# Dataflow Analysis

CS 6340

{HEADSHOT}

The field of software analysis is highly diverse: there are many approaches with different strengths and limitations in aspects such as soundness, completeness, applicability, and scalability.

In this lesson, we will introduce dataflow analysis, one of the dominant approaches to software analysis. We will see specific examples of useful dataflow analyses from the literature, and we will learn about a general technique to design a dataflow analysis.

After this lesson, you should be able to design your own dataflow analyses for a basic yet powerful programming language that captures the essence of realistic programming languages like C and Java.

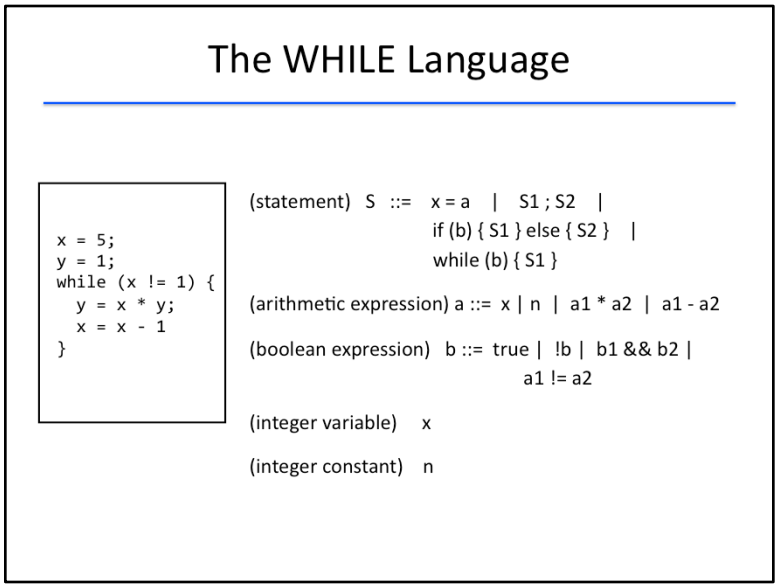
## What Is Dataflow Analysis?

- Static analysis reasoning about flow of data in program
- Different kinds of data: constants, variables, expressions
- Used by bug-finding tools and compilers

Dataflow analysis is a kind of static analysis for reasoning about the flow of data in runs of a program.

The data can be of different kinds: constants (such as the number 7 or the string literal “hello”), variables (such as ‘foo’), expressions (such as 7 * foo), and so on.

This information in turn is used by bug-finding tools to find programming errors and by compilers to generate efficient code for the given program.




Throughout this lesson, we will work with a simple programming language, called the WHILE language.

Here is an example program written in this language to compute the factorial of 5. The program has two integer variables x and y. It initializes these two variables and then updates them in a loop. Variable x contains the factorial of 5 at the end of the program.

Here is a formal grammar that precisely describes the syntax of programs written in the WHILE language. You can learn more about this notation by clicking on the link in the instructor notes. [https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form]

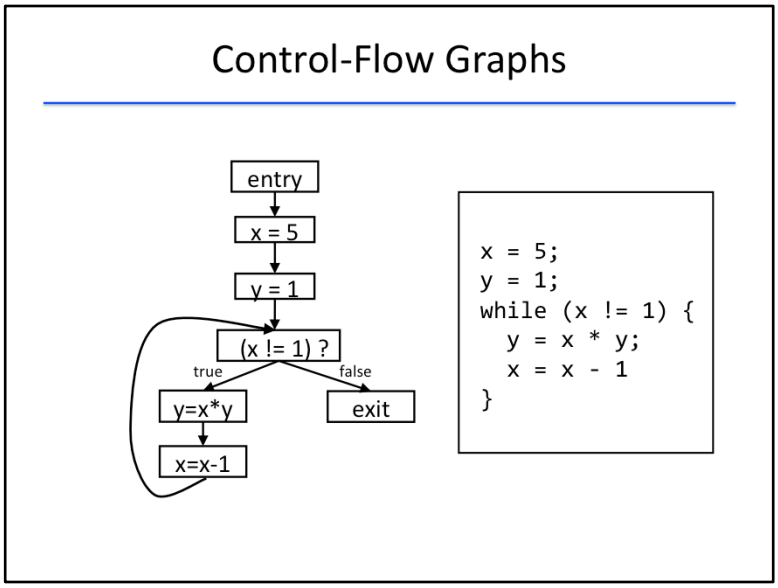
A program in this language is a statement S, which can be an assignment statement, a sequential composition of two statements, an if-then-else statement, or a while statement. Notice that this definition of a statement is recursive, so it can be used to describe arbitrarily large programs: programs with nested if-then-else statements, programs with nested loops, and so on.

For simplicity, we have only integer variables in this language. Furthermore, assignments to such variables can only be arithmetic expressions of a limited form. In particular, an arithmetic expression can be an integer variable which we denote using x, or an integer constant which we denote using n, or a multiplication of two expressions, or a subtraction of one expression from another expression.

The definition of arithmetic expressions is also recursive, allowing us to write programs with arbitrarily large expressions. It's easy to extend the syntax of these expressions to include other operators such as addition and division, but we will leave those out for now to keep it simple.

Finally, to express conditions in if-then-else statements and while statements, we have boolean expressions. A boolean expression may be the constant true, the negation of another boolean expression, the conjunction of two boolean expressions b1 and b2, or a comparison between two arithmetic expressions a1 and a2. Again, to keep things simple, we allow limited kinds of boolean expressions, although it is easy to extend the syntax of this language to include operators besides the ones shown here.

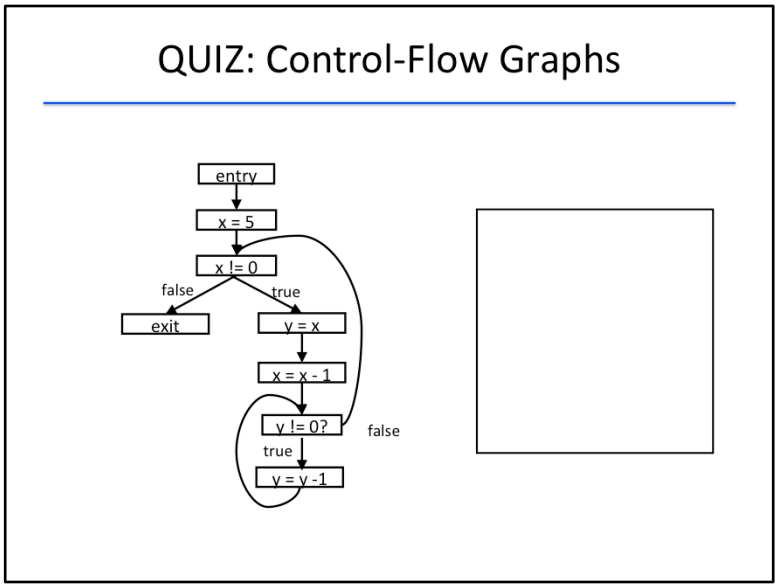
Notice that the WHILE language does not have fancy constructs such as functions, pointers, or threads that are provided in commonly used programming languages like C and Java. This is because the




Dataflow analysis typically operates on a suitable intermediate representation of the program. One such representation shown here, which we also saw earlier in the course, is a control-flow graph.

A control-flow graph is a graph that summarizes the flow of control in all possible runs of the program. Each node in the graph corresponds to a unique primitive statement in the program, such as an assignment or a condition test, and each edge outgoing from a node denotes a possible immediate successor of that statement in some run of the program.

Take a moment to convince yourself that this graph is indeed a control-flow graph of this program.

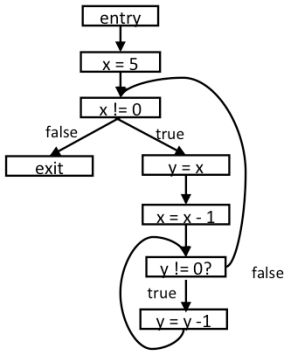



{QUIZ SLIDE}

To check your understanding of control-flow graphs, let's do an exercise converting a control-flow graph into the program it came from.

Here is a control-flow graph. In the adjoining box, write the program corresponding to this control-flow graph in the syntax of the WHILE language. Click the "Submit" button to check your answer.

## QUIZ: Control-Flow Graphs




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

{SOLUTION SLIDE}

The program corresponding to this control flow graph has two variables, x and y. It initializes the variable x to 5 and then executes a nested while statement.

The outer while-loop decrements x in each iteration and terminates when x becomes 0. Also, at the start of each iteration, the variable y is initialized to the current value of x.

The inner while-loop decrements y in each iteration and terminates when y becomes 0.

## Soundness, Completeness, Termination

- Impossible for analysis to achieve all three together
- Dataflow analysis sacrifices completeness
- Sound: Will report all facts that could occur in actual runs
- Incomplete: May report additional facts that can't occur in actual runs

Recall from before that it is impossible to design a software analysis that is sound, complete, and guaranteed to terminate. This impossibility holds for dataflow analyses, as they are a kind of software analysis.

Dataflow analyses choose to sacrifice completeness to guarantee termination and soundness.

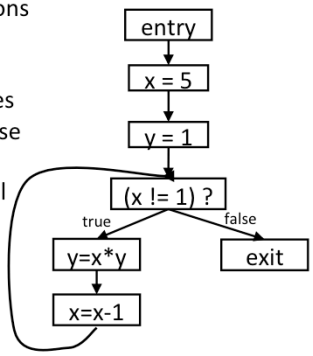
Since dataflow analysis is sound, it will report all dataflow facts that could occur in actual runs.

However, because dataflow analysis is incomplete, it may report dataflow facts that can never occur in actual runs.

Let's see next how a dataflow analysis achieves soundness by sacrificing completeness.

## Abstracting Control-Flow Conditions

- 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).

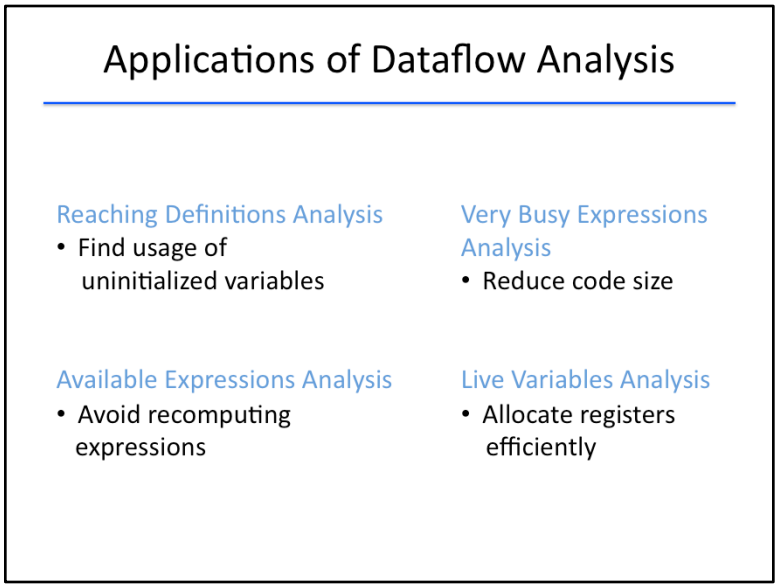



The primary source of incompleteness in dataflow analyses arises from abstracting away control-flow conditions with non-deterministic choice, which we will denote throughout this course using the star symbol.

For this example program, dataflow analysis replaces the condition (x != 1) with non-deterministic choice. Strike out boolean expression (x != 1) and replace it with a *.

Non-deterministic choice simply means that the analysis will assume that the condition can evaluate to true or false, even if, for example, in actual runs the condition always evaluates to true.

By doing this, not only does a dataflow analysis ensure that it will consider all paths that are possible in actual runs of the program, and thereby guarantees soundness, but it also considers paths that are never possible in actual runs, which leads to incompleteness.




We will learn how dataflow analysis works on a control-flow graph through a series of four classical dataflow analyses in the literature. These analyses are: Reaching Definitions Analysis, Very Busy Expressions Analysis, Available Expressions Analysis, and Live Variables Analysis.

Before we dive into the details of these four analyses, let's take a look at four practical applications that motivate them.

Reaching Definitions Analysis produces information that can be used by a software quality tool for discovering usage of potentially uninitialized variables in a program.

Very Busy Expressions Analysis computes information that can help reduce code size. This application can be critical to certain embedded devices that have code size constraints, such as pacemakers.

Available Expressions Analysis produces information that can be used by a compiler to avoid recomputing the same program expression multiple times in an execution, thereby producing more efficient code.

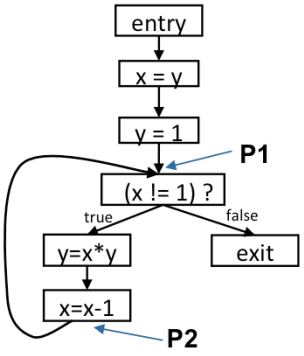
Finally, Live Variables Analysis computes information that can be used by a compiler to efficiently allocate registers to program variables. Register allocation is the component of a compiler that most impacts the performance of the generated code.

Next, we will dive into how each of these four analyses work, starting with Reaching Definitions Analysis.

## Reaching Definitions Analysis

Goal: Determine, for each program point, which assignments have been made and not overwritten, when execution reaches that point along some path

- "Assignment" == "Definition"




We will use Reaching Definitions Analysis to introduce the key concepts of dataflow analysis.

Each dataflow analysis has a goal that specifies the kind of data flow information that the analysis computes. The goal of reaching definitions analysis is to determine which assignments might reach each program point. More accurately, this analysis determines, for each program point, which assignments potentially have been made and not overwritten, when the program's execution reaches that point along some path.

For the purpose of this analysis, we will use the terms "assignment" and "definition" interchangeably, since an assignment corresponds to a definition in the WHILE language.

Let us look at the following example program. There are four definitions in this program: x = y, y = 1, y = x * y, and x = x - 1.

Consider two program points: P1, at the entry of this condition, and P2, at the exit of this assignment.

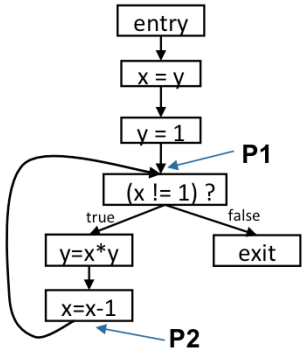
Let's consider the definition x = y. This definition reaches point P1 as there is no overwriting assignment to x along this path.

But this definition does not reach point P2, as x is overwritten by assignment x = x - 1 every time execution reaches P2.

Please take a moment to understand the goal of reaching definitions analysis. We will next do a quiz to practice a few more reaching definitions in this control-flow graph.

## QUIZ: Reaching Definitions Analysis

1. The assignment y = 1 reaches P1 ☐
2. The assignment y = 1 reaches P2 ☐
3. The assignment y = x * y reaches P1 ☐



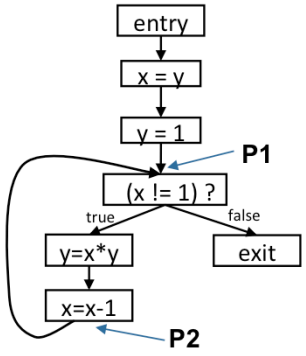

{QUIZ SLIDE}

To check your understanding of reaching definitions analysis, here's a short quiz.

Check the boxes corresponding to the statements that are true about reaching definitions analysis.

## QUIZ: Reaching Definitions Analysis

1. The assignment y = 1 reaches P1 ■
2. The assignment y = 1 reaches P2 □
3. The assignment y = x * y reaches P1 ■




{SOLUTION SLIDE}

The 1st statement is True. Indeed, the definition y = 1 reaches P1 along this path. (gesture)

The 2nd statement is False. This is because y is overwritten by the definition y = x * y every time execution reaches P2. (gesture)

The 3rd statement is True. Indeed, the definition y = x * y reaches P1 along this path. (gesture)

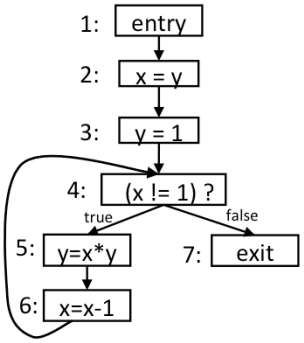
Notice that different definitions can reach a given program point, as in the case of definitions y = 1 and y = x * y reaching P1. And conversely, a given definition can reach multiple program points, as in the case of y = x * y reaching both P1 and P2.

Next, let's take a look at how a dataflow analysis represents the results it computes.

## Result of Dataflow Analysis (Informally)

- Set of facts at each program point
- For reaching definitions analysis, fact is a pair of the form:

  <defined variable name, defining node label>
- Examples: <x,2> , <y,5>




Informally speaking, the result of a dataflow analysis is a set of facts at each program point.

For example, reaching definitions analysis computes the set of definitions that may reach each program point.

To identify each program point uniquely, let's assign a distinct label to each node in the control-flow graph. Here, I've labelled the nodes in this control-flow graph from 1 through 7.

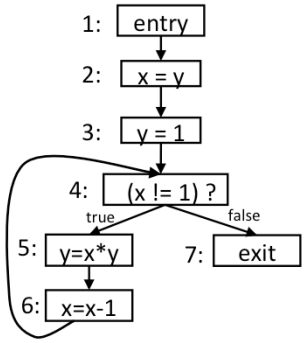
Then, we can denote each reaching definition as a pair comprising the name of the defined variable, along with the label of the node that defines it.

For example, the definition of variable x at the node labeled 2 is denoted as follows: <x, 2>. And the definition of variable y at the node labeled 5 is denoted as follows: <y, 5>.

Now, let's make this notions of a dataflow analysis result more precise.

## Result of Dataflow Analysis (Formally)

- Give distinct label n to each node
- IN[n] = set of facts at entry of node n
- OUT[n] = set of facts at exit of node n
- Dataflow analysis computes IN[n] and OUT[n] for each node
- Repeat two operations until IN[n] and OUT[n] stop changing
  - Called "saturated" or "fixed point"




Then, for each node with label n, we use IN(n) to denote the set of facts at the entry of the node, and OUT(n) to denote the set of facts at the exit of the node.

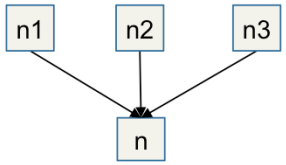
A dataflow analysis computes the IN and OUT sets of facts for each node in the control-flow graph.

It does so by repeatedly applying two operations until the sets of IN and OUT facts for each node in the graph stop changing. At that point, we say that the result of the dataflow analysis is saturated or that it has reached a fixed point.

We will next introduce these two operations for reaching definitions analysis. Subsequently, we will see slight variations of these operations for the other three dataflow analyses.

## Reaching Definitions Analysis: Operation #1

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

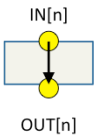



$IN[n] = OUT[n1] \cup OUT[n2] \cup OUT[n3]$

Here's the first of the two operations of reaching definitions analysis. This operation states how to compute the set of facts at the entry of a particular node in the control-flow graph. We do this by taking the union of the sets of facts at the exit of that node's immediate predecessors: that is, the union of OUT[n'] for each predecessor node n' of n.

## Reaching Definitions Analysis: Operation #2

OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]




n: b ?  GEN[n] = ∅  KILL[n] = ∅

n: x = a  GEN[n] = { <x, n> }
KILL[n] = { <x, m> : m != n }

The second operation of reaching definitions analysis tells us 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. Unlike the previous operation that we just saw, this operation depends on the statement that occurs at the node we are looking at.

We'll first state this operation in its general form, and then we'll show specific instances of it corresponding to each kind of primitive statement in the WHILE language.

The general form of this operation states that the set of facts at the exit of node n is equal to the set of facts at the entry of node n, minus any definitions that are overwritten by node n, unioned with any new definitions that are generated by node n. We call these sets of definitions as the KILL set and the GEN set, respectively.

Determining the GEN and KILL sets requires knowledge of the statement that occurs at node n. For control-flow graphs of programs in the WHILE language, this statement can be either a condition or an assignment. Let's consider each of these two cases in turn.

If the statement at node n is a condition, then both the GEN and KILL sets are empty, since there are no definitions overwritten or generated by a conditional statement. (So, in this case, the set OUT[n] will equal the set IN[n].)

If the statement at node n is an assignment of the form x = a, then the GEN and KILL sets are more interesting. The GEN set contains the definition of variable x at node n itself, reflecting the fact that this definition is generated by node n. The KILL set, on the other hand, contains every definition of variable x except the one at node n, reflecting the fact that all those definitions will be overwritten by the one at node n. We denote this set compactly using set comprehension notation. You can review this notation by following the link in the instructor notes.

[https://en.wikipedia.org/wiki/Set-builder_notation]

## Overall Algorithm: Chaotic Iteration

for (each node n):
    IN[n] = OUT[n] = ∅
OUT[entry] = { <v, ?> : v is a program variable }
repeat:
    for (each node n):
        $IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$
        OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]
until IN[n] and OUT[n] stop changing for all n

Equipped with the two operations of reaching definitions analysis, let's step through the overall reaching definition analysis algorithm.

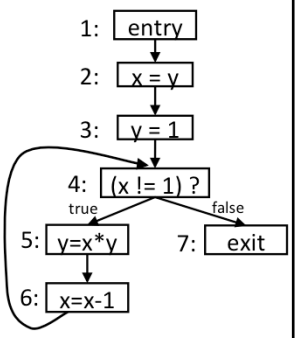
The algorithm starts by initializing the IN and OUT set of each node n in the control-flow graph to the empty set. The only exception is the OUT set of the entry node of the control-flow graph, which is initialized to contain a hypothetical definition for each variable v in the program. It captures the fact that each variable is undefined, or uninitialized, at the start of the program.

The algorithm then performs its main task from which it derives the name chaotic iteration algorithm. The name highlights two important properties of the algorithm.

First, it is iterative: it repeatedly updates the IN and OUT sets of each node in the control-flow graph until they stop changing. Second, it is chaotic in the sense that in each iteration, it visits all nodes in the control-flow graph and applies the two operations we just discussed to update the IN and OUT sets of each node. Crucially, the order in which the nodes are visited does not matter, lending the adjective chaotic in the name of the algorithm.

## Reaching Definitions Analysis Example

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




Now let's see how the chaotic iteration algorithm for reaching definitions analysis works on our example control-flow graph.

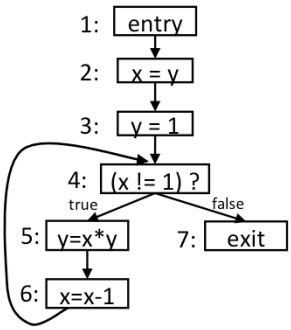
We will use this table to track the values of the IN and OUT sets of each of the 7 nodes in this control-flow graph.

The algorithm starts by initializing all these entries to the empty set, except for the OUT set of the entry node, which captures the fact that both variables x and y are undefined at this point. Also, we will ignore the IN set of the entry node and the OUT set of the exit node as these nodes do not contain any statement and are merely placeholders.

Next, the algorithm repeatedly picks each of the remaining 5 nodes and applies the two rules that update their IN and OUT sets.

## 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 | ∅ | -- |




For instance, it updates the IN set of node 2 to reflect the fact that both variables x and y remain undefined at this point.
Strike out ∅ in IN column of row 2 and replace it with { <x,?>, <y,?> }.

It also updates the OUT set of node 2 to reflect the fact that the definition of variable x generated at node 2 reaches the exit of node 2. Notice that node 2 also kills the incoming fact that x is undefined, but retains the incoming fact that y is undefined.
Strike out ∅ in OUT[2] and replace it with { <x,2>, <y,?> }.

Likewise, it updates the IN set of node 3 to reflect the fact that the definition of x at node 2 reaches it and the fact that y is still undefined.
Strike out ∅ in IN[3] and replace it with { <x,2>, <y,?> }.

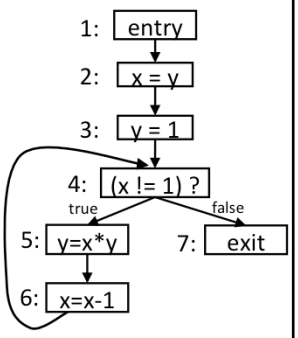
Continuing further, it updates the OUT set of node 3 to reflect the fact that not only is the incoming definition of variable x not overwritten by node 3, but furthermore, node 3 generates a new definition of variable y. Both these definitions reach the exit of node 3.
Strike out ∅ in OUT[3] and replace it with { <x,2>, <y,3> }.

Recall that the IN set at node 2 contains the hypothetical definition of variable y, indicating that y is uninitialized at this point. A bug-finding tool could use this information to deduce that the use of variable y at node 2 might be uninitialized, a potential programming error.

Let's update the remaining entries of this table in the following quiz.

## QUIZ: Reaching Definitions Analysis

| 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 | | -- |




{QUIZ SLIDE}

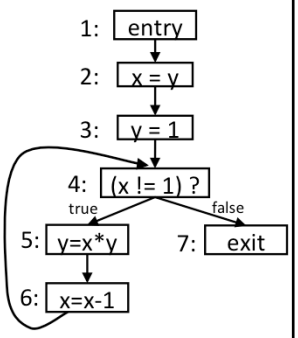
Fill in the blank entries in the table with the final values of the corresponding IN and OUT sets computed by the chaotic iteration algorithm for reaching definitions analysis.

Keep in mind the two operations that the algorithm applies to each node. The first operation states that the IN set of a node is the union of the OUT sets of its immediate predecessor nodes. The second operation states that the OUT set of a node contains all the definitions in the IN set of that node, minus definitions that are overwritten by that node, plus any new definitions that are generated by that node.

Remember to keep iterating through the whole table, applying these two operations to each node until there are no changes in a given pass through the table.

## QUIZ: Reaching Definitions Analysis

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




{SOLUTION SLIDE}

Let's review the solution.

Consider the condition node labeled 4. Its IN set consists of all facts from the OUT sets of its immediate predecessors, which are nodes 3 and 6. So the IN set contains the definition of x at node 2 and the definition of y at node 3 (write <x,2> and <y,3> in IN[4]). The condition node neither generates nor overwrites any definitions, so we copy these two facts to the OUT set (write <x,2> and <y,3> in OUT[4]).

Next, we copy the OUT set of node 4 to the IN set of node 5 (write <x,2> and <y,3> in IN[5]).

Now, to compute the OUT[5], we first copy <x,2> and <y,3> from the IN[5] (write <x,2> and <y,3> in OUT[5]). Since the node overwrites the definition of y, we delete <y,3> and add <y,5> (erase <y,3> and write <y,5> in OUT[5]).

Since node 5 is the only immediate predecessor of node 6, we copy OUT[5] to IN[6] (write <x,2> and <y,5> in IN[6]).

Then, at node 6, we first copy IN[6] to OUT[6] (write <x,2> and <y,5> in OUT[6]). Since node 6 redefines x, we delete <x,2> and replace it with <x,6> (erase <x,2> and write <x,6> in IN[6]).

Finally, we look at the immediate predecessor of node 7, which is node 4, to determine the IN set of node 7 (write <x,2> and <y,3> in IN[7]).

Now we loop back to our earlier if-statement. We've already visited this point, so we already have an IN set. Therefore we'll union the existing IN set with the OUT of the newly found predecessor, node 6. (write <y,5> and <x,6> in IN[4]) This gives us the new set: <x,2>,<x,6>,<y,3> ,<y,5>. This will in turn modify our OUT to be <x,2>,<x,6>,<y,3> ,<y,5> as well (write <y,5> and <x,6> in OUT[4]).

This change propagates down through the left branch, updating the IN and OUT sets of statement 5

## Does It Always Terminate?

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 is set of all definitions in program, which is finite
  => IN and OUT cannot grow forever

=> IN and OUT will stop changing after some iteration

At this point, you might be wondering whether the chaotic iteration algorithm is guaranteed to terminate for every program in the WHILE language, despite its seemingly chaotic nature.

The answer, perhaps somewhat surprisingly, is yes.

The reason for this is two-fold. First, the two operations that this algorithm applies in each iteration are monotonic, that is, they never cause the IN and OUT sets to shrink. We indeed observed this behavior in the table of the reaching definitions example, where these sets always grew bigger.

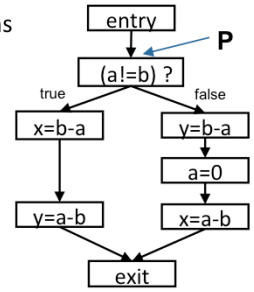
Secondly, the largest that any such set can get is the set of all definitions in the program, which is finite. This implies that the IN and OUT sets cannot grow forever.

Together, these two reasons ensure that all IN and OUT sets will stop changing in some iteration of the chaotic iteration algorithm, which is the condition under which the algorithm terminates.

## Very Busy Expressions Analysis

Goal: Determine very busy expressions at the exit from the 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




Now let's move on to the second of the four classical dataflow analyses. This one is called Very Busy Expressions analysis. We will present this analysis by following a recipe similar to the one we used to present Reaching Definitions analysis.

The goal of Very Busy Expressions analysis is to compute expressions that are very busy at the exit from each program point.

An expression is very busy if, no matter what path is taken, the expression is always used before any of the variables occurring in it are redefined.

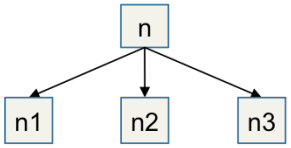
Let us look at the following example program. Let's consider these two expressions in this program: a - b and b - a.

Consider the program point P at the entry of the condition statement. (write P and arrow)

The expression b - a is very busy at this point since it is used along both the paths from that point, here and here (gesture), before any of the variables occurring in the expression is redefined. Now let's consider expression a - b. This expression is used on both the paths as well, but variable a in the expression is redefined along one of those paths here (gesture), before the expression is used here (gesture). So expression a - b is not very busy at program point P.

## Very Busy Expressions Analysis: Operation #1

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

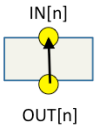



$OUT[n] = IN[n1] \cap IN[n2] \cap IN[n3]$

Here's the first of the two operations of very-busy-expressions analysis. This rule states how to compute the set of facts at the exit of a particular node in the control-flow graph. We do this by taking the intersection of the sets of expressions at the entry of that node's immediate successors: that is, the intersection of IN[n'] for each successor node n' of n.

## Very Busy Expressions Analysis: Operation #2

IN[n] = (OUT[n] - KILL[n]) ∪ GEN[n]




n: b ?  GEN[n] = ∅  KILL[n] = ∅

n: x = a  GEN[n] = { a }
KILL[n] = { expr e : e contains x }

The second operation of very-busy-expressions analysis tells us how to compute the set of facts at the entry of a particular node from the set of facts at the exit of that node. The operation of this rule depends on the statement that occurs at the node we are looking at.

Like we did for reaching-definitions analysis, we'll first state this operation in its general form, and then we'll show specific instances of the operation corresponding to each kind of primitive statement in the WHILE language.

The general form of this operation states that the set of expressions at the entry of node n is equal to the set of expressions at the exit of node n, minus any expressions using variables that are modified by node n, unioned with any new expressions that are used by node n. Like before, we call these sets of expressions the KILL set and the GEN set.

If the statement at node n is a conditional statement, then the KILL set and GEN set are empty, since nothing is modified or generated by the node.

If the statement at node n is an assignment statement of the form x = a, then the GEN set will contain the expression assigned to variable x at node n itself, reflecting the fact that this expression is used by node n. The KILL set, on the other hand, contains every expression using x, reflecting the fact that all those expressions will be modified by node n.

## Overall Algorithm: Chaotic Iteration

for (each node n)
  IN[n] = OUT[n] = set of all exprs in program
IN[exit] = ∅
repeat:
  for (each node n)
    $OUT[n] = \bigcap_{n' \in successors(n)} IN[n']$
    IN[n] = (OUT[n] - KILL[n]) ∪ GEN[n]
until IN[n] and OUT[n] stop changing for all n

The overall algorithm for very busy expressions analysis is nearly identical to that for reaching definitions analysis but has three notable differences.

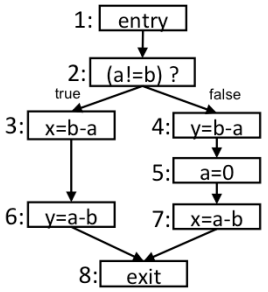
First, the two operations that are applied in each iteration of the algorithm are slightly different. Notice that the roles of IN and OUT sets in these two operations are switched, reflecting the fact that very-busy-expressions analysis propagates information backwards in a control-flow graph, in contrast to reaching-definitions analysis, which propagates information forward.

Moreover, we take the intersection of sets as opposed to their union, which causes the IN and OUT sets of very busy expressions to shrink as the algorithm progresses. In contrast, recall that in reaching-definitions analysis, the IN and OUT sets of reaching definitions grow because we use the union operation.

This also makes it clear why we initialize all IN and OUT sets in very busy expressions analysis to the set of all expressions in the program, since these sets will shrink as the algorithm progresses. In contrast, recall that the IN and OUT sets in reaching definitions analysis are initialized to the empty set, and those sets grow as that algorithm progresses. (However, by convention, we still set the IN set of the exit node of the control-flow graph to be empty, since no expressions are very busy at the end of the program.)

## Very Busy Expressions Analysis Example

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




Now let's see how the chaotic iteration algorithm for very busy expressions analysis works on our example control-flow graph.

We will use this table to track the values of the IN and OUT sets of each of the 8 nodes in this control-flow graph.

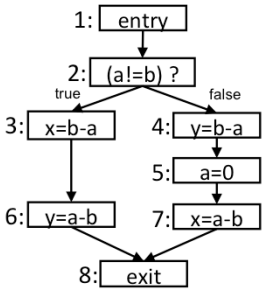
The algorithm starts by initializing all these entries to the set of all expressions, except the OUT set of the exit node, which it initializes to the empty set.

As in the case of reaching definitions analysis, we will ignore the IN set of the entry node and the OUT set of the exit node as these nodes do not contain any statement and are merely placeholders.

Next, the algorithm repeatedly picks each of the remaining nodes and applies the two rules that update their IN and OUT sets.

## Very Busy Expressions Analysis Example

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




For instance, it updates the OUT set of node 7 to the empty set, reflecting the fact that no expressions are very busy at this point.
Erase { b-a, a-b } in OUT[7] and replace it with ∅.

Likewise, it updates the IN set of node 7 to reflect the fact that expression a - b is very busy at this point, as it is indeed used immediately thereafter.
Erase { b-a, a-b } in IN[7] and replace it with { a-b }.

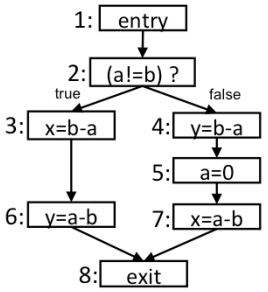
Continuing along this branch, it similarly updates the OUT set of node 6 to the empty set, as no expressions are very busy at this point.
Erase { b-a, a-b } in OUT[6] and replace it with ∅.

Similarly, it continues further backwards and updates the IN set of node 6 to reflect the fact that expression a - b is very busy at this point.
Erase { b-a, a-b } in IN[6] and replace it with { a-b }.

Let's update the remaining entries of this table in the following quiz.

## QUIZ: Very Busy Expressions Analysis

| n | IN[n] | OUT[n] |
|---|---|---|
| 1 | -- | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | ∅ | { a-b } |
| 6 | { a-b } | ∅ |
| 7 | { a-b } | ∅ |
| 8 | ∅ | -- |




{QUIZ SLIDE}

Fill in the blank entries in the table with the final values of the corresponding IN and OUT sets computed by the chaotic iteration algorithm for very busy expressions analysis.

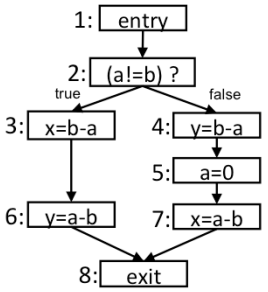
Keep in mind the two operations that the algorithm applies to each node.

The first operation states that the OUT set of a node is the intersection of the IN sets of its immediate successor nodes. The second operation states that the IN set of a node contains all the expressions in the OUT set of that node, minus expressions that are overwritten by that node, plus any new expressions that are generated by that node.

Remember to keep iterating through the whole table, applying these two operations to each node until there are no changes in a given pass through the table.

## QUIZ: Very Busy Expressions Analysis

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




{SOLUTION SLIDE}

Let's trace the chaotic iteration algorithm through until it completes its iteration. Note that since we don't have any loops in this program, we will only need to make one pass upwards before the IN and OUT sets stabilize.

Let's look at node 4. Its only successor is node 5, so we copy IN[5] to OUT[4] (replace OUT[4] by the empty set). Since OUT[4] is empty, we have no expressions to kill. All we need to do is add b-a to the set since the expression is being used at this node (replace IN[4] by the set {b-a}).

For node 3, our OUT set is the same as the IN set of its sole successor, node 6 (replace OUT[3] by {a-b}). Since x is not present in any expressions in the OUT set of node 3, we don't kill any expressions. We'll just add the expression being defined to the IN set (replace IN[3] by the set {a-b, b-a}).

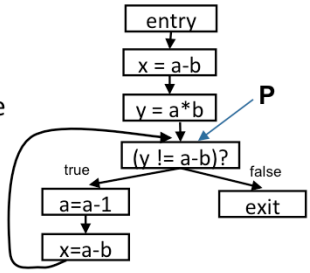
Now, for program point 2, we have two different successors we must use to determine our OUT set. We need to make sure that any expression we set as very busy is such for all execution paths. Thus, we take the intersection of program point 3 and 4's IN sets and get just b-a (replace OUT[2] by the set {b-a}). Since this node evaluates a boolean expression, we don't kill any expressions. We just add the boolean expression being evaluated to the OUT set to get our IN set (replace IN[2] by {b-a, a!=b}).

Finally we reach the entry, whose OUT set is the same as the IN set of its only successor (replace OUT[1] by the set {b-a, a!=b}).

Since there are no loops in the program, repeating the chaotic iteration algorithm will not cause any of the IN or OUT sets to change, so this completes our analysis.

## Available Expressions Analysis

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




Next, let's move on to the third of our four classical dataflow analyses, called Available Expressions Analysis. The goal of this analysis is to determine, for each program point, which expressions have already been computed, and not later modified, on all paths to the program point.

Let us look at the following example program. There are there expressions of interest in this program: a - b, a * b, and a - 1.

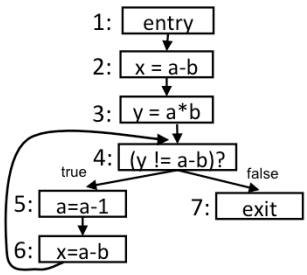
Consider the program point P at the entry of this condition (draw P and the arrow).

At this point, the expression a - b is said to be available. To see why, let's show that this expression is already computed and not later modified along every path that reaches this point. There are two paths. Along this path (gesture), the expression a - b is computed here (gesture), and neither a nor b is modified later. Likewise, along this other path (gesture), the expression a - b is computed here (gesture), and neither a nor b is modified later.

On the other hand, the expression a * b is not available at program point P. This is because, although this expression is available along this path (gesture), it is not available along this other path (gesture). In particular, the variable a occurring in this expression is modified here along this path (gesture), and the expression is not computed after this modification in order to become available later at this point (gesture).

## Available Expressions Analysis

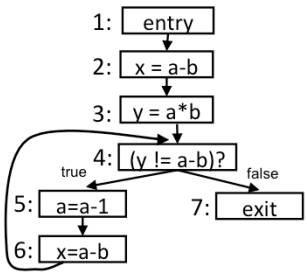
| n | IN[n] | OUT[n] |
|---|---|---|
| 1 | -- | ∅ |
| 2 | {a-b, a*b, a-1} | {a-b, a*b, a-1} |
| 3 | {a-b, a*b, a-1} | {a-b, a*b, a-1} |
| 4 | {a-b, a*b, a-1} | {a-b, a*b, a-1} |
| 5 | {a-b, a*b, a-1} | {a-b, a*b, a-1} |
| 6 | {a-b, a*b, a-1} | {a-b, a*b, a-1} |
| 7 | {a-b, a*b, a-1} | -- |




Let's walk through an example. Our IN set in this case will be any expressions which have been calculated earlier in the code without having the variables in their calculations overwritten. Our OUT set will be our IN set minus any expressions which have a variable that is overwritten by that statement, plus any expressions that are generated by that statement.

## Available Expressions Analysis

| n | IN[n] | OUT[n] |
|---|---|---|
| 1 | -- | ∅ |
| 2 | ∅ | {a-b} |
| 3 | {a-b} | {a-b, a*b} |
| 4 | {a-b, a*b} | {a-b, a*b} |
| 5 | {a-b, a*b} | ∅ |
| 6 | ∅ | {a-b} |
| 7 | {a-b, a*b} | -- |




Let's walk through the first three program points. For the entry, we have the empty set as our OUT set (gesture to OUT[1]).

This OUT set will be copied to the IN set of node 2 (write ∅ in IN[2]). To compute the OUT set of node 2, we first copy the IN set of node 2, which is empty, and add expression a-b which is generated at node 2 (write {a-b} in OUT[2]).

Node 3 takes in the expression a-b from node 2 (write {a-b} in IN[3]). In addition, node 3 generates another expression, a*b, which we add to its OUT set (write {a-b, a*b} in OUT[3]).

Node 4 is a bit tricky. Our IN set at this point is a-b and a*b (write {a-b, a*b} in IN[4]). Note that this is not the only path to this program point, but we have not computed the OUT set of node 4's other predecessor yet, so we'll need to make at least another pass through the table before the algorithm stabilizes. Nothing is killed or generated at this node, so we copy the IN set to the OUT set (write {a-b, a*b} in OUT[4]).

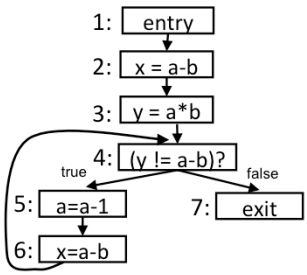
At node 5, we first copy the OUT set of node 4 to the IN set (write {a-b, a*b} in IN[5]). Node 5 seems to generate a-1 but in fact it does not, since it immediately overwrites a. Also, we'll need to kill each expression in our IN set that uses a. That's expressions a-b and a*b. So we're left with an empty OUT set, which we also copy over to the IN set of node 6 (write ∅ in OUT[5] and IN[6]).

At node 6 we generate a-b once again, and so our new OUT set is a-b (write {a-b} in OUT[6]).

At node 7, we copy OUT[4] to IN[7] (write {a-b, a*b} in IN[7]) at which point we've completed our first pass through the table.

## Available Expressions Analysis

| n | IN[n] | OUT[n] |
|---|---|---|
| 1 | -- | ∅ |
| 2 | ∅ | {a-b} |
| 3 | {a-b} | {a-b, a*b} |
| 4 | {a-b} | {a-b} |
| 5 | {a-b} | ∅ |
| 6 | ∅ | {a-b} |
| 7 | {a-b} | -- |




In the next pass, we revisit node 4. We need to make sure that the expressions in IN[4] are available on all program paths to this point, so we take the intersection of OUT[3] with OUT[6], leaving just a-b (replace IN[4] by {a-b}). This reflects the fact that the expression a * b is not in fact available at this point.

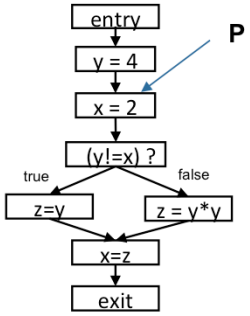
Passing through the remaining nodes after node 4, OUT[4], IN[5], and IN[7] become just a-b (replace all of these by {a-b}).

No more changes will be made by additional iterations, so this completes the analysis.

## Live Variables Analysis

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




Now let's move on to the final of our four classical dataflow analyses, called Live Variable Analysis.

The goal of live variable analysis is to determine for each program point, which variables may be live at the exit from the point, where a variable is live if there is a path to a use of the variable that does not re-define the variable.

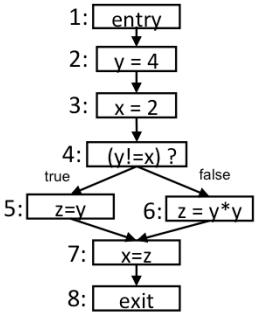
Let us look at the following example program. There are three variables in this program: x, y, and z.

Consider the program point P at the entry of this assignment to x (draw P and the arrow).

The variable y is live here because there is a path to a use of y here (gesture) that does not re-define y. The variable x, on the other hand, is not live at program point P because, even though there is a path to a use of x here (gesture), that path redefines x here (gesture). The variable z is also not live at program point P because along each of these paths emanating from P, z is redefined here and here (gesture) before it is used.

## Live Variables Analysis

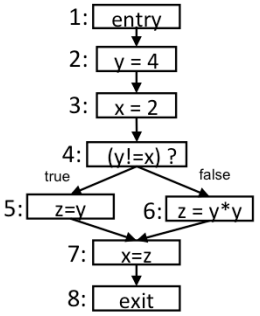
| n | IN[n] | OUT[n] |
|---|---|---|
| 1 | -- | ∅ |
| 2 | ∅ | ∅ |
| 3 | ∅ | ∅ |
| 4 | ∅ | ∅ |
| 5 | ∅ | ∅ |
| 6 | ∅ | ∅ |
| 7 | ∅ | ∅ |
| 8 | ∅ | -- |




Let's work through this program and complete the live variables analysis for it. Remember that the IN set of a node is every live variable before the node, and the OUT set is every variable which is live after the node.

## Live Variables Analysis

| n | IN[n] | OUT[n] |
|---|---|---|
| 1 | -- | ∅ |
| 2 | ∅ | { y } |
| 3 | { y } | { x, y } |
| 4 | { x, y } | { y } |
| 5 | { y } | { z } |
| 6 | { y } | { z } |
| 7 | { z } | ∅ |
| 8 | ∅ | -- |




We'll start at the exit point. We won't be needing any variables at this point, so it begins with the empty set as its IN set (gesture at IN[8]).

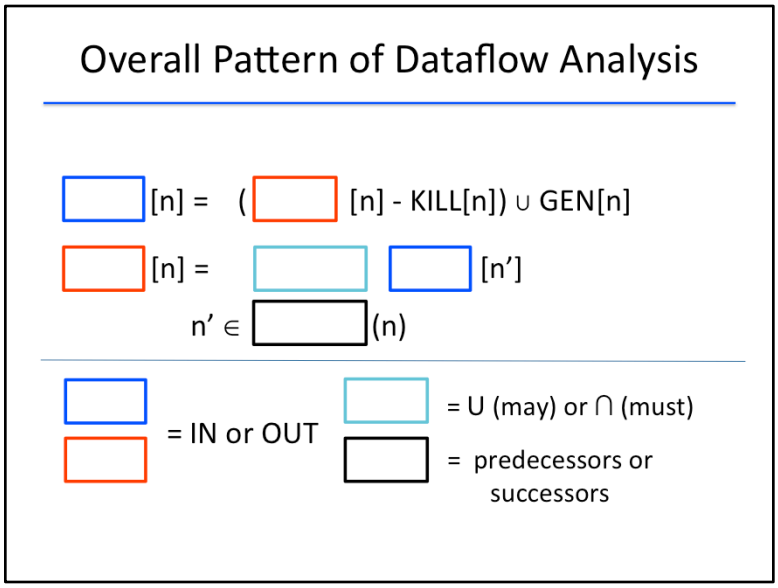
At node 7, the OUT set is the same as the IN set of its successor, node 8 (gesture at OUT[7]). For the IN set, we take whatever our OUT set is, kill any variables which we redefine, and then add any variables which are used in the node. This results in an IN set of z (write {z} in IN[7]).

Both nodes 5 and 6 take the IN set of node 7 as their OUT set (write {z} in OUT[5] and OUT[6]). Node 6 redefines z and uses y, so we remove z from OUT and add in y to obtain the IN set (write {y} in IN[6]). Node 5 also redefines z and uses y, so we kill z and add y to get the IN set (write {y} in IN[5]).

Next we have program point 4, an if-statement. Its OUT set is {y}, the union of the IN sets of nodes 5 and 6 (write {y} in OUT[4]). It uses both y and x without redefining any variables, so its IN set is {x,y} (write {x,y} in IN[4]). We then copy over this IN set to the OUT set of node 3 (write {x,y} in OUT[3]).

The rest of the example does not use any variables but sets them to constants. Node 3 kills x in between its OUT and IN sets (write {y} in IN[3] and OUT[2]). Node 2 kills y in between its OUT and IN sets (write ∅ in OUT[1]).

What's interesting is that even though this program has 3 variables x, y, and z, at no point are more than two of these three variables simultaneously live. This information can be used to generate assembly code that uses only two instead of three registers for storing the contents of these variables. Using fewer registers in turn can generate more efficient assembly code, by avoiding the need to store the contents of these variables in memory.




As you may have noticed at this point, the four dataflow analyses that we discussed follow a common pattern in the two operations that they apply.

This pattern is as follows (show the pattern).

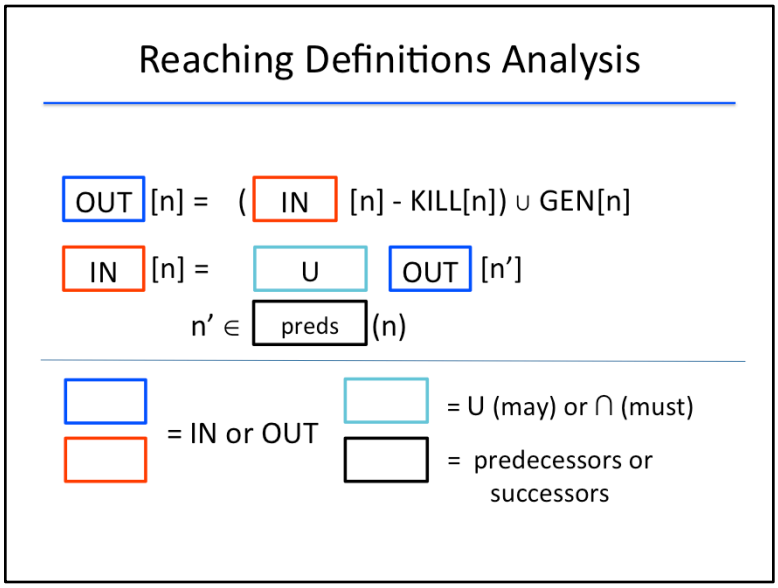
The blue and red boxes represent the IN or OUT sets. The purple box represents the set union or set intersection operator. The black box represents the immediate predecessors or immediate successors of a node.

Each of our four dataflow analyses corresponds to a different instantiation of these boxes. Although this looks like a lot of choices, there are in fact only two: first, whether the analysis propagates information forward or backward, and second, whether the analysis computes "may" or "must" information.

We already saw what forward versus backward propagation looks like when we discussed the four analyses earlier. Forward versus backward propagation is decided by mapping the blue and red boxes to IN and OUT sets appropriately, and by mapping the black box to predecessors or successors accordingly.

Now let's see what "may" versus "must" information means. Intuitively, an analysis is said to compute "may" facts if those facts hold along some path in the control-flow graph. In contrast, an analysis is said to compute "must" facts if those facts hold along all paths. Thus, the "may" versus "must" property of an analysis is decided by mapping the purple box to the set union operator in the case of "may" analysis and to the set intersection operator in the case of "must" analysis.

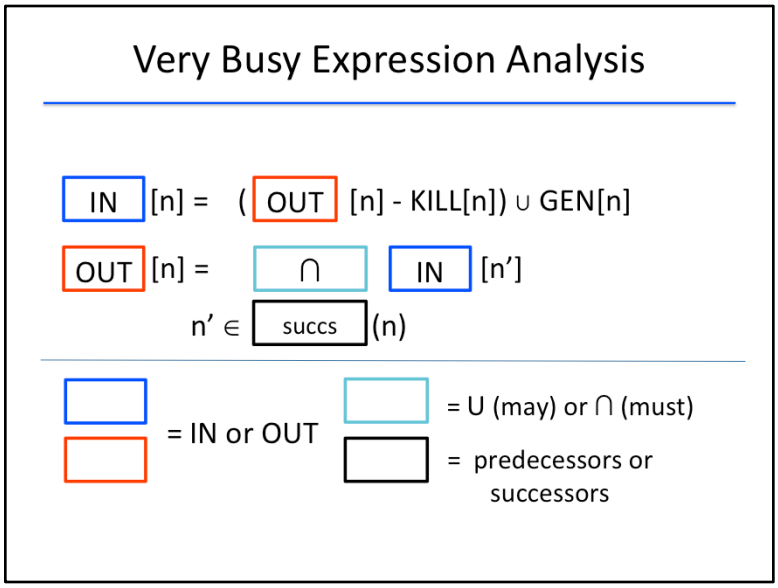
Now that we have reviewed the overall pattern, let's instantiate it in turn for each of our four dataflow analyses.




We'll start with reaching-definitions analysis, whose rules we examined earlier in the lesson.

This analysis computes "may" information, which is evident from the goal of the analysis, which is to find definitions that could reach a program point along some path. Therefore, we fill in the purple box with set union.

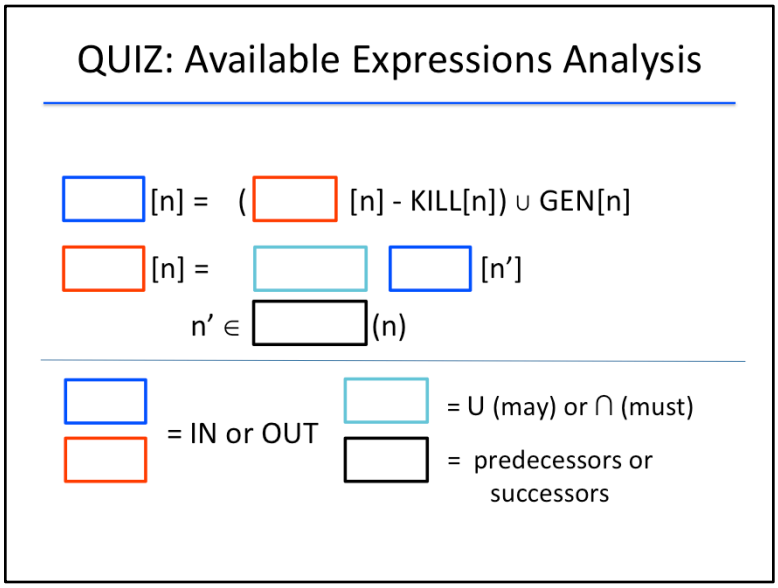
Furthermore, the analysis propagates this information about reaching definitions forward in the control-flow graph. Hence, we fill in the blue boxes with OUT sets, the red boxes with IN sets, and the black box with predecessors.




Next let's look at very-busy-expressions analysis. This is the polar opposite of reaching definitions analysis: it is a backward, "must" analysis.

This analysis computes "must" information, because the goal of the analysis is to find expressions that are used along all paths before any variable occurring in them is redefined. Therefore, we fill in the purple box with the set intersection operator.

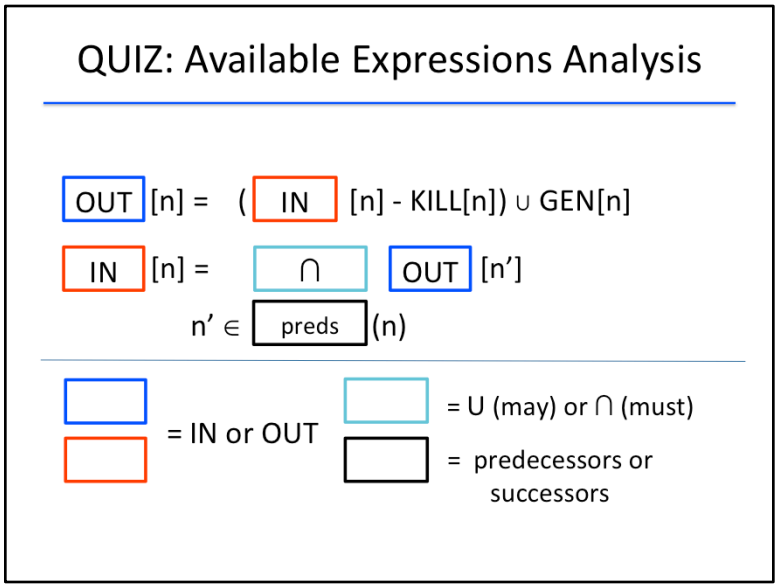
Furthermore, the analysis propagates this information about very busy expressions backwards in the control-flow graph. Therefore, we fill in the blue boxes with IN sets, the red boxes with OUT sets, and the black box with successors.




{QUIZ SLIDE}

Now let's fill in the pattern for available expressions analysis. We will do this in the form of an exercise.

Fill in the six boxes with the appropriate values. Type either the word "union" or "intersect" in the purple box, type either predecessors or successors in the black box, and type either "IN" or "OUT" in the red and blue boxes. Remember to use the same value in the two blue boxes, and likewise, the same value in the two red boxes.

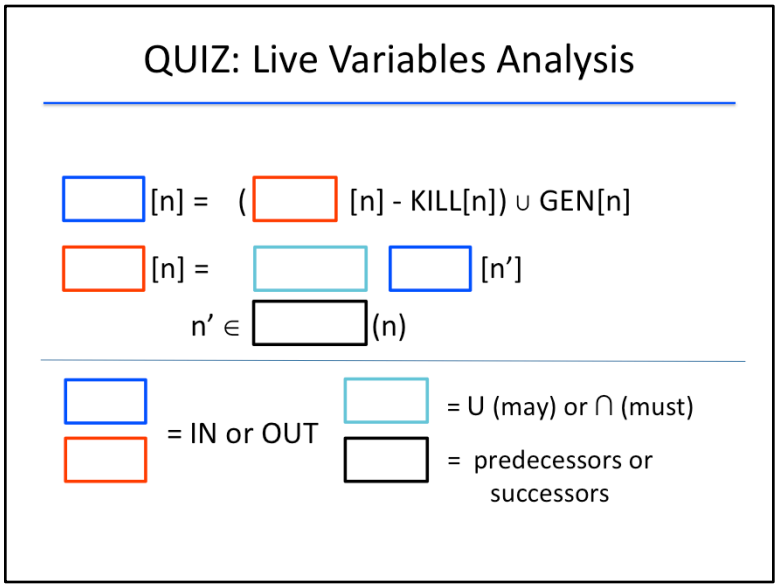



{SOLUTION SLIDE}

Let's review the solution. Available-expressions analysis is a forward, must analysis.

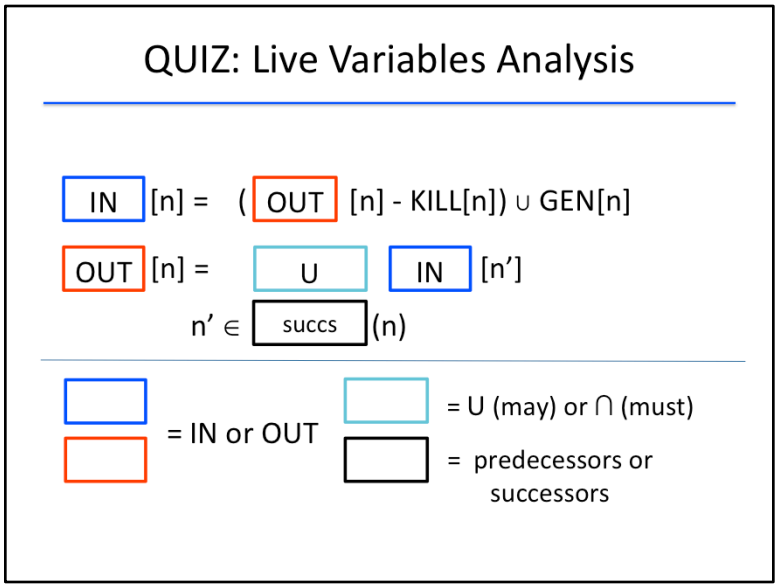
It is a must analysis because it seeks to find expressions that are available at a program point, that is, expressions that must have been computed along all paths leading to a point. So we instantiate the purple box with the set intersection operator.

This analysis propagates information about available expressions forward in the control-flow graph. Hence, we instantiate the blue boxes with OUT sets, the red boxes with IN sets, and the black box with predecessors.




{QUIZ SLIDE}

Finally, let's instantiate the pattern for live variables analysis. Let's do this in the form of an exercise as well. Fill in the six boxes with the appropriate values.




{SOLUTION SLIDE}

Let's review the solution. Live-variables analysis is a backward, may analysis.

It is a may analysis because it seeks to find live variables: variables that may be used along some path before being re-defined. So we fill in the purple box with the set union operator.

This analysis propagates information about live variables backwards in the control-flow graph. Hence, we instantiate the blue boxes with IN sets, the red boxes with OUT sets, and the black box with successors.

## QUIZ: Classifying Dataflow Analyses

Match each analysis with its characteristics.

| | May | Must |
|---|---|---|
| Forward | | |
| Backward | | |

Very Busy Expressions | Reaching Definitions | Live Variables | Available Expressions

{QUIZ SLIDE}

We have seen four different dataflow analyses along with their characteristics along two important dimensions: forward versus backward and may versus must.

Let's finish the lesson with a brief review of these characteristics. Match each of the four dataflow analyses with their corresponding properties. Type in the letter of the corresponding analysis into each box.

## QUIZ: Classifying Dataflow Analyses

Match each analysis with its characteristics.

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

{SOLUTION SLIDE}

Let's review the solution.

Reaching definitions analysis is a forward, may analysis. Available expressions analysis is a forward, must analysis. Live variables analysis is a backward, may analysis. And finally, very busy expressions analysis is a backward, must analysis.

## What Have We Learned?

- What is dataflow analysis
- Reasoning about flow of data using control-flow graphs
- Specifying dataflow analyses using local rules
- Chaotic iteration algorithm to compute global properties
- Four classical dataflow analyses
- Classification: forward vs. backward, may vs. must

Let's recap the main topics that we have covered in this lesson.

We introduced dataflow analysis, a common kind of static analysis that enables to reason about the flow of data in program runs.

We learnt how to reason about this flow of data using a program representation called a control-flow graph which concisely represents all runs of a program.

We saw how a dataflow analysis can be specified using local dataflow rules.

We learnt the chaotic iteration algorithm which repeatedly applies such rules to compute global dataflow properties.

We defined and saw examples of the four different classical dataflow analyses: Reaching Definitions Analysis, Very Busy Expressions Analysis, Available Expressions Analysis, and Live Variables Analysis. We compared and contrasted these analyses along two important dimensions: forward vs. backward, and may vs. must.

This lesson focused on how to reason about the flow of primitive data such as integers. In the next lesson, we will learn how to reason about the flow of non-primitive data, better known as pointers, objects, or references.