Dataflow Analysis

Introduction

1. Software analysis is very diverse
   - Many approaches with different strengths and limitations
     - Soundness
     - Completeness
     - Applicability
     - Scalability
   - General technique for designing a dataflow analysis

What is Dataflow Analysis?

1. Static analysis for reasoning about flow of data in a program
   - Different kinds of data
     - Constants: 7
     - Variables: foo
     - Expressions: 7 * foo
   - Used by bug-finding tools and compilers

The While Language

1. Backus-Naur grammar
   - Statement: assignment, if, else, while
   - Arithmetic expression
   - Boolean expression
   - Integer variable
   - Integer constant
   - No functions, pointers, or threads
2. Presence of loops makes While expressive enough that interesting properties of programs written in this language are undecidable but simple enough

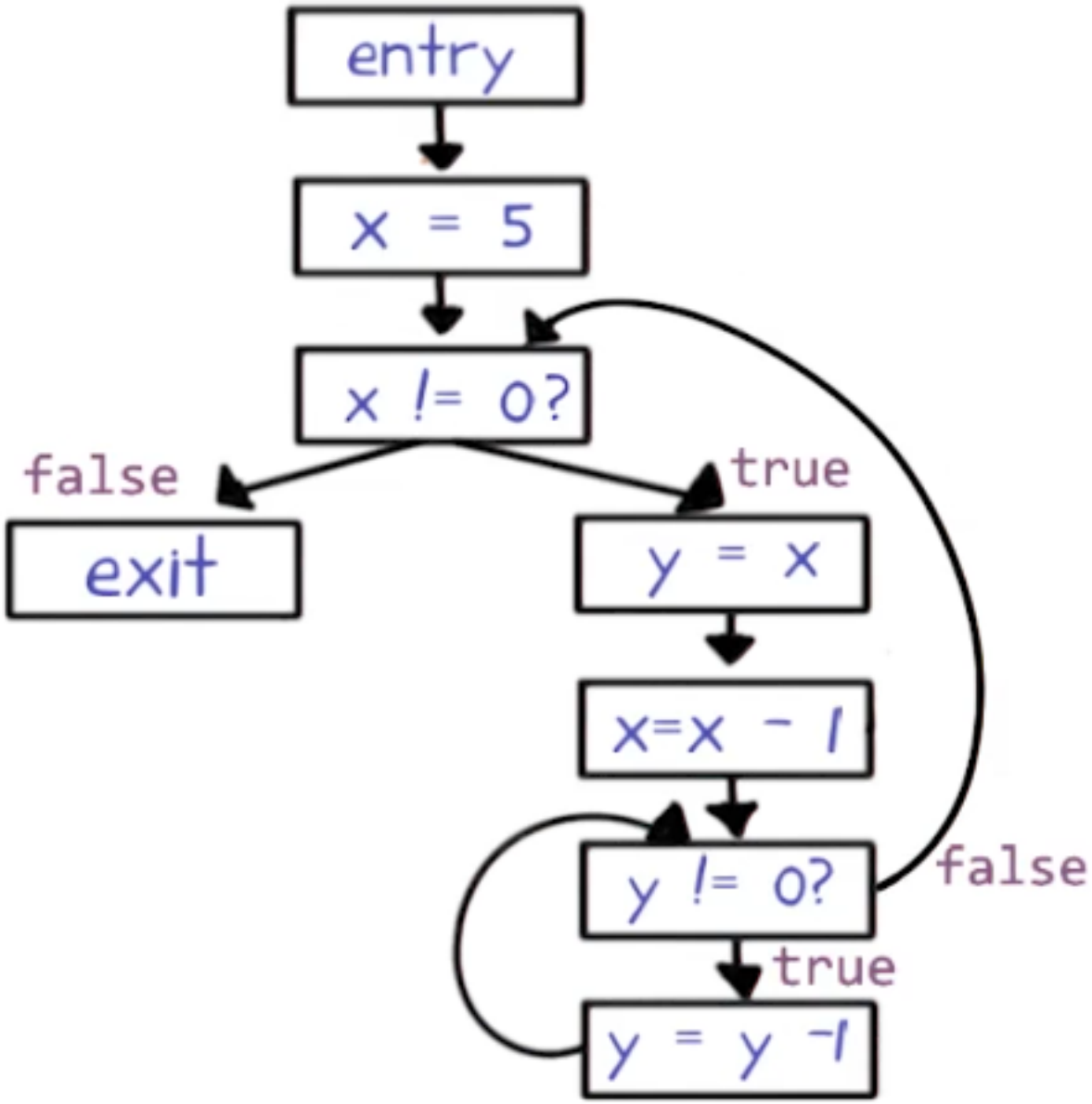
```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1;
}
```

Control-Flow Graphs

1. Dataflow analysis typically operates on the intermediate representation of the program

Control-Flow Graphs 2

1. Write the program corresponding to the following control flow graph



Control Flow

```
x = 5;
while (x != 0){
    y = x;
    x = x - 1;
    while (y != 0){
        y = y - 1;
    }
}
```

Soundness, Completeness, and Termination

1. Impossible for analysis to achieve all three together

- Dataflow analysis sacrifices completeness, but guarantee soundness and termination
- Sound: Will report all facts that could occur in actual runs
- Incomplete: May report additional facts that can't occur in actual runs

Abstracting Control-Flow Conditions

1. Abstracts away control-flow conditions with non-deterministic choice
   - Non-deterministic choice: Assumes condition can evaluate to true or false
   - Considers all paths possible in actual runs (sound) and maybe paths that are never possible (incomplete)

Applications of Dataflow Analysis

1. Reaching Definitions Analysis
   - Find usage of uninitialized variables
2. Very Bust Expressions Analysis
   - Reduce code size
3. Available Expressions Analysis
   - Avoid recomputing expressions
4. Live Variables Analysis
   - Allocate registers efficiently

Reaching Definitions Analysis

1. Goal
   - Determine, for each program point, which assignments have been made and not overwritten, when execution reaches that point along some path
     - "Assignment" == "Definition"
   - Go until the variable is overwritten

Reaching Definitions Analysis 2

1. Consider the following program:

```
x = 5;
y = 1;
while (x != 1) { // P1
    y = x * y;
    x = x - 1; // P2
}
```

2. Which of the following are true?
   - The assignment y = 1 reaches P1 (true)
   - The assignment y = 1 reaches P2 (false)
   - The assignment y = x * y reaches P1 (true)

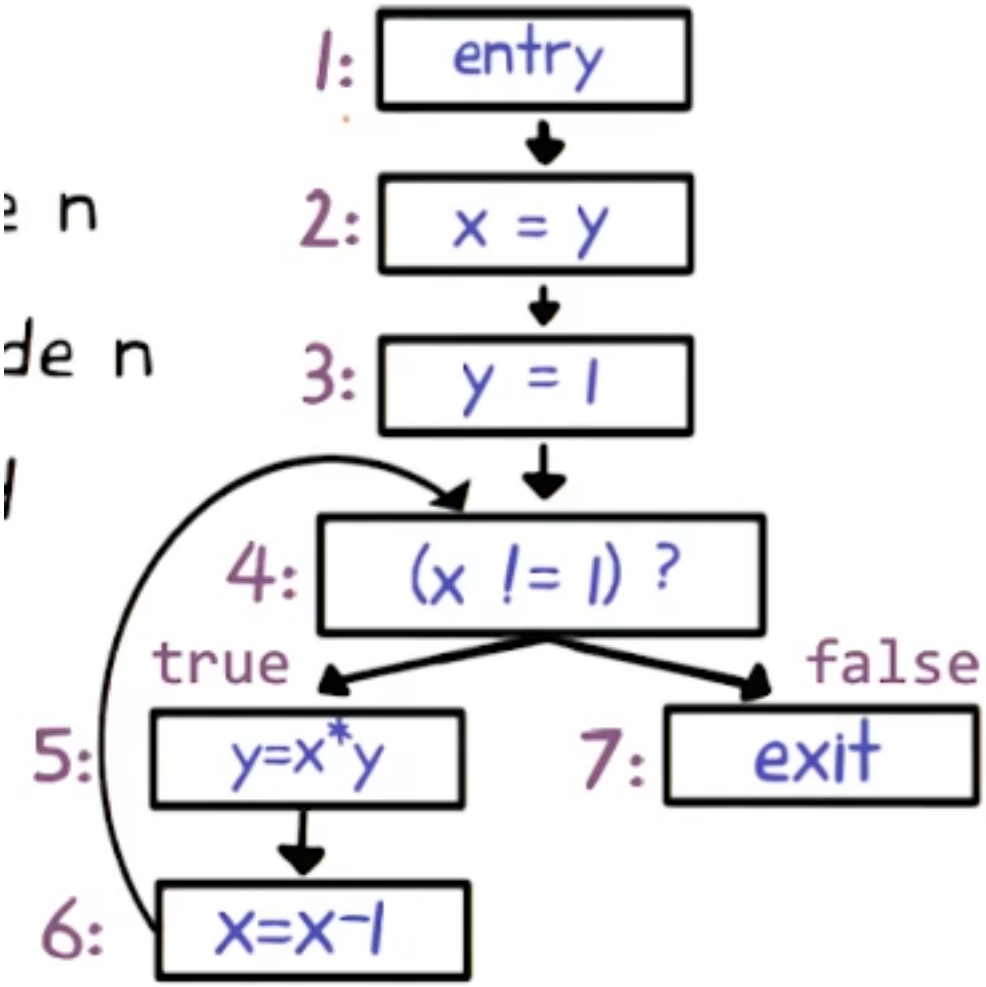
Result of Dataflow Analysis (Informally)

1. Result of a dataflow analysis is a set of facts at each program point
   - Reaching definitions analysis computes the set of defintions that may reach each program point
   - For reaching definitions analysis, fact is a pair of the form:
     - <defined variable name, defining node label>

Result of Dataflow Analysis (Formally)

1. Give distinct label n to each node

2. IN(n) = set of facts at entry of node n
3. OUT(n) = set of facts at exit of node n
4. Dataflow analysis computes IN(n) and OUT(n) for each node
5. Repeat two operations until IN(n) and OUT(n) stop changin
    - Called “saturated” or “fixed point”



Result of Dataflow Analysis

RDA Operation 1

1. How to compute the set of facts at the entry of a particular node of a control flow graph
    - Union of the sets of facts at the exit of that node’s predecessors
    - IN[n] = U OUT[n’]
        - n’ = predecessors(n)

RDA Operation 2

1. How to compute the set of facts at the exit of a particular node from the set of facts at the entry of that node
    - Depends on the statement of the node we’re examining

- OUT[n] = (IN[n] - KILL[n]) U GEN[n]
  - GEN and KILL sets require knowledge of the statement at n
- x = a;
  - GEN[n] = {<x,n>}
  - KILL[n] = {<x,m> : m != n}

RDA Chaotic Iteration Algorithm

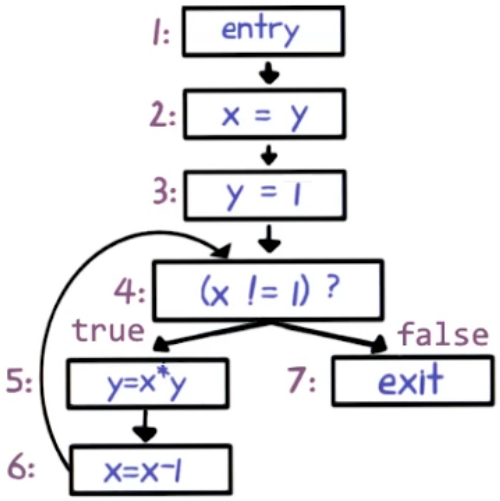
1. Algorithm:

```
for(each node n):
    IN[n] = OUT[n] = 0
OUT[entry] = {<v,?> : v is a program variable}
Repeat:
    for(each node n):
        IN[n] = U OUT[n'] // n' = predecessors(n)
        OUT[n] = (IN[n] - KILL[n]) U GEN[n]
    until IN[n] and OUT[n] stop changing for all n
```

2. Iterative and chaotic
   - Chaotic: Reaches every node in each iteration and updates the in and out sets
   - Order in which nodes are visited does not matter

Reaching Definitions Analysis Example

n	IN[n]	OUT[n]
1	-	{ <x,?>, <y,?> }
2	{ <x,?>, <y,?> }	{ <x,2>, <y,?> }
3	{ <x,2>, <y,?> }	{ <x,2>, <y,3> }
4		
5		
6		
7		-



Reaching Definitions Analysis

Reaching Definitions Analysis 3

n	IN[N]	OUT[n]
1	-	{<x,?>,<y,?>}
2	{<x,?>,<y,?>}	{<x,2>,<y,?>}
3	{<x,2>,<y,?>}	{<x,2>,<y,3>}

n	IN[N]	OUT[n]
4	{<x,2>,<y,3>,<y,5>,<x,6>}	{<x,2>,<y,3>,<y,5>,<x,6>}
5	{<x,2>,<y,5>,<x,6>}	{<x,2>,<y,5>,<x,6>}
6	{<x,2>,<y,5>,<x,6>}	{<y,5>,<x,6>}
7	{<x,2>,<y,3>,<y,5>,<x,6>}	-

Does It Always Terminate?

1. Chaotic Iteration algorithm always terminates
   - The two operations of reaching definitions analysis are monotonic
     - IN and OUT sets never shrink, only grow
   - Largest they can be set of all definitions in program
     - IN and OUT cannot grow forever
   - IN and OUT will stop changing after some iteration

Very Busy Expressions Analysis

1. Goal: Determine very busy expressions at the exit from each program point
   - 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

VBEA Operation 1

1. Operation #1
   - OUT[n] = intersect(IN[n1], IN[n2], . . . , IN[Nn])
     - For each succesor

VBEA Operation 2

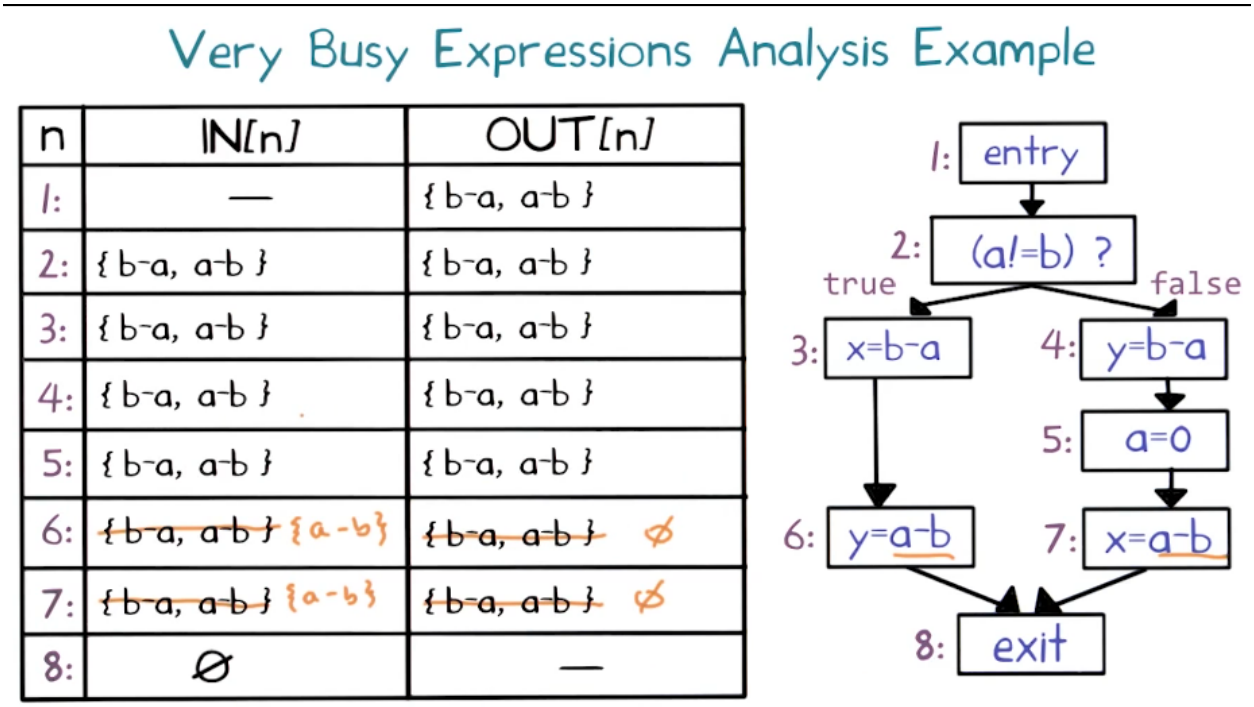
1. Operation #2
   - IN[n] = (OUT[n] - KILL[n]) U GEN[n]
2. If the statement is a condition:
   - GEN[n] = 0
   - KILL[n] = 0
3. If the statement is an assignment (x = a):
   - GEN[n] = {a}
   - KILL[n] = {expr e : e contains x}

VBEA Chaotic Iteration Algorithm

1. Algorithm:

```
for(each node n):
    IN[n] = OUT[n] = set of all expressions in program
IN[exit] = 0
Repeat:
    for(each node n):
        OUT[n] = intersect(IN[n']) // n' = successors(n)
        IN[n] = (OUT[n] - KILL[n]) U GEN[n]
    until IN[n] and OUT[n] stop changing for all n
```

Very Busy Expressions Analysis Example



Very Busy Expressions Analysis

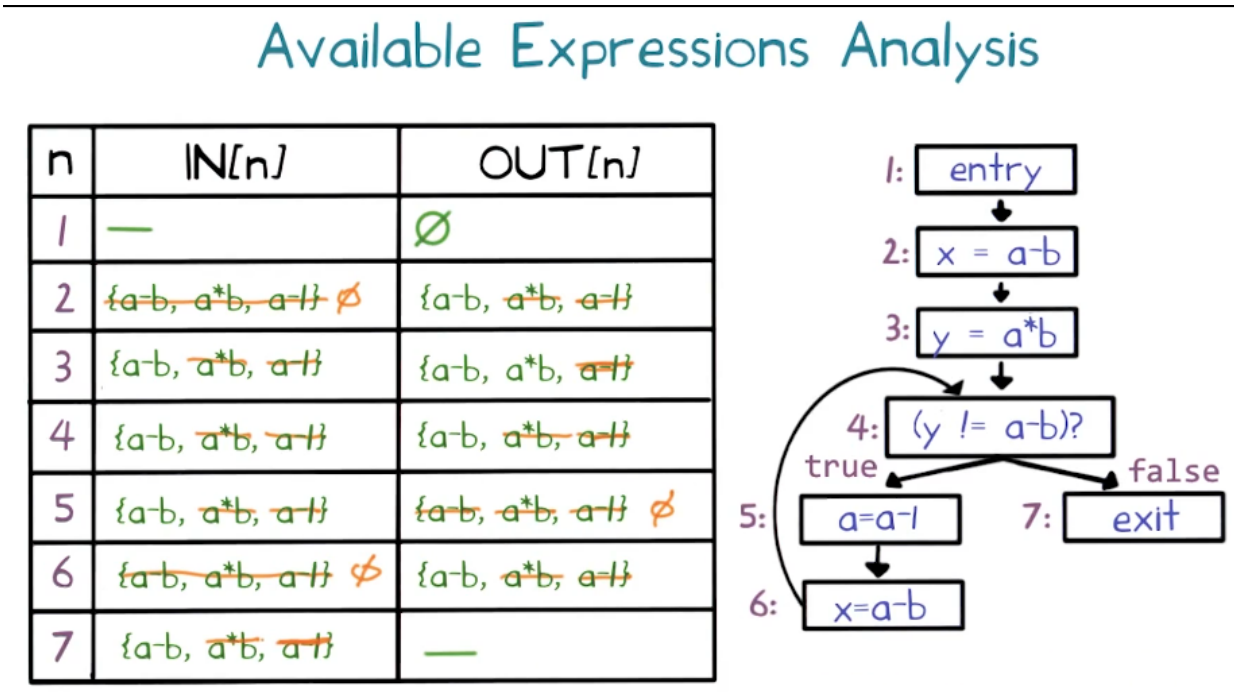
Very Busy Expressions Analysis 2

n	IN[N]	OUT[n]
1	-	{b-a}
2	{b-a}	{b-a}
3	{a-b,b-a}	{a-b}
4	{b-a}	0
5	0	{a-b}
6	{a-b}	0
7	{a-b}	0
8	0	-

Available Expressions Analysis

1. Goal: Determine, for each program point, which expressions must already have been computed, and not later modified, on all paths to the program point

Available Expressions Analysis 2

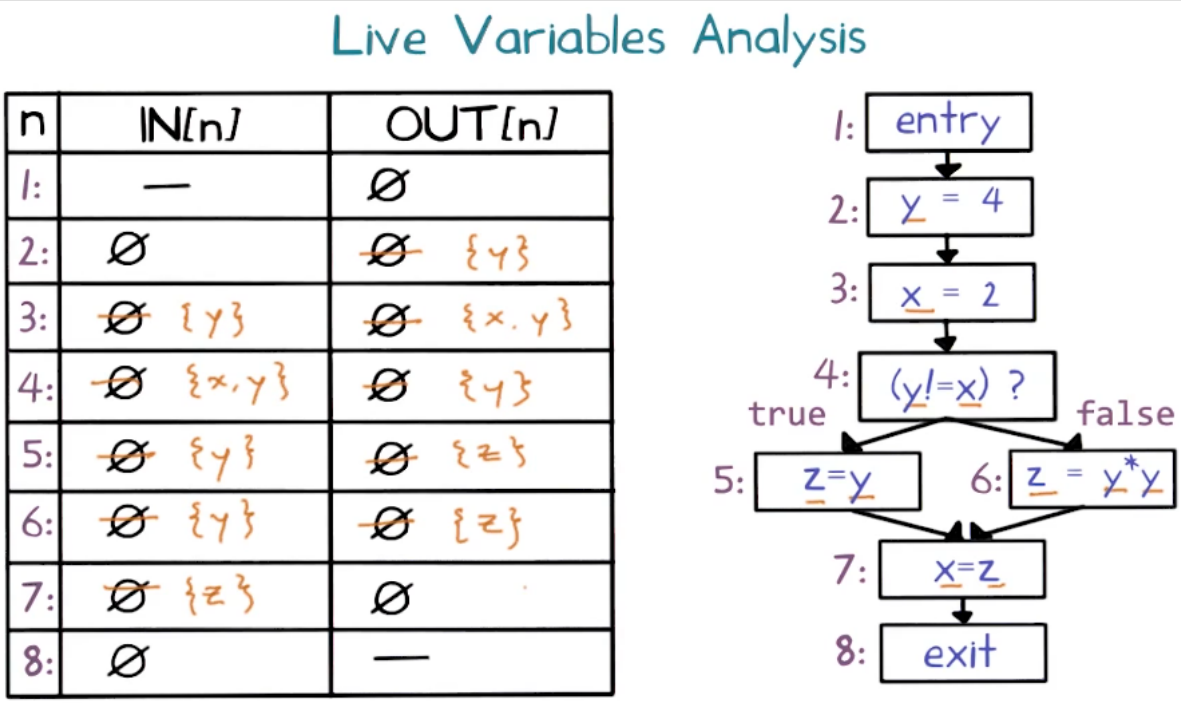


Available Expressions Analysis

Live Variables Analysis

1. Goal: 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
   - If there are three variables but only two are ever live, we only need to use two registers when generating the assembly code

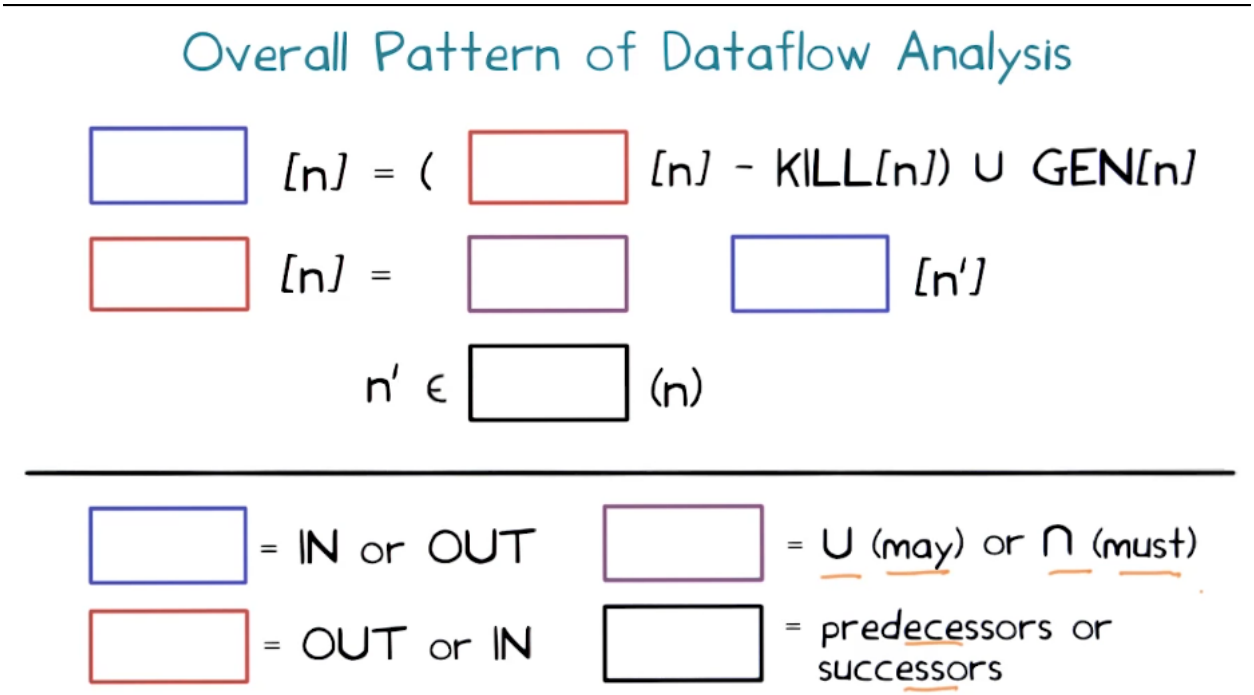
Live Variables Analysis 2



Live Variables Analysis

Overall Pattern of Dataflow Analysis

1. Each of the four dataflow analyses corresponds to a different instantiation of these boxes
   - Does the analysis propagate information forward or backward in the control flow graph?
   - Does the analysis compute may or must information?



Overall Pattern of Dataflow Analysis

Reaching Definitions Analysis 4

1. RDA
   - OUT[n] = (IN[n] - KILL[n]) union GEN[n]
   - IN[n] = union(OUT[n'])
   - n' = predecessors(n)

Very Busy Expression Analysis

1. VBEA
   - IN[n] = (OUT[n] - KILL[n]) union GEN[n]
   - OUT[n] = intersect(IN[n'])
   - n' = successors(n)

Available Expressions Analysis 3

1. AEA
   - OUT[n] = (IN[n] - KILL[n]) union GEN[n]
   - IN[n] = intersect(OUT[n'])
   - n' = predecessors(n)

Live Variables Analysis 3

1. LVA
   - IN[n] = (OUT[n] - KILL[n]) union GEN[n]
   - OUT[n] = union(IN[n'])
   - n' = successors(n)

Classifying Dataflow Analyses

	May	Must
Forward	RDA	AEA
Backward	LVA	VBEA

Conclusion

1. What is dataflow analysis?
   - Static analysis that allows us to reason about the flow of data in program runs
2. Reasoning about flow of data using control-flow graphs
3. Specifying dataflow analyses using local rules
4. Chaotic iteration algorithm to compute global properties
5. Four classical dataflow analyses
   - Reaching Definitions Analysis
   - Available Expressions Analysis
   - Very Busy Expressions Analysis
   - Live Variables Analysis
6. Classification: forward vs backward, may vs must