Optimal Code Generation (for Expressions) and Data-Flow Analysis

Pat Morin

COMP 3002

Carleton UNIVERSITY

Canada's Capital University

Outline

- Optimal code generation
  - For expressions
  - By dynamic programming
- Data-Flow Analysis
  - Examples and Applications

Carleton UNIVERSITY
Canada's Capital University

Code Generation Using Ershov Numbers

Andrei Petrovych Ershov - Computer Science God

Carleton UNIVERSITY
Canada's Capital University

Optimal Code Generation for Expressions

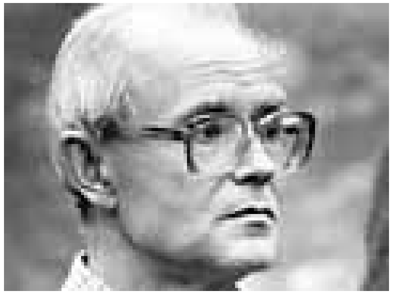
- When a basic block consists of a single expression in which each operand appears only once, we can generate "optimal" code
  - Uses the minimum number of registers, or
  - Uses minimum possible stack space
- We will label each node *v* of the expression tree with the smallest number of registers required to evaluate *v*'s subtree without using temporary variables
  - These labels are called *Ershov numbers*

Carleton UNIVERSITY
Canada's Capital University

Ershov Numbers

- We use the following rules to put a number on each node:
  - The label of a leaf is 1
  - The label of a unary node is equal to the label of its child
  - The label of a binary node is
    - The larger of the labels of its two children, if they are different, or
    - One plus the label of its two children, if they are the same

Carleton UNIVERSITY
Canada's Capital University



Understanding Ershov Numbers

- Ershov variables tell us the minimum number of registers required to evaluate an expression without requiring extra load/store operations
- The key rule with Ershov numbers happens with binary operators

Carleton UNIVERSITY
Canada's Capital University

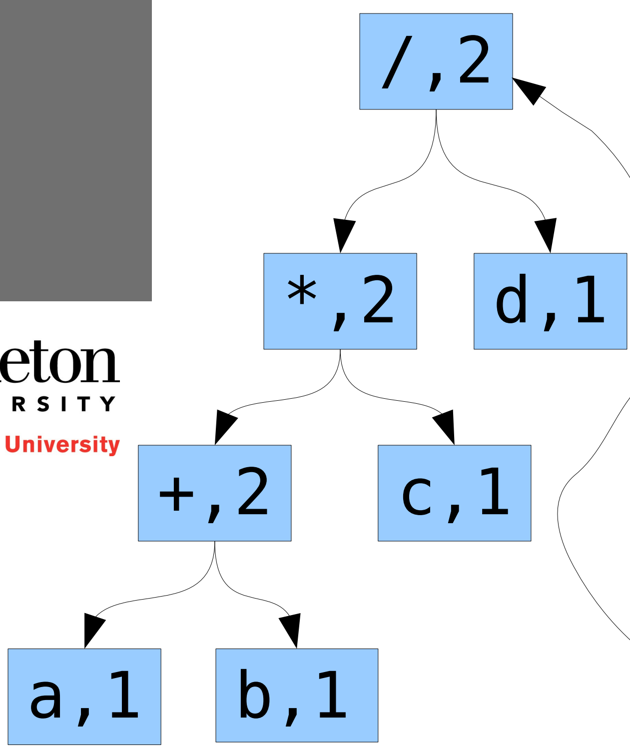
Ershov Number (Cont'd)

- If left child requires n registers and right child requires m >= n registers
  - Compute right child first, using m registers and store its value
  - Computer left child using n registers and store its value
    - requires n + 1 registers because of stored value
  - Combine two results and store in 1 register
- Total number of registers required in max(m, n+1)
  - Equal to m if m != n
  - Otherwise equal to m+1 = n+1

Carleton UNIVERSITY
Canada's Capital University

Ershov Number Example

- The following expression tree can be computed using two registers



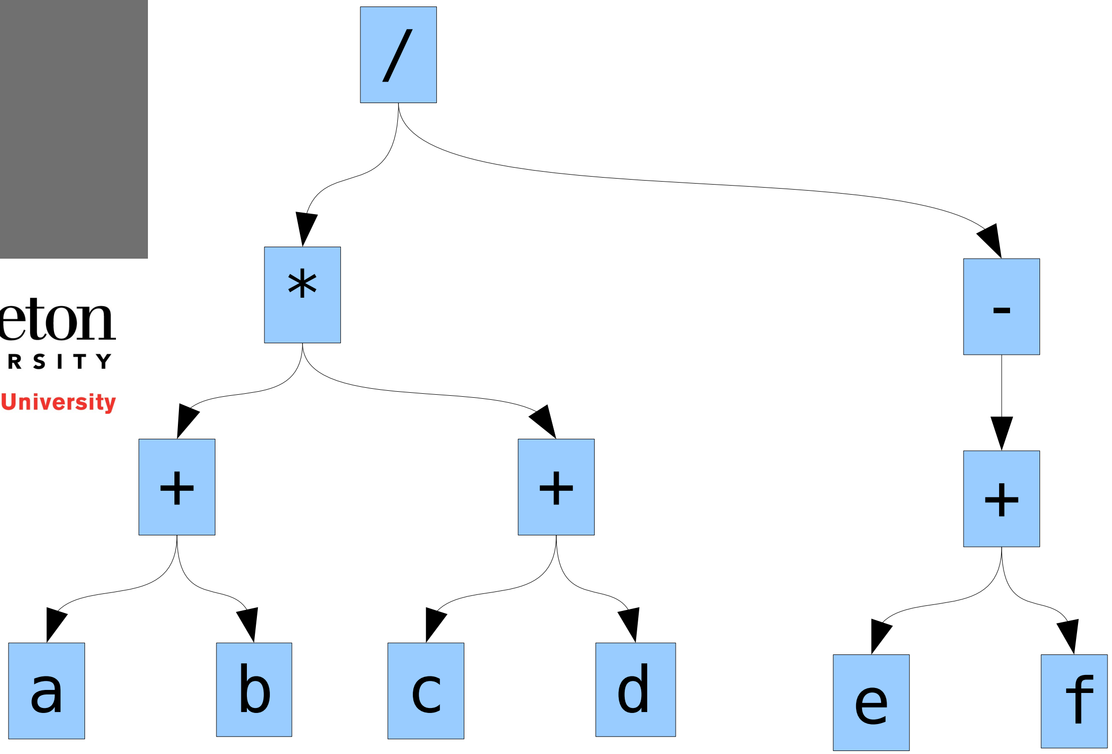

```
mov a, R0
mov b, R1
add R0, R1 ; result in R0
mov c, R1
mul R0, R1 ; result in R0
mov d, R1
div R0, R1 ; result in R0
```

((a+b)*c)/d

2 registers

Ershov Number Example

- Compute the Ershov numbers for the following



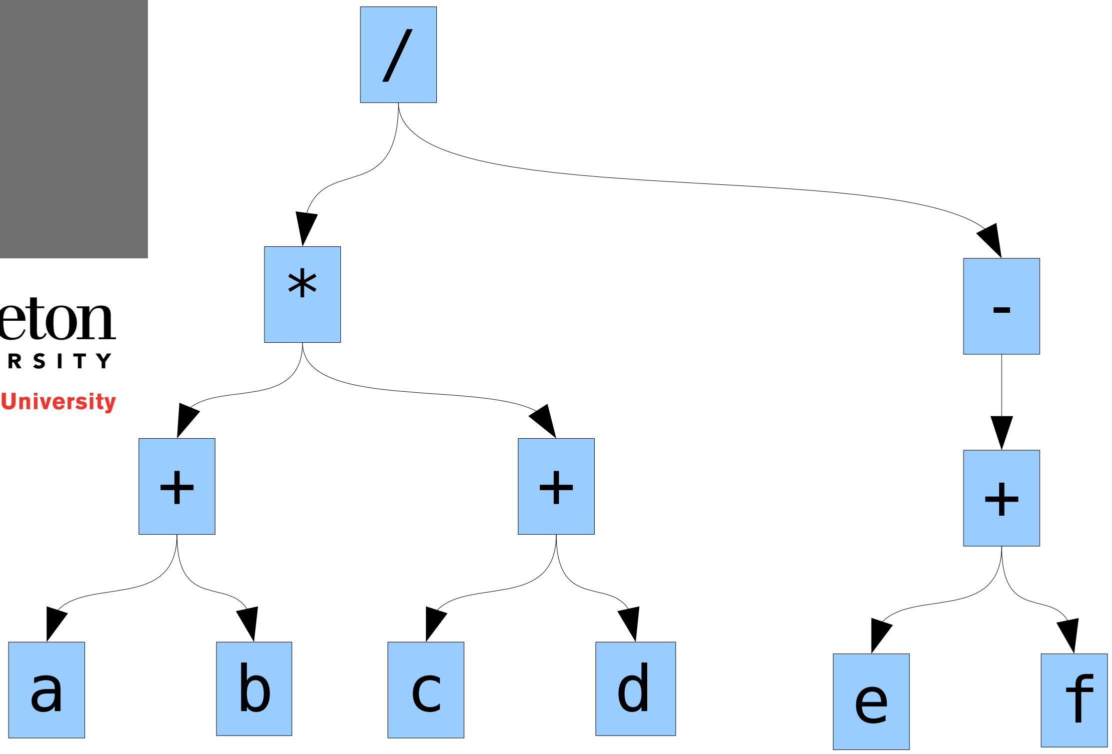
Register Shortages

- If the root's Ershov number k is greater than the number of registers r, then we need a different strategy
  1. Recursively generate code for the child with larger Ershov number
  2. Store the result in memory
  3. Recursively generate code for the smaller child
  4. Load the stored result from Step 2
  5. Generate code for the root
- It is possible to prove that this does the minimum number of possible load/store operations

Carleton UNIVERSITY Canada's Capital University

Ershov Number Example

- Generate code on a 2-register machine for the following:



Code Generation by Dynamic Programming

Carleton UNIVERSITY

Canada's Capital University

Dynamic programming matrix:

j → (sequence **y**)

i ↓ (sequence **x**)

		0	1 **T**	2 **G**	3 **C**	4 **T**	5 **C**	6 **G**	7 **T**	8 = N **A**
0		0	–6	–12	–18	–24	–30	–36	–42	–48
1	**T**	–6	5	–1	–7	–13	–19	–25	–31	–37
2	**T**	–12	–1	3	–3	–2	–8	–14	–20	–26
3	**C**	–18	–7	–3	8	2	3	–3	–9	–15
4	**A**	–24	–13	–9	2	6	0	1	–5	–4
5	**T**	–30	–19	–15	–4	7	4	–2	6	0
M = 6	**A**	–36	–25	–21	–10	1	5	2	0	11

Optimum alignment scores 11:

T – – T C A T A

T G C T C G T A

+5 –6 –6 +5 +5 –2 +5 +5

Dynamic Programming and Ershov Numbers

- Ershov's algorithm produces an optimal result when
  - Every operand is distinct
  - Operands operate on two registers
  - Cost of every instruction is the same
- It is a special case of *dynamic programming*
  - To solve for a binary node *T*:
    - First solve each subtree of *T* independently
    - Try all different ways of combining *T*'s subtrees
- This can be generalized to less restrictive assumptions

Carleton UNIVERSITY
Canada's Capital University

Dynamic Programming

- Main idea:
  - Compute the cost of generating each subtree if
    - 1 register is available
    - 2 registers are available
    - 3 registers are available ...
  - To compute subtree v with 2 children using i registers we can
    - use i registers for left(v) and i-1 registers for right(v), or
    - use i-1 registers for left(v) and i registers for right(v), or
    - use i registers for left(v) and i registers for right(v)
  - In Case 3, we'll have to store left(v) while we compute right(v) and then load it

Carleton UNIVERSITY
Canada's Capital University

Dynamic Programming Algorithm

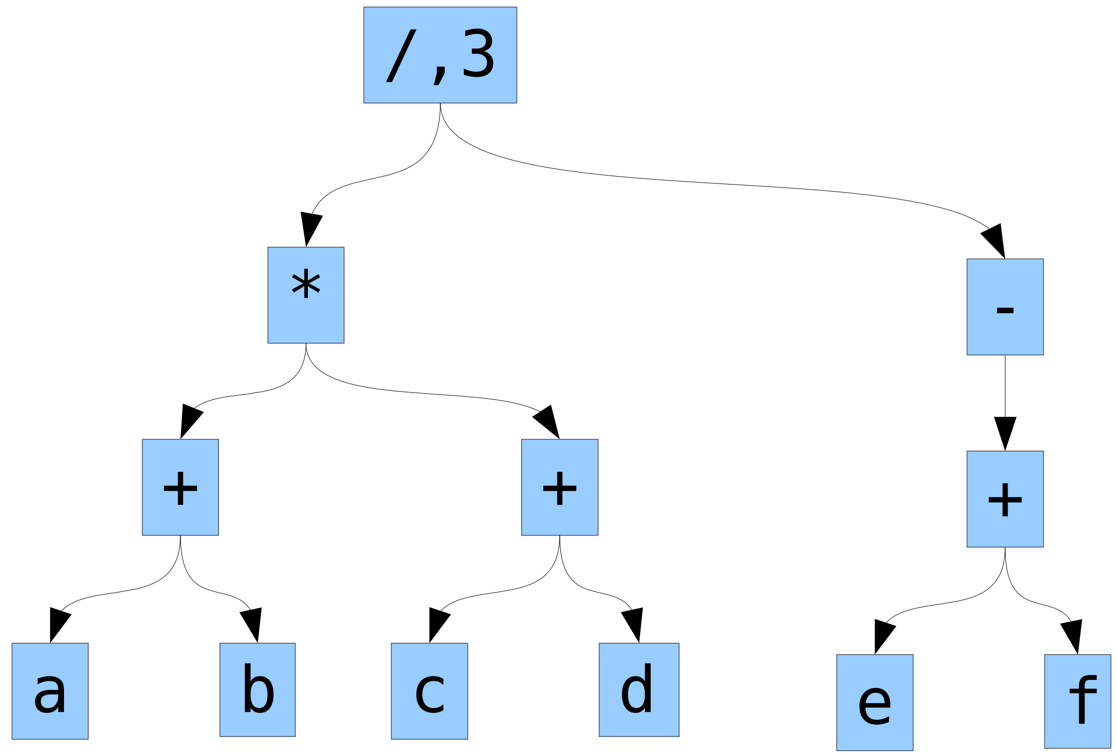
1. For each node v of T, compute the *cost vector* `C[1],...,C[r]`:
   - `C[i]` is the cost of evaluating *v* if `i` registers are available

- Cost of a leaf is 1 load
- For a node v of T with two children u and w we can compute $C_v[1],\ldots,C_v[r]$ using the rules
  - $C_v[i] <= C_u[i] + C_w[i-1] + op(v)$ [1]
  - $C_v[i] <= C_w[i] + C_u[i-1] + op(v)$ [2]
  - $C_v[i] <= C_w[i] + store + C_u[i] + load + op(v)$ [3]
  - $C_v[i] = \min\{ [1], [2], [3] \}$

Carleton UNIVERSITY
Canada's Capital University

Dynamic Programming Example

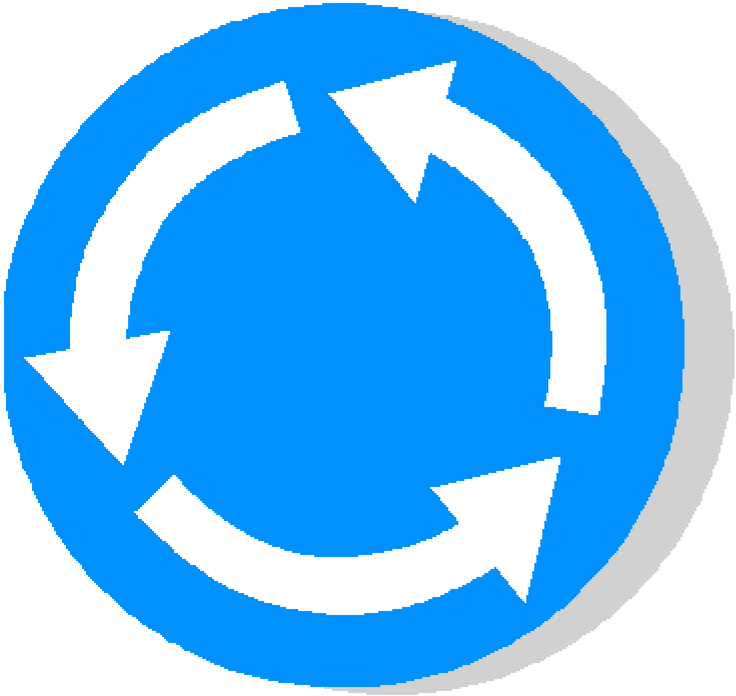
1	2	3



Dynamic Programming Extensions

- For more complicated machines, just make more rules
- Can accommodate several variations:
  - Parse trees whose nodes can have *d* children
    - Just try all *d*! different possible orders
  - Different possible instructions
  - Instructions that allow one (or both) operands to be in memory
  - ....
- Does not make optimal use of common subexpressions
  - In a tree with i subtrees we would have to try something like r^{i+1} combinations

Data-Flow Analysis



Carleton
UNIVERSITY
Canada's Capital University

Data-Flow Analysis

- Data-flow analysis studies execution paths of programs and the evolution of data through these paths
- *Program points* are the spaces between instructions
  - A basic block with k instructions contains k+1 program points
    - one before each instruction
    - one after the last instruction

Carleton UNIVERSITY
Canada's Capital University

Program Paths

- A *program path* $p_1,\ldots,p_t$ is a sequence of program points
  - Within a basic block a program point pi comes before a statement and pi+1 comes after the statement
  - The last program point in a basic block *B* can be followed by the first program point of any basic block that is a successor of *B*.
- We want to reason about the state of the program at program points

Carleton UNIVERSITY
Canada's Capital University

Program Points

```
  -- p1 --
mov a, R0
  -- p2 --
mov b, R1
  -- p3 --
add R0, R1
  -- p4 --
mov c, R1
  -- p5 --
mul R0, R1
  -- p6 --
mov d, R1
  -- p7 --
div R0, R1
  -- p8 --
```

Carleton UNIVERSITY
Canada's Capital University

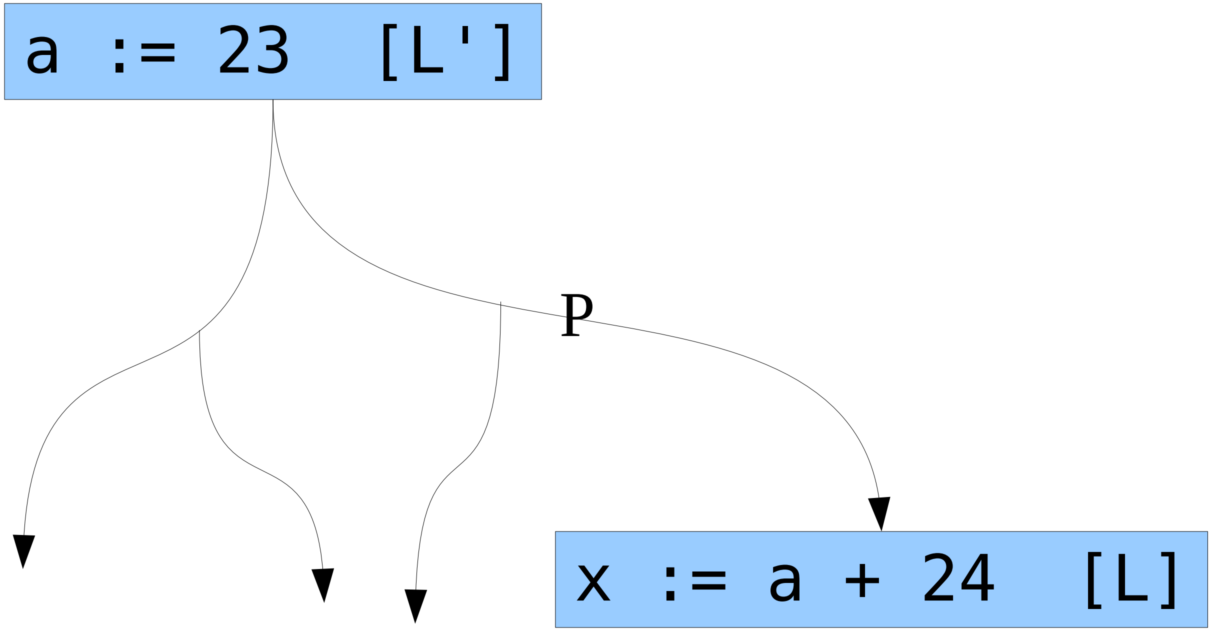
Example: Reaching Definitions

- For a variable *a* used in an instruction *L*
  - An instruction *L*' is a *reaching definition* of *a* at *L* if
    - *L*' sets the value of *a*
    - there is a program path *P* from the point after *L*' to the point before *L* and
    - *P* contains no statement that kills (redefines) *a*
- Reaching definitions can be very useful
  - In debugging, if *a* takes on an incorrect value, we would like to know where this could have happened
  - In optimization if *L* computes an expression with *a* then knowing about *a* may simplify this expression
- Notice: different applications require different information

Carleton UNIVERSITY
Canada's Capital University

Reaching Definitions

- The definition of a at L' reaches L



Data-Flow Schema

- With each program point we associate a data-flow value
  - represents all possible program states at that program point
- For an instruction L
  - in[L] is the data-flow value at the point before L
  - out[L] is the data-flow value at the point after L
- For a block B
  - in[B] is data-flow value before B's first instruction
  - out[B] is data-flow value after B's last instruction
- To speed things up, we sometimes only specify in and out for the basic blocks

Carleton UNIVERSITY
Canada's Capital University

Control-Flow

- Within a basic block with two consecutive statements L1 and L2, we have
  – in[L2] = out[L1]
- The first line of a basic block B is more complicated
  – in[L] = function(out[L1], out[L2], ..., out[Lk])
  – L1,...,Lk are the last instructions in the basic blocks B1,...,Bk that have B as a successor

Carleton UNIVERSITY
Canada's Capital University

Example: Reaching Definitions

- At each point, we want to know all reaching definitions of variable *a*
- Within a basic block
  - out[L] = in[L] if L does not define *a*
  - out[L] = L if L does define *a*
- For the first line L of a basic block B that follows blocks B1,…,Bk
  - in[L] = union(out[B1],…, out[Bk]);

Carleton UNIVERSITY
Canada's Capital University

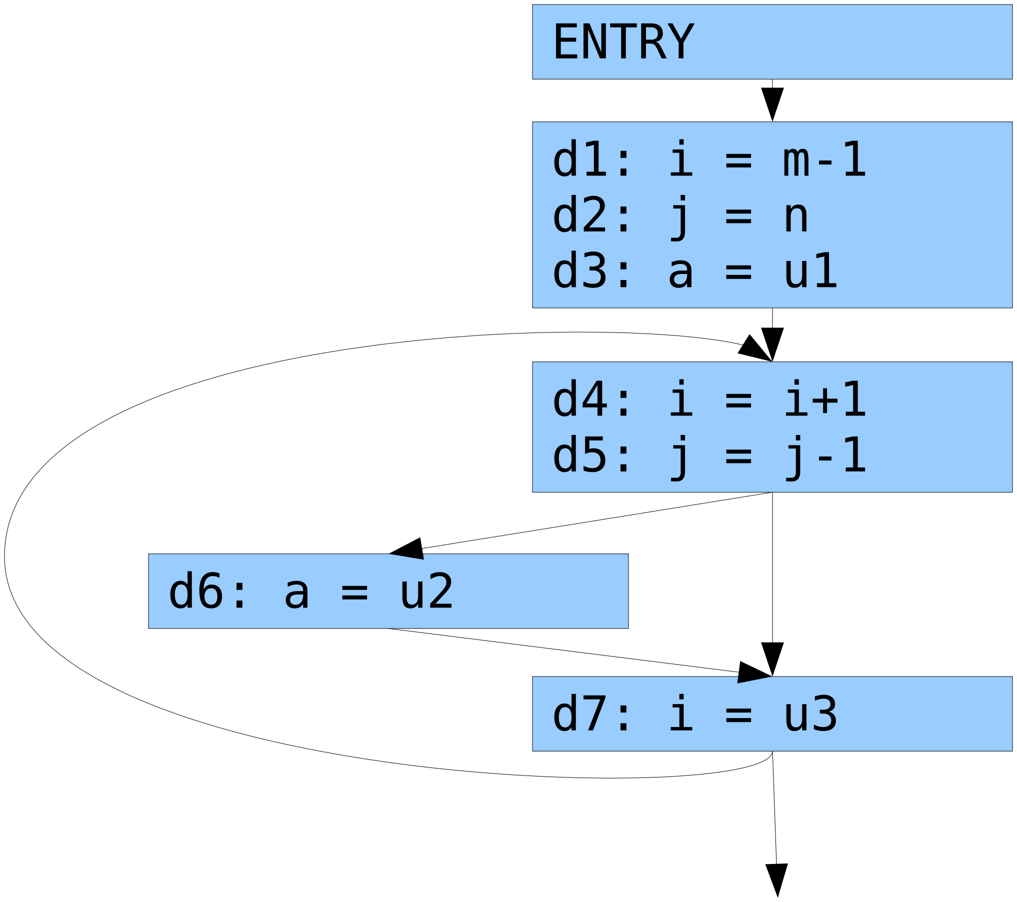
Computing Reaching Definitions

- We now have a set of equations for reaching definitions
  - How do we solve them?
- Iterative algorithm:
  1. initialize in[B] = out[B] = ∅ for every block B
  2. repeat
     1. for each block B with predecessors B1,...,Bk
        1. in[B] = union(out[B1],...,out[Bk])
        2. process each line of B using the equations
           1. out[L] = in[L] if L does not define *a*
           2. out[L] = L if L does define *a*
  3. until out[B] does not change for any block B

Carleton UNIVERSITY
Canada's Capital University

Reaching Definitions Example

- Compute reaching definitions of *a*



Applications of Reaching Definitions

- All kinds of optimizations
  - In a statement that uses variable a, if only there is only one reaching definition of a (or all reaching definitions agree) then we may be able to
    - use reduction in strength
      - b*a = b*2 = b+b
    - use algebraic simplification
      - b*a = b*1 = b
    - convert a conditional into an unconditional jump
      - if a then goto Li = if 1 then goto Li = goto Li
    - perform constant folding
      - 2 + a = 2 + 2 = 4

Carleton UNIVERSITY
Canada's Capital University

Undefined Variables

- Reaching definitions can also be used to check if the value of *a* is defined before it is used
  - Place a "fake" definition at line -1 (entry)
  - If this definition reaches any use of *a* then *a* is potentially used before it is defined
- Useful for catching programmer errors

Carleton UNIVERSITY
Canada's Capital University

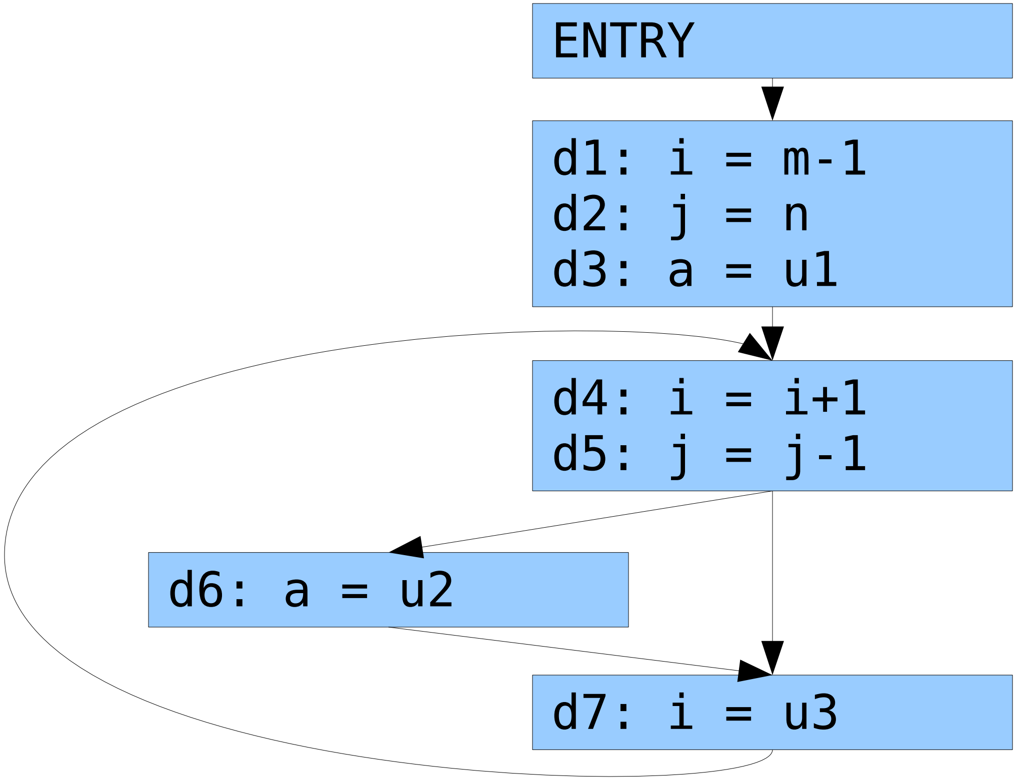
Live-Variable Analysis

- For each point *p* we would like to know if the value of variable *a* at *p* is ever used
- We use backward data-flow
  - in[L] = true if L uses *a*
  - in[L] = false if L defines but does not use *a*
  - in[L] = out[L] if L does not define or use *a*
- For a block B with successors B1,...,Bk
  - out[B] = OR(in[B1],...,in[Bk])
- in[entry] = 0

Carleton UNIVERSITY
Canada's Capital University

Live-Variable Analysis Example

- Determine where variables i and j are live



Applications of Live-Variable Analysis

- Live-variable analysis is used in code generation for basic blocks:
  - dead variables don't need their values stored
  - a dead variable in a register should be overwritten before a live variable

Carleton UNIVERSITY
Canada's Capital University

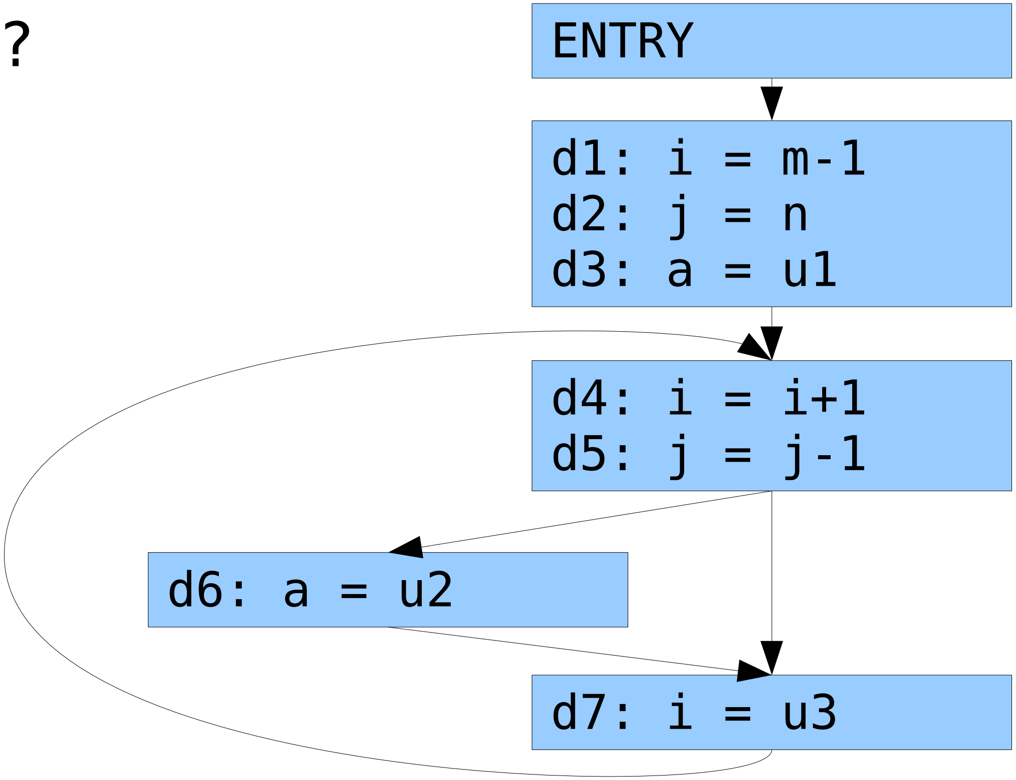
Available Expressions

- x+y is *available* at a point p if
  - every path from entry to p evaluates x+y
  - no path changes the values of x or y after the evaluation
- For an instruction L,
  - out[L] = true if L computes x+y
  - out[L] = false if L modifies x or y
  - out[L] = in[L] otherwise
- For a block B with predecessors B1,…,Bk,
  - in[B] = AND(out[B1],…,out[Bk])
- Initially, out[B] = true for every block B except the entry block out[entry] = false

Carleton UNIVERSITY
Canada's Capital University

Available Expressions

- Where are the expressions m-1 and i+1 available?



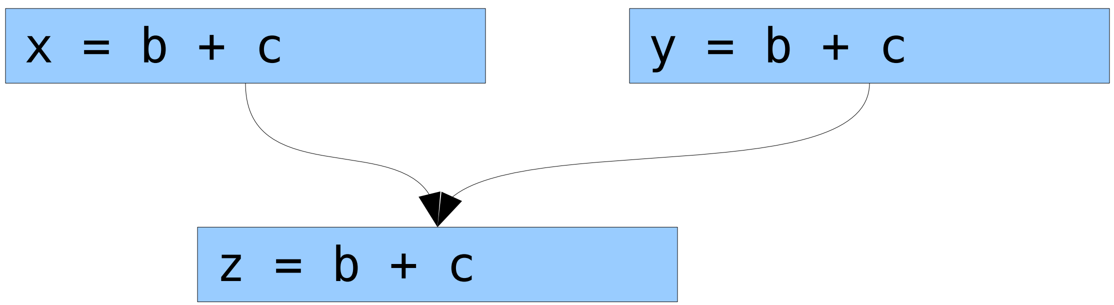
Carleton UNIVERSITY
Canada's Capital University

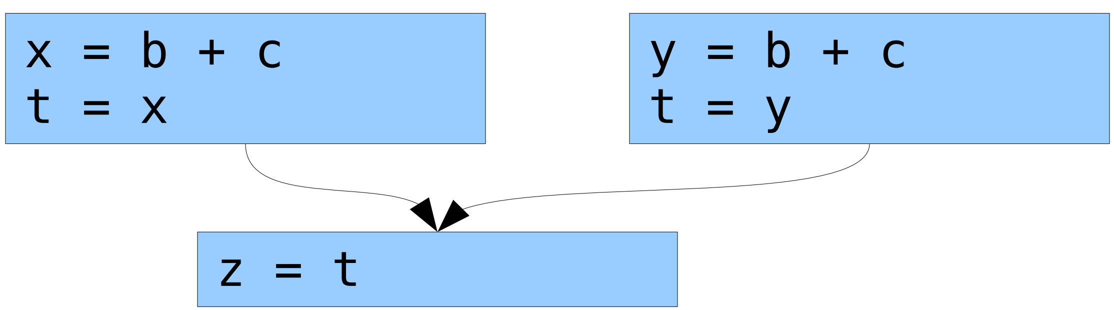
Partial Redundancy Elimination

- Redundancy occurs when the same expression is evaluated more than once (with the same input values) along an execution path
- If an expression computed at instruction L is available then it is (fully) redundant
- If an expression computed at instruction L might be available then is (partially) redundant

Fully-Redundant Expressions

- Fully redundant expressions can be stored and used (sometimes the store is unnecessary)

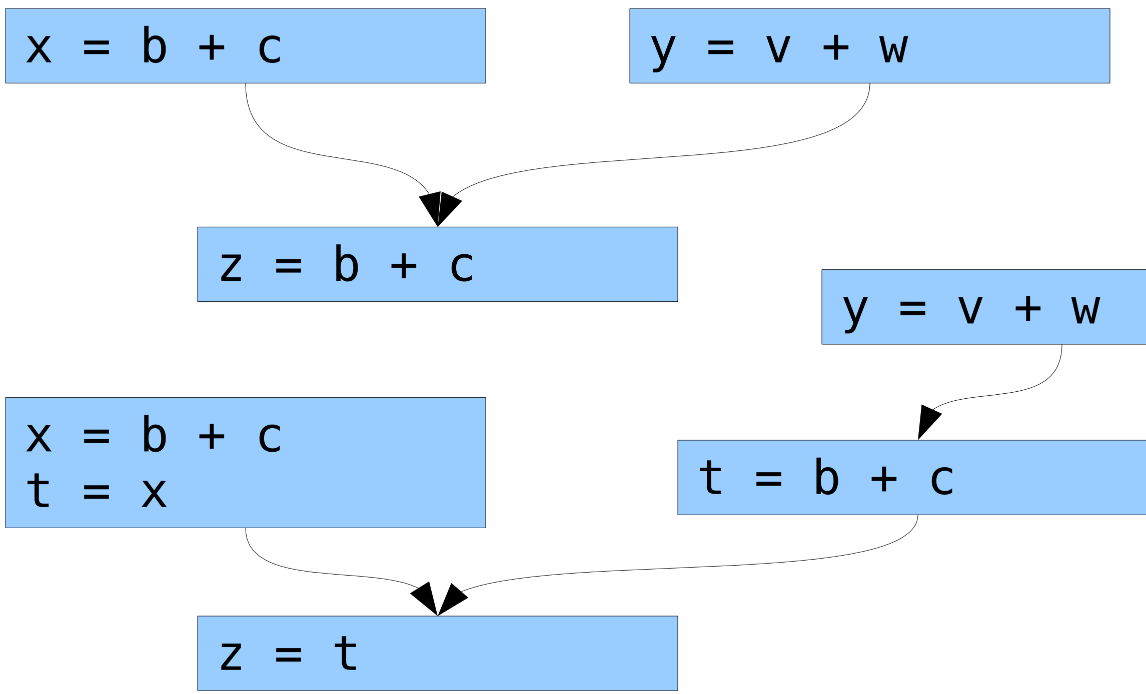




Carleton UNIVERSITY
Canada's Capital University

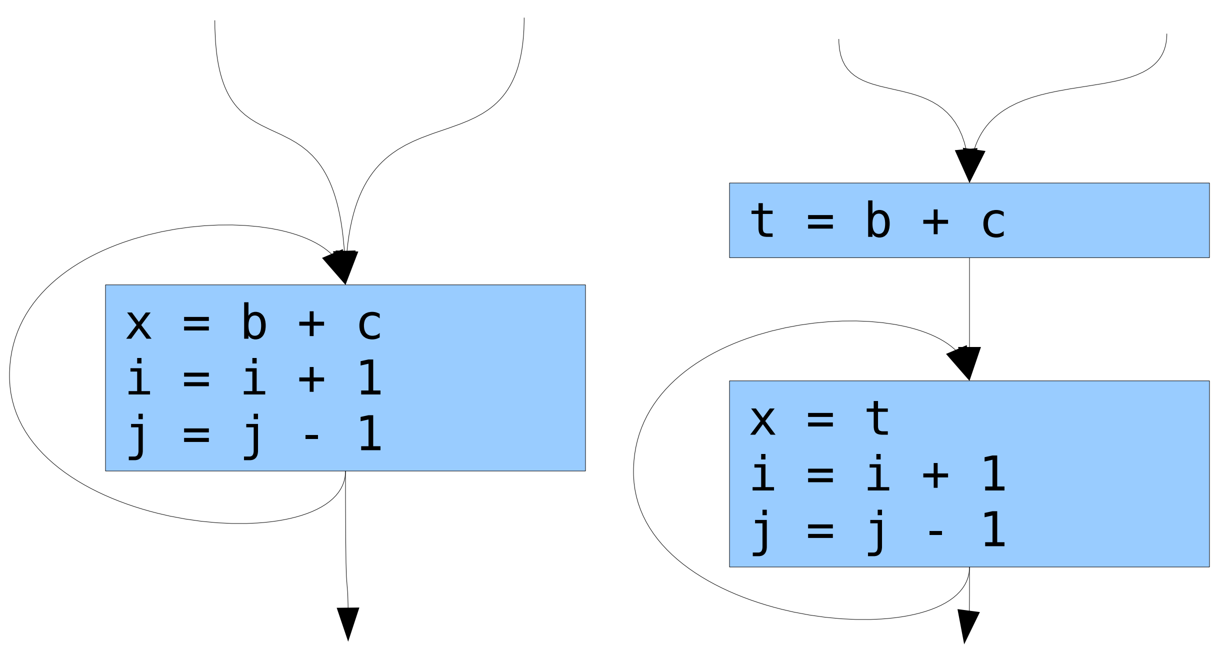
Partially-Redundant Expressions

- If an expression is available in some (but not all) predecessors B then it is partially redundant (see Section 9.5.5)



Loop-Invariant Expressions

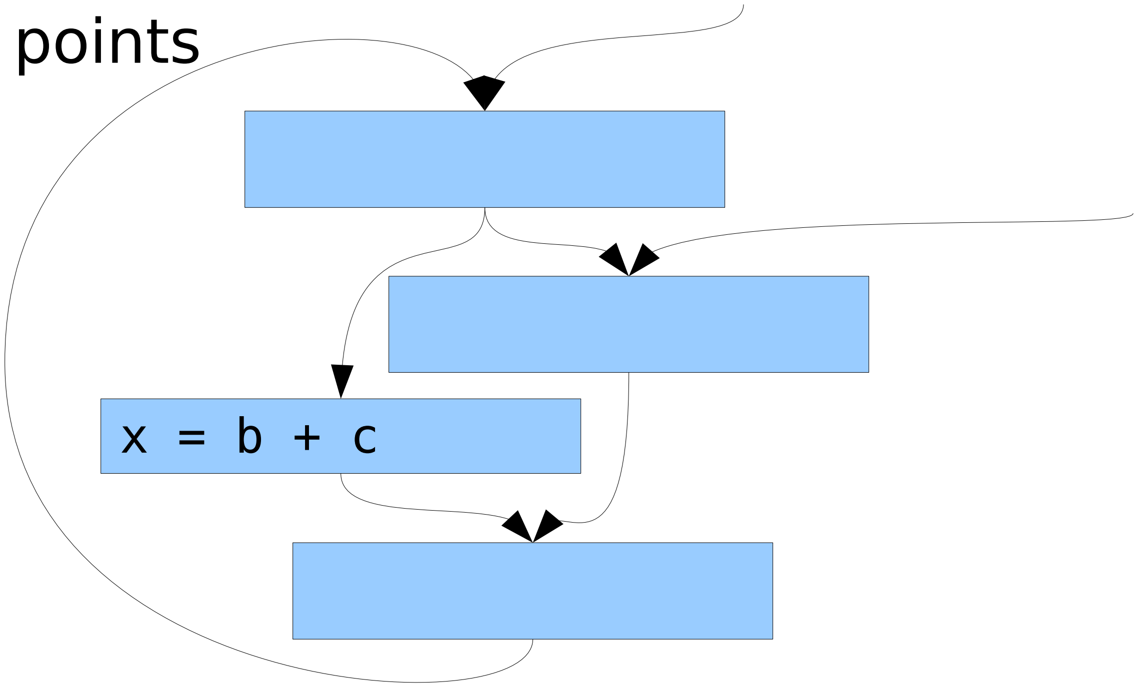
- The expression b+c is loop invariant if neither b nor c is redefined within the loop
  - All reaching definitions of b and c are outside the loop



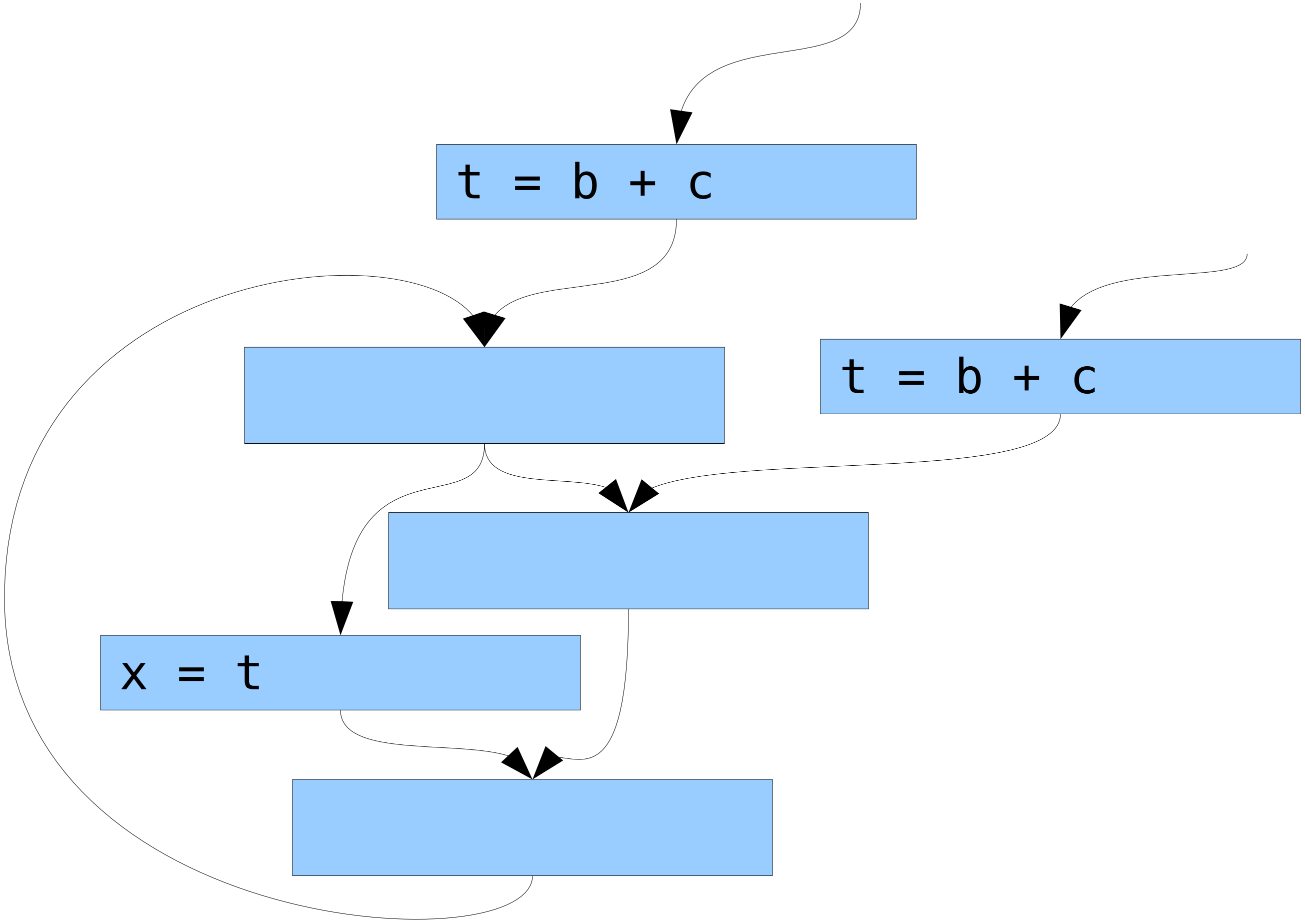
Carleton UNIVERSITY
Canada's Capital University

Loop-Invariant Expressions

- Note: A loop is any strongly connected component of the flow graph
- We have to be careful to cover all loop entry points



Loop-Invariant Expressions



Summary of Data Flow Analysis

- Data-flow analysis lets the compiler reason about program state at various points in time
  - Can check reaching definitions, live variables, and available expressions (among others)
  - Has real theoretical underpinnings (see Sec. 9.3)
- For live variables and available expressions:
  - we chose in[L] and out[L] are in the set {true, false} and we combine them with AND and OR
  - to compute all live variables or all available subexpressions in and out can be sets that are combined with intersection and union

Carleton UNIVERSITY
Canada's Capital University

Neat Application

- Smartphones apps have access to
  - Personal information
  - Network
- We want to avoid personal information being sent over the network
  - Define 'tainted variables'
    - Any variable from a syscall that retrieves personal information
    - taint spreads to other variables (and files) by dataflow analysis
  - No network routine should be given a tainted variable as an argument

Carleton UNIVERSITY
Canada's Capital University

What We Didn't See

- Speed of convergence of iterative algorithm
  - Using depth-first order makes the algorithm more efficient
  - Number of iterations is at most the length of the longest path in the flow graph
- Loop analysis and dominance
- Induction variables
- Theoretical foundations
  - Abstraction, monotone frameworks, and distributive frameworks, semilattices

Carleton UNIVERSITY
Canada's Capital University