

COMPILER CONSTRUCTION

Code Analysis & Optimizations II

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2017

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



Chapter X

Code Analysis and Optimizations



Outline

- Using Control Flow Graphs
- Data Flow Analysis Example with Live Variable Analysis

- Note:
  - The contents of Code Analysis and Optimizations do not follow the chapter order in the textbook
  - 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

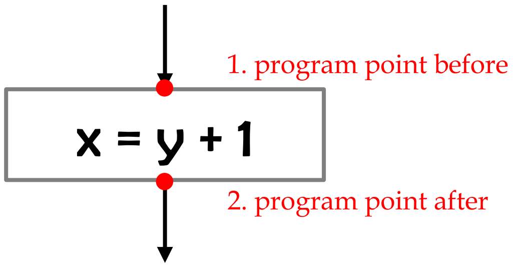
Using CFGs

- Use CFG representation to **statically** extract information about the program
  - Reason at compile-time
  - About the run-time values of variables and expressions in all program executions
- Key ideas
  - Define **program points** in the CFG
  - Reason statically about how the ***information flows* between these program points**
- Extracted information examples
  - **Live variables**
  - **Copy propagation analysis**



Program Points

- **Two program points** for each instruction:
  1. a program point before each instruction
  2. a program point after each instruction

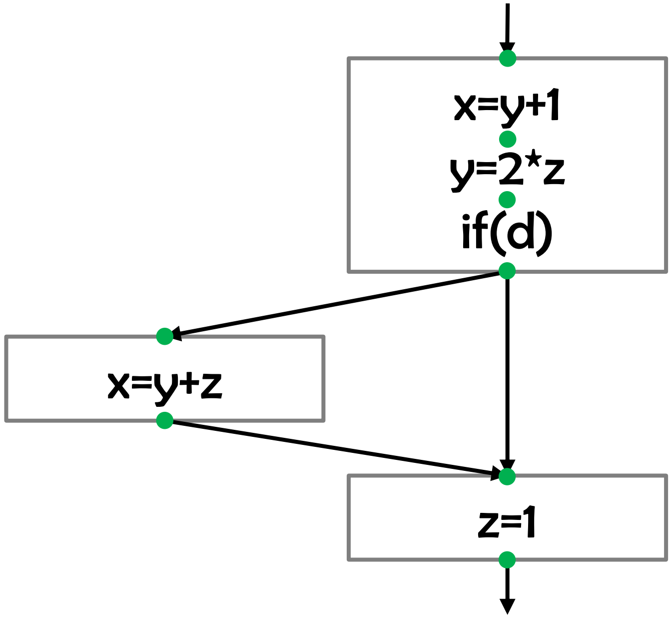


- In a basic block (BB):
  - The program point after an instruction is the
  →program point before the successor instruction

Program Point: Example

- Multiple successor BBs means that
  - point at the end of a block has multiple successor program points

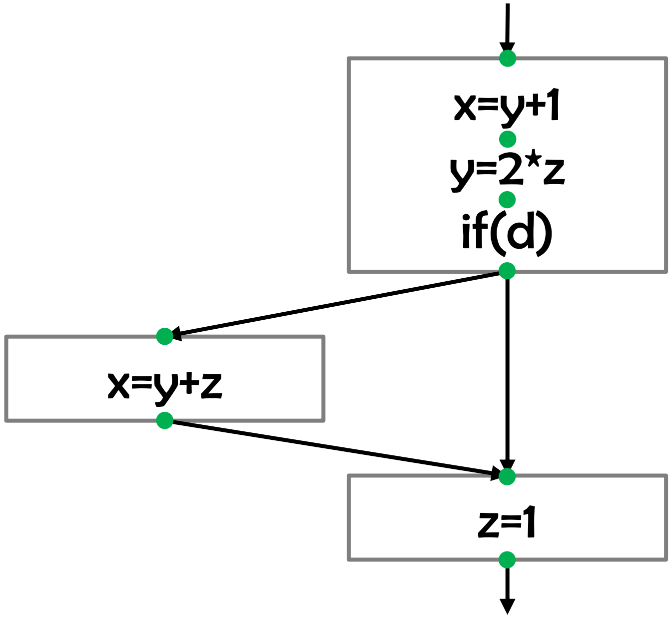
- Depending on the execution, control flows
  - from a program point of its multiple predecessors (of the BB), and
  - from a program point to one of its successors



- **How does *information* propagate between program points?**

Flow of Extracted Information

- Question 1:
  - How does **information flow between the program points** before and after an **instruction**?
- Question 2:
  - How does information flow between successor and predecessor **basic blocks**?
- In plaintexts …
  - Q1: what is the effect of instructions?
  - Q2: what is the effect of control flow?





Using CFGs

- To extract information
  - reason about how it propagates between program points

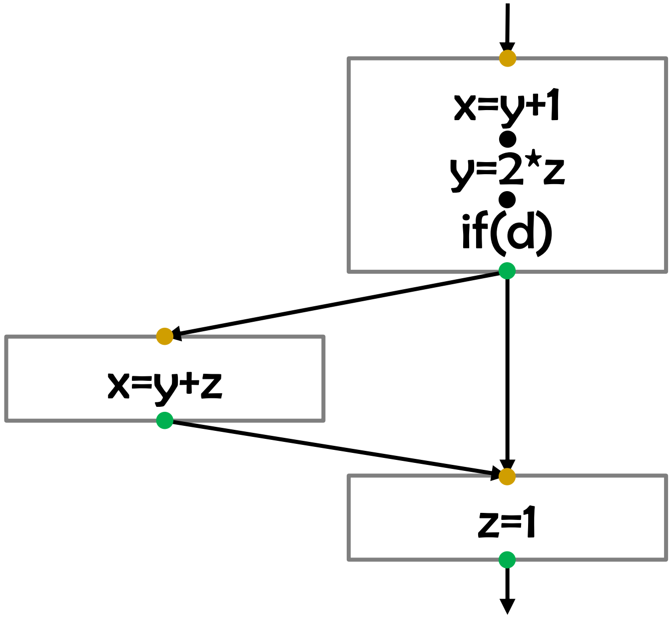
- Two typical program analyses are presented
  - Live variable analysis, which computes live variables are live at each program point
  - Copy propagation analysis, which computes the variable copies available at each program point

- We show how to use CFGs to compute the information at each program point for the analysis

Live Variable Analysis

- Computes live variables at each program point
  - I.e., variables holding values which may be used later (in some execution of the program)

- For an **instruction I**, consider:
  - **in[I]**: live variables at program point before **I**
  - **out[I]**: live variables at program point after **I**
- For a **basic block B**, consider:
  - **in[B]**: live variables at beginning of **B**
  - **out[B]**: live variables at end of **B**



- If **I** is the first instruction in **B**, then **in[B]= in[I]**
- If **I'** is the last instruction in **B**, then **out[B]= out[I']**

How to Compute Liveness?

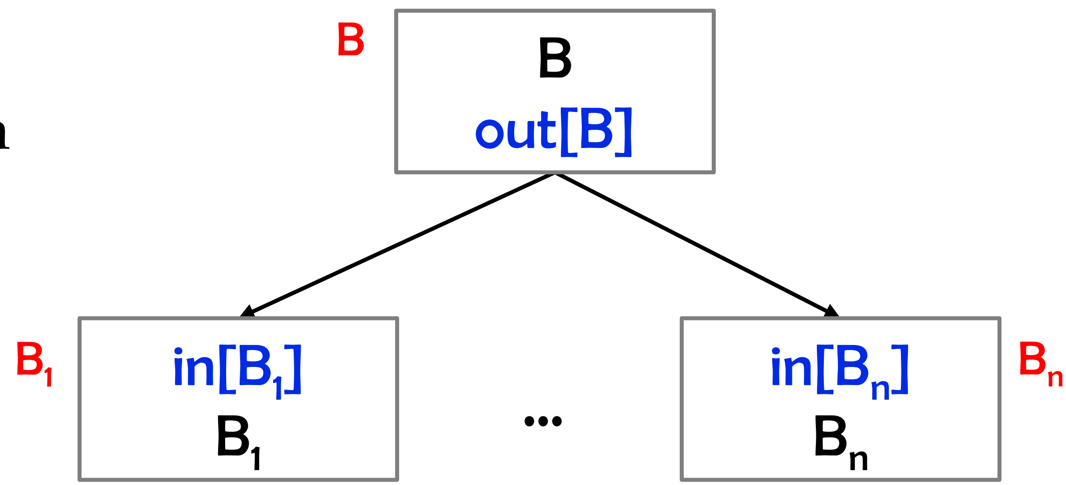
- Rephrase Question 1:
  - For each instruction **I**, what is the relation between **in[I]** and **out[I]**

in[I]

I

out[I]

- Rephrase Question 2:
  - For each basic block **B** with successor blocks B_1, B_2, …, B_n, what is the relation between **out[B]** and **in[B_1], in[B_2], … in[B_n]**





Part 1: Analyze Instructions

- Question:
  - What is the relation between sets of live variables before and after an instruction?

in[I]

I

out[I]

- Examples of the appearances of in/out sets:

in[I] = {x, y, z}	in[I] = {y, z, t}
x = y + z	x = y + z
out[I] = {y, z}	out[I] = {y, z, t}

- … is there a general rule? Yes…



Analyze Instructions

- Knowing variables live *after* **I**, can compute variables live *before* **I**
  - All variables live after **I** are also live before **I**, unless defines (writes) them
  - All variables that **I** uses (reads) are also live before instruction **I**
- Formal definition:
  - **in[I] = (out[I] – def[I]) ∪ use[I]**

  where:
  - **def[I]**: variables defined (written) by instruction **I**
  - **use[I]**: variables used (read) by instruction **I**
  - We learn how to compute **def[I]** and **use[I]** first

in[I]

I

out[I]

Example: Computing Use/Def

- Compute **use[I]** and **def[I]** for each instruction **I**:

Instruction **I**	**use[I]**	**def[I]**
x = y OP z	{y, z}	{x}
x = OP y	{y}	{x}
x = y	{y}	{x}
if (x)	{x}	{}
return x	{x}	{}
x = f(y_1, …, y_n)	{y_1, …, y_n}	{x}

We ignore load and store instructions for now



Example: Relationships between in/out Sets of Consecutive Instructions

- Property of the **in/out** sets for three instructions, I1, I2, and I3, in **B**

  Live1 = in[B] = $in[I_1]$

  Live2 = $out[I_1]$ = $in[I_2]$

  Live3 = $out[I_2]$ = $in[I_3]$

  Live4 = $out[I_3]$ = out[B]

Basic Block B

	Live1
I_1	x=y+1
	Live2
I_2	y=2*z
	Live3
I_3	if(d)
	Live4

- Calculate live sets:

  Live1 = (Live2 - {x}) ∪ {y}

  Live2 = (Live3 - {y}) ∪ {z}

  Live3 = (Live4 - {}) ∪ {d}

in[I] = (out[I] – def[I]) ∪ use[I]



Backward Flow for Liveness Analysis

- Relation:
  - in[I] = (out[I] – def[I]) ∪ use[I]

in[I]
I
out[I]

- **The information flows backward!!!**

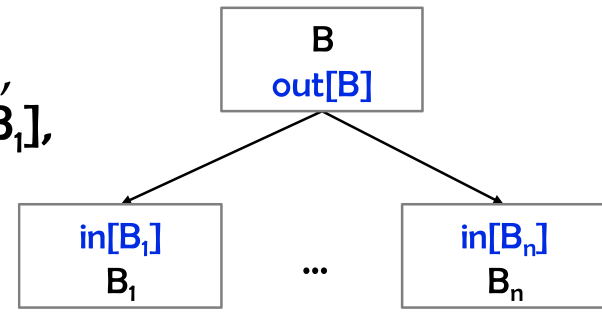
- **Information is calculated at different levels:**
  - **Instructions:** can compute **in[I]** if we know **out[I]**
  - **Basic Blocks:** information about live variable flows from **out[B]** to **in[B]**

Basic Block B

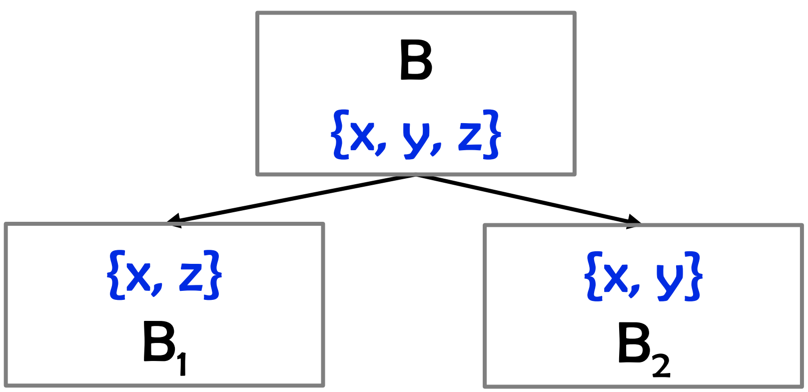
in[B]
x=y+1
y=2*z
if(d)
out[B]

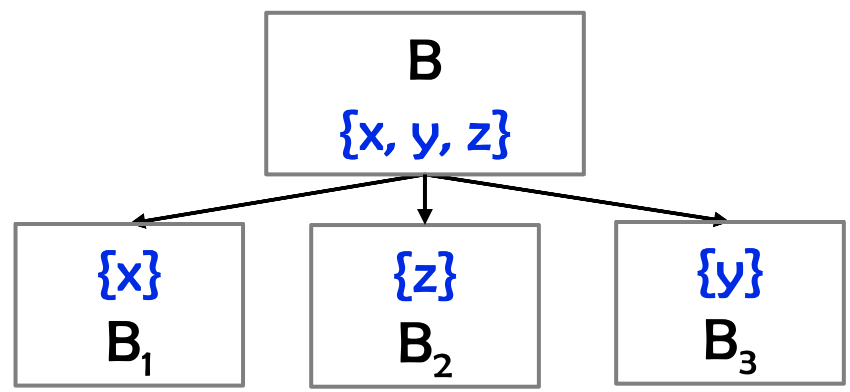
Part 2: Analyze Control Flow

- Question:
  - For each basic block **B** with successor blocks B_1, B_2, …, B_n, what is the relation between **out[B]** and **in[B$_1$], in[B$_2$], … in[B$_n$]**



- Examples of the appearances of in/out sets:

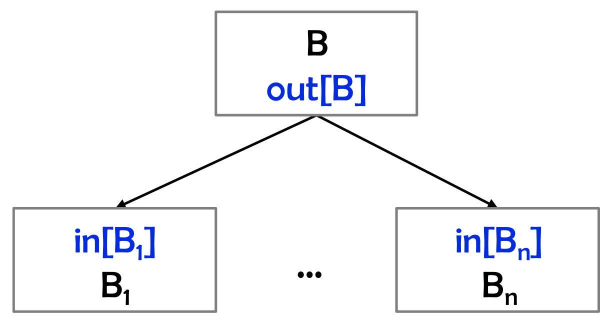




  - Information flows **backward** in the above examples
- What is the general rule?

Analyze Control Flow (Backward)

- Rule:
  - A variables list live at **end** of basic block B if it is live at the beginning of one **successor** block
- **Characterizes all possible program executions!**
- Formal definition:
  - $out[B] = \bigcup_{B' \in successor(B)} in[B']$



- Again, information flows backward: from successors **B’** of **B** to basic block **B**



Constraint System (Backward)

- Put **Part I and Part II** together
  - Start with CFG and derive a system of constraints between live variable sets

$$\begin{cases} in[I] = (\, out[I] - def[I] \,) \cup use[I] & \text{For each instruction I (Part I)} \\ out[B] = \displaystyle\bigcup_{B' \in \text{successor}(B)} in[B'] & \text{For each basic block B (Part II)} \end{cases}$$

- Solve constraints:
  1. Start with **empty sets** of live variables
  2. **Iteratively** apply constraints for instructions & BBs
  3. Stop when reaching a **fixed point**
  - Refer to the algorithm given in the next page

Constraint Solving Algorithm

for all instructions in **in[I] = out[I] = ∅**

repeat

for each instruction **I**

Apply the constrains on the instructions for each BB

$in[I] = (out[I] - def[I]) \cup use[I]$

for each basic block **B**

Apply the constrains on the BBs (Refer to the property in this page)

$out[B] = \bigcup_{B' \in successor(B)} in[B']$

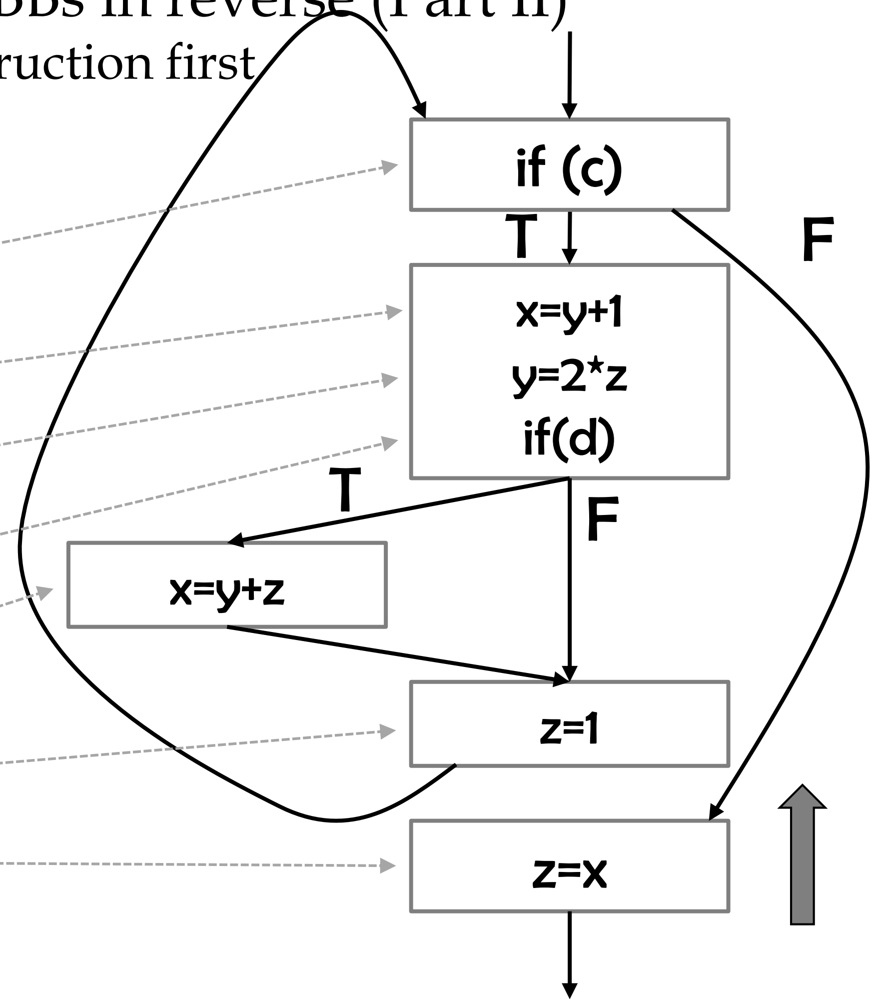
until no change in live sets

- It is important for the backward analysis to compute **in** with **out** (for a instruction and a BB)
- The order, for which instruction/BB applies the constrains, does not matter. Why?

An Example

- Give the use and def sets for each instruction (Part I)
- You can compute **in/out** sets for BBs in reverse (Part II)
  - by deriving the **in/out** for each instruction first

use[I]	def[I]
{c}	{}
{y}	{x}
{z}	{y}
{d}	{}
{y, z}	{x}
{}	{x}
{x}	{z}



Copy Propagation

- Determine copies available at each program point

- For an **instruction I**, consider:
  - **in[I]**: copies available at program point before **I**
  - **out[I]**: copies available at program point after **I**
- For a **basic block B**, consider:
  - **in[B]**: copies available at beginning of **B**
  - **out[B]**: copies available at end of **B**

- If **I** is the first instruction in **B**, then **in[B]= in[I]**
- If **I'** is the last instruction in **B**, then **out[B]= out[I']**

Constraint System for Copy Propagation

- **Same** ***methodology*** with live variable analysis

1. Express flow of information
   (i.e., available copies)
   - For points before and after each instruction (**in[I], out[I]**)
   - For points at entry and exit of BBs (**in[B], out[B]**)

2. Build constraint system
   - using the relations between available copies

3. Solve constraints
   - to determine available copies at each point in the program

Analyze Instructions for Available Copies

- Compute **out[I]** with **in[I]**
  - Remove from **in[I]** all copies of <u=v>, if variable u or v is written by **I**, and Keep all other copies from **in[I]**
  - If **I** is of the form x=y, add it to **out[I]**

in[I]
I
out[I]

- Formal definition:
  - **out[I] = (in[I] – kill[I]) ∪ gen[I]**

  where:
  - **kill[I]**: copies *killed* by instruction **I**
  - **gen[I]**: copies *generated* by instruction **I**



Computing Kill/Gen

- Compute gen[I] and kill[I] for each instruction I

Instruction I	gen[I]	kill[I]
x = y OP z	{}	{remove tuple contains x}
x = OP y	{}	{remove tuple contains x}
x = y	{x=y}	{remove tuple contains x}
if (x)	{}	{}
return x	{}	{}
x = f(y_1, …, y_n)	{}	{remove tuple contains x}

We ignore load and store instructions for now

Forward Flow for Copy Propagation

- Relation:
  - out[I] = (in[I] – kill[I]) ∪ gen[I]

in[I]
I
out[I]

- **The information flows forward!!!**

Basic Block B

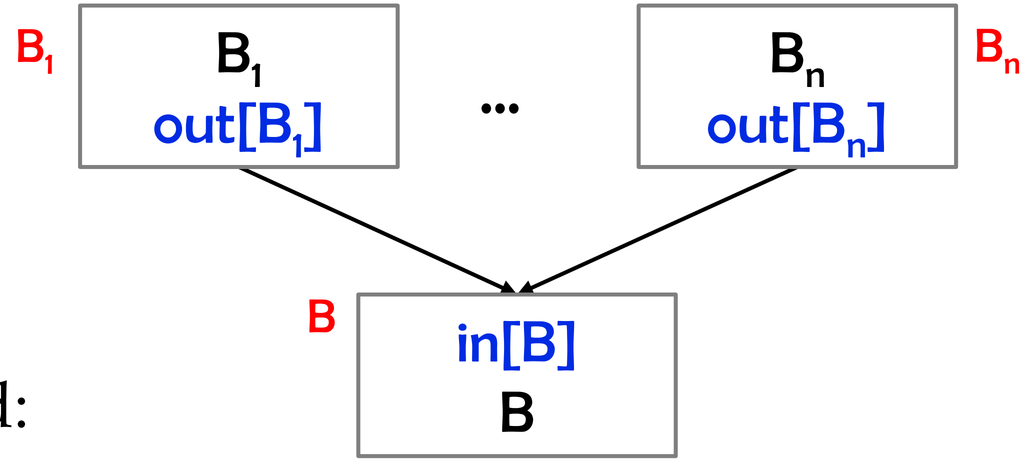
in[B]
x=y+1
y=2*z
if(d)
out[B]

- **Instructions:** can compute **out[I]** if we know **in[I]**
- **Basic Blocks:** information about available copy flows from **in[I]** to **out[I]**



Analyze Control Flow (Forward)

- Rule:
  - A copy assignment is available at **end** of basic block **B** if it is either in **gen[B]** of block **B** or it is available on entry of block **B** and not in **kill[B]**
  - A copy assignment is available on **entry** of block **B** if it is available on exit from all predecessors of block **B**
- **Characterizes all possible program executions!**
- Formal definition:
  - $in[B] = \bigcap_{B' \in predecessor(B)} out[B']$
- Information flows forward:
  - from predecessors **B'** of **B** to basic block **B**



Constraint System (Forward)

- Build constraints
  - Start with CFG and derive a system of constraints between sets of available copies:=

$$
\begin{cases}
out[I] = (in[I] - kill[I]) \cup gen[I] & \text{For each instruction I} \\
in[B] = \bigcap_{B' \in predecessor(B)} out[B'] & \text{For each basic block B}
\end{cases}
$$

- Solve constraints:
  1. Start with empty sets of available copies
  2. Iteratively apply constraints
  3. Stop when we reach a fixed point

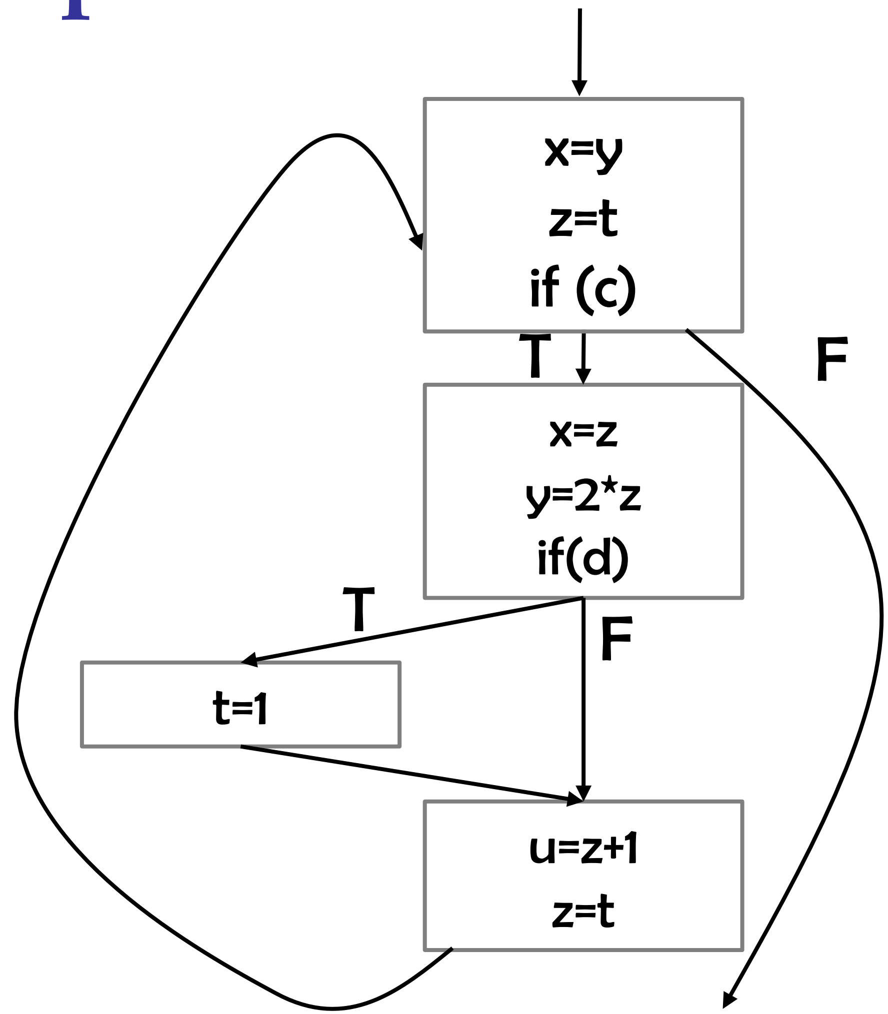
The while-loop Example

- What are the available copies at the end of the program?

  x=y?

  z=t?

  x=z?





Summary

- Extracting information about live variables and available copies is similar
  - Define the required information
    - Define information before/after instructions
    - Define information at entry/exit of basic blocks
  - Build constraints for instructions/control flow
  - Solve constraints to get needed information
- … is there a general framework?
  - Yes!!! Dataflow analysis framework
- We give the simplified definition below
  - You should find online materials to get familiar with the ideas introduced in this file



Data Flow Analysis (Generalization)

- Live variable analysis and detection of available copies are similar
  - Define some information that they need to compute
  - Build constraints for the information
  - Solve constraints iteratively:
    - The information always *increases* during iteration
    - Eventually, it reaches a fixed point
- A general framework is necessary
  - The framework is applicable to many other analyses
  - I.e., live variable/copy propagation are the instances of the framework

Dataflow Analysis Framework

- A common framework for many compiler analyses
  - Compute some information at each program point
  - The computed information characterizes all possible executions of the program

- **Basic methodology:**
  - Describe information about the program using an algebraic structure called ***lattice,***
    - which **formally defines** the representation of the information (e.g., set) , the related operations (e.g., union and join), the **transfer function** for **I** and **B**, etc.
  - Build constraints which show how instructions and control flow modify the information in the ***lattice***
  - Iteratively solve constraints

Transfer Functions (Instruction)

- Dataflow analysis defines a **transfer function**,
  - **F** for each **instruction** in the program,
  - which describes how the instruction modifies the information in the lattice
- Consider
  - in[I] is information before I
  - out[I] is information after I
- The transfer function of a instruction I
  - for forward analysis: out[I] = F(in[I])
  - for backward analysis: in[I] = F(out[I])

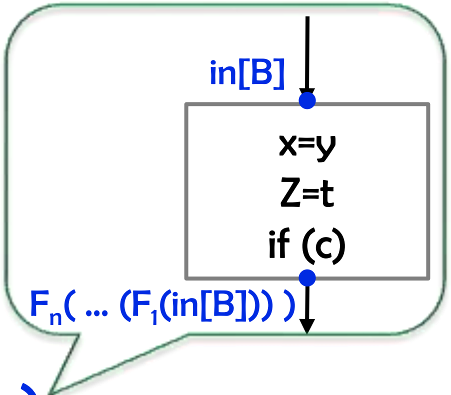


Transfer Functions (Basic Block)

- Extend the concept of transfer function to BBs using function composition

Summarize the effect of instructions within a BB

- Consider
  - a basic block **B** consists of instructions $\mathbf{I_1, \ldots, I_n}$ with transfer functions $\mathbf{F_1, \ldots, F_n}$, respectively
  - **in[B]** is information before **B**
  - **out[B]** is information after **B**



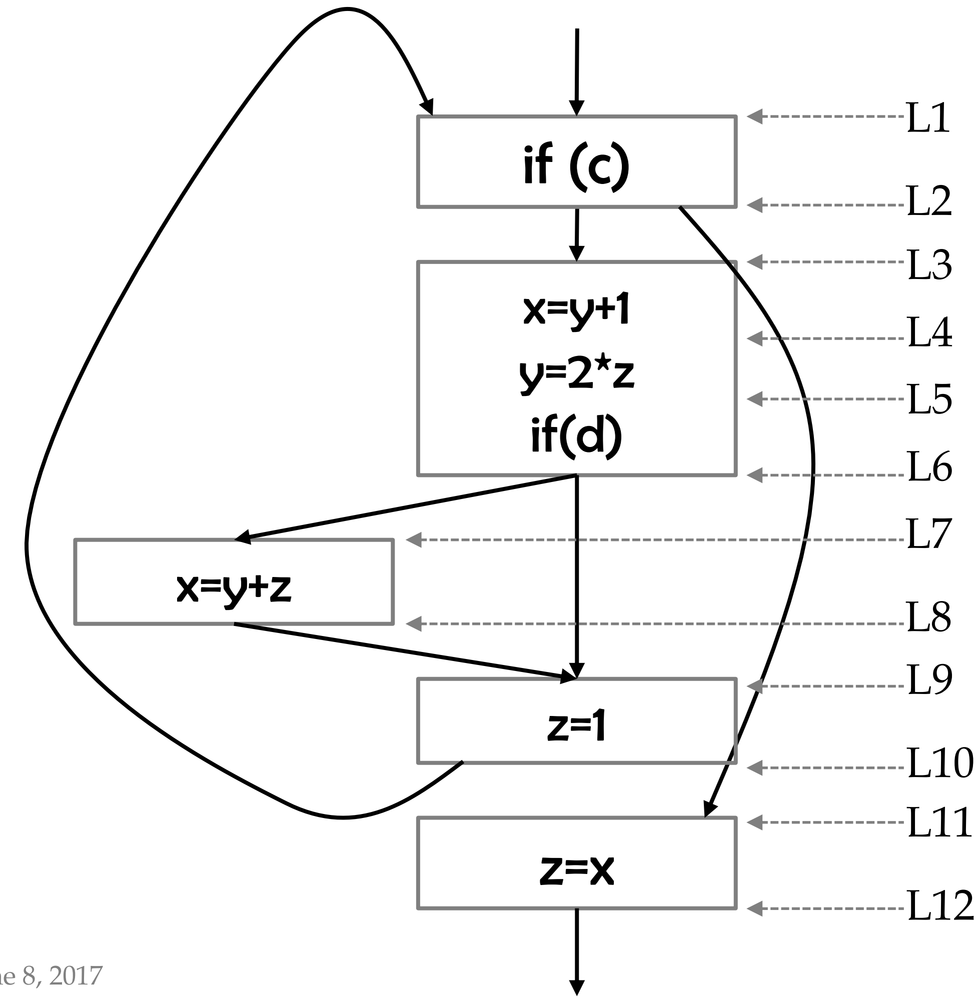
- The transfer function of a basic block **B**
  - for forward analysis: $\mathbf{out[B] = F_n(\ldots (F_1(in[B])))}$
  - for backward analysis: $\mathbf{in[B] = F_1(\ldots (F_n(out[B])))}$

Live Variable Analysis (Revisit)

- What are the **live variables** at each program point?

- Method:
  1. Define sets of live variables
  2. Build constrains
  3. Solve constraints

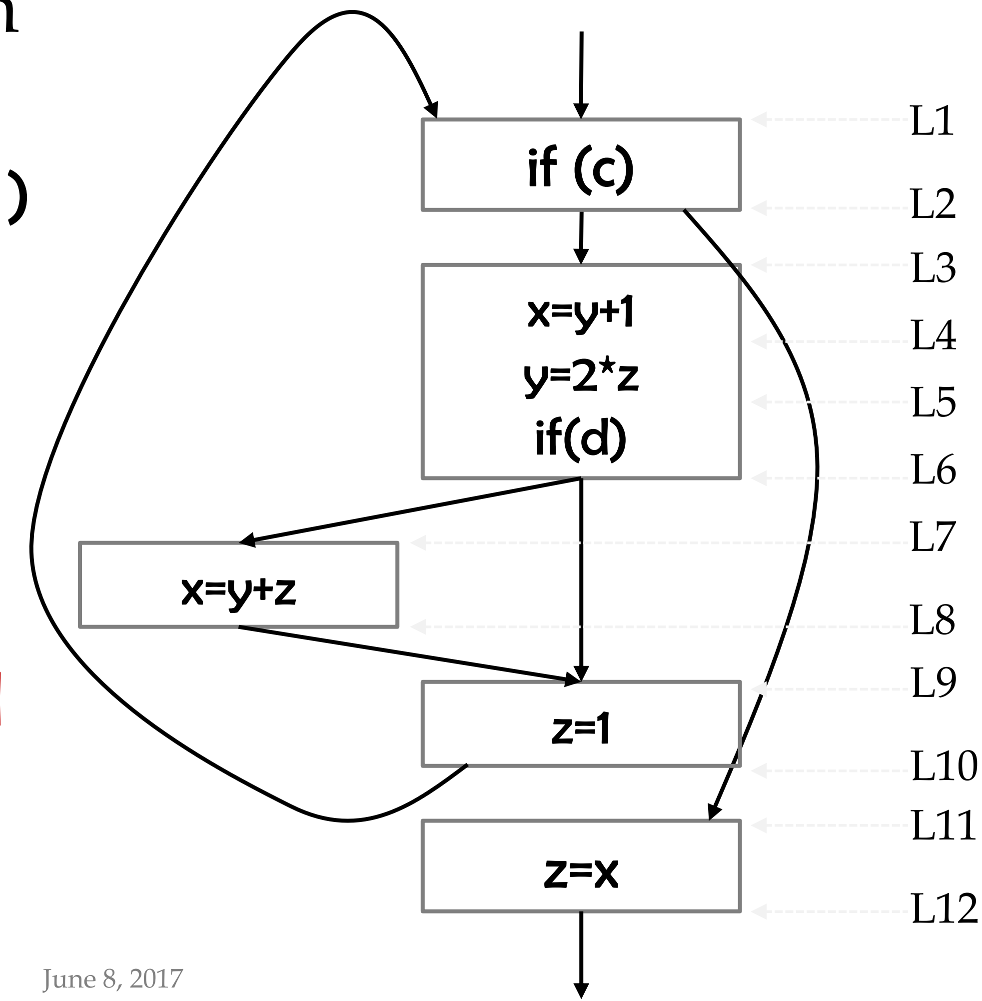
- We use **the analysis** as an example to concrete illustrate
  - how to perform Data Flow Analysis
  - by iteratively visiting the established graph and
  - applying the rules
  - until the fixed-point is reached!



Derive Constraints

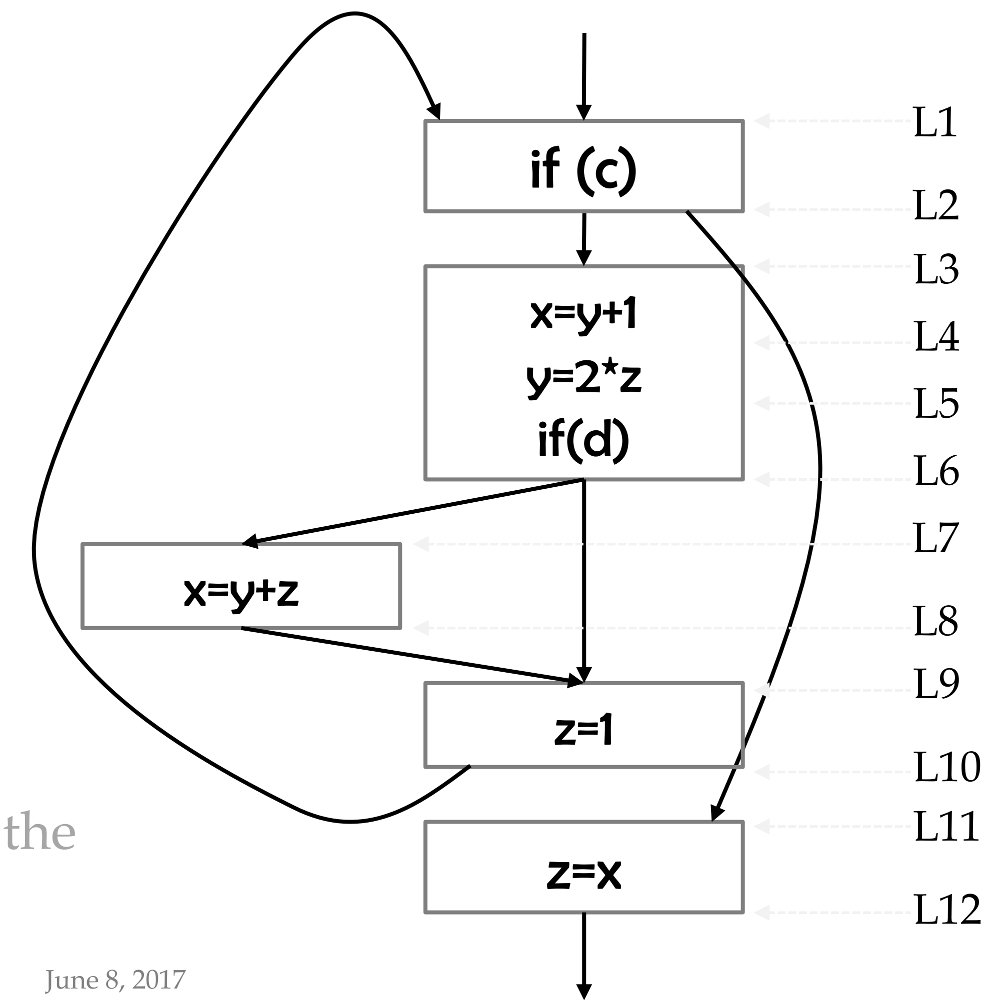
- Constraints for each instruction:
  - $in[I] = (out[I] - def[I]) \cup use[I]$

- Constraints for control flow:
  - $out[B] = \bigcup_{B' \in successor(B)} in[B']$



Derive Constraints (Cont'd)

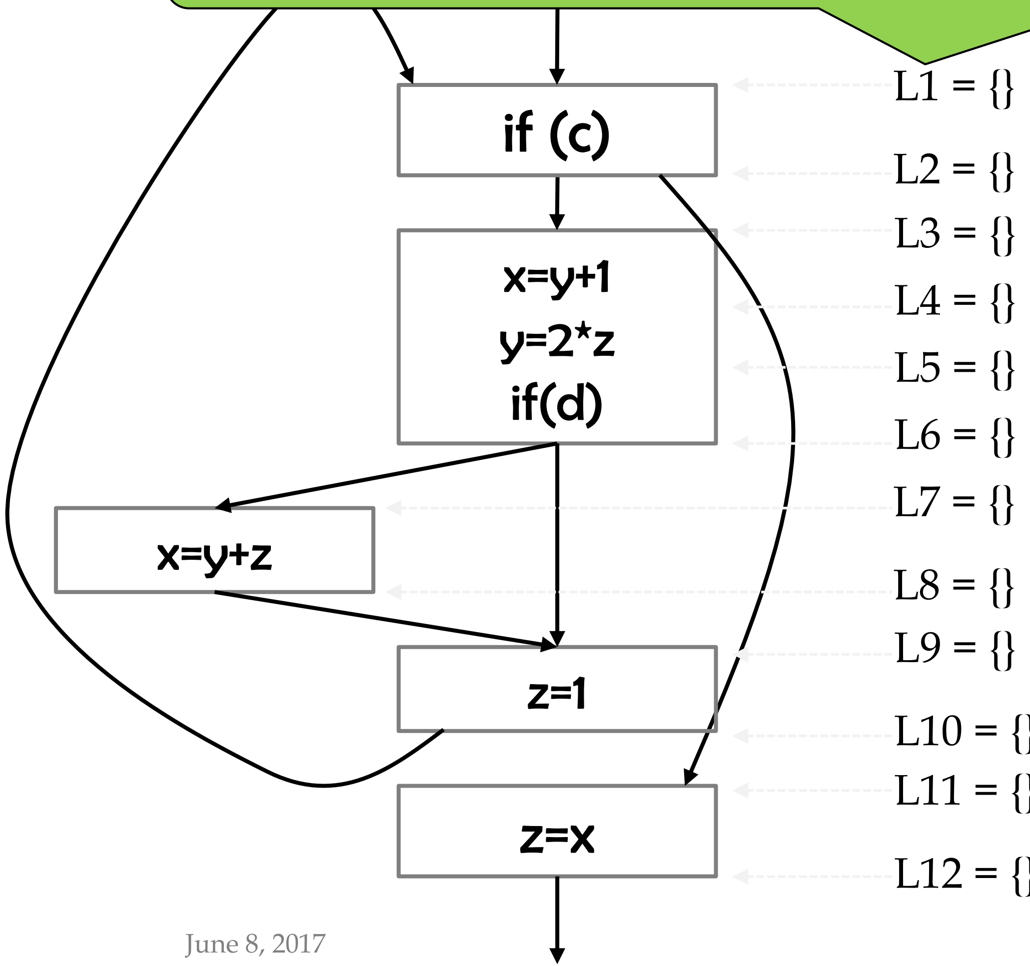
- L1 = L2 ∪ {c}
- L2 = L3 ∪ L11
- L3 = (L4 – {x}) ∪ {y}
- L4 = (L5 – {y}) ∪ {z}
- L5 = L6 ∪ {d}
- L6 = L7 ∪ L9
- L7 = (L8 – {x}) ∪ {y, z}
- L8 = L9
- L9 = L10 – {z}
- L10= L1
- L11= (L12 – {z}) ∪ {x}
- L12= … (? we do not know the successor inst. of L12)



Initialization for Data Flow Analysis

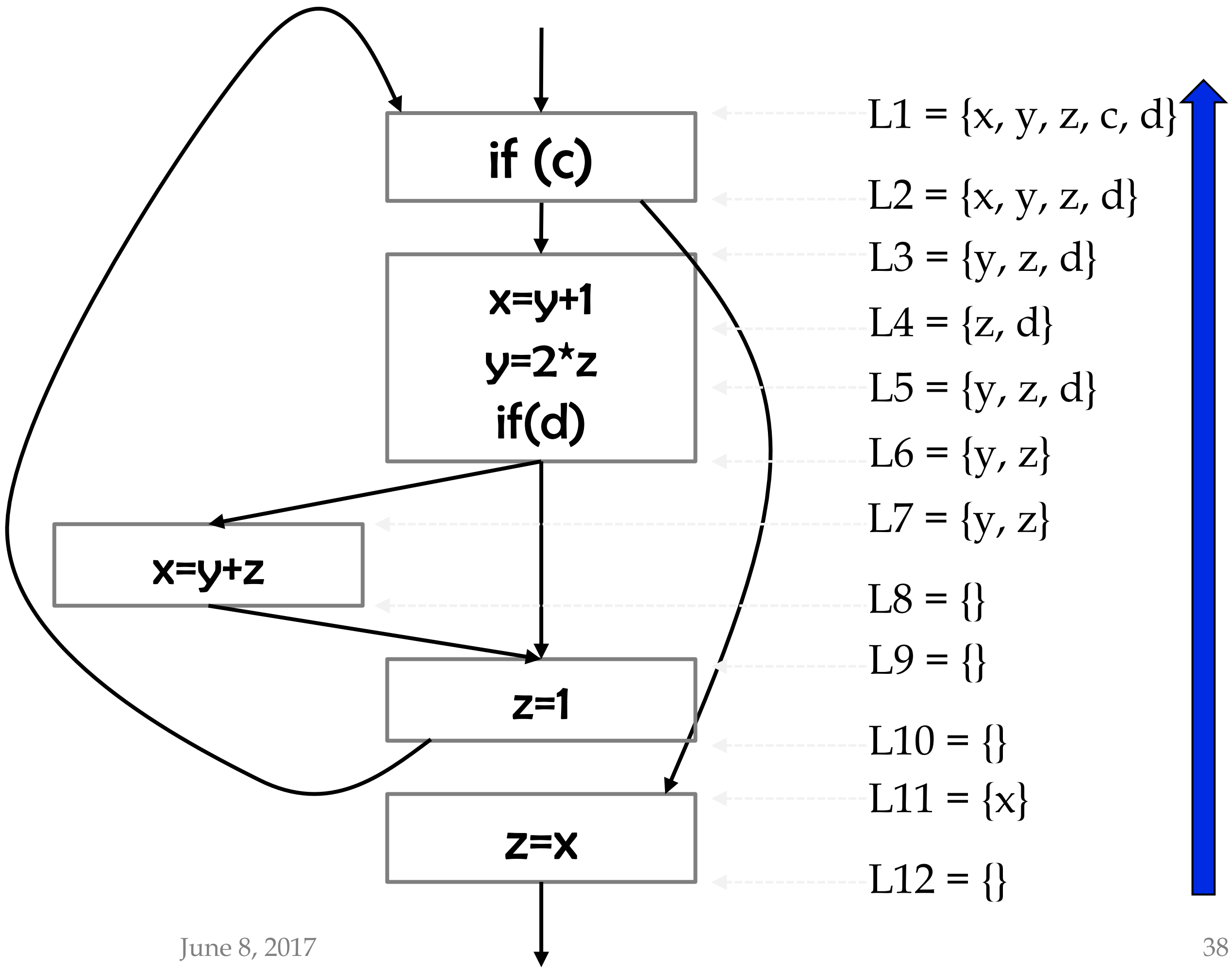
In practice, the program point information will be attached to the CFG to record the information in question, i.e., live variables

- $L1 = L2 \cup \{c\}$
- $L2 = L3 \cup L11$
- $L3 = (L4 - \{x\}) \cup \{y\}$
- $L4 = (L5 - \{y\}) \cup \{z\}$
- $L5 = L6 \cup \{d\}$
- $L6 = L7 \cup L9$
- $L7 = (L8 - \{x\}) \cup \{y, z\}$
- $L8 = L9$
- $L9 = L10 - \{z\}$
- $L10 = L1$
- $L11 = (L12 - \{z\}) \cup \{x\}$



Iteration 1 of Data Flow Analysis

- L1 = L2 ∪ {c}
- L2 = L3 ∪ L11
- L3 = (L4 – {x}) ∪ {y}
- L4 = (L5 – {y}) ∪ {z}
- L5 = L6 ∪ {d}
- L6 = L7 ∪ L9
- L7 = (L8 – {x}) ∪ {y, z}
- L8 = L9
- L9 = L10 – {z}
- L10= L1
- L11= (L12 – {z}) ∪ {x}

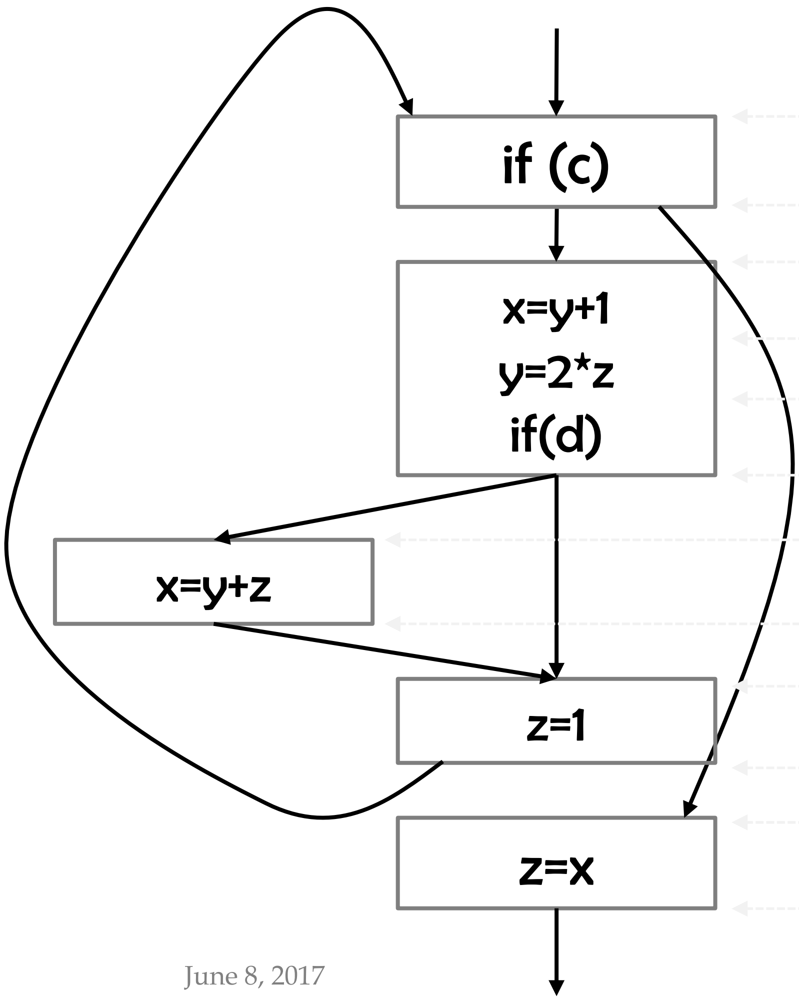


Apply the above constrains to the graph iteratively backward

Iteration 2 of Data Flow Analysis

- L1 = L2 ∪ {c}
- L2 = L3 ∪ L11
- L3 = (L4 – {x}) ∪ {y}
- L4 = (L5 – {y}) ∪ {z}
- L5 = L6 ∪ {d}
- L6 = L7 ∪ L9
- L7 = (L8 – {x}) ∪ {y, z}
- L8 = L9
- L9 = L10 – {z}
- L10= L1
- L11= (L12 – {z}) ∪ {x}

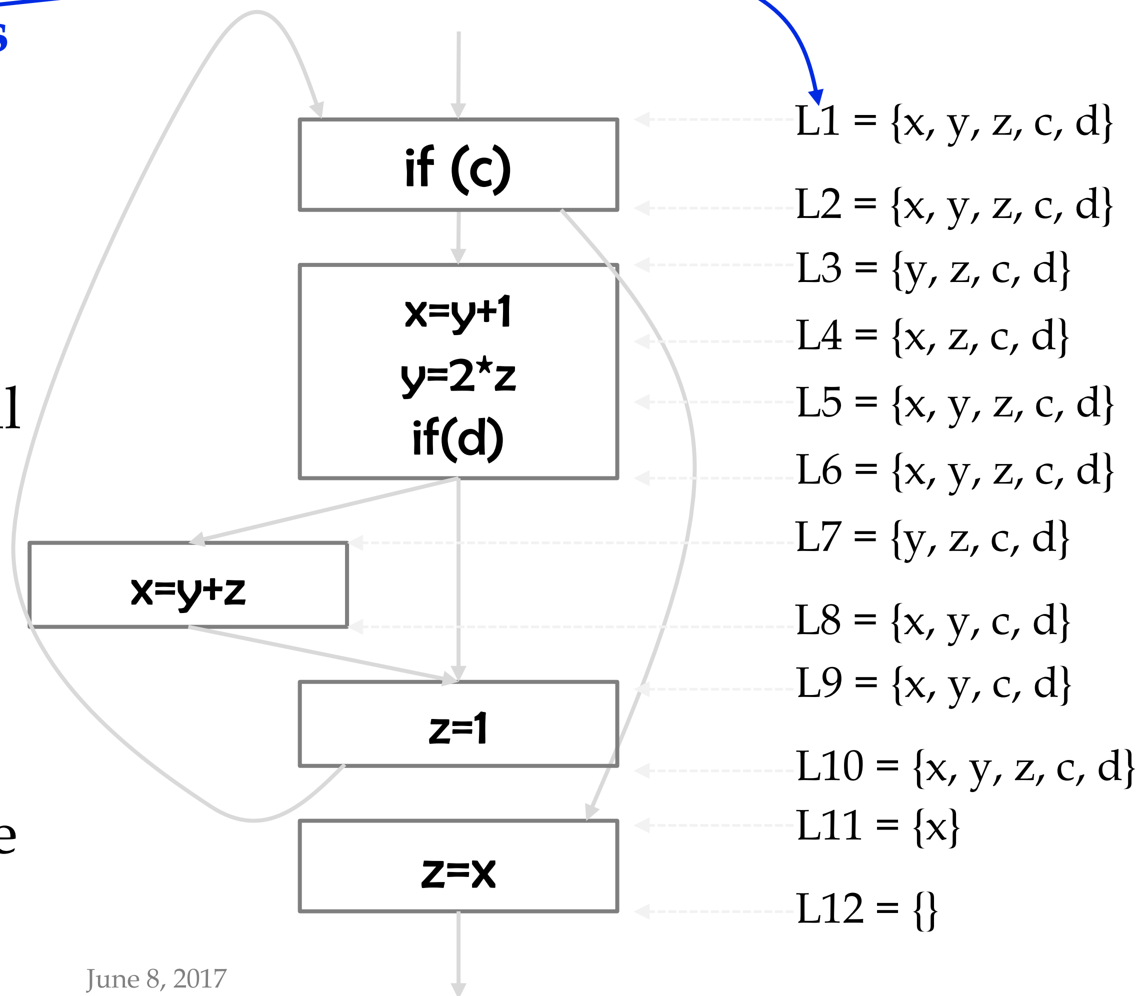
We visit through the program points again!



L1 = {x, y, z, c, d}
L2 = {x, y, z, c, d}
L3 = {y, z, c, d}
L4 = {x, z, c, d}
L5 = {x, y, z, c, d}
L6 = {x, y, z, c, d}
L7 = {y, z, c, d}
L8 = {x, y, c, d}
L9 = {x, y, c, d}
L10 = {x, y, z, c, d}
L11 = {x}
L12 = {}

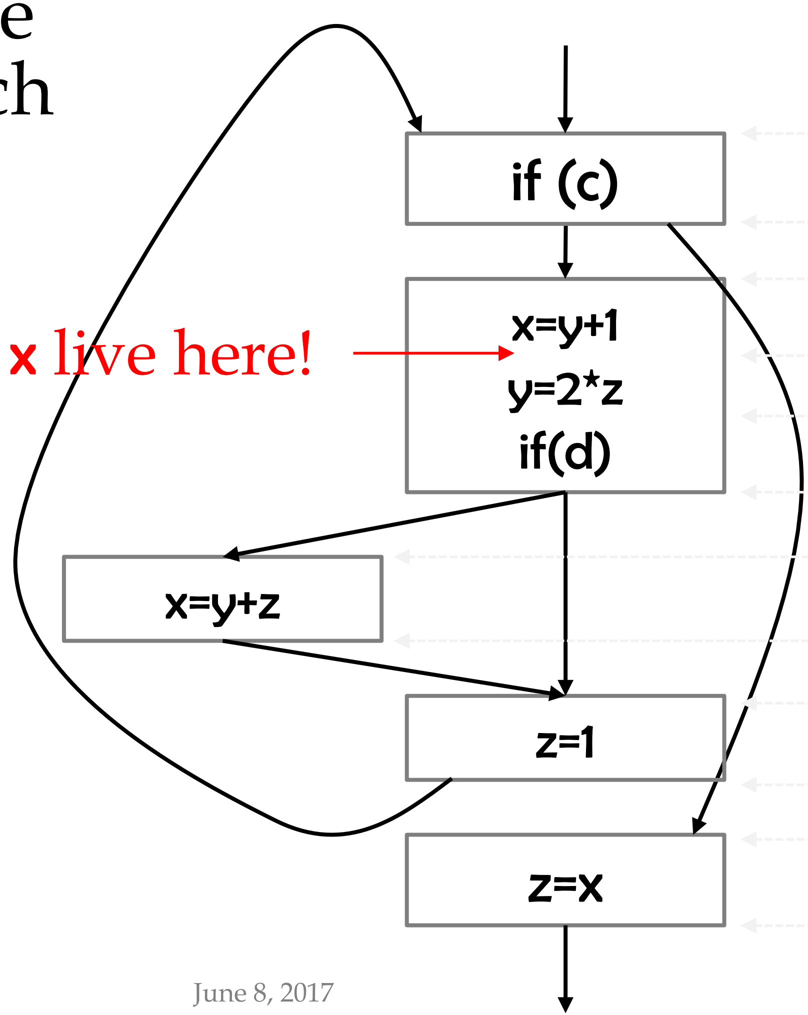
Reached Fixed-point at Iteration 3!!!

- Fixed-point means
  - the **live variable sets** of program points are the same with those we got in the previous iteration
  - I.e., **the sets** of the iteration 3 is identical to those listed in iteration 2 (previous slide)
- This implies that
  - compiler should keep the live variable sets of the previous iteration



Final Result

- Indicate the live variables at each program point



L1 = {x, y, z, c, d}

L2 = {x, y, z, c, d}

L3 = {y, z, c, d}

L4 = {x, z, c, d}

L5 = {x, y, z, c, d}

L6 = {x, y, z, c, d}

L7 = {y, z, c, d}

L8 = {x, y, c, d}

L9 = {x, y, c, d}

L10 = {x, y, z, c, d}

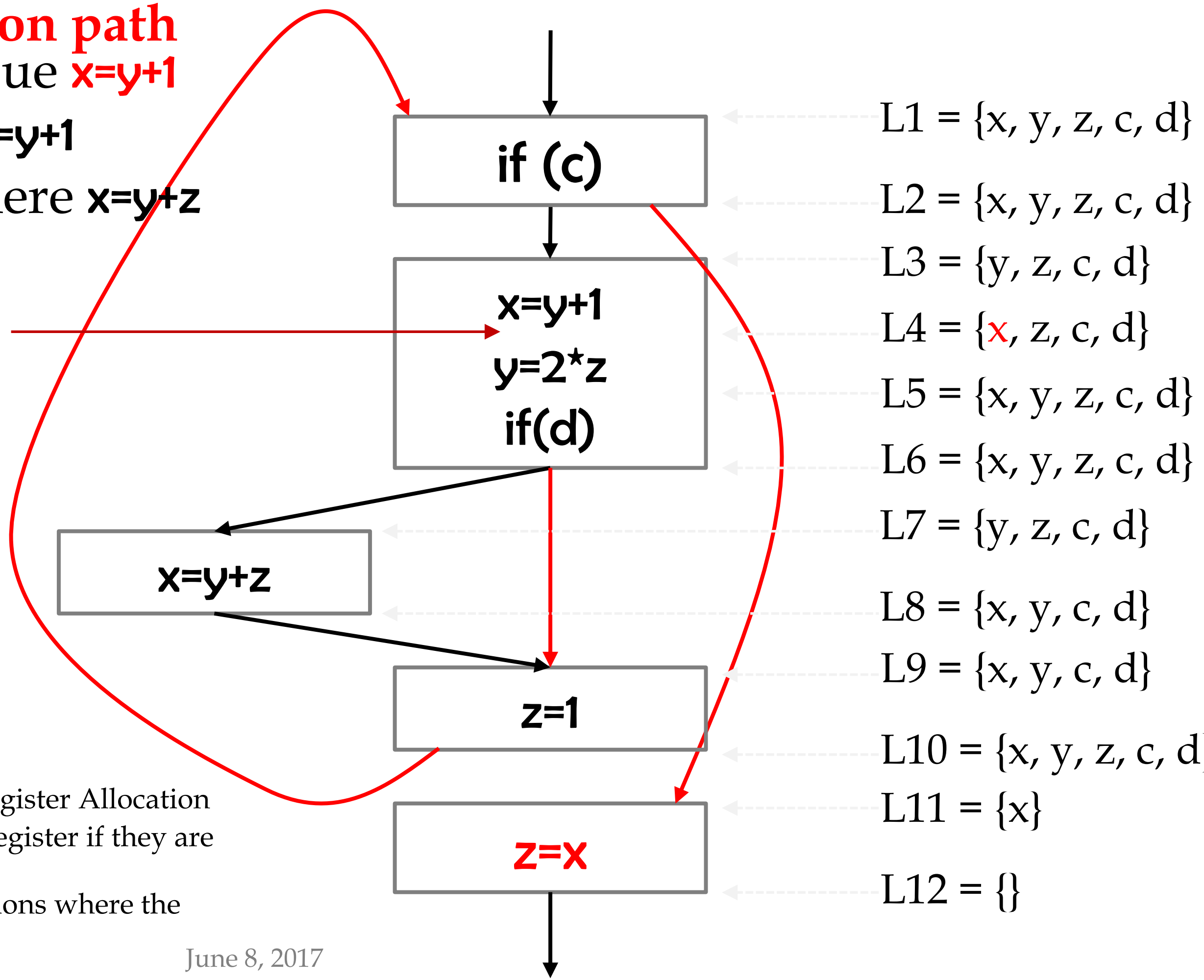
L11 = {x}

L12 = {}

Characterize All Executions

- The analysis detects that
  - there is **an execution path** which uses the value **x=y+1**
  - in such a path, **z=x=y+1**
  - There is a path where **x=y+z**

x live here! (x=y+1)



- NOTE:
  - Live variable analysis is useful for Register Allocation
  - I.e., two **variables** can use the same register if they are never in use at the same time
  - It could be used to identify the situations where the variable is defined but never used



Summary

- What we have learnt here
  - Define some information that they need to compute
  - Build constraints for the information
  - Solve constraints iteratively on the CFG
    - The information always ***increases*** during iteration
    - Eventually, it reaches a fixed-point
- By extending the procedure, we will have a general framework for Data Flow Analysis
  - Computes desired information at each program point
  - The computed information characterizes all possible execution paths of the program
  - Note: it is a **conservative** method to compute the information

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?