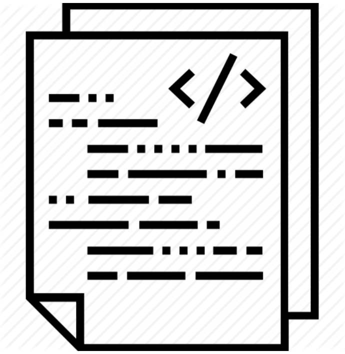
Advanced Software Testing and Debugging (CS598)

Program Analysis Basics

Fall 2020

Lingming Zhang

Program analysis



Program

+

Program Analysis

=

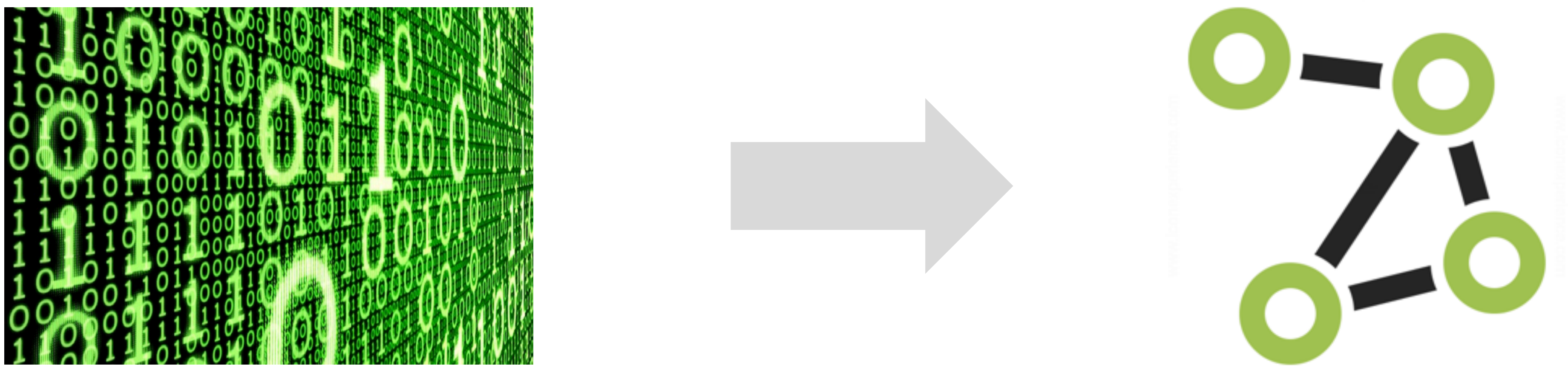
- Is it correct?
- Is it robust?
- Is it safe?
- Is it optimizable?
- …

Program analyzers aim to automatically analyze the behavior of computer programs regarding certain properties

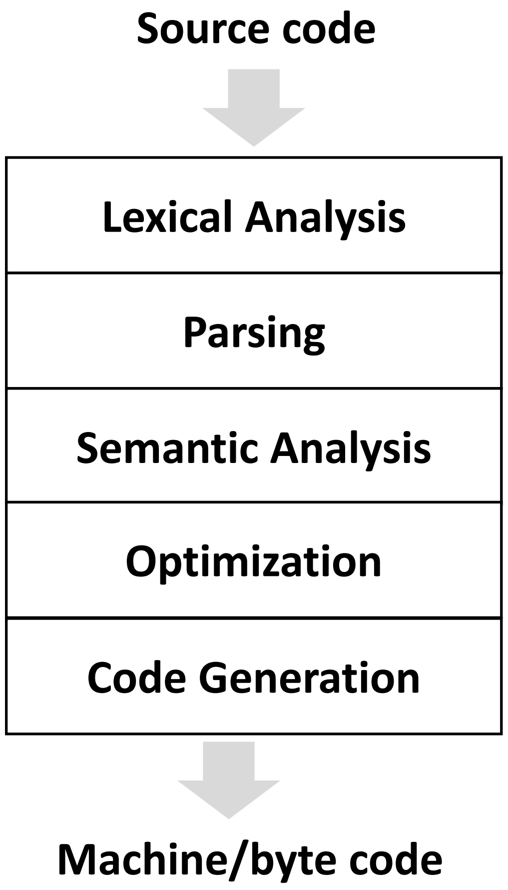
How do we analyze arbitrary programs?

Abstraction!

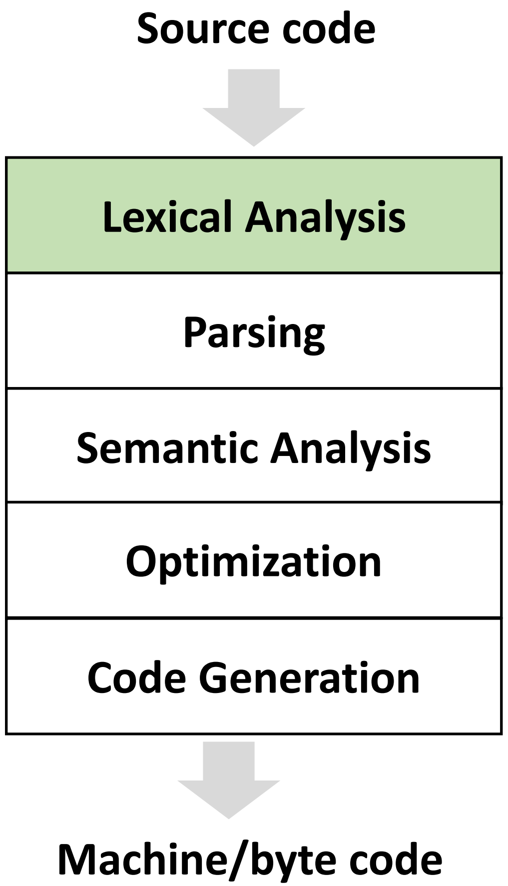
- Transform programs under analysis into structured code representations
  - Easier parsing
  - Easier modification
  - Easier generation



What code representations are used in a typical compiler pass?



Lexical analysis



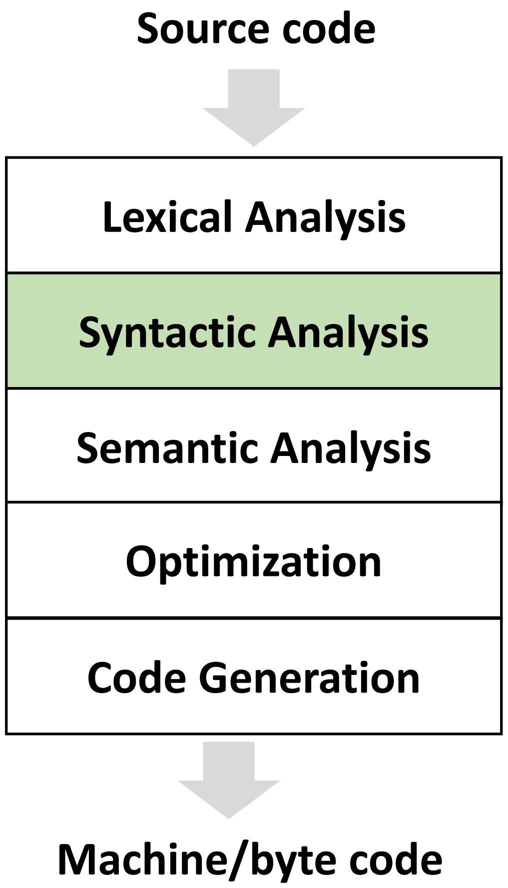
- **Input**: source code text (sequence of chars)
- **Output**: sequence of tokens

```
while (y < z) {
    x = a + b;
    y += x;
}
```

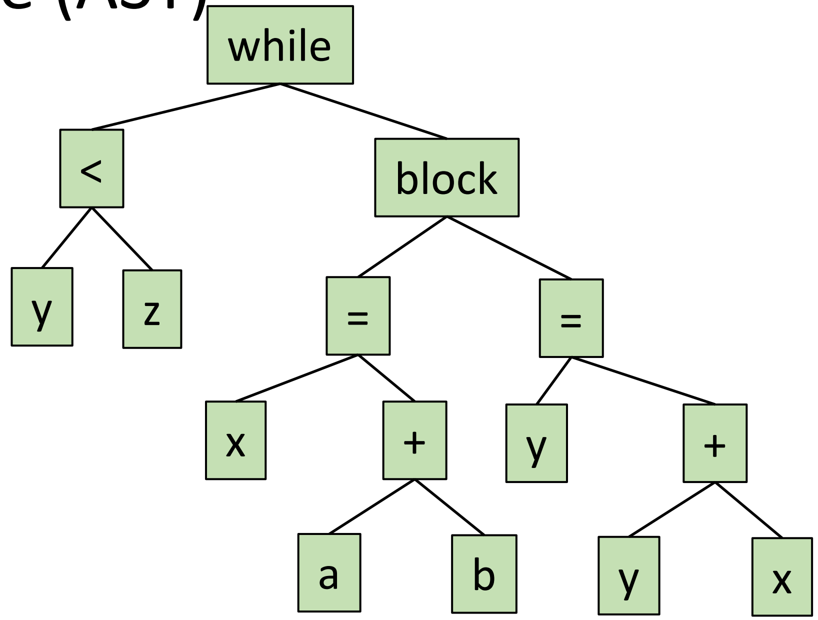
```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

Syntactic analysis

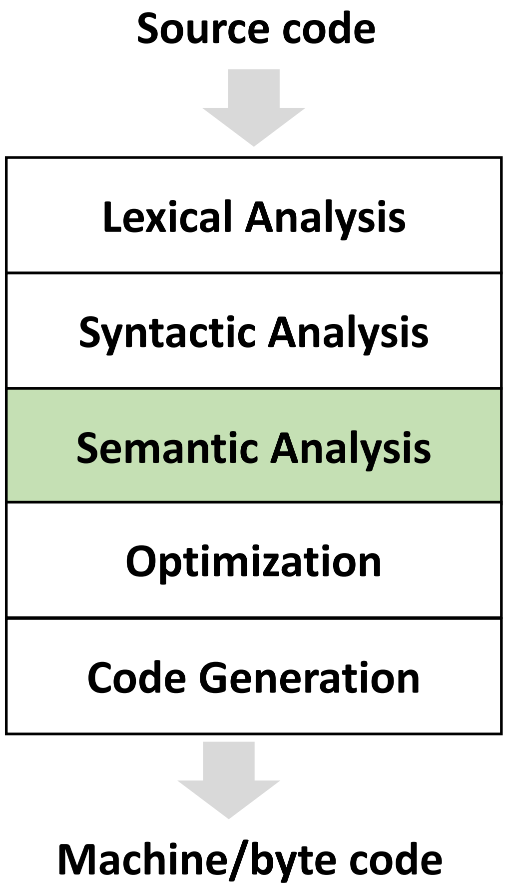
- **Input**: sequence of tokens from lexical analysis
- **Output**: abstract syntax tree (AST)




```
while (y < z) {
    x = a + b;
    y += x;
}
```

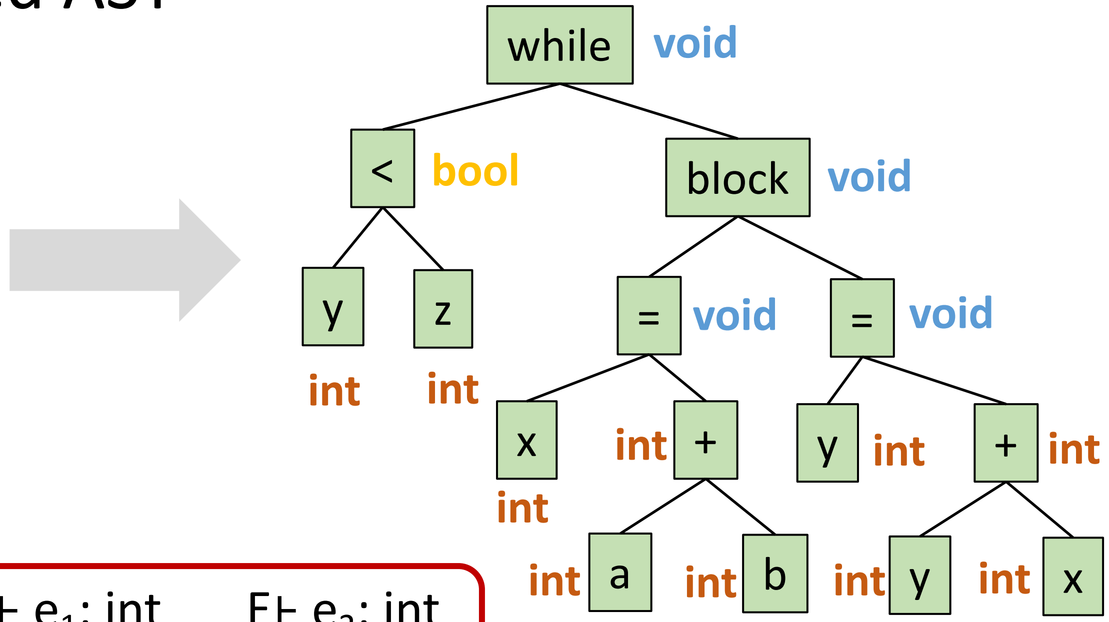


Semantic analysis



- **Input**: abstract syntax tree (AST)
- **Output**: annotated AST

```
while (y < z) {
    x = a + b;
    y += x;
}
```



$$\frac{\text{i is an integer literal}}{E \vdash i: int}$$

$$\frac{E \vdash e_1: int \qquad E \vdash e_2: int}{E \vdash e_1 + e_2: int}$$

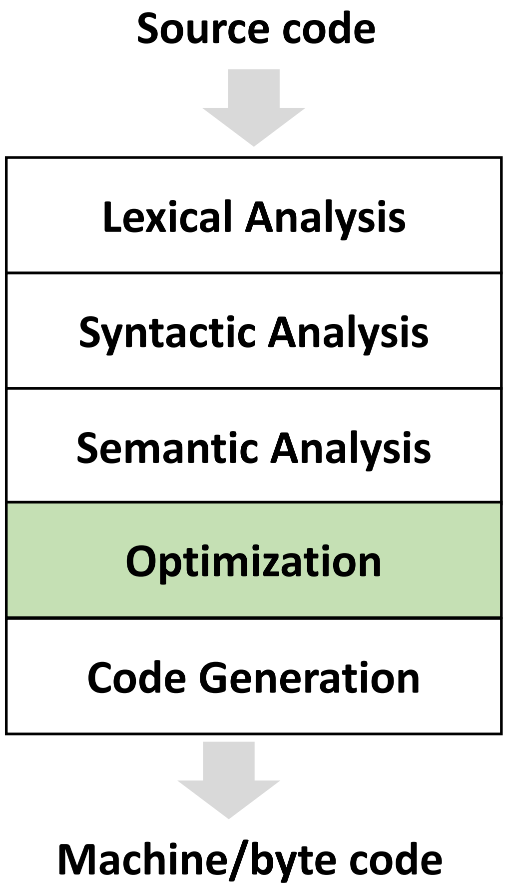
$$\frac{x{:}T \text{ is in } E}{E \vdash x: T}$$

$$\frac{E \vdash e_1: int \qquad E \vdash e_2: int}{E \vdash e_1 * e_2: int}$$

...

Type checking rules

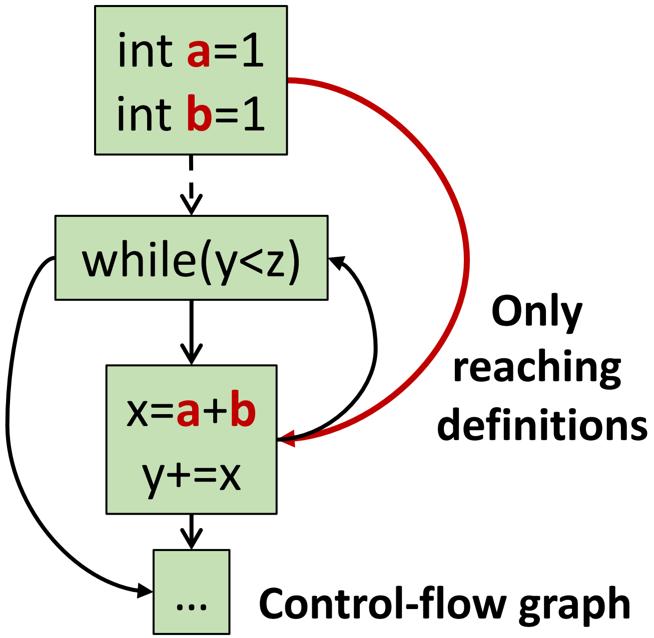
Optimization



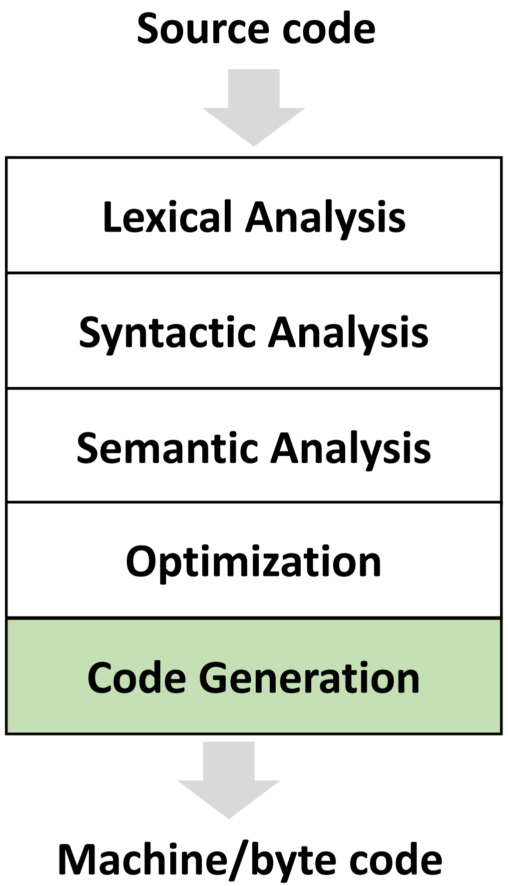
- **Input**: original code representation
- **Output**: optimized code representation

```
int a=1;
int b=1;
…
while (y < z) {
    x = a + b;
    y += x;
}
```

```
int a=1;
int b=1;
…
while (y < z) {
    y += 2;
}
```



Code generation



- **Input**: optimized code representation
- **Output**: final target code

```
while (y < z) {
    x = a + b;
    y += x;
}
```

Topics

- Abstract syntax tree (AST)
- Control-flow graph (CFG)
- Control-flow-based code coverage
- Data-flow analysis
- Data-flow-based code coverage

How do we describe a programming language?

Example program:

```
while (y < z) {
    x = a + b;
    y += x;
}
```

A grammar covering this program and similar ones:

Stmt ⟶ WhileStmt | AssignStmt | CompoundStmt

WhileStmt ⟶ "while" "(" Exp ")" Stmt

AssignStmt ⟶ ID "=" Exp ";"

CompoundStmt ⟶ "{" StmtList "}”

StmtList ⟶ ε | Stmt StmtList

Exp ⟶ Less | Add | ID

Less ⟶ Exp "<" Exp

Add ⟶ Exp "+" Exp

Context-free grammar

- A context-free grammar G = **⟨Σ, N, P, S⟩**, where
  - ***Σ***: alphabet (finite set of symbols, or terminals)
    - Often written in lowercase
  - **N**: a finite, nonempty set of nonterminal symbols, $\mathbf{N} \cap \boldsymbol{\Sigma} = \emptyset$
    - Often at least the first letter in UPPERCASE
  - **P**: the set of production rules, each with the form $\mathbf{X \rightarrow Y_1\ Y_2 \ldots Y_n}$
    - where $\mathbf{X \in N}$, $\mathbf{n \geq 0}$, and $\mathbf{Y_k \in N \cup \Sigma}$
  - **S**: the start symbol (one of the nonterminals), i.e., $\mathbf{S \in N}$

Grammar (P):

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

=

Grammar (P):

$E \rightarrow E+E$

$\quad | \ E*E$

$\quad | \ (E)$

$\quad | \ id$

***Σ*:** +, * (,), id

N: E

S: E

Context-free grammar

Example program:

```
while (y < z) {
    x = a + b;
    y += x;
}
```

A grammar covering this program and similar ones:

Stmt ⟶ WhileStmt | AssignStmt | CompoundStmt

WhileStmt ⟶ "while" "(" Exp ")" Stmt

AssignStmt ⟶ ID "=" Exp ";"

CompoundStmt ⟶ "{" StmtList "}"

StmtList ⟶ ε | Stmt StmtList

Exp ⟶ Less | Add | ID

Less ⟶ Exp "<" Exp

Add ⟶ Exp "+" Exp

Σ: ID, "while", "(", "=", "{", …

N: Stmt, WhileStmt, …

S: Stmt

Context-free grammar: generating strings

- **G** defines a language **L(G)** over the alphabet Σ
- Σ^* is the set of all possible sequences of Σ symbols
- **L(G)** is the subset of Σ^* that can be derived from the start symbol **S**, by following the production rules **P**
  - A **derivation** is such a sequence of productions applied

Grammar:

E ⟶ E+E
 | E*E
 | (E)
 | id

E ⟶ E+E
⟶ E * E+E
⟶ id * E + E
⟶ id * id + E
⟶ id * id + id

Derivation

id
id * id
id * id + id
id * id + id *id
id + id + id + id
…

L(G)

Context-free grammar: parsing strings

- Checking if input string (e.g., code) **s** ∈ **L(G)**, i.e., checking for **acceptance**
- Algorithm: Find a derivation starting from the start symbol of **G** to **s**

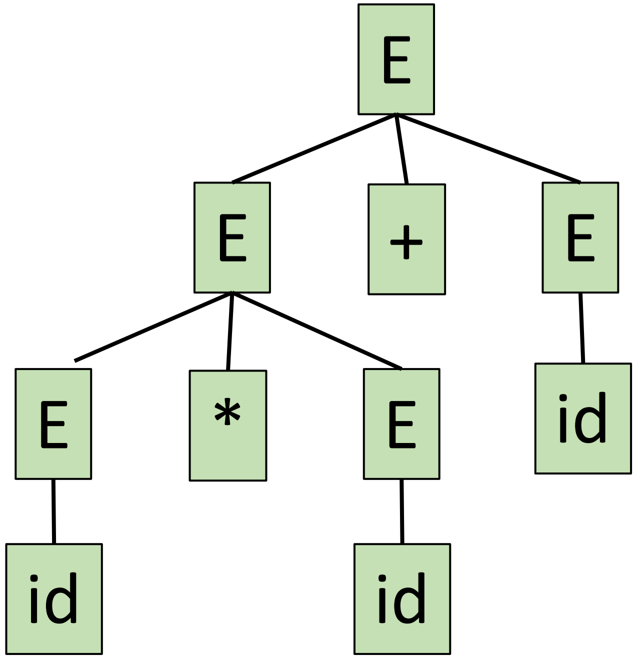
id * id + id

Source

Language grammar:
E ⟶ E+E | E*E | (E) | id

E ⟶ E+E
⟶ E * E+E
⟶ id * E + E
⟶ id * id + E
⟶ id * id + id

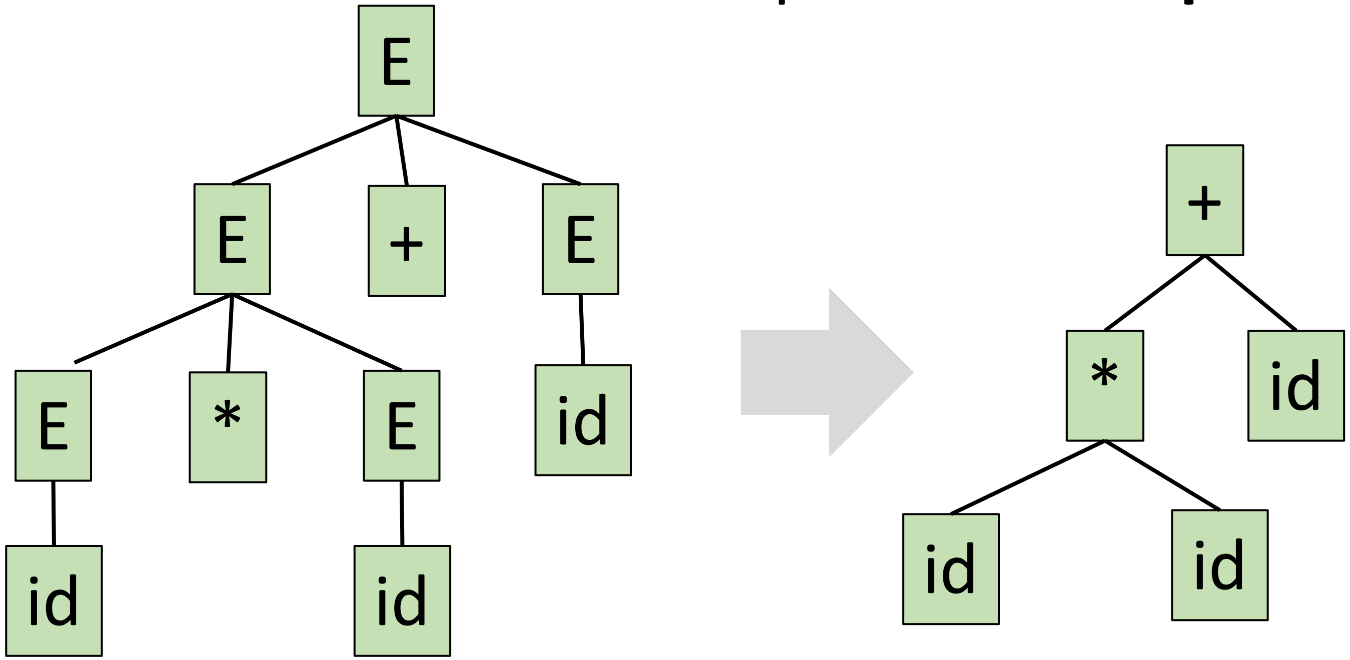
Derivation



Parse tree

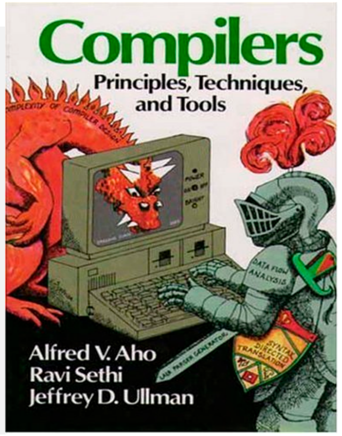
Abstract syntax tree (AST)

- Simplified syntactic representations derived from code parse tree
- Represents the abstract syntactic structure of a language construct
- Usually the interior and root nodes represent **operators**, and the children of each node represent the **operands** of that operator



Parse tree **AST**

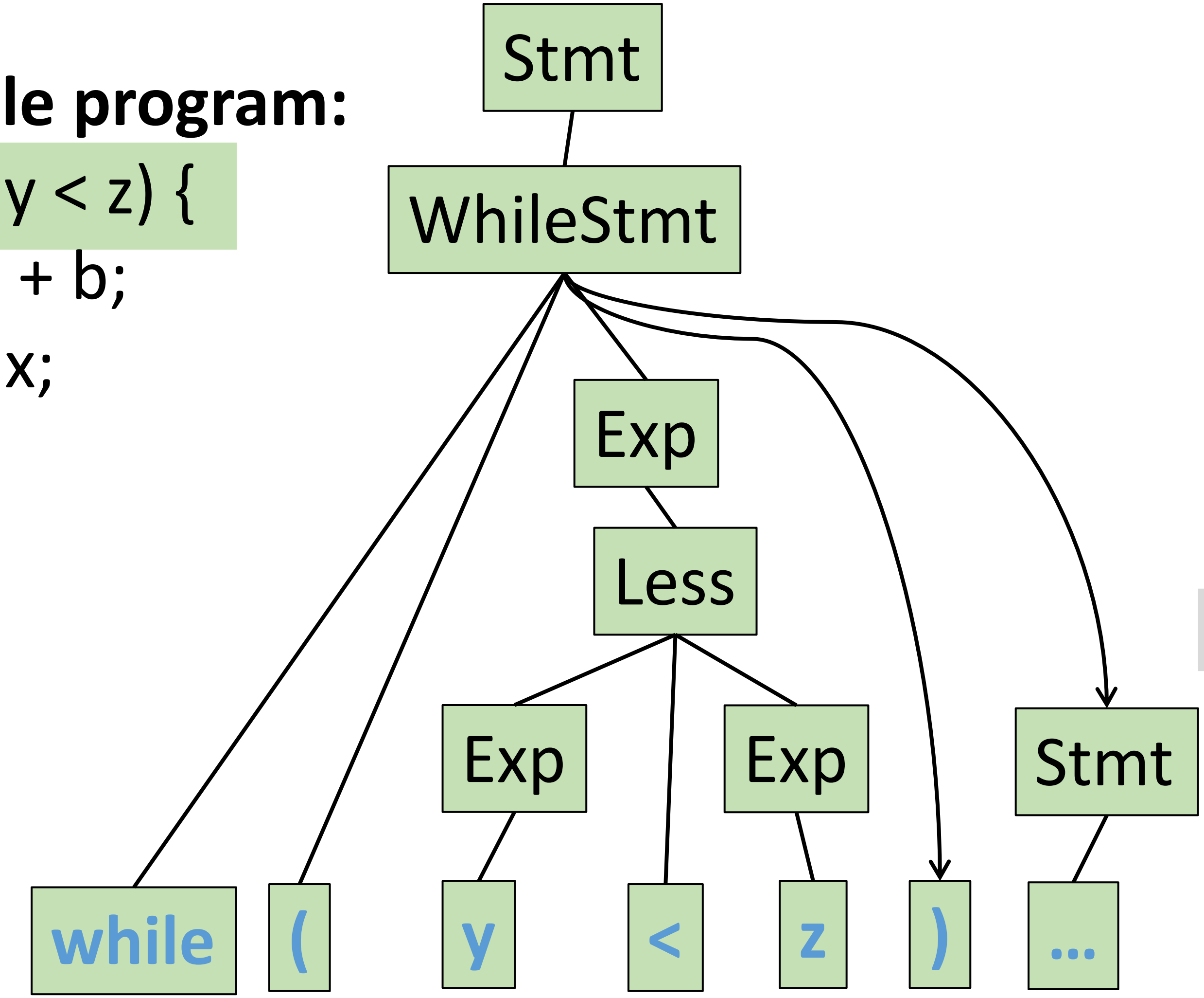
ASTs differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees…



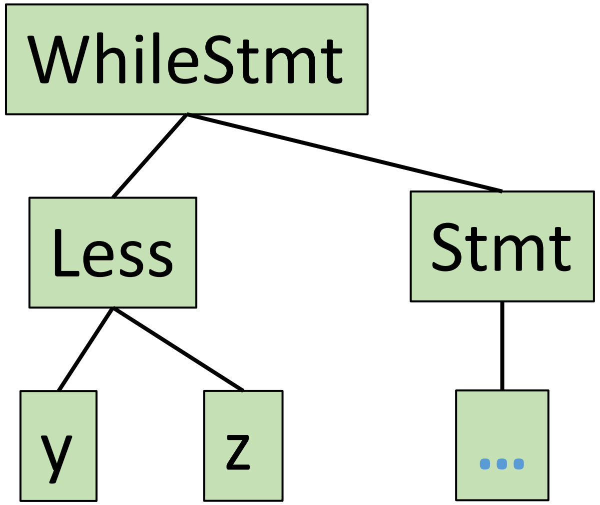
AST: more examples

Example program:

```
while (y < z) {
    x = a + b;
    y += x;
}
```



Parse tree

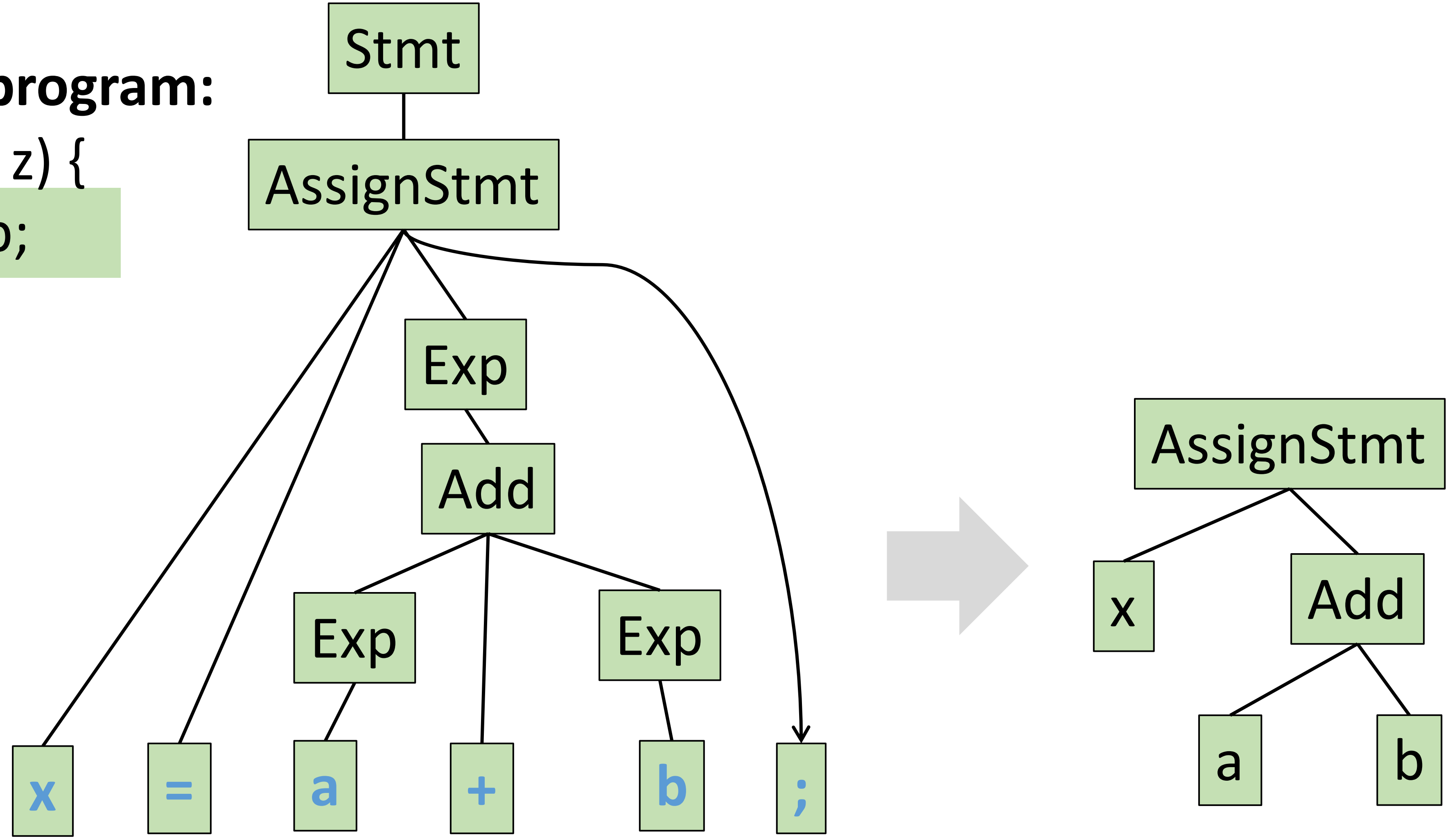


AST

AST: more examples

Example program:

```
while (y < z) {
    x = a + b;
    y += x;
}
```

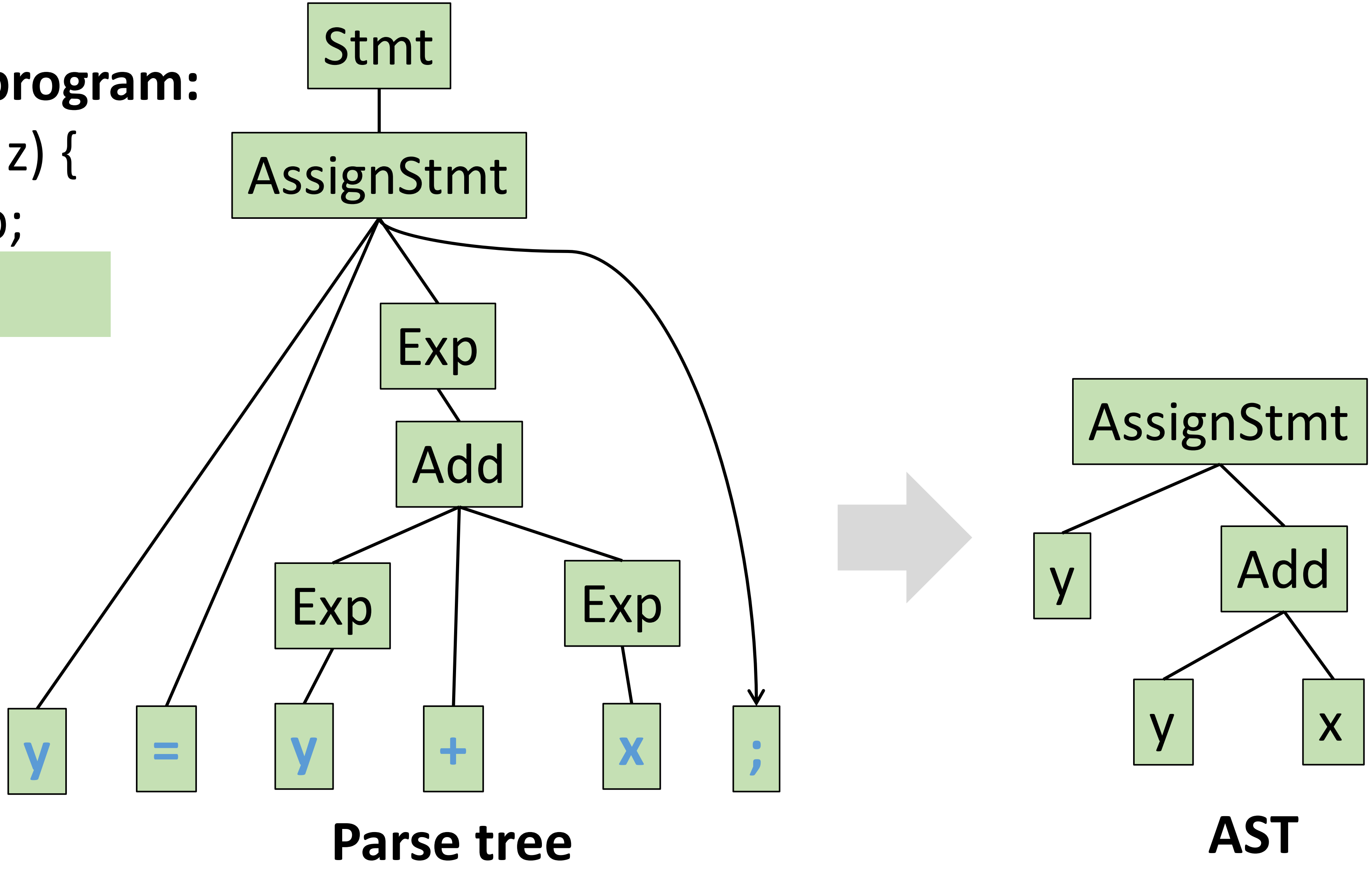


Parse tree **AST**

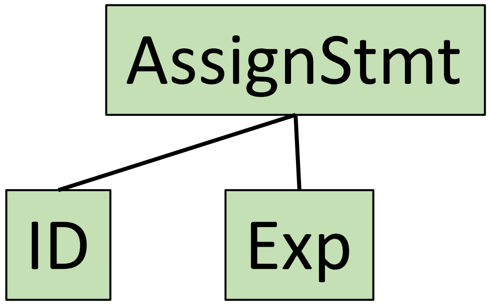
AST: more examples

Example program:

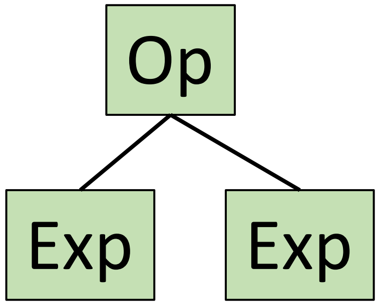
```
while (y < z) {
    x = a + b;
    y += x;
}
```



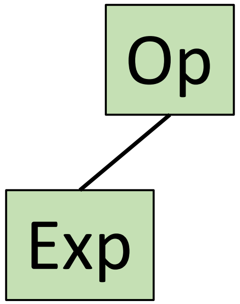
AST: typical structures



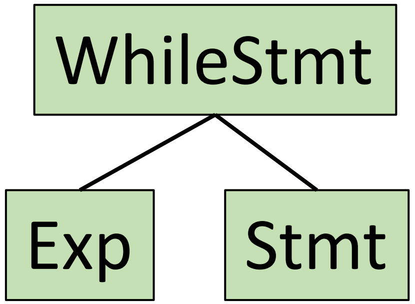
Assignment



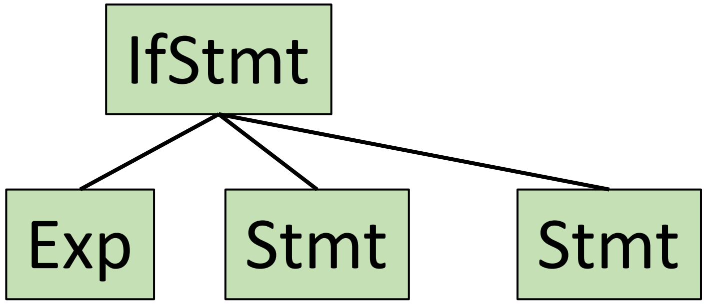
Binary operator



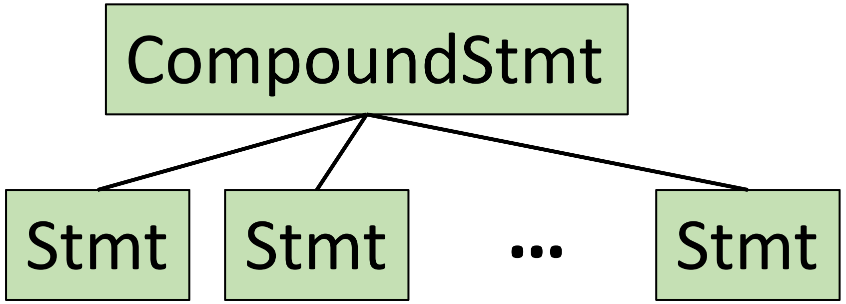
Unary operator



Loop



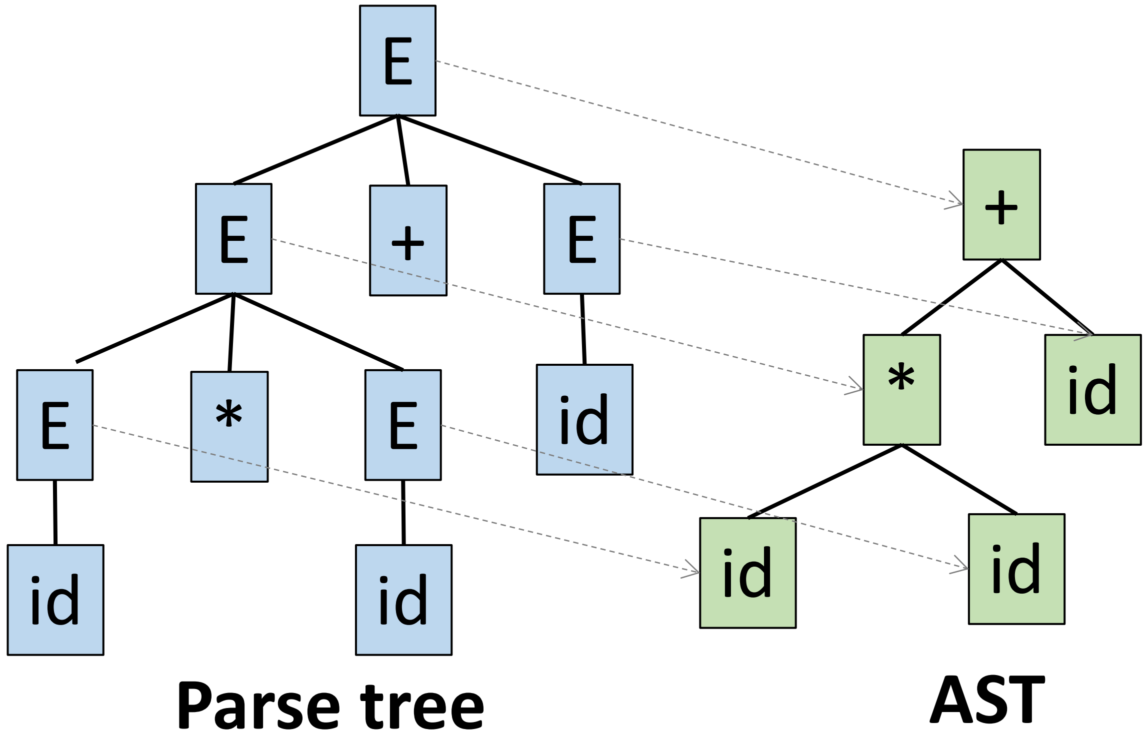
Conditional check



Compound statement

Mapping between parse tree and AST

Production	Semantic Rules
$E \longrightarrow E_1 + E_2$	E.node = new Node("+", E_1.node, E_2.node)
$E \longrightarrow E_1 * E_2$	E.node = new Node("*", E_1.node, E_2.node)
$E \longrightarrow (E_1)$	E.node = E_1.node
$E \longrightarrow id$	E.node = new Leaf(id, id.entry)



AST applications

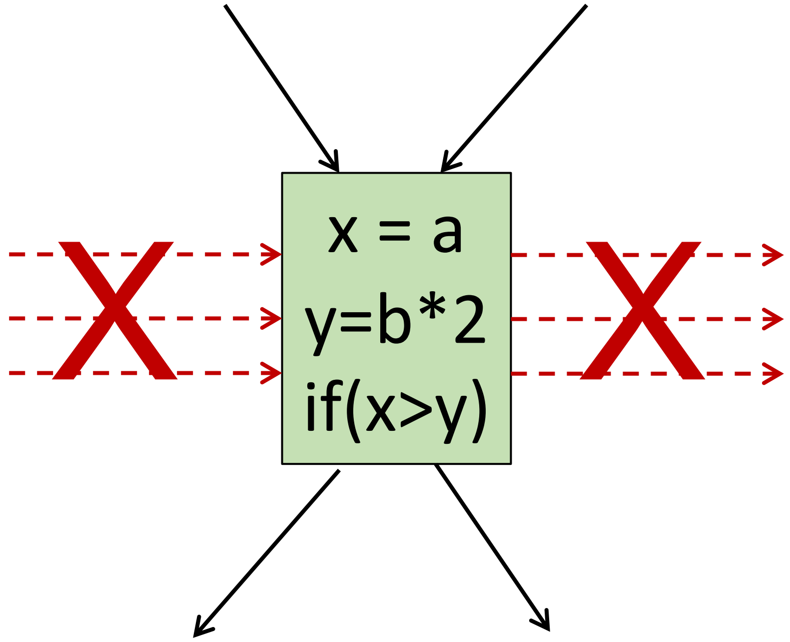
- AST provides a basic model of source code, supporting reading, modifying, and even generating source code in a systematic way
  - Compilers
  - Program analysis
  - Source code instrumentation
  - Automated program repair
  - Code generation
  - …

Topics

- Abstract syntax tree (AST)
- Control-flow graph (CFG)
- Control-flow-based code coverage
- Data-flow analysis
- Data-flow-based code coverage

Basic block

- A basic block is a sequence of straight-line code that can be entered only at the beginning and exited only at the end



Building basic blocks:

1. Identify leaders :
   - The first instruction in a procedure, or
   - The target of any branch, or
   - An instruction immediately following a branch
2. Gobble all subsequent instructions until the next leader

Basic block example

Building basic blocks:

1. Identify leaders :
   - The first instruction in a procedure, or
   - The target of any branch, or
   - An instruction immediately following a branch
2. Gobble all subsequent instructions until the next leader

Program

```
1  while (x < y) {
2     y = f (x, y);
3     if (y == 0) {
4        break;
5     } else if (y<0) {
6        y = y*2;
7        continue;
8     }
9     x = x + 1;
10 }
11 print (y);
```

Leaders

1

2

4

5

6

9

11

Basic blocks

1: while (x<y)

2: y=f(x,y)
3: if(y==0)

4: break

5: else if(y<0)

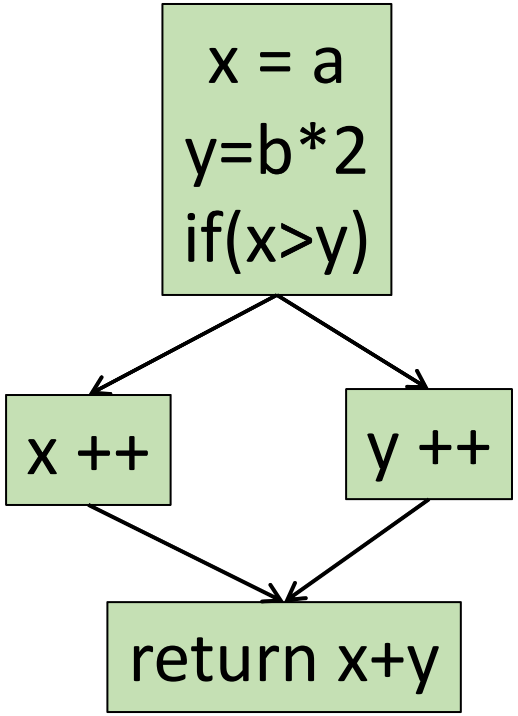
6: y=y*2
7: continue

9: x=x+1

11: print(y)

Control-flow graph (CFG)

- A control-flow graph (CFG) is a rooted directed graph **G=⟨N, E⟩**
  - **N** is the set of basic blocks
  - **E** is the flow of control between basic blocks



Building CFG:

1. Each CFG node represents a basic block
2. There is an edge from node **i** to **j** if
   - Last statement of block **i** branches to the first statement of **j**, or
   - Block **i** is immediately followed in program order by block **j** (fall through)

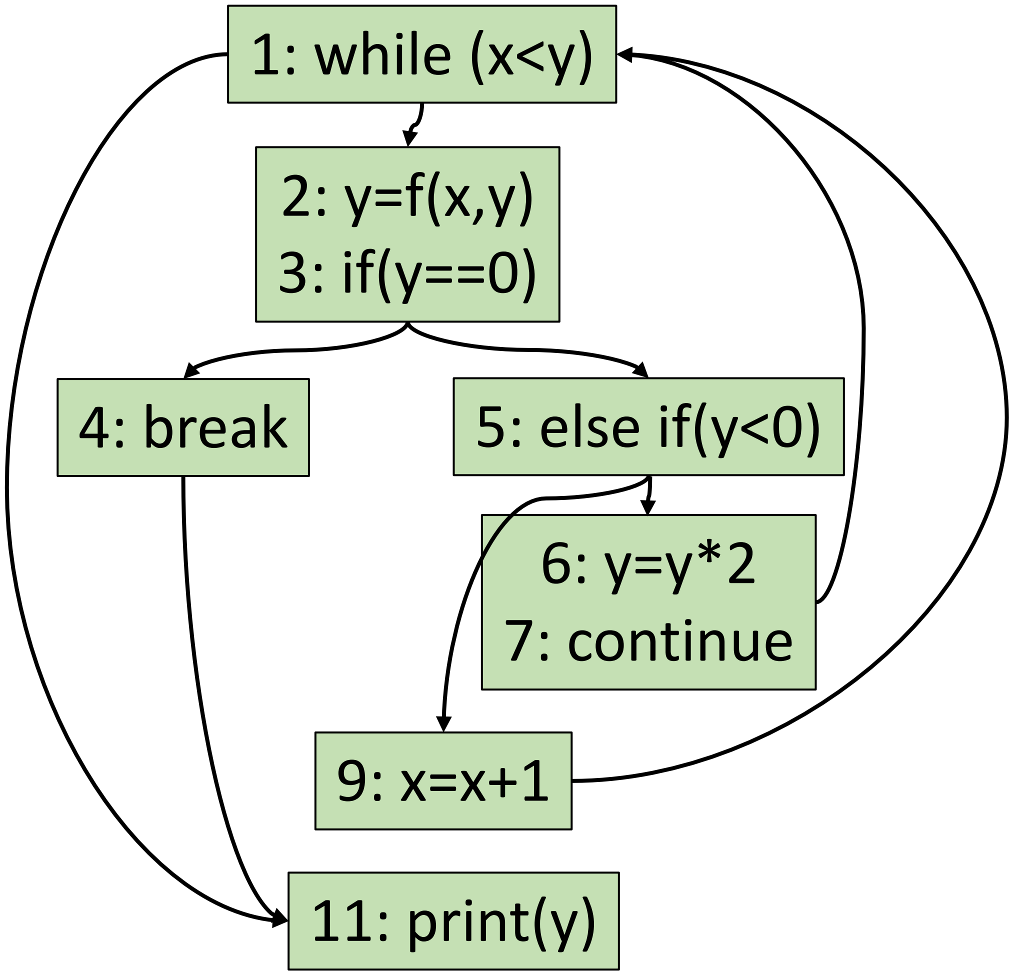
That said, as long as the execution of node **i** could be followed by node **j**, connect them!

CFG example

Program

```
1  while (x < y) {
2     y = f (x, y);
3     if (y == 0) {
4        break;
5     } else if (y<0) {
6        y = y*2;
7        continue;
8     }
9     x = x + 1;
10 }
11 print (y);
```

CFG



Topics

- Abstract syntax tree (AST)
- Control-flow graph (CFG)
- Control-flow-based code coverage
- Data-flow analysis
- Data-flow-based code coverage

Control-flow-based code coverage

- Given the CFG, define a coverage target and write tests to achieve it
  - Higher coverage=> more code portions tested=> potentially better tests!
- **A practical way to measure test quality!**
- Typical control-flow-based code coverage
  - Statement coverage
  - Branch coverage (aka decision coverage)
  - Path coverage
  - Condition coverage
  - Modified condition/decision coverage (MCDC)
  - …

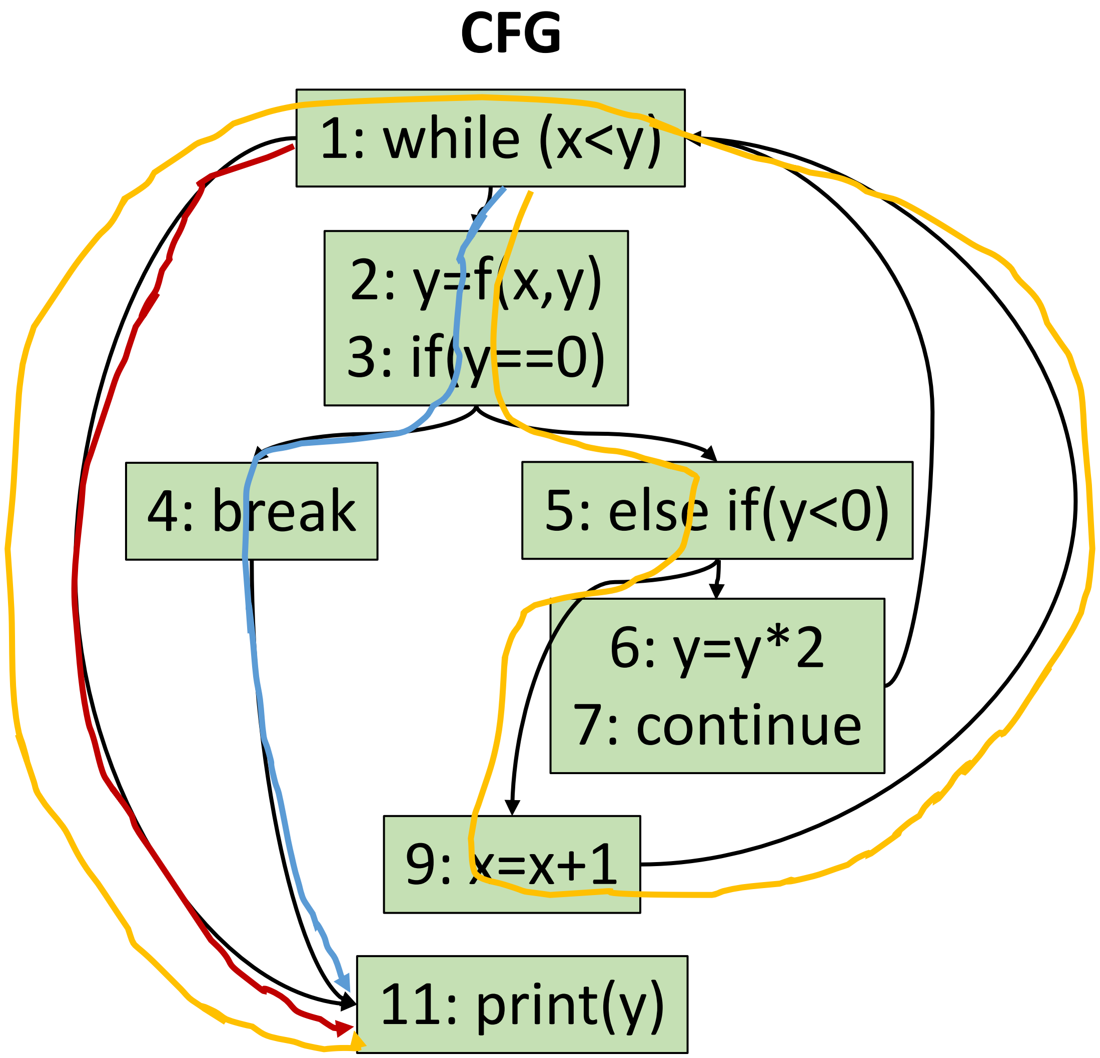
Statement coverage

- **Target**: covering all CFG nodes

Test1: 1-11
Test2: 1-2-3-4-11
Test3: 1-2-3-5-9-1-11

Are they covering all statements?

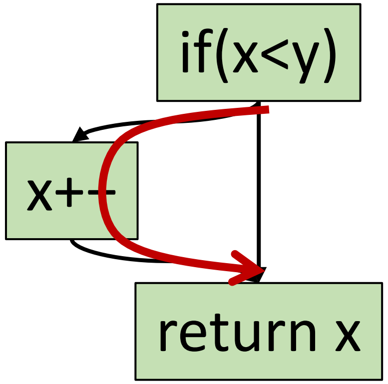
NO, statement coverage: 7/9, statements 6 and 7 never covered!



Branch coverage (decision coverage)

- **Target**: covering all CFG edges
- Equivalent to covering all branches of the predicate nodes
  - True and false branches of each **if** node
  - The two branches corresponding to the condition of a loop
  - All alternatives in a **switch** node
- Is branch coverage equivalent to statement coverage?

```
if (x < y) {
   x++;
}
return x;
```



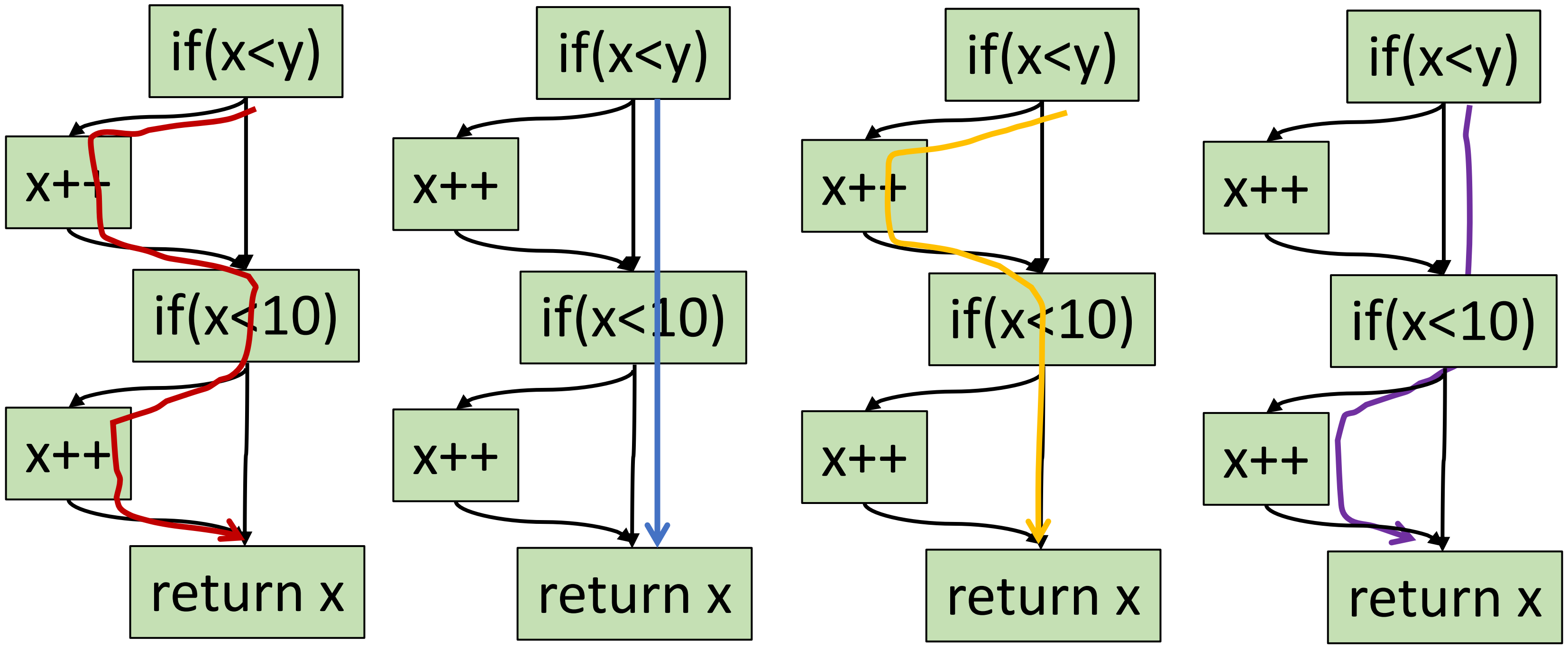
Test1: x=1, y=2

Statement coverage: 3/3

Branch coverage: 1/2

Path coverage

- **Target**: covering all possible paths on CFG
- Is path coverage equivalent to branch coverage?



Test1: x=1, y=2

Test2: x=10, y=2

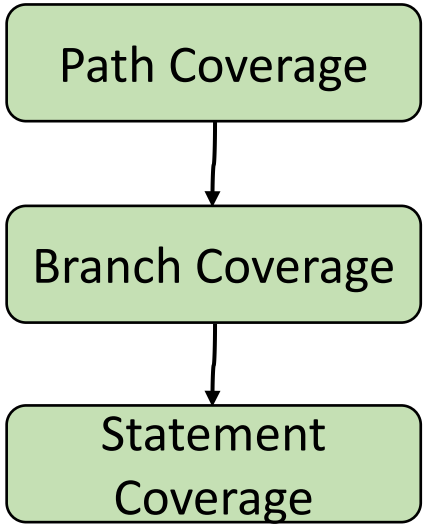
Branch coverage: 4/4

Path coverage: 2/4

The number of paths could be infinite (loops) or exponential (branches)!

Control-flow-based coverage: summary

- Path coverage strictly subsumes branch coverage
- Branch coverage in turn strictly subsumes statement coverage



Coverage subsumption graph

Topics

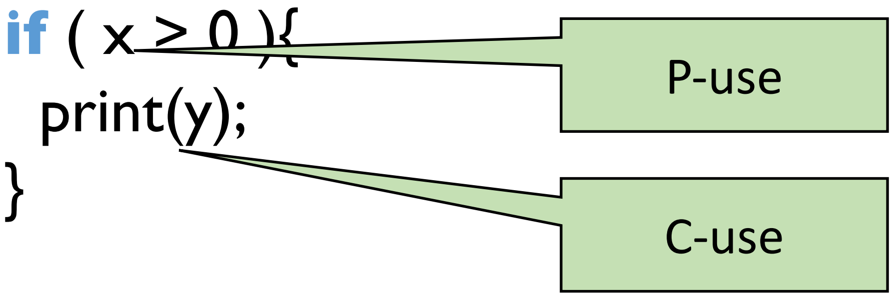
- Abstract syntax tree (AST)
- Control-flow graph (CFG)
- Control-flow-based code coverage
- Data-flow analysis
- Data-flow-based code coverage

Data-flow analysis

- A framework for proving facts (e.g., reaching definitions) about programs
- Operates on control-flow graphs (CFGs), typically
- Works best on properties about how program computes
- Based on all paths through program
  - Including infeasible paths

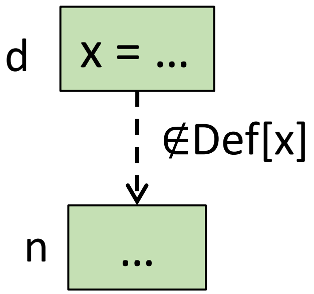
Variable definition/use

- A program variable is **defined** whenever its value is modified:
  - On the left-hand side of an assignment statement: **y = 17**
  - In an input statement: **read(y)**
  - As a call-by-reference parameter in a subroutine call: **update(x, &y)**
- A program variable is **used** whenever its value is read:
  - **P-use** (predicate-use): use in the predicate of a branch statement
  - **C-use** (computation-use): all other uses

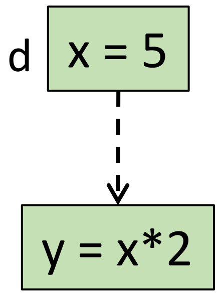


A typical analysis: reaching definitions

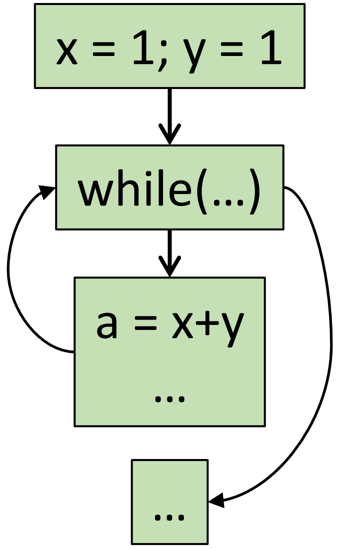
- A definition (statement) **d** of a variable **v** **reaches** CFG node **n** if there is a path from **d** to **n** such that **v** is not redefined along that path



- Reaching definitions applications:
  - Build use/def chains
  - Constant propagation
  - Loop invariant code motion

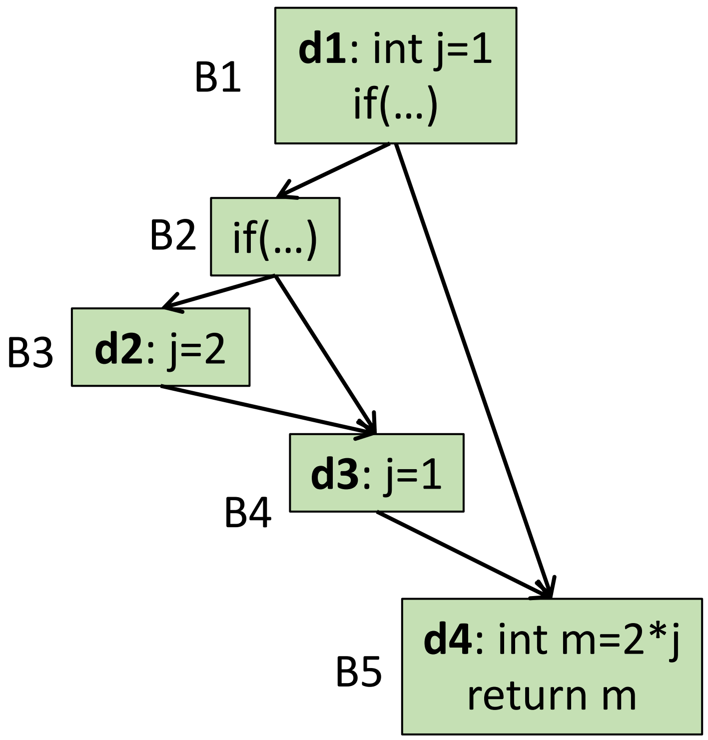


Is this the only def of x reaching n?
Can we replace y=x*2 with y=10?



Any other reaching definitions of x/y in the loop?
Can we move "a=x+y" out of the loop?

Reaching definitions: example

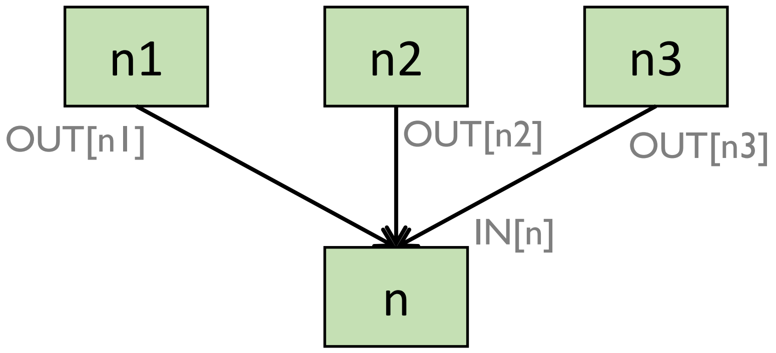


n	IN[n]	OUT[n]
B1	∅	{d1}
B2	{d1}	{d1}
B3	{d1}	{d2}
B4	{d1, d2}	{d3}
B5	{d1, d3}	{d1, d3, d4}

IN[n]: set of facts (reaching definitions) at entry of node n
OUT[n]: set of facts (reaching definitions) at exit of node n

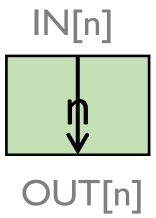
Constant propagation can be applied to B5 as j is always 1!

Reaching definitions: transfer functions



$IN[n]=OUT[n1] \cup OUT[n2] \cup OUT[n3]$

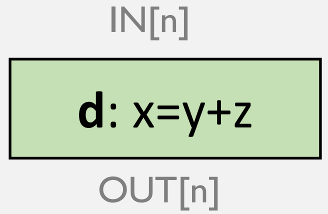
$$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$



KILL[n] = a set of definitions killed by definitions in node n

GEN[n] = a set of locally available definitions in node n

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$



GEN[n] = {d}

KILL[n] = Def[x] – {d}, where Def[x] : set of all definitions of **x**

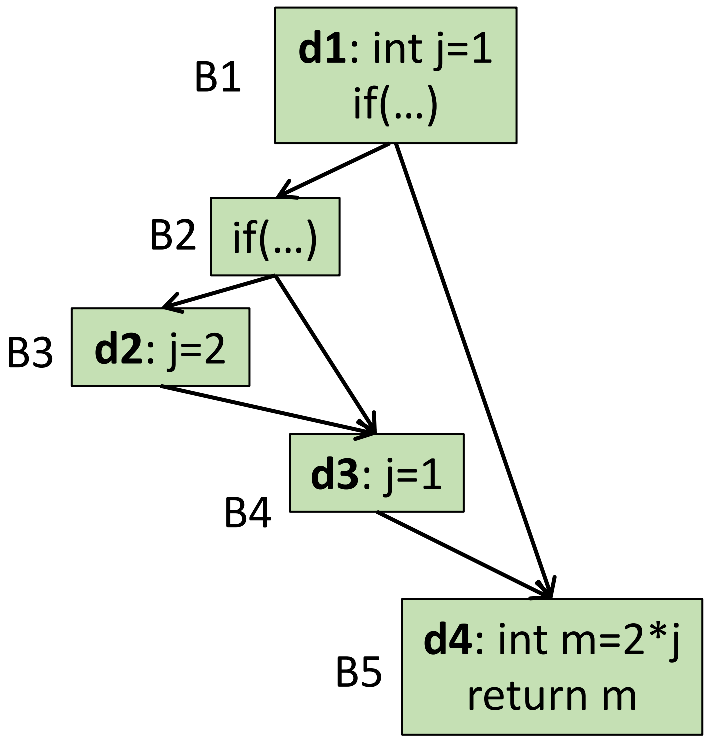
Reaching definitions algorithm

```
for (each node n):
    IN[n] = OUT[n] = ∅

for (each node n):
    IN[n] = ∪ n' ∈ predecessors(n) OUT[n']
    OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]
```

Any issues?

Reaching definitions: example



n	IN[n]	OUT[n]
B1	∅	{d1}
B2	{d1}	{d1}
B3	{d1}	{d2}
B4	{d1}	{d3}
B5		

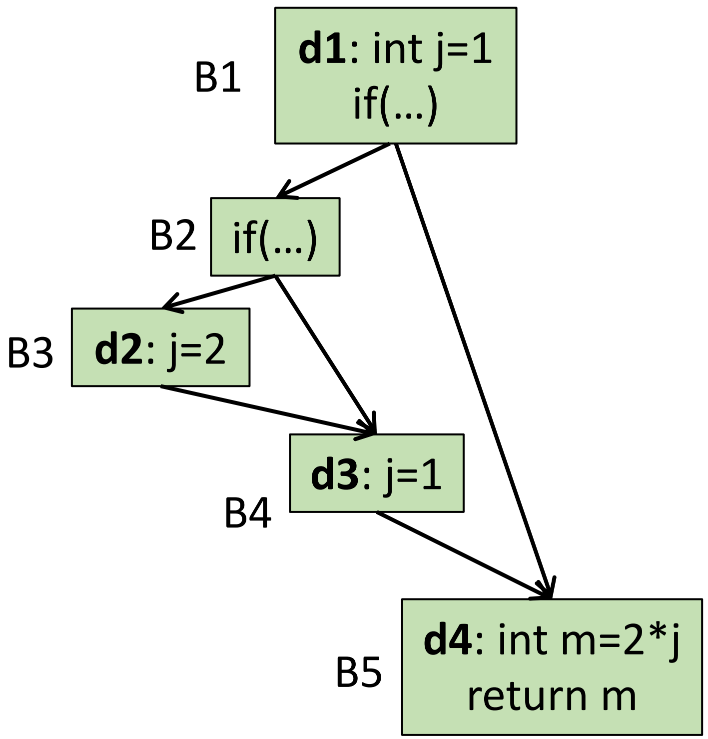
Order matters!

The IN set for B4 is incorrect (should be {d1,d2})!

Reaching definitions algorithm: revised

for (each node n):
    $IN[n] = OUT[n] = \emptyset$
repeat:
for (each node n):
    $IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$
    $OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$
until fixed point: IN[n] and OUT[n] stop changing for all n

Reaching definitions: revisit the example



n	GEN[n]	Kill[n]	IN[n]	OUT[n]
B1	{d1}	{d2, d3}	Ø	{d1}
B2	Ø	Ø	{d1}	{d1}
B3	{d2}	{d1, d3}	{d1}	{d2}
B4	{d3}	{d1, d2}	{d1, d2}	{d3}
B5	{d4}	Ø	{d1, d3}	{d1, d3, d4}

$$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

IN[n] = a set of **reaching definitions** before n
OUT[n] = a set of **reaching definitions** after n
KILL[n] = a set of **definitions** killed by definitions in node n
GEN[n] = a set of locally available **definitions** in node n

Does it always terminate?

The two operations of reaching definitions analysis are monotonic

IN and OUT sets never shrink, only grow

Largest they can be is set of all definitions in program, i.e., finite

IN and OUT cannot grow forever

IN and OUT will stop changing after some iteration

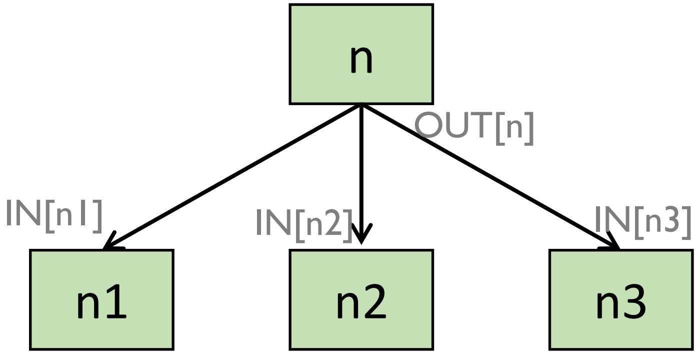
$IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$

$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$

Other classical dataflow analyses

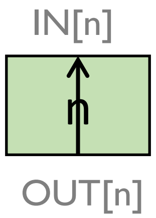
- **Live Variables Analysis:** for dead code elimination
  - Determine for each program point which variables could be *live* at the point's exit
  - A variable is **live** if there is a path to a use of the variable that doesn't redefine the variable
- **Available Expressions Analysis**: for avoiding recomputing expressions
  - Determine, for each program point, which expressions must already have been computed, and not later modified, on all paths to the program point
- **Very Busy Expressions Analysis:** for reducing code size
  - An expression is very *busy* if, no matter what path is taken, the expression is used before any of the variables occurring in it are redefined

Live variables: transfer functions



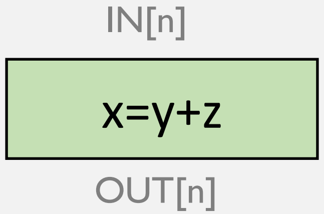
OUT[n]=IN[n1] ∪ IN[n2] ∪ IN[n3]

$$OUT[n] = \bigcup_{n' \in successors(n)} IN[n']$$



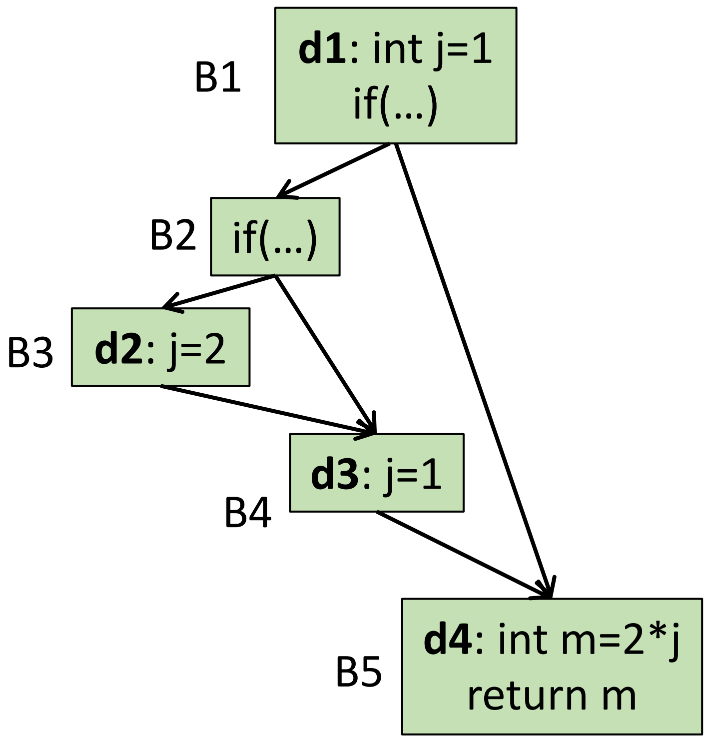
KILL[n] = a set of **variables** defined in node n
GEN[n] = a set of **variables** used in node n

$$IN[n] = (OUT[n] - KILL[n]) \cup GEN[n]$$



GEN[n] = {y, z}
KILL[n] = {x}

Live variables analysis: example



n	GEN[n]	Kill[n]	IN[n]	OUT[n]
B1	∅	{j}	∅	{j}
B2	∅	∅	∅	∅
B3	∅	{j}	∅	∅
B4	∅	{j}	∅	{j}
B5	{j}	{m}	{j}	∅

$$OUT[n] = \bigcup_{n' \in \text{succcessors}(n)} IN[n']$$

$$IN[n] = (OUT[n] - KILL[n]) \cup GEN[n]$$

IN[n] = a set of **live variables** before n
OUT[n] = a set of **live variables** after n
KILL[n] = a set of **variables** defined in node n
GEN[n] = a set of **variables** used in node n

Reaching definitions vs. live variables

- Facts: set of definitions
- Direction: forward
- Join operator: ∪
- Transfer functions:
  - $IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$
  - $OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$

Reaching definitions

- Facts: set of variables
- Direction: backward
- Join operator: ∪
- Transfer functions:
  - $OUT[n] = \bigcup_{n' \in succcessors(n)} IN[n']$
  - $IN[n] = (OUT[n] - KILL[n]) \cup GEN[n]$

Live variables

Classifying all four dataflow analyses

	May	**Must**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

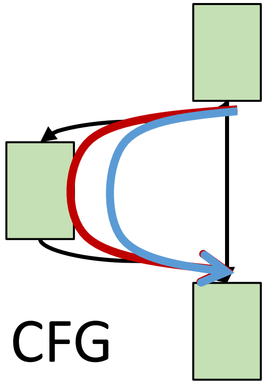
- Forward = Data flow from in to out
- Backward = Data flow from out to in
- Must = At join point, property must hold on all paths that are joined
- May = At join point, property may hold on some paths that are joined

Topics

- Abstract syntax tree (AST)
- Control-flow graph (CFG)
- Control-flow-based code coverage
- Data-flow analysis
- Data-flow-based code coverage

Dataflow-based code coverage

- Why another family of code coverage?



CFG

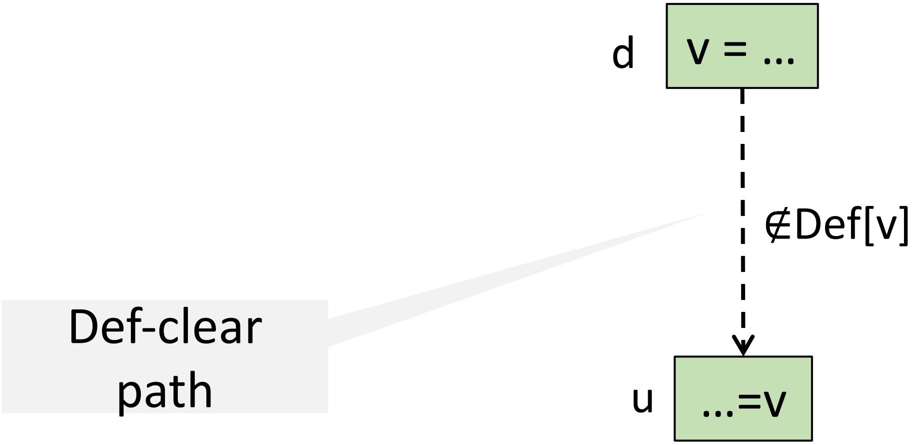
Are test1 and test2 always identical?

Although the paths are the same, different tests may have different variable values defined/used!

- A family of dataflow criteria is then defined, each providing a different degree of **data** coverage
  - Existing control-flow coverage criteria only consider the execution paths (**structure**)
  - In the program paths, which variables are defined and then used should also be covered (**data**)

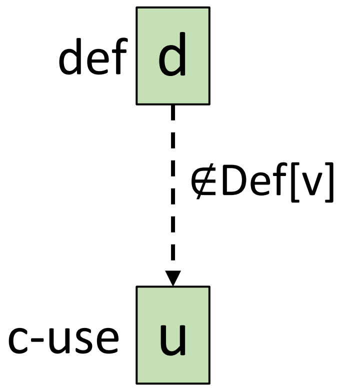
Def-clear path

- A path $\langle \mathbf{d, n_1, \ldots, n_m, u} \rangle$ is a **def-clear path** from **d** to **u** with respect to **v** if it has no variable re-definition of **v** on the path
  - I.e., the definition of **v** at **d** can reach **u**



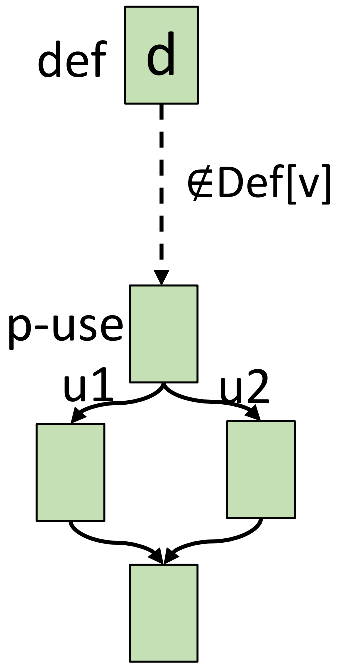
DU-pair

- A **DU-pair** with respect to a variable **v** is a pair (**d**,**u**) such that
  - **d** is a node defining **v**
  - **u** is a node or edge using **v**
    - When it is a p-use of **v**, **u** is an outgoing edge of the predicate statement
  - There is a def-clear path with respect to **v** from **d** to **u**



c-use

Du pair:
(d, u)

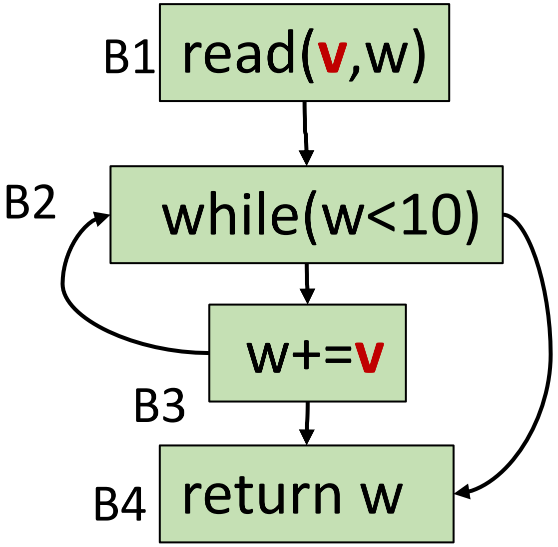


p-use

Du pairs:
(d, u1)
(d, u2)

DU-path

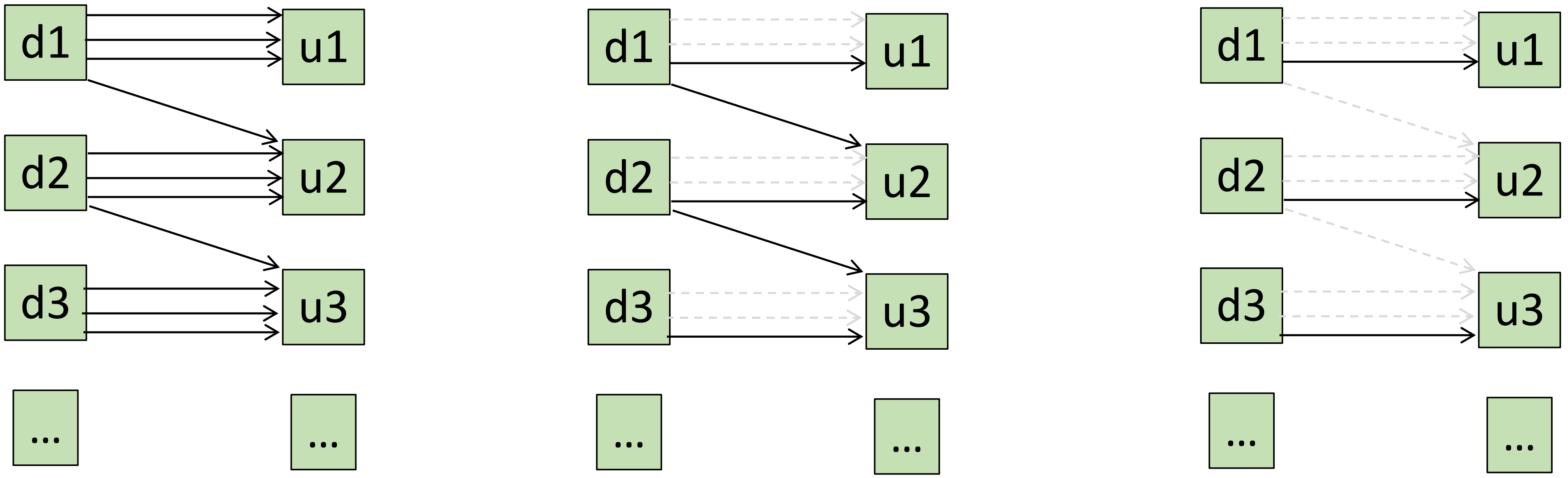
- A path $\langle \mathbf{n_1, \ldots, n_j, n_k} \rangle$ is a **DU-path** for variable **v** if $\mathbf{n_1}$ contains a definition of **v** and either
  - $\mathbf{n_k}$ is a c-use of **v** and $\langle \mathbf{n_1, \ldots, n_j, n_k} \rangle$ is a def-clear **simple** path for **v** (all nodes, except possibly **d** and **u**, are distinct), or
  - $\langle \mathbf{n_j, n_k} \rangle$ is a p-use of **v** and $\langle \mathbf{n_1, \ldots, n_j} \rangle$ is a def-clear **loop-free** path for x (all nodes are distinct)



Def-clear paths	**DU paths**
1-2-3	1-2-3
1-2-3-2-3	~~1-2-3-2-3~~
1-2-3-2-3-2-3	~~1-2-3-2-3-2-3~~
1-2-3-2-3-2-3-2-3	~~1-2-3-2-3-2-3-2-3~~
…	…

Typical dataflow-based coverage

- Identify all DU pairs and construct test cases that cover these pairs
  - Variations with different “strength”

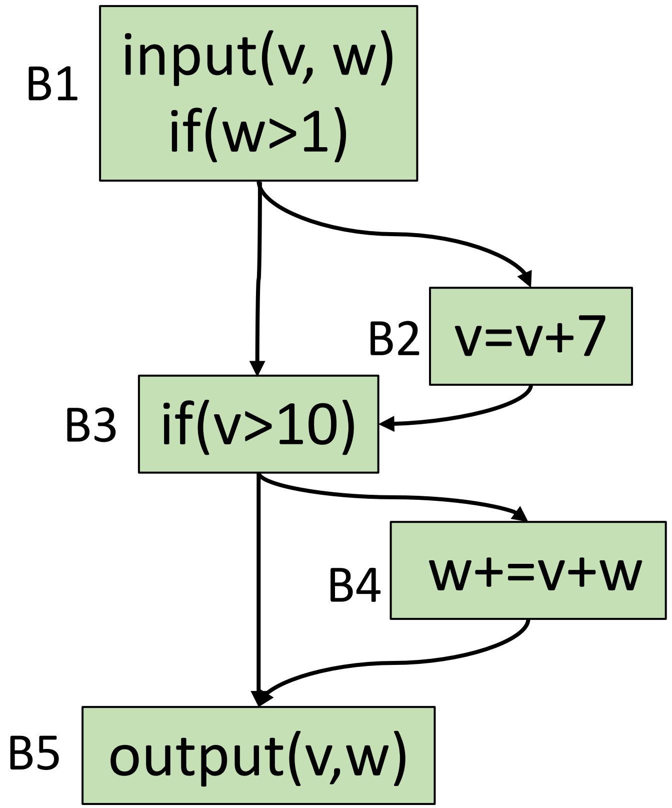


All-DU-Paths **All-Uses** **All-Defs**

Typical dataflow-based coverage: definitions

- **All-DU-paths**: for every du-pair **(d, u)** of every variable **v**, cover all possible def-clear DU paths from **d** to **u**
- **All-Uses**: for every du-pair **(d, u)** of every variable **v**, cover at least one def-clear path from **d** to **u**
- **All-Defs:** for each definition **d** of each variable **v**, cover at least one du-pair for **d**

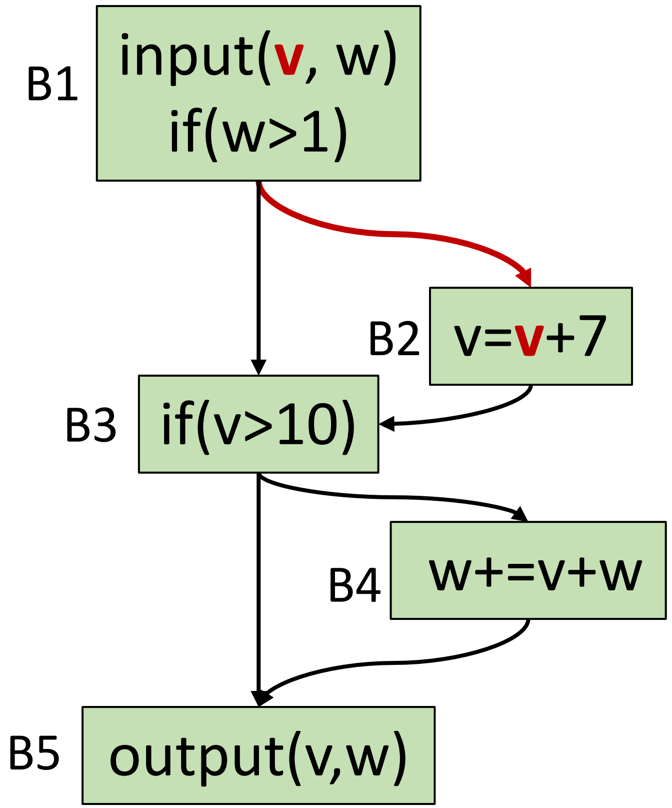
Typical dataflow-based coverage: example



du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>

With respect to variable **v**
(**w** should be analyzed similarly)

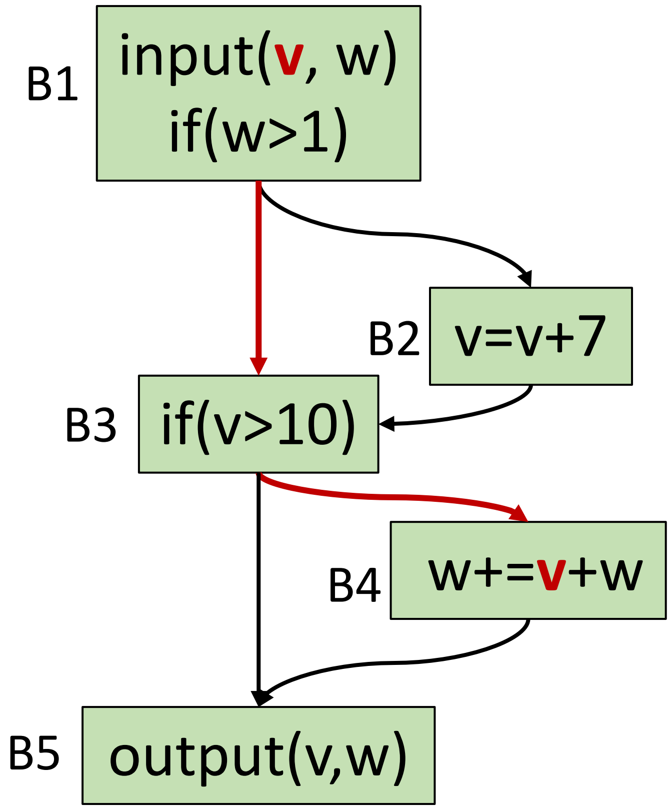
Typical dataflow-based coverage: example



du-pair	path(s)
(1,2)	**<1,2>**
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>

With respect to variable **v**
(**w** should be analyzed similarly)

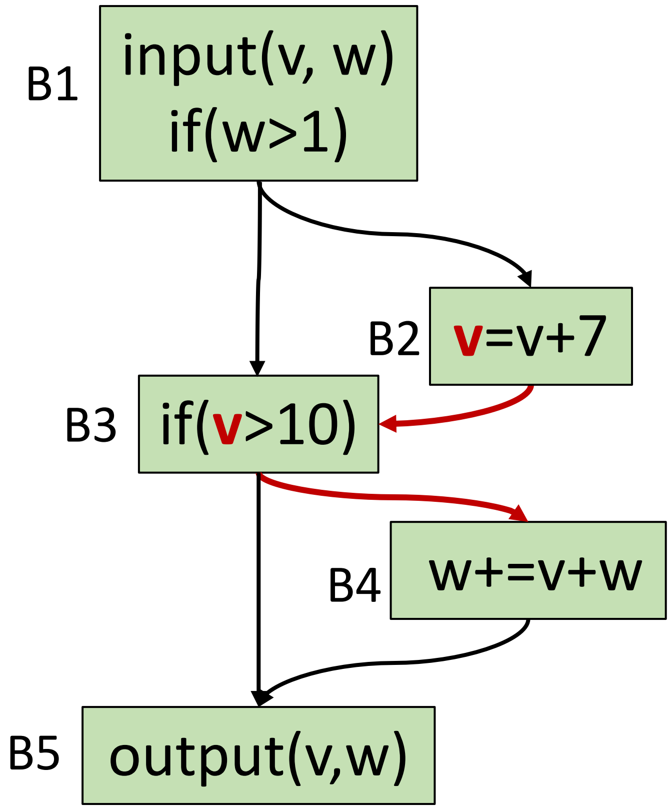
Typical dataflow-based coverage: example



du-pair	**path(s)**
(1,2)	<1,2>
(1,4)	**<1,3,4>**
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>

With respect to variable **v**
(**w** should be analyzed similarly)

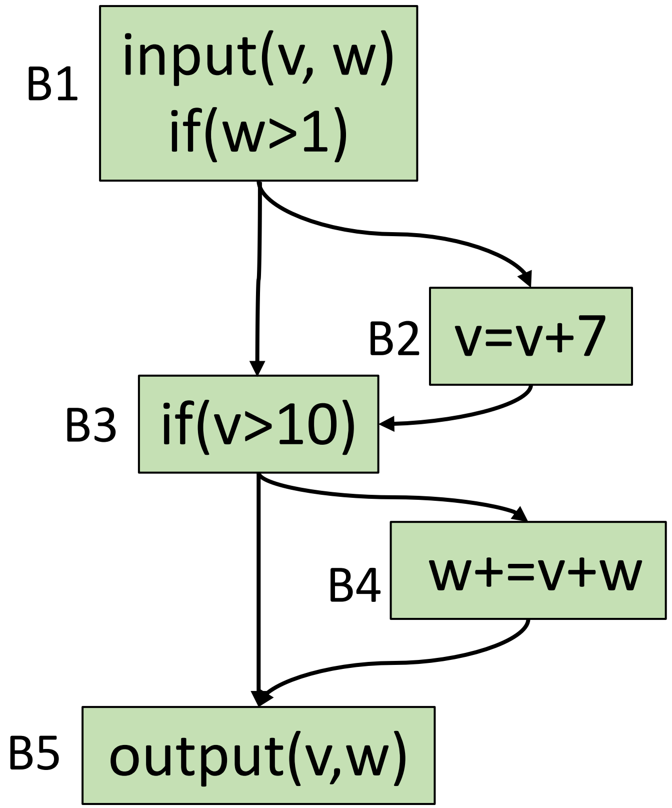
Typical dataflow-based coverage: example



du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	**<2,3,4>**
(2,<3,5>)	<2,3,5>

With respect to variable **v**
(**w** should be analyzed similarly)

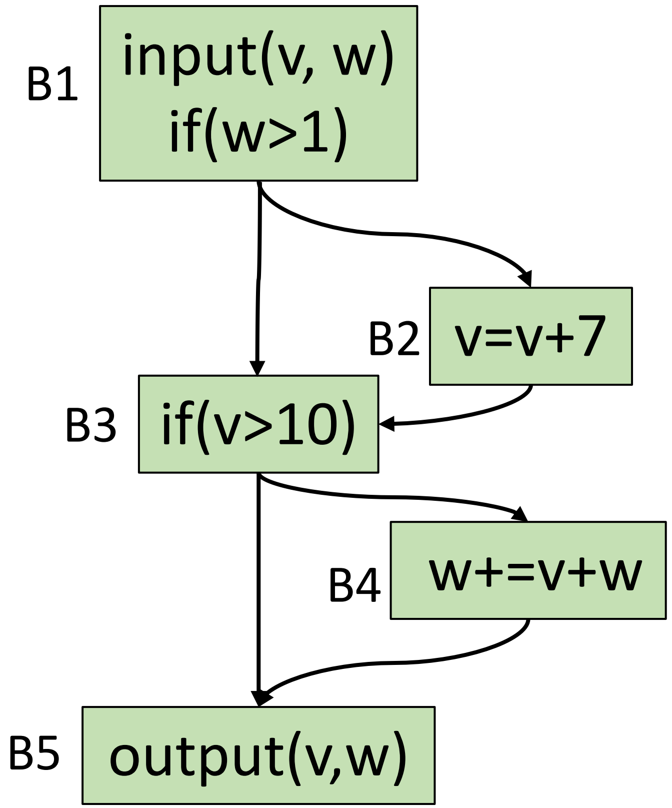
Typical dataflow-based coverage: example



du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>

With respect to variable **v**
(**w** should be analyzed similarly)

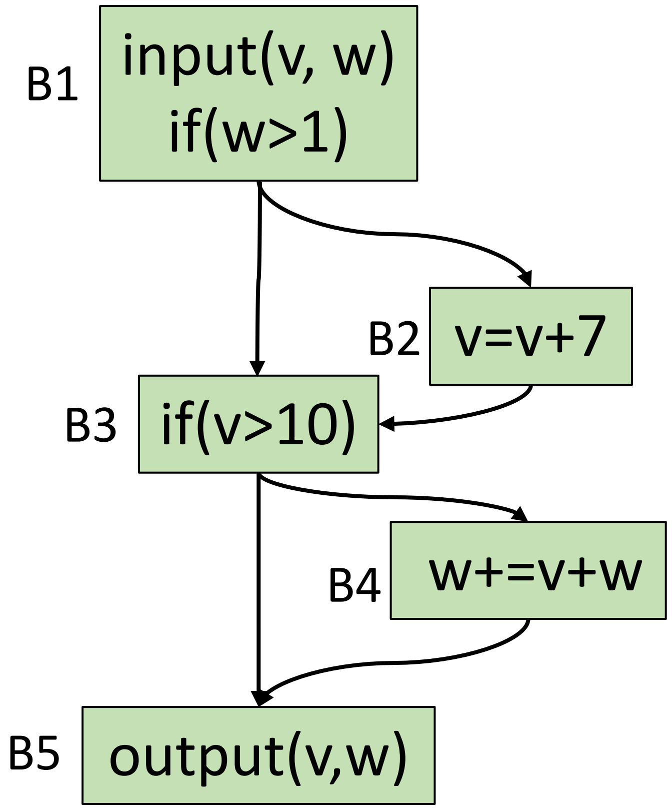
Typical dataflow-based coverage: example



du-pair	path(s)	All-Defs
(1,2)	<1,2>	X
(1,4)	<1,3,4>	
(1,5)	<1,3,4,5>	
	<1,3,5>	
(1,<3,4>)	<1,3,4>	
(1,<3,5>)	<1,3,5>	
(2,4)	<2,3,4>	X
(2,5)	<2,3,4,5>	
	<2,3,5>	
(2,<3,4>)	<2,3,4>	
(2,<3,5>)	<2,3,5>	

With respect to variable **v**
(**w** should be analyzed similarly)

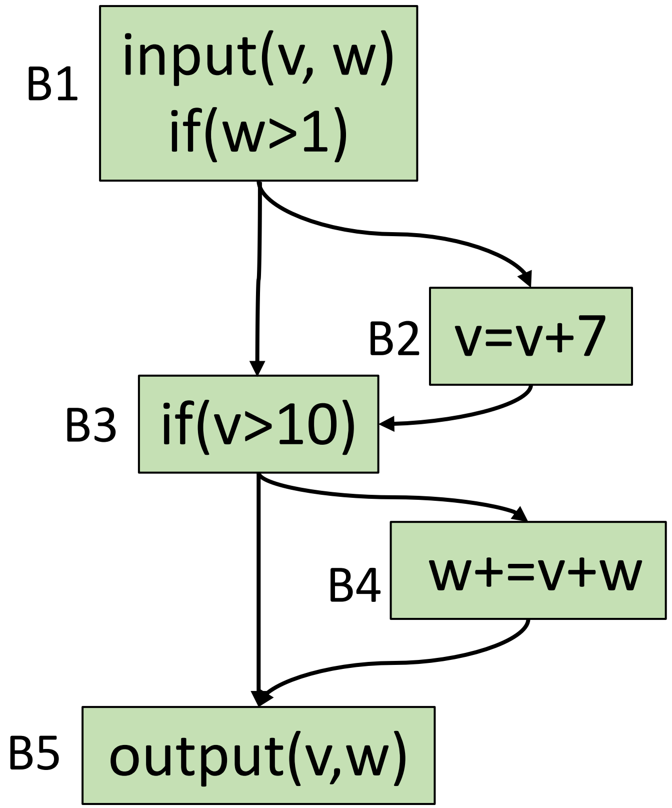
Typical dataflow-based coverage: example



du-pair	path(s)	All-Defs	All-Uses
(1,2)	<1,2>	X	X
(1,4)	<1,3,4>		X
(1,5)	<1,3,4,5>		X
	<1,3,5>		
(1,<3,4>)	<1,3,4>		X
(1,<3,5>)	<1,3,5>		X
(2,4)	<2,3,4>	X	X
(2,5)	<2,3,4,5>		X
	<2,3,5>		
(2,<3,4>)	<2,3,4>		X
(2,<3,5>)	<2,3,5>		X

With respect to variable **v**
(**w** should be analyzed similarly)

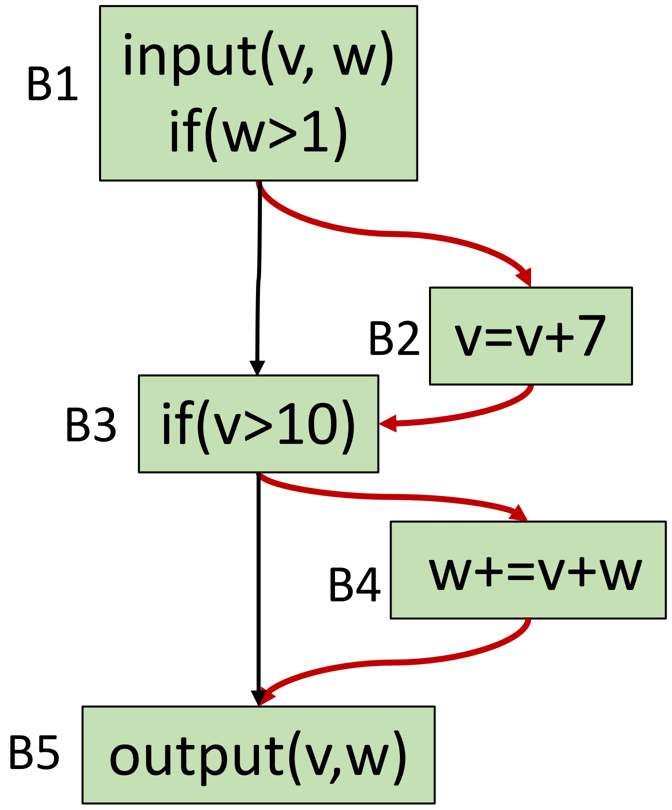
Typical dataflow-based coverage: example



du-pair	**path(s)**	**All-Defs**	**All-Uses**	**All-DU-Paths**
(1,2)	<1,2>	X	X	X
(1,4)	<1,3,4>		X	X
(1,5)	<1,3,4,5>		X	X
	<1,3,5>			X
(1,<3,4>)	<1,3,4>		X	X
(1,<3,5>)	<1,3,5>		X	X
(2,4)	<2,3,4>	X	X	X
(2,5)	<2,3,4,5>		X	X
	<2,3,5>			X
(2,<3,4>)	<2,3,4>		X	X
(2,<3,5>)	<2,3,5>		X	X

With respect to variable **v**
(**w** should be analyzed similarly)

Typical dataflow-based coverage: example



du-pair	**path(s)**	**Covered**
(1,2)	<1,2>	X
(1,4)	<1,3,4>	
(1,5)	<1,3,4,5>	
	<1,3,5>	
(1,<3,4>)	<1,3,4>	
(1,<3,5>)	<1,3,5>	
(2,4)	<2,3,4>	X
(2,5)	<2,3,4,5>	X
	<2,3,5>	
(2,<3,4>)	<2,3,4>	X
(2,<3,5>)	<2,3,5>	

With respect to variable **v**
(**w** should be analyzed similarly)

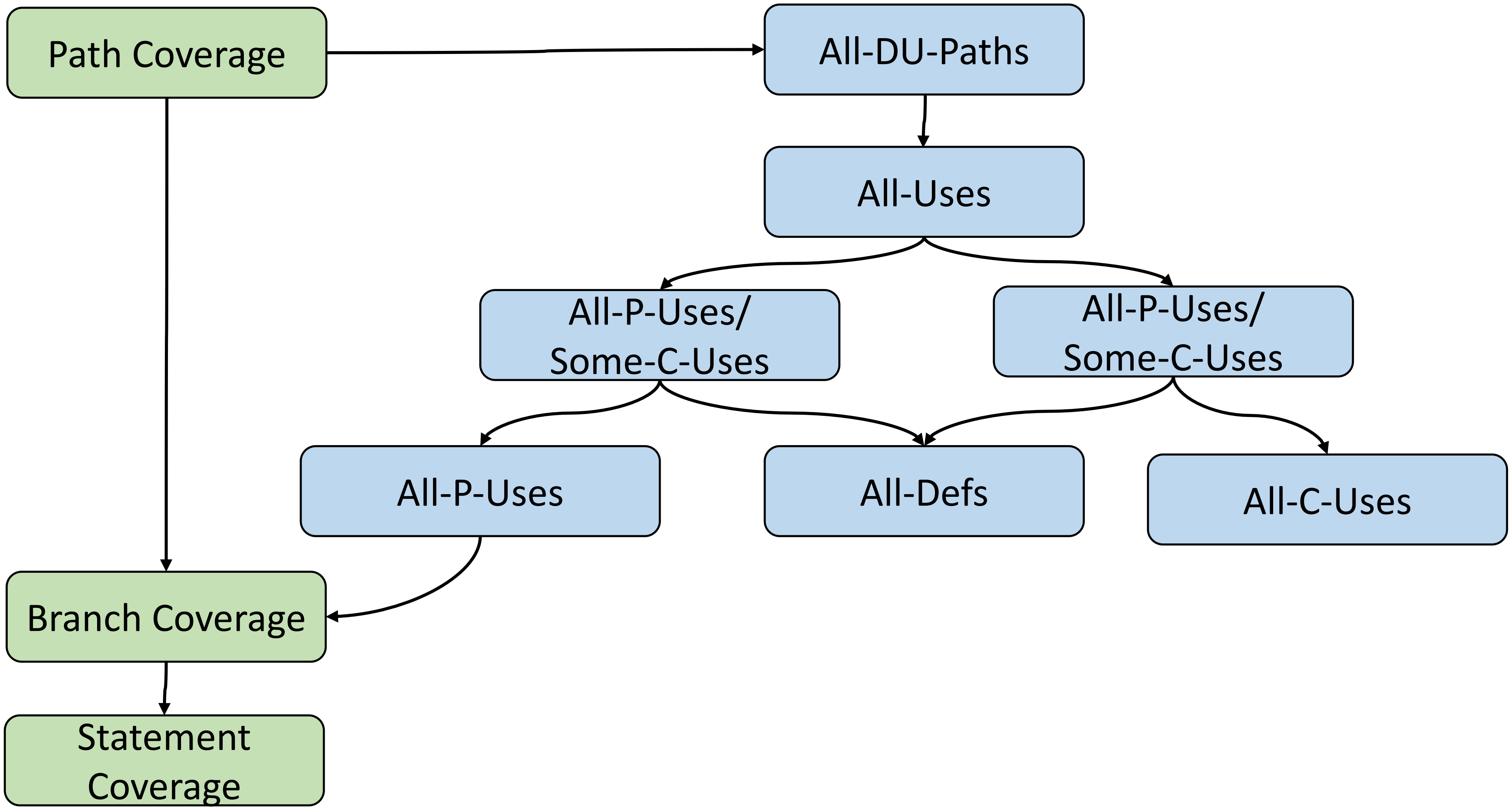
Test1: 1-2-3-4-5

Only All-Defs,
needs more tests!

More dataflow coverage

- **All-P-Uses/Some-C-Uses**: for each definition **d** of each variable **v**, cover at least one def-clear path from **d** to ***any*** p-use of **v**
  - If no p-use of **v**, at least one def-clear path to ***one*** c-use of **v** must be covered
- **All-C-Uses/Some-P-Uses**: for each definition **d** of each variable **v**, cover at least one def-clear path from **d** to ***any*** c-use of **v**
  - If no c-use of **v**, at least one def-clear path to ***one*** p-use of **v** must be covered
- **All-P-Uses**: for each definition **d** of each variable **v**, cover at least one def-clear path from **d** to ***any*** p-use of **v**
- **All-C-Uses**: for each definition **d** of each variable **v**, cover at least one def-clear path from **d** to ***any*** c-use of **v**

Coverage subsumption graph



Interprocedural analysis

- So far, all the analyses we covered are **intraprocedural**
  - Analyzing each function (a.k.a, method/procedure) separately
- However, real-world programs usually involve the connection of a large number of functions, thus we need **interprocedural** analysis:
  - **Call-graph analysis**: analyzing the potential invocation relationship between different functions [Tip et al.]
  - **Interprocedural CFG**: connecting intraprocedural CFGs with call-graph
  - **Interprocedural dataflow analysis**: analyzing dataflow across functions [Reps et al.]
  - **Taint analysis**: tracking how private information flows through the program and if it is leaked to public observers [Arzt et al.]

- ❑ Tip et al., Scalable Propagation-Based Call Graph Construction Algorithms, 2000, OOPSLA
- ❑ Reps et al., Precise Interprocedural Dataflow Analysis via Graph Reachability, 1987, POPL
- ❑ Arzt et al., FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, 2014, PLDI

Do I need to implement such basic program analyses from scratch?

- Java
  - ASM (https://asm.ow2.io/)
    - A lightweight bytecode-level analysis and manipulation framework
  - Soot (https://github.com/soot-oss/soot)
    - An Intermediate Representation (IR) level analysis and manipulation framework
  - Wala (https://github.com/wala/WALA)
    - An IR-level analysis and manipulation (via Shrike) framework for Java and JavaScript
  - Eclipse JDT (https://www.eclipse.org/jdt/)
    - A source-level code analysis and manipulation framework
- C/C++
  - LLVM (http://llvm.org/)
    - Highly customizable and modular compiler framework

Further readings

- Aho et al., Compilers: Principles, Techniques, and Tools (2nd Edition)
- Rapps and Weyuker. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, 11(4), April 1985, pp. 367-375
- Ferrante et al., The program dependence graph and its use in optimization, 1987, TOPLAS
- Horwitz et al., Interprocedural slicing using dependence graphs, 1988, PLDI

Thanks and stay safe!