MC910_MO615 – Exercises 2s2013

Question 1 (Dataflow Analysis): To deal with security issues, several programming languages have a notion of "tainted" data. The idea is that any value read from the outside environment is marked as being tainted, i.e., potentially dangerous. The results of any operations that use tainted data are also marked tainted. There is also a way of marking a value as "not-tainted", presumably to be used only after verifying that it is "safe", whatever that may mean. For this problem, assume that the available operations in our intermediate language are:

$x = y \oplus z$	binary operation: x is tainted if either y or z or both are tainted
$x = y$	assignment: x is tainted if y is tainted
x = read()	input: result x is tainted
x = clean(y)	clean: assign y to x, but mark x as not tainted

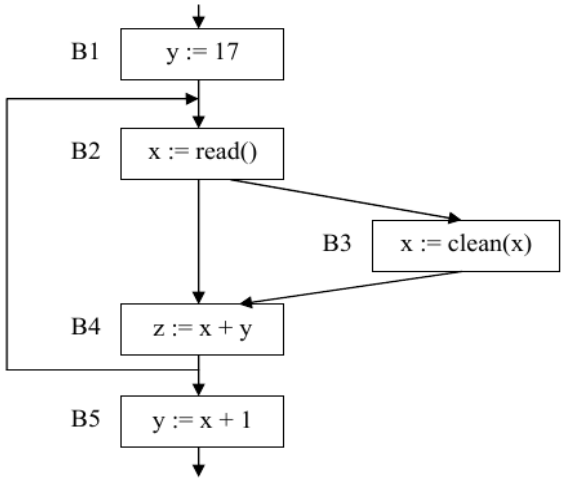
Now we would like to use dataflow analysis on this low-level code to discover tainted variables. A variable that might contain a tainted value is marked as tainted. Only if we know that the value is guaranteed not to be tainted do we mark it so. To discover tainted variables we define the following sets for each basic block b:

$IN_{[b]}$ =	set of all possibly tainted variables on entry to block b
$OUT_{[b]}$ =	set of all possibly tainted variable on exit from block b
$GEN_{[b]}$ =	set of all variables marked tainted in block b and not cleaned before exit
$CLEAN_{[b]}$=	set of all variables cleaned in block b and not later tainted before exit

The sets $GEN_{[b]}$ and $CLEAN_{[b]}$ can be computed once based on the static contents of each block b. The $IN_{[b]}$ and $OUT_{[b]}$ sets need to be computed iteratively during the analysis.

(a) Give appropriate dataflow equations for the IN and OUT sets for a block b in terms of the IN, OUT, GEN, and CLEAN sets. As usual, these equations will involve some combination of local information about the block itself as well as information about the block's predecessors and successors.

(b) Now consider the following flowgraph:



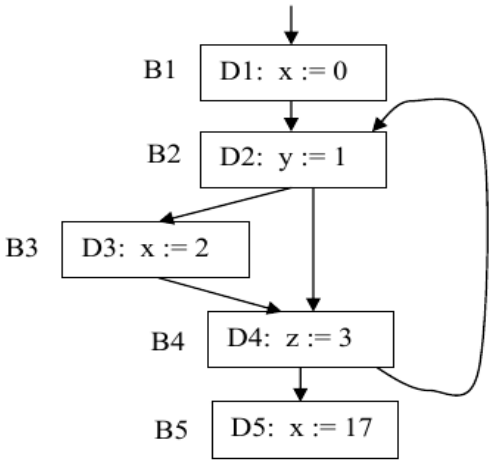
Complete the following table using iterative dataflow analysis to identify the tainted variables in the IN and OUT sets for each block in the above graph. You should first fill in the GEN and CLEAN entries for each block, then iteratively solve for IN and OUT. Choose whichever direction (forward or backward) you wish to solve the equations. You should assume that there are no tainted variables in the IN set for block B1.

Block	GEN	CLEAN	IN 1	OUT 1	IN 2	OUT 2	IN 3	OUT 3
B1								
B2								
B3								
B4								
B5								

Question 2 (Dataflow Analysis) The dataflow problem for reaching definitions can be defined as follows:

$GEN_{[b]}$ = { set containing definition d from block b, if any }
$KILL_{[b]}$ = { all other definitions that assign to a variable that is defined in b }
$IN_{[b]} = \bigcup_{p \in pred[b]} OUT_{[p]}$
$OUT_{[b]} = GEN_{[b]} \cup (IN_{[b]} - KILL_{[b]})$

Compute the reaching definitions for the nodes in the following flowgraph. You should first fill in GEN and KILL in the table below for each block, then iteratively solve for the IN and OUT sets. Choose whichever direction to solve for IN and OUT that you wish (forward or backward).

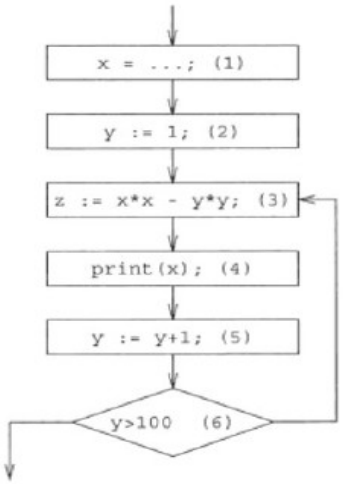


Block	GEN	CLEAN	IN 1	OUT 1	IN 2	OUT 2	IN 3	OUT 3
B1								
B2								
B3								
B4								
B5								

Question 3 (Dataflow Analysis) An optimization technique called “code hoisting” moves expressions to the earliest point beyond which they would always be evaluated. An expression that is always evaluated beyond a given point is called very busy at that point. Once it is known at which points an expression is very busy, the evaluation of that expression can be moved to the earliest of those points. Determining these point is a backwards dataflow problem.

(a) Give the general dataflow equations to determine the points at which an expression is very busy

(b) Consider the following CFG. Give the KILL and GEN for the expression x * x



(c) Solve the dataflow equations for the expression x * x. What optimization becomes possible?

Question 4 (Instruction Selection) The Jouette machine has control-flow instructions as follows:

BRANCHGE	if ri ≥ 0 goto L
BRANCHLT	if ri < 0 goto L
BRANCHEQ	if ri = 0 goto L
BRANCHNE	if ri ≠ 0 goto L
JUMP	goto ri

where the JUMP instruction goes to an address contained in a register. Use these instructions to implement the following tree patterns:

Assume that a CJUMP is always followed by its false label. Show the best way to implement each pattern; in some cases you may need to use more than one instruction or make up a new temporary.

Question 5 (Instruction Selection) Consider the variable *x* and the vector *a* as being stored in the stack and *i* into a register. Considering the sentence below:

$a[i] = x + 1$

(a) Draw the tree intermediate representation from the specification of the instruction set of Jouette architecture shown in the table below.

(b) Make coverage patterns for Jouette architecture using the algorithm Maximal Munch.

(c) Show the sequence of instructions that will be issued in the order they should be performed, using the syntax assembly of Jouette machine given in the table below.

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

Jouette-machine instructions

Question 6 (Machine-Independent Optimizations) For the flow graph in Figure below:

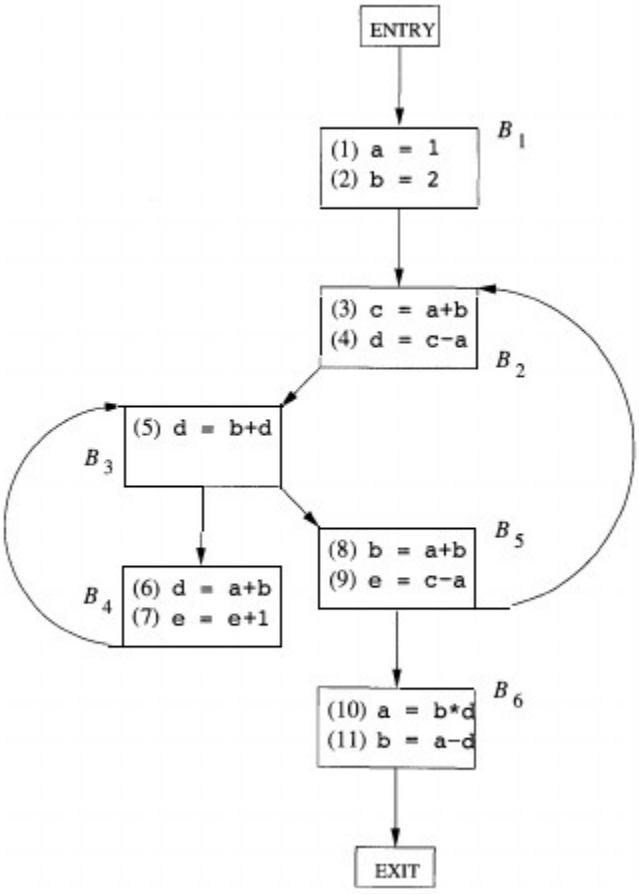
a) Identify the loops of the flow graph;

b) Statements (1) and (2) in B_1 are both copy statements, in which *a* and *b* are given constant values. For which uses of *a* and *b* can we perform copy propagation and replace these uses of variables by uses of a constant? Do so, wherever possible;

c) Identify any global common subexpressions for each loop;

d) Identify any induction variables for each loop. Be sure to take into account any constants introduced in (b);

e) Identify any loop-invariant computations for each loop.


ENTRY
(1) a = 1
(2) b = 2
B1
(3) c = a+b
(4) d = c-a
B2
(5) d = b+d
B3
(8) b = a+b
(9) e = c-a
B5
(6) d = a+b
(7) e = e+1
B4
(10) a = b*d
(11) b = a-d
B6
EXIT

Flow Graph