# CS412/CS413

# Introduction to Compilers

Tim Teitelbaum

## Lecture 24: Control Flow Graphs

24 Mar 08

# Optimizations

- Code transformations to improve program
  - Mainly: improve execution time
  - Also: reduce program size
- Can be done at high level or low level
  - E.g., constant folding
- Optimizations must be safe
  - Execution of transformed code must yield same results as the original code for all possible executions

# Optimization Safety

- Safety of code transformations usually requires certain information that may not be explicit in the code
- Example: dead code elimination

```
(1)   x = y + 1;
(2)   y = 2 * z;
(3)   x = y + z;
(4)   z = 1;
(5)   z = x;
```

- What statements are dead and can be removed?

# Optimization Safety

- Safety of code transformations usually requires certain information which may not explicit in the code
- Example: dead code elimination

```
(1)   x = y + 1;
(2)   y = 2 * z;
(3)   x = y + z;
(4)   z = 1;
(5)   z = x;
```

- Need to know whether values assigned to x at (1) is never used later (i.e., x is dead at statement (1))
  - Obvious for this simple example (with no control flow)
  - Not obvious for complex flow of control

# Dead Variable Example

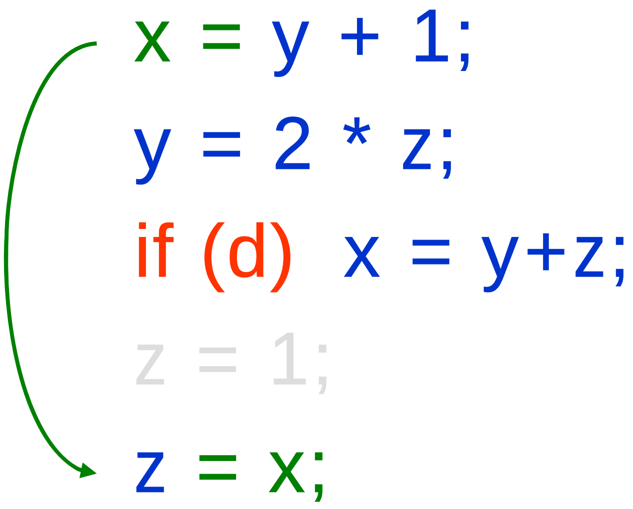
- Add control flow to example:

```
x = y + 1;
y = 2 * z;
if (d)  x = y+z;
z = 1;
z = x;
```

- Is ‘x = y+1’ dead code? Is ‘z = 1’ dead code?

# Dead Variable Example

- Add control flow to example:



```
x = y + 1;
y = 2 * z;
if (d)  x = y+z;
z = 1;
z = x;
```

- Statement x = y+1 is not dead code!
- On some executions, value is used later

# Dead Variable Example

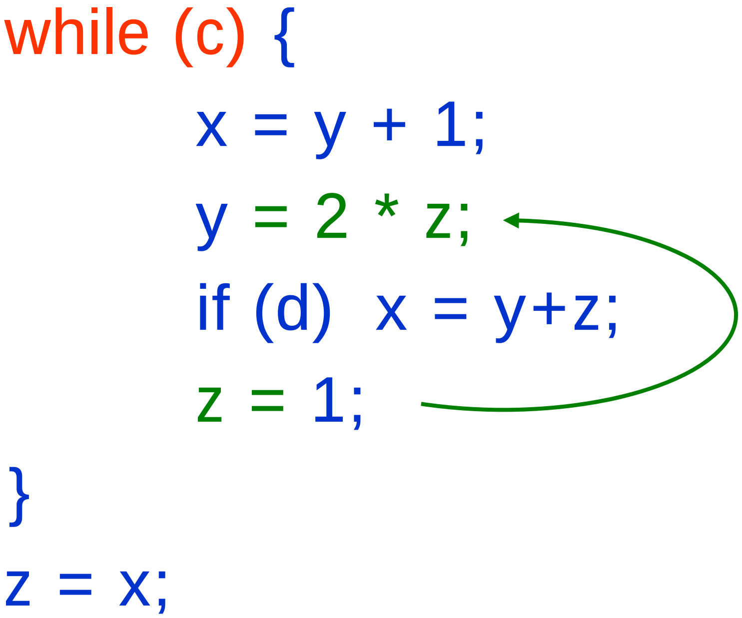
- Add more control flow:

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```

- Is ‘x = y+1’ dead code? Is ‘z = 1’ dead code?

# Dead Variable Example

- Add more control flow:

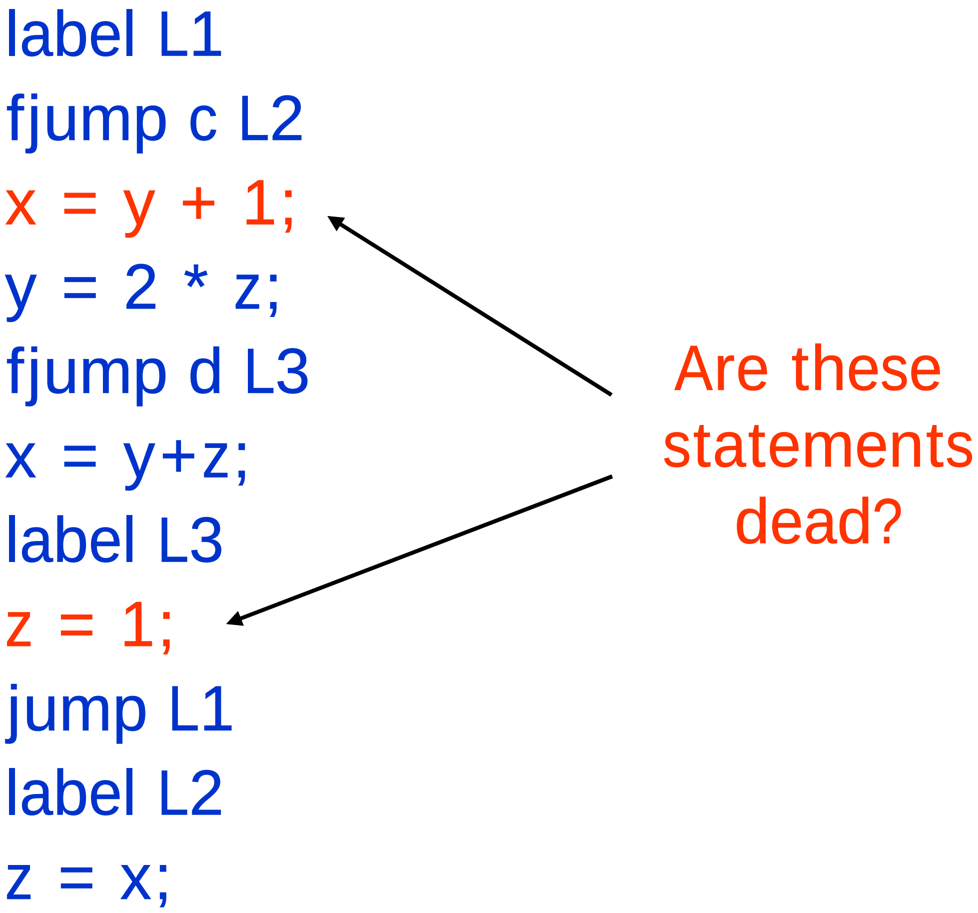


```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```

- Statement ‘x = y+1’ not dead (as before)
- Statement ‘z = 1’ not dead either!
- On some executions, value from ‘z=1’ is used later

# Low-level Code

- Harder to eliminate dead code in low-level code:

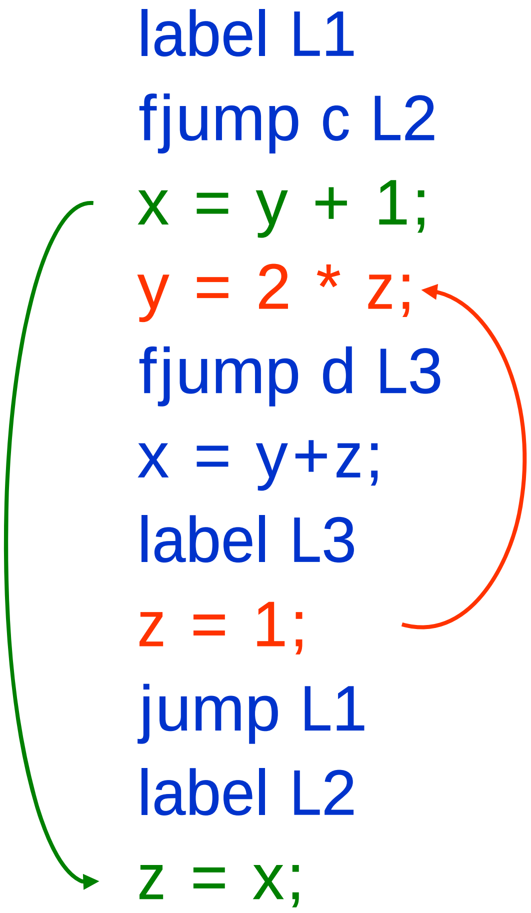


```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3
z = 1;
jump L1
label L2
z = x;
```

Are these statements dead?

# Low-level Code

- Harder to eliminate dead code in low-level code:



```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3
z = 1;
jump L1
label L2
z = x;
```

# Optimizations and Control Flow

- Application of optimizations requires information
  - Dead code elimination: need to know if variables are dead when assigned values
- Required information:
  - Not explicit in the program
  - Must compute it statically (at compile-time)
  - Must characterize all dynamic (run-time) executions
- Control flow makes it hard to extract information
  - Branches and loops in the program
  - Different executions = different branches taken, different number of loop iterations executed

# Control Flow Graphs

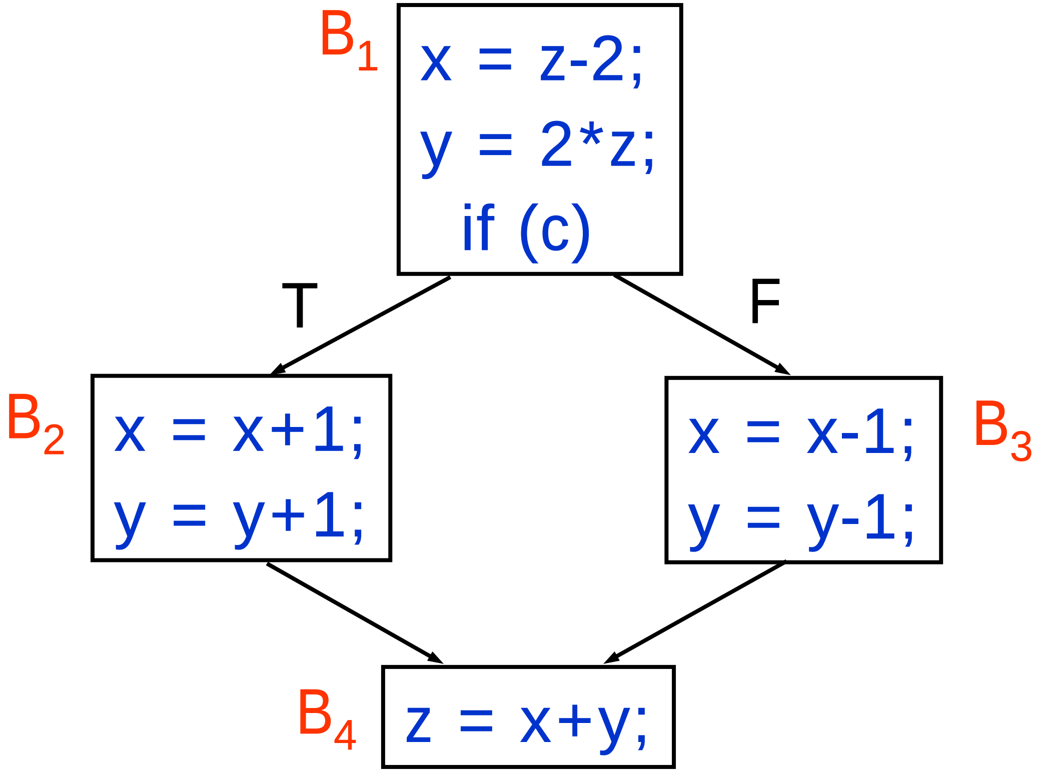
- Control Flow Graph (CFG) = graph representation of computation and control flow in the program
  - framework for static analysis of program control-flow
- Nodes are basic blocks = straight-line, single-entry code, no branching except at end of sequence
- Edges represent possible flow of control from the end of one block to the beginning of the other
  - There may be multiple incoming/outgoing edges for each block

# CFG Example

## Program

```
x = z-2 ;
y = 2*z;
if (c)  {
    x = x+1;
    y = y+1;
}
else {
    x = x-1;
    y = y-1;
}
z = x+y;
```

## Control Flow Graph



$B_1$

```
x = z-2;
y = 2*z;
  if (c)
```

T

F

$B_2$

```
x = x+1;
y = y+1;
```

$B_3$

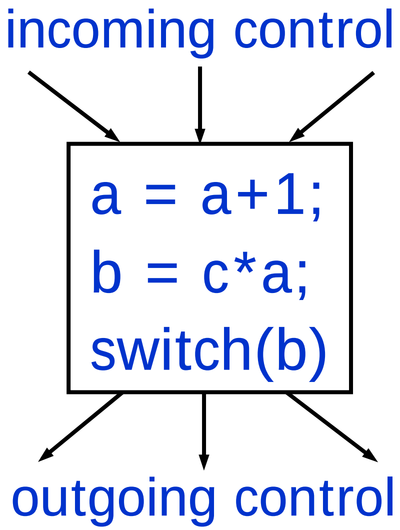
```
x = x-1;
y = y-1;
```

$B_4$

```
z = x+y;
```

# Basic Blocks

- Basic block = sequence of consecutive statements such that:
  - Control enters only at beginning of sequence
  - Control leaves only at end of sequence



incoming control

```
a = a+1;
b = c*a;
switch(b)
```

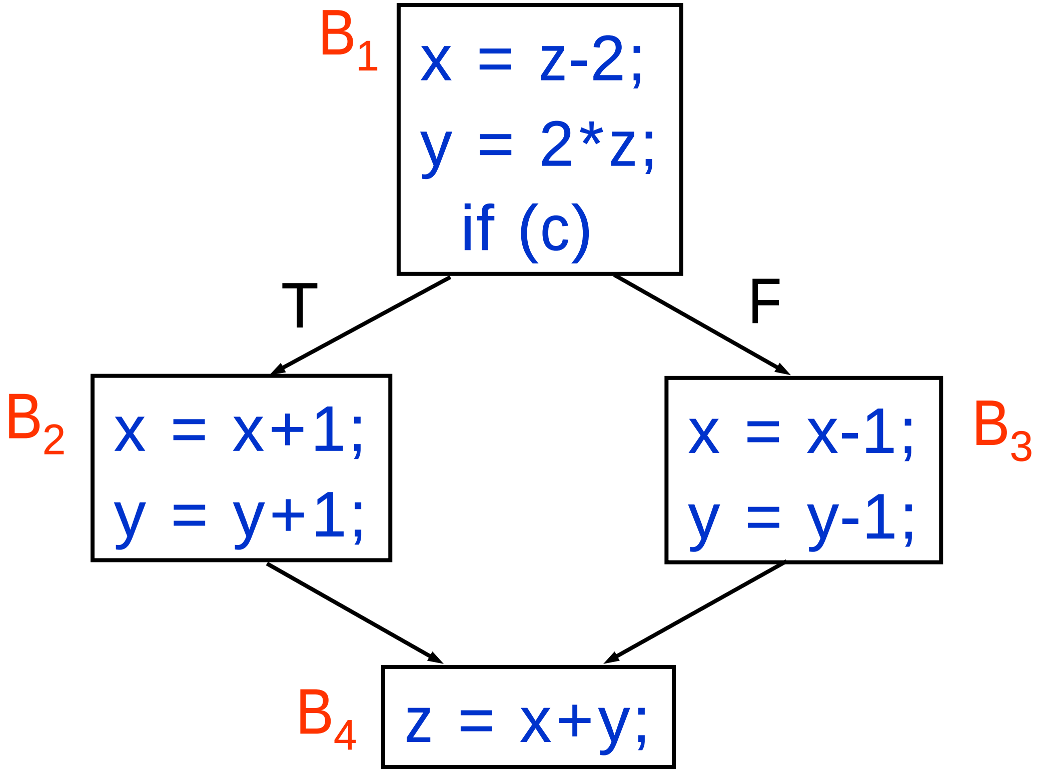
outgoing control

- No branching in or out in the middle of basic blocks

# Computation and Control Flow

- Basic Blocks = Nodes in the graph = computation in the program
- Edges in the graph = control flow in the program

## Control Flow Graph



$B_1$

```
x = z-2;
y = 2*z;
  if (c)
```

$B_2$

```
x = x+1;
y = y+1;
```

$B_3$

```
x = x-1;
y = y-1;
```

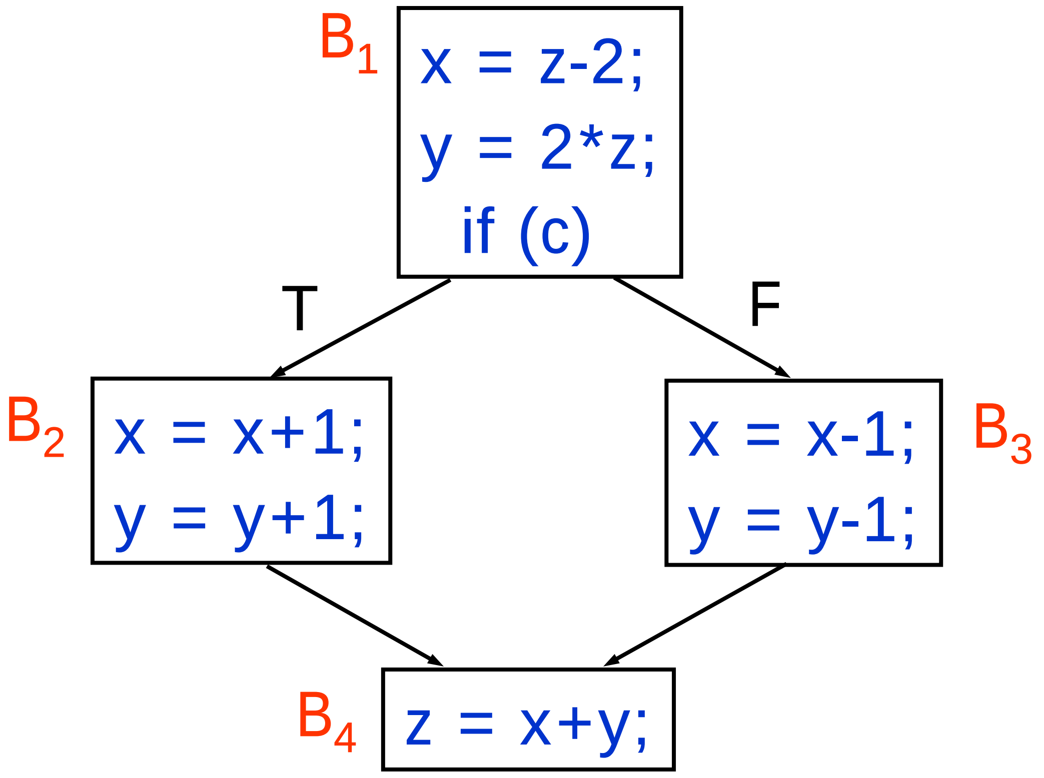
$B_4$

```
z = x+y;
```

# Multiple Program Executions

- CFG models all program executions
- Possible execution = path in the graph
- Multiple paths = multiple possible program executions

## Control Flow Graph



$B_1$

```
x = z-2;
y = 2*z;
  if (c)
```

T

F

$B_2$

```
x = x+1;
y = y+1;
```

$B_3$

```
x = x-1;
y = y-1;
```

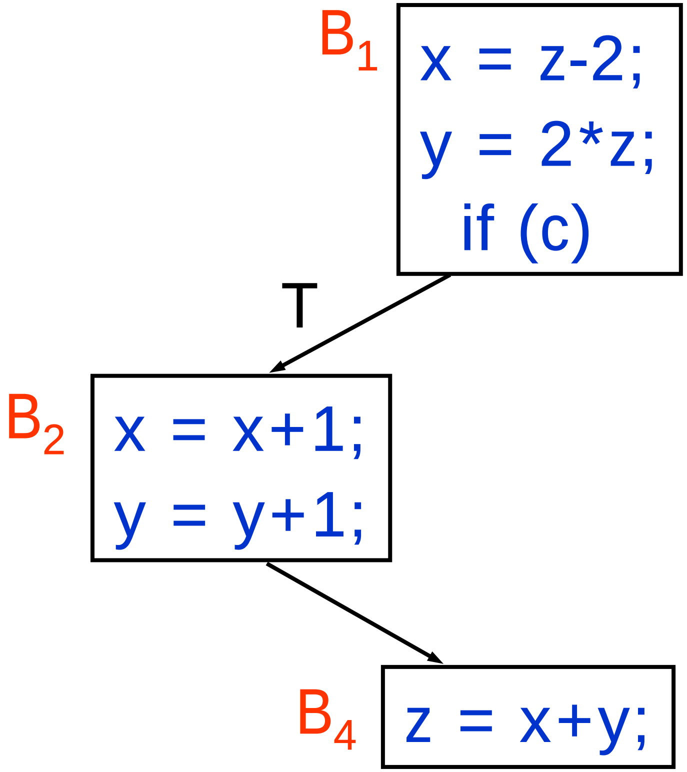
$B_4$

```
z = x+y;
```

# Execution 1

- CFG models all program executions
- Possible execution = path in the graph
- Execution 1:
  - c is true
  - Program executes basic blocks $B_1$, $B_2$, $B_4$

## Control Flow Graph



$B_1$

```
x = z-2;
y = 2*z;
  if (c)
```

T

$B_2$

```
x = x+1;
y = y+1;
```

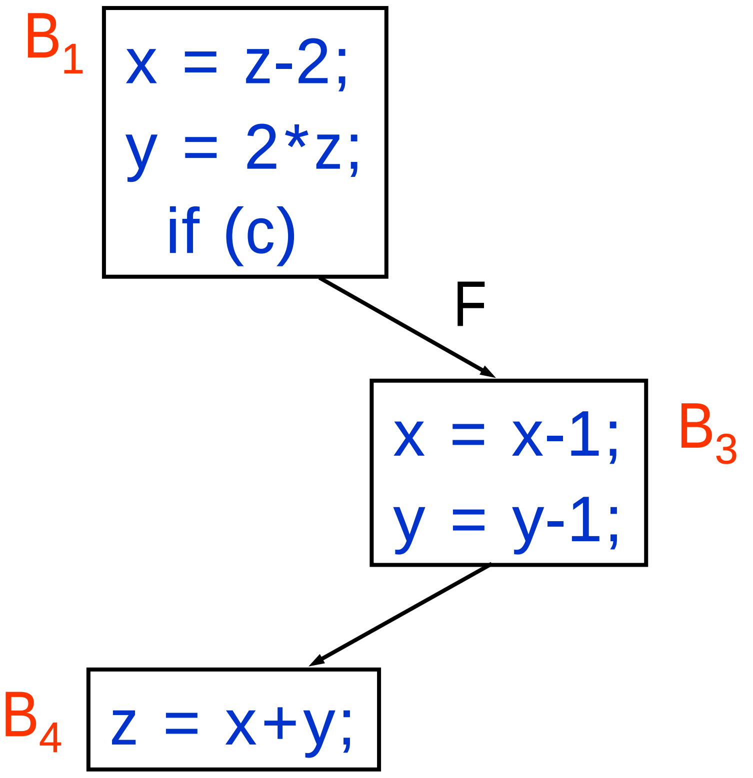
$B_4$

```
z = x+y;
```

# Execution 2

- CFG models all program executions
- Possible execution = path in the graph
- Execution 2:
  - c is false
  - Program executes basic blocks $B_1$, $B_3$, $B_4$

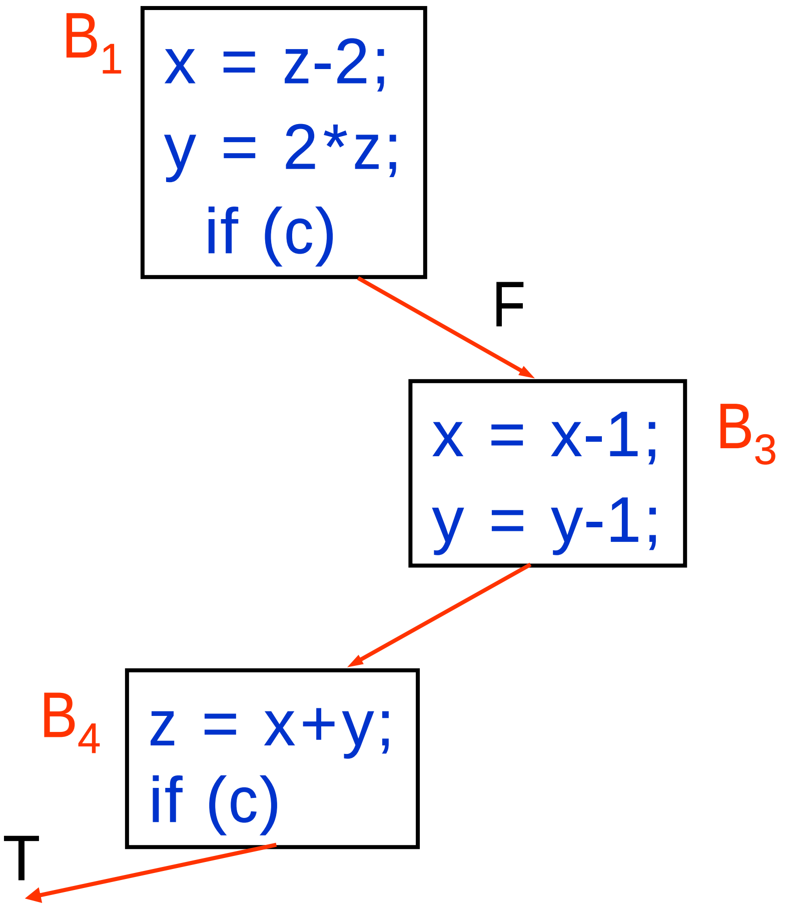
## Control Flow Graph



# Infeasible Executions

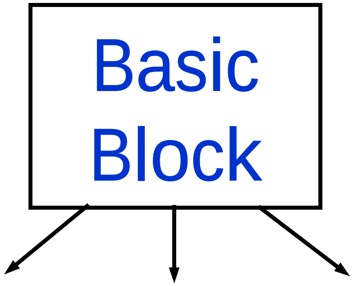
- CFG models all program executions, and then some
- Possible execution = path in the graph
- Execution 2:
  - c is false and true (?!)
  - Program executes basic blocks $B_1$, $B_3$, $B_4$
  - and the T successor of $B_4$

## Control Flow Graph



# Edges Going Out

- Multiple outgoing edges
- Basic block executed next may be one of the successor basic blocks
- Each outgoing edge = outgoing flow of control in some execution of the program

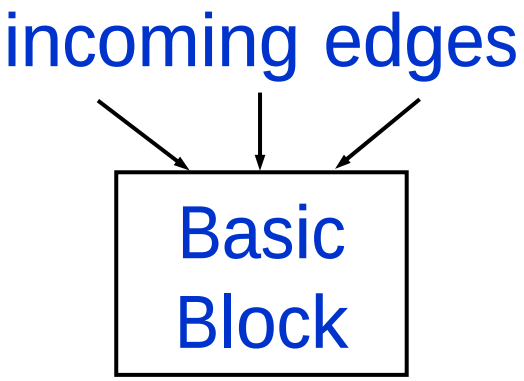


Basic Block

outgoing edges

# Edges Coming In

- Multiple incoming edges
- Control may come from any of the predecessor basic blocks
- Each incoming edge = incoming flow of control in some execution of the program



# Building the CFG

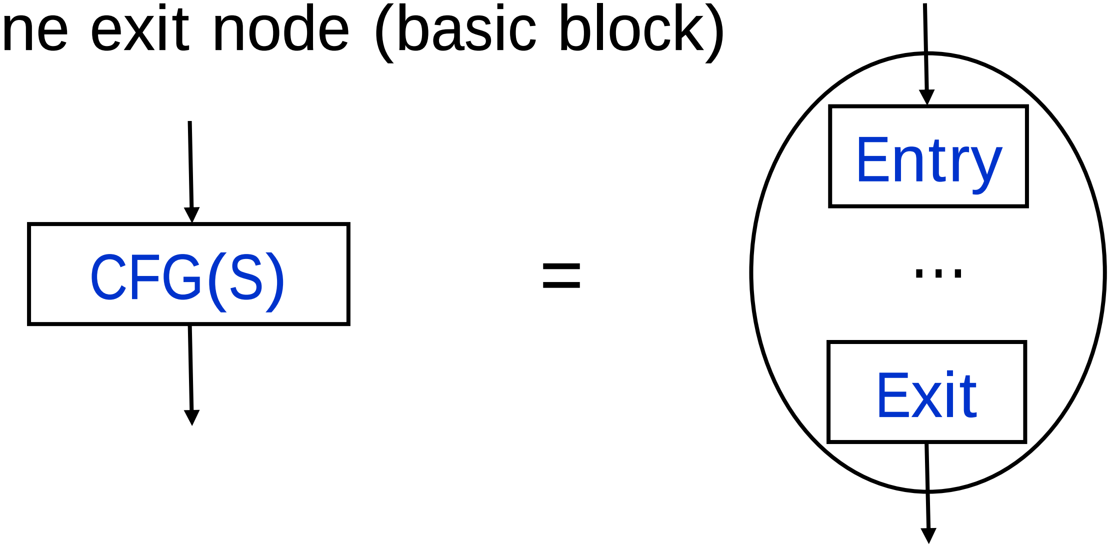
- Can construct CFG for either high-level IR or the low-level IR of the program

- Build CFG for high-level IR
  - Construct CFG for each high-level IR node

- Build CFG for low-level IR
  - Analyze jump and label statements

# CFG for High-level IR

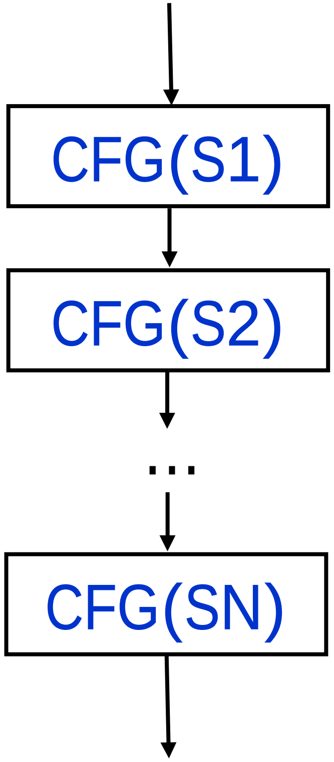
- CFG(S) = flow graph of high-level statement S
- CFG(S) is single-entry, single-exit graph:
  - one entry node (basic block)
  - one exit node (basic block)



- Recursively define CFG(S)

# CFG for Block Statement

- CFG( S1; S2; ...; SN ) =



CFG(S1)

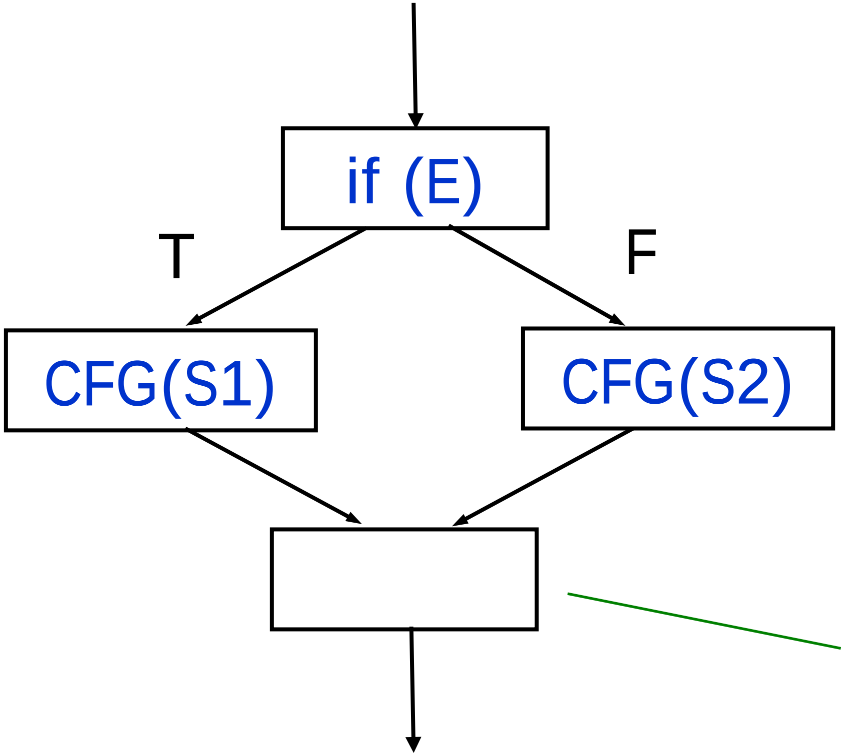
CFG(S2)

…

CFG(SN)

# CFG for If-then-else Statement

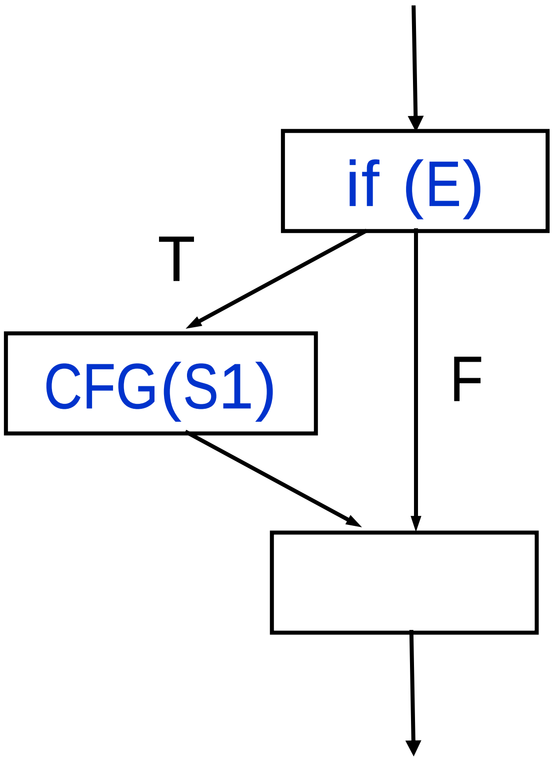
- CFG ( if (E) S1 else S2 )



Empty basic block

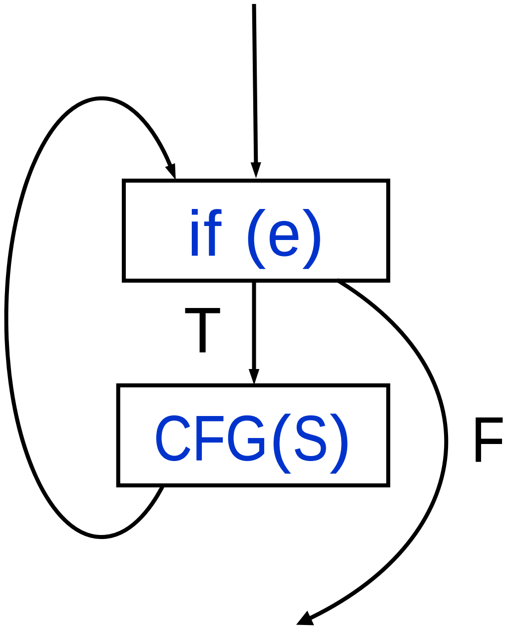
# CFG for If-then Statement

- CFG( if (E) S )



# CFG for While Statement

- CFG for: while (e) S



# Recursive CFG Construction

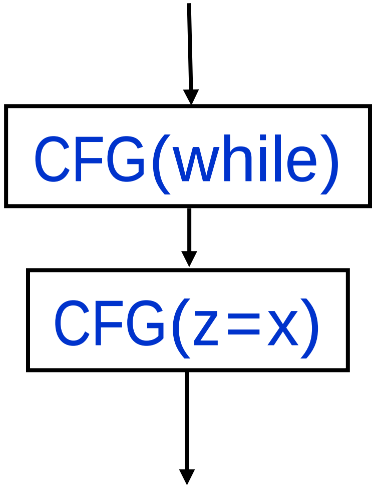
- Nested statements: recursively construct CFG while traversing IR nodes
- Example:

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```

# Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



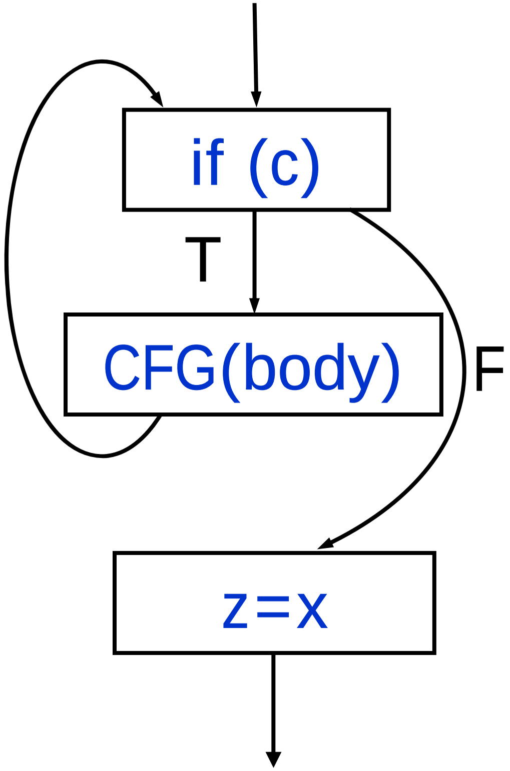
CFG(while)

CFG(z=x)

# Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes

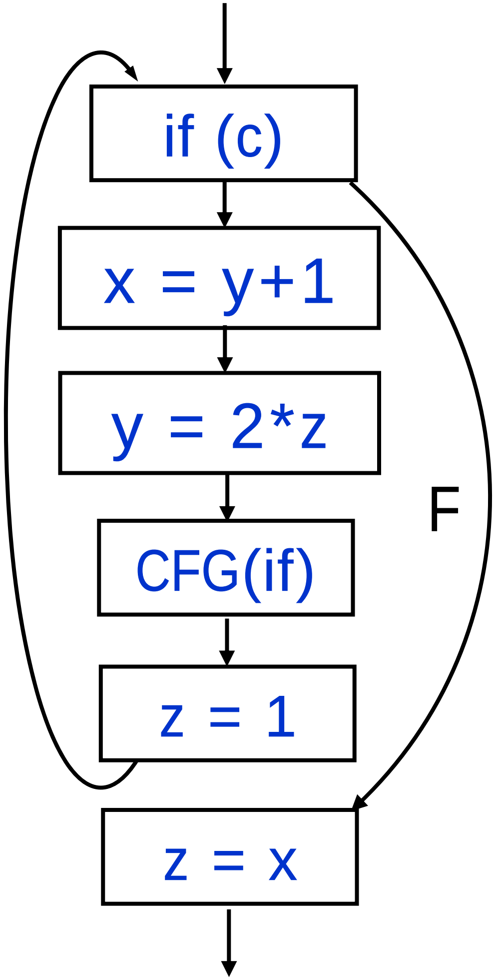
```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



# Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



# Recursive CFG Construction

- Simple algorithm to build CFG
- Generated CFG
  - Each basic block has a single statement
  - There are empty basic blocks

- Small basic blocks = inefficient
  - Small blocks = many nodes in CFG
  - Compiler uses CFG to perform optimization
  - Many nodes in CFG = compiler optimizations will be time- and space-consuming

# Efficient CFG Construction

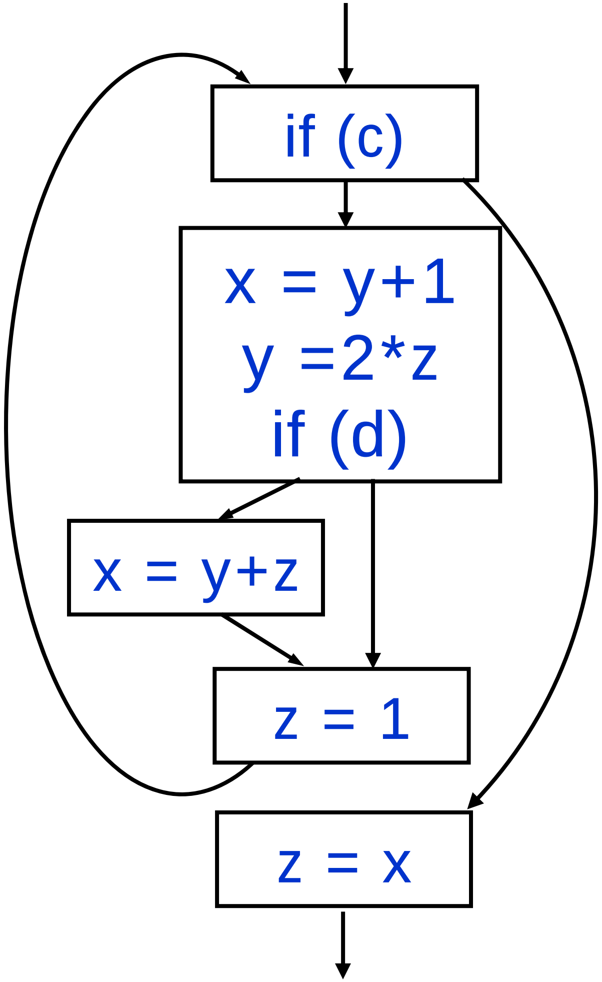
- Basic blocks in CFG:
  - As few as possible
  - As large as possible

- There should be no pair of basic blocks (B1,B2) such that:
  - B2 is a successor of B1
  - B1 has one outgoing edge
  - B2 has one incoming edge
- There should be no empty basic blocks

# Example

- Efficient CFG:

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



# CFG for Low-level IR

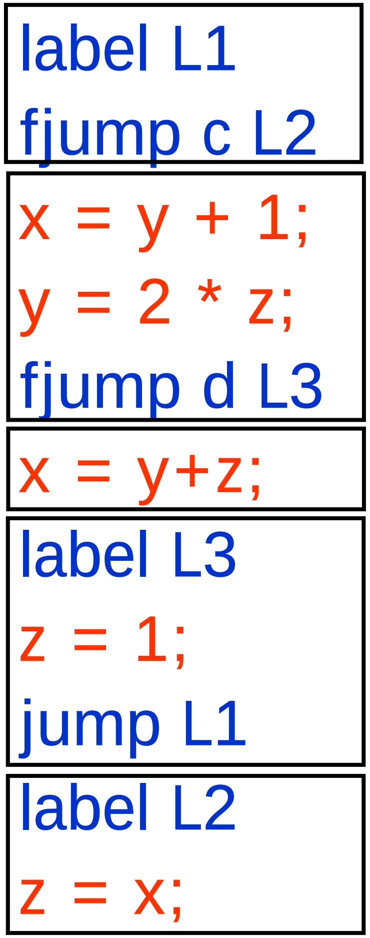
- Identify pre-basic blocks as sequences of:
  - Non-branching instructions
  - Non-label instructions
- No branches (jump) instructions = control doesn't flow out of basic blocks
- No labels instructions = control doesn't flow into blocks

```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3

z = 1;
jump L1
label L2
z = x;
```

# CFG for Low-level IR

- Basic block start:
  - At label instructions
  - After jump instructions
- Basic blocks end:
  - At jump instructions
  - Before label instructions



```
label L1
fjump c L2
```

```
x = y + 1;
y = 2 * z;
fjump d L3
```

```
x = y+z;
```

```
label L3
z = 1;
jump L1
```

```
label L2
z = x;
```

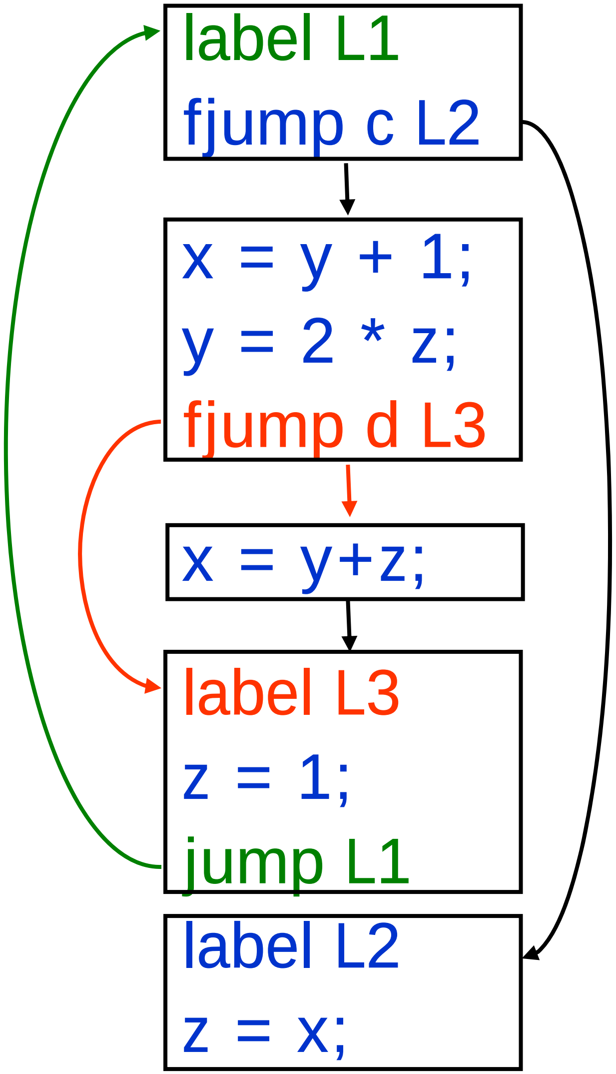
# CFG for Low-level IR

- Conditional jump:

  2 successors

- Unconditional jump:

  1 successor



```
label L1
fjump c L2
```

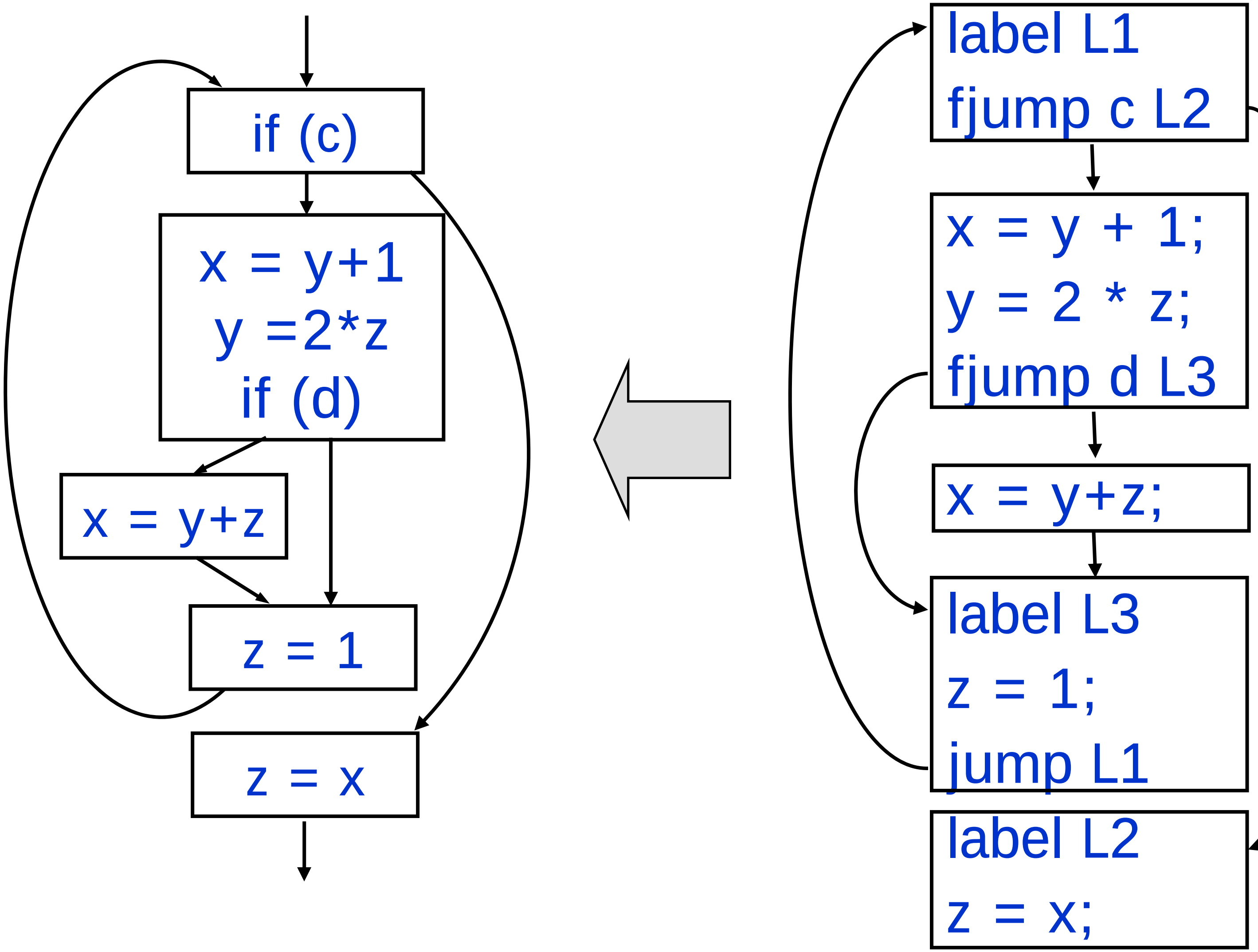
```
x = y + 1;
y = 2 * z;
fjump d L3
```

```
x = y+z;
```

```
label L3
z = 1;
jump L1
```

```
label L2
z = x;
```

# CFG for Low-level IR



if (c)

x = y+1
y =2*z
if (d)

x = y+z

z = 1

z = x

```
label L1
fjump c L2

x = y + 1;
y = 2 * z;
fjump d L3

x = y+z;

label L3
z = 1;
jump L1

label L2
z = x;
```