# Data-Flow Analysis

Dragon Book, Chapter 9, Section 9.2, 9.3, 9.4

# Data-Flow Analysis

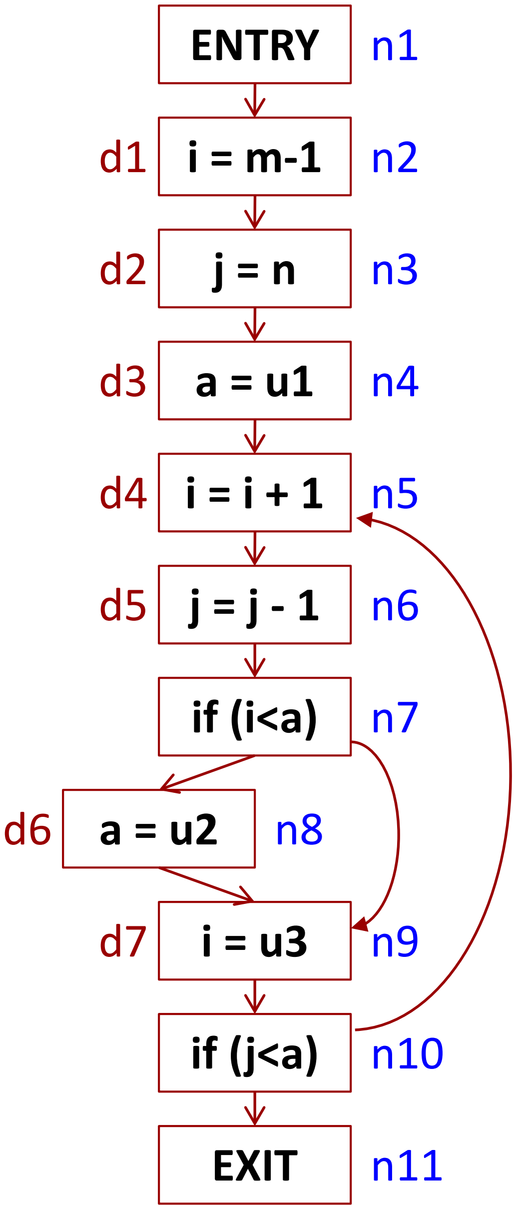
- Data-flow analysis is a sub-area of static program analysis (aka compile-time analysis)
  - Used in the compiler back end for optimizations of three-address code and for generation of target code
  - For software engineering tools: software understanding, restructuring, testing, verification
- Attaches to each CFG node some information that describes properties of the program at that point
  - Based on lattice theory
- Defines algorithms for inferring these properties
  - e.g., fixed-point computation

# Example: Reaching Definitions

- A classical example of a data-flow analysis
  - We will consider intraprocedural analysis: only inside a single procedure, based on its CFG
- For ease of discussion, pretend that the CFG nodes are individual instructions, not basic blocks
  - Each node defines two program points: immediately before and immediately after
- Goal: identify all connections between variable definitions ("write") and variable uses ("read")
  - **x = y + z** has a definition of **x** and uses of **y** and **z**

# Reaching Definitions

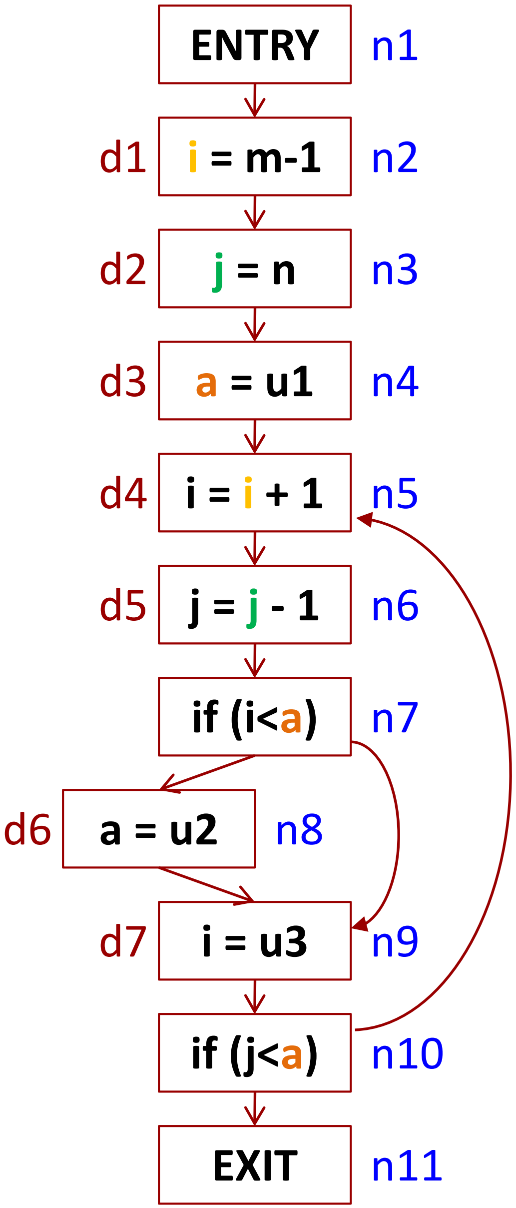
- A definition *d* reaches a program point *p* if there exists a CFG path that
  - starts at the program point immediately after *d*
  - ends at *p*
  - does **not** contain a definition of *d* (i.e., *d* is not “killed”)
- The CFG path may be impossible (*infeasible*) at run time
  - Any compile-time analysis has to be *conservative*, so we consider all paths in the CFG
- For a CFG node *n*
  - IN[*n*] is the set of definitions that reach the program point immediately before *n*
  - OUT[*n*] is the set of definitions that reach the program point immediately after *n*
  - Reaching definitions analysis computes IN[*n*] and OUT[*n*]




OUT[n1] = { }
IN[n2] = { }
OUT[n2] = { d1 }
IN[n3] = { d1 }
OUT[n3] = { d1, d2 }
IN[n4] = { d1, d2 }
OUT[n4] = { d1, d2, d3 }
IN[n5] = { d1, d2, d3, d5, d6, d7 }
OUT[n5] = { d2, d3, d4, d5, d6 }
IN[n6] = { d2, d3, d4, d5, d6 }
OUT[n6] = { d3, d4, d5, d6 }
IN[n7] = { d3, d4, d5, d6 }
OUT[n7] = { d3, d4, d5, d6 }
IN[n8] = { d3, d4, d5, d6 }
OUT[n8] = { d4, d5, d6 }
IN[n9] = { d3, d4, d5, d6 }
OUT[n9] = { d3, d5, d6, d7 }
IN[n10] = { d3, d5, d6, d7 }
OUT[n10] = { d3, d5, d6, d7 }
IN[n11] = { d3, d5, d6, d7 }

# Uses of Reaching Definitions Analysis

- Def-use (du) chains
  - For a given definition (i.e., write) of a variable, which statements read the value created by the def?
- Use-def (ud) chains
  - For a given use (i.e., read) of a variable, which statements performed the write of this value?
  - The reverse of du-chains
- Goal: potential write-read (flow) data dependences
  - Compiler optimizations
  - Program understanding (e.g., slicing)
  - Data-flow-based testing: coverage criteria
  - Semantic checks: e.g., use of uninitialized variables




OUT[n1] = { }
IN[n2] = { }
OUT[n2] = { d1 }
IN[n3] = { d1 }
OUT[n3] = { d1, d2 }
IN[n4] = { d1, d2 }
OUT[n4] = { d1, d2, d3 }
IN[n5] = { d1, d2, d3, d5, d6, d7 }
OUT[n5] = { d2, d3, d4, d5, d6 }
IN[n6] = { d2, d3, d4, d5, d6 }
OUT[n6] = { d3, d4, d5, d6 }
IN[n7] = { d3, d4, d5, d6 }
OUT[n7] = { d3, d4, d5, d6 }
IN[n8] = { d3, d4, d5, d6 }
OUT[n8] = { d4, d5, d6 }
IN[n9] = { d3, d4, d5, d6 }
OUT[n9] = { d3, d5, d6, d7 }
IN[n10] = { d3, d5, d6, d7 }
OUT[n10] = { d3, d5, d6, d7 }
IN[n11] = { d3, d5, d6, d7 }

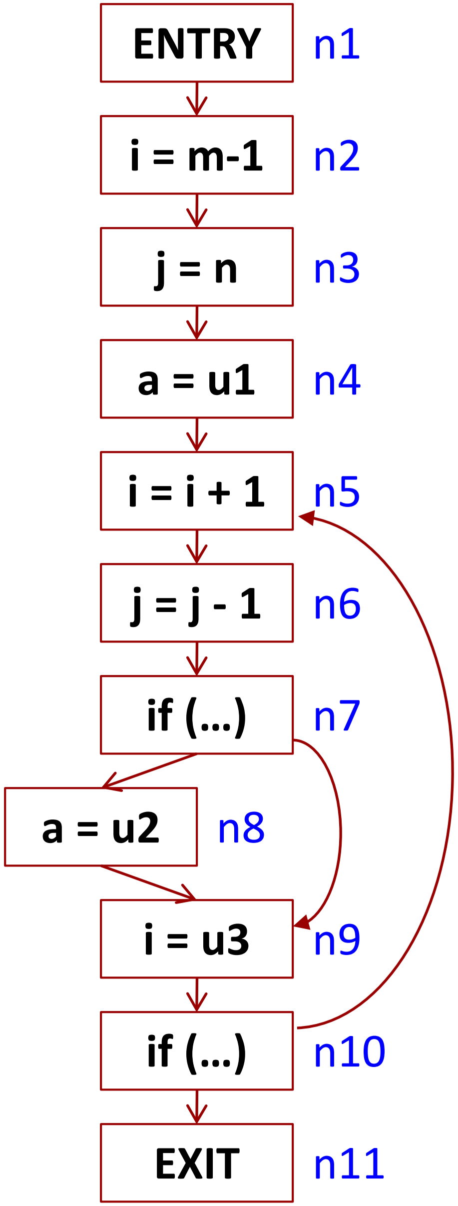
*Def-use chains for d1:*
DU(d1): uses of i in nodes with d1 ∈ IN[*n*]
DU(d1) = { n5 }

Other examples:
DU(d2) = { n6 }
DU(d3) = { n7, n10 }
DU(d4) = { n7 }
DU(d5) = { n10, n6 }
DU(d6) = { n10, n7 }
DU(d7) = { n5 }

*Use-def chains:*
UD(i@n5) = { d1, d7 }
UD(j@n6) = { d2, d5 }
UD(i@n7) = { d4 }
UD(a@n7) = { d3, d6 }
UD(j@n10) = { d5 }
UD(a@n10) = { d3,d6 }

# Example: Live Variables

- A variable *v* is **live** at a program point *p* if there exists a CFG path that
  - starts at *p*
  - ends immediately before some statement that reads *v*
  - does **not** contain a definition of *v*
- Thus, the value that *v* has at *p* could be used later
  - "could" because the CFG path may be infeasible
  - If *v* is not live at *p*, we say that *v* is **dead** at *p*
- For a CFG node *n*
  - IN[*n*] is the set of variables that are live at the program point immediately before *n*
  - OUT[*n*] is the set of variables that are live at the program point immediately after *n*




OUT[n1] = { m, n, u1, u2, u3 }
IN[n2] = { m, n, u1, u2, u3 }
OUT[n2] = { n, u1, i, u2, u3 }
IN[n3] = { n, u1, i, u2, u3 }
OUT[n3] = { u1, i, j, u2, u3 }
IN[n4] = { u1, i, j, u2, u3 }
OUT[n4] = { i, j, u2, u3 }
IN[n5] = { i, j, u2, u3 }
OUT[n5] = { j, u2, u3 }
IN[n6] = { j, u2, u3 }
OUT[n6] = { u2, u3, j }
IN[n7] = { u2, u3, j }
OUT[n7] = { u2, u3, j }
IN[n8] = { u2, u3, j }
OUT[n8] = { u3, j, u2 }
IN[n9] = { u3, j, u2 }
OUT[n9] = { i, j, u2, u3 }
IN[n10] = { i, j, u2, u3 }
OUT[n10] = { i, j, u2, u3 }
IN[n11] = { }

Uses of Live Variables

- Dead code elimination: e.g., when **x** is not live at **x=y+z**
- Register allocation

# Example: Constant Propagation

- Can we guarantee that the value of a variable *v* at a program point *p* is always a known constant?
- Compile-time constants are quite useful
  - Constant folding: e.g., if we know that v is always 3.14 immediately before **w = 2*v**; replace it **w = 6.28**
    - Often due to symbolic constants
  - Dead code elimination: e.g., if we know that v is always false at **if (v) …**
  - Program understanding, restructuring, verification, testing, etc.
- Very similar to the abstract interpretation we discussed earlier

# Basic Ideas

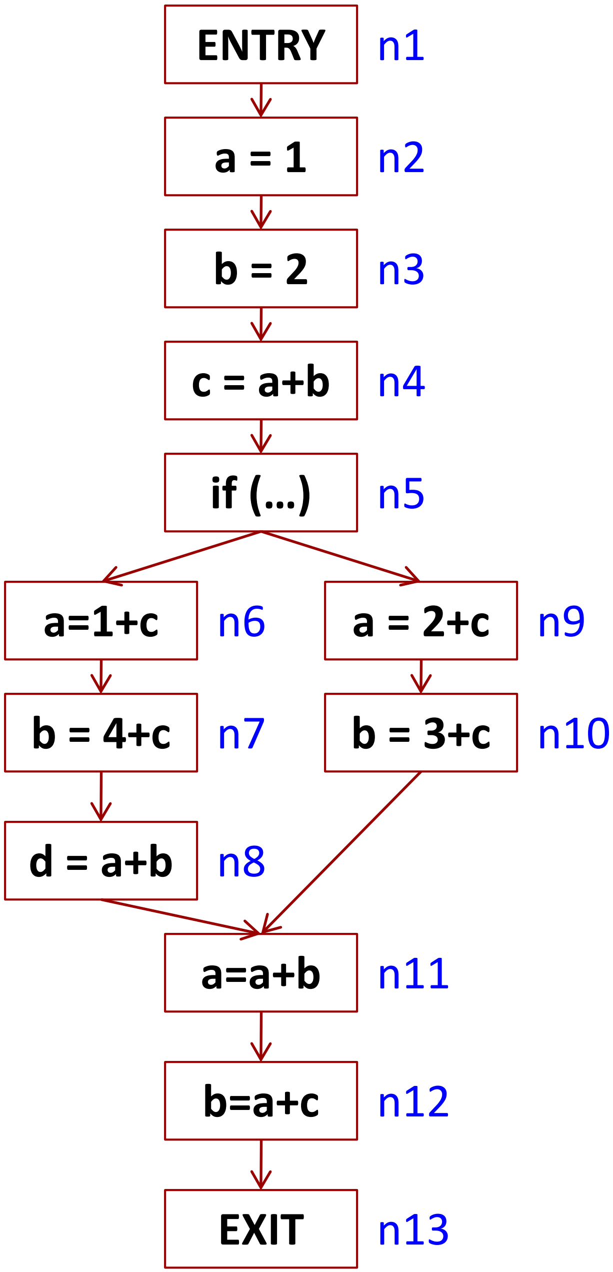
- At each CFG node *n*, IN[*n*] is a map Vars → Values
  - Each variable v is mapped to a value x ∈ Values
  - Values = all possible constant values ∪ { *any* }
- Special value *any* (not-a-constant) means that the variable cannot be definitely proved to be a compile-time constant at this program point
  - E.g., the value comes from user input, file I/O, network
  - E.g., the value is 5 along one branch of an if statement, and 6 along another branch of the if statement
  - E.g., value comes from some variable with *any* value

# Formulation as a System of Equations

- OUT[ENTRY] = empty map
- For any other CFG node *n*
  - IN[*n*] = Merge(OUT[*m*]) for all predecessors *m* of *n*
  - OUT[*n*] = Update(IN[*n*])
- Merging two maps: if v is mapped to $c_1$ and $c_2$ respectively, in the merged map v is mapped to:
  - if $c_1$ = *any* or $c_2$ = *any*, the result it *any*
  - Else if $c_1 \neq c_2$, the result is *any*
  - Else the result is $c_1$ (in this case we know that $c_1 = c_2$)
  - Remember IfStmt from Project 4?

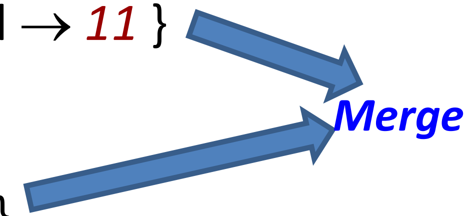
# Formulation as a System of Equations

- Updating a map at an assignment **v = …**
  - If the statement is not an assignment, OUT[*n*] = IN[*n*]
- The map does not change for any w ≠ v
- If we have ***v = c***, where *c* is a constant: in OUT[*n*], v is now mapped to *c*
- If we have **v = p + q** (or similar binary operators) and IN[*n*] maps p and q to $c_1$ and $c_2$ respectively
  - If both $c_1$ and $c_2$ are constants: result is $c_1+c_2$
  - Else, $c_1$ or $c_2$ or both are *any* and the result is *any*




OUT[n1] = { }
OUT[n2] = { a → *1* }
OUT[n3] = { a → *1*, b → *2* }
OUT[n4] = { a → *1*, b → *2*, c → *3* }

OUT[n6] = { a → *4*, b → *2*, c → *3* }
OUT[n7] = { a → *4*, b → *7*, c → *3* }
OUT[n8] = { a → *4*, b → *7*, c → *3*, d → *11* }




OUT[n9] = { a → *5*, b → *2*, c → *3* }
OUT[n10] = { a → *5*, b → *6*, c → *3* }

IN[n11] = { a → *any*, b → *any*, c → *3* }
OUT[n11] = { a → *any*, b → *any*, c → *3* }

OUT[n12] = { a → *any*, b → *any*, c → *3* }

Note: at the exit node a and b are compile-time constants, but this analysis is not powerful enough to infer this