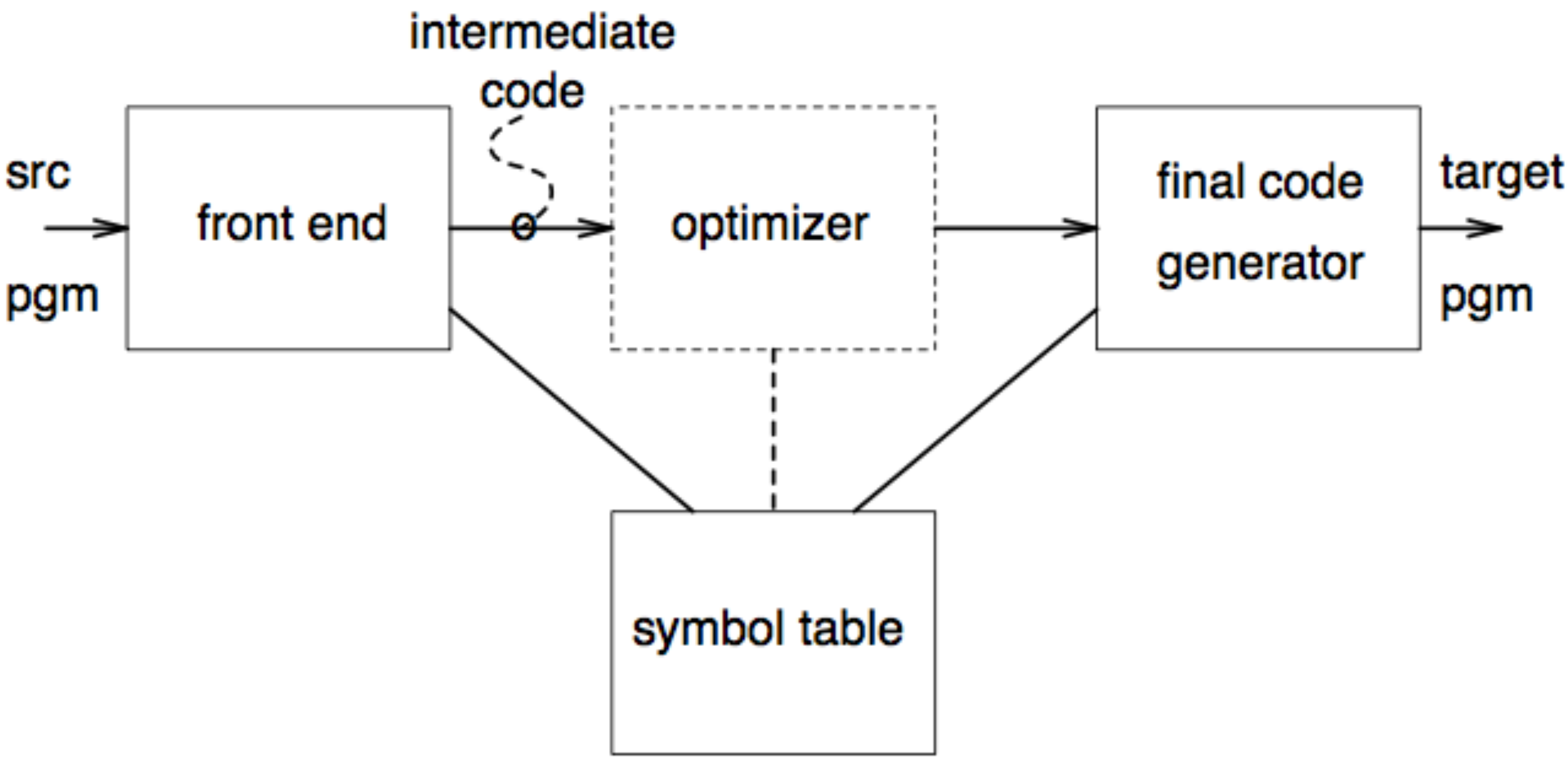
Final Code Generation and Code Optimization

Final Code Generation



Translating 3-address code to final code

3-Address Code	*MIPS assembly code*

`x = A[i]`	*load* `i` *into* reg_1
	`la` reg_2, `A`
	`add` reg_2, reg_2, reg_1
	`lw` reg_2, (reg_2)
	`sw` reg_2, `x`

`x = y+z`	*load* `y` *into* reg_1 *load* `z` *into* reg_2 `add` reg_3, reg_1, reg_2 `sw` reg_3, `x`
`if x >= y goto L`	*load* `x` *into* reg_1 *load* `y` *into* reg_2 `bge` reg_1, reg_2, `L`

Improving Code Quality : *Peephole Optimization*

- redundant instruction elimination, e.g.:

```
  ...                    ...
  goto L      ⇒      L:
L:                       ...
  ...
```

- flow-of-control optimizations, e.g.:

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

<u>**Improving Code Quality**</u> : *Peephole Optimization*

- algebraic simplifications, e.g.:
  - instructions of the form `x := x+0` or `x := x*1` can be eliminated.
  - special case expressions can be simplified, e.g.: `x := 2*y` can be simplified to `x := y+y`.

<u>Improving Code Quality</u> : *Code Optimization*

- Examine the program to find out about certain properties of interest ("Dataflow Analysis").

- Use this information to change the code in a way that improves performance. ("Code Optimization").

Improving Code Quality : *Code Optimization*

Code Motion out of Loops : if a computation inside a loop produces the same result for all iterations (e.g., computing the base address of a local array), it may be possible to move the computation outside the loop.

```
for ( i=0; i < N; i++) {
   base = &a[0];
   crt = *(base + i);
}
```

original code

```
base = &a[0];
for ( i=0; i < N; i++) {
   crt = *(base + i);
}
```

optimized code

Improving Code Quality : *Code Optimization*

Common Subexpression Elimination : if the same expression is computed in many places (e.g., array addresse computations; results of macro expansion), compute it once and reuse the result.

```
e1 = *(&a[0]+offset +i);
e2 = *(&a[0]+offset +j);
```

→

```
tmp = &a[0]+offset;
e1 = *(tmp +i);
e2 = *(tmp +j);
```

original code

optimized code

Improving Code Quality : *Code Optimization*

Copy Propagation : If we have an intermediate code "copy" instruction '**x := y**', replace subsequent uses of **x** by **y** (where possible).

```
y = …
x = y;
b = x / 2;
```

```
y = …
b = y / 2;
```

original code optimized code

Improving Code Quality : *Code Optimization*

Dead Code Elimination : delete instructions whose results are not used.

```
if (1)
   x = y;
else
   x = z;
```

original code

```
x = y;
```

optimized code

Basics of Code Optimization and Machine Code Generation

- Construct Control Flow Graph (CFG) Representation for the Intermediate Code
  → *Algorithm for building CFG*

- Perform Data Flow Analysis to Collect Information Needed for Performing Optimizations
  → *Variable Liveness Analysis*

- Perform Optimizations and Generate Machine Code
  → *Algorithm for Register Allocation*

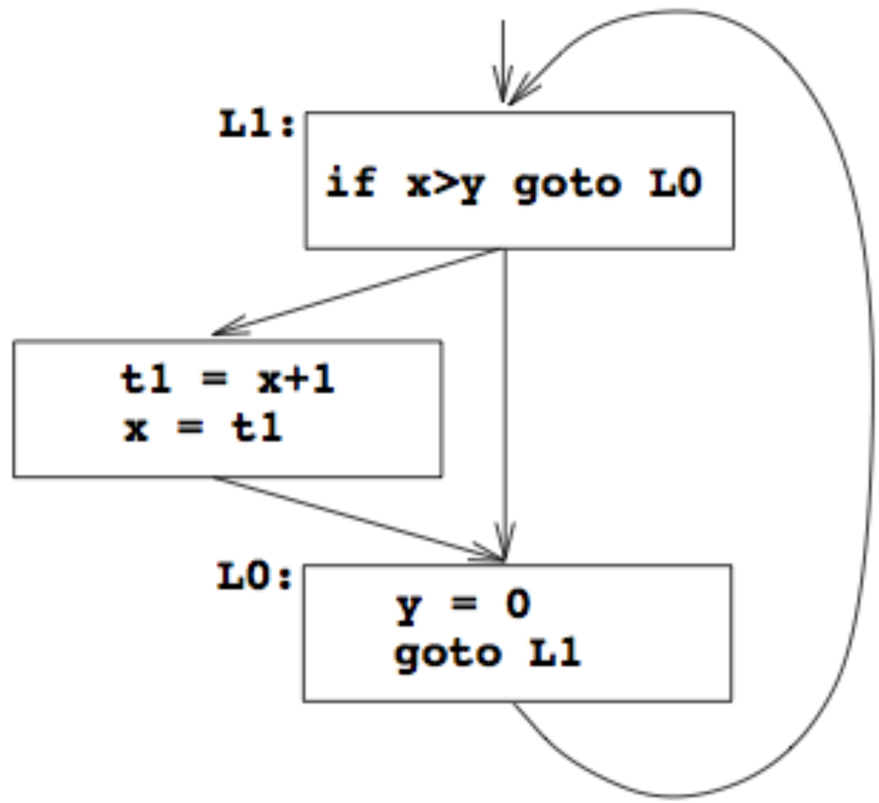
Basic Blocks and Flow Graphs

- For program analysis and optimization, it is usually necessary to know control flow relationships between different pieces of code.
- For this, we:
  - group 3-address instructions into *basic blocks*
  - represent control flow relationships between basic blocks using a *control flow graph*.

Example:

```
L1:  if x > y goto L0
     t1 = x+1
     x = t1
L0:  y = 0
     goto L1
```

$\Rightarrow$



Definition : A *basic block* is a sequence of consecutive instructions such that:

1. control enters at the beginning;
2. control leaves at the end; and
3. control cannot halt or branch except at the end.

Identifying basic blocks :

1. Determine the set of *leaders*, i.e., the first instruction of each basic block:
   (a) The first instruction of the function is a leader.
   (b) Any instruction that is the target of a branch is a leader.
   (c) Any instruction immediately following a (conditional or unconditional) branch is a leader.
2. For each leader, its basic block consists of itself and all instructions upto, but not including, the next leader (or end of function).

Example

```
/* dot product:  prod = Σ_{i=1}^{N} a[i] * b[i]  */
```

$$\texttt{/* dot product: prod} = \sum_{i=1}^{N} \mathtt{a}[i] * \mathtt{b}[i] \texttt{ */}$$

No.	leader?	Instruction	basic block
(1)	√	prod = 0	1
(2)		i = 1	1
(3)	√	t1 = 4*i	2
(4)		t2 = a[t1]	2
(5)		t3 = 4*i	2
(6)		t4 = b[t3]	2
(7)		t5 = t2*t4	2
(8)		t6 = prod+t5	2
(9)		prod = t6	2
(10)		t7 = i+1	2
(11)		i = t7	2
(12)		if i ≤ N goto (3)	2

Control Flow Graphs

Definition : A flow graph for a function is a directed graph $G = (V, E)$ whose nodes are the basic blocks of the function, and where $a \to b \in E$ iff control can leave a and immediately enter b.

The distinguished *initial* node if a flow graph is the basic block whose leader is the first instruction of the function.

Constructing the flow graph of a function :

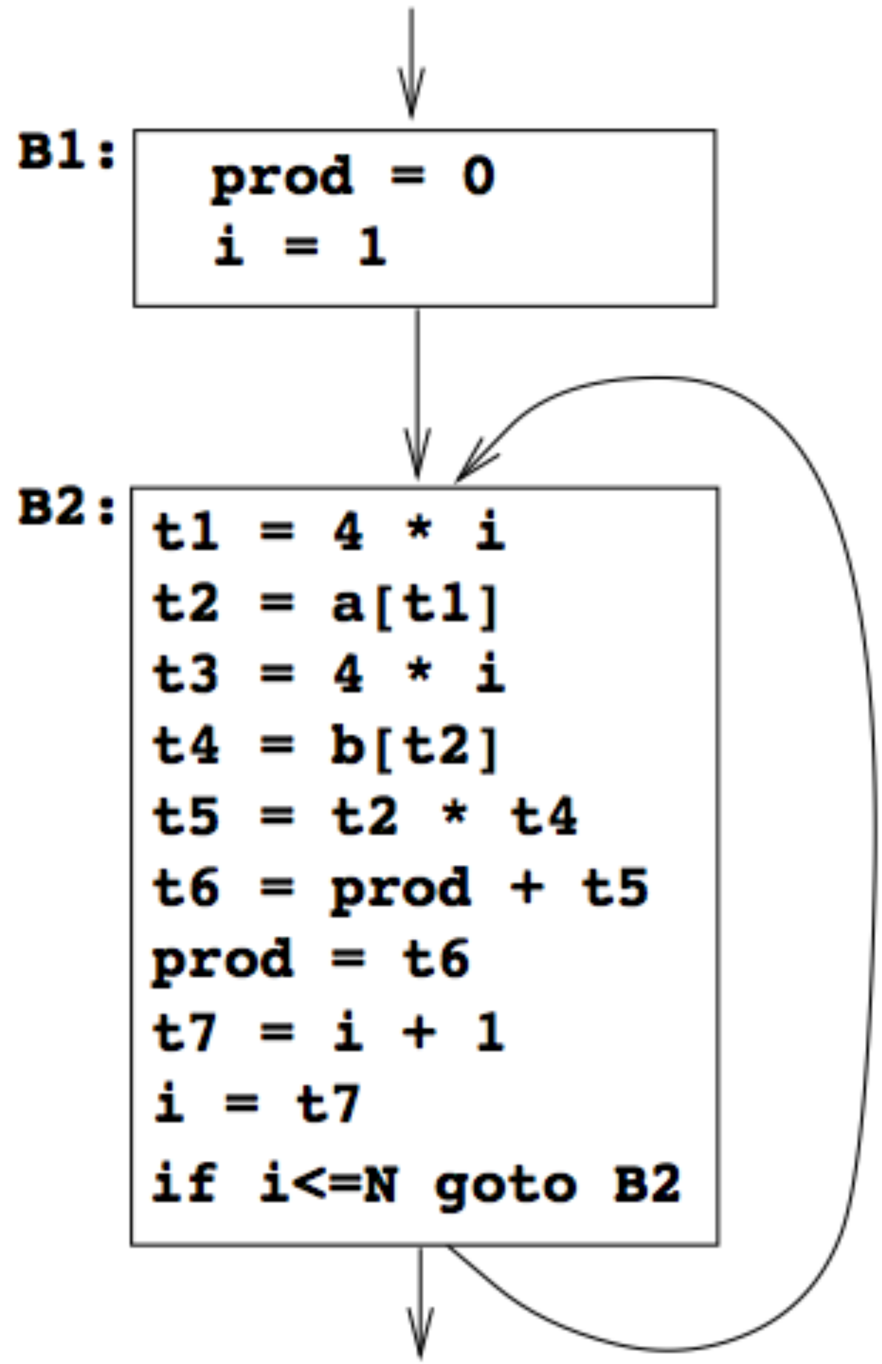
1. Identify the basic blocks of the function.
2. There is a directed edge from block B_1 to block B_2 if
   (a) there is a (conditional or unconditional) jump from the last instruction of B_1 to the first instruction of B_2; or
   (b) B_2 immediately follows B_1 in the textual order of the program, and B_1 does not end in an unconditional jump.

Predecessors and Successors : if there is an edge $a \rightarrow b$ then a is a *precedessor* of b, and b is a *successor* of a.

Example :

```
L1:  prod = 0
     i = 1
L2:  t1 = 4*i
     t2 = a[t1]
     t3 = 4*i
     t4 = b[t3]
     t5 = t2*t4
     t6 = prod+t5
     prod = t6
     t7 = i+1
     i = t7
     if i ≤ N goto L2
```

$\Rightarrow$



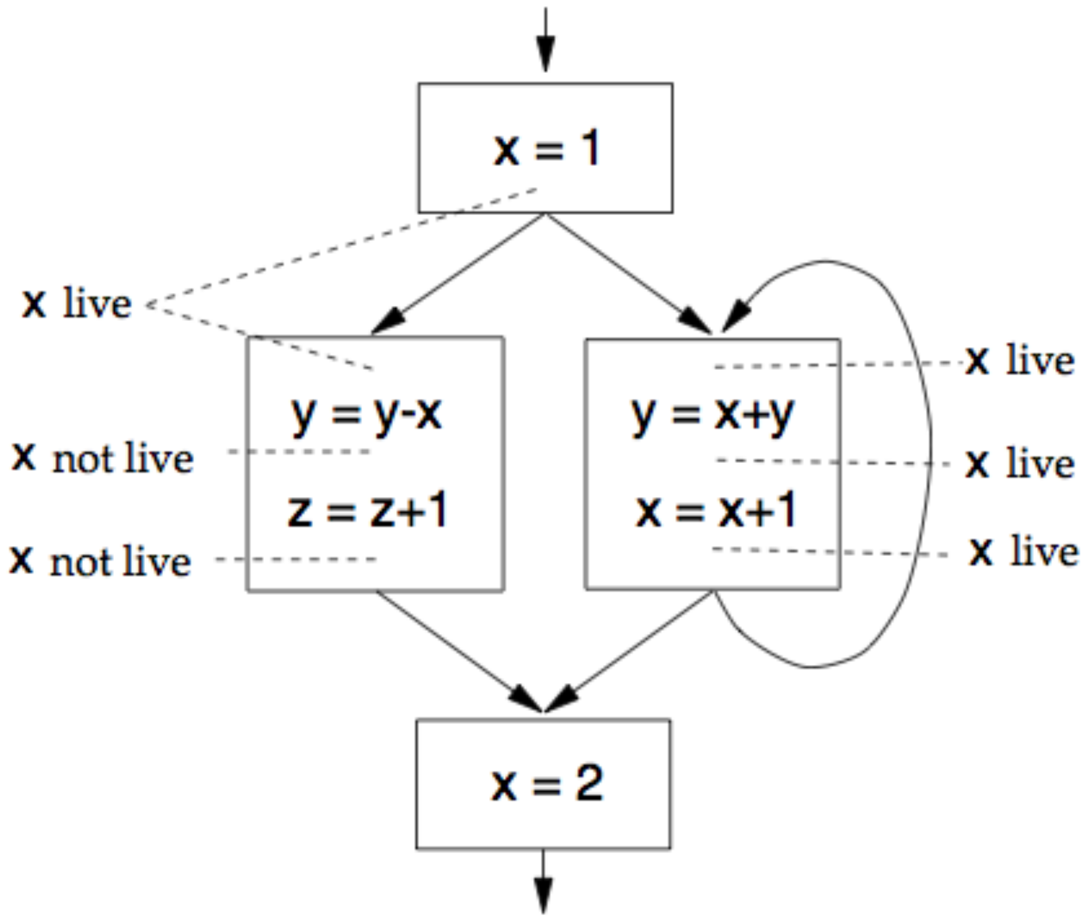
Improving Code Quality : *Register Allocation*

- Rationale
  - A value in a register can be accessed much more efficiently than one in memory
- Liveness Analysis to build Live Ranges
  - Identifies durations for which each variable could benefit from using a register
- Perform Register Allocation
  - CPU has limited registers → keep frequently used values in registers

Variable Liveness

Definition : A variable is *live* at a point in a program if it *may* be used at a later point before being redefined.

Example :



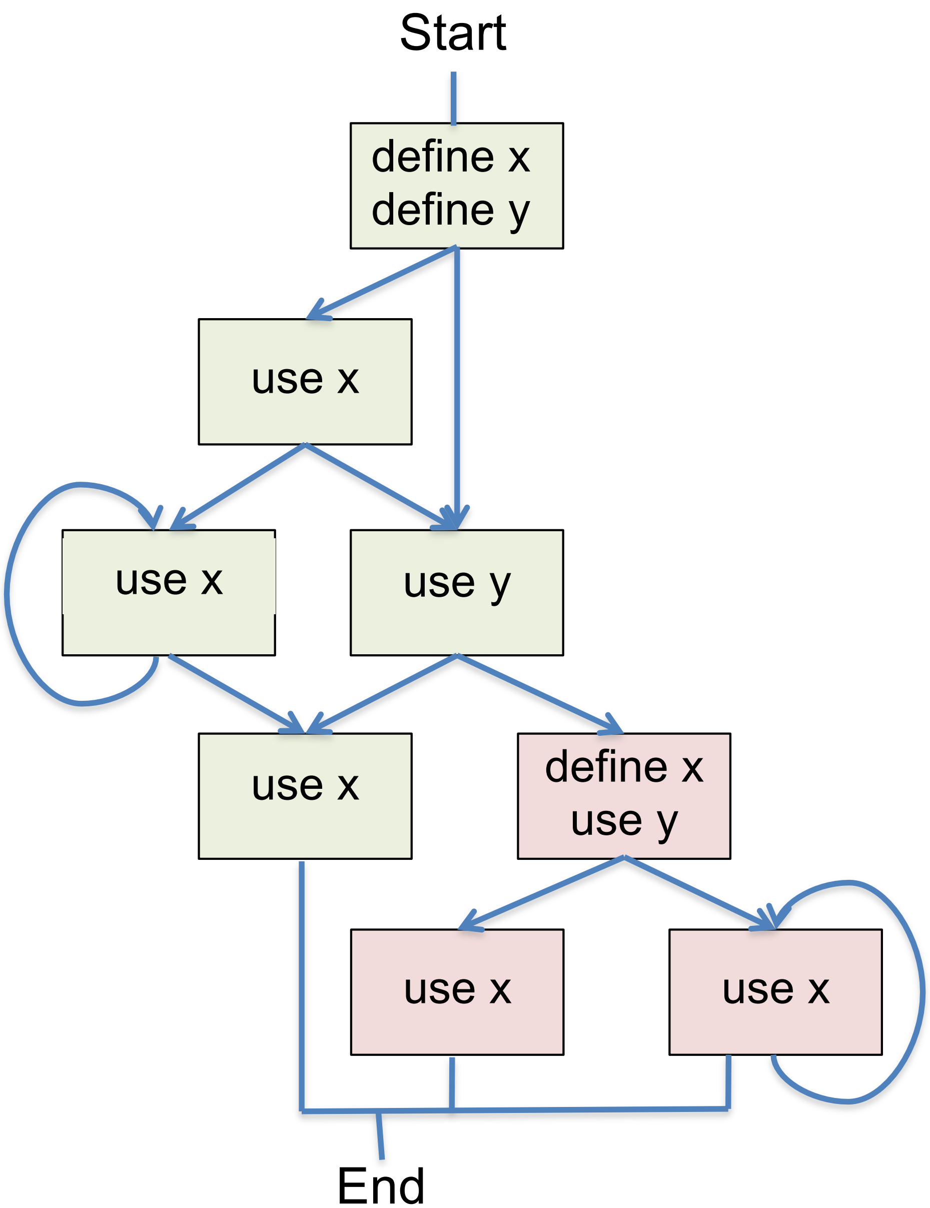
Live Ranges

Definition : A *live range* is an isolated and connected group of basic blocks in which a variable is live.

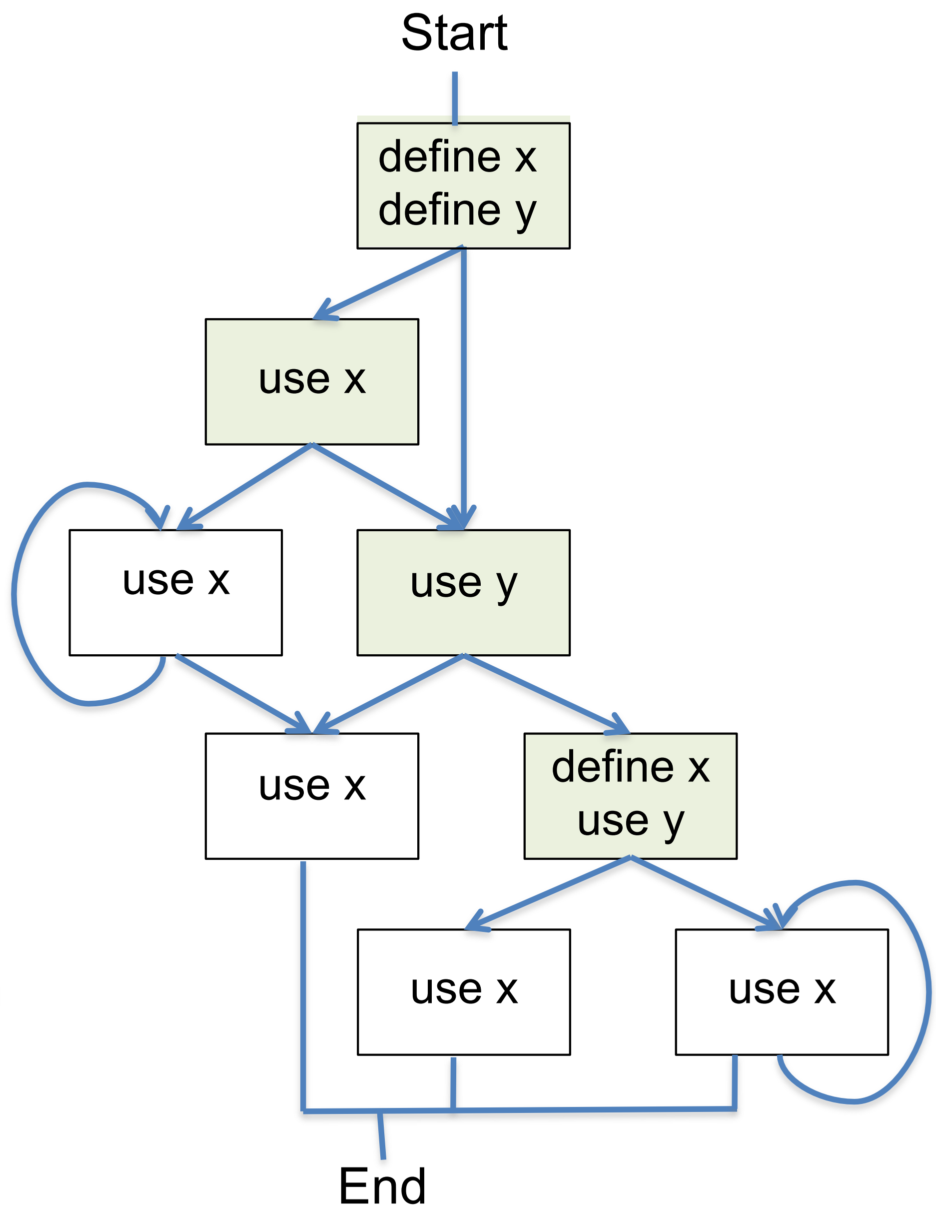
- Usually, a live range begins at a definition point of a variable and ends at its last uses.
- Different variables may have different live ranges.

  (⇒ a given basic block may be part of many different live ranges.)
- A given variable may have several different live ranges.

2 Live Ranges of x

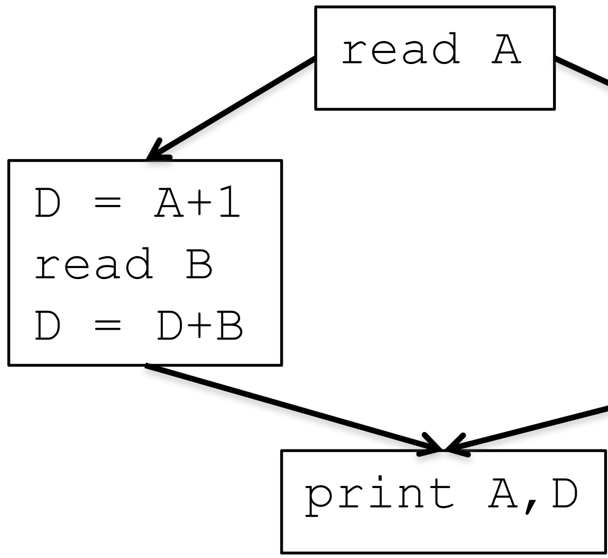


1 Live Range of y

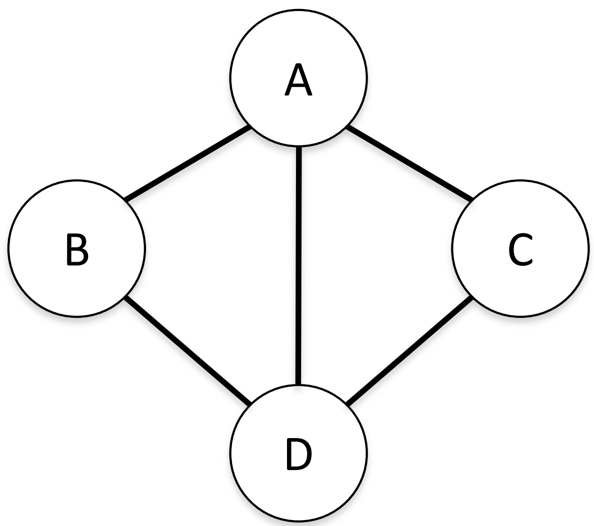


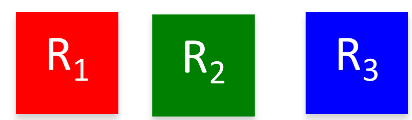
Global Register Allocation : considers the entire body of a function or procedure:

- Tries to keep frequently accessed values in registers, esp. across loops.
- Uses loop nesting depth as a guide to frequency of access: variables in the most deeply nested loops are assumed to be accessed the most frequently.



<u>Register Interference Graph</u>
nodes: live ranges
edges: live ranges overlap





k-coloring, where *k* is the number of registers

Attempt n-coloring

Color the interference graph using R colors where R is the number of registers.

Observation: If there is a node n with < R neighbors, then no matter how the neighbors are colored, there will be at least one color left over to color node n.

Remove n and its edges to get G'

Repeat the above process to get G''

....

If an empty graph results, R-coloring is possible. Assign colors in reverse of the order in which they were removed.

Attempt Coloring Contd..

Input: Graph G

Output: N-coloring of G

While there exists n in G with < N edges do

    Eliminate n & all its edges from G; list n

End while

If G is empty the

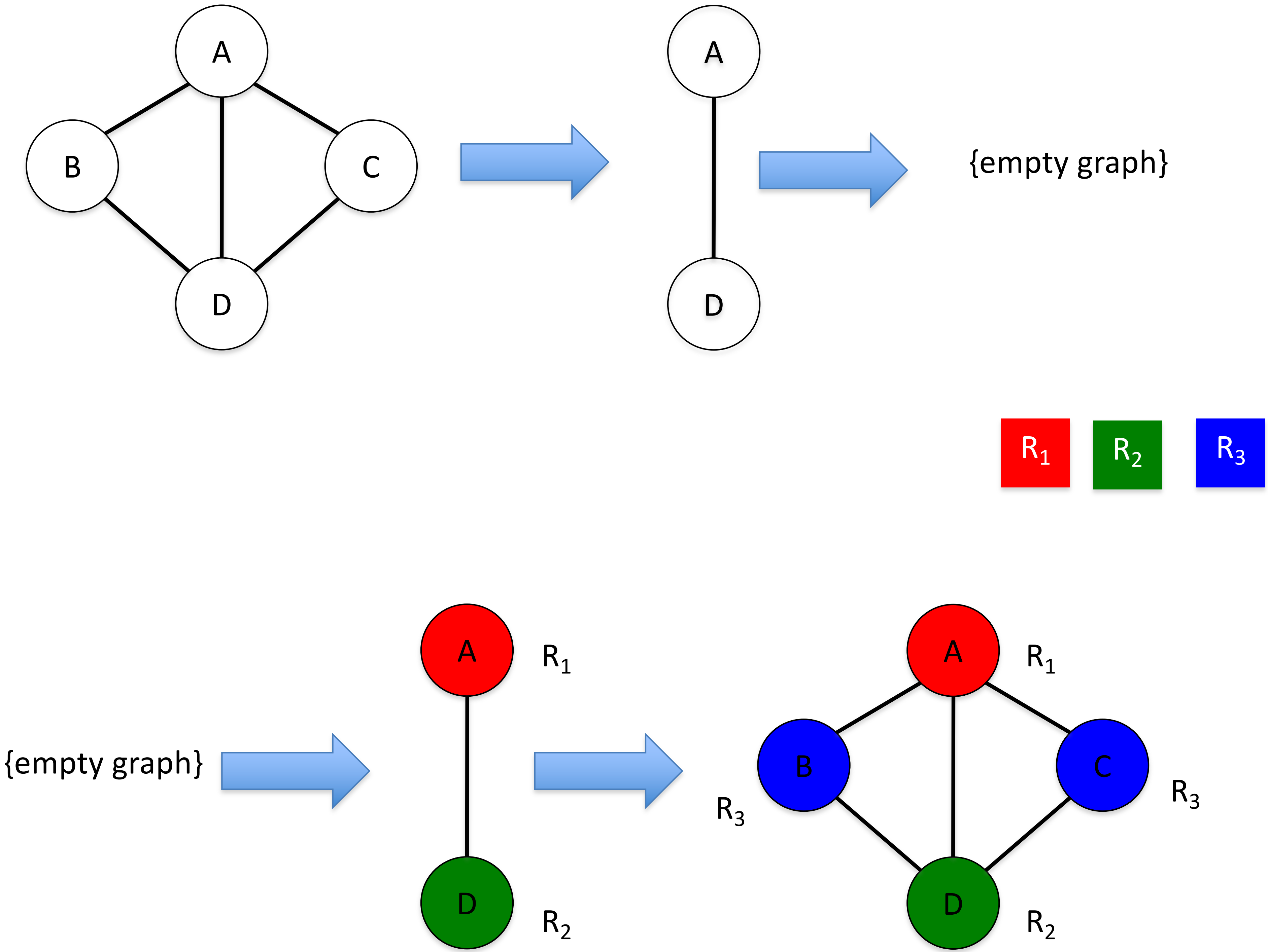
    for each node i in list in reverse order do

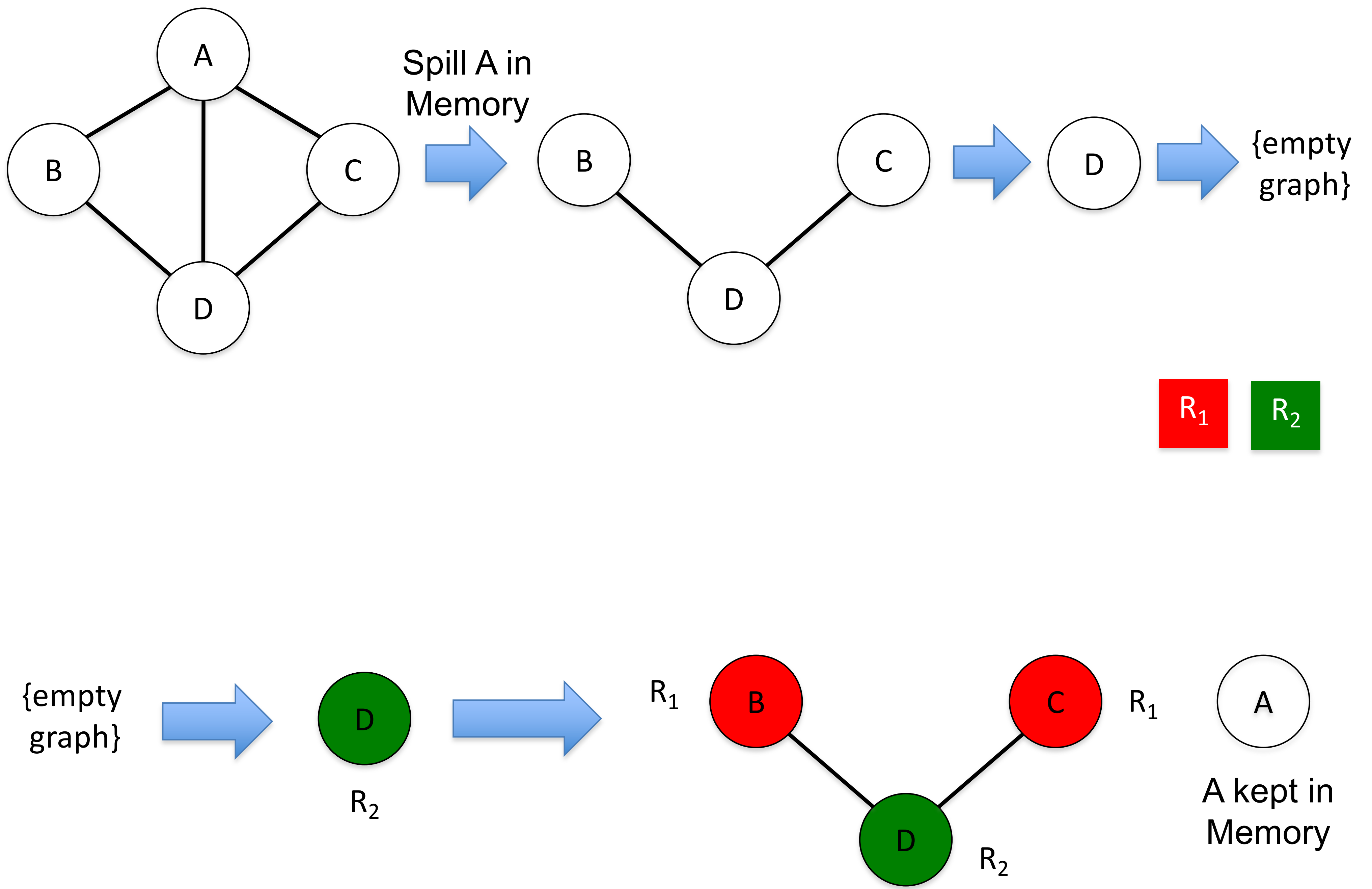
        Add i & its edges back to G;

        choose color for i

    endfor

End if


A
B
C
D
A
D
{empty graph}
R1
R2
R3
{empty graph}
A
R1
D
R2
A
R1
B
R3
C
R3
D
R2


A
B
C
D
Spill A in Memory
B
C
D
D
{empty graph}
R_1
R_2
{empty graph}
D
R_2
R_1
B
C
R_1
A
D
R_2
A kept in Memory

Liveness Analysis and Live Range Construction

- Global Analysis
  → *Finds what variables are live at basic block boundaries*
- Local Analysis
  → *Finds what variables are live at all points within basic blocks*
- Build Live Ranges

Computing Liveness Information (within a basic block)

Suppose we know which variables are live at the exit from the basic block. Then:

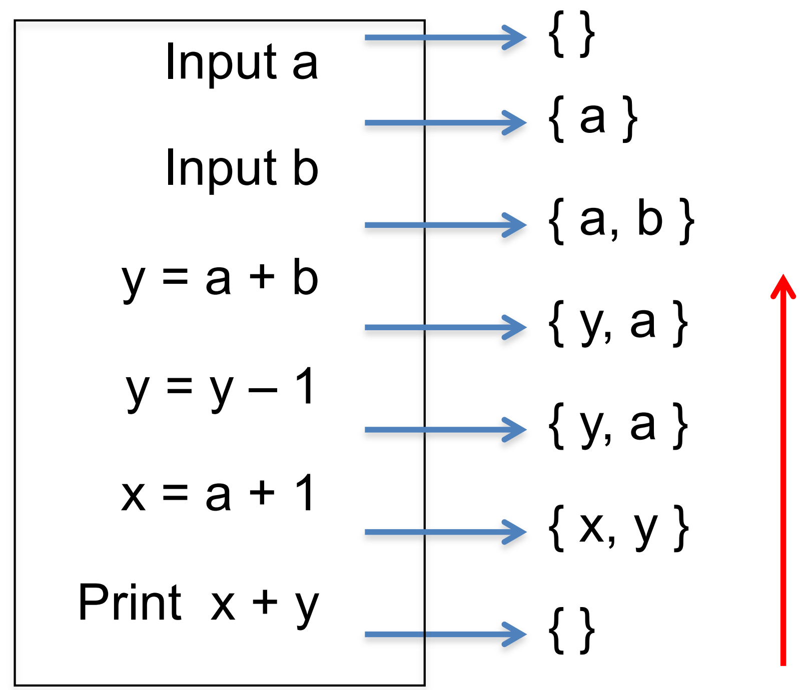
- Scan backwards from the end of the block. At the point immediately before an instruction

  I : **x** := **y** *op* **z**

  we have:

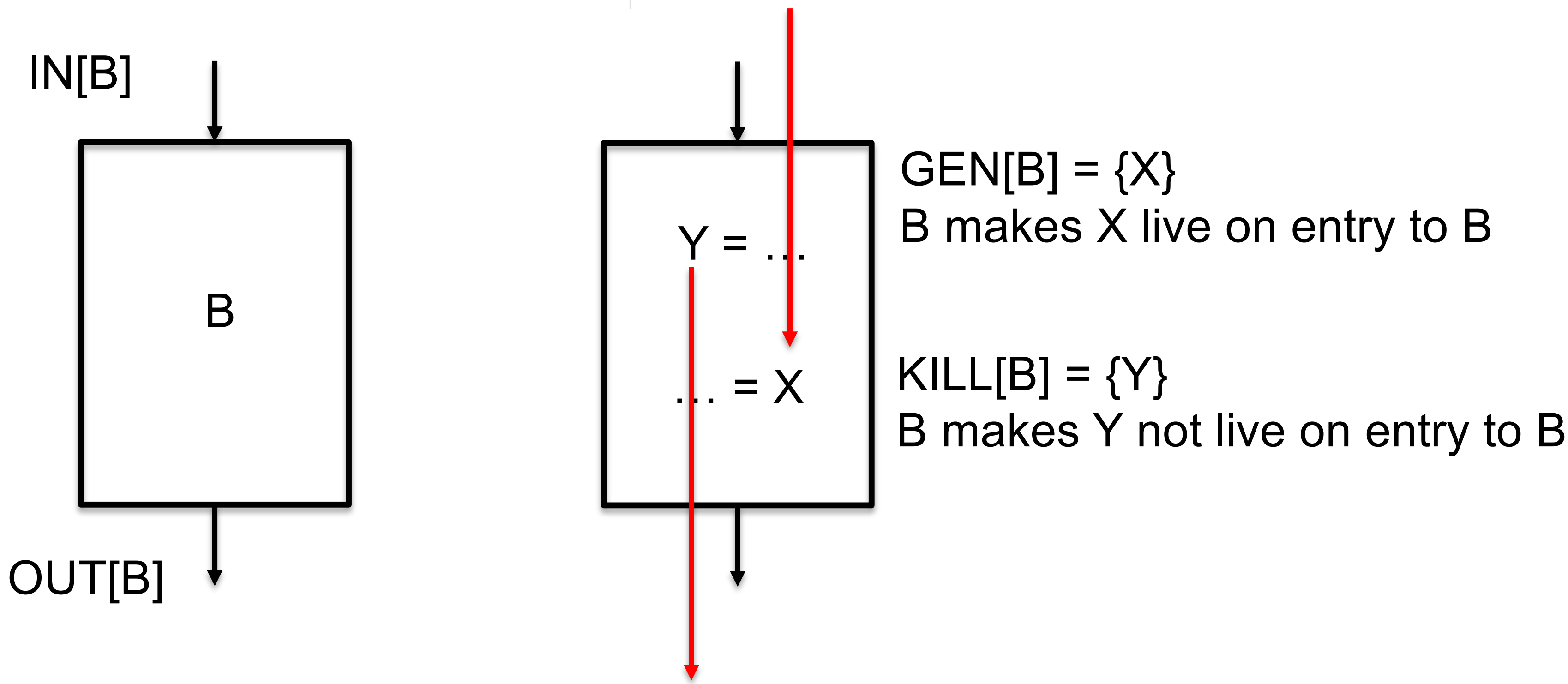
  -- x is not live
  -- y and z are live

  Live Before I = (Live After I – {x}) U {y, z}



Computing Liveness Information (dataflow analysis)

We compute IN[B] and OUT[B], the sets of variables that are live at the beginning and end of each basic block, respectively, in a flow graph, as follows:



Initialization:

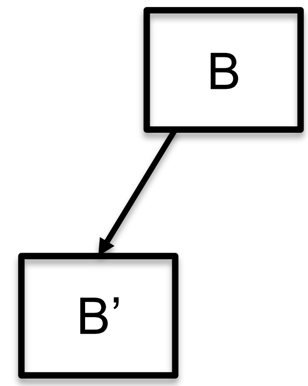
$\text{IN}[B] = \emptyset$ for all B

$$\text{OUT}[B] = \begin{cases} \text{all globals} & \text{if } B \text{ is an exit block} \\ \emptyset & \text{otherwise} \end{cases}$$

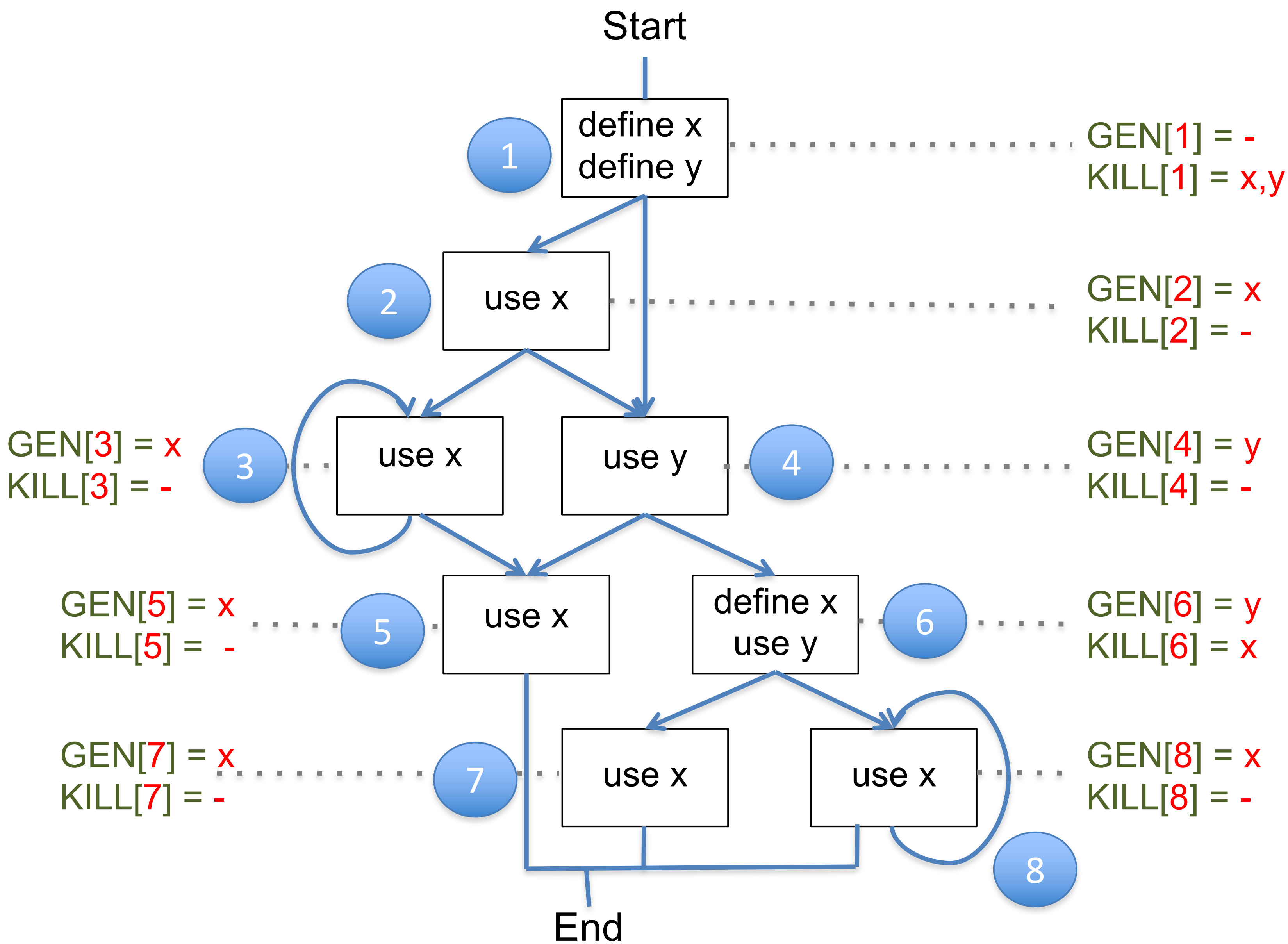
of a function other than main()

Propagation: For each non-exit block B:

- $$\text{OUT}[B] = \bigcup_{B' \in successors(B)} \text{IN}[B']$$
- $\text{IN}[B] = (OUT[B] - KILL[B]) \bigcup GEN[B]$, where
  $GEN[B] = \{v : \ variable\ v\ is\ read\ before\ being\ written\}$
  $KILL[B] = \{v : \ variable\ v\ is\ defined\ in\ B\}$



Since a flow graph may have cycles, we need to iterate this step until there is no change to any IN or OUT set.


Start
define x
define y
1
GEN[1] = -
KILL[1] = x,y
use x
2
GEN[2] = x
KILL[2] = -
GEN[3] = x
KILL[3] = -
3
use x
use y
4
GEN[4] = y
KILL[4] = -
GEN[5] = x
KILL[5] = -
5
use x
define x
use y
6
GEN[6] = y
KILL[6] = x
GEN[7] = x
KILL[7] = -
7
use x
use x
GEN[8] = x
KILL[8] = -
8
End

IN[1] = (OUT[1] – KILL[1]) U GEN[1] = OUT[1] - {x,y}

OUT[1] = IN[2] U IN[4]

IN[2] = (OUT[2] – KILL[2]) U GEN[2] = OUT[2] U {x}

OUT[2] = IN[3] U IN[4]

IN[3] = (OUT[3] – KILL[3]) U GEN[3] = OUT[3] U {x}

OUT[3] = IN[3] U IN[5]

IN[4] = (OUT[4] – KILL[4]) U GEN[4] = OUT[4] U {y}

OUT[4] = IN[5] U IN[6]

IN[5] = (OUT[5] – KILL[5]) U GEN[5] = OUT[5] U {x}

OUT[5] = {}

IN[6] = (OUT[6] – KILL[6]) U GEN[6] = (OUT[6] – {x}) U {y}

OUT[6] = IN[7] U IN[8]

IN[7] = (OUT[7] – KILL[7]) U GEN[7] = OUT[7] U {x}

OUT[7] = {}

IN[8] = (OUT[8] – KILL[8]) U GEN[8] = OUT[8] U {x}

OUT[8] = IN[8]

$$OUT(b) = \bigcup_{s \text{ in } Succ(b)} IN(s)$$

$$IN(b) = (OUT(b) - KILL(b)) \cup GEN(b)$$

IN[1] = OUT[1] - {x,y}

OUT[1] = IN[2] U IN[4]

IN[2] = OUT[2] U {x}

OUT[2] = IN[3] U IN[4]

IN[3] = OUT[3] U {x}

OUT[3] = IN[3] U IN[5]

IN[4] = OUT[4] U {y}

OUT[4] = IN[5] U IN[6]

IN[5] = OUT[5] U {x}

OUT[5] = {}

IN[6] = (OUT[6] – {x}) U {y}

OUT[6] = IN[7] U IN[8]

IN[7] = OUT[7] U {x}

OUT[7] = {}

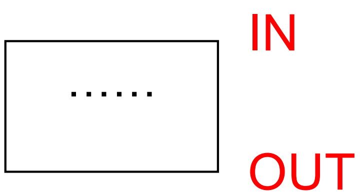
IN[8] = OUT[8] U {x}

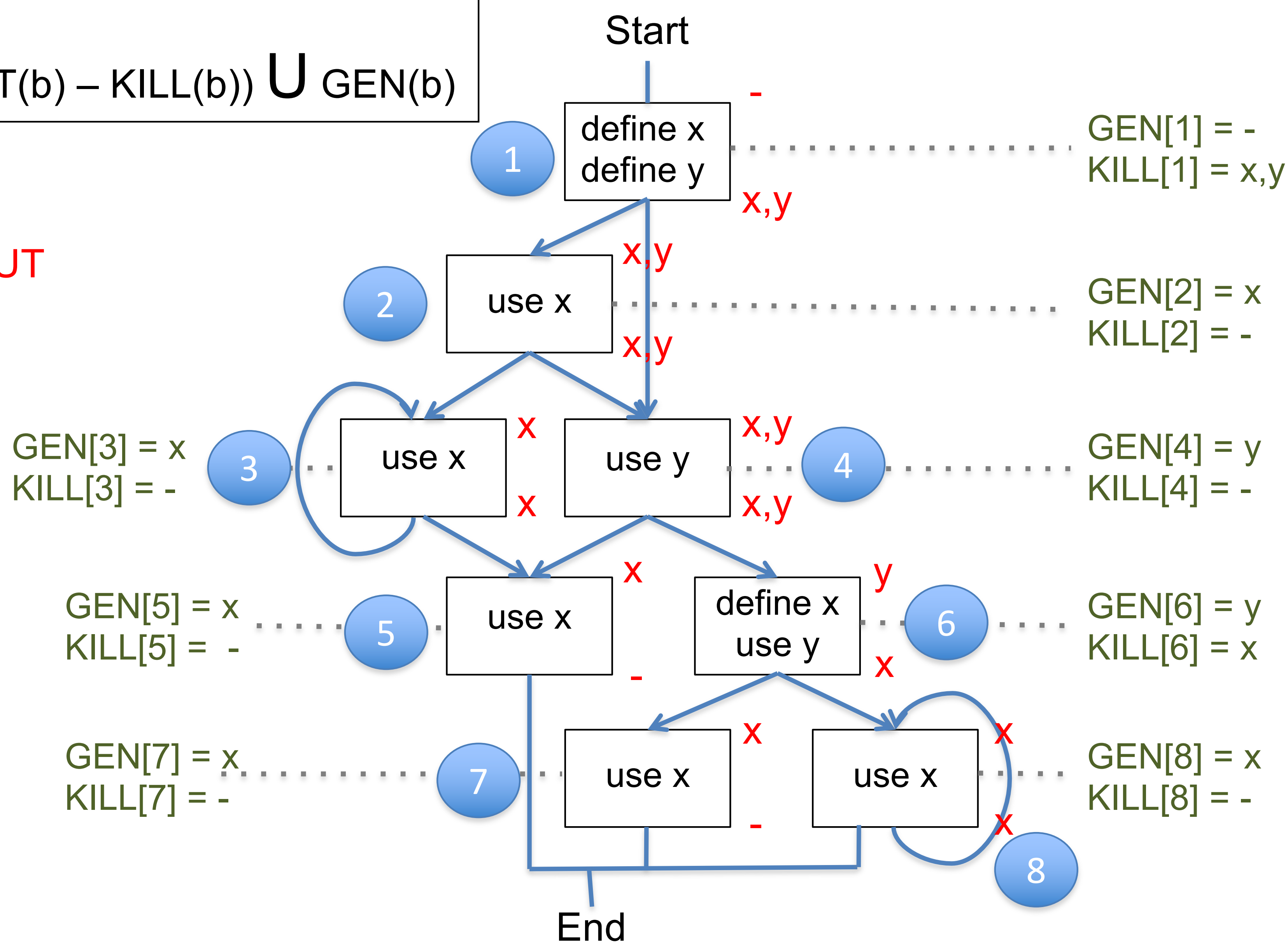
OUT[8] = IN[8]

IN[1]	{}	{}	{}	{}
OUT[1]	{}	{x,y}	{x,y}	{x,y}
IN[2]	{}	{x,y}	{x,y}	{x,y}
OUT[2]	{}	{x,y}	{x,y}	{x,y}
IN[3]	{}	{x}	{x}	{x}
OUT[3]	{}	{x}	{x}	{x}
IN[4]	{}	{x,y}	{x,y}	{x,y}
OUT[4]	{}	{x,y}	{x,y}	{x,y}
IN[5]	{}	{x}	{x}	{x}
OUT[5]	{}	{}	{}	{}
IN[6]	{}	{y}	{y}	{y}
OUT[6]	{}	{x}	{x}	{x}
IN[7]	{}	{x}	{x}	{x}
OUT[7]	{}	{}	{}	{}
IN[8]	{}	{x}	{x}	{x}
OUT[8]	{}	{}	{x}	{x}

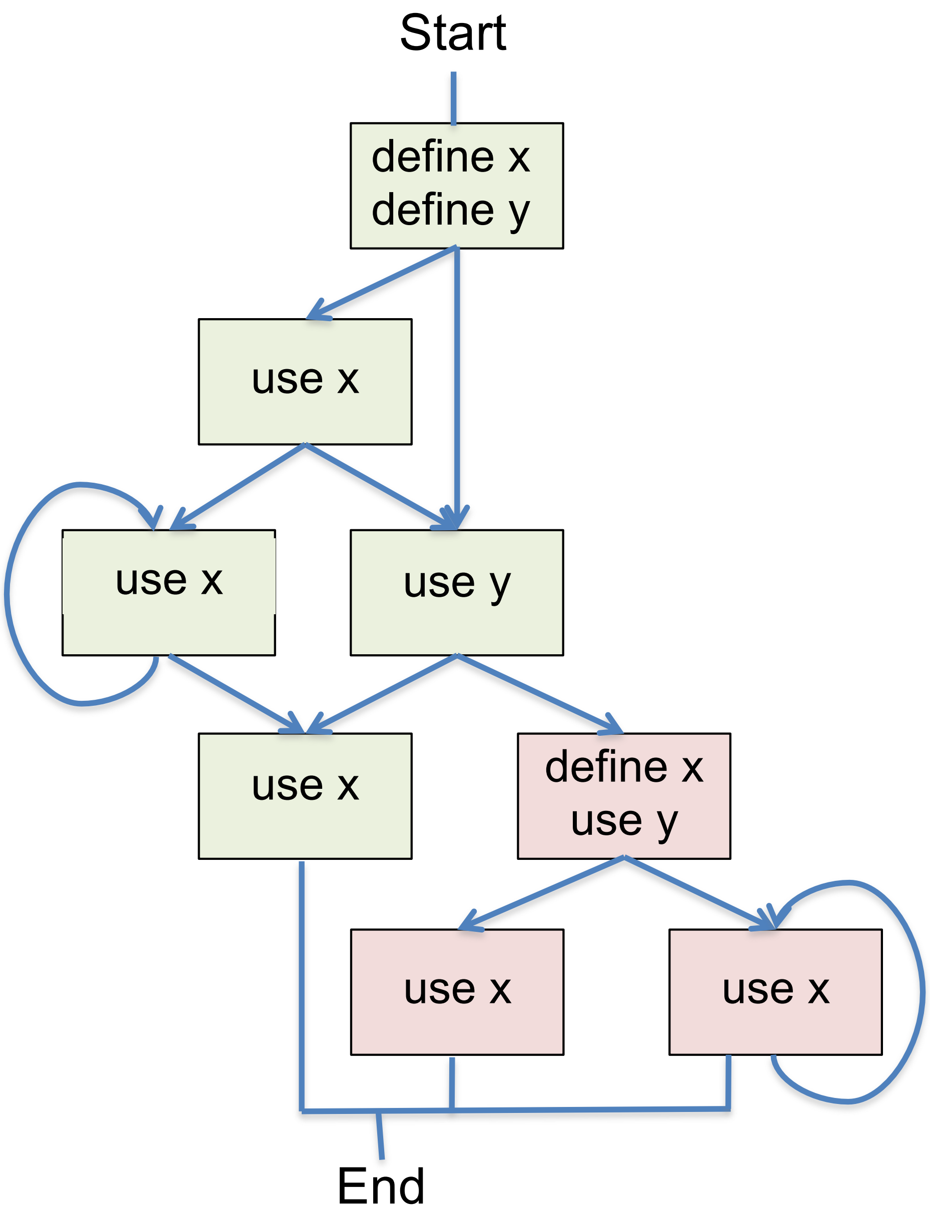
$$OUT(b) = \bigcup_{s \text{ in } Succ(b)} IN(s)$$

$$IN(b) = (OUT(b) - KILL(b)) \cup GEN(b)$$

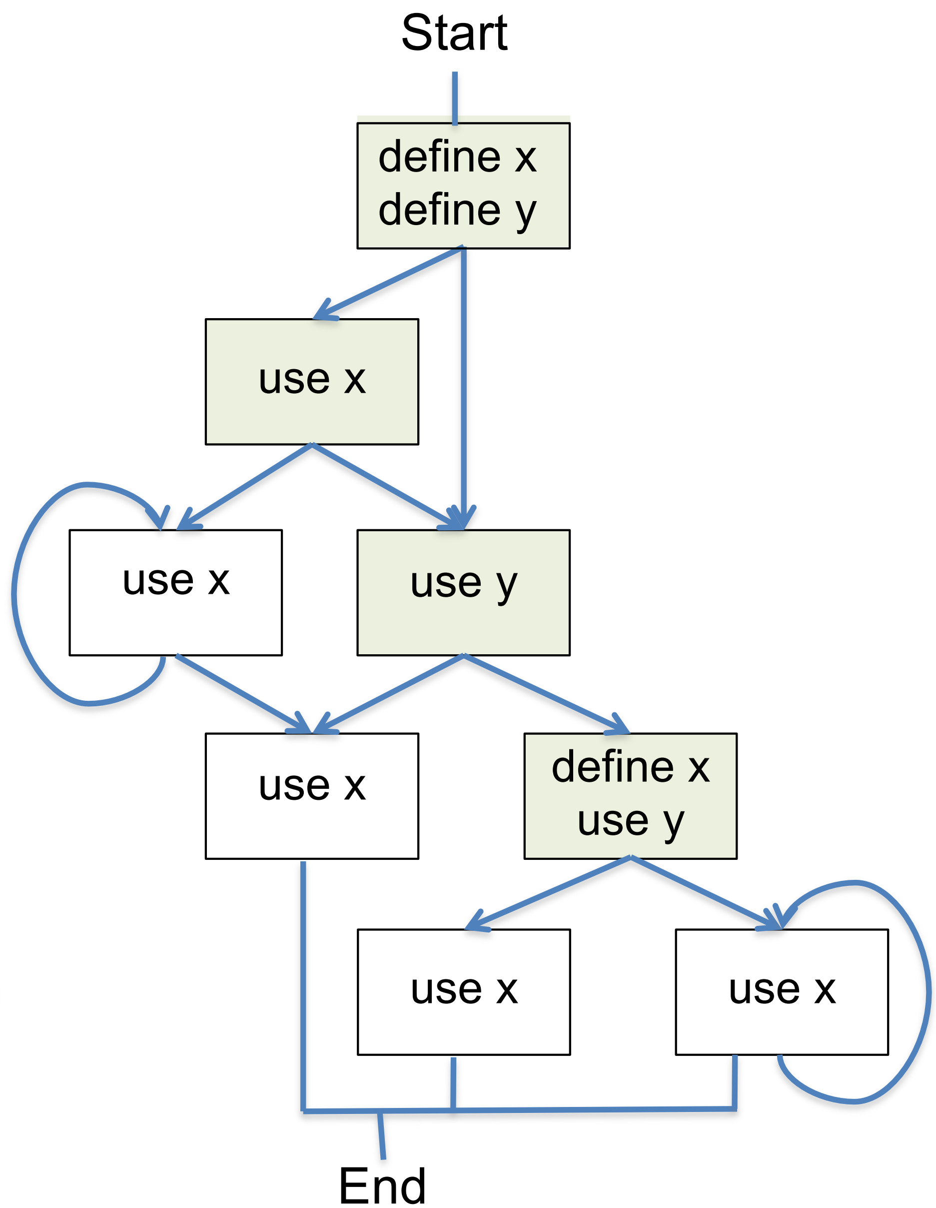




2 Live Ranges of x



1 Live Range of y

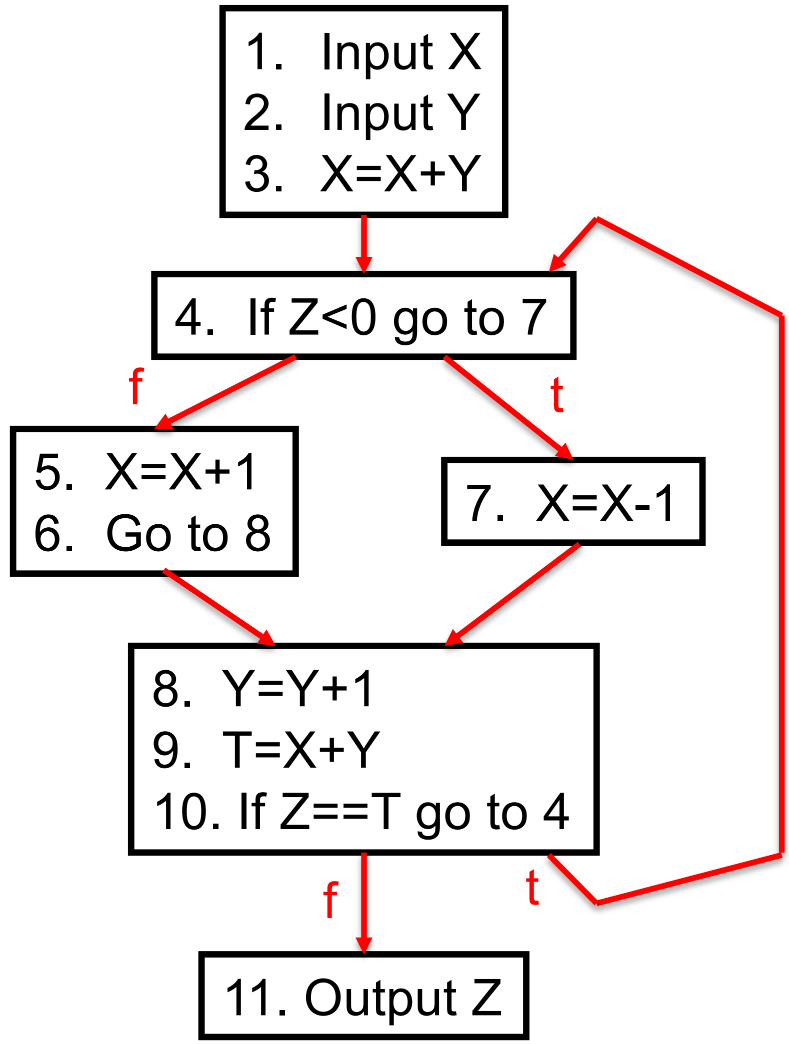


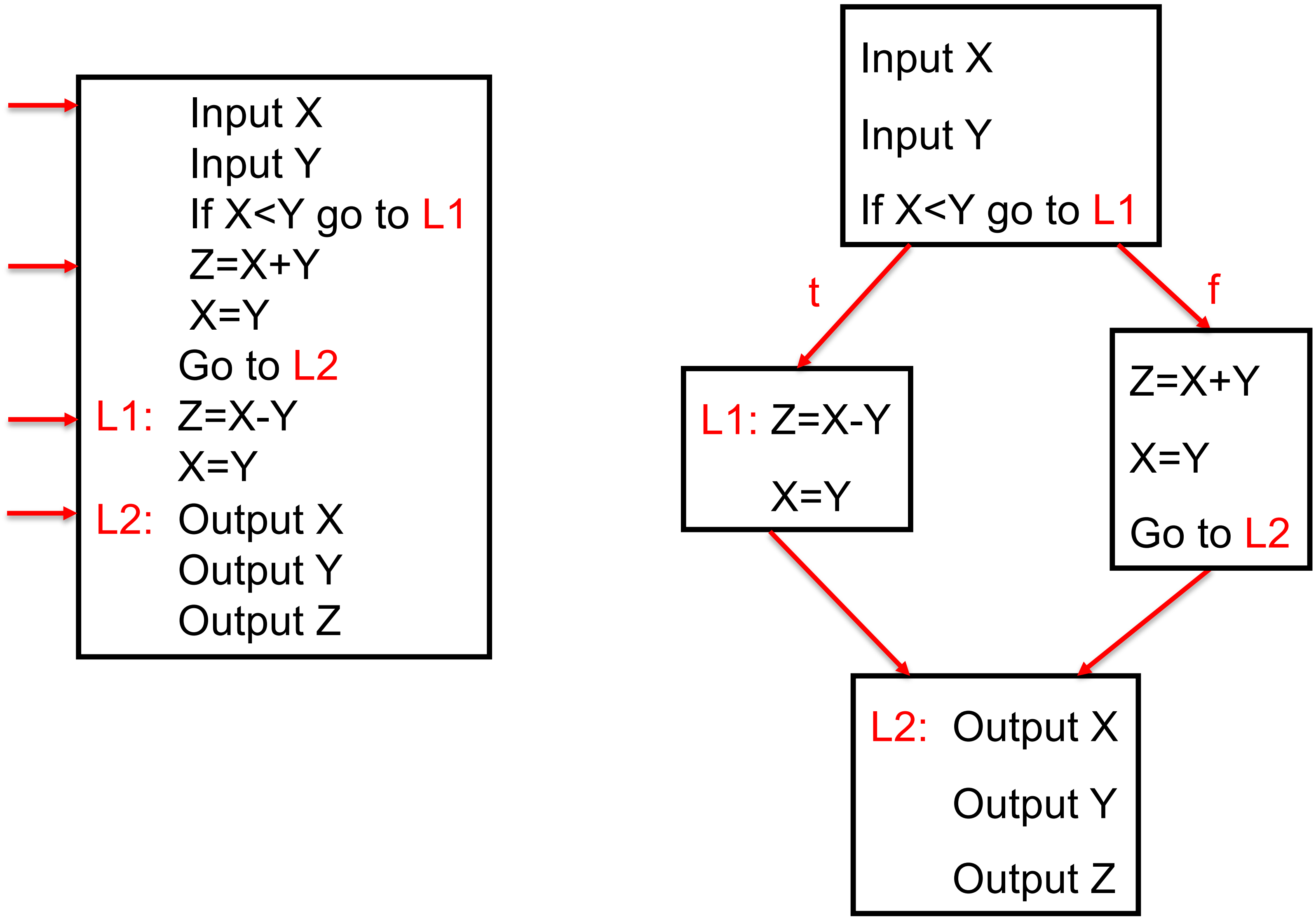
Algorithm for solving data flow equations:

```
For each block B do
    if B is the exit block then
        OUT[B] = set of global variables
        IN[B] = (OUT[B] – KILL[B]) U GEN[B]
    else
        OUT[B] = IN[B] = { }
    endif
Endfor
DONE = false
While not DONE do
    DONE = true;
    for each B which is not the exit block do
        new  =        U            IN[B']
                 B' ε SUCC(B)
        if new != OUT[B] then
            DONE = false;
            OUT[B] = new;
            IN[B] = (OUT[B] – KILL[B]) U GEN[B]
        Endif
    Endfor
Endwhile
```

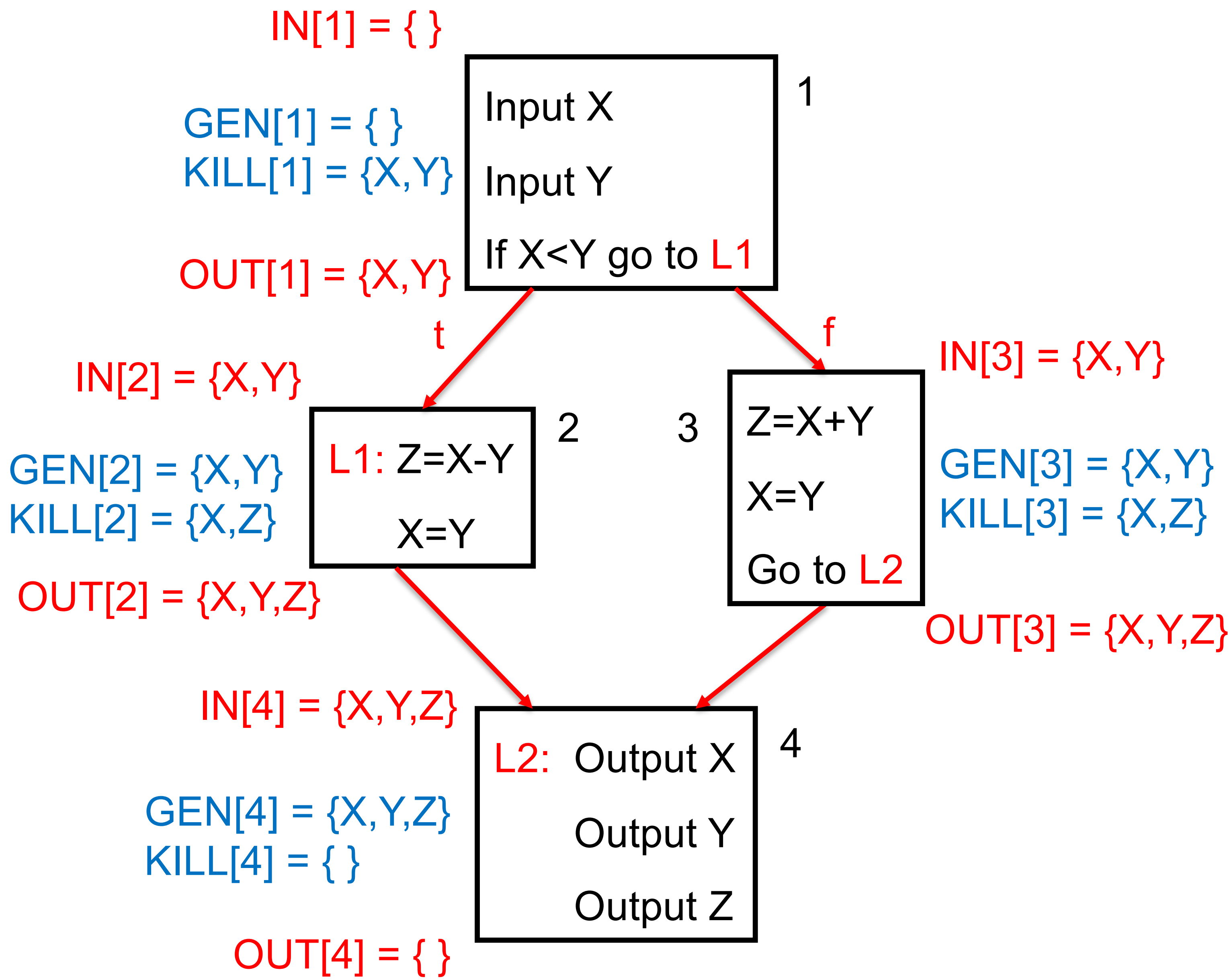
Sample Problems for Review

1. Input X
2. Input Y
3. X=X+Y
4. If Z<0 go to 7
5. X=X+1
6. Go to 8
7. X=X-1
8. Y=Y+1
9. T=X+Y
10. If Z==T go to 4
11. Output Z

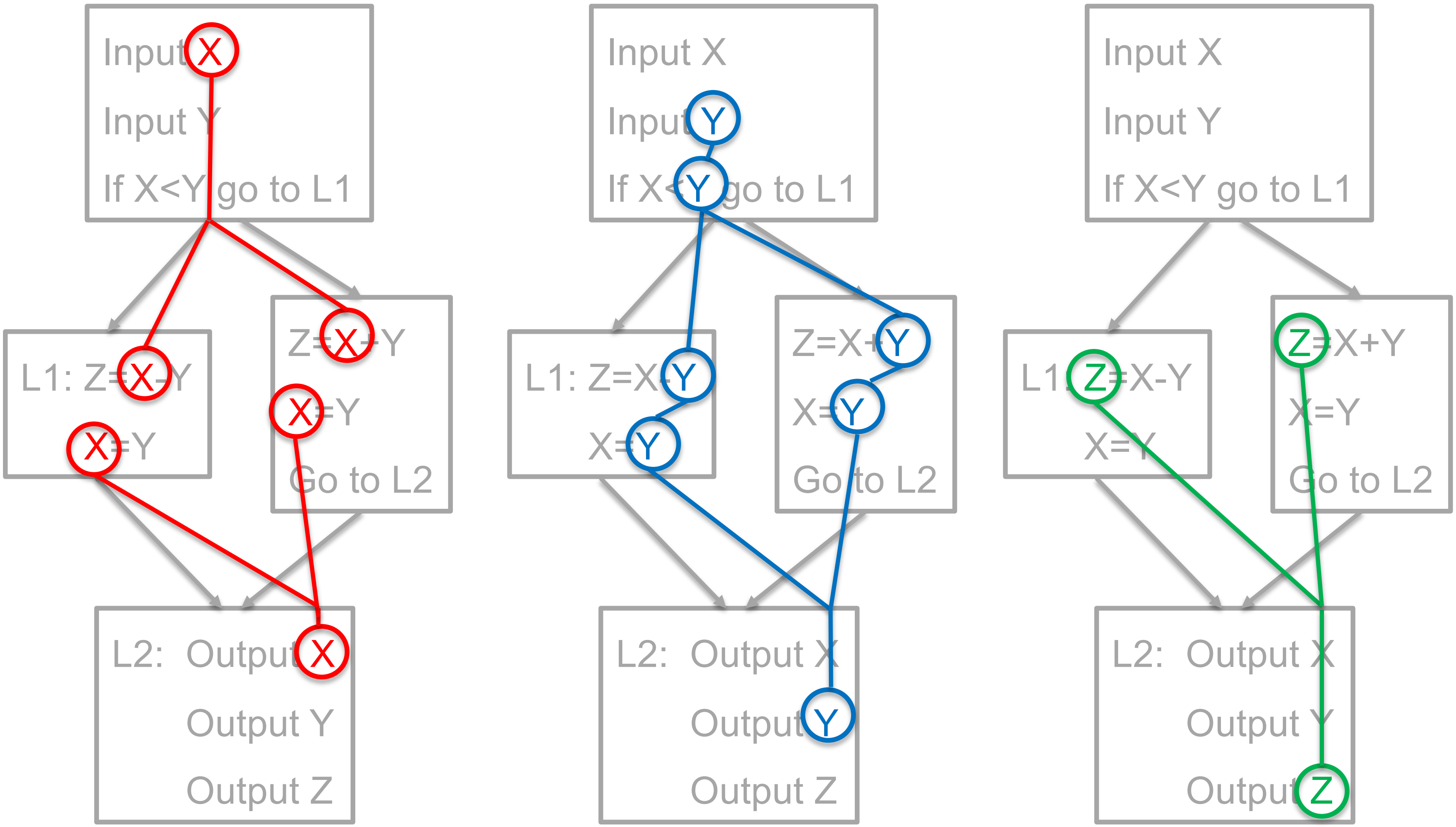




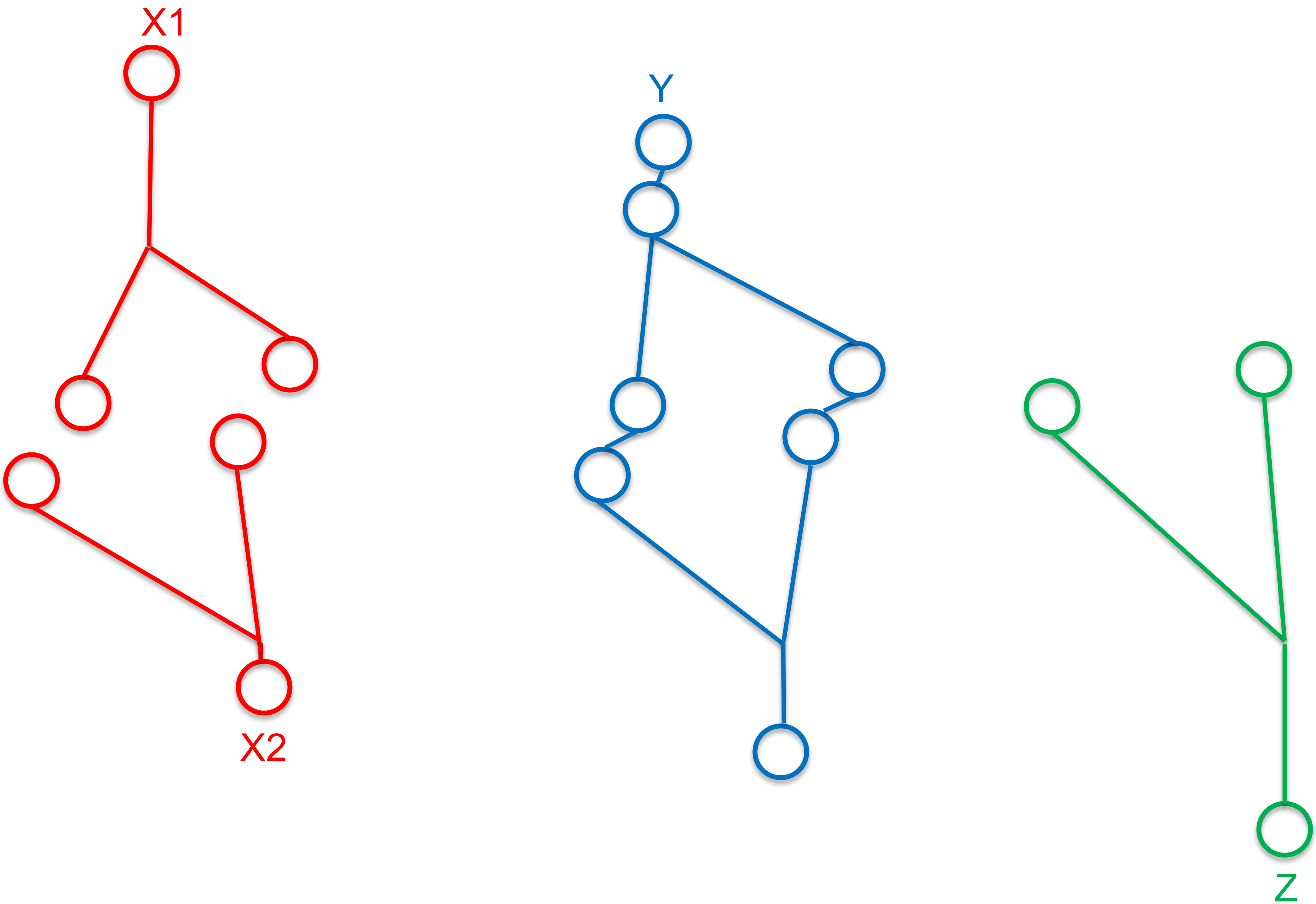
Input X
Input Y
If X<Y go to L1
Z=X+Y
X=Y
Go to L2
L1: Z=X-Y
X=Y
L2: Output X
Output Y
Output Z
Input X
Input Y
If X<Y go to L1
t
f
L1: Z=X-Y
X=Y
Z=X+Y
X=Y
Go to L2
L2: Output X
Output Y
Output Z


IN[1] = { }
GEN[1] = { }
KILL[1] = {X,Y}
OUT[1] = {X,Y}
Input X
Input Y
If X<Y go to L1
1
t
f
IN[2] = {X,Y}
GEN[2] = {X,Y}
KILL[2] = {X,Z}
OUT[2] = {X,Y,Z}
L1: Z=X-Y
X=Y
2
3
Z=X+Y
X=Y
Go to L2
IN[3] = {X,Y}
GEN[3] = {X,Y}
KILL[3] = {X,Z}
OUT[3] = {X,Y,Z}
IN[4] = {X,Y,Z}
GEN[4] = {X,Y,Z}
KILL[4] = { }
OUT[4] = { }
L2: Output X
Output Y
Output Z
4

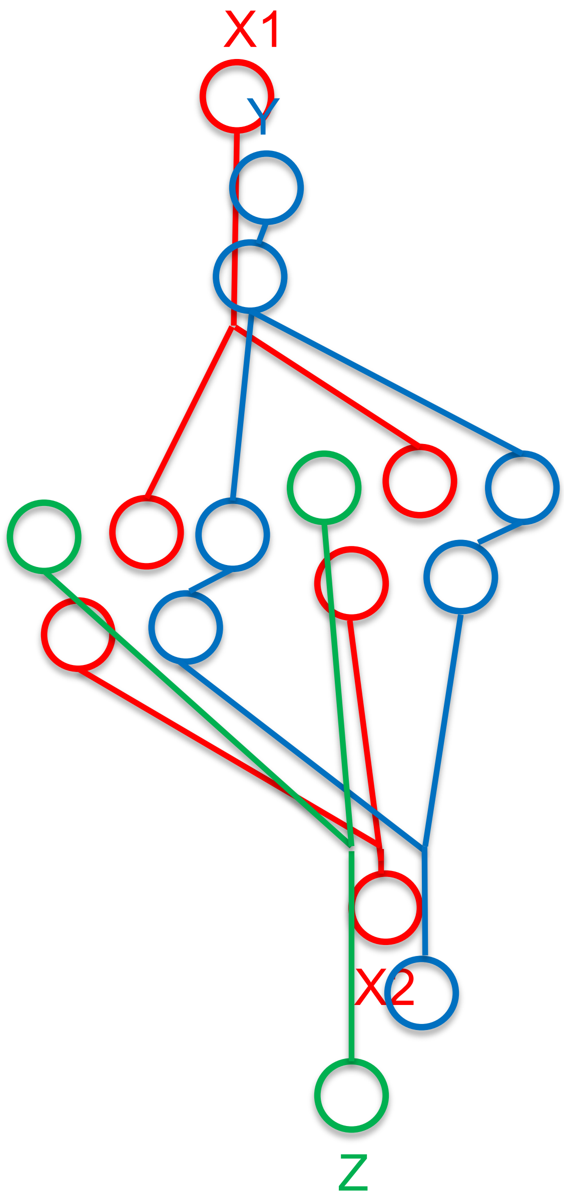
LIVE RANGES OF X, Y and Z

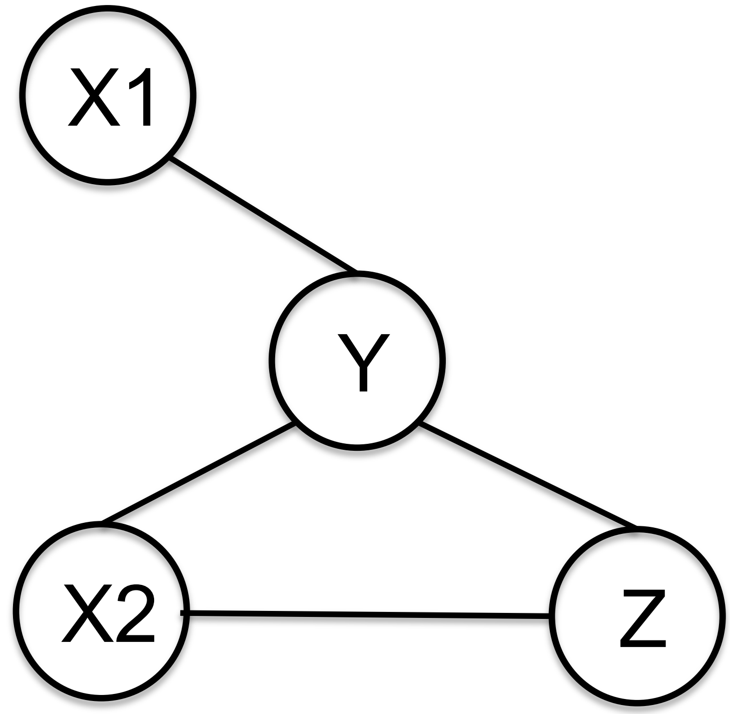


LIVE RANGES OF X, Y and Z

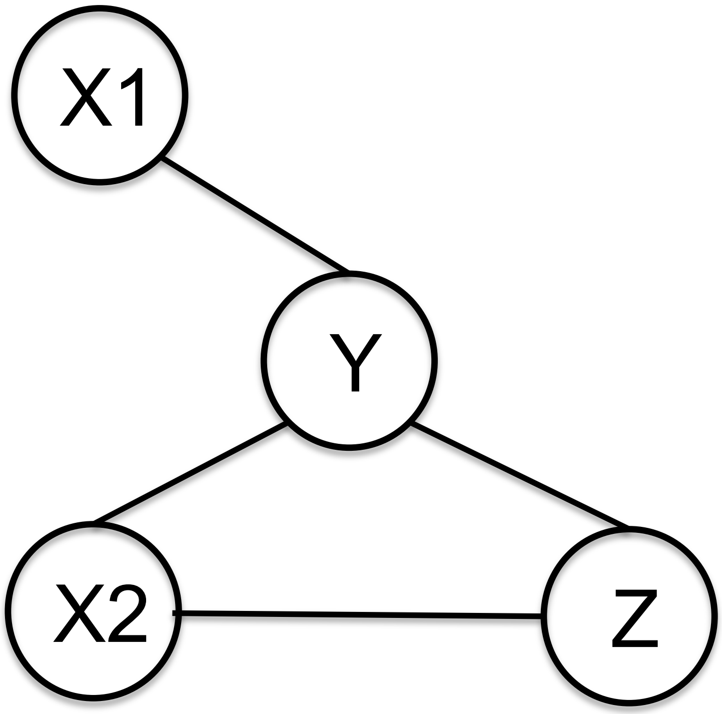


INTERFERENCE GRAPH





REGISTER ALLOCATION: R1, R2, R3

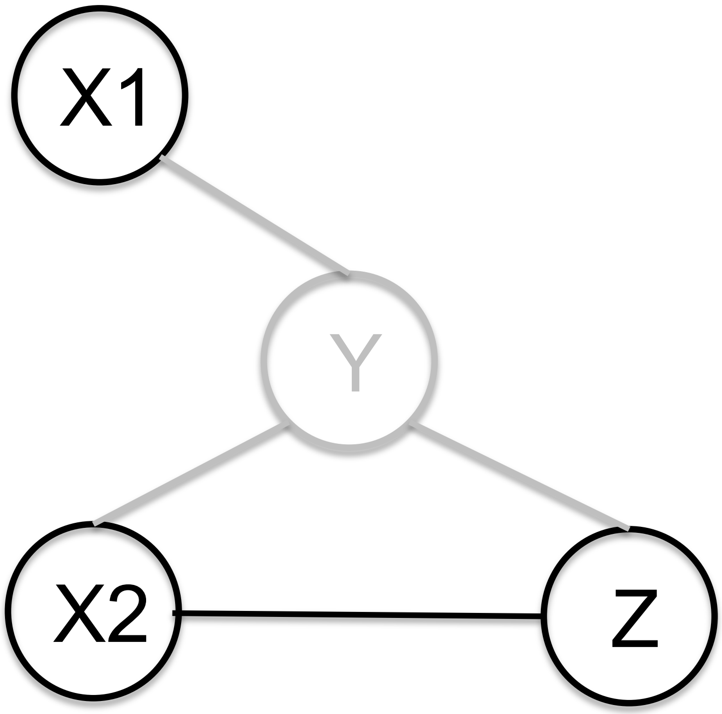


REMOVE DEGREE<3
X1, X2, Z; Y

COLOR IN REVERSE ORDER

Y	R1
Z	R2
X2	R3
X1	R2 or R3

REGISTER ALLOCATION: R1, R2



REMOVE DEGREE<2

X1; spill Y; X2, Z

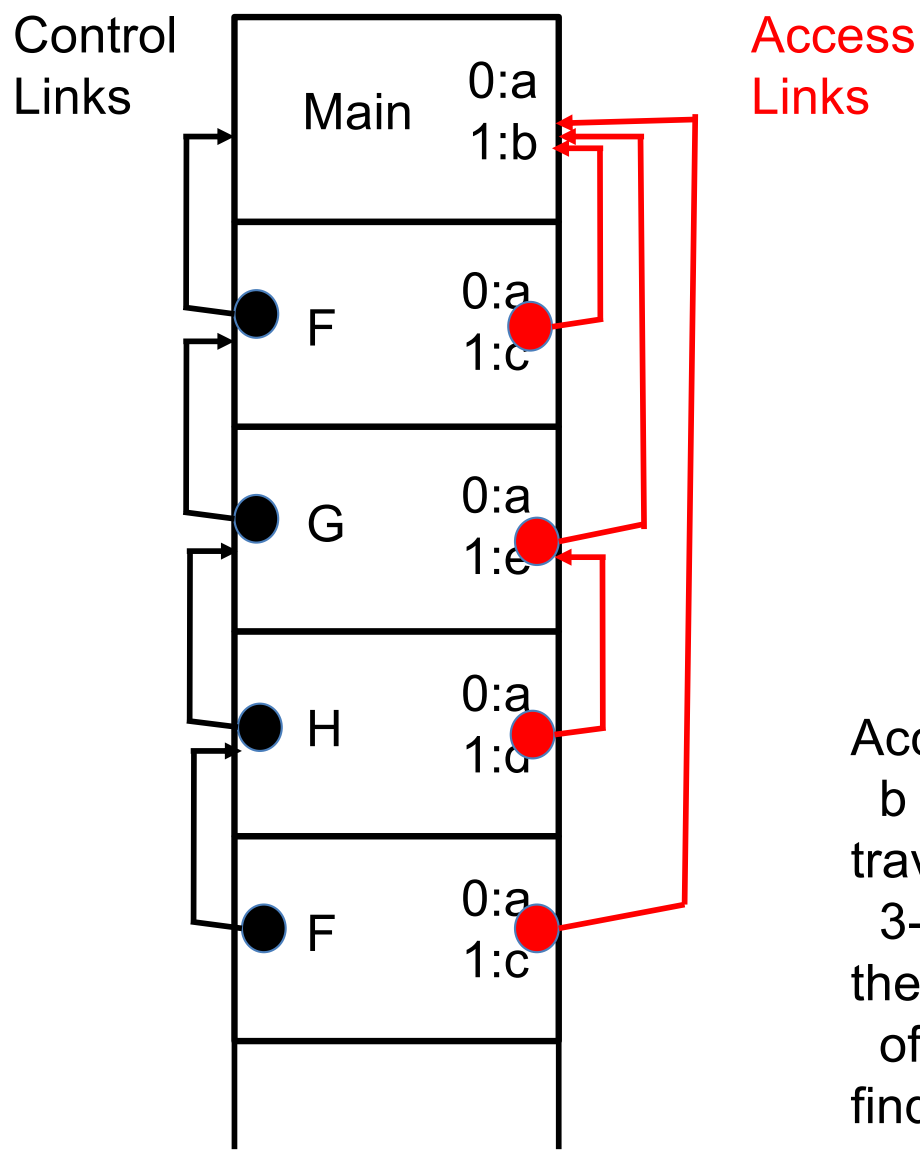
COLOR IN REVERSE ORDER

Z	R1
X2	R2
X1	R1 or R2

Main → F → G → H → F

c-d → 1-1 2-1 2-2 3-1

```
0 Main () {
        Int a, b;

     1 F() {
            Int a, c;
          2 Call G();
       }

     1 G() {
            Int a, e;
          2 H() {
                 Int a, d;

               3 Call F();
            }

          2 Call H();
       }

     1 Call F()
}
```



Access
b in H
traverse
3-1=2 links
then at
offset at of 1
find b

CONSTRUCT	GRAMMAR RELEVANT PRODUCTIONS

```
if x < y then
      <otherstatements>
elseif a > b then
      <otherstatements>
……
elseif c == d then
      <otherstatements>
else
      <otherstatements>
endif
```

```
<S> → if <condt> then <otherstatements> <rest>

<rest> → elseif <condt> then <otherstatements> <rest>
         |  else <otherstatements> endif

<condt> → id relop id
```

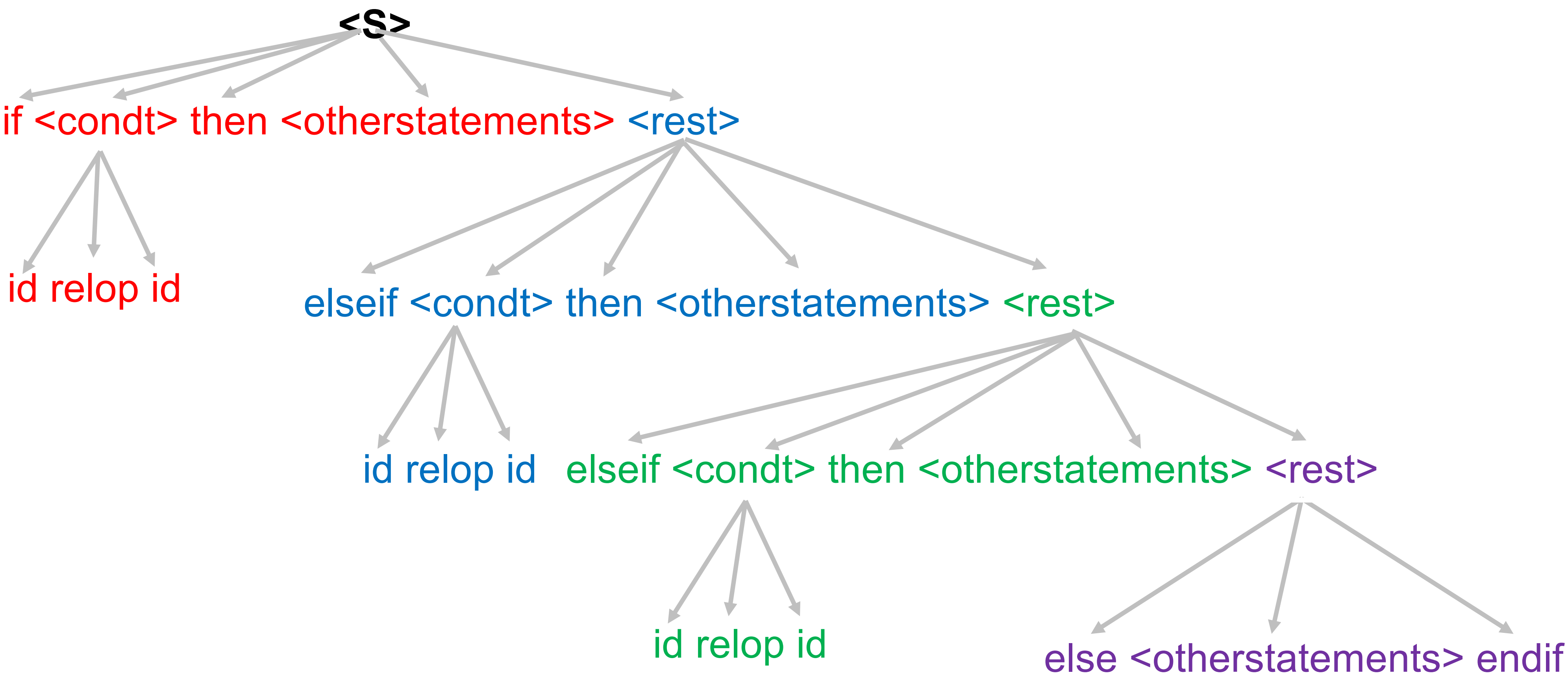
Question:
Provide SEMANTIC RULES that generate code and finally place it in attribute **<S>.code**

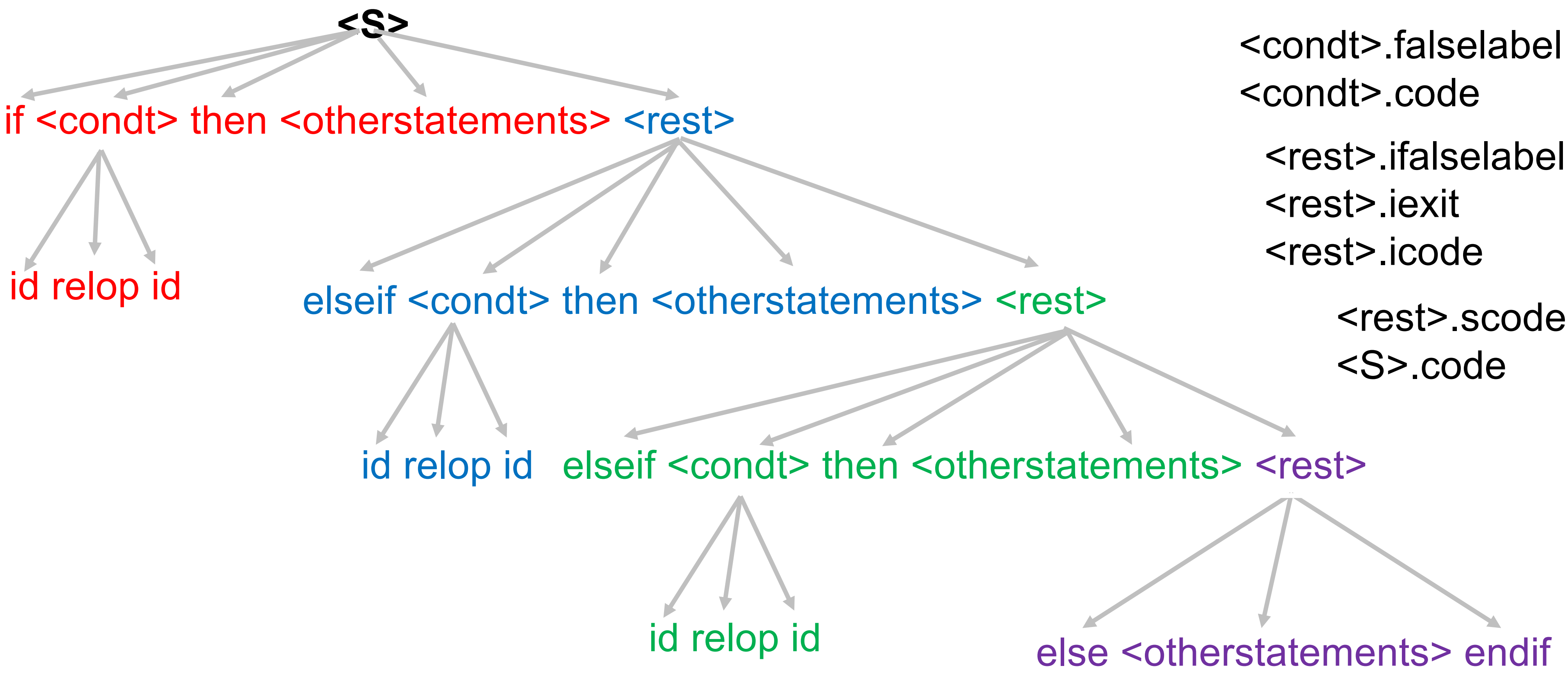
CONSTRUCT

```
if x < y then
      <otherstatements>
elseif a > b then
      <otherstatements>
elseif c == d then
      <otherstatements>
else
      <otherstatements>
endif
……
```

INTERMEDIATE CODE

```
     if x < y go to L1
     go to L2
L1: <otherstatements>
     go to exitL
L2: If a > b go to L3
     go to L4
L3: <otherstatements>
     go to exitL
L4: if c==d go to L5
     go to L6
L5: <otherstatements>
     go to exitL
L6: <otherstatements>
exitL: …….
```


<S>
if <condt> then <otherstatements> <rest>
id relop id
elseif <condt> then <otherstatements> <rest>
id relop id
elseif <condt> then <otherstatements> <rest>
id relop id
else <otherstatements> endif



if x < y go to L1	L2: If a > b go to L3	L4: if c==d go to L5	L6: <otherstatements>
go to L2	go to L4	go to L6	exitL: …….
L1: <otherstatements>	L3: <otherstatements>	L5: <otherstatements>	
go to exitL	go to exitL	go to exitL	
L2: <rest>	L4: <rest>	L6: <rest>	

<S>.code = rcbcgcpc

<rest>.scode = rcbcgcpc

<condt>.falselabel = L2
<condt>.code = ….
<rest>.ifalselabel = L2
<rest>.iexit = exitL
<rest>.icode = rc

<rest>.scode = rcbcgcpc

<condt>.falselabel = L4
<condt>.code = ….
<rest>.ifalselabel = L4
<rest>.iexit = exitL
<rest>.icode = rcbc

<condt>.falselabel = L6
<condt>.code = ….
<rest>.ifalselabel = L6
<rest>.iexit = exitL
<rest>.icode = rcbcgc

<rest>.scode = rcbcgcpc

```
    if x < y go to L1
    go to L2
L1: <otherstatements>
    go to exitL
L2: <rest>
```

```
L2: If a > b go to L3
    go to L4
L3: <otherstatements>
    go to exitL
L4: <rest>
```

```
L4: if c==d go to L5
    go to L6
L5: <otherstatements>
    go to exitL
L6: <rest>
```

```
L6: <otherstatements>
exitL: …….
```

```
<condt> → id1 relop id2   {
            truelabel = newlabel();
            <condt>.falselabel = newlabel();
            <condt>.code = gen("if" id1.place "relop" id2.place "go to" truelabel)
                      || gen("go to" <condt>.falselabel) || gen(truelabel":")
         }


<S> → if <condt> then <otherstatements>
              {
                 <rest>.ifalselabel = <condt>.falselabel;
                 <rest>.iexit = newlabel();
                 <rest>.icode = <condt>.code || <otherstatements>.code ||
                                       gen("go to" <rest>.iexit)
              }
      <rest> { <S>.code = <rest>.scode }
```

```
<rest1> → elseif <condt> then <otherstatements>
            {
              <rest2>.icode = <rest1>.icode || gen(<rest1>.ifalselabel ":") ||
                 <condt>.code || <otherstatements>.code || gen("go to" <rest1>.iexit);
              <rest2>.ifalselabel = <condt>.falselabel;
              <rest2>.iexit = <rest1>.iexit
            }
        <rest2>  { <rest1>.scode = <rest2>.scode }

<rest1> → else <otherstatements> endif
            {
               <rest1>.scode = <rest1>.icode || gen(<rest1>.ifalselabel ":")
                     || <otherstatements>.code || gen(<rest1>.iexit ":")
            }
```