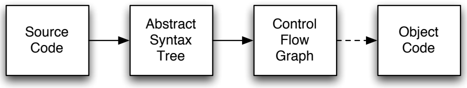
CMSC 631 — Program Analysis and Understanding
Fall 2004

Data Flow Analysis

Compiler Structure



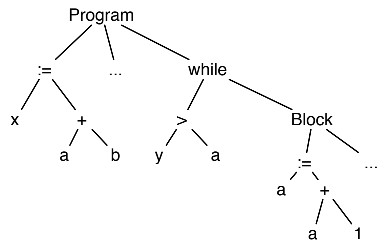
- Source code parsed to produce AST
- AST transformed to CFG
- Data flow analysis operates on control flow graph (and other intermediate representations)

Abstract Syntax Tree (AST)

- Programs are written in text
  - I.e., sequences of characters
  - Awkward to work with
- First step: Convert to structured representation
  - Use lexer (like flex) to recognize *tokens*
    - Sequences of characters that make words in the language
  - Use parser (like bison) to group words structurally
    - And, often, to produce AST

Abstract Syntax Tree Example

```
x := a + b;
y := a * b;
while (y > a) {
  a := a + 1;
  x := a + b
}
```



ASTs

- ASTs are *abstract*
  - They don't contain all information in the program
    - E.g., spacing, comments, brackets, parentheses
  - Any ambiguity has been resolved
    - E.g., a + b + c produces the same AST as (a + b) + c
- For more info, see CMSC 430
  - In this class, we will generally begin at the AST level

Disadvantages of ASTs

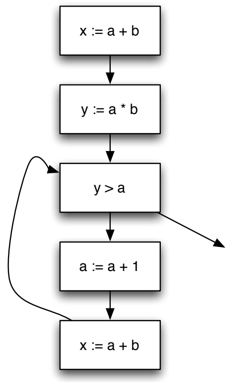
- AST has many similar forms
  - E.g., for, while, repeat...until
  - E.g., if, ?:, switch
-
- Expressions in AST may be complex, nested
  - (42 * y) + (z > 5 ? 12 * z : z + 20)
- Want simpler representation for analysis
  - ...at least, for dataflow analysis

Control-Flow Graph (CFG)

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow
- Statements may be
  - Assignments x := y op z or x := op z
  - Copy statements x := y
  - Branches goto L or if x relop y goto L
  - etc.

Control-Flow Graph Example

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

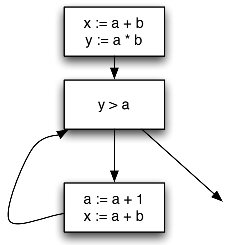


Variations on CFGs

- We usually don't include declarations (e.g., int x;)
  - But there's usually something in the implementation
- May want a unique entry and exit node
  - Won't matter for the examples we give
- May group statements into basic blocks
  - A sequence of instructions with no branches into or out of the block

Control-Flow Graph w/Basic Blocks

```
x := a + b;
y := a * b;
while (y > a + b) {
   a := a + 1;
   x := a + b
}
```



- Can lead to more efficient implementations
- But more complicated to explain, so...
  - We'll use single-statement blocks in lecture today

CFG vs. AST

- CFGs are much simpler than ASTs
  - Fewer forms, less redundancy, only simple expressions
- But...AST is a more faithful representation
  - CFGs introduce temporaries
  - Lose block structure of program
- So for AST,
  - Easier to report error + other messages
  - Easier to explain to programmer
  - Easier to unparse to produce readable code

Data Flow Analysis

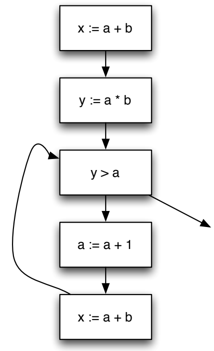
- A framework for proving facts about programs
- Reasons about lots of little facts
- Little or no interaction between facts
  - Works best on properties about *how* program computes
- Based on all paths through program
  - Including infeasible paths

Available Expressions

- An expression e is available at program point p if
  - e is computed on every path to p, and
  - the value of e has not changed since the last time e is computed on p
- Optimization
  - If an expression is available, need not be recomputed
    - (At least, if it's still in a register somewhere)

Data Flow Facts

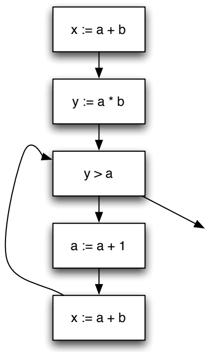
- Is expression e available?
- Facts:
  - a + b is available
  - a * b is available
  - a + 1 is available



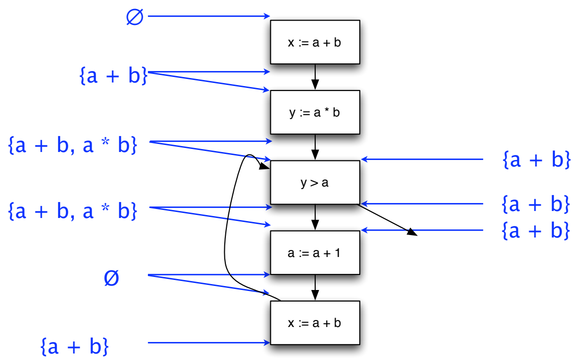
Gen and Kill

- What is the effect of each statement on the set of facts?

Stmt	Gen	Kill
x := a + b	a + b	
y := a * b	a * b	
a := a + 1		a + 1, a + b, a * b



Computing Available Expressions



Terminology

- A *joint point* is a program point where two branches meet
- Available expressions is a *forward must* problem
  - Forward = Data flow from in to out
  - Must = At join point, property must hold on all paths that are joined

Data Flow Equations

- Let s be a statement
  - succ(s) = { immediate successor statements of s }
  - pred(s) = { immediate predecessor statements of s}
  - In(s) = program point just before executing s
  - Out(s) = program point just after executing s
- $In(s) = \bigcap_{s' \in pred(s)} Out(s')$
- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
  - Note: These are also called *transfer functions*

Liveness Analysis

- A variable v is *live* at program point p if
  - v will be used on some execution path originating from p...
  - before v is overwritten
- Optimization
  - If a variable is not live, no need to keep it in a register
  - If variable is dead at assignment, can eliminate assignment

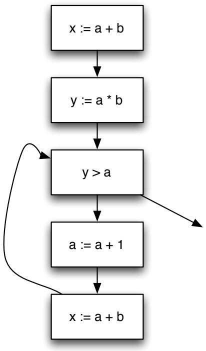
Data Flow Equations

- Available expressions is a forward must analysis
  - Data flow propagate in same dir as CFG edges
  - Expr is available only if available on all paths
- Liveness is a *backward may* problem
  - To know if variable live, need to look at future uses
  - Variable is live if available on some path
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_{s' \in succ(s)} In(s')$

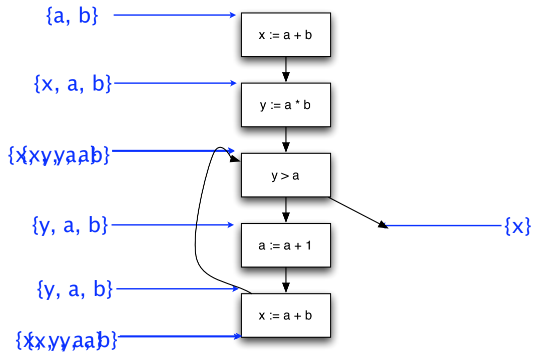
Gen and Kill

- What is the effect of each statement on the set of facts?

Stmt	Gen	Kill
x := a + b	a, b	x
y := a * b	a, b	y
y > a	a, y	
a := a + 1	a	a



Computing Live Variables



Very Busy Expressions

- An expression e is very busy at point p if
  - On every path from p, e is evaluated before the value of e is changed
- Optimization
  - Can hoist very busy expression computation
- What kind of problem?
  - Forward or backward? backward
  - May or must? must

Reaching Definitions

- A *definition* of a variable v is an assignment to v
- A definition of variable v reaches point p if
  - There is no intervening assignment to v
- Also called def-use information
- What kind of problem?
  - Forward or backward? forward
  - May or must? may

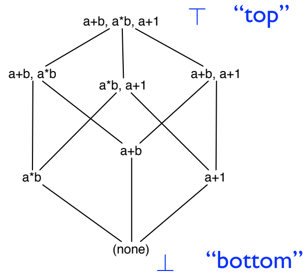
Space of Data Flow Analyses

	May	Must
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

- Most data flow analyses can be classified this way
  - A few don't fit: bidirectional analysis
- Lots of literature on data flow analysis

Data Flow Facts and Lattices

- Typically, data flow facts form a lattice
  - Example: Available expressions



Partial Orders

- A partial order is a pair $(P, \leq)$ such that
  - $\leq \subseteq P \times P$
  - $\leq$ is reflexive: $x \leq x$
  - $\leq$ is anti-symmetric: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - $\leq$ is transitive: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Lattices

- A partial order is a lattice if $\sqcap$ and $\sqcup$ are defined on any set:
  - $\sqcap$ is the *meet* or *greatest lower bound* operation:
    - $x \sqcap y \leq x$ and $x \sqcap y \leq y$
    - if $z \leq x$ and $z \leq y$, then $z \leq x \sqcap y$
  - $\sqcup$ is the *join* or *least upper bound* operation:
    - $x \leq x \sqcup y$ and $y \leq x \sqcup y$
    - if $x \leq z$ and $y \leq z$, then $x \sqcup y \leq z$

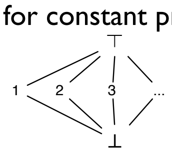
Lattices (cont'd)

- A finite partial order is a lattice if meet and join exist for every pair of elements
- A lattice has unique elements $\bot$ and $\top$ such that
  - $x \sqcap \bot = \bot$ $\quad x \sqcup \bot = x$
  - $x \sqcap \top = x$ $\quad x \sqcup \top = \top$
- In a lattice,

  $x \leq y$ iff $x \sqcap y = x$
  $x \leq y$ iff $x \sqcup y = y$

Useful Lattices

- $(2^S, \subseteq)$ forms a lattice for any set S
  - 2^S is the powerset of S (set of all subsets)
- If (S, ≤) is a lattice, so is (S, ≥)
  - I.e., lattices can be flipped
- The lattice for constant propagation



Forward Must Data Flow Algorithm

- Out(s) = Gen(s) for all statements s
  - Or, if you want, Out(s) = Top
- W := { all statements } (worklist)
- repeat
  - Take s from W
  - In(s) := $\cap_{s' \in \text{pred}(s)}$ Out(s')
  - temp := Gen(s) ∪ (In(s) - Kill(s))
  - if (temp != Out(s)) {
    - Out(s) := temp
    - W := W ∪ succ(s)
  - }
- until W = ∅

Monotonicity

- A function f on a partial order is *monotonic* if

$$x \leq y \Rightarrow f(x) \leq f(y)$$

- Easy to check that operations to compute In and Out are monotonic
  - In(s) := $\cap_{s' \in \text{pred}(s)}$ Out(s')
  - temp := Gen(s) ∪ (In(s) - Kill(s))
- Putting these two together,
  - temp := $f_s(\sqcap_{s' \in \text{pred}(s)} Out(s'))$

Termination

- We know the algorithm terminates because
  - The lattice has finite height
  - The operations to compute In and Out are monotonic
  - On every iteration, we remove a statement from the worklist and/or move down the lattice