Unit V

UNIT V CODE OPTIMIZATION AND CODE GENERATION **9**

Principal Sources of Optimization-DAG- Optimization of Basic Blocks-Global Data Flow Analysis-Efficient Data Flow Algorithms-Issues in Design of a Code Generator - A Simple Code Generator Algorithm.

TOTAL: 45 PERIODS

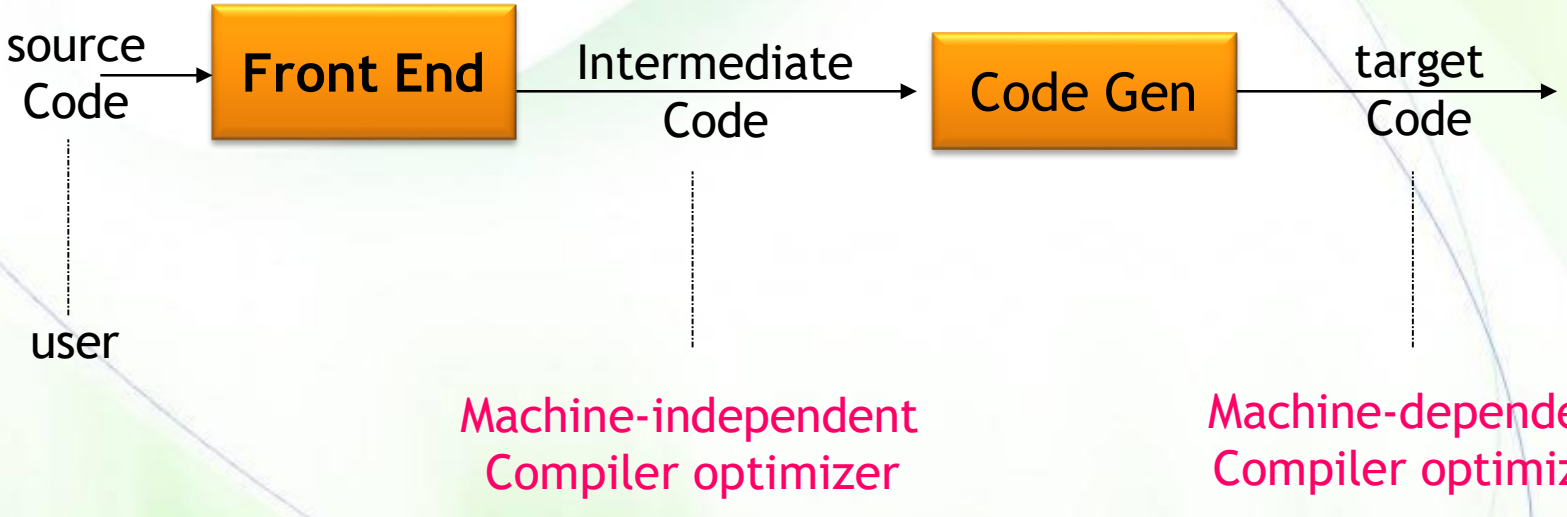
Optimization: The Idea

- Transform the program to improve efficiency

- **Performance**: faster execution
- **Size**: smaller executable, smaller memory

- Tradeoffs:
  - **Performance** vs. **Size**
  - **Compilation speed** and **memory**

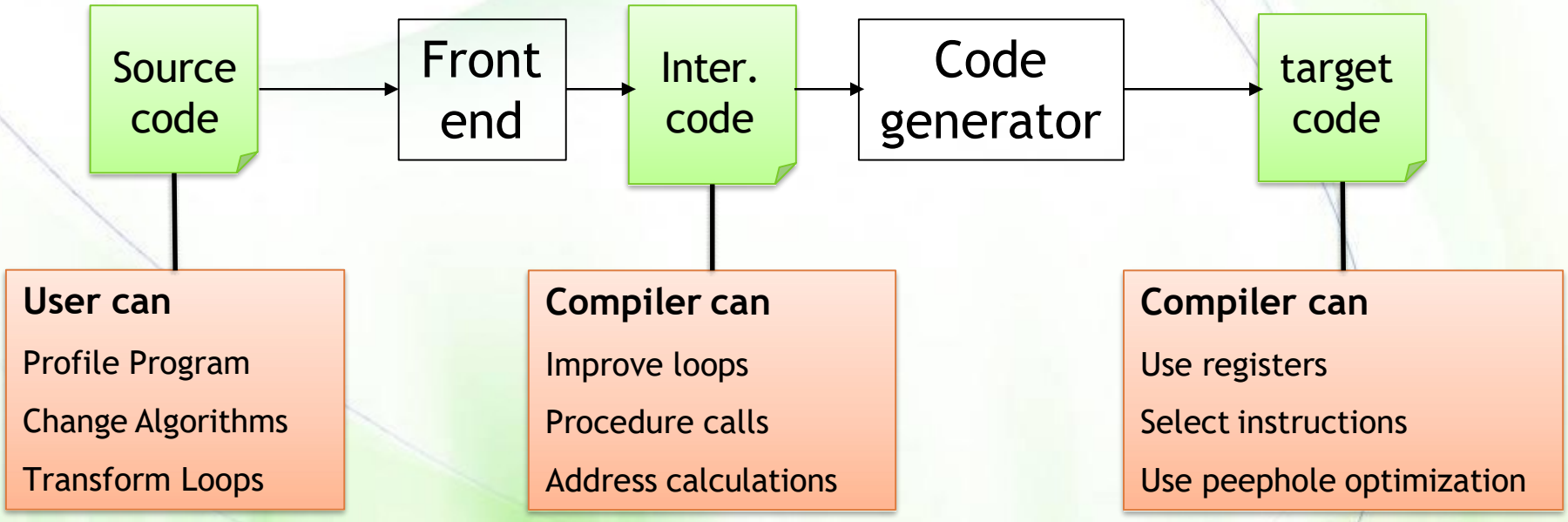
- A main goal is to achieve a better performance



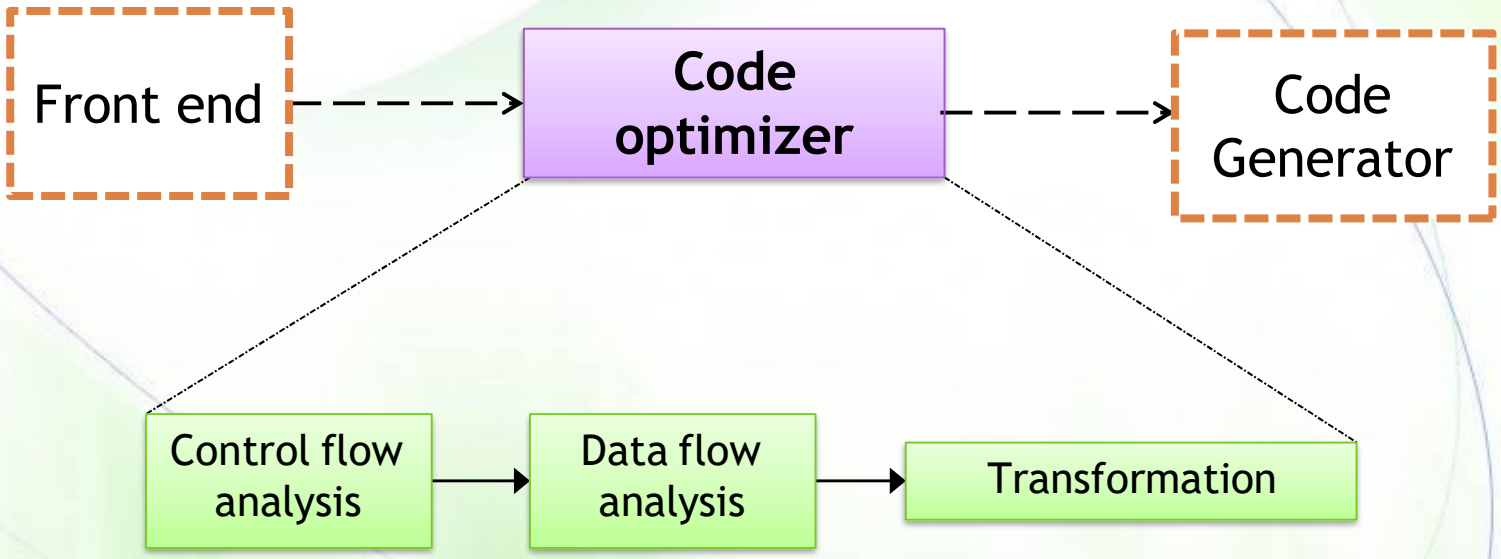
Criteria needed to perform Code Optimization

- The transformation must preserve the meaning of the program (ie) optimization should not change the output of the program for the given input or it should not raise an error.
- The transformation should reduce the running time drastically and reduce the usage of memory
- The transformation should be somewhat worth.

- Optimization can be done in almost all phases of compilation.



General Organization of an Optimizing Compiler



- Control Flow Analysis
  - Analysis is on control structure (loops and conditional statements)
  - Spends more times in loops
- Data Flow Analysis
  - Previous computations of data need to be again recomputed
- Transformation
  - Neglect repeated data & looping structure

Basic blocks (BB)

- **Basic block:** a sequence of consecutive statements with exactly 1 entry and 1 exit without any halt or branching.

Algorithm to Partition a sequence of statements into BBs

1. **Determine leaders** (first statements of BBs)
   i. The first statement is a leader
   ii. The target of a conditional is a leader
   iii. A statement following a branch is a leader
2. **For each leader,** its basic block consists of the leader and all the statements up to the next leader.

- B1→B2 if and only if B2 can be executed immediately after B1.

Example

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

Three-address code

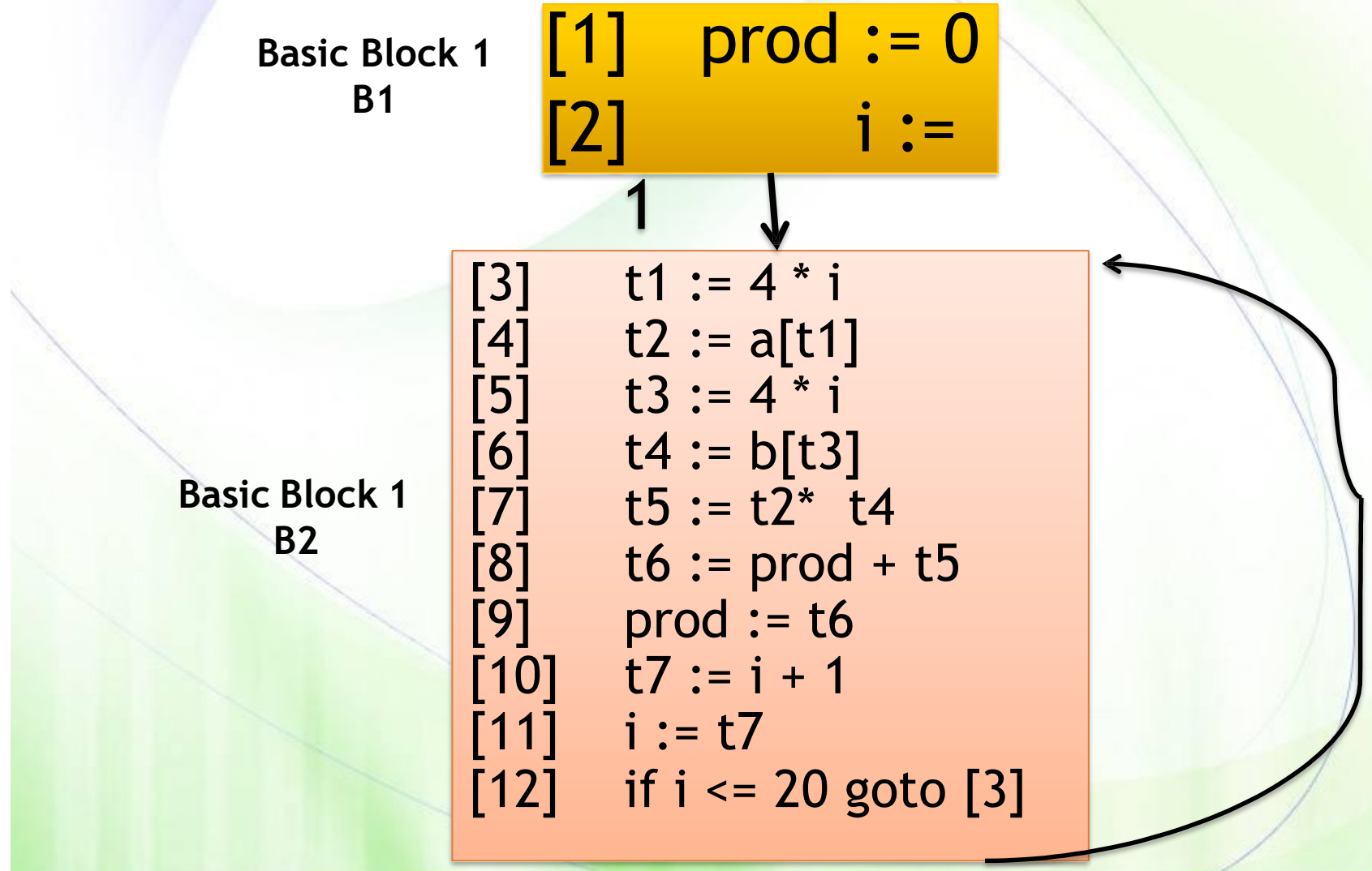
```
[1]   prod := 0
[2]   i := 1
[3]   t1 := 4 * i
[4]   t2 := a[t1]
[5]   t3 := 4 * i
[6]   t4 := b[t3]
[7]   t5 := t2*  t4
[8]   t6 := prod + t5
[9]   prod := t6
[10]  t7 := i + 1
[11]  i := t7
[12]  if i <= 20 goto [3]
```

Statement 1 is a leader by rule (i)

Statement 1 is a leader by rule (i),(ii)

Note:
4 * i is done because array element occupies 4 bytes

Flow graph for above Example



Transformations

- Possible changes performed on the basic blocks without affecting the nature of the basic blocks
  (ie) reducing overall running time or reducing space

↓

Optimizing the code

2 types

- Structure Preserving Transformations
- Algebraic Transformations

STRUCTURE-PRESERVING TRANSFORMATIONS

1. Common sub-expression elimination
2. Dead-code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

Structure Preserving Transformations

Common sub-expression elimination

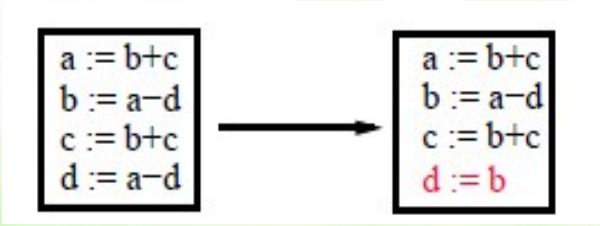
Consider the basic block

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

Compute with values
a = 5, b = 4, c = 3, d = 2

- The 2nd and 4th statements compute the same expression,

We can transform into



Contd ...

Dead Code Elimination:

- A variable is said to be in **live state** if it's used otherwise it's in the dead state

Renaming of temporary variables:

t := b+c

and u := b+c

- Change all instances from 't' to 'u' without affecting the basic block.

Interchanging 2 adjacent elements:

Consider,

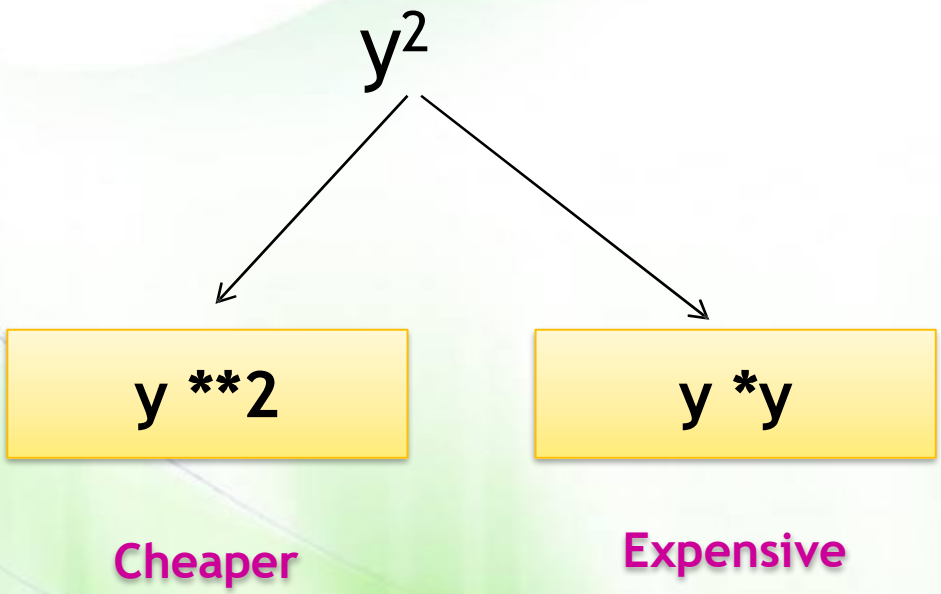
x:= y+z

a:=b+c

- Easy to exchange 2 statements without affecting the structure.

Algebraic Transformations

- Mathematical transformations
- Expensive mathematical operations are transformed to cheaper operations



Flow graphs

- It is a flow chart-like graph to represent a program where

  nodes -- basic blocks

  edges -- flow of control.

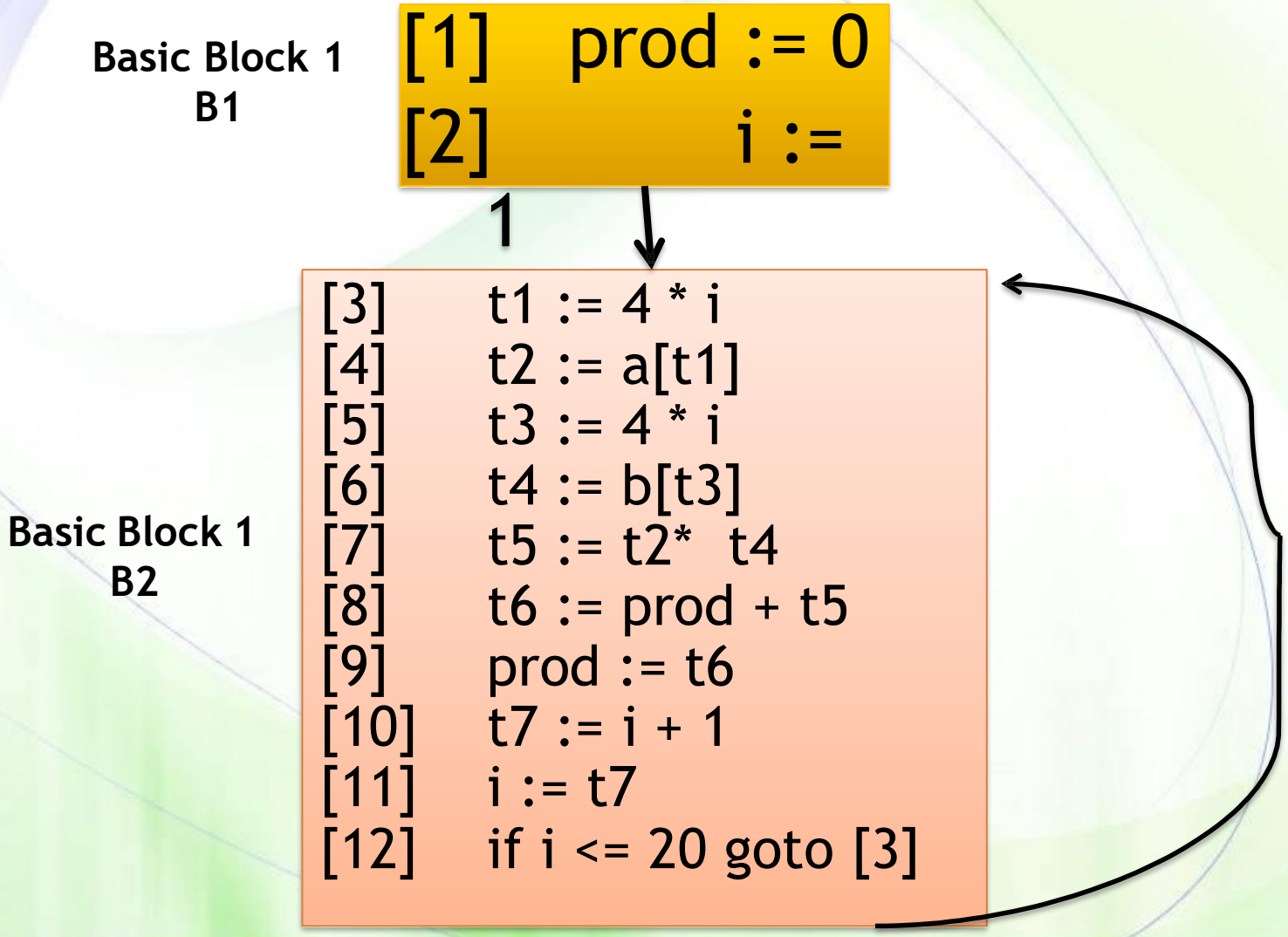
Rules:

If there is a directed edge from block B_1 to block B_2, if B_2 can immediately follow B_1 only if

- There is conditional/unconditional jump from the last statement of B_1 to the first statement of B_2, or
- B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.

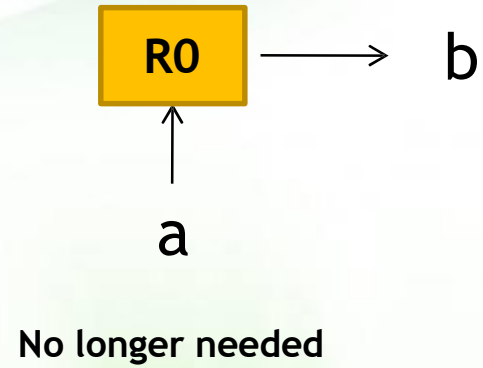
Now, we say that B_1 is the predecessor of B_2, and B_2 is a successor of B_1.

Flow graph for above Example



Next use Information

- Whenever a value in the register is no longer needed, we can allocate this register to another register.



- Next use information is a collection of names that are used subsequently.

Principle sources of optimization

A Running Example: Quicksort

Repeatedly asked 16 mark Question

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

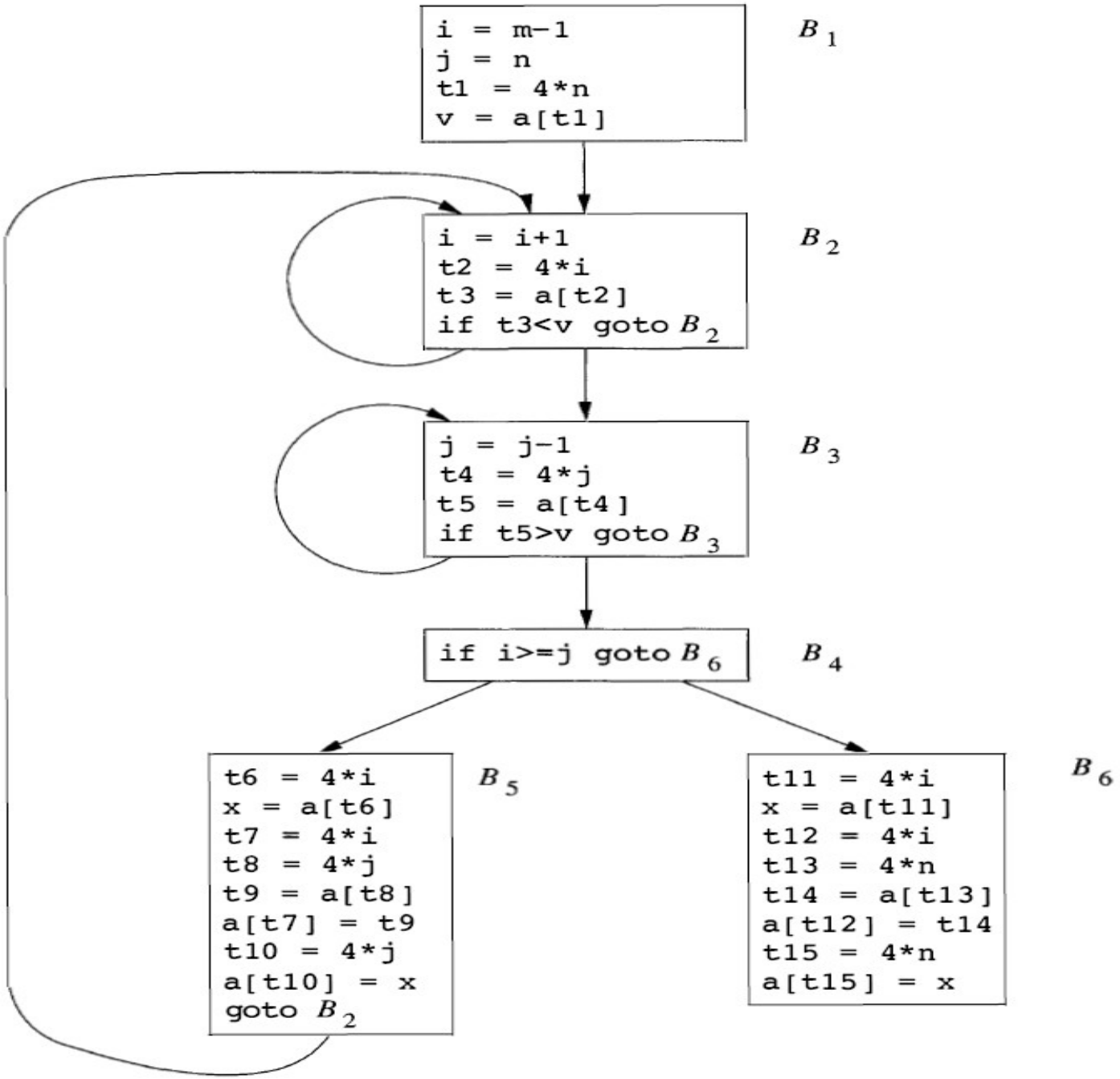
Three Address Statement for the Quicksort Algorithm

```
(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
```

Leader by Rule 1

Leader by Rule 2

Leader by Rule 3


i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
B5
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
B6

Principle Sources of Code Optimization

Repeatedly asked 16 mark Question

- **Transformations**
  - Local Transformation
    - Transformation inside the Block
  - Global Transformation
    - Transformation between blocks

Types of Transformations

- **Function Preserving Transformations**
  - Common-sub expression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding and Constant propagation
- **Loop Optimizations**
  - Code Motion
  - Induction variable elimination
  - Reduction in strength

Function Preserving Transformations

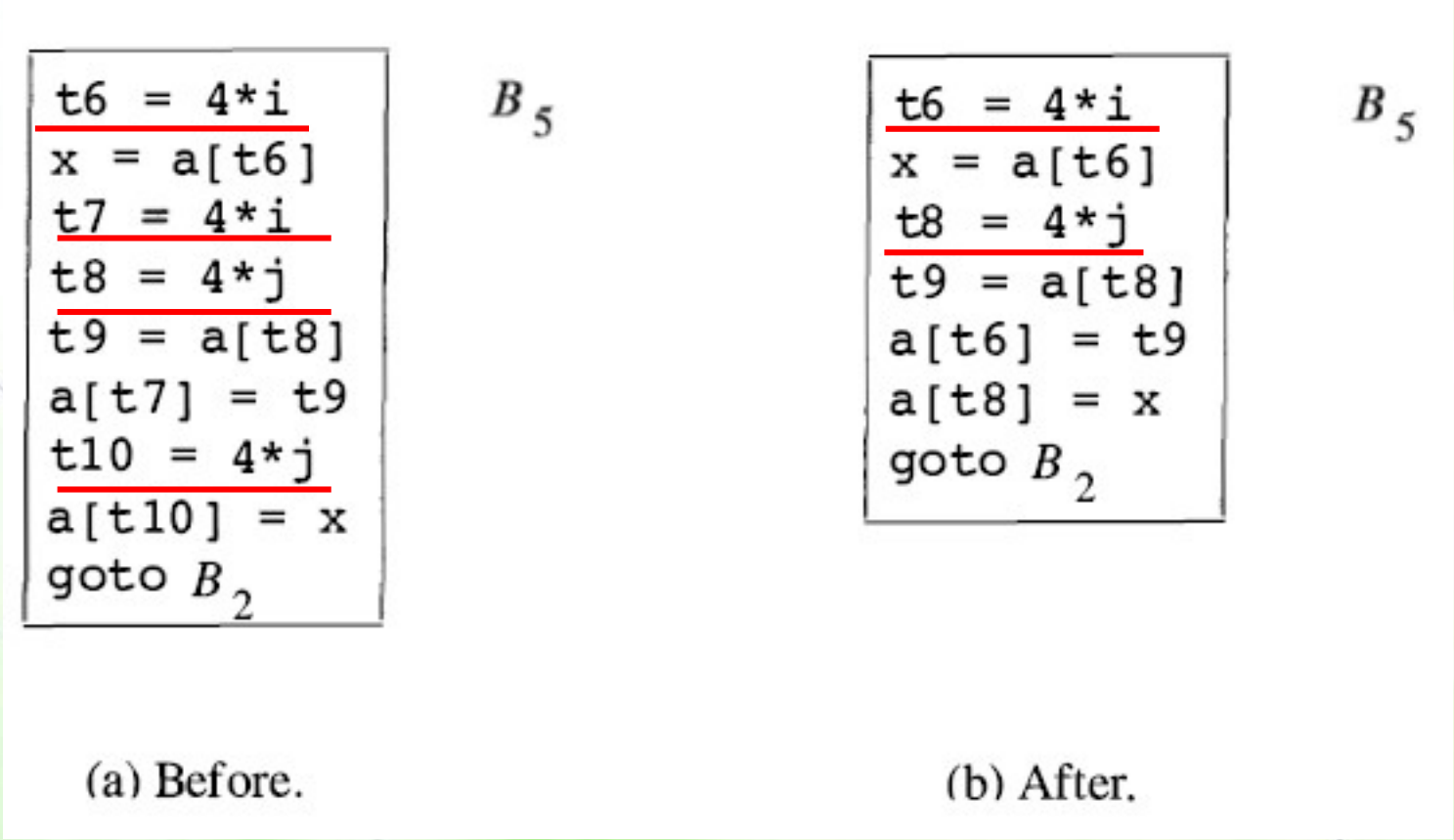
- Common-sub expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

Common sub expression Elimination

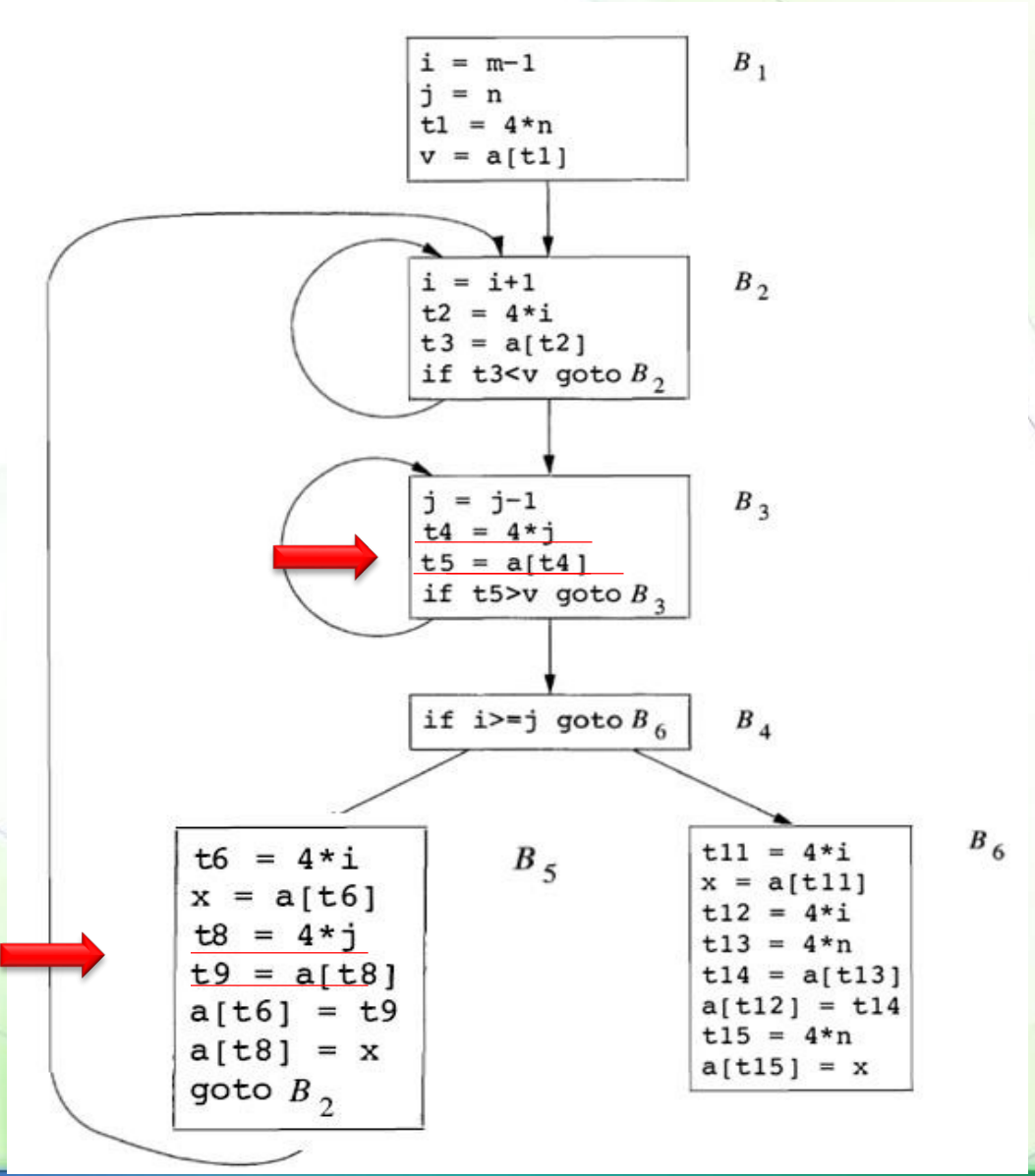
- An occurrence of an expression E is called common sub expression if E was previously computed and the values of variables have not changed since previous computation.

- Consider Block 5 (B5) → Eliminate Common subexpression

Local Transformation

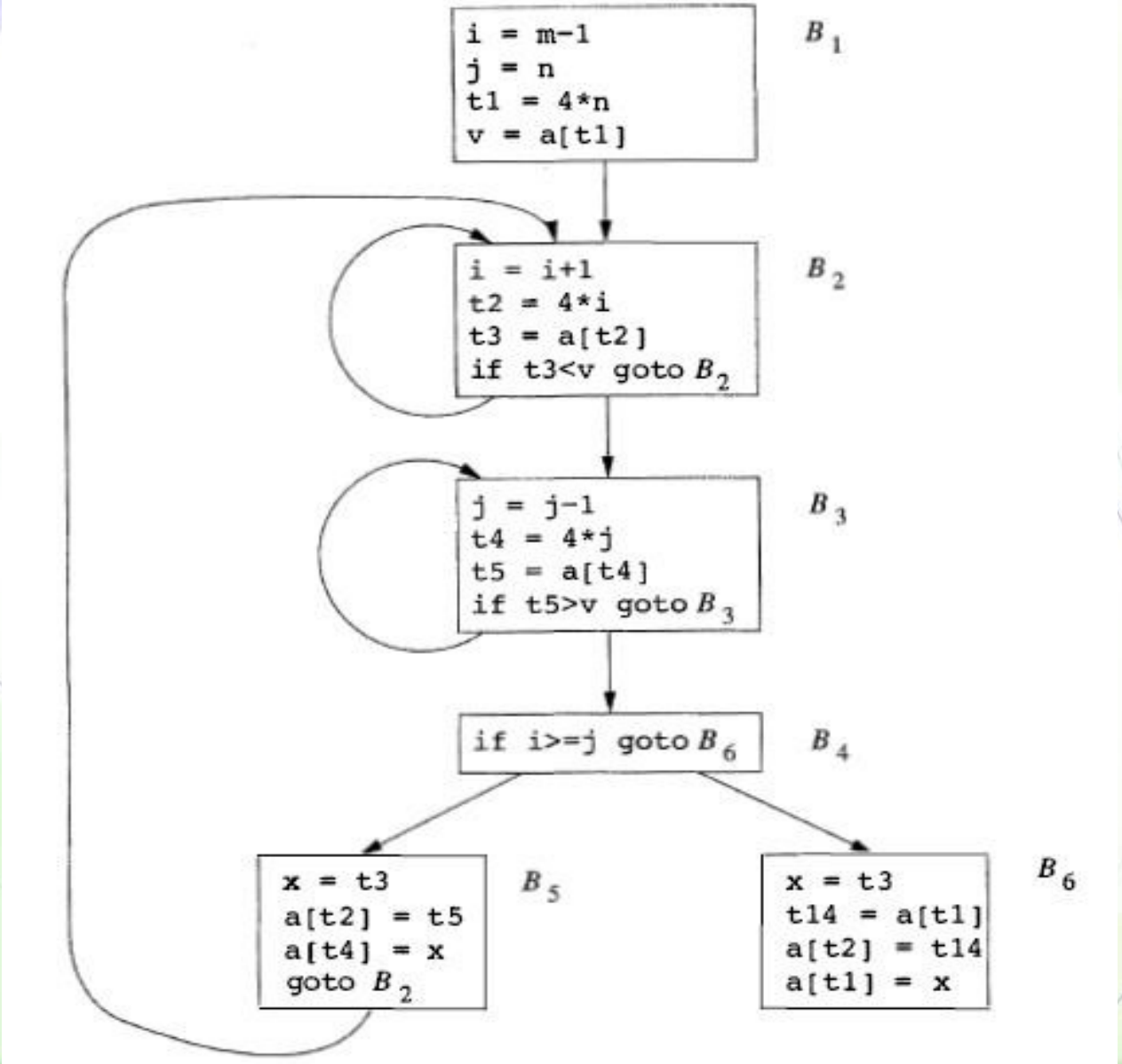


Global Transformation



After both Local and Global Transformations

(ie). After common sub expression elimination

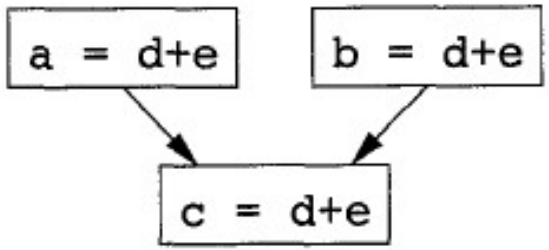


Copy Propagation

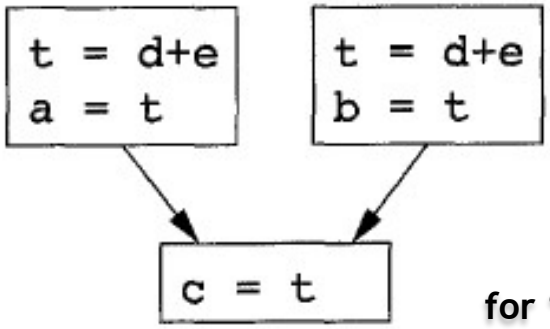
- If any statement of the form

  u = v

  Then it is called as **copy statement** or copies



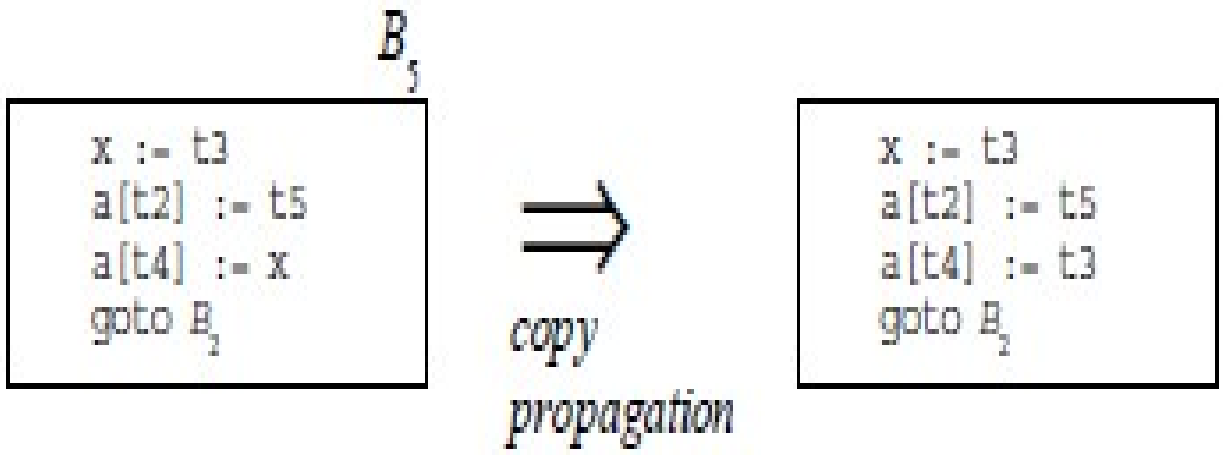
(a)



(b)

for 'c', it is incorrect to assign c := a (or) c := b

- Copy Propagation Contd…



- Advantages:
  - **It will not perform any reduction in statements but the number of variables used are reduced.**

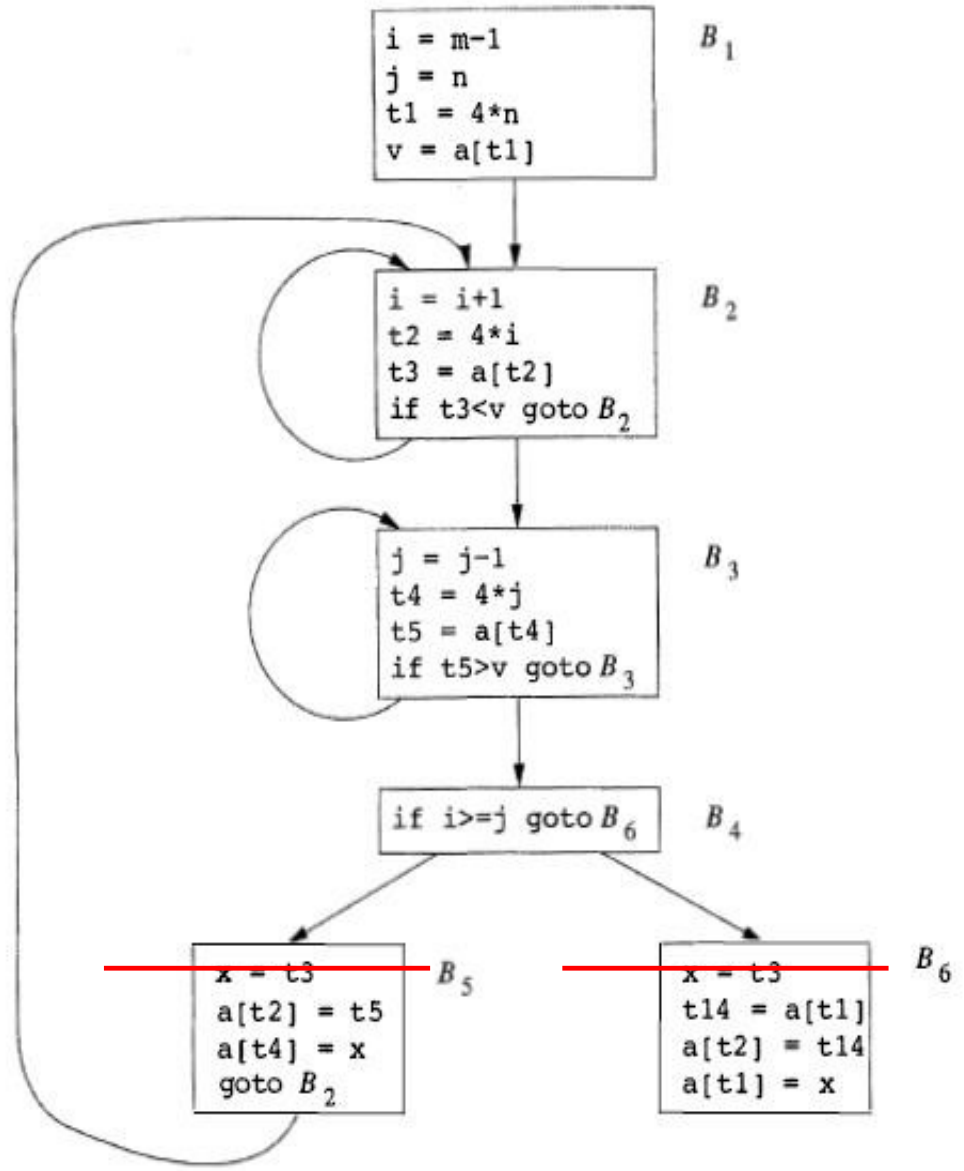
Dead-Code Elimination

- ***Live*** variable
  - A variable is **live** at a point in a program if its value can be used subsequently;
  - otherwise, it is ***dead*** at that point.

Example:

```
i := 0;
if (i ==1)
{
   c = d *e;
}
```

It is a dead code (since 'i' is assigned to 0)



Consider B5

Since ‘x’ is in a dead state, its statement can be eliminated.

Constant folding and constant propagation

- **Constant folding**: Evaluation of an expression with constant operands.

  Example:

  - Area of the square $A = \pi r^2$
  - Instead of having π * (r pow 2), we can directly substitute, 3.14 * (r pow 2)

- **Constant Propagation**: Replace a variable with constant which has been assigned to it earlier.

  Example:

```
pi := 3.14286
area = pi * r ** 2
```

Loop Optimizations

- Optimizations on loop (i.e) time taken for executing the loops.
  - Code Motion
    - Moves code outside a loop
  - Induction variable elimination
    - Eliminate a induction variable (i.e) variables i,j in block B2 & B3 respectively
  - Reduction in strength
    - Replaces expensive operations by cheaper one (i.e) multiplication by addition.

Code Motion

- An important modification that decreases the amount of code in a loop.

- **Loop-invariant computation**
  - An expression that yields the same result independent of the number of times a loop is executed.

```
while (i <= limit-2)
```



```
t = limit -2
while (i <= t)
```

Induction Variable Elimination & Reduction in strength

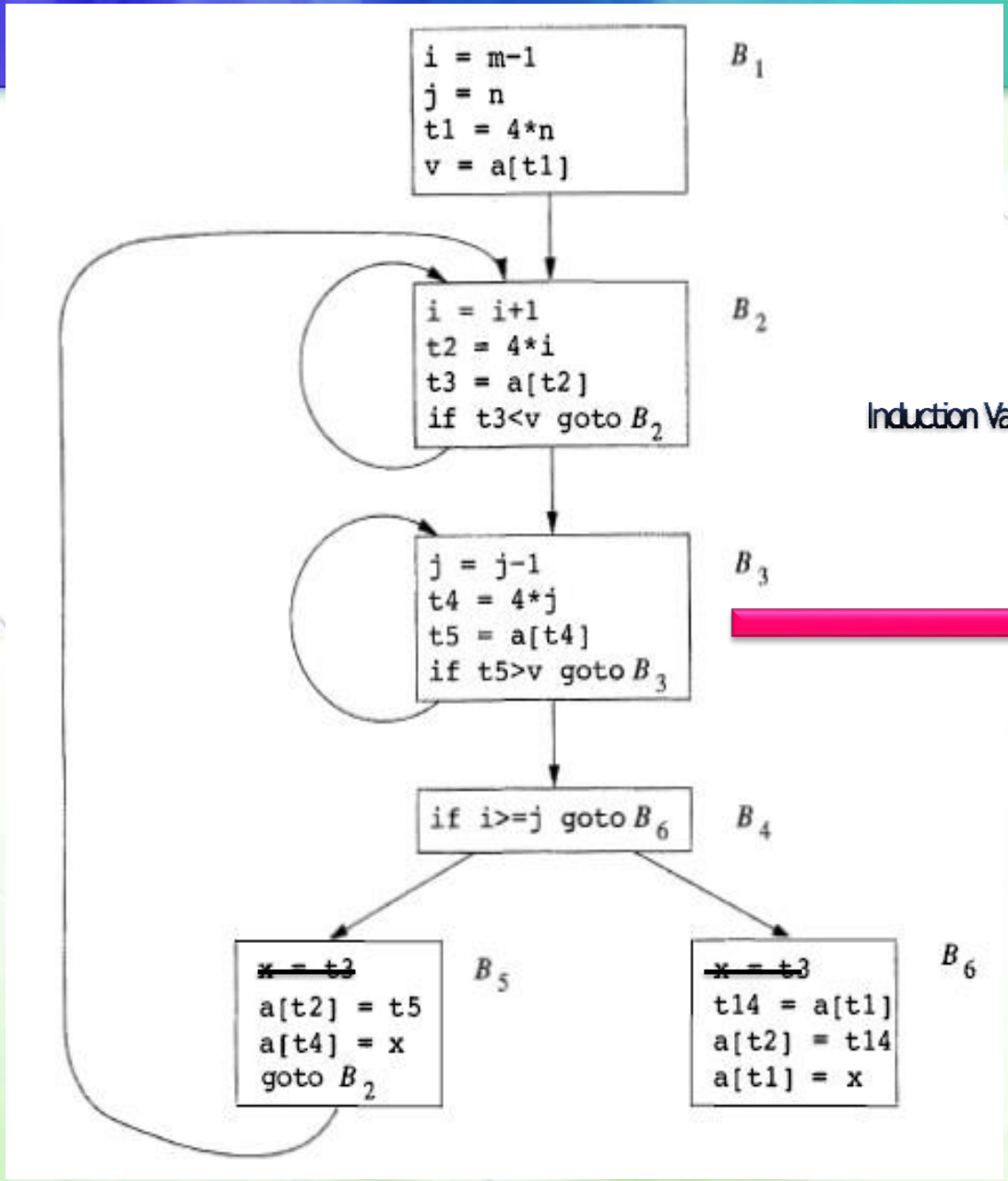
When j := 4
t4 := 4 * j
:= 4 * 4
t4 := 16

When j := 3
t4 := 4 * 3
:= 4 * 3
t4 := 12

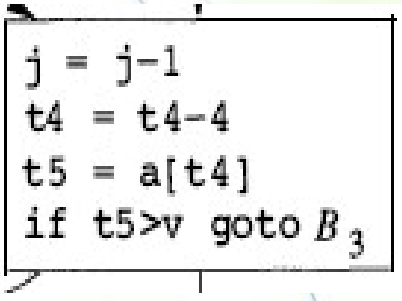
When j := 2
t4 := 4 * 2
:= 4 * 2
t4 := 8

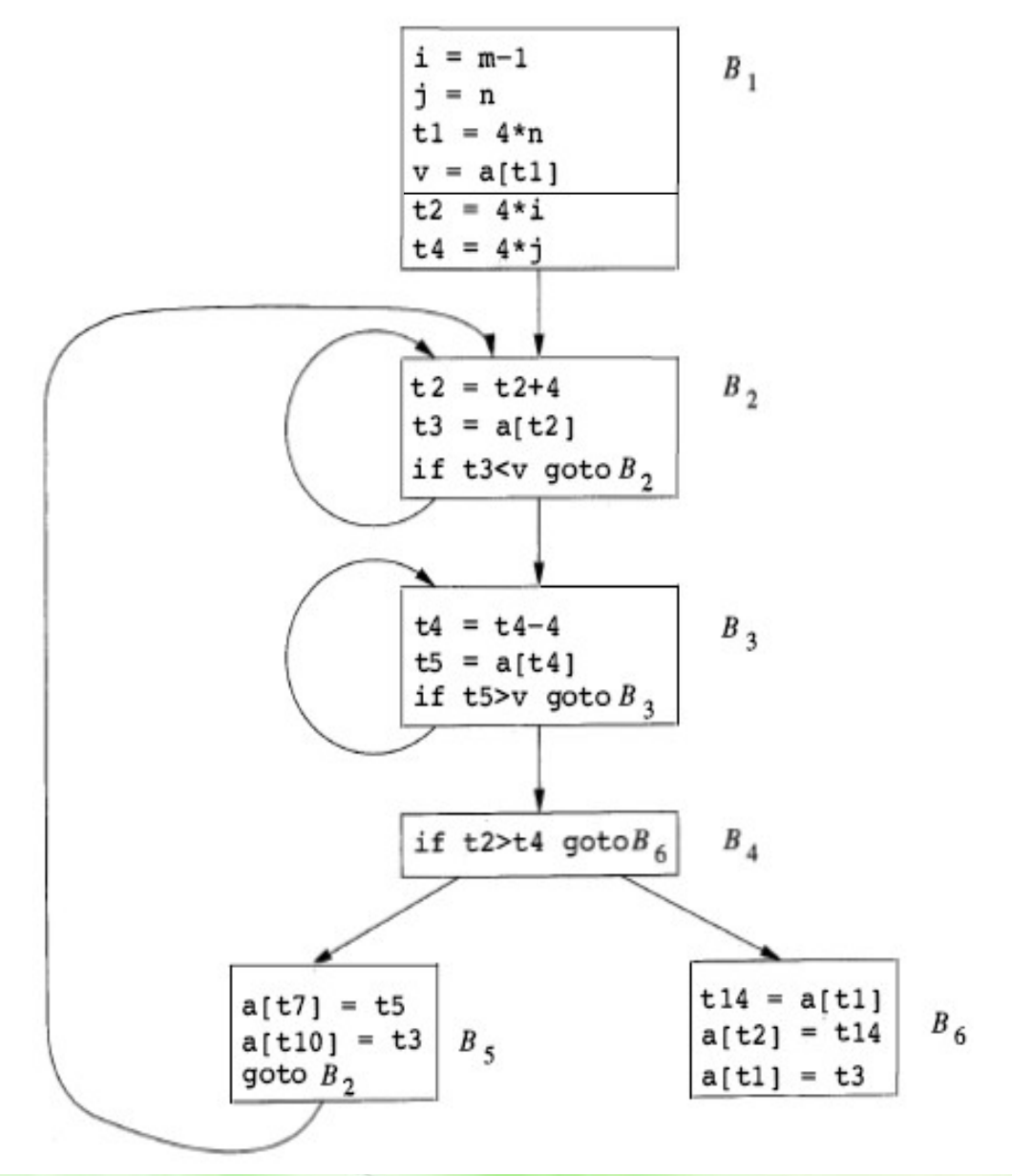
When j := 1
t4 := 4 * 1
:= 4 * 1
t4 := 4

**when j is decremented by 1,
t4 is decremented by 4.**



Induction Variable Elimination





Final result after all transformations

- **Loop unrolling:**
  - Duplicates the body of the loop multiple times

```
for (i=0; i<50; i++)
{
   a[i] = b[i];
}
```



```
for (i=0; i<50; i = i+2)
{
   a[i] = b[i];
   a[i+1] = b[i+1];
}
```

Optimization of Basic Blocks

- Algebraic Identities

  - Important class of optimizations of Basic Blocks
    - Arithmetic

```
x + 0 = 0 + x = x
x - 0 = x
```

```
x * 1 = 1 * x = x
x / 1 = x
```

    - Reduction in strength

```
x ** 2 = x * x
x / 2  = x >> 1
```

```
2 * x = x + x
4 * x = x << 2
```

    - Constant folding

```
x = 2*3.14*R
```

```
PI = 3.14
x = 2*PI*R
```



```
x = 6.28*R
```

    - Commutative and Associative laws

```
a := b + c
e := c + d + b
```



```
a := b + c
e := a + d
```

Loops in Flow graphs

- Algorithm for Identifying Loops:
  - **Compute Dominator Information.**
  - **Identify Back Edges.**
  - **Construct Loops corresponding to Back Edges.**

Dominators: Characteristics

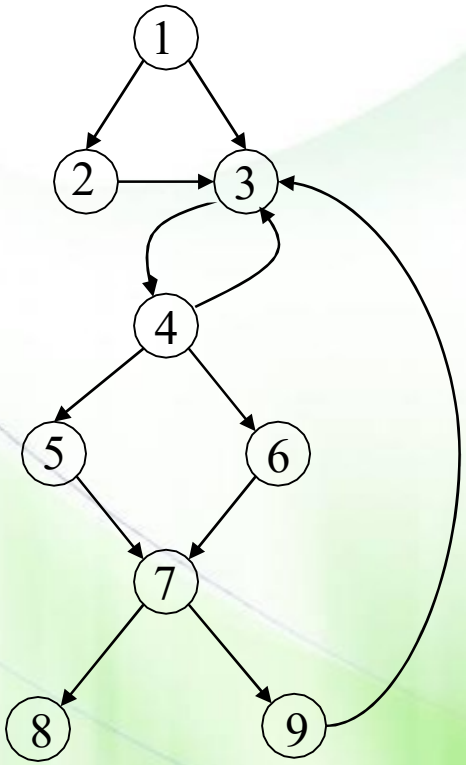
- Dominators:
  - In the flow graph **G** , a node **d** dominates a node **n**, if every path in G from the initial node to n goes through d.
    - Represented as: **d dom n**

Properties of Dominators

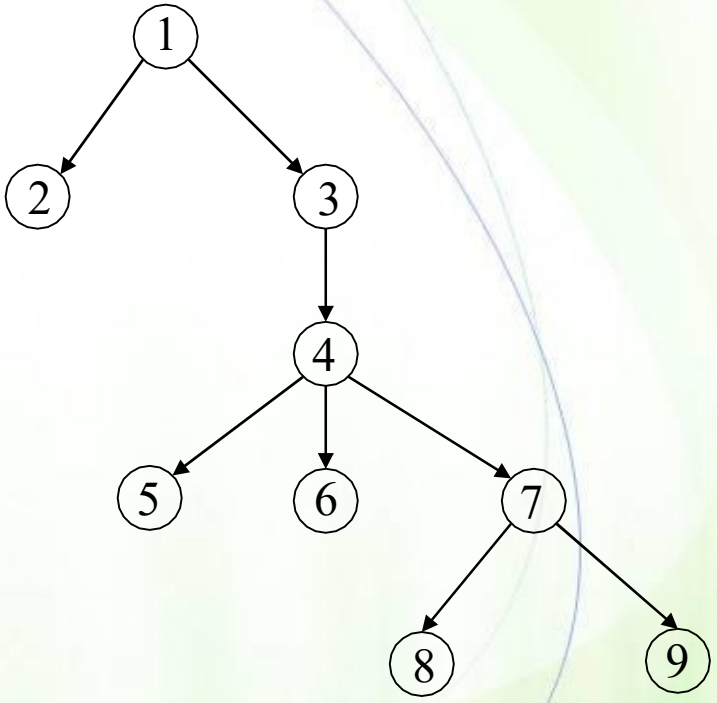
- Start node dominates every node in the flow graph.
- Every node dominates itself.
  - d dom d **(Reflexive Property)**
- If a dom b and b dom c then
  - a dom c **(Transitive Property)**
- if a dom b and b dom a then
  - a = b **(Anti-symmetric Property)**

Dominator Tree

- A useful method for presenting dominator information in a tree is called ***dominator tree***, in which initial node is the root, and each node d dominates only its descendants in the tree.



Flow Graph



Dominator Tree

Natural loops

1. A loop has a single entry point, called the “header”. Header dominates all node in the loop
2. There is at least one path back to the header from the loop nodes (i.e. there is at least one way to iterate the loop)

- Natural loops can be detected by *back edges*.
  - B*ack edges*: edges where the sink node (head) dominates the source node (tail) in *G*

Code improving transformations

- Some techniques for optimization may be applied to the code, which leads to better performance.
- The techniques may be tried for both local and global transformations.
  - Compile time evaluation
  - Common sub expression elimination
  - Variable propagation
  - Code movement optimization
  - Strength reduction
  - Dead code elimination
  - Loop optimization

- Compile time evaluation
  - Modifying the codes a little, where the expressions are pre-computed at the compilation time.
    - Constant folding
    - Constant propagation
- Common sub expression elimination
- Variable propagation (Copy Propagation)
- Code movement optimization
  - Reduce size – code space reduction
  - Reduce execution frequency – execution frequency reduction
- Strength reduction (* operation replaced by +)
- Dead code elimination
- Loop optimization

Code movement optimization

- Reduce size – code space reduction

```
if a<b then
   z = x**2
else
   y = x**2 +10

In both the cases, x**2 is calculated.
```



```
 temp = x**2;

if a<b then
   z = temp;
else
   y = temp +10
```

- Reduce execution frequency – execution frequency reduction

```
if (a<b) then
    z = x * 2
else
    y = 10
g = x * 2
```



```
if (a<b) then
    temp = x * 2
    z = temp
else
    y = 10
    temp = x * 2
g = temp;
```

Global Data-Flow Analysis

- Knowledge about the behavior of a variable is essential for performing transformations.
- Control flow information is also required to do transformations across basic blocks.
- In addition, to apply global optimizations on basic blocks, data-flow information is collected by solving systems of data-flow equations

Introduction to Global Data flow Analysis

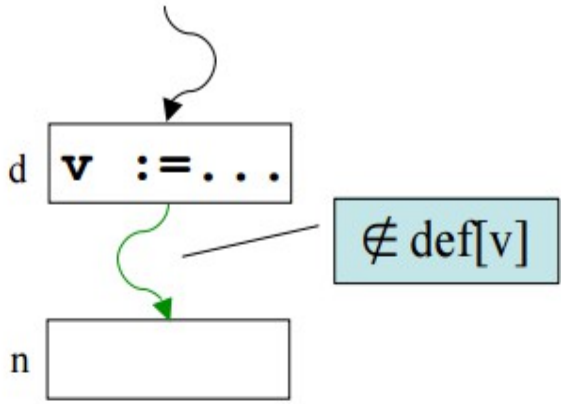
- Examination of the entire program to perform optimization is known as global data flow analysis.

- To perform global data flow analysis, we are going to use data flow equations.

Reaching Definition

Definition

– A definition (statement) d of a variable **v** **reaches** node n if there is a path from d to n such that **v** is not redefined along that path

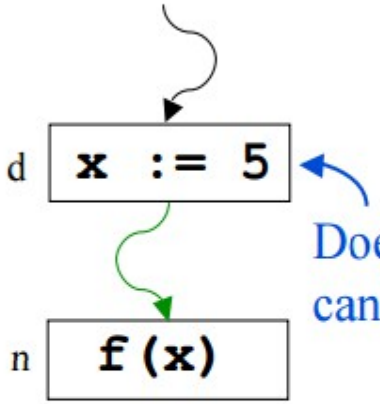


Uses of reaching definitions

– Build use/def chains

– Constant propagation

– Loop invariant code motion



Does this def of **x** reach n?
can we replace n with **f(5)** ?

```
1   a = . . .;
2   b = . . .;
3   for (. . .) {
4       x = a + b;
5       . . .
6   }
```

Reaching definitions of **a** and **b**

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of **a** or **b** inside the loop

Computing Reaching Definitions

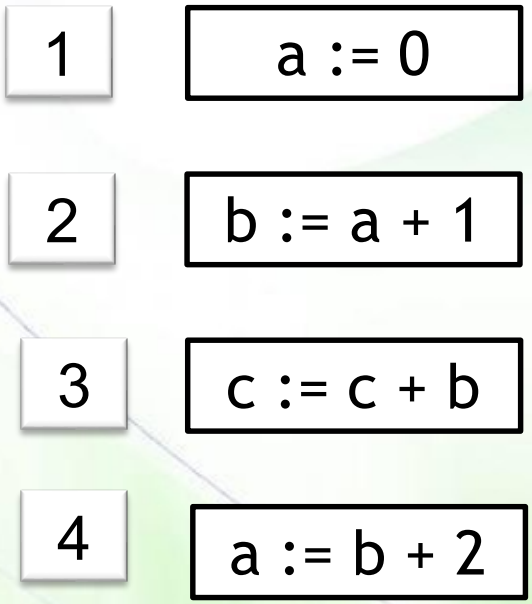
Gen[n]: Definitions that are generated by node n (at most one)

Kill[n]: Definitions that are killed by node n

Defining Gen and Kill for various statement types

statement	Gen[s]	Kill[s]	statement	Gen[s]	Kill[s]
s: t = b op c	{s}	def[t] – {s}	s: goto L	{}	{}
s: t = M[b]	{s}	def[t] – {s}	s: L:	{}	{}
s: M[a] = b	{}	{}	s: f(a,…)	{}	{}
s: if a op b goto L	{}	{}	s: t=f(a, …)	{s}	def[t] – {s}

Liveliness and use

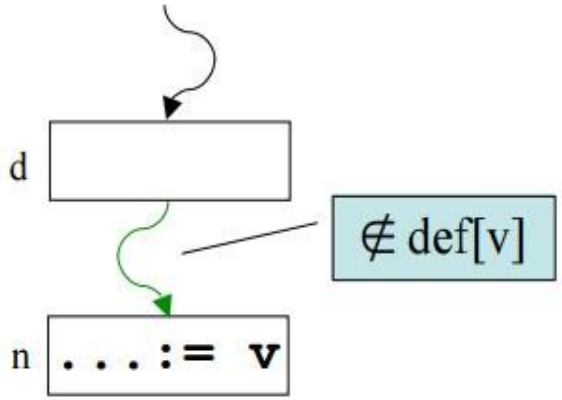


Node	Use	Gen
1		a
2	a	b
3	b,c	c
4	b	a

Liveness Analysis

Definition

– A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).



Uses of Liveness

– Register allocation

– Dead-code elimination

```
1   a = . . .;
2   b = . . .;
3   ...
4   x = f(b);
```

If **a** is not live out of statement 1 then statement 1 is dead code.

What Use is Liveness Information?

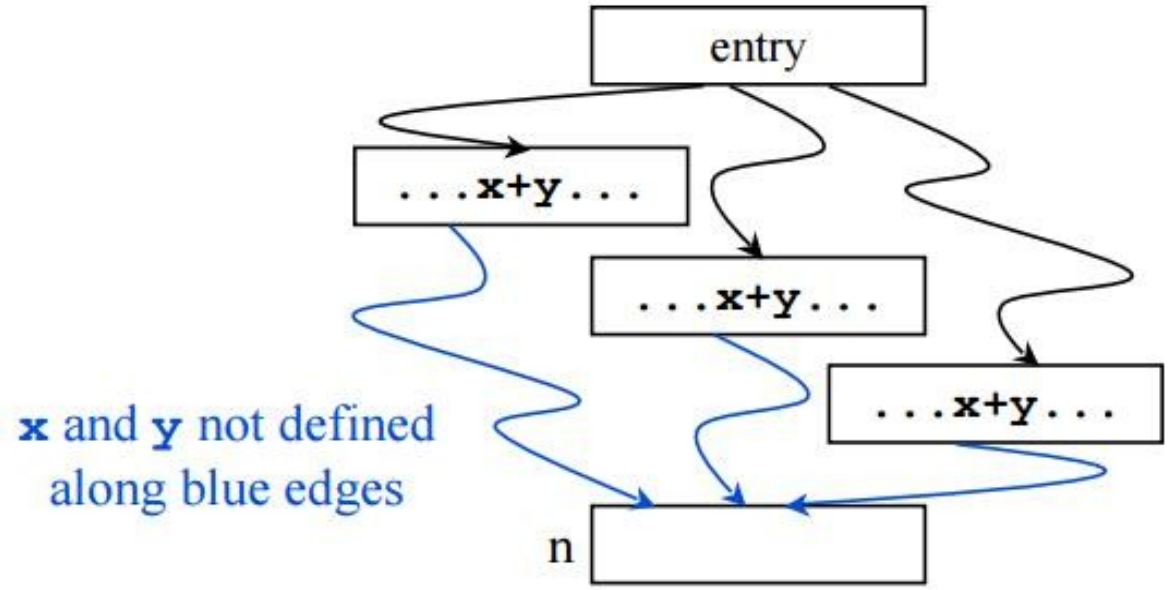
- **Register allocation.**
- – If a variable is dead, can reassign its register

- **Dead code elimination.**
- – Eliminate assignments to variables not read later.

Available Expressions

Definition

– An expression, **x+y**, is **available** at node n if every path from the entry node to n evaluates **x+y**, and there are no definitions of **x** or **y** after the last evaluation



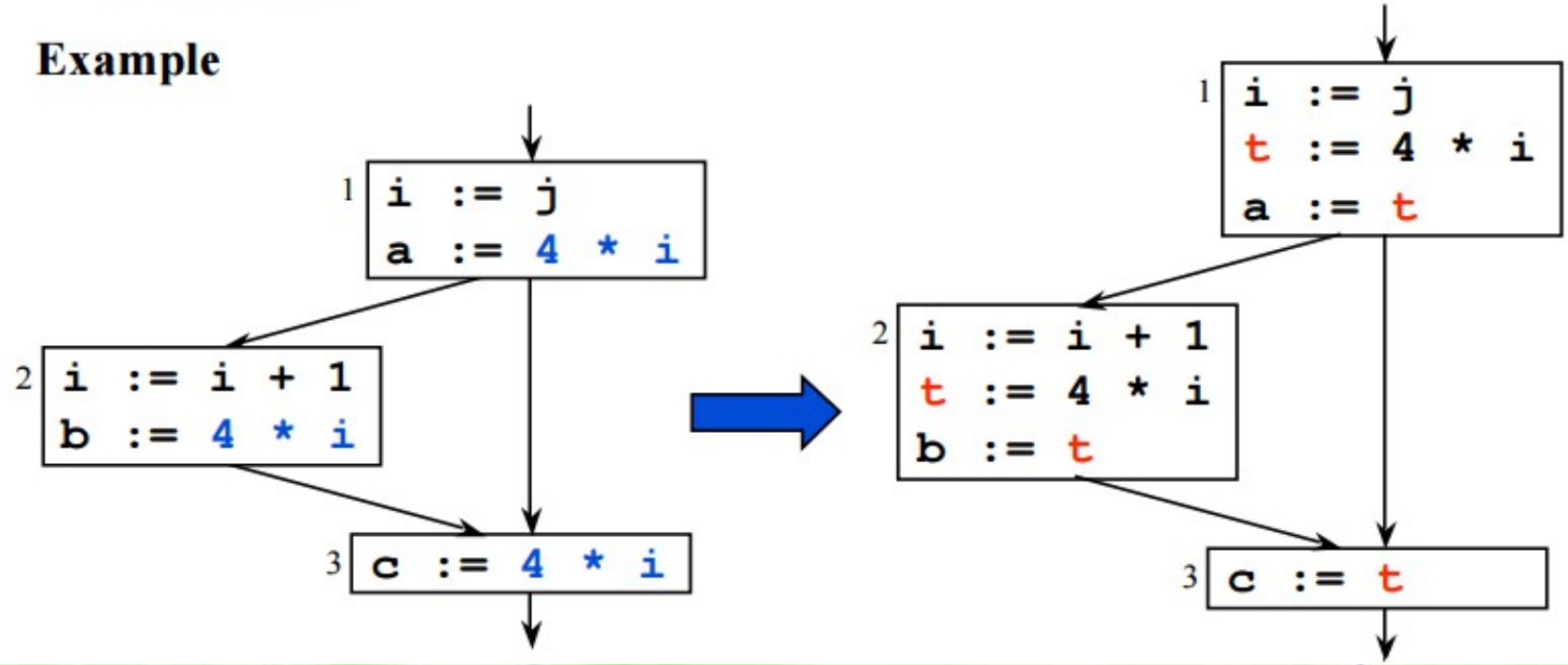
Available Expressions for CSE

How is this information useful?

Common Subexpression Elimination (CSE)

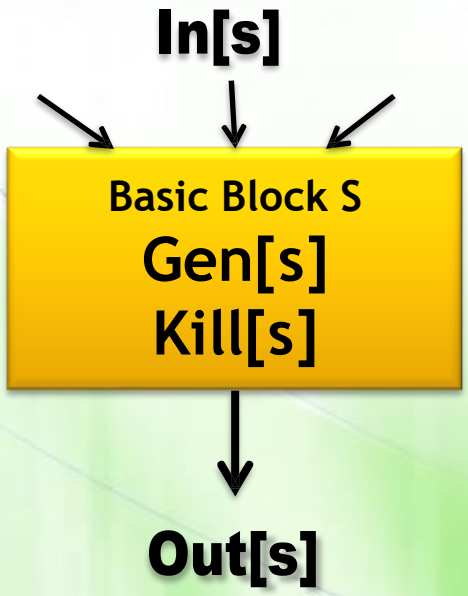
– If an expression is available at a point where it is evaluated, it need not be recomputed

Example



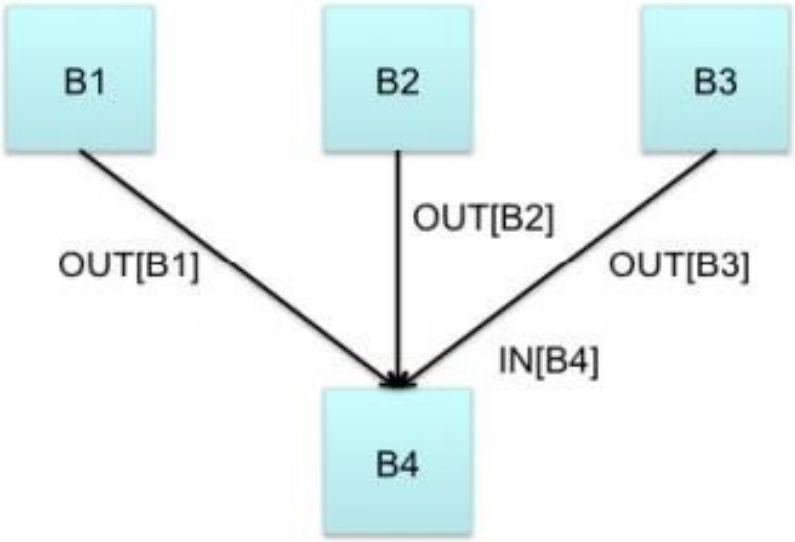
Data flow equation

- Components of data flow equations
  - **In[s]**: Any information given as the i/p to the Basic blocks.
  - **Gen[s]**: information generated/collected within the BB.
  - **Kill[s]**: information (i.e) lost its values in the BB.
  - **Out**[s]: Output generated by basic block.

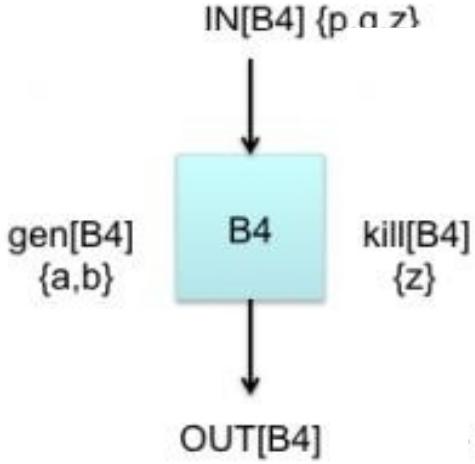


A typical dataflow equation has the form

out[S] = gen[S]∪(in[S]−kill[S])

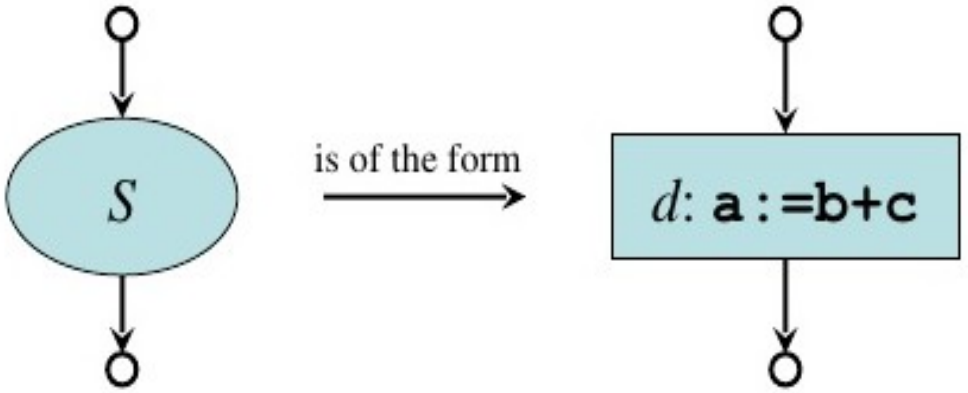


$IN[B4] = OUT[B1] \cup OUT[B2] \cup OUT[B3]$



$OUT[B4] = gen[B4] \cup (IN[B4] - kill[B4])$

Data flow equations for Reaching definitions (RD)



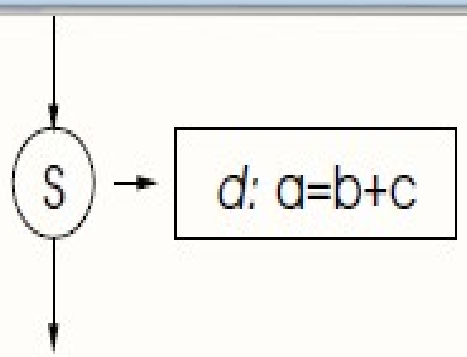
Then, the data-flow equations for S are:

$$gen[S] = \{d\}$$
$$kill[S] = D_{\mathtt{a}} - \{d\}$$
$$out[S] = gen[S] \cup (in[S] - kill[S])$$

where $D_{\mathtt{a}}$ = all definitions of **a** in the region of code

Data flow equations for Reaching definitions (RD)

Figure : Data-flow equations for an assignment statement.

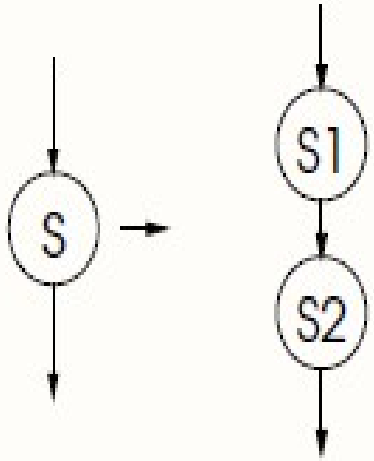


$gen[S]=\{d\}$

$kill[S]=D_a-\{d\}$

$out[S]=gen[S]\cup(in[S]-kill[S])$

Figure : Data-flow equations for sequence of statements.



$gen[S]=gen[S2]\cup(gen[S1]-kill[S2])$

$kill[S]=kill[S2]\cup(kill[S1]-gen[S2])$

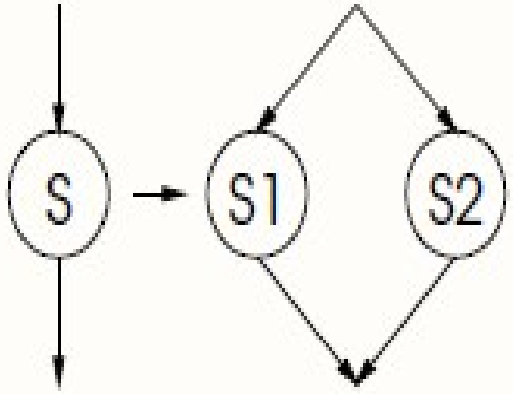
$in[S1]=in[S]$

$in[S2]=out[S1]$

$out[S]=out[S2]$

Contd...

Figure : Data-flow equations for an alternative



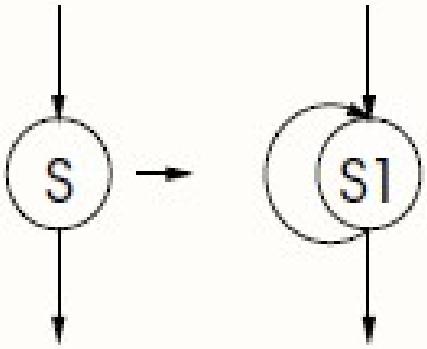
$gen[S]=gen[S1] \cup gen[S2]$

$kill[S]=kill[S1] \cap kill[S2]$

$in[S1]=in[S2]=in[S]$

$out[S]=out[S1] \cup out[S2]$

Figure : Data-flow equations for while loop.



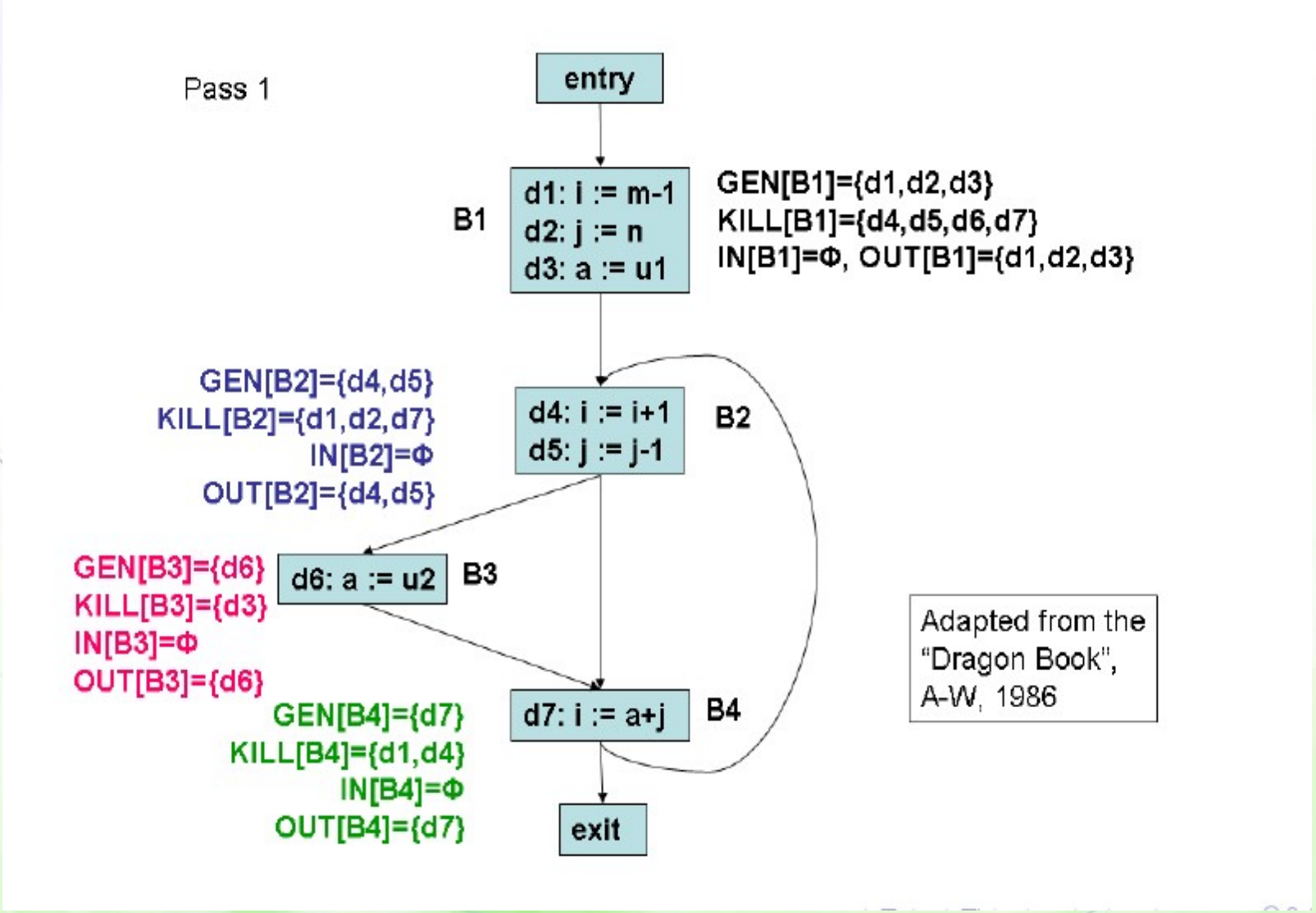
$gen[S]=gen[S1]$

$kill[S]=kill[S1]$

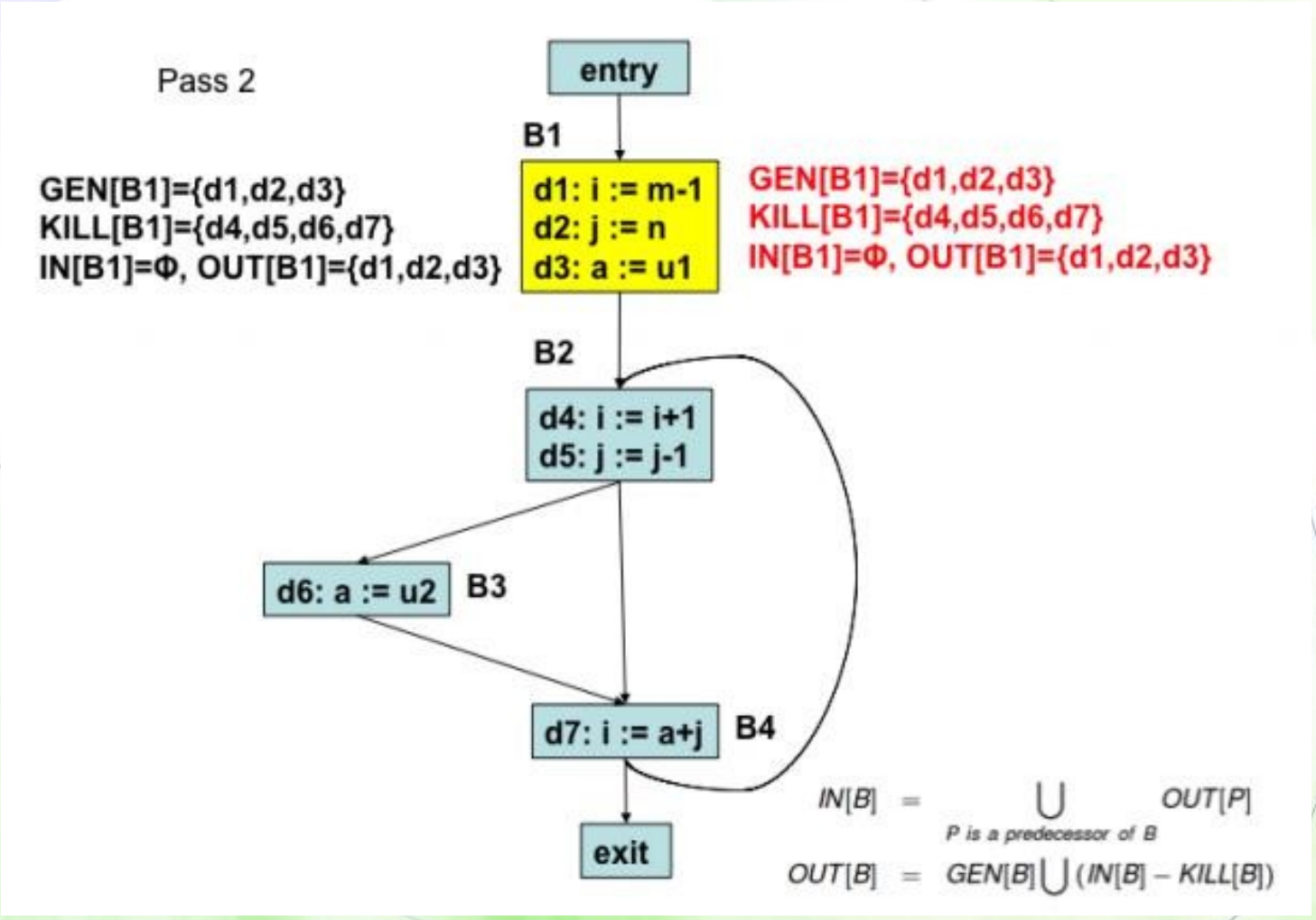
$in[S1]=in[S] \cup gen[S1]$

$out[S]=out[S1]$

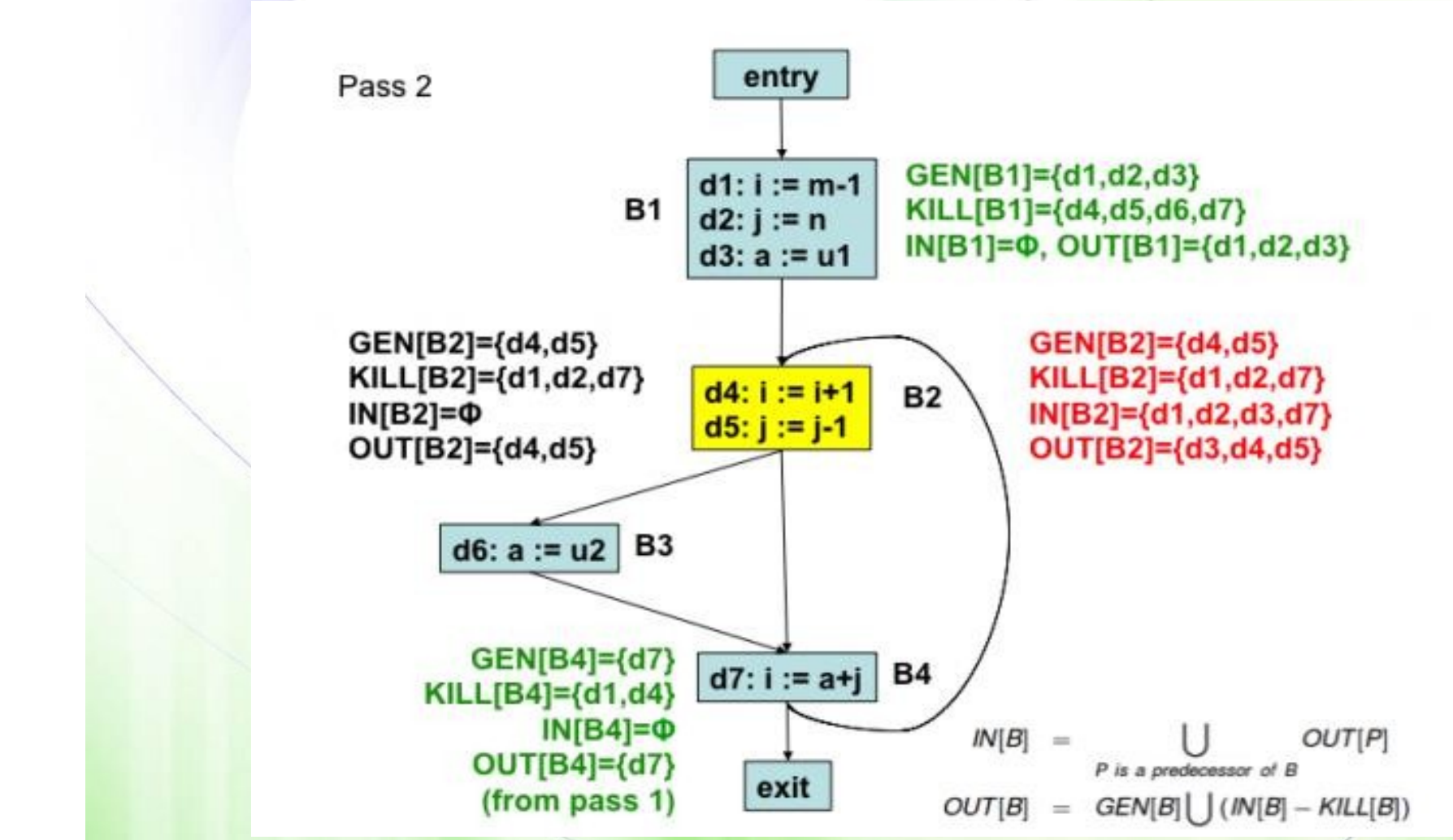
Reaching Definition analysis - Pass 1



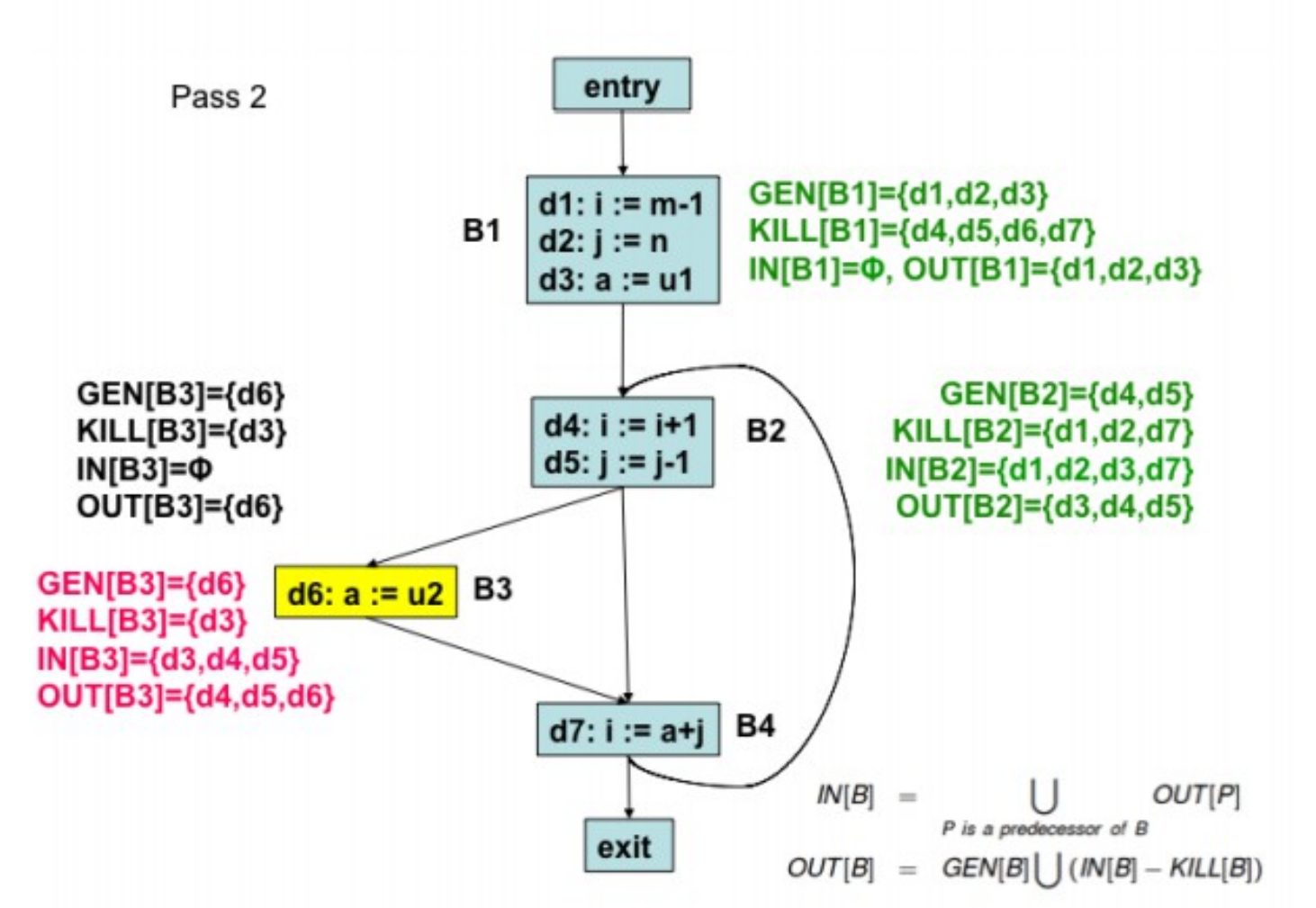
Reaching Definitions Analysis Pass 2.1



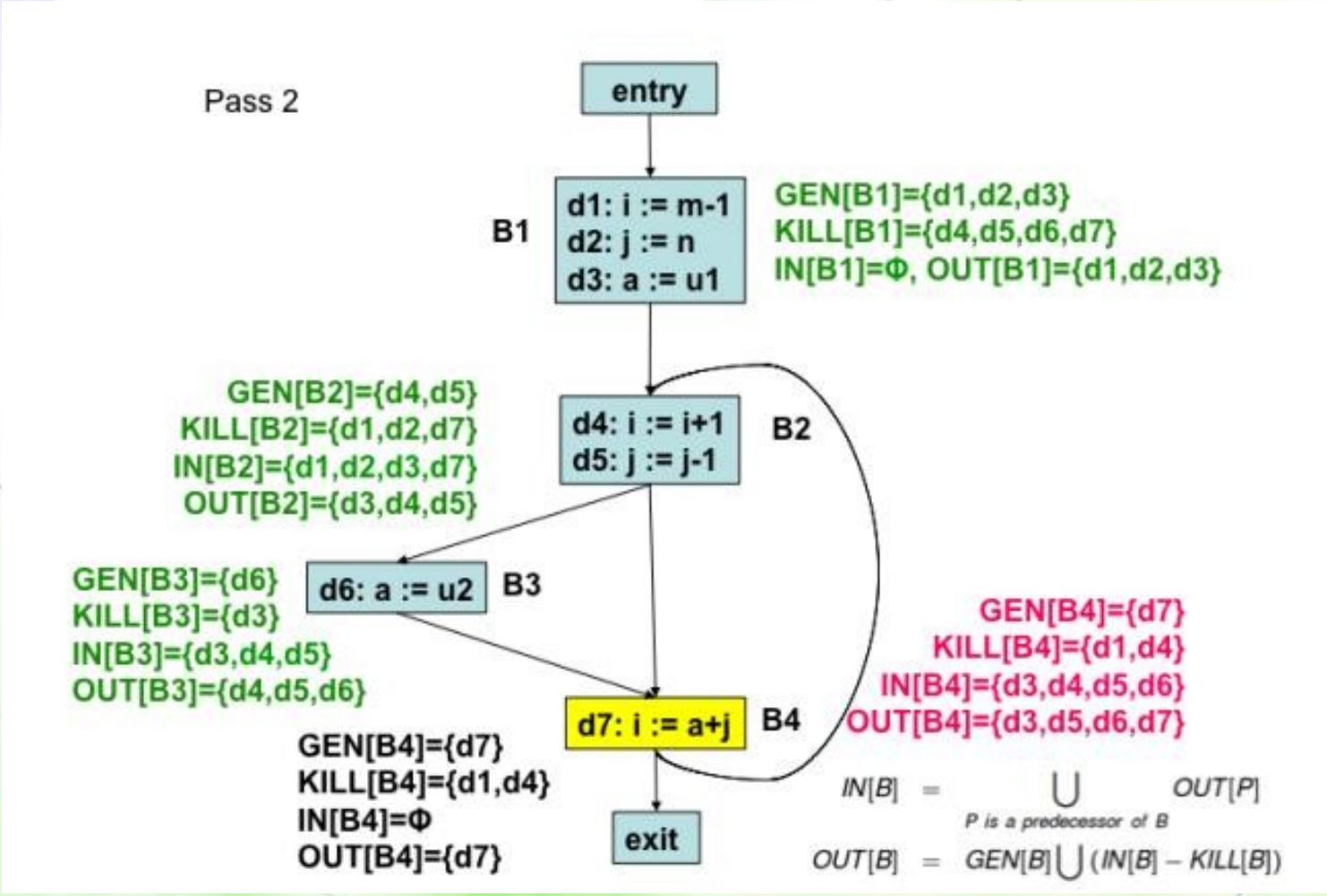
Reaching Definitions Analysis Pass 2.2



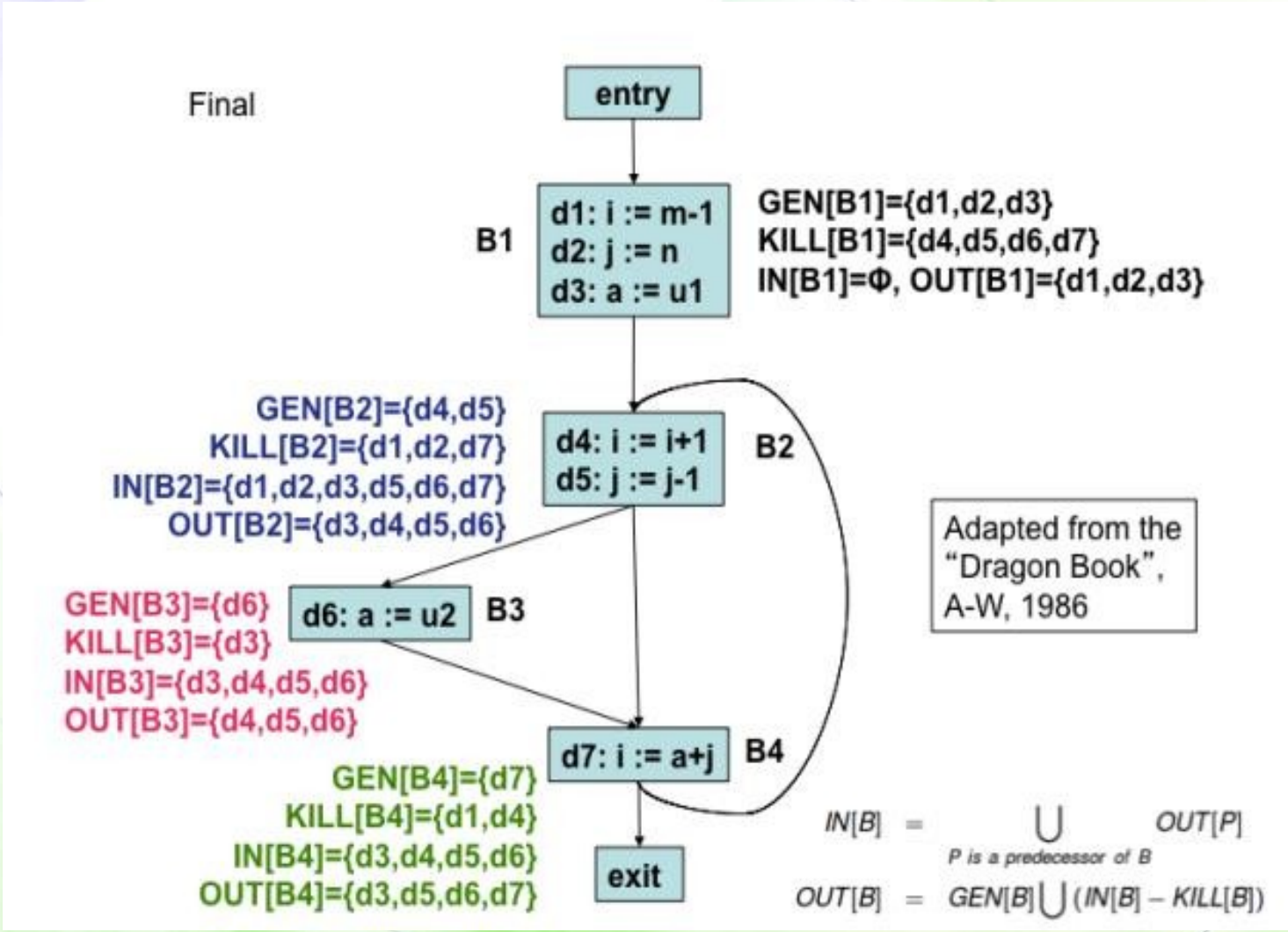
Reaching Definitions Analysis Pass 2.3



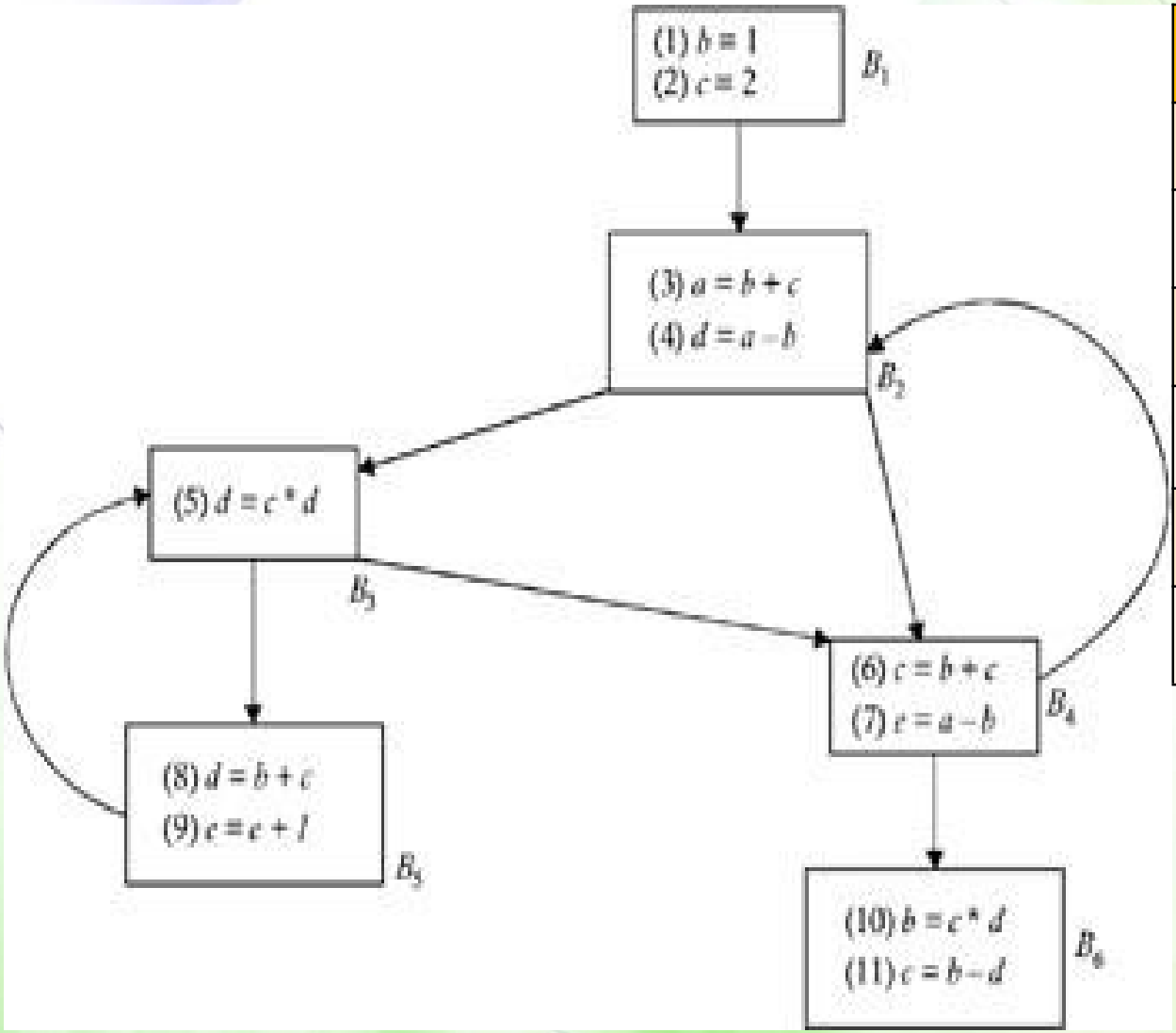
Reaching Definitions Analysis Pass 2.4



Reaching Definition analysis - Final



2nd Example



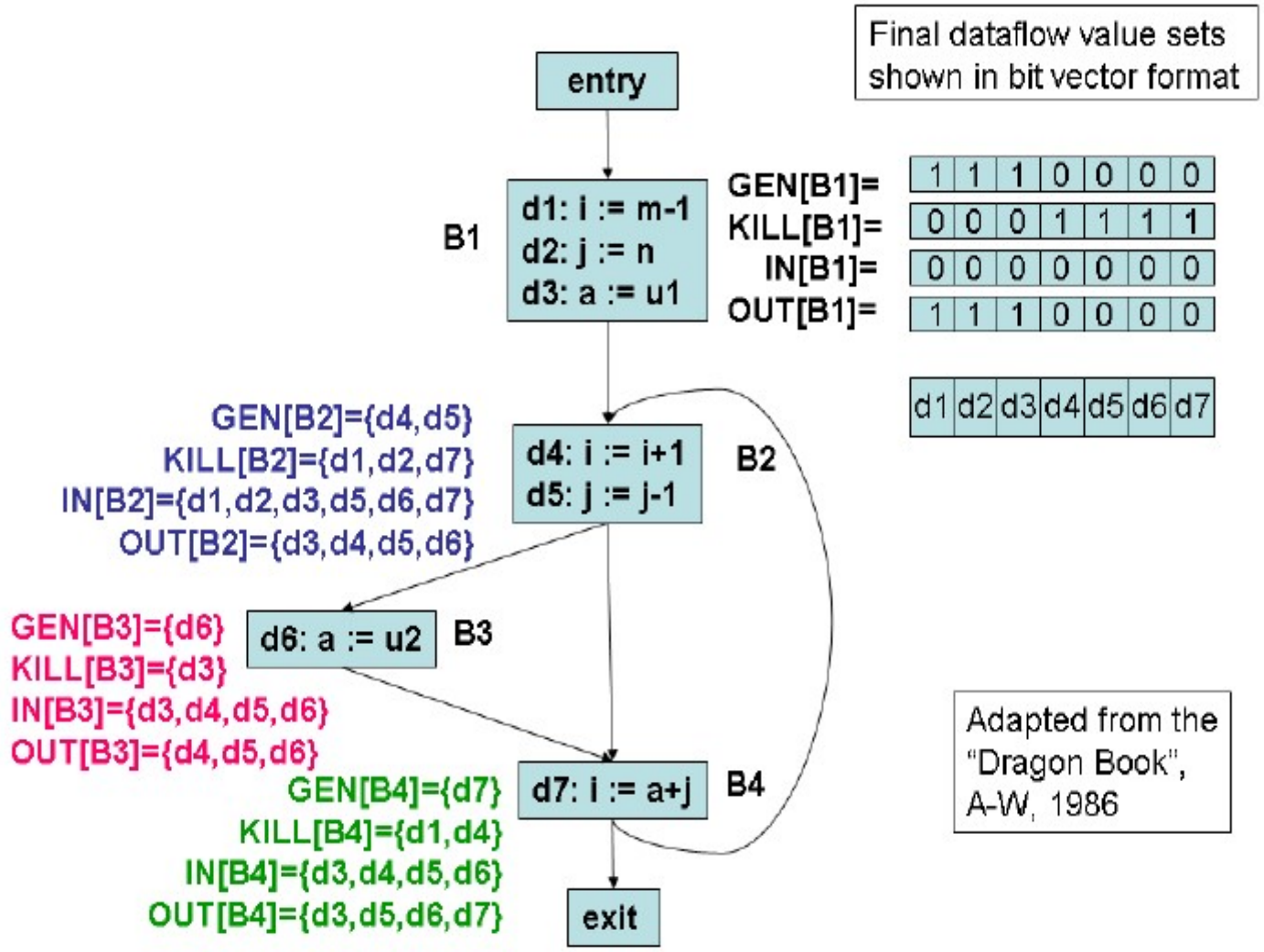
Block	GEN	KILL
*B*1	{1,2}	{6,10,11}
*B*2	{3,4}	{5,8}
*B*3	{5}	{4,8}
*B*4	{6,7}	{2,9,11}
*B*5	{8,9}	{4,5,7}
*B*6	{10,11}	{1,2,6}

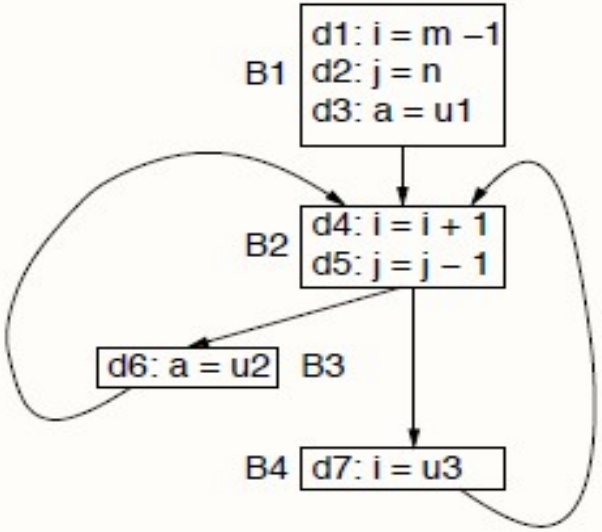
Algorithm for Reaching Definitions

```
for each block B do
{
   IN[B] = ∅ ;
   OUT[B] = GEN[B];
}
change = true;
while change do
{
    change = false;
    for each block B do
    {
        IN[B] = ∪ OUT[P] ;        P is a predecessor of B
        oldout = OUT[B];
        OUT[B] = GEN[B] U (IN[B] - KILL[B]);
        if (OUT[B] ≠ oldout) then
            change = true;
      }
```

```
}
```

Reaching Definitions: Bit Vector Representation





gen[B1] = {d1,d2,d3}
kill[B1] = {d4,d5,d6,d7}

gen[B2] = {d4,d5}
kill[B2] = {d1,d2,d7}

gen[B3] = {d6}
kill[B3] = {d3}

gen[B4] = {d7}
kill[B4] = {d1,d4}

Block B	Initial		Pass 1		Pass 2	
	in[*B*]	*out*[*B*]	*in*[*B*]	*out*[*B*]	*in*[*B*]	*out*[*B*]
B1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

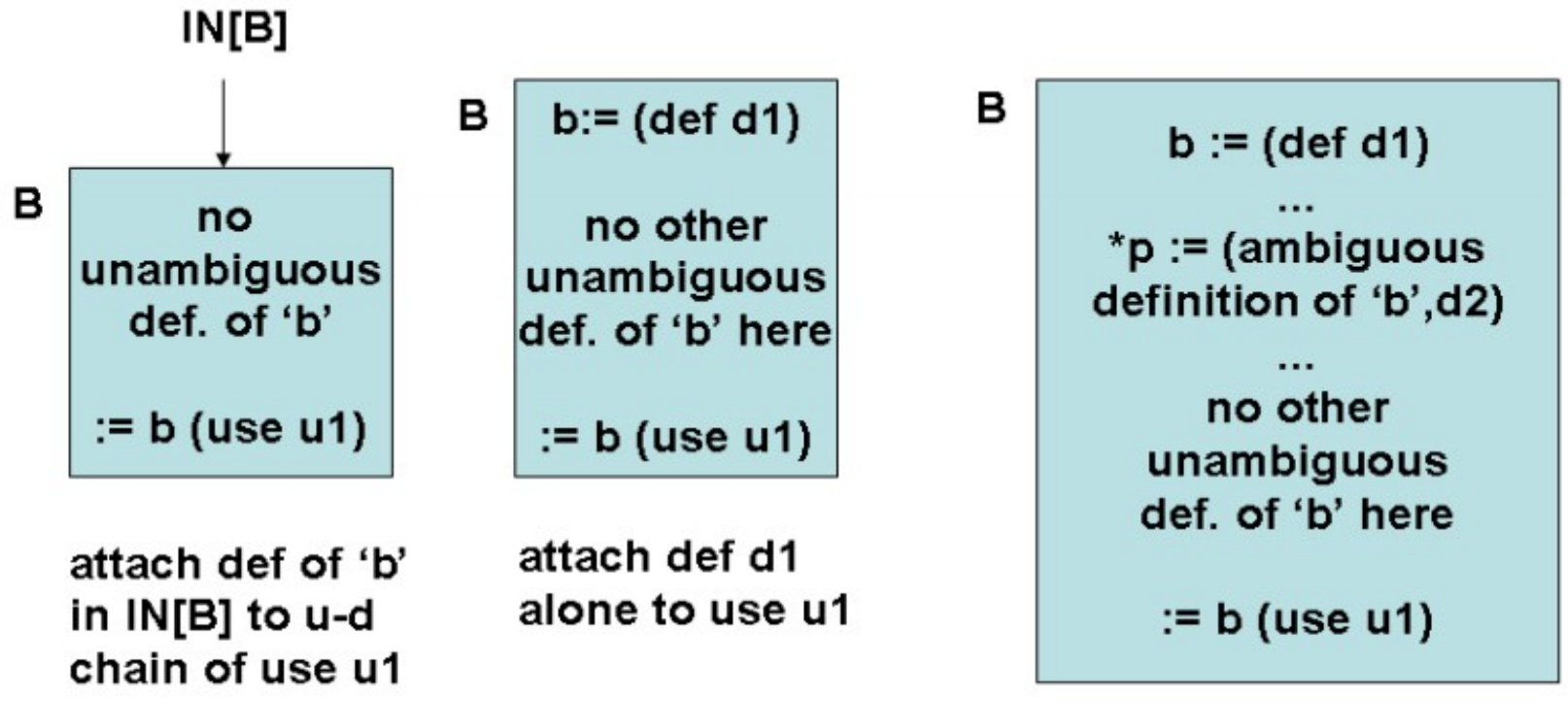
Uses of reaching definitions

- **Constant Propagation**
  - Whether different definitions for the variable reach the same constant or not.
- **Copy Propagation**
  - whether a particular copy statement reaches a use.
- **Code Motion**
  - Check whether a computation is loop-invariant

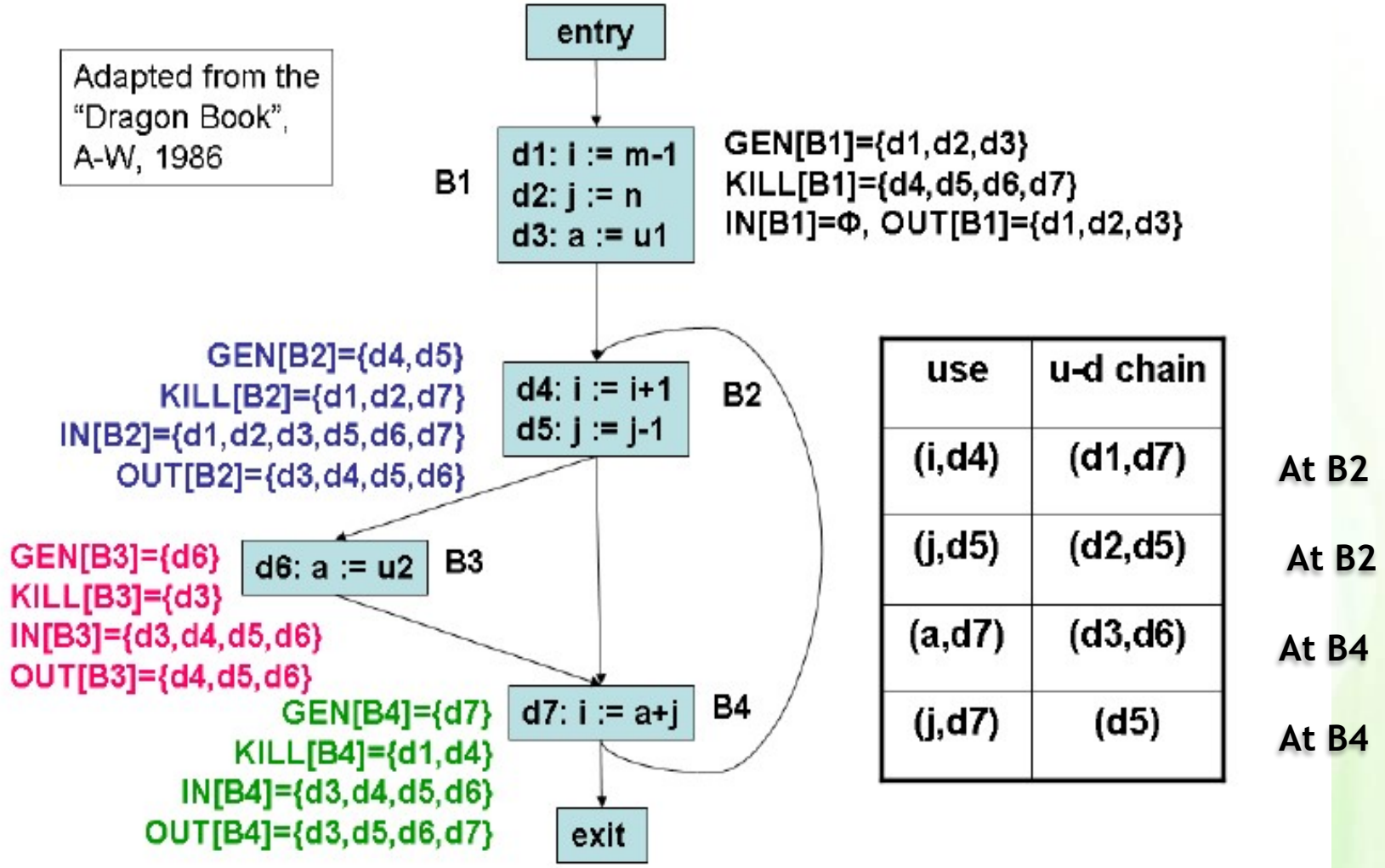
Use-Definition Chains (u-d chains)

- Reaching definitions may be stored as u-d chains for convenience.
- A u-d chain is a list of a use of a variable and all the definitions that reach that use. **(Important 2 mark Question)**
- u-d chains may be constructed once reaching definitions are computed
  - case 1: If use u1 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 u1 of b
  - case 2: If any unambiguous definition of b preceeds a use of b, then only that definition is on the u-d chain of that use of b
  - case 3: 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 Chain Construction

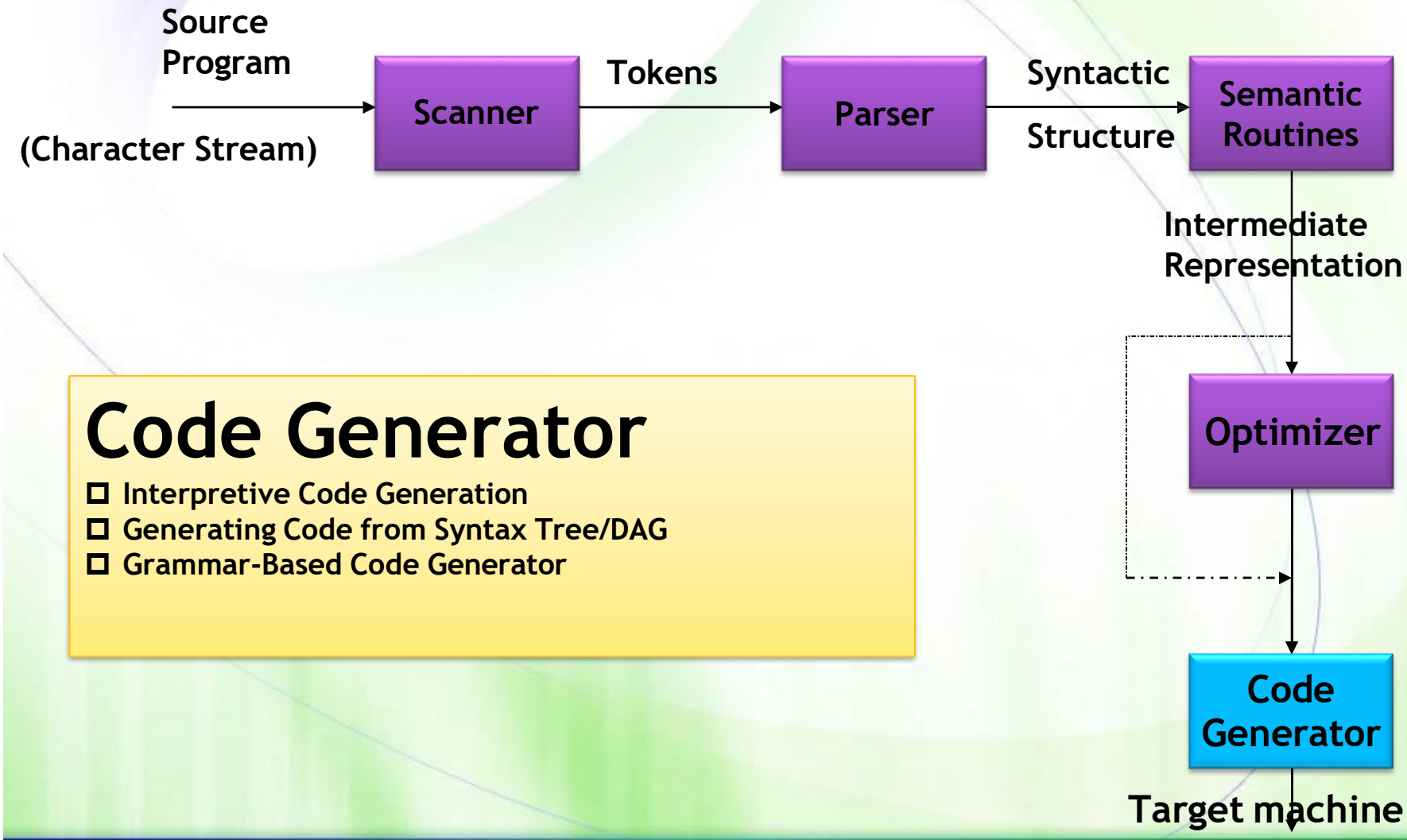


Three cases while constructing u-d chains from the reaching definitions


Adapted from the "Dragon Book", A-W, 1986
entry
B1
d1: i := m-1
d2: j := n
d3: a := u1
GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}
GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d5,d6,d7}
OUT[B2]={d3,d4,d5,d6}
d4: i := i+1
d5: j := j-1
B2
GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5,d6}
OUT[B3]={d4,d5,d6}
d6: a := u2
B3
GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]={d3,d4,d5,d6}
OUT[B4]={d3,d5,d6,d7}
d7: i := a+j
B4
exit
use
u-d chain
(i,d4)
(d1,d7)
At B2
(j,d5)
(d2,d5)
At B2
(a,d7)
(d3,d6)
At B4
(j,d7)
(d5)
At B4

Code Generation

The Structure of a Compiler



Code Generator

- Interpretive Code Generation
- Generating Code from Syntax Tree/DAG
- Grammar-Based Code Generator

Issues in the design of Code Generator University 16 mark Question

Issues arises during code generation phase are

(i) Input to the code generator (Intermediate Code)
(ii) Target Program
(iii) Memory Management
(iv) Instruction Selection
(v) Register allocation
(vi) Evaluation Order

Input to the code generator

- Input may of several forms
  - Linear representation – Postfix notation
  - Three address representation – Quadraples
  - Virtual Machine representation – stack machine codes, byte codes
  - Graphical representation – Syntax tree's, DAGs

Target Programs

- The output of the code generator is the target program that may be of several forms.
  - Absolute Machine code
    - Has fixed memory location & can be executed fastly
    - Example: PL/C and WATFIV
  - Relocatable Machine code
    - Allows subprograms to be compiled separately
    - All relocatable codes are linked together and loaded – **linkers and loaders**
    - Very much expensive
  - Assembly Language
    - Makes the execution faster
    - More preferable

Memory Management

Functions:

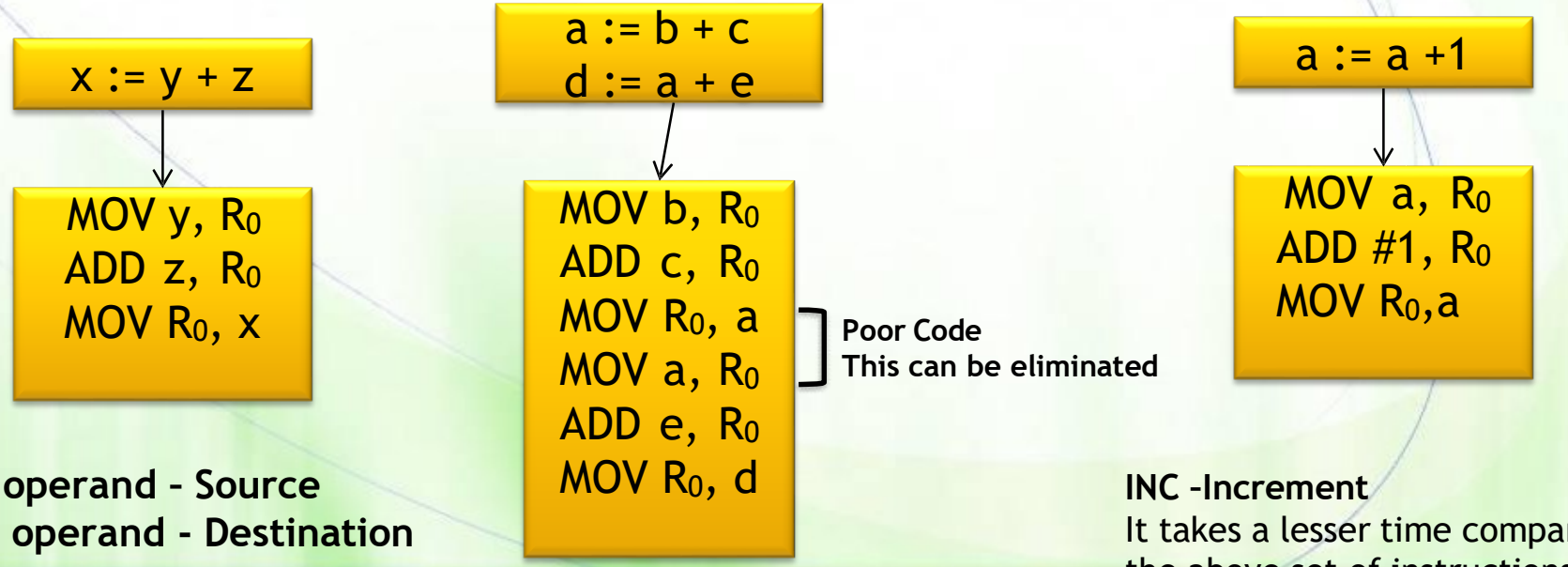
- Names in the symbol table is mapped with the correct addresses
- In symbol table, the information's like type, width, amount of storage needed is added
- Example:
  - j: goto i

  - If (i<j) → backward jump, generate a jump with the target address
  - If (i>j) → forward jump, store the location on quadraple I's list. When we process quadraple I, all the instructions are filled using backpatching.

Instruction Selection

Instructions are selected in such a way that it should reduce the cost and maximize the speed
maximize the speed

- Important factors to be considered
  - Uniformity
  - Completeness
  - Cost for evaluating the Instruction
  - Size of the Instruction

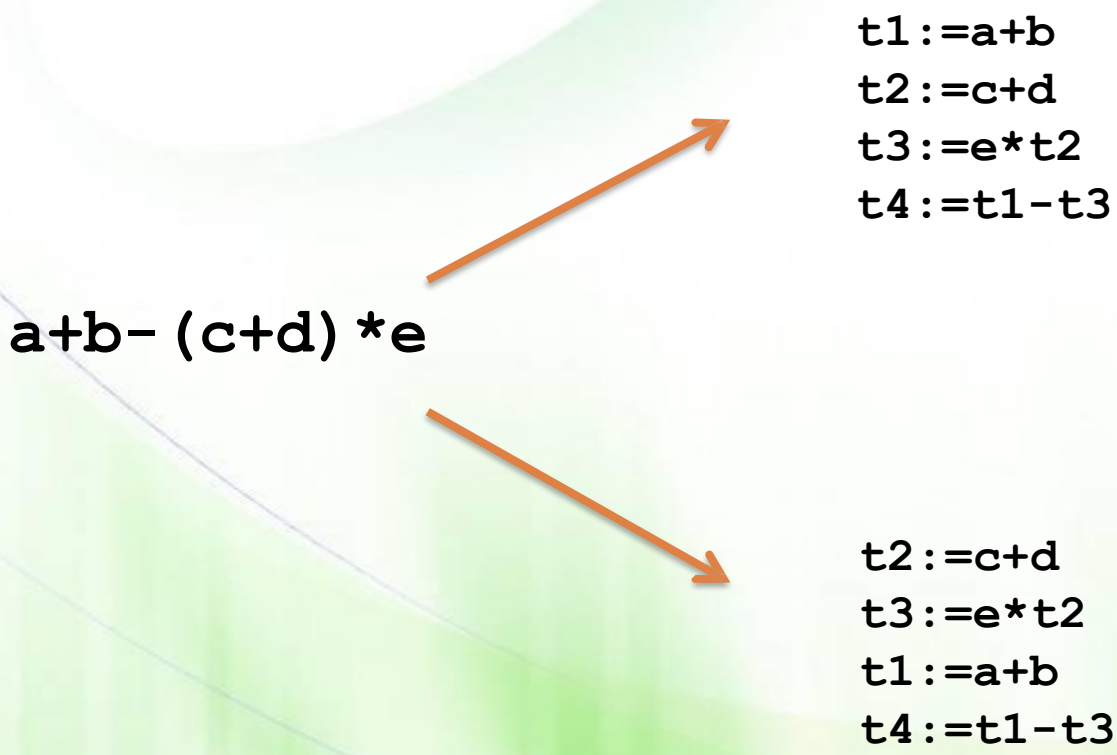


Register allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Two subproblems
  - Register allocation: select the set of variables that will reside in registers at each point in the program
  - Register assignment: select specific register that a variable reside in.

Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.



Target machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set

- ***Our (hypothetical) machine:***
  - Byte-addressable (word = 4 bytes)
  - Has *n* general purpose registers **R0**, **R1**, …, **R***n*-1
  - Two-address instructions of the form

op source, destination

Opcodes

MOV (move content of source to destination)
ADD (add content of source to destination)
SUB (subtract content of source from destination)

Data Fields

The Target Machine – Contd

Addressing modes

- **Addressing mode**: Different ways in which location of an operand can be specified in the instruction.

Mode	Form	Address	Added Cost
Absolute	**M**	**M**	1
Register	**R**	**R**	0
Indexed	*c*(**R**)	*c*+*contents*(**R**)	1
Indirect register	***R**	*contents*(**R**)	0
Indirect indexed	****c*(**R**)	*contents*(*c*+*contents*(**R**))	1
Literal	**#***c*	N/A	1

Instruction Costs

Instruction Cost = 1 + Source Cost + Destination Cost

Source and Destination having Registers	Cost = 0
Source and Destination having Memory Locations	Cost = 1
Source and Destination having Literals	Cost = 1

Examples

MOV R0, R1

Cost = 1+ source cost + Destination cost

= 1+ 0 + 0

= 1

MOV R0, M

Cost = 1+ source cost + Destination cost

= 1+ 0 + 1

= 2

ADD #1, R0

Cost = 1+ source cost + Destination cost

= 1+ 1 + 0

= 2

Examples - Contd

- Suppose we translate **a:=b+c** into

```
MOV b,R0        Cost : 1+1+0 = 2
ADD c,R0        Cost : 1+1+0 = 2
MOV R0,a        Cost : 1+0+1 = 2
```

- Assuming addresses of **a**, **b**, and **c** are stored in **R0, R1, R2**

```
MOV *R1,*R0     Cost : 1+0+0 = 1
ADD *R2,*R0     Cost : 1+0+0 = 1
```

- Assuming **R1** and **R2** contain values of **b** and **c**

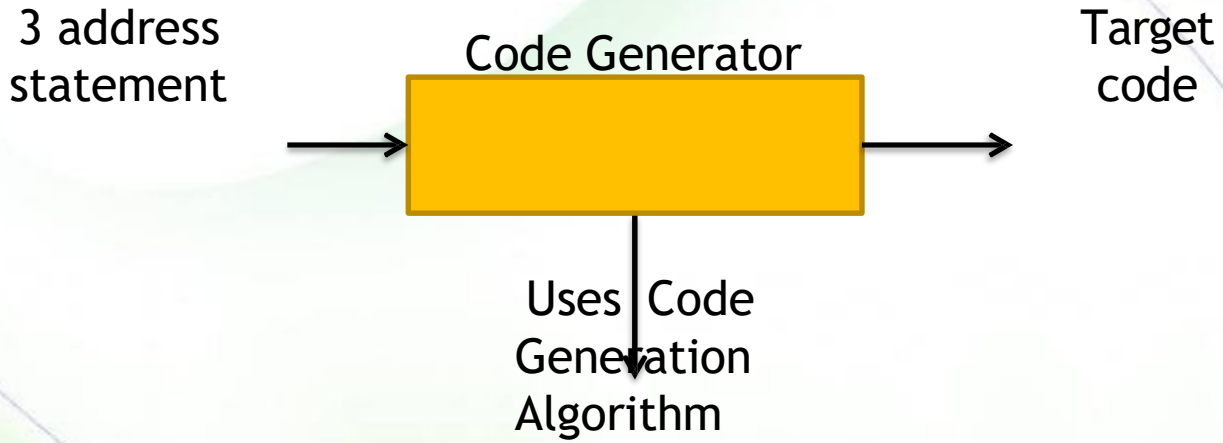
```
ADD R2,R1       Cost : 1+0+0 = 1
MOV R1,a        Cost : 1+0+1 = 2
```

- Assuming **b** and **c**

```
MOV b,a         Cost : 1+1+1 = 3
MOV c,a         Cost : 1+1+1 = 3
```

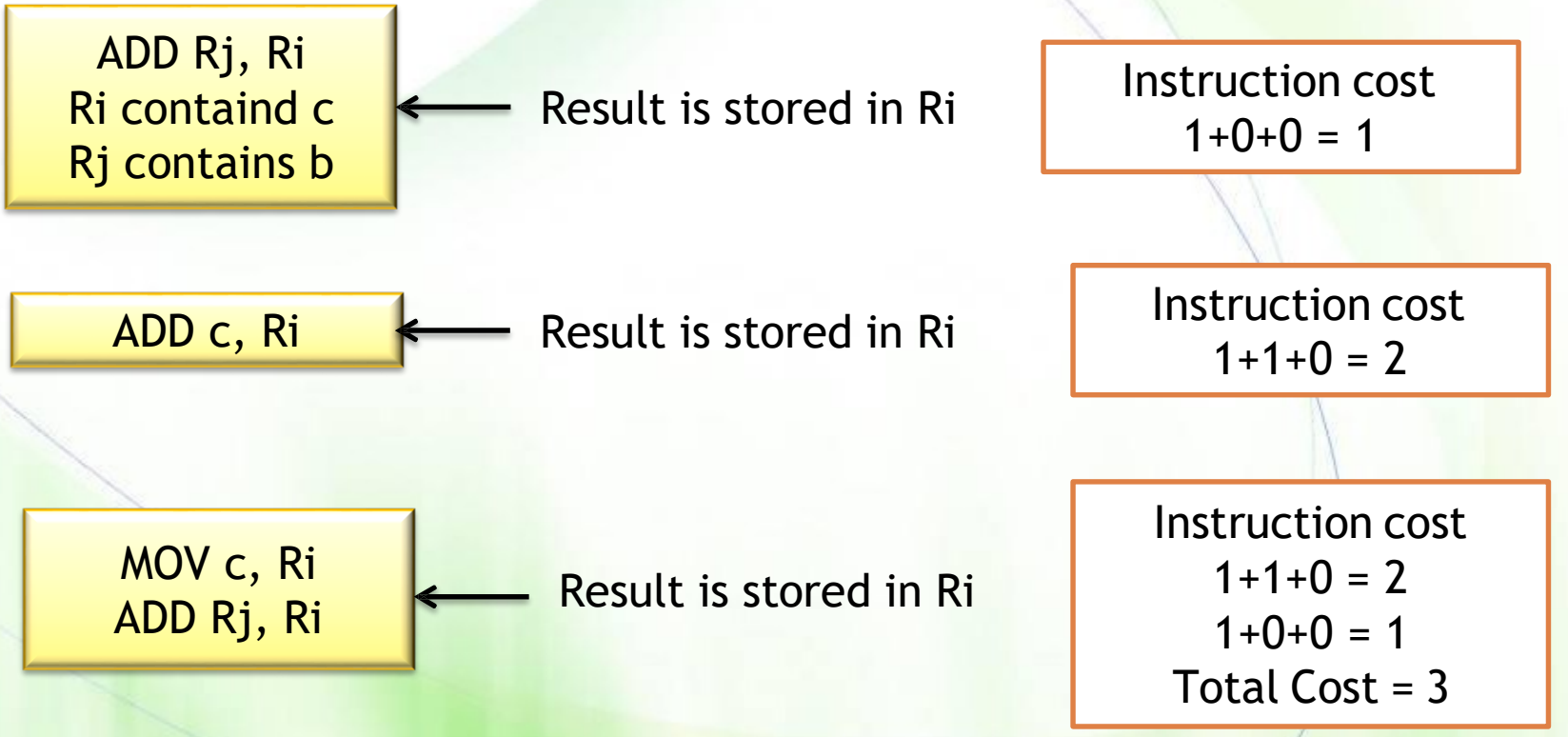
Sample Code Generator

- Code generator – used for the conversion of three address statement to target program.



Contd...

a := b + c


ADD Rj, Ri
Ri containd c
Rj contains b
Result is stored in Ri
Instruction cost
1+0+0 = 1
ADD c, Ri
Result is stored in Ri
Instruction cost
1+1+0 = 2
MOV c, Ri
ADD Rj, Ri
Result is stored in Ri
Instruction cost
1+1+0 = 2
1+0+0 = 1
Total Cost = 3

Next use information

- for register and temporary allocation
- remove variables from registers if not used
- statement X = Y op Z
  defines X and uses Y and Z
- scan each basic blocks backwards
- assume all temporaries are dead on exit and all user variables are live on exit

Algorithm to compute next use information

- Suppose we are scanning

  i : X := Y op Z in backward scan

  - attach to i, information in symbol table about X, Y, Z
  - set X to not live and no next use in symbol table
  - set Y and Z to be live and next use in i in symbol table

Example

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

Example

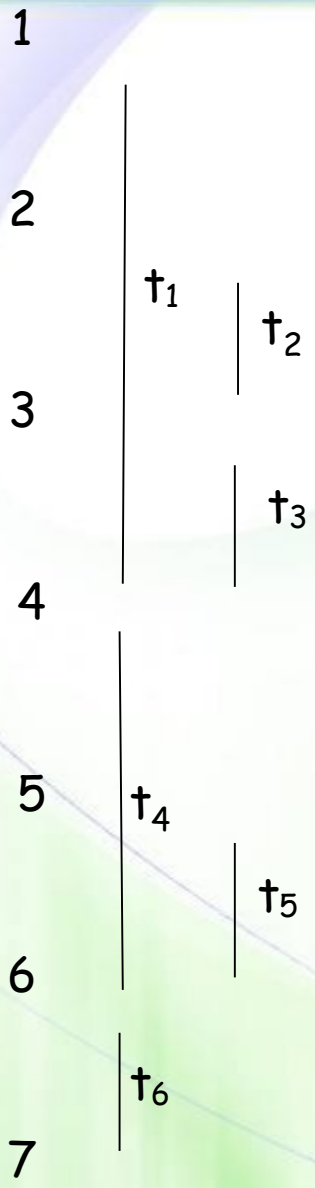
STATEMENT

7: no temporary is live
6: t_6:use(7), t_4 t_5 not live
5: t_5:use(6)
4: t_4:use(6), t_1 t_3 not live
3: t_3:use(4), t_2 not live 2: t_2:use(3)
1: t_1:use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

Example ...



1: $t_1 = a * a$

2: $t_2 = a * b$ 3:

$t_2 = 2 * t_2$

4: $t_1 = t_1 + t_2$

5: $t_2 = b * b$

6: $t_1 = t_1 + t_2$

7: $X = t_1$

Descriptors

- **Address Descriptor**
  - Keeps track of where the variables are currently now.

a	R0
b	MEM
x	MEM,R2
y	R3
t1	R3
t2	R4,R31
t3	<nowhere>
⋮	⋮

- **Register Descriptor**
  - Keeps track of what is currently in each register

R0	a
R1	EMPTY
R2	x
R3	y,t1
⋮	⋮
R31	t2

Code Generation Algorithm

- Input: A basic block (sequence of three address statements)
- For each three-address statement x := y op z we:
  - Invoke function *getreg* [description next] to determine the location L where the result of the computation *y op z* should be stored
  - Consult the address descriptor for y to determine y', (one of) the location(s) of y
    - Prefer register for y' is y is stored both in memory and register
    - If value of y not already in L, generate instruction MOV y',L to place a copy of y in L

Contd…

- Generate the instruction OP z', L where z' is a current location of z.
  - Prefer register to memory location if z is in both
  - Update address descriptor of x to indicate that x is in location L
  - If L is a register, update its descriptor to indicate that it contains the value of x , and remove x from all other register descriptors.
- If current values of y and/or z
  - have no next uses,
  - are not live on exit from the block,
  - and are in registers
  - => alter register descriptor to indicate that after execution of x:=y op z, those registers will no longer contain y and/or z.

The function getreg()

Returns location L to hold the value of x for the assignment x := y op z

1. If the name y is in a register
   - that holds the value of no other names
   - and y is not live
   - and y has no next use after execution of x:=y op z
   - => return the register of y for L
   - => update address descriptor for y to indicate that y is no longer in L
2. Failing (1), return an empty register for L if there is one
3. Failing (2), … (cont.)

Contd...

3. .. If x has a next use within the block, or op is an operator that requires a register (e.g. indexing), then
   - Find an occupied register R
   - Store R's contents to a memory location (MOV R, M), if not already
   - there
   - Update address descriptor for M
   - Return R
   - Which R to pick?
     - One whose value is already in memory
     - One whose data is referenced further in the future
4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L

Code Generation Examples

Assignment statements

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

Three address statement:

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

With ‘d’ live at end.

Assignment statements contd...

Statement Sequence	Generated Code	Instruction Cost	Register Descriptor	Address Descriptor
t := a-b	MOV a, R0 SUB b, RO	1+1+0 = 2 1+1+0 = 2	Initially all registers are empty Now R0 contains 't'	't' is in R0
u := a-c	MOV a, R1 SUB c, R1	1+1+0 = 2 1+1+0 = 2	R0 contains 't' R0 contains 'u'	't' is in R0 'u' is in R1
v := t+u	ADD R1, R0	1+0+0 = 1	R0 contains 'v' R1 contains 'u'	'u' is in R1 'v' is in R0
d := u+v	ADD R1,R0 MOV R0,d	1+0+0 = 1 1+1+0 = 2	R0 contains 'd'	'd' is in R0 and also in memory
		TOTAL COST = 12		

Contd...

the above cost can be reduced as follows

MOV a, R0	-	2
MOV R0,R1	-	1
SUB b, RO	-	2
SUB c, R1	-	2
ADD R1, R0	-	1
ADD R1,R0	-	1
MOV R0,d	-	2
TOTAL COST	-	11

Code Generation Examples

Indexed statements

Statement Sequence	i in Register Ri		i in Memory Mi		i in stack	
	Code	Cost	Code	Cost	Code	Cost
a := b[i]	MOV b(Ri), R	2	MOV Mi, R MOV b(R), R1	2 2	MOV Si(A), R MOV b(R), R1	2 2
a[i] := b	MOV b,a(Ri)	3	MOV Mi, R MOV b, a(R)	2 3	MOV Si(A), R MOV b, a(R)	2 3

Code Generation Examples

Pointer statements

Statement Sequence	p in Register Rp		p in Memory Mp		p in stack	
	Code	Cost	Code	Cost	Code	Cost
a := *p	MOV *Rp, a	2	MOV Mp, R MOV *R,R	3	MOV Sp(A), R MOV *R, R	2 2
*p:= b	MOV a, *Rp	2	MOV Mp, R MOV a, *R	2 2	MOV a,R MOV R, *Sp(A)	2 2

Directed Acyclic Graph - DAG

- Directed acyclic graph(DAG) is a useful data structure for implementing Transformations on basic blocks.

- Steps to construct DAG
  - Leaves are labeled by unique identifiers
  - Interior nodes – operators

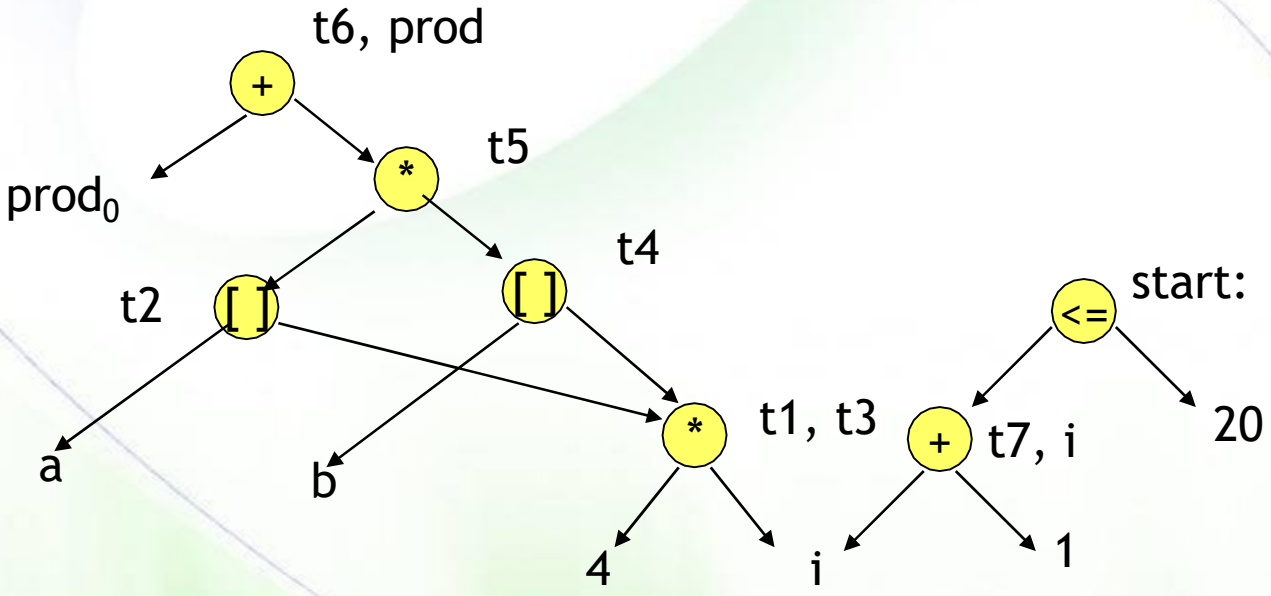
- **AIM**

basic block => dag => improved block => assembly

Three-address code for Block 2

```
[1]    prod := 0
[2]    i := 1
[1]    t1 := 4 * i
[2]    t2 := a[t1]
[3]    t3 := 4 * i
[4]    t4 := b[t3]
[5]    t5 := t2*  t4
[6]    t6 := prod + t5
[7]    prod := t6
[8]    t7 := i + 1
[9]    i := t7
[10]  if i <= 20 goto [1]
```

Corresponding DAG



leaves – identifiers and constants
interior nodes – operators
t1 – temporary variables

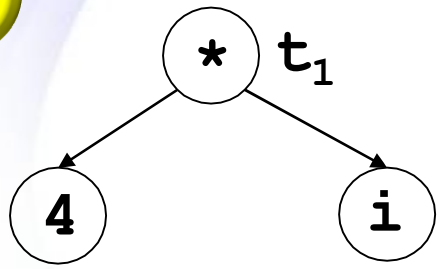
Algorithm: Construction of the DAG

- Input: Basic Block.
- Output: DAG for the basic block containing:
- a label for each node.
  - For leaves an identifier - constants are permitted.
  - For interior nodes an operator symbol.
- for each node a list of attached identifiers to hold the computed values
  i. x:= y op z
  ii. x:= op y
  iii. x:= y
- Condition (i) : If y is undefined, then create a leaf labeled y, and now node(y) be this node. If z is undefined, then create a leaf labeled z, and now node(z) be this node.
- Condition (ii) : for the case(i), create a node(op) whose left child is node(y) and right child is node(z). If it does not already exists create a node.
- Condition (iii) : Delete x from the list of attached nodes. Append x to the list of identifiers for node(n) and set node(x) to n.

Example: DAG construction from BB

$t_1 := 4 * i$

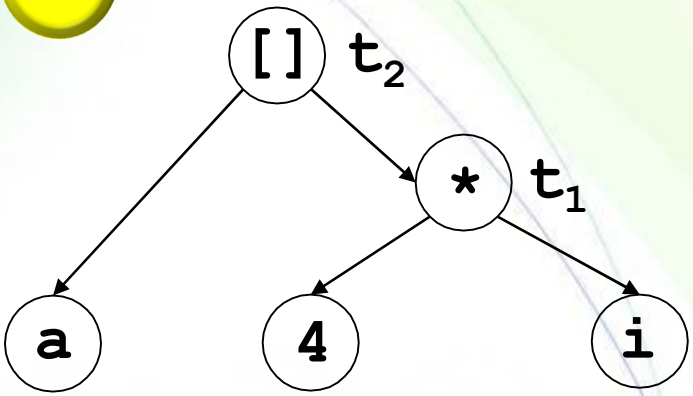
1



$t_1 := 4 * i$

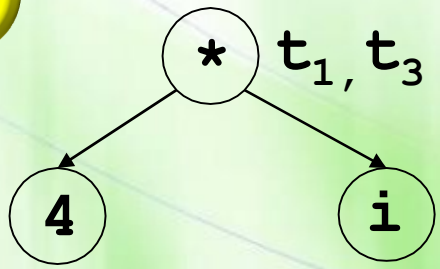
$t_2 := a[t_1]$

2



$t_3 := 4 * i$

3

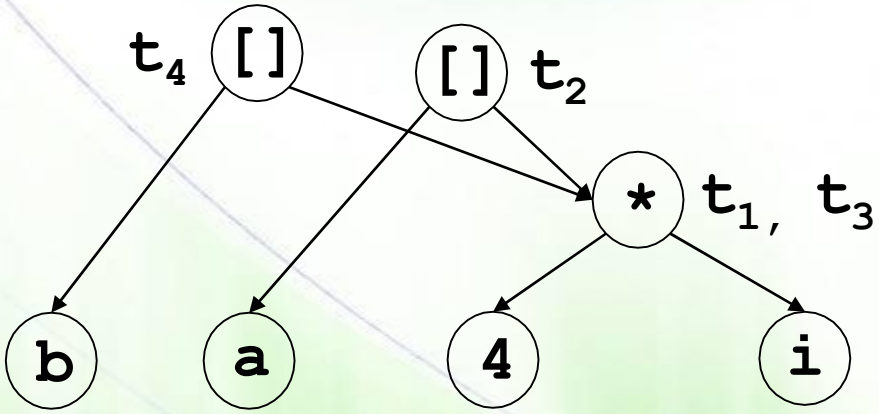


Node for 4*1 already exists, no need to create a new node. But attach to the list of identifiers

Example Contd...

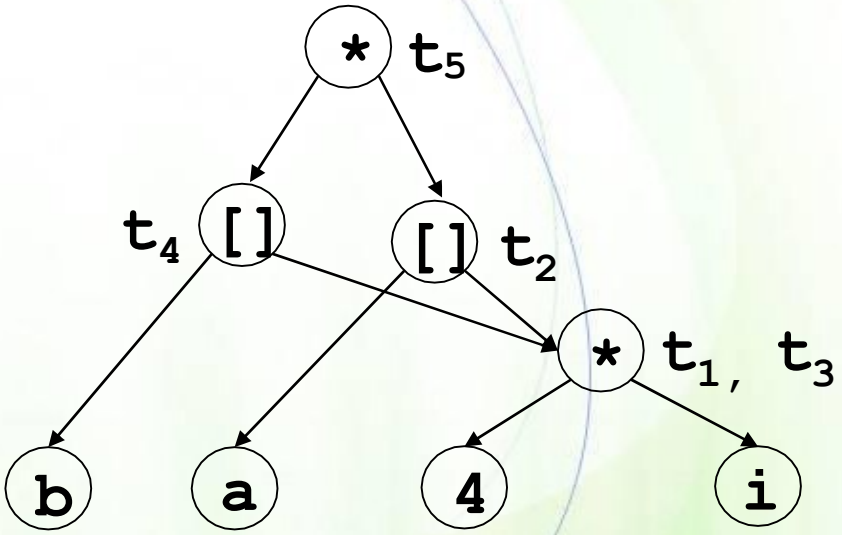
$t_1 := 4 * i$
$t_2 := a[t_1]$
$t_3 := 4 * i$

4 $t_4 := b[t_3]$



$t_1 := 4 * i$
$t_2 := a[t_1]$
$t_3 := 4 * i$
$t_4 := b[t_3]$

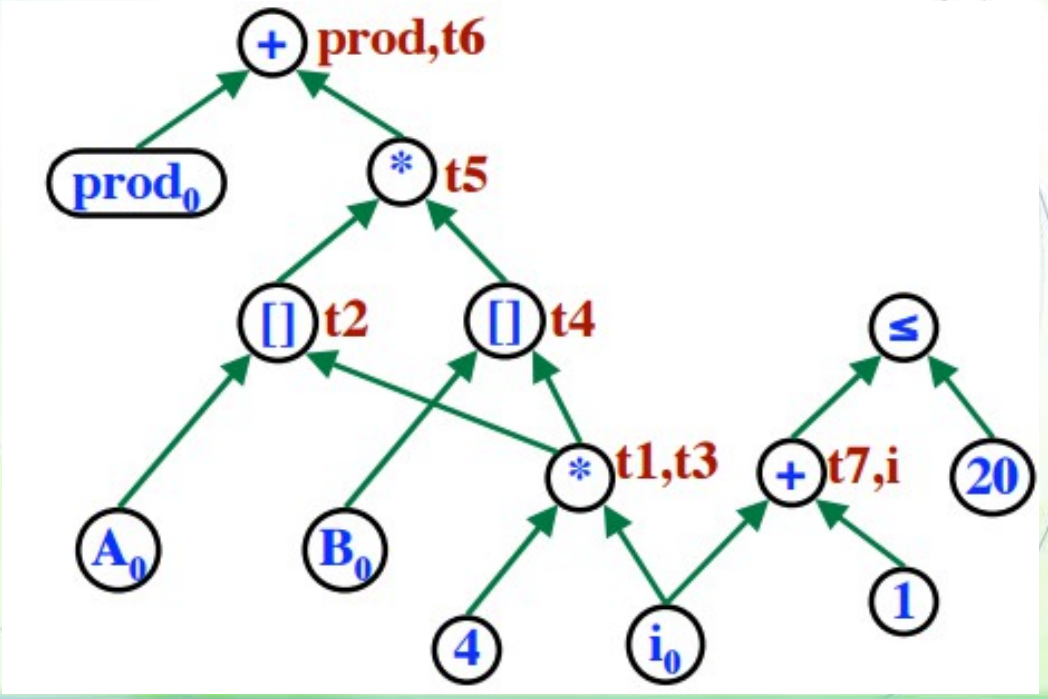
5 $t_5 := t_2 * t_4$



Example Contd...

6

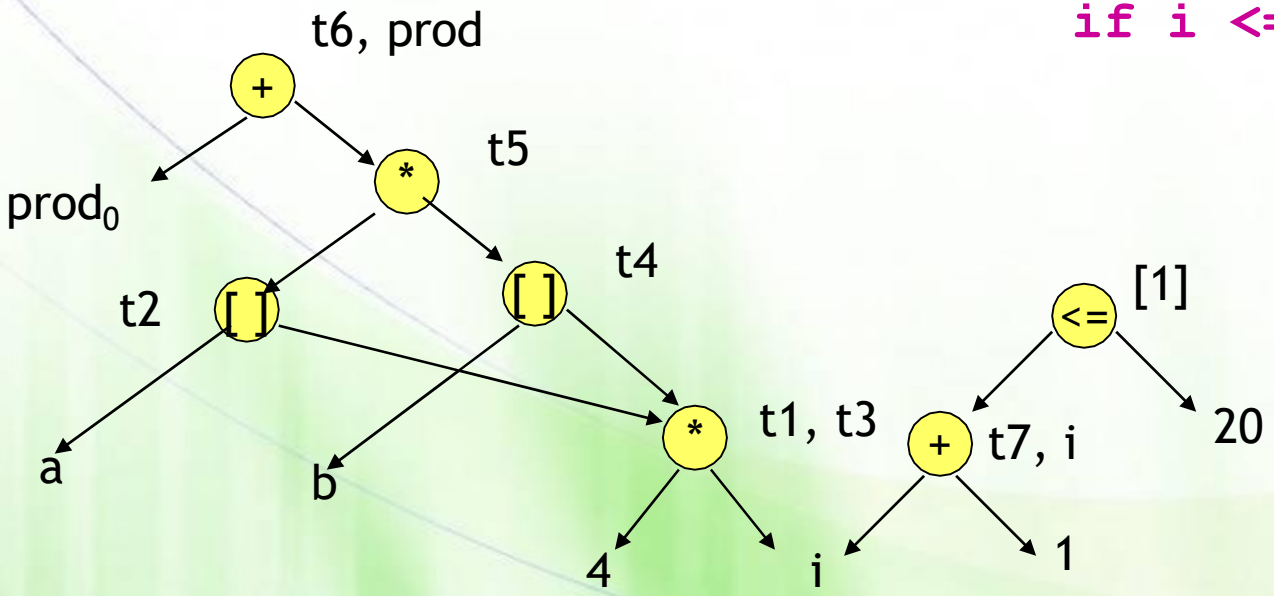
t_1 := 4 * i
t_2 := a[t_1]
t_3 := 4 * i
t_4 := b[t_3]
t_5 := t_2 * t_4
t_6 := Prod + t_5
Prod := t_6



Example Contd...

7

$t_1 := 4 * i$
$t_2 := a[t_1]$
$t_3 := 4 * i$
$t_4 := b[t_3]$
$t_5 := t_2 * t_4$
$t_6 := Prod + t_5$
$Prod := t_6$
$t_7 := i + 1$
$i := t_7$
if i <= 20 goto [1]



Reduced Code for DAG

```
[1]    t1 := 4 * i
[2]    2 := a[t1]
[3]    t4 := b[t1]
[4]    t5 := t4 * t2
[5]    Prod := Prod + t5
[6]    i := i + 1
[7]    if i <= 20 goto [1]
```

Why DAG mainly???

1. Automatically eliminate common sub expressions
2. Simplifying the list of quadruples by eliminating the common sub expressions
3. Determine which names are used inside and outside the blocks.
4. Determine which statements of the block, could have their values used outside the blocks

Peephole Optimization

Important 8 Marks

- **Peephole optimization** is a kind of optimization performed over a very small set of instructions in the generated codes.
- Peephole -- Window
- **Characteristics of peephole optimization**

1. Redundant instruction elimination.
2. Flow of control optimization.
3. Algebraic simplification.
4. Reduction in strength
5. Machine idioms

Contd…

1. **Redundant instruction elimination.**
   - The redundant loads and stores can be eliminated in this type of transformations.
   - Example :
     - MOV R0,x
     - MOV x,R0
   - Eliminate the second instruction.

2. **Flow of control optimization.**
   - Using peephole optimization unnecessary jumps on jumps can be eliminated
     - **Unreachable Code**
     - **Avoid Jumps**

Contd…

Unreachable code:

- unlabelled instruction following an unconditional jump may be eliminated

```
#define DEBUG 0
...
if (debug) {
/* print stmts */
}
```

```
        if debug = 1 goto L1
        goto L2
L1:     /* print */
L2:
```

Jumps

- Jumps to jumps
- Jumps to conditional jumps
- Conditional jumps to jumps

- Case 1: Jumps to jumps

```
        goto L1
        ...
L1:     goto L2
```



```
goto L2
```

If there is no jump to L1, then remove the statement
L1: goto L2

- Case 2: Jumps to conditional jumps

```
        if a < b goto L1
        ...
L1:     goto L2
```

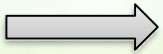


```
if a < b goto L2
```

- Case 3: Conditional jumps to jumps

```
        goto L1
        ...
L1:     if a < b goto L2

L2:
```



```
        if a < b goto L2

L2:
```

3. Algebraic simplification.

The statements such as

```
x := x + 0
x := x* 1
```

Can be eliminated by Peephole optimization

4. Reduction in strength

- Expensive instructions can be replaced by cheaper instructions .
  - For example, **y **2 is cheaper than y** * y.
- Similarly addition and subtraction is cheaper than multiplication and division.

5. Machine idioms

- The target instructions can be replaced by equivalent machine instructions.

```
x := x + 1
x := x - 1
```



```
x ++     (Auto increment)
x --    (Auto decrement)
```

UNIT - V completed