

COMPILER CONSTRUCTION

Code Analysis & Optimizations I

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2018

Courtesy of CS412/413 Introduction to Compilers, Cornell University. By Prof. Radu Rugina.



Chapter X

Code Analysis and Optimizations

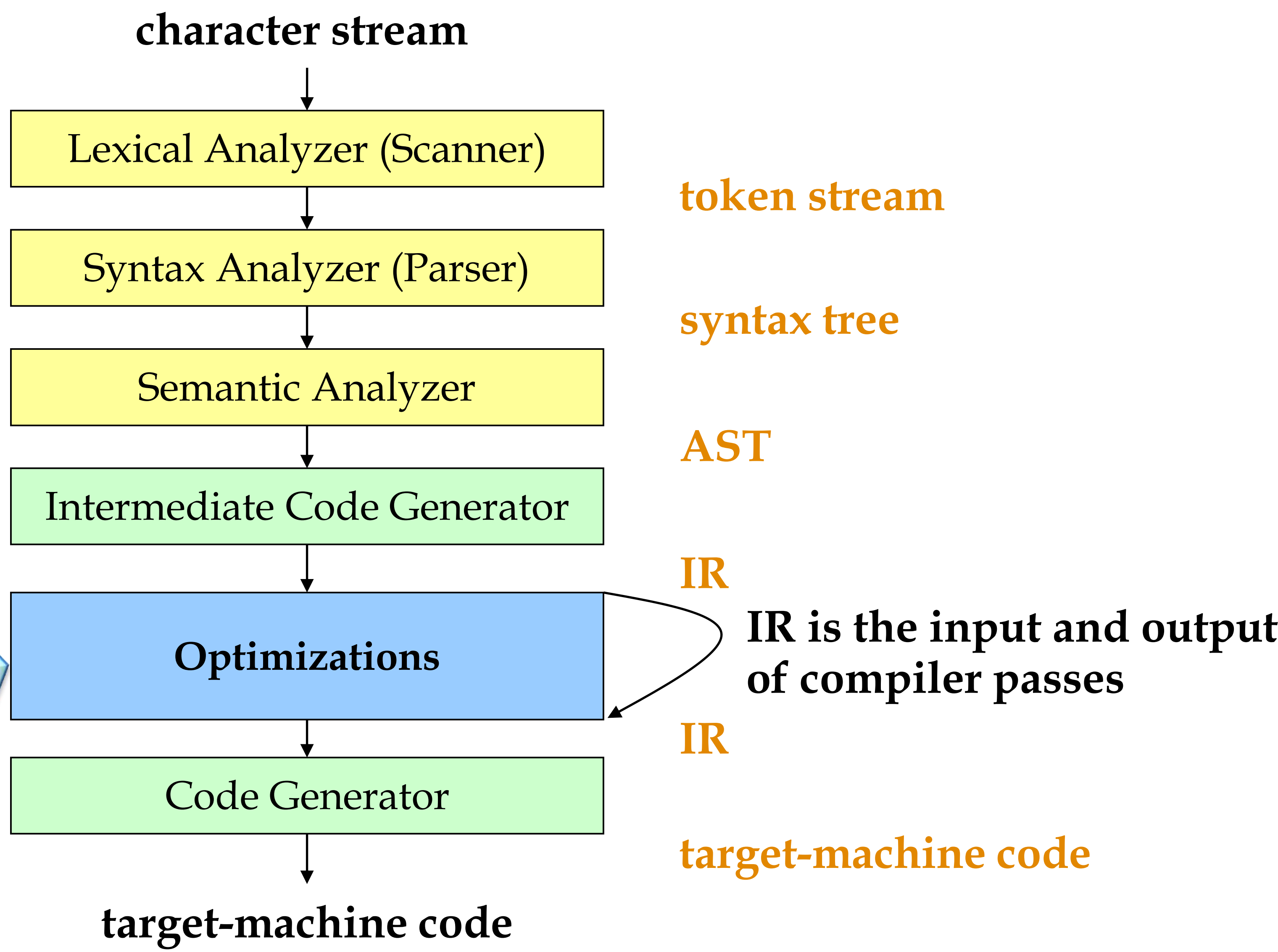
Outline

- Introductions to code optimization techniques
- Control Flow Analysis

- Note:
  - The contents of Code Analysis and Optimizations do not follow the chapter order in the book
  - You can find related materials online to know more about the contents, or
  - refer to the books:
  - Some of the contents are available in the Ch. 14 of the **textbook**
    - Ch. 14 ~ 14.2.1, 14.3.2 (Live Variables), 14.4, and 14.5

  (The information of the following books is listed in **Reference** slide at the end of the file)
  - You may refer to **Advanced Compiler Design & Implementation**
    - Sec. 7.1, 8.1, 8.3, 8.4, and 14.1.3 (Live Variables Analysis)
  - You may refer to the reference book (**Dragon Book**)
    - Sec. 8.4 and 8.5
    - Sec. 9.1, 9.2, 9.3 (optional), and 9.4

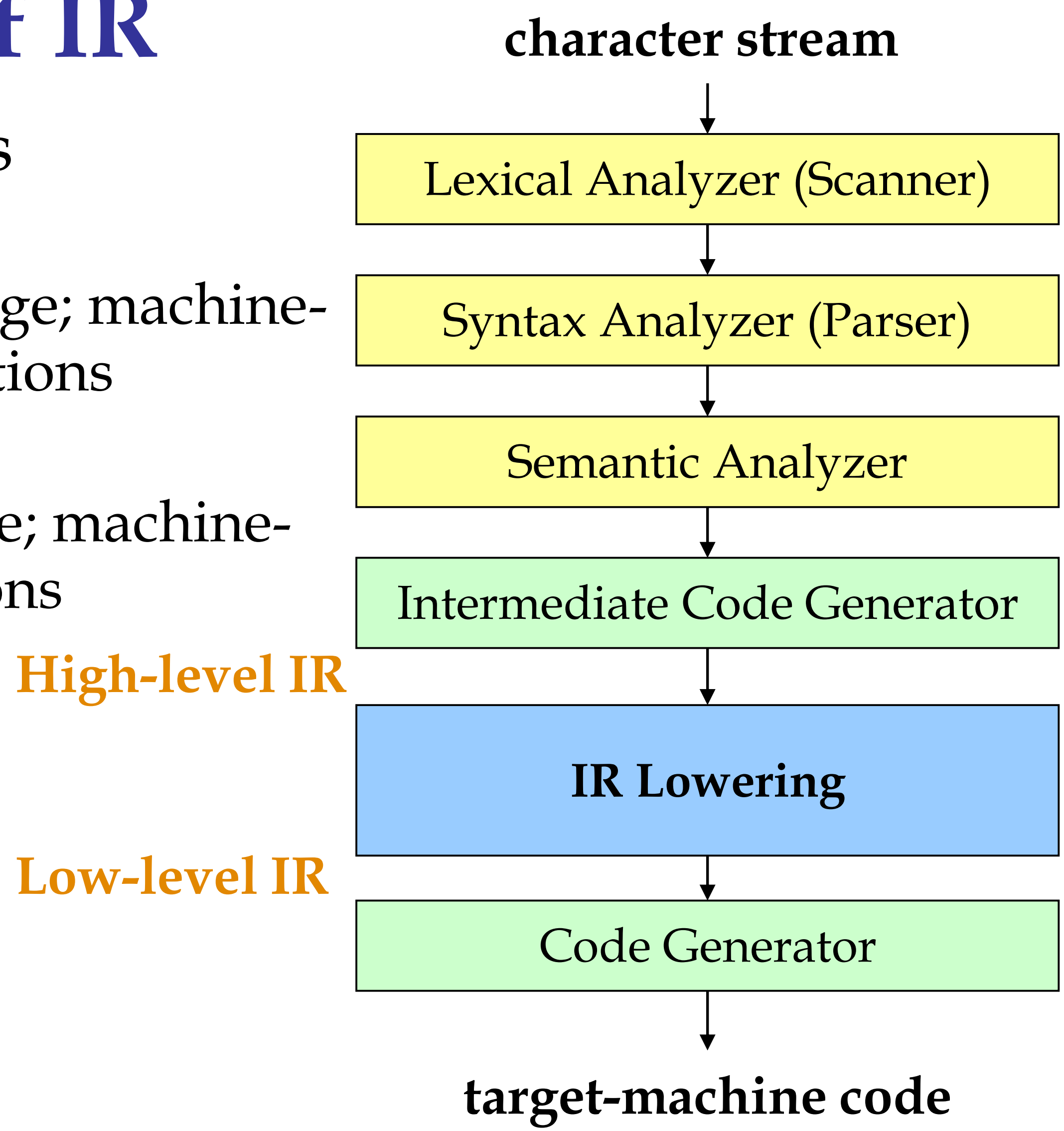
Compiler Optimizations



- Simplified figure
- Opt. are done at different levels of IRs

Another View of IR

- Different levels of IRs
  - **High-level IR**
    Close to source language; machine-independent optimizations
  - **Low-level IR**
    Close to target machine; machine-dependent optimizations



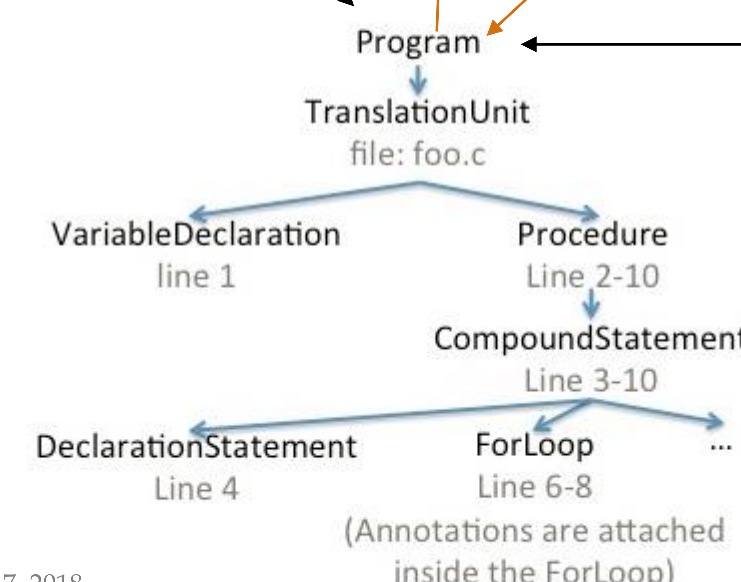
Another View of IR (Cont'd)

Optimizations are done at different level IR(s)

Convert source program to IR

Root of IR structure

```
0  /* file: foo.c */
1    int c = 10;
2    int main(void)
3    {
4       int i, a[100], b[100];
5       #pragma acc parallel loop
6        for( i=0; i<100; i++ ) {
7           a[i] = c*b[i];
8        }
9       ...
10 }
```





What Are the Optimizations?

- Optimizations
  - are referred to as the **code transformations** that improve the execution efficiency of the program (in the current context)
- Execution efficiency may refer to:
  - **Space**: improve memory usage, e.g., smaller size of machine code
  - **Time**: improve execution time, e.g., less running time of the program
  - **Energy**: improve consumed energy, e.g., less energy consumed during the program execution
- Adopt conservative approach for optimizations
  - Transformations must be **safe**,
  - i.e., the program semantics should be preserved &
  - the program will be executed ***correctly***



Why Optimizations?

- High-level languages are more
  - human readable, modular, cleaner,
  - but may not be good for efficient machine execution

- High-level language may make some optimizations inconvenient or impossible to express

- High-level unoptimized code may be more readable: cleaner, modular
  - **int square(x) { return x*x; }**
  - **int mulby2(y) { return 2*y; }**

a. What is the potential problem? What will happen if the functions are called many many times?
b. How to do the optimization?

Tips: a. Overhead of prologue and epilogue code; b. 1. inline. 2. use cheap inst., e.g., shift.

Where to Optimize?

- Goal: improve execution time
- Problem: many optimizations trade off space versus time
- Example: Loop Unrolling

```
/* Copy 20 elements */
for (i=0; i<20; i++)
{
    a[i] = b[i];
}
```

Unrolled

```
/* Unrolled four times */
for (i=0; i<20; i+=4)
{
    a[i] = b[i];
    a[i+1] = b[i+1];
    a[i+2] = b[i+2];
    a[i+3] = b[i+3];
}
```



Where to Optimize? (Cont'd)

- Goal: improve execution time
- Problem: many optimizations trade off space versus time
- Example: Loop Unrolling
  - Increase code size and speed up one loop
  - Frequently executed code with long loops: space/time tradeoff is generally a win
  - Infrequently executed code: may want to optimize code space at expense of time
  - Want to optimize program **hot spots**
    - which requires performance analysis beforehand



Many Possible Optimizations

- Many ways to improve program efficiency
- Some of the most common optimizations:
  a. Function inlining
  b. Function cloning
  c. Constant folding
  d. Constant propagation
  e. Dead code elimination
  f. Loop-invariant code motion
  g. Common sub-expression elimination
  h. Data prefetching
  i. Loop unrolling
  j. … YOU NAME IT
- A quick question:
  - Is the order of the optimizations affecting the performance?
  - E.g., **$\text{execution\_time}_{prog}(a+b+c+d)$ =?= $\text{execution\_time}_{prog}(d+c+b+a)$**

Tips: It depends, but most of time the answer is YES.



Constant Propagation

- Replace use of variable with constant
  - if value of variable is known to be a constant
- Example
  - `n = 10`
  - `c = 2`
  - `for (i=0; i<n; i++) { s = s+i*c; }`
- Replace n, c:
  - `for (i=0; i<10; i++) { s = s+i*2; }`
- Each variable must be replaced only when it has known **constant value**:
  - Forward from a constant assignment
  - Until **next assignment of the variable**
  - that requires another program analysis to know exact locations at which a variable's *uses* and *defines* occur

Constant Folding

- Transform the computation expression into the corresponding constant
  - Evaluate an expression if operands are known at compile time, i.e., they are constants
- Example
  - x = 1.1 * 2; => x = 2.2;
- Performed at every state of compilation
  - E.g., Constants are created during translations or optimizations
  - int x = a[2];

Low level code

```
=> t1 = 2*4;
   t2 = a + t1;
   x = *t2
```

Applied CF on the low level code

```
=> t1 = 8;
   t2 = a + t1;
   x = *t2
```

Can we apply further opt. on the low-level code?

Tips: Yes! Constant propagation could be further applied on the resulting code.

Algebraic Simplification

- More general form of constant folding: take advantage of usual simplification rules

```
a * 1  => a          a * 0  => 0
a / 1  => a          a + 0  => a
b || false => b      b || true => b
```

- Repeatedly apply the above rules
  - (y*1+0)/1 => y*1+0 => y*1 => y

- Must be careful with floating point!!!



Copy Propagation

- Replace uses of **x** with **y**
  - after seeing assignment **x = y**

```
x = y;
if (x > 1)
    s = x * f(x - 1);
```

```
x = y;
if (y > 1)
    s = y * f(y - 1);
```

- What if there was an assignment **y = z** before?
  - Transitively apply replacements
  - You might like to find out how!

Common Subexpression Elimination

- Reuse the computed value
  - instead of computing the same expression multiple times
- Example:

```
a = b + c;          a = b + c;
c  = b + c;   =>    c  = a;
d = b + c;          d = b + c;
```



Unreachable Code Elimination

- Eliminate code which is never executed
- Example:

```
#define debug false
s = 1;
if (debug)
    printf("state = %d.", s);
```

→

```
s = 1;
```

- Unreachable code may not be obvious in low IR
  - Or, in high-level languages with unstructured "**goto**" statements



Unreachable Code Elimination (Cont'd)

- Unreachable code in **while/if** statements when:
  - Loop condition is always false (loop never executed)
  - Condition of an if statement is always true or always false (only one branch executed)
- Example:

```
if (false) S;             => ;
if (true) S; else S';     => S;
if (false) S; else S';    => S';
while (false) S;          => ;
while(2>3) S;             => ;
```



Dead Code Elimination

- Eliminate the statement
  - if effect of the statement is never observed

```
x = y + 1;
y = 1;
x = 2 * z;
```

```
x = y + 1;
y = 1;
x = 2 * z;
```

- Variable is dead if never used after definition
- Eliminate assignments to dead variables
- Other optimizations may create dead code



Loop Optimizations

- Program hot spots are usually loops
  - There are exceptions: OS kernels, compilers
- Most execution time in most programs is spent in loops: 90/10 is typical
  - High performance computing applications are good examples
- Loop optimizations are important, effective and numerous



Loop-invariant Code Motion

- Hoist the statement out of the loop
  - if result of the statement or expression does not change among loop iterations, and
  - it has no externally-visible side-effect
- Often useful for array element addressing computations
  - which are sometimes invariant code and not visible at source level
- Require analysis to identify loop-invariant expressions

Code Motion Example I

- Identify invariant expression:

```
for (i=0; i<n; i++)
    a[i] = a[i] + (x*x)/(y*y)
```

- Hoist the expression out of the loop:

```
c = (x*x)/(y*y)
for (i=0; i<n; i++)
    a[i] = a[i] + c
```



Code Motion Example II

- Can also hoist statements out of loops
- Assume x not updated in the loop body:

```
...
while (...) {
    y = x*x;
    ...
}
...
```

```
...
y = x*x;
while (...) {

    ...
}
...
```

- Is it a safe transformation?

Tips: Might be not! Why? y's scope is changed.

Strength Reduction

- Replaces expensive operations by cheap ones
  - E.g., using adds/substracts instead of multiplies/divides
- Strength reduction more effective in loops
- Induction variable
  - is a loop variable whose value is depending linearly on the iteration number
- Apply strength reduction into induction variables

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

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



Strength Reduction (Cont'd)

- Can apply strength reduction to computation other than induction variables:

x * 2	→	x + x
$i * 2^c$	→	i << c
$i / 2^c$	→	i >> c

Induction Variable Elimination

- If there are multiple induction variables in a loop
  - eliminate the ones which are used only in the test condition
- Need to rewrite test using the other induction variables
- Usually applied after strength reduction

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

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

Loop Unrolling

- Execute loop body multiple times at each iteration
- Example:
  - `for (i=0; i<n; i++) { S; }`
- Unroll loop four times:
  - `for (i=0; i<n; i+=4) { S; S; S; S;}`

- Space-time tradeoff: program size increases
- Get rid of ¾ of conditional branches!!!
- Open the door for more aggressive optimizations



Function Inlining

- Replace a function call with the function body

```
int g (int x) {
    return f(x) – 1;
}
int f (int n) {
    int b = 1;
    while (n--) { b = 2*b; }
    return b;
}
```

```
int g (int x) {
    int r;
    int n = x;
    {
        int b = 1;
        while (n--) { b = 2*b; }
        r = b;
    }
    return r -1;
}
```



Function Cloning

- Create specialized versions of functions
  - that are called from different call sites with different arguments

```
void f (int x[], int n, int m) {
    for (int i = 0; i<n; i++) { x[i] = x[i] + i * m; }
}
```

- For a call **f(a, 10, 1)**, create a specialized version of **f**:

```
void f' (int x[]) {
    for (int i = 0; i<10; i++) { x[i] = x[i] + i; }
}
```

- For another call **f(b, p, 0)**, create another version **f''**



When to Apply Optimizations

IR Levels

- High level IR
- Low level IR
- Assembly

Optimizations applied

- Function inlining
- Function cloning
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength reduction
- Constant folding & propagation
- Branch prediction/optimization
- Loop unrolling
- Register allocation
- Cache optimization



Summary

- Many useful optimizations that can transform code to make it faster
- Whole is greater than sum of parts
  - Optimizations should be applied together, sometimes more than once, at different levels
- Program transformations should be performed safely



Optimization Safety

- Optimizations must be safe
  - Execution of transformed code must **yield same results** as the original code for **all possible executions**
- Safety of code transformations
  - usually requires certain information which may be not explicit in the code
- Example: dead code elimination
  - **Improper code (statements) elimination will lead to compilation error or execution error**



Optimization Safety (Cont'd)

- Example: dead code elimination

  1. x = y + 1;
  2. y = 2 * z;
  3. x = y + z;
  4. z = 1;
  5. z = x;

- Which statements are dead and can be removed safely?

Optimization Safety (Cont'd)

- Example: dead code elimination
  a) x = y + 1;
  b) **y = 2 * z;**
  c) **x = y + z;**
  d) z = 1;
  e) **z = x;**
- Need to know what value assigned to x at **a)** is never used later
  - I.e., **x** is dead at statement **a)**
  - Obvious for this simple example (with no control flow)
  - Not obvious for complex flow of control



Dead Code Example

- Add control flow to the previous example

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

- Now, what is dead code?
  - x = y + 1;
  - z = 1;

Dead Code Example (Cont'd)

- Add control flow to the previous example

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

- Statement is **x = y + 1;** not dead code now!!!
- On some executions, value (of **x**) is used later

Dead Code Example (Cont'd)

- Add more control flow to the previous example:

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

- Now, what is dead code?
  - x = y + 1;
  - z = 1;

Dead Code Example (Cont'd)

- Add more control flow to the previous example:

```
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 (i.e., next iteration)

Low-level Code

- Much 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
```

Now, are the two statements dead?

fjump is a conditional jump instruction

Low-level Code (Cont'd)

- Much 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
```

It is harder to analyze flow of control in low-level code



Optimizations vs. Control Flow

- Application of optimizations requires information
  - Dead code elimination: need to know if variables are dead when assigned values
  - It gets harder when the low-level code with ***goto***s
- 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



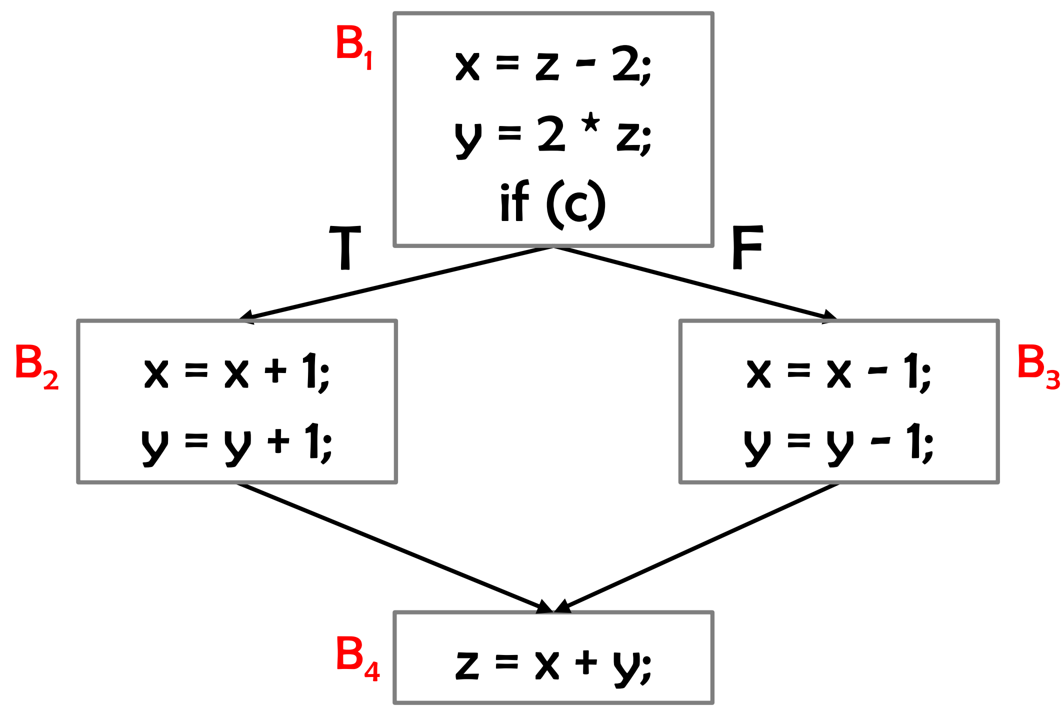
Control Flow Graphs

- Control Flow Graph (CFG)
  - Graph representation of computation and control flow in the program
  - Framework to statically analyze program control-flow
- Nodes are **Basic Blocks** **(BBs)**
  - each of which contains sequences of consecutive non-branching statements
- 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

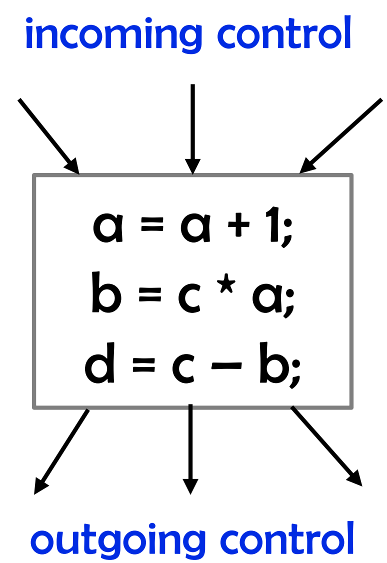
- Program and its CFG

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



Basic Blocks

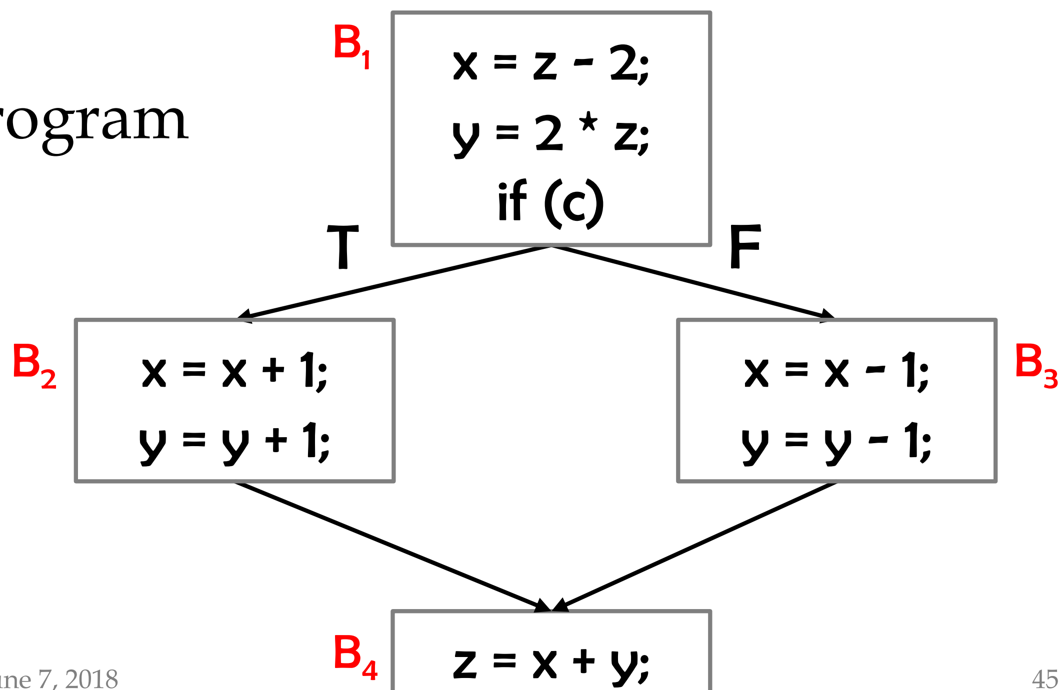
- **Basic block**
  - Sequence of consecutive statements such that:
  - control enters only at beginning of sequence, and
  - control leaves only at the end of sequence
- **No branching in or out in the middle of basic blocks**





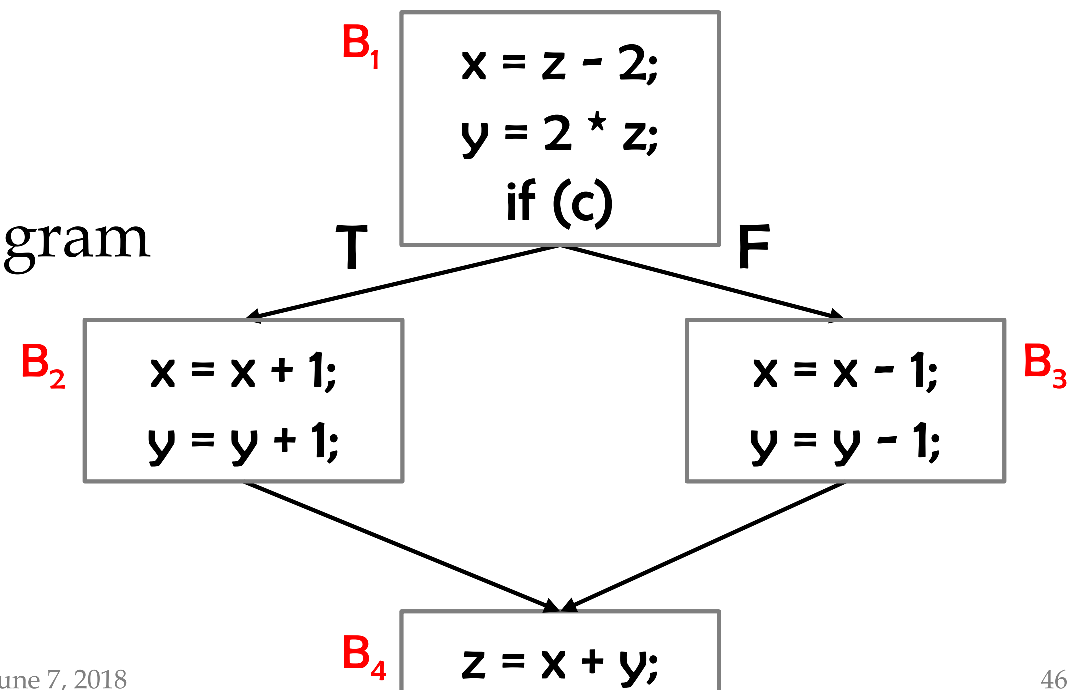
Computation and Control Flow

- Basic Blocks
  - → Nodes in the CFG
  - → Computation in the program
- Edges in the graph
  - → Control flow in the program



Multiple Program Executions

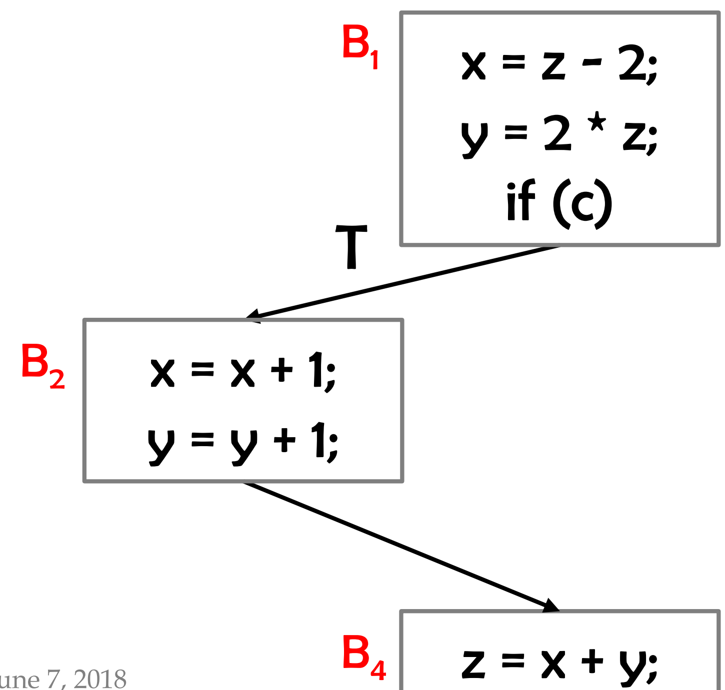
- CFG models all program executions
- Possible execution
  → A path in the graph
- Multiple paths
  → Multiple possible program executions





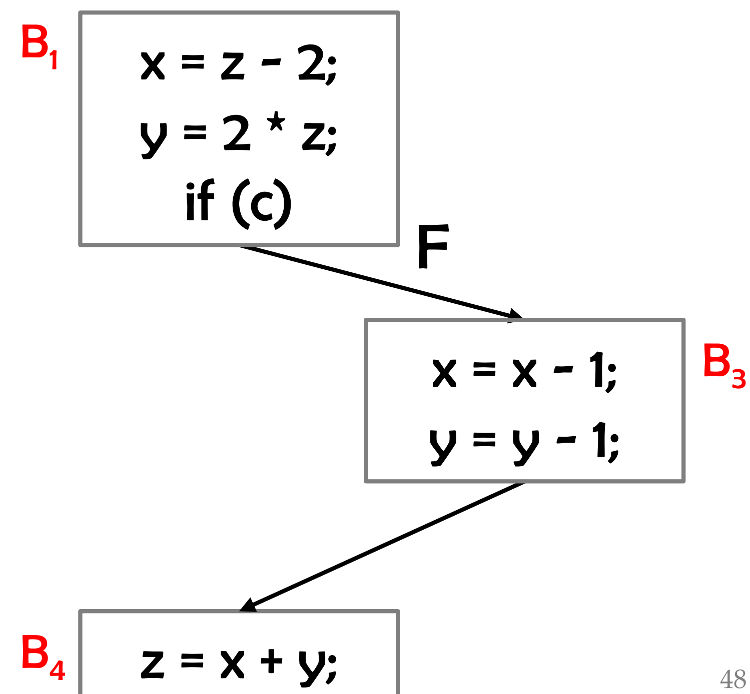
Execution 1

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



Execution 2

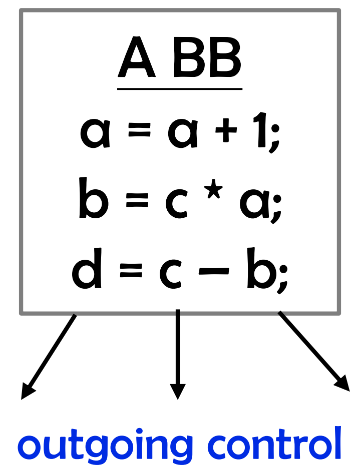
- CFG models all program executions
- Possible execution
  → A path in the graph
- Execution 2:
  - c is true
  - Program executes basic blocks, B_1, B_3, and B_4



Edges Going Out

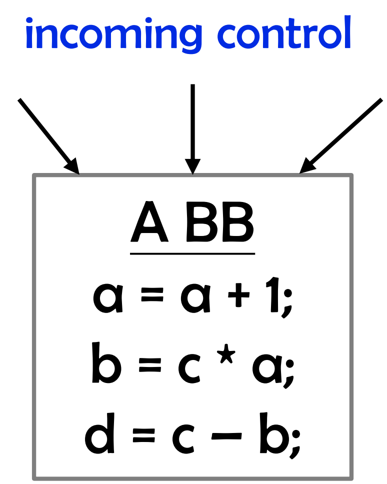
- Multiple outgoing edges
- Basic block executed next **may be** one of the successor basic blocks
- Each outgoing edge

  → Outgoing flow of control in **some execution** of the program



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



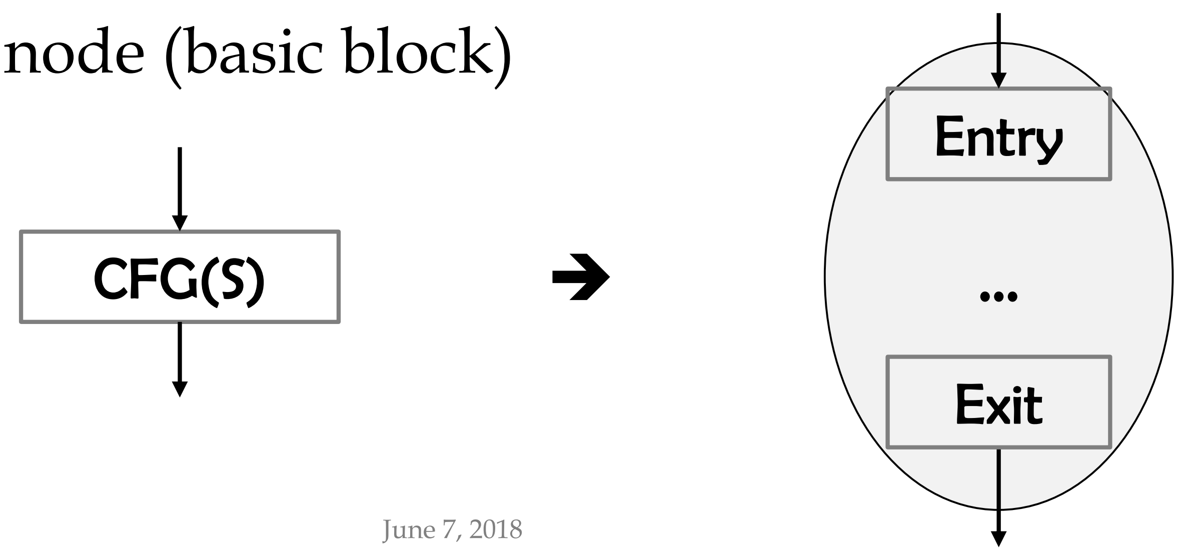
Build the Control Flow Graph

- During the compilation process
  - compiler represents the program using either High IR or Low IR
  - Can construct CFG for either of the two intermediate representations

- Build CFG for High IR
  - Construct CFG for each High IR node
- Build CFG for Low IR
  - Analyze jump and label statements

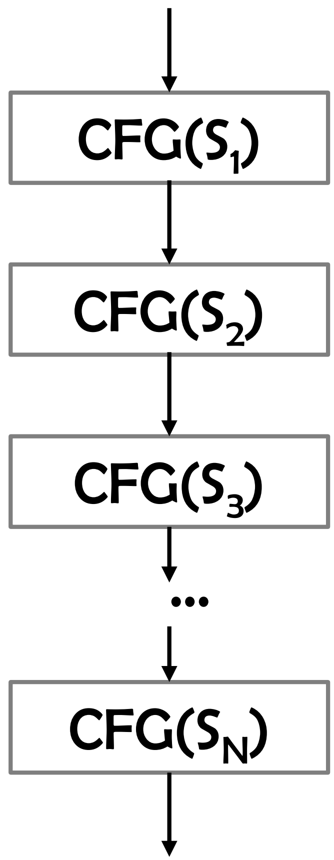
CFG for High-level IR

- CFG(**S**)
  - Flow graph of a high-level statement *S*
- CFG(**S**) is single-entry, single-exit graph
  - One entry node (basic block)
  - One exit node (basic block)



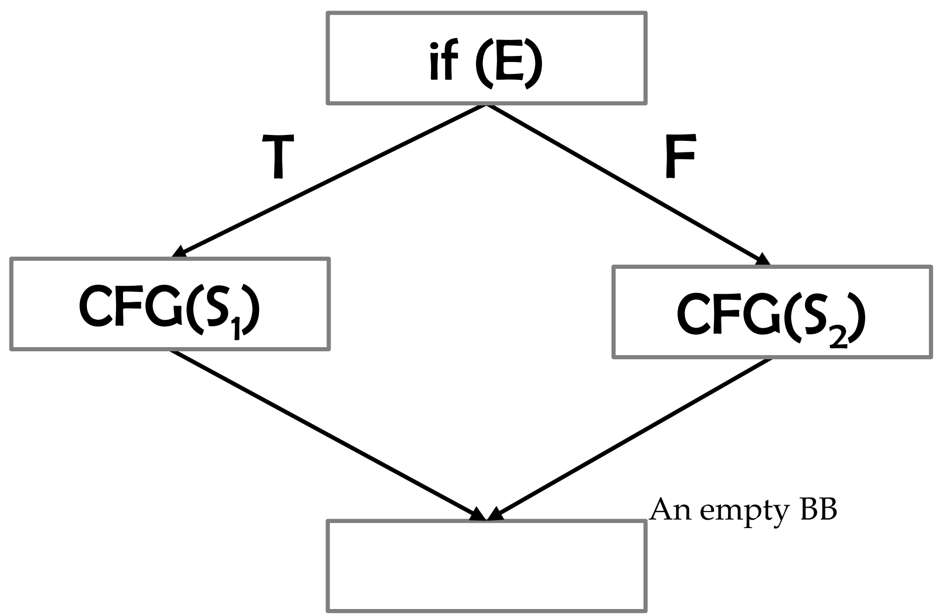
CFG for Block Statement

- CFG (S_1; S_2; …; S_N)



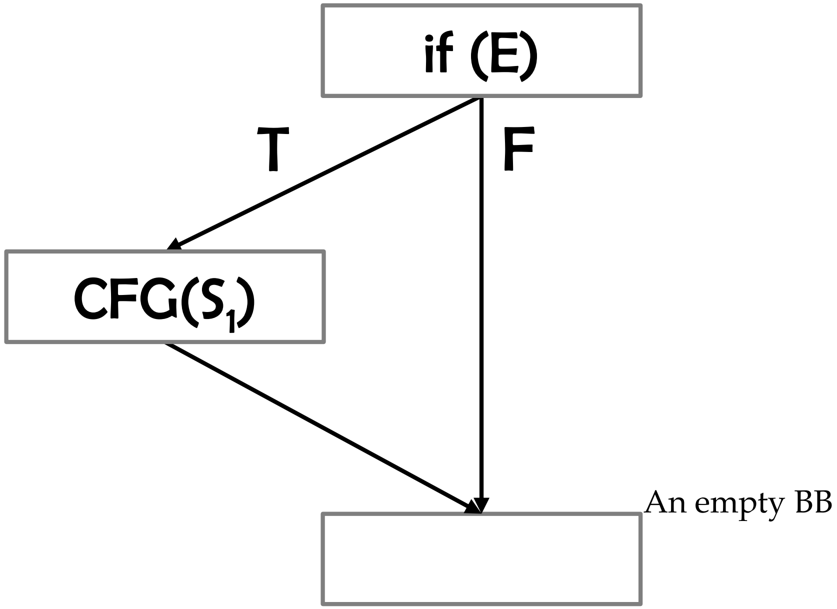
CFG for If-then-else Statement

- CFG (**if (E) S_1; else S_2;**)



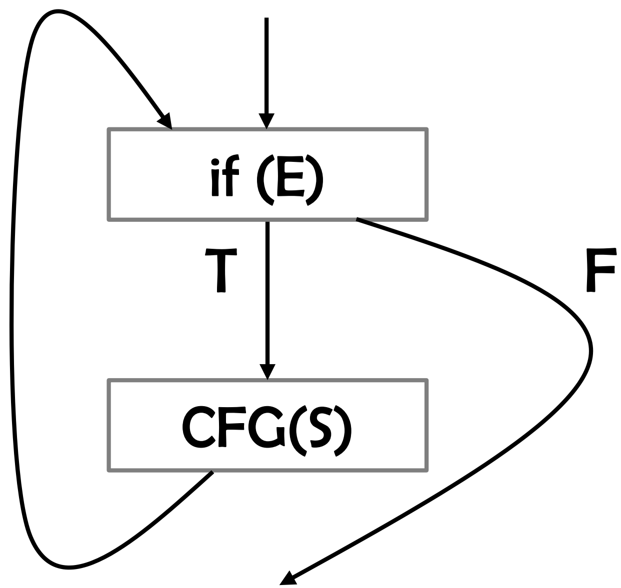
CFG for If-then Statement

- CFG (**if (E) S_1;**)



CFG for While Statement

- CFG (**while (E) S;**)





Recursive CFG Construction

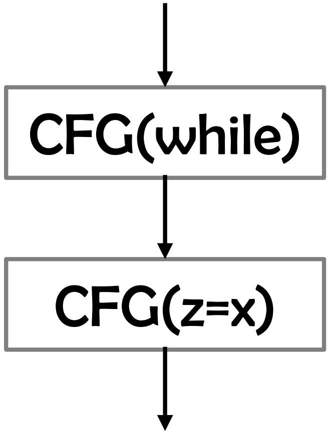
- Nested Statements
  - Recursively construct CFG while traversing IR nodes
- Example

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

Recursive CFG Construction (Cont’d)

- Nested Statements
  - Recursively construct CFG while traversing IR nodes
- Example

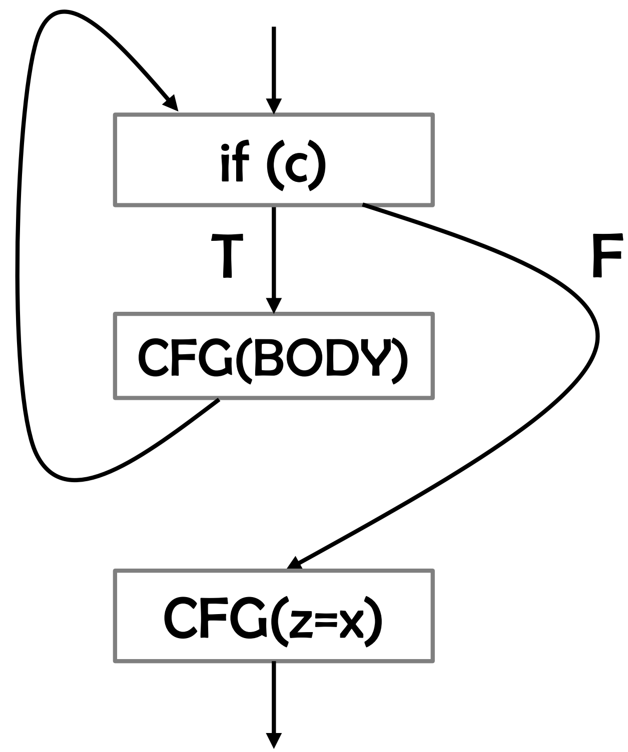
```
while (c) {
      x = y + 1;
      y = 2 * z;
      if (d) x = y + z;
      z = 1;
}
z = x;
```



Recursive CFG Construction (Cont'd)

- Nested Statements
  - Recursively construct CFG while traversing IR nodes
- Example

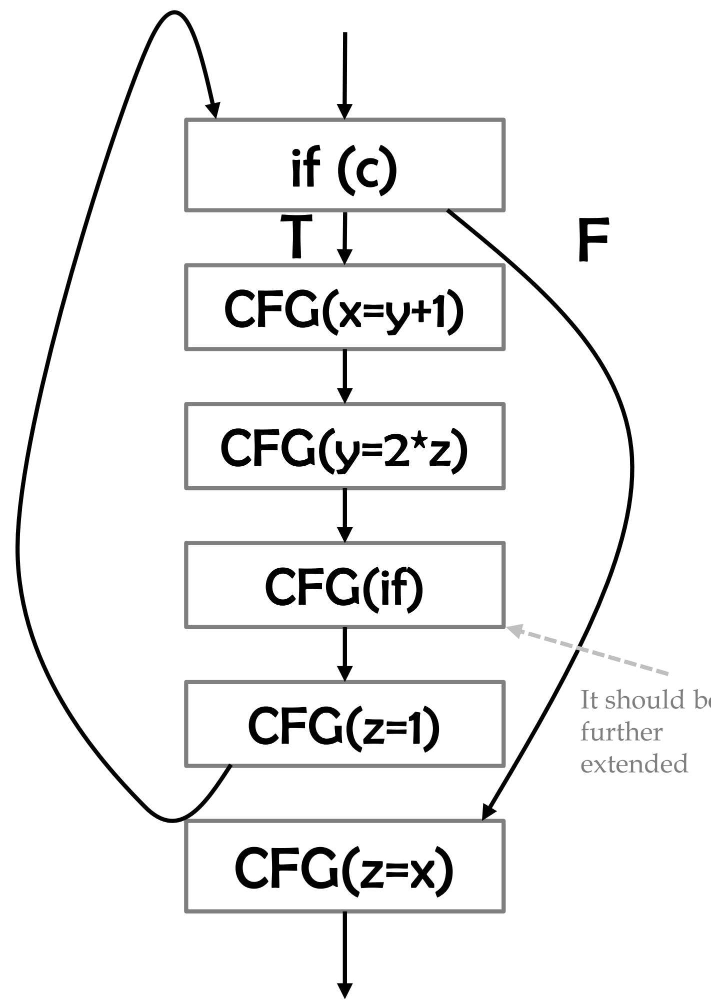
```
while (c) {
    x = y + 1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
z = x;
```



Recursive CFG Construction (Cont'd)

- 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;
```



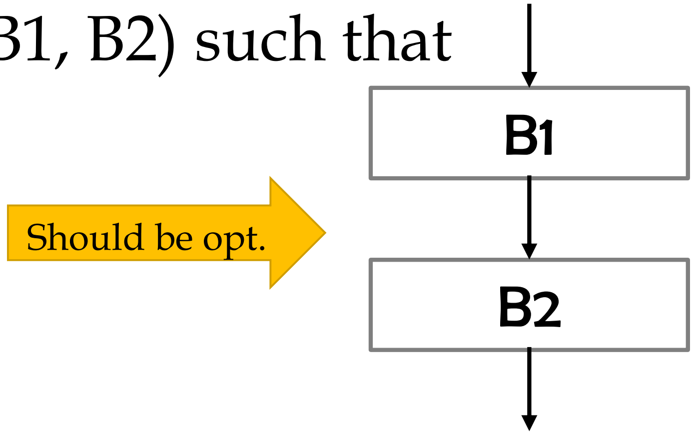
Recursive CFG Construction (Cont'd)

- The above shows a simple algorithm to build CFG
- Generated CFG (with **small BBs**)
  - 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

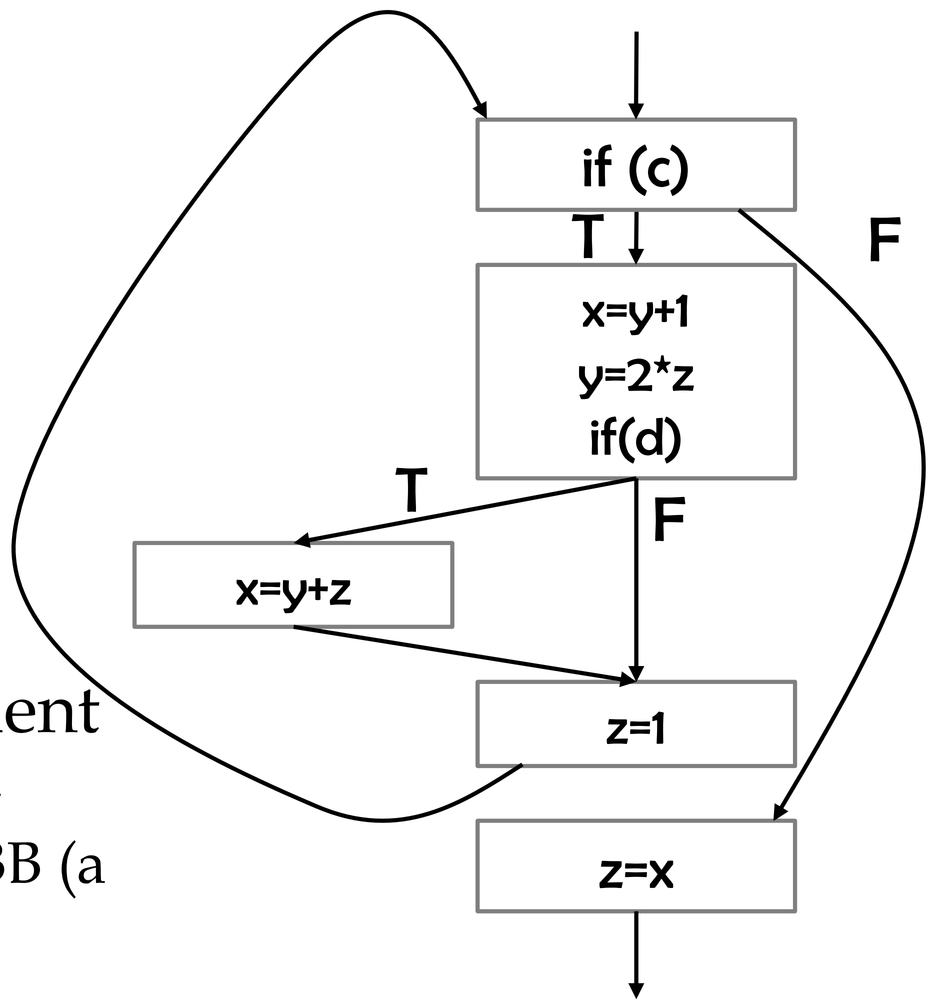
- Basic blocks in CFG:
  - As few as possible
    (number of BBs)
  - As large as possible
    (number of Instructions within BBs)
- For efficient CFG construction:
  1. There should be no pair of BBs (B1, B2) such that
     - B2 is a successor of B1
     - B1 has one outgoing edge
     - B2 has one incoming edge
  2. There should be no empty BBs



Example

- The while loop code:

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



- The CFG is more efficient than the previous one,
  - where a statement is a BB (a node of a CFG)



CFG for Low-level IR

- Identify BBs as sequences of:
  1. Non-branching instructions
  2. Non-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
```

1. No branches (jump) instructions

→ Control does not flow out of BBs

2. No labels instructions

→ Control does not flow into BBs

CFG for Low-level IR (Cont'd)

- Basic block starts:
  - 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 (Cont'd)

- Basic block starts:
  - 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 (Cont'd)

- Basic block starts:
  - 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 (Cont'd)

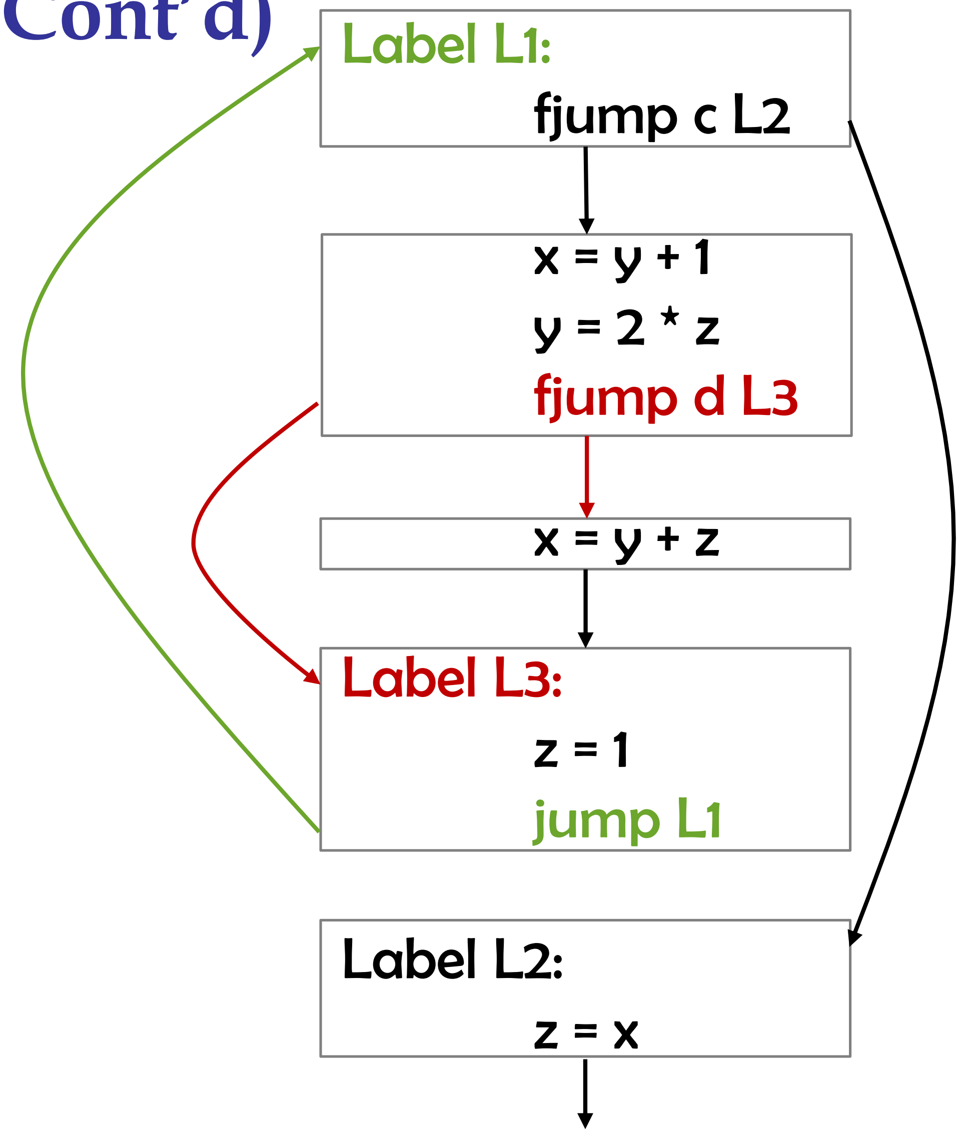
- Basic block starts:
  - at label instructions
  - after jump instructions

- Basic blocks end:
  - at jump instructions
  - before label instructions

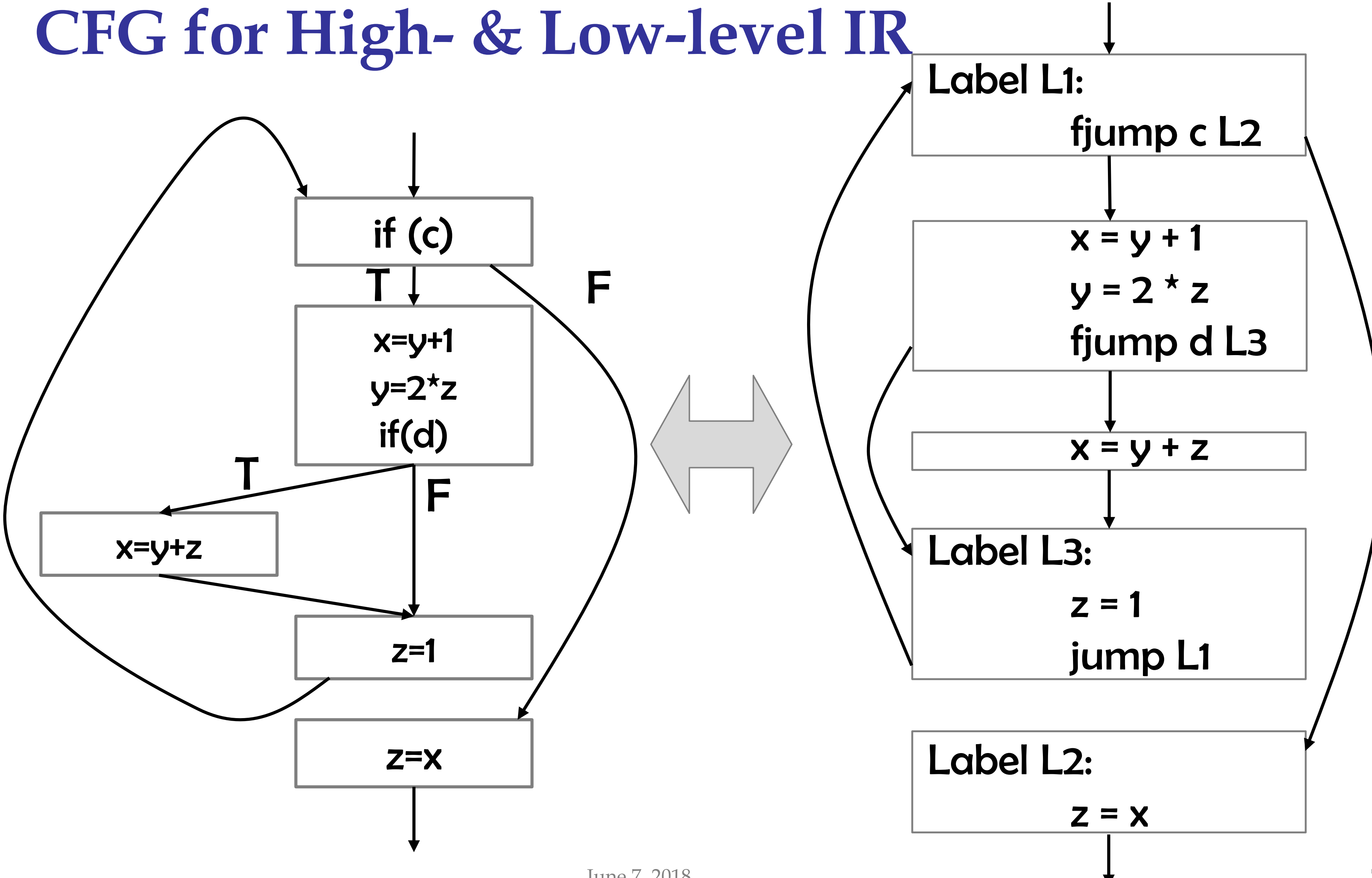
```
Label L1:
        fjump c L2
---------------------
        x = y + 1
        y = 2 * z
        fjump d L3
---------------------
        x = y + z
---------------------
Label L3:
        z = 1
        jump L1
---------------------
Label L2:
        z = x
```

CFG for Low-level IR (Cont'd)

- Conditional jump:
  - 2 successors
  - I.e., RED and BLACK
- Unconditional jump:
  - 1 successor
  - I.e., GREEN



CFG for High- & Low-level IR





Summary of CFG

- Control Flow Graph
  - Global representation of computation and control flow in the program
  - Framework to statically analyze program control-flow
- In a CFG
  - Nodes are BBs that represent computation
  - Edges characterize control flow between BBs
- Can build the CFG representation either from the high IR or from the low IR



Reference

1. Advanced Compiler Design & Implementation, 1st Edition, by Steven Muchnick, **ISBN-10:** 1558603204, **ISBN-13:** 978-1558603202, 1997, Morgan Kaufmann
2. Compilers: Principles, Techniques, and Tools, 2nd Edition, by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, **ISBN-10:** 0321486811, **ISBN-13:** 978-0321486813, 2006, Addison Wesley



QUESTIONS?