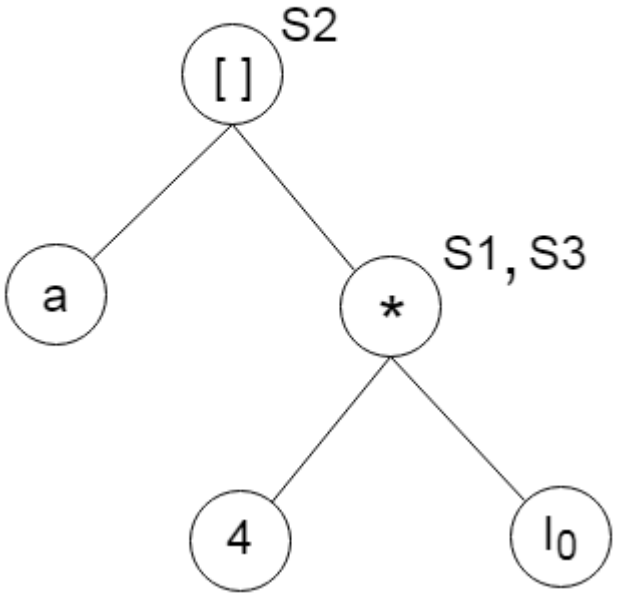
Statement (1)



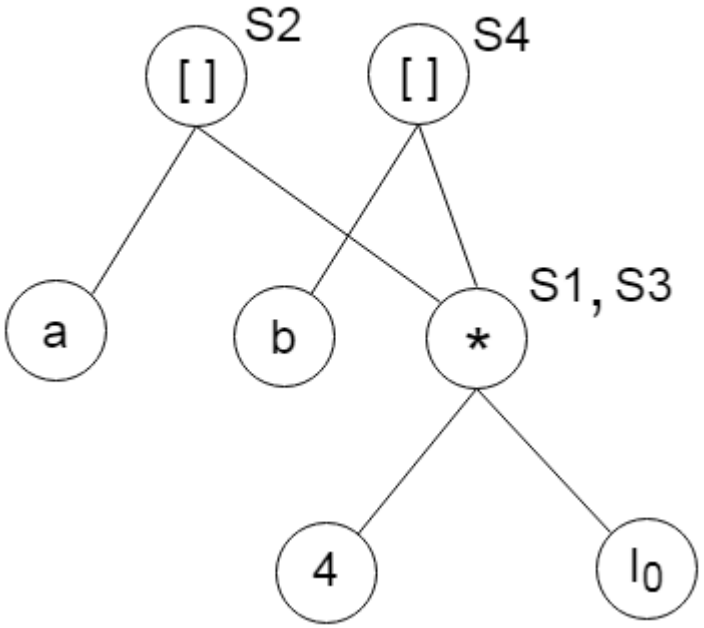
Statement (2)



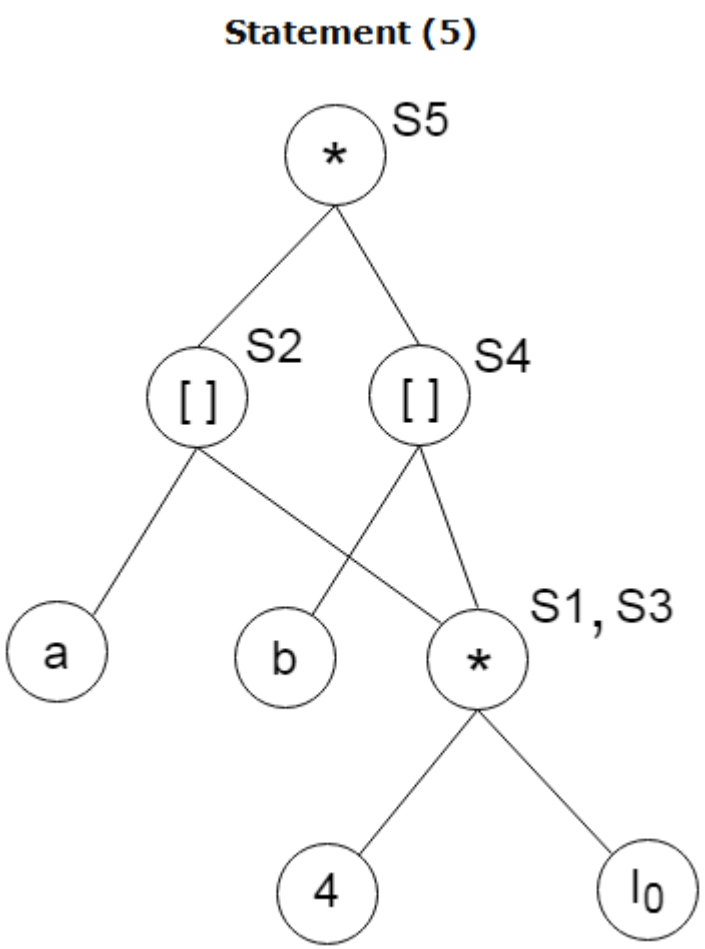
4 * I0 node exist already hence attach identifier S3 to the existing node for statement (3)

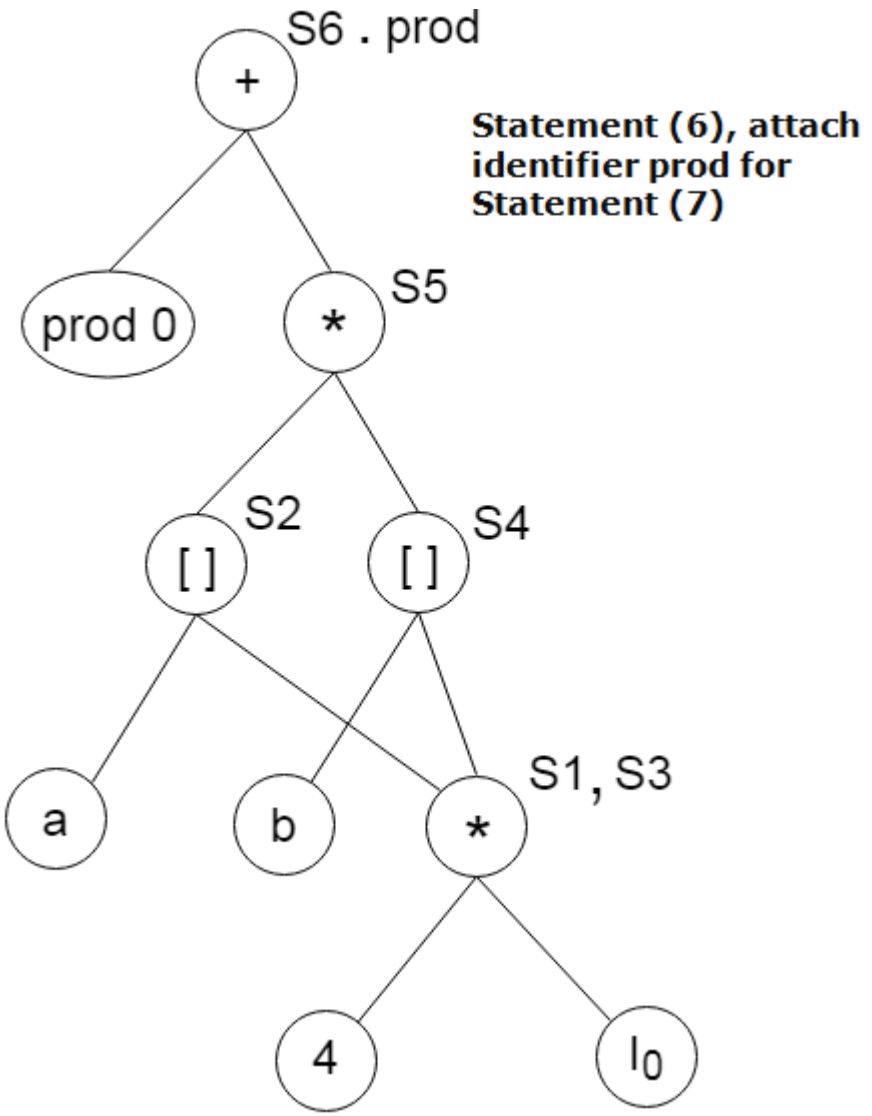


Statement (4)

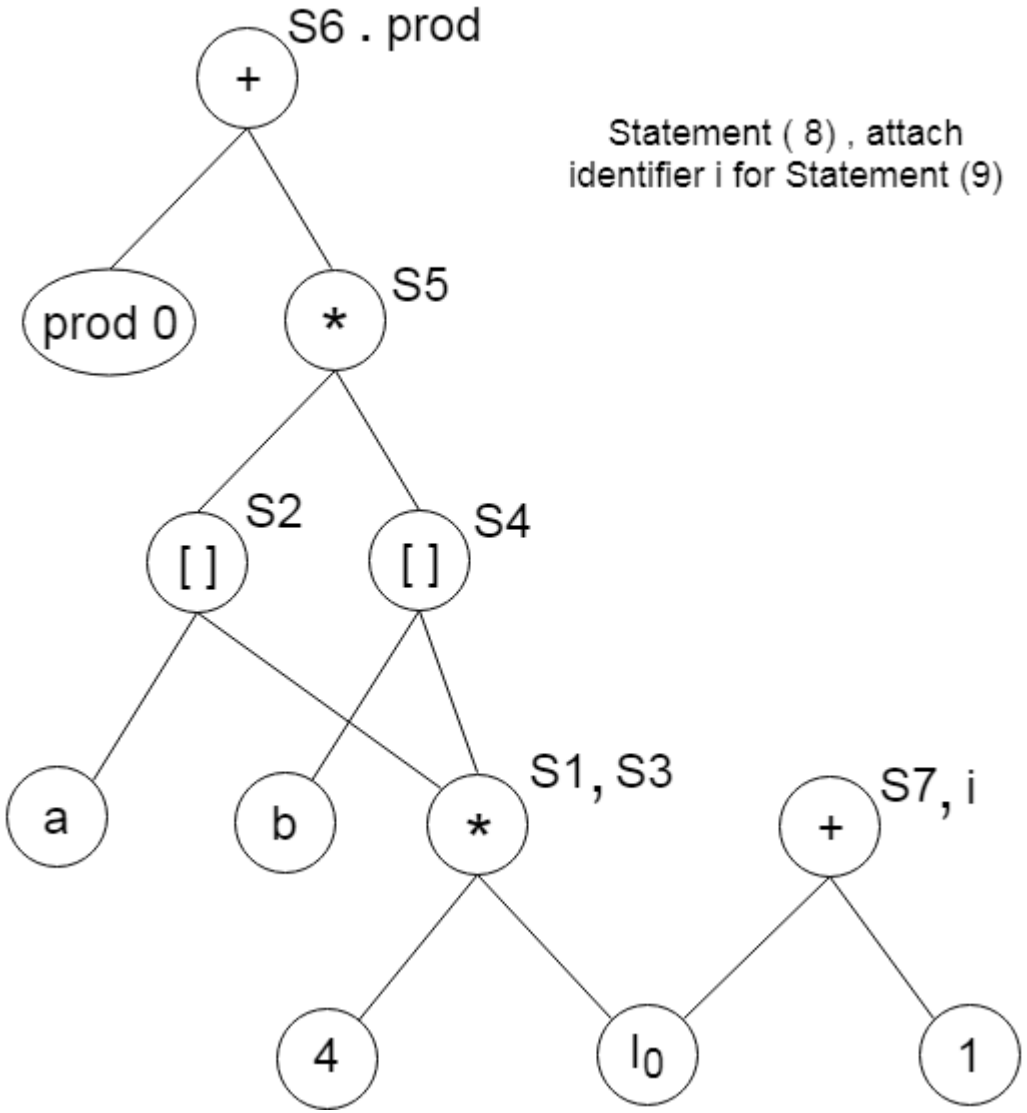


1. S1:= 4 * i
2. S2:= a[S1]
3. S3:= 4 * i
4. S4:= b[S3]
5. S5:= s2 * S4
6. S6:= prod + S5
7. Prod:= s6
8. S7:= i+1
9. i := S7
10. if i<= 20 goto (1)

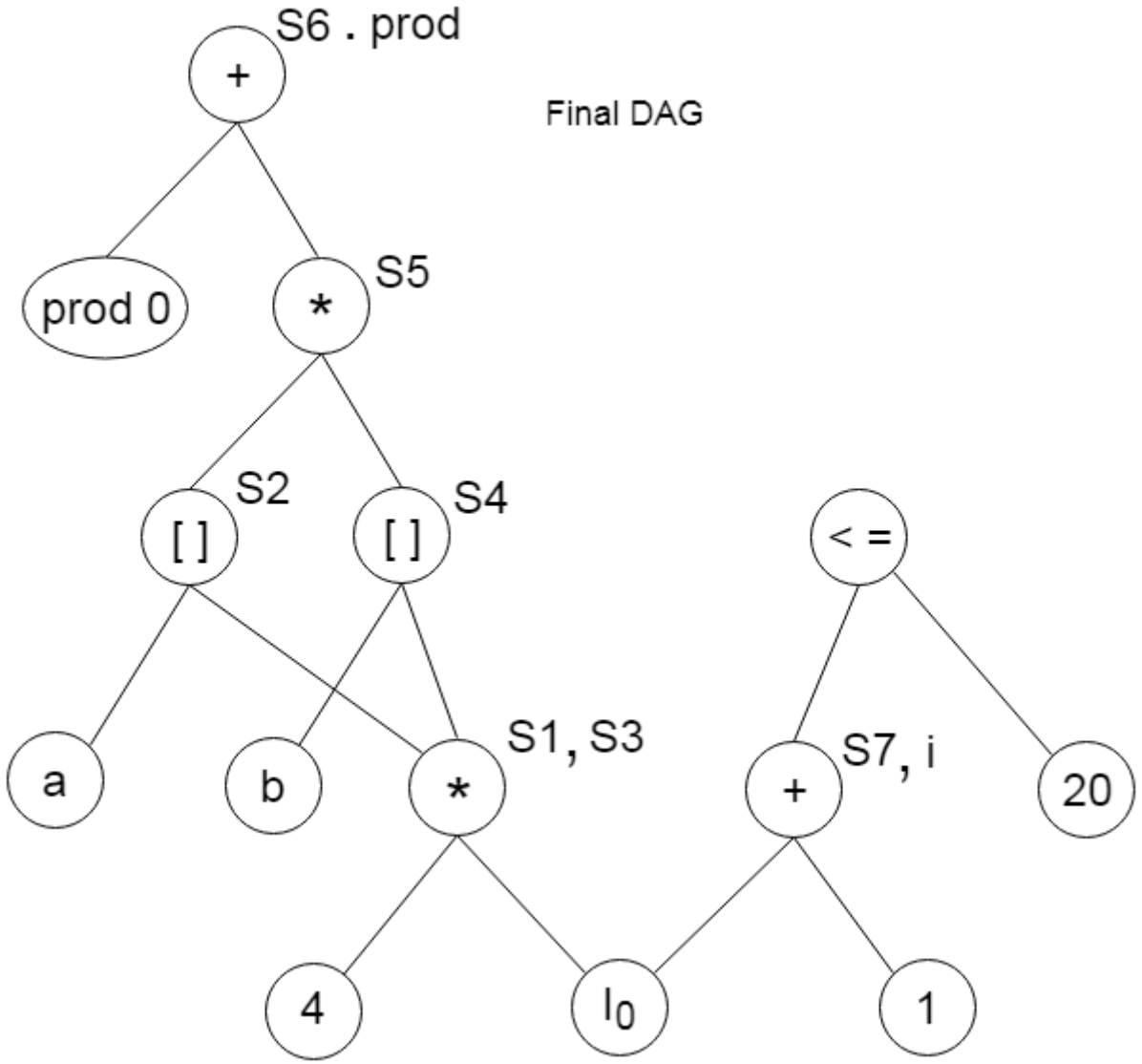




1. S1:= 4 * i
2. S2:= a[S1]
3. S3:= 4 * i
4. S4:= b[S3]
5. S5:= s2 * S4
6. S6:= prod + S5
7. Prod:= s6
8. S7:= i+1
9. i := S7
10. if i<= 20 goto (1)



1. S1:= 4 * i
2. S2:= a[S1]
3. S3:= 4 * i
4. S4:= b[S3]
5. S5:= s2 * S4
6. S6:= prod + S5
7. Prod:= s6
8. S7:= i+1
9. i := S7
10. if i<= 20 goto (1)



Applications of DAG

- We can automatically detect common sub expressions.
- We can determine which identifiers have their values used in the block.
- We can determine which statements compute values that could be used outside the block.

Code Generation

- **Code generator** converts the intermediate representation of source code into a form that can be readily executed by the machine. Some essential property which code generation phase must possess are as follows:
  - Correctness: Must give the expected outcome.
  - High Quality: No error or ambiguity in the code.
  - Efficient use of resources of target machine: Resources like registers, cpu time, main memory must be used efficiently
  - Quick code generation: Does not take much time.

Code Generation

- Output of code generator phase is machine code which can be of two types:
  - Absolute machine code:
    - Have fixed locations in memory and immediately executed.
    - Hardware dependent.
    - Useful for small program
  - Relative machine code:
    - Does not have fixed location
    - Code can be placed wherever linker find the space in RAM.
    - Generally used by all compilers.

Issues in Code Generation

- **Input to Code Generation:** The input to code generator is the intermediate code generated by semantic analysis phase or code optimization phase. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.
- **Target Program:** The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.
- **Memory Management:** Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

Issues in Code Generation

- **Instruction Selection:** Selecting the best instructions will improve the efficiency of the program.

```
P:=Q+R
S:=P+T
```

```
MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

```
MOV Q, R0
ADD R, R0
ADD T, R0
MOV R0, S
```

- **Register Allocation:** Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important.

```
t:=a+b
t:=t*c
t=t/d
```

```
Mov a, R0
Add b, R0
Mul c, R0
Div d, R0
Mov R0, t
```

Example of Code Generation

The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

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

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
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 R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Peephole Optimization

- This technique of optimization can be applied on source code, target code or on assembly code. This optimization technique works locally. A bunch of statements is analyzed and are checked for the following possible optimization:
- Redundant instruction elimination
  - At source code level, the following can be done by the user:

```
int add_ten(int x)
{ int y, z;
y = 10;
z = x + y;
 return z; }
```

```
int add_ten(int x)
{ int y;
y = 10;
y = x + y;
return y; }
```

```
int add_ten(int x)
{
int y = 10;
return x + y;
}
```

```
int add_ten(int x)
{
return x + 10;
}
```

  - 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
```

We can delete the first instruction and re-write the sentence as: MOV x, R1

Peephole Optimization

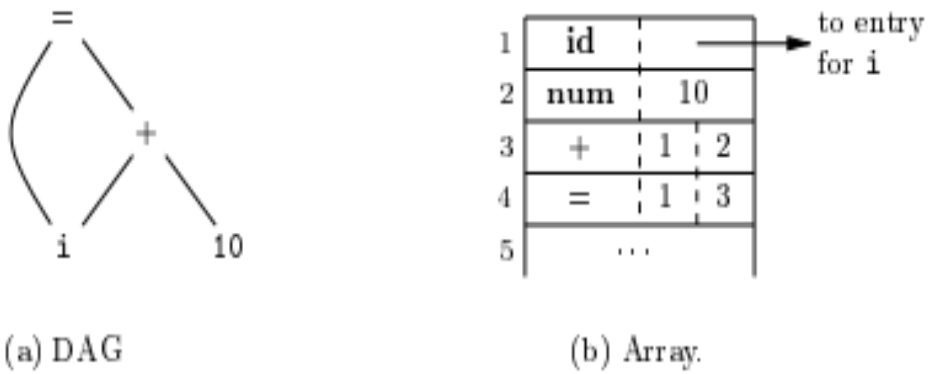
- This technique of optimization can be applied on source code, target code or on assembly code. This optimization technique works locally. A bunch of statements is analyzed and are checked for the following possible optimization:
- Flow of Control Optimization
  - There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:
    - MOV R1, R2
    - GOTO L1
    - ...
    - L1 : GOTO L2
    - L2 : INC R1
  - In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:
    - MOV R1, R2
    - GOTO L2
    - ...
    - L2 : INC R1

Peephole Optimization

- This technique of optimization can be applied on source code, target code or on assembly code. This optimization technique works locally. A bunch of statements is analyzed and are checked for the following possible optimization:
- Algebraic Expression Simplification
  - There are occasions where algebraic expressions can be made simple. For example, the expression **a = a + 0 or a=a*1** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.
- Strength Reduction
  - There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result. E.g. multiplication can be reduced by addition

Value Numbers

- Value numbering is a technique of determining when two computations in a program are equivalent and eliminating one of them with a semantics preserving optimization.
- Nodes of DAG are stored in an array of records. Each row of the array represents one record corresponding to every node of DAG.
- In each record, the first field is an operation code, indicating the label of the node, leaves have one additional field, which holds the lexical value, interior nodes have two additional fields indicating the left and right children.

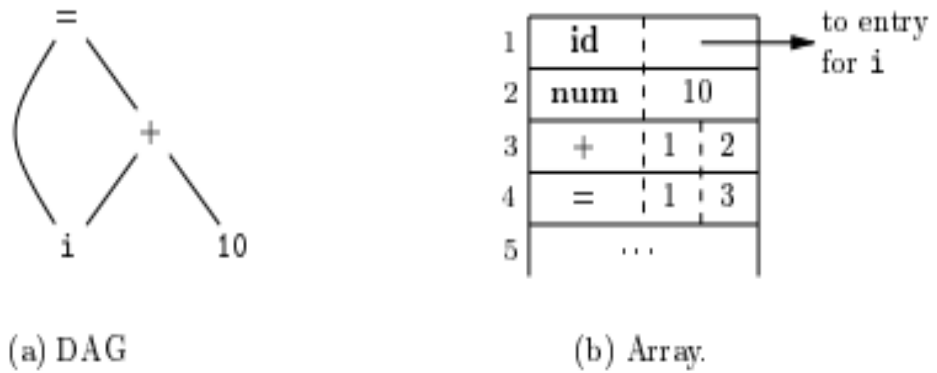


(a) DAG (b) Array.

Nodes of a DAG for $i = i + 10$ allocated in an array

Value Numbers

- We refer to the node by giving the integer index of the record for that node within the array. This integer is called the value number for the node or for the expression represented by that node.
- Value numbers helps us in construction of efficient DAG corresponding to an expression.
- Let the signature of an interior node be the triple (op; l; r), where op is the label, l its left child's value number, and r its right child's value number. A unary operator may be assumed to have r = 0.



(a) DAG (b) Array.

Nodes of a DAG for $i = i + 10$ allocated in an array

- Search the array for a node M with label op, left child l, and right child r. If there is such a node, return the value number of M . If not, create in the array a new node N with label op, left child l, and right child r, and return its value number.

Algebraic Laws

- We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.
- Certain computations which are actually same but looks different to compiler, so they are not identified as common subexpression.
- By applying the different algebraic laws we can convert different looking subexpressions into similar looking subexpressions without changing their semantics.
- In this manner we can remove some common subexpressions by applying the algebraic laws.

Global Data Flow Analysis

- Certain optimization can only be achieved by examining the entire program. It can't be achieved by examining just a portion of the program.
- To ensure efficient code optimization and generation, compiler collect information about program and distribute this information to each block in the flow graph. This process is known as **data-flow graph analysis**.
- Data flow analysis refers to a body of techniques that derive information about the flow of data along program execution paths. It is essential for performing transformation across basic blocks.

Global Data Flow Analysis-Terminology

- Every assignment statement is a **definition**.
- Position between adjacent statement, above the first statement and after the last statement are called as **points**. If we consider all the blocks then each will have multiple points. Generally we merge the last point of the current block with the first point of the successor block.
- **Path** is a sequence of statements between any two points. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either
  - Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
  - Pi is the end of some block and pi+1 is the beginning of a successor block.

Global Data Flow Analysis-Terminology

- Gen(b): gen of block b is the set of statement number in which variables are defined in block.
- Kill(b): kill of block b is the set of statement numbers involving the redefinition of statements of b in other blocks.
- In(b): U Out(t) where t is a predecessor of block b. In the beginning In(b) is φ for all the blocks and Out(b)=Gen(b) for all the blocks.
- Data-flow information can be collected by using the following equation: out [S] = gen [S] U (in [S] – kill [S])
- This equation can be read as “ the information at the end of a statement S is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement.”

Global Data Flow Analysis-Terminology

- A definition d reaches a point P if there exists path from the point immediately following d to point P such that d is not killed(not redefined) along that path.
- A variable v is live at point P if the value of v is used along some path in the flow graph starting at P, otherwise variable is dead.

Global Data Flow Analysis-Terminology

An Iterative Algorithm for Computing Reaching Definitions

for each block B do { $IN[B] = \phi$; $OUT[B] = GEN[B]$; }
$change = true$;
while $change$ do { $change = false$;
  for each block B do {

$$IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P];$$

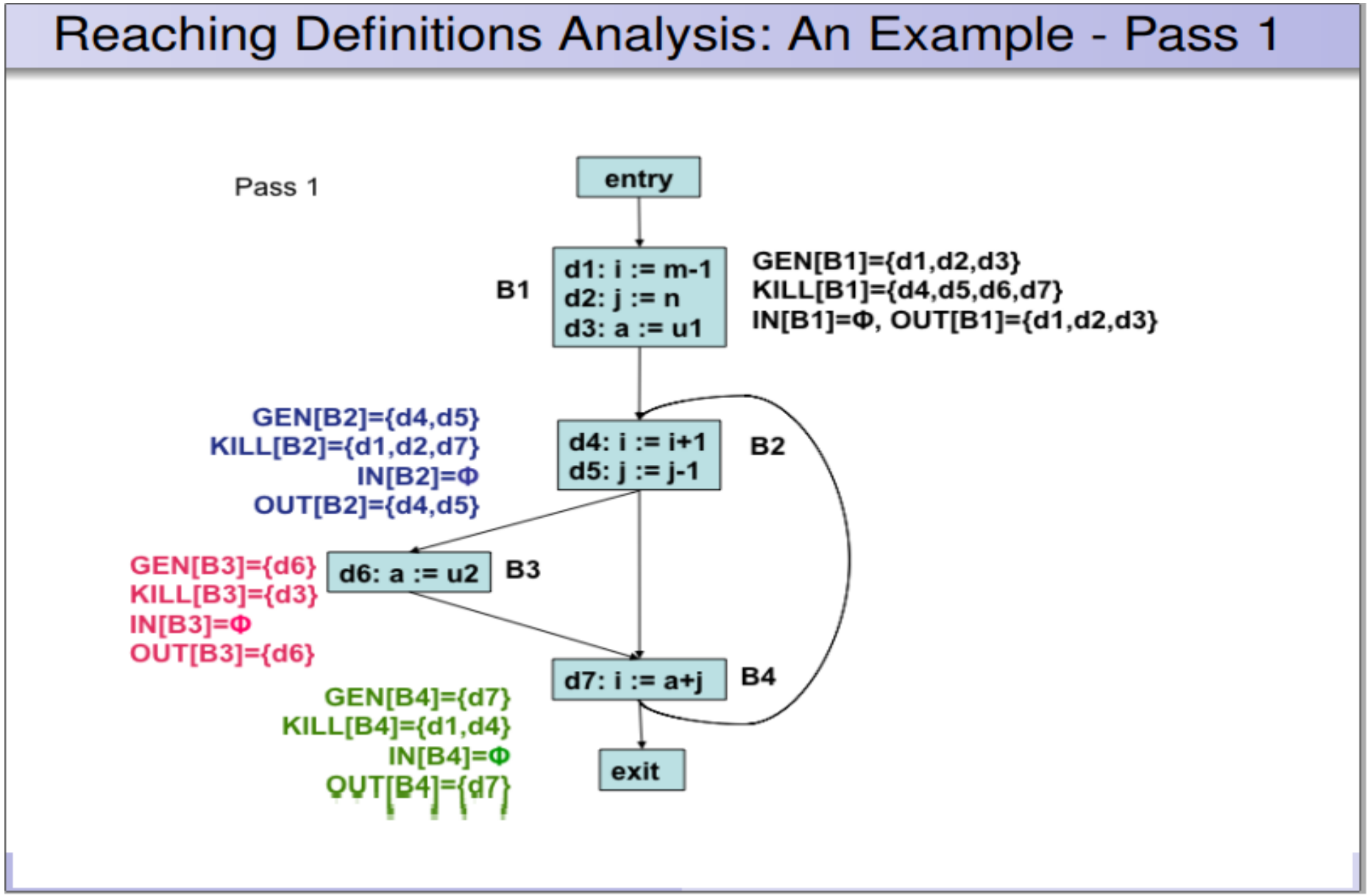
$$oldout = OUT[B];$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B]);$$

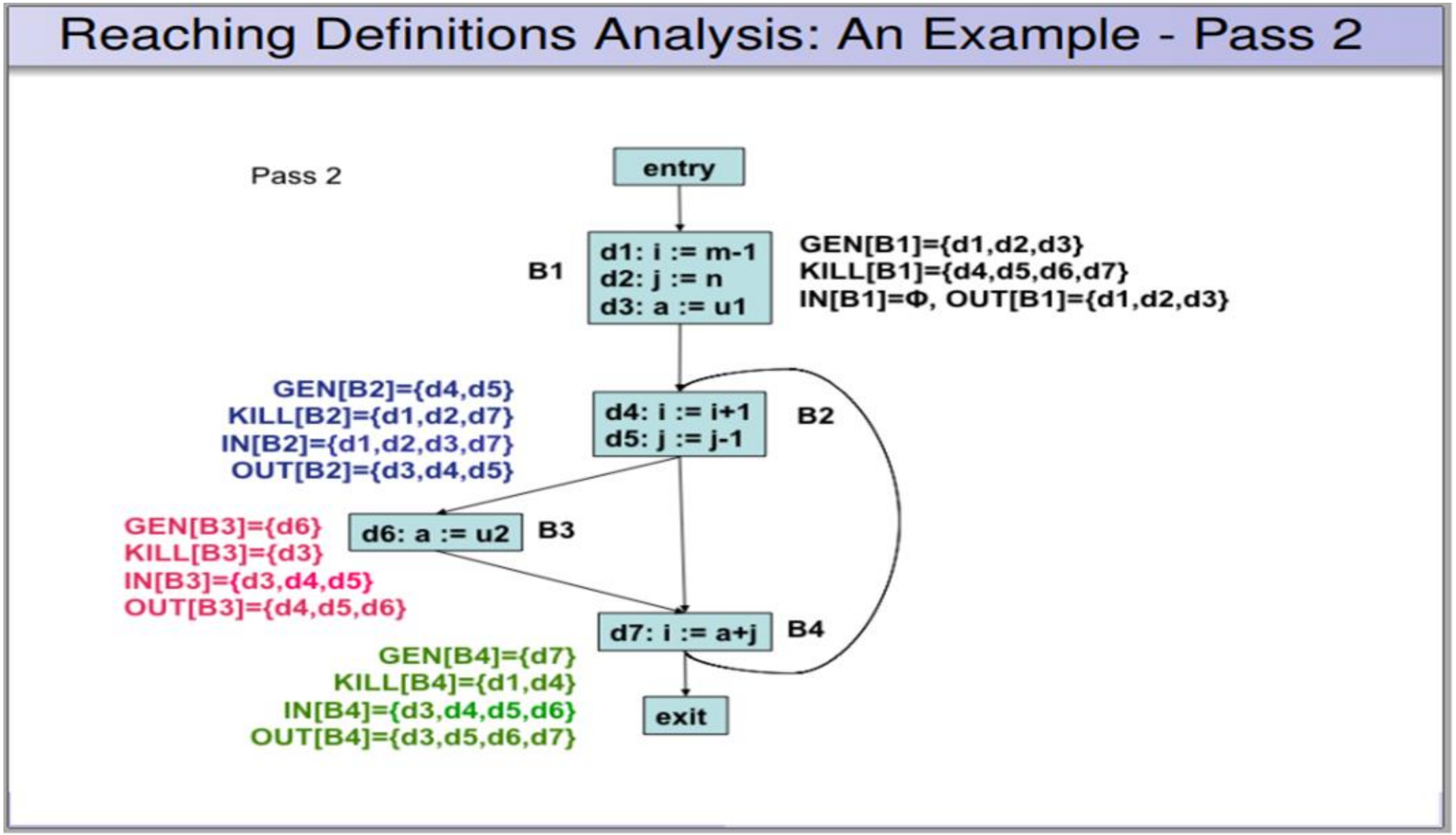
    if ($OUT[B] \neq oldout$) $change = true$;
  }
}

- GEN, $KILL$, IN, and OUT are all represented as bit vectors with one bit for each definition in the flow graph

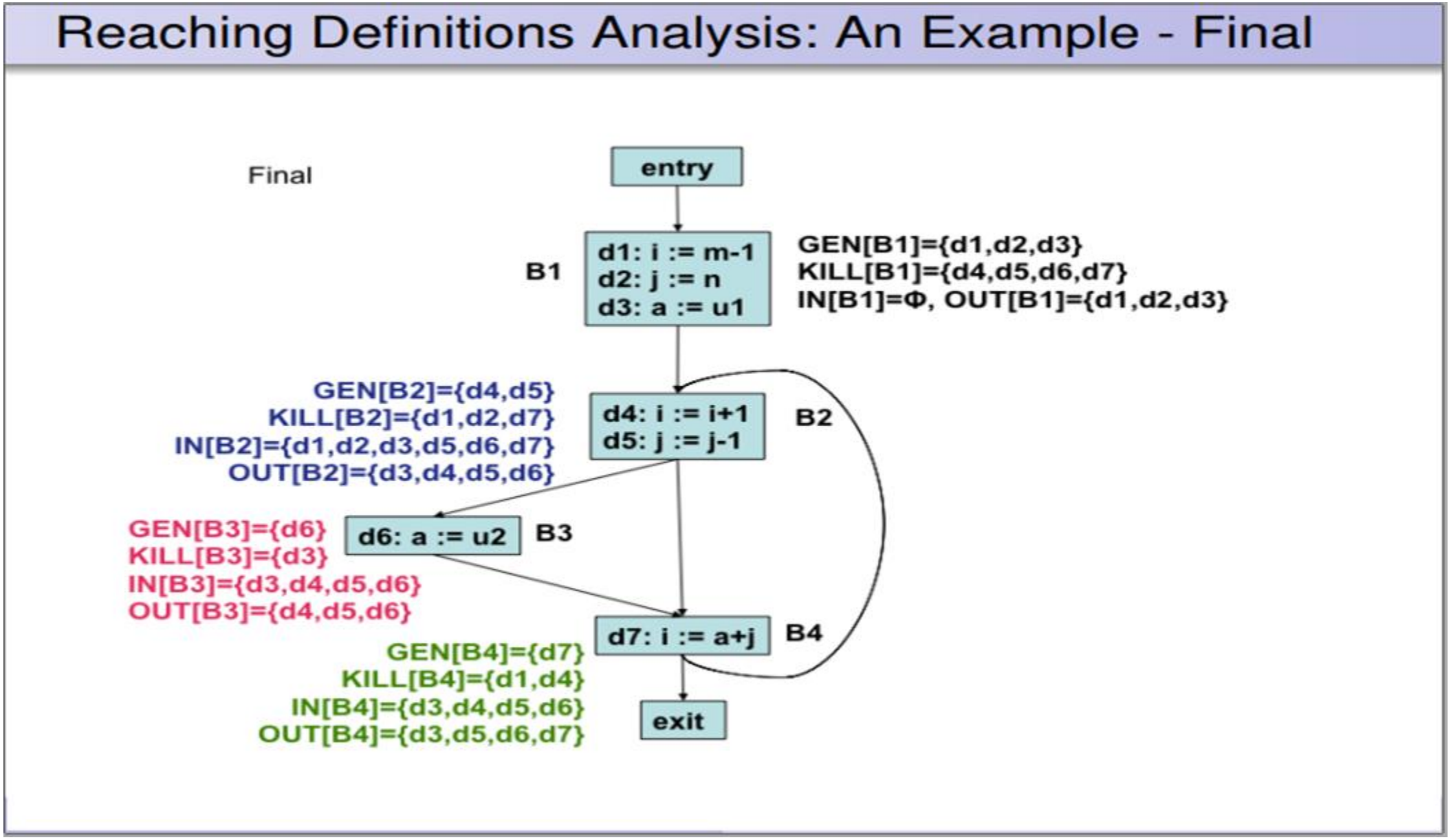
Global Data Flow Analysis-Terminology



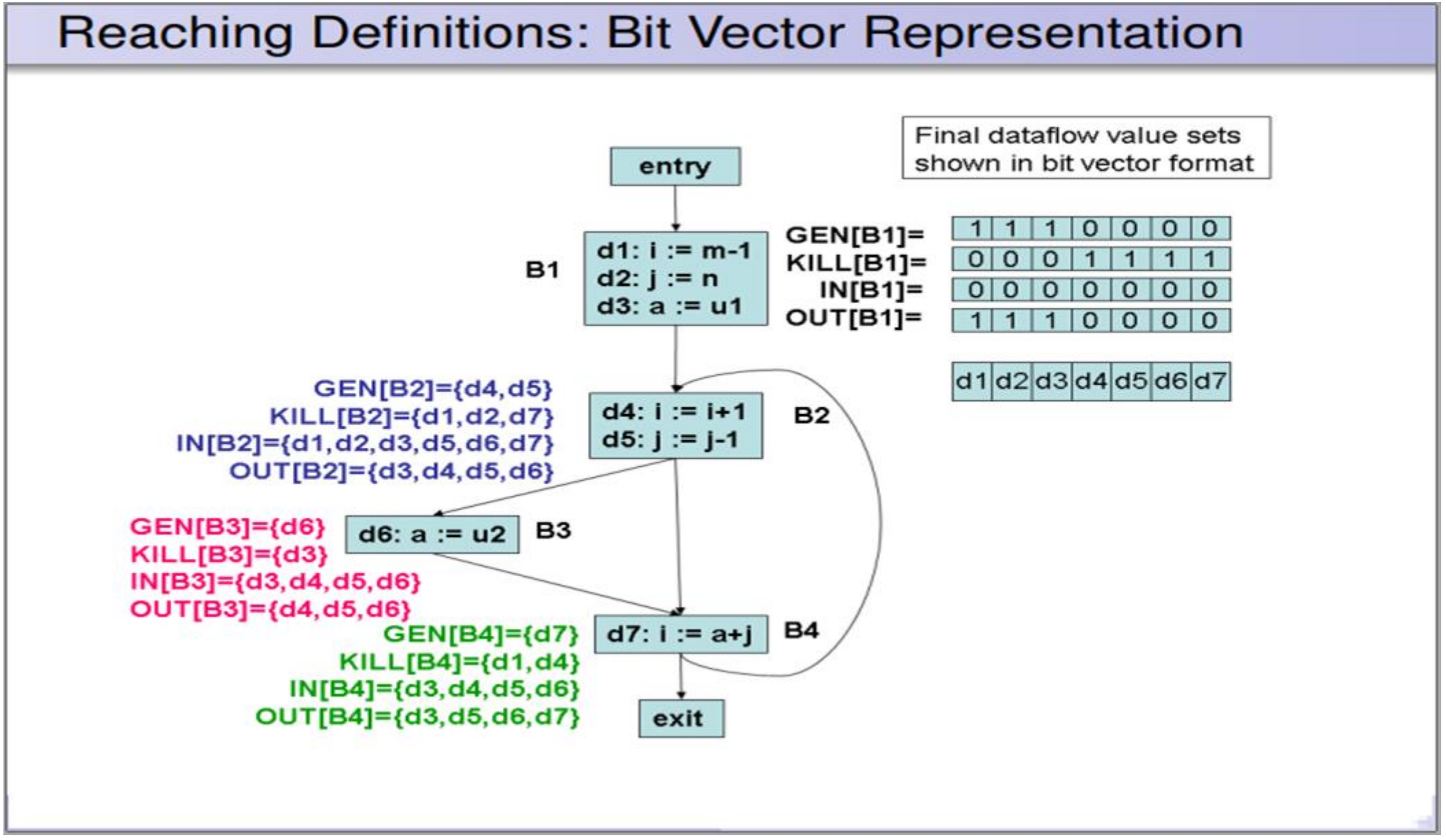
Global Data Flow Analysis-Terminology



Global Data Flow Analysis-Terminology

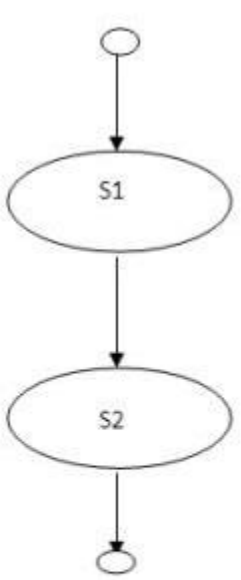


Global Data Flow Analysis-Terminology

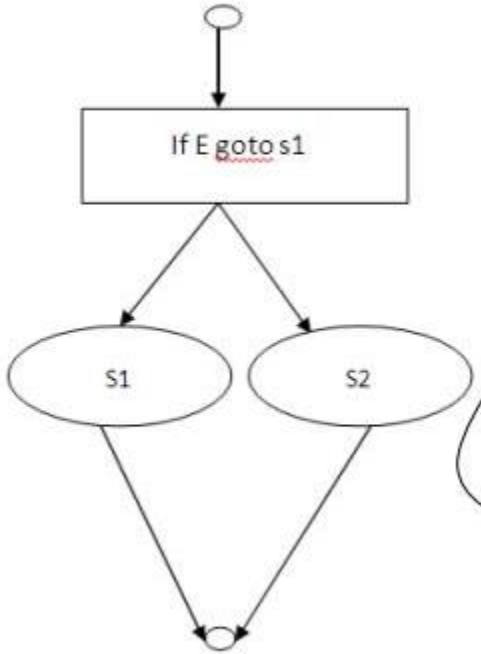


Global Data Flow Analysis-Terminology

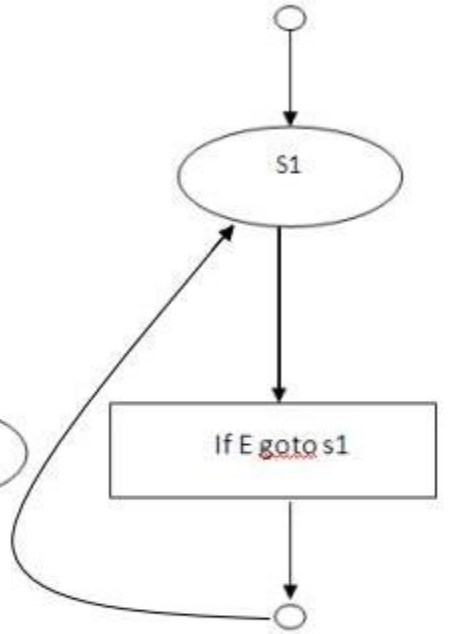
- S->id: = E| S; S | if E then S else S | do S while E
- E->id + id| id



S1 ; S2

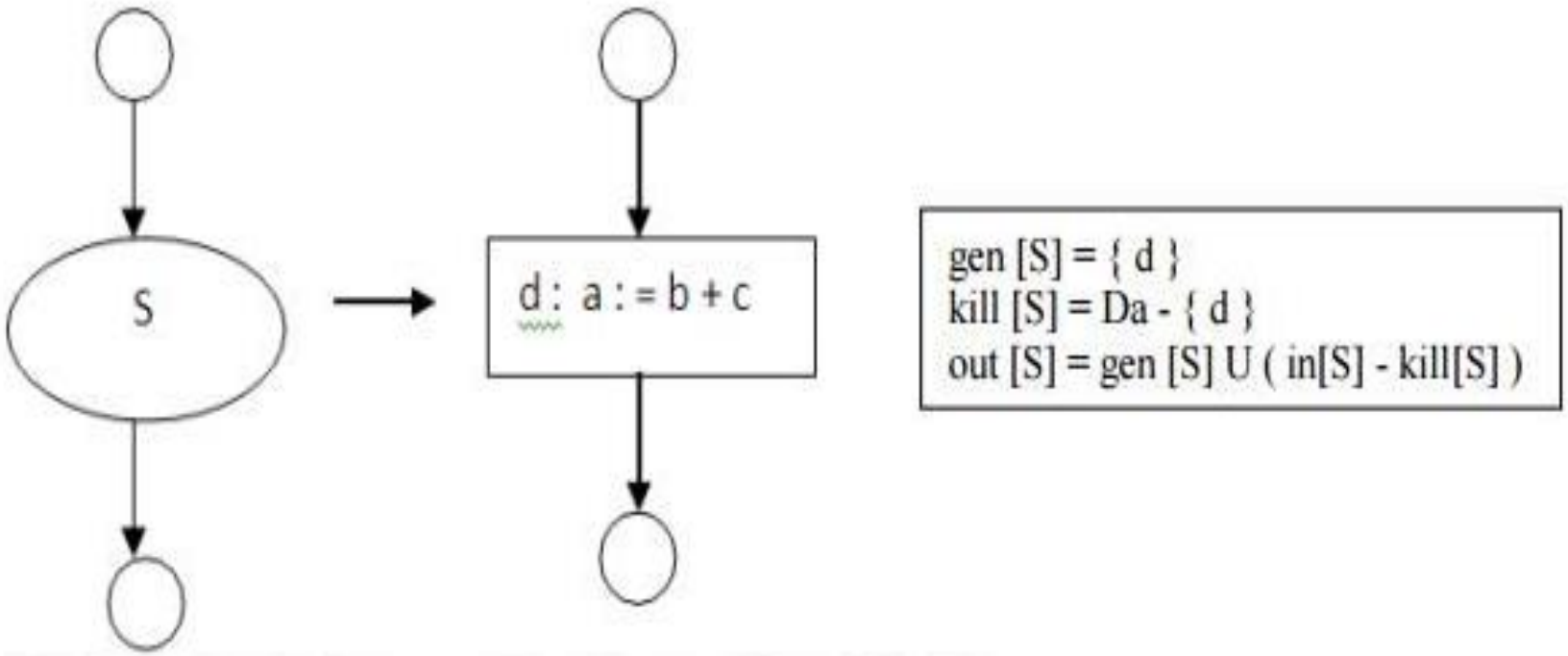


IF E then S1 else S2



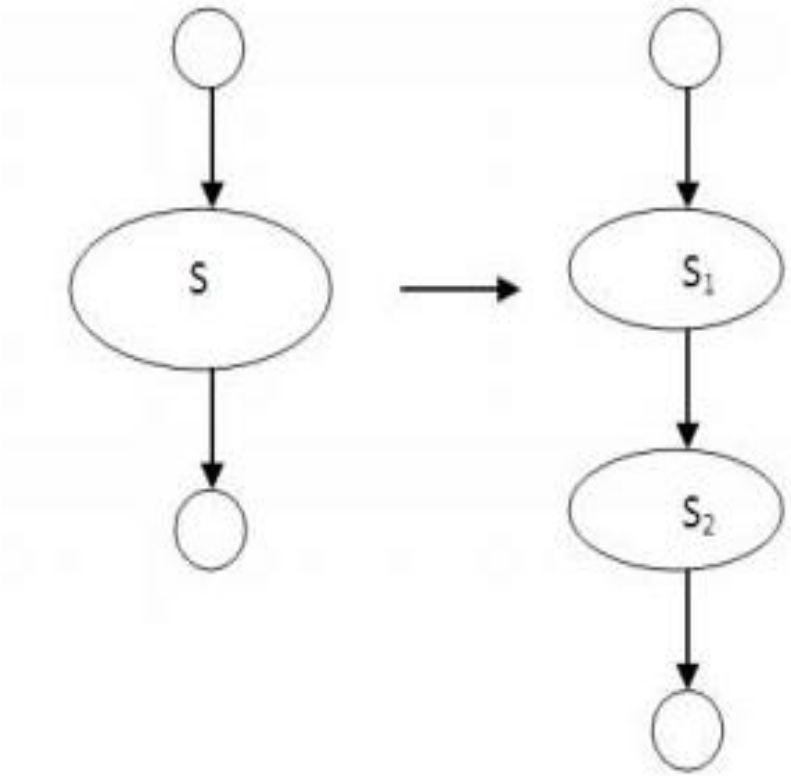
do S1 while E

Global Data Flow Analysis-Terminology



Da is the set of all definitions in the program for variable a.

Global Data Flow Analysis-Terminology



gen[S]=gen[S2] U (gen[S1]-kill[S2])
Kill[S] = kill[S2] U (kill[S1] - gen[S2])

in [S1] = in [S]
in [S2] = out [S1]
out [S] = out [S2]

Global Data Flow Analysis-Terminology

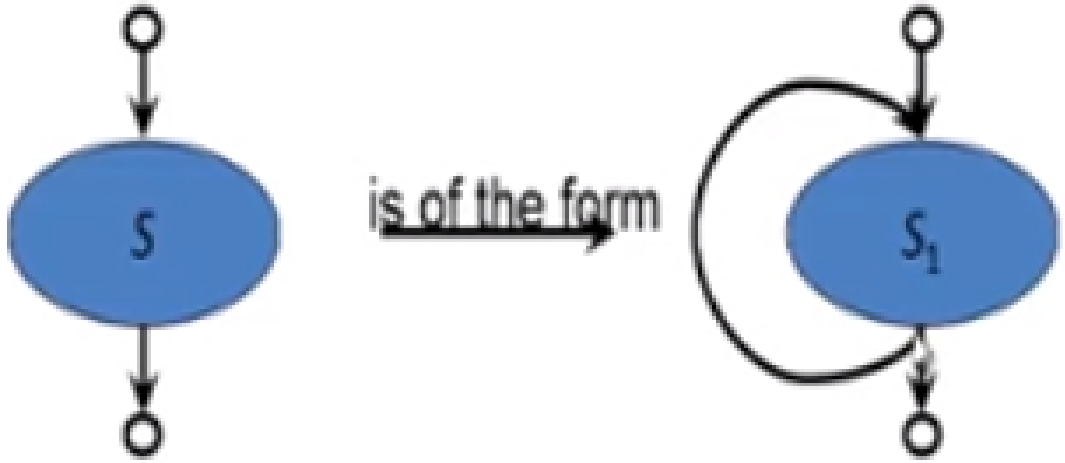
When S is of the form if ... then S1 else S2

- gen[S]=gen[S1]+gen[S2]
- kill[S]=kill[S1]∩kill[S2]
- in[S1] = in[S]
- in[S2] = in[S]
- out[S] = out[S1]+out[S2]

Global Data Flow Analysis-Terminology

When S is of the form do S1 while ...:

- gen[S] = gen[S1]
- kill[S] = kill[S1]
- in[S1] = in[S]+gen[S1]
- out[S] = out[S1]



Global Data Flow Analysis-Terminology

- Representation of Sets:
  - Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position i if and only if the definition numbered i is in the set.
  - The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implement complement of B and then using logical and to compute A

Use-Definition Chains (UD Chains)

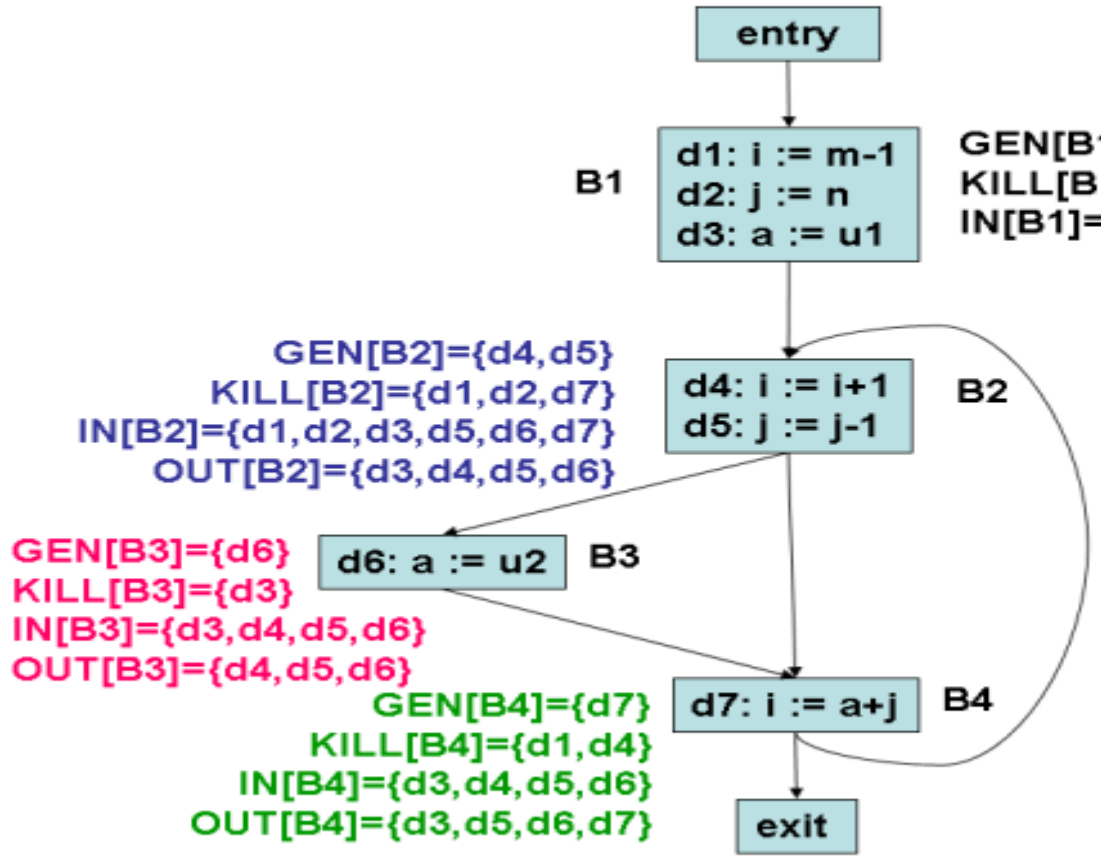
- It is often convenient to store the reaching definition information as “use definition chains” or “ud-chains”.
- UD chains may be constructed once reaching definitions are computed.
- A u-d chain is a list of a use of a variable and all the definitions that reach that use.

Use-Definition Chains (UD Chains)

- UD chain construction rules are as follows:
  - Case1: If use *u*1 of a variable *b* in block B is preceded by no unambiguous definition of *b*, then attach all definitions of *b* in *IN*[*B*] to the u-d chain of that use *u*1 of *b.*
  - Case2: If any unambiguous definition of *b* precedes a use of *b*, then *only that definition* is on the u-d chain of that use of *b.*
  - Case3: If any ambiguous definitions of *b* precede a use of *b*, then each such definition for which no unambiguous definition of *b* lies between it and the use of *b*, are on the u-d chain for this use of *b.*

Use-Definition Chains (UD Chains)

Use-Definition Chain Example



use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	(d3,d6)
(j,d7)	(d5)