Compiler Design

Fall 2015

Data-Flow Analysis

Sample Exercises and Solutions

Prof. Pedro C. Diniz

USC / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
pedro@isi.edu

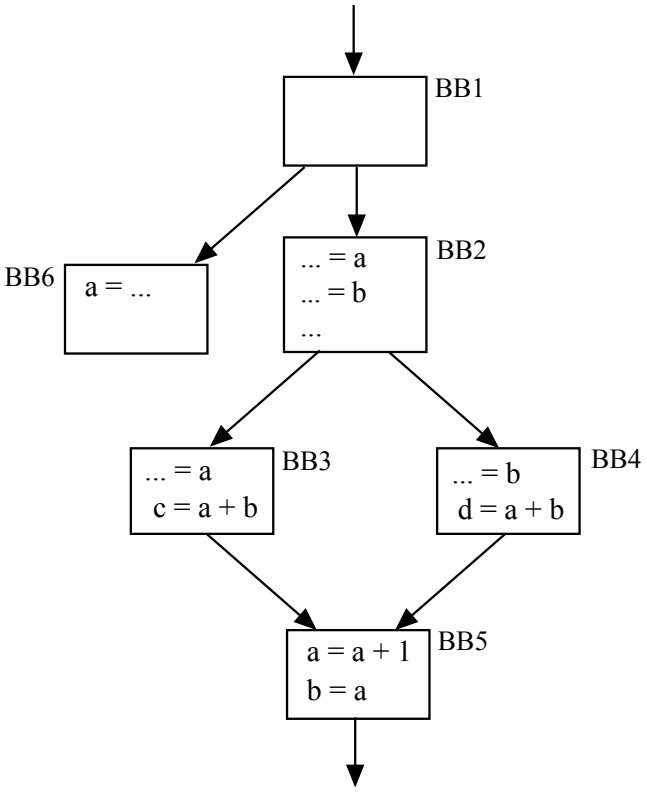
Problem 1:

Your goal is to solve the Anticipation Analysis data-flow problem. The idea is to understand how early one could compute an expression in the program before the expression needs to be used.

Definition: *We say an expression e is anticipated at point p if the same expression, computing the same value, occurs after p in every possible execution path starting at p.*

This is a necessary condition for inserting a calculation at p, although it is not sufficient, since the insertion may not be profitable. Your data-flow analysis must determine whether a particular expression `a + b` in the program is anticipated at the entry of each basic block. (This can easily be generalized to the analysis of anticipation for every expression in the program).

For the example in the CFG below the expression `a+b` is anticipated in the beginning of the basic block BB2 but not at the beginning of basic block BB1 since in the later case there is a control path in which the value of the expression changes due to the assignment in basic block BB6.



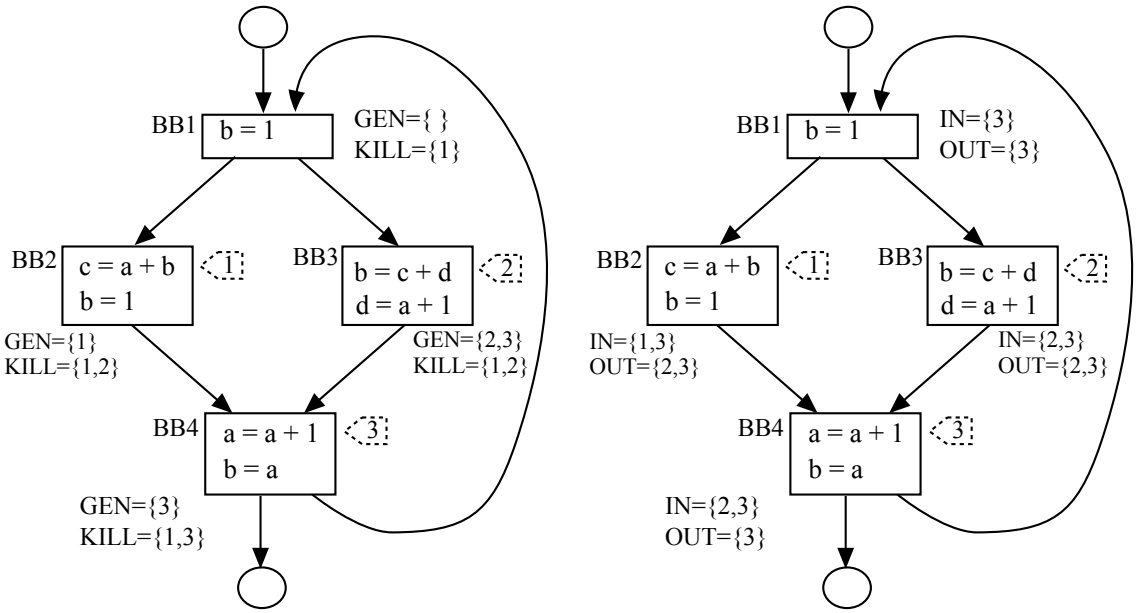
Describe your approach to anticipation analysis by explaining the following:

1. Direction of the problem, backwards or forward and why?
2. Your representation format and the initial values based on the suggested representation?
3. The definition and rationale for the GEN and KILL sets.
4. The equations that the iterative approach needs to solve.

Answers:

1. This is essentially the inverse version of the available expressions data-flow problem. Here we are asked if a given expression is anticipated at a given point *p* of the program, i.e., if for all paths starting at *p* and observe if none of its arguments are redefined. We thus work backwards from the use or creation point of the expression and trace back to see until when are one of its operands redefined. At bifurcation points we will determine if on the other path (along which we did not come) the same expression is also anticipated, i.e. there is a path to a use of the same expression whose operands are not redefined.
2. The lattice of values consists of sets of all the expressions computed in the program, in this case we can number the expressions in the program and use sets of integers to represent which of the expressions at point *p* are anticipated. The initial values for the OUT of all the basic blocks are the universe, i.e., all expressions are initially anticipated.
3. The GEN set for an instruction and also the basic block, define which set of expressions a given instruction generates. The KILL set is the set of all expressions a given assignment statement eliminates and thus uses the LHS variables and kills all the expressions where the LHS variable appears.
4. Given that this is a backwards problem we have to define the IN of each basic block as a function of its OUT. The meet function is the intersection given that for one expression to be anticipated at a given point p it needs to be anticipated at all paths starting at p. To compute the IN of each basic block/instruction we use the equation IN = GEN + (OUT – KILL) where + and – stand for set union and difference respectively.

We illustrate the application of this Data-Flow Analysis to the following program and corresponding CFG where we have omitted the computation of the data-flow solution to all the intermediate program point in each basic block (the so called local phase).



Problem 2:

```
01        a = 1
02        b = 2
03 L0:    c = a + b
04        d = c - a
05        if c < d goto L2
06 L1:    d = b + d
07        if d < 1 goto L3
08 L2:    b = a + b
09        e = c – a
10        if e = 0 goto L0
11        a = b + d
12        b = a – d
13        goto L4
14 L3:    d = a + b
15        e = e + 1
16        goto L3
17L4:    return
```

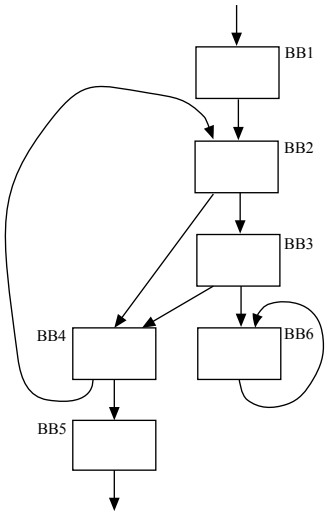
For the code shown above, determine the following:

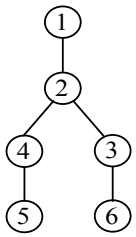
a) The basic blocks of instructions.
b) The control-flow graph (CFG)
c) For each variable, its corresponding *du*-chain.
d) The live variables at the end of each basic block. You do not need to determine the live variables before and after each basic block and justify your answer for the value presented for the basic block containing instructions at line 6 and 7.
e) Is the live variable analysis a forward or backward data-flow analysis problem? Why and what does guarantee its termination when formulated as a data-flow analysis iterative problem?

Answers:

a) b) We indicate the instruction in each basic block and the CFG and dominator tree below.

BB1: {01, 02}
BB2: {03, 04, 05}
BB3: {06, 07}
BB4: {08, 09, 10}
BB5: {11, 12, 13}
BB6: {14, 15, 16}





c) The def-use chains can be obtained by inspection of which definitions are not killed along all path from its definition point to the use points. For example for variable "a" its definition at the instruction 1, denoted by a1 reaches the use points at the instructions 3, 4, 8, 9 and 15, hence the notation {d1, u3, u4, u8, u9, 15}. Definition a1 does not reach use u12 because there is another definition at 11 that masks it, hence another du chain for "a" denoted as {d11, u12}. The complete list is shown below.

a: {d1, u3, u4, u8, u9, 15}
a: {d11, u12}
b: {d2, u3, u4, u6, u14, u8}
b: {d8, u11, u8, u14}
b: {d12}
c: {d3, u4, u5, u9}
d: {d4, u5, u6}
d: {d6, u7}
d: {d14, u6}
e: {d9, u10, u15}
e: {d14, u15}

d) At the end of each basic block we can inspect the code and ask if a given variable is still used after the control flow leaves that basic block. The results of this inspection is depicted below:

BB1: {a, b}
BB2: {a, b, c, d, e}
BB3: {a, b, c, d, e}
BB4: {a, b, d, e}
BB5: { }
BB6: {a, b, c, e}

For BB6 the live variable solution at the exit of this basic block has {a, b, c, e} as for variable "d" there is no path out of this basic block where the current value of the variable is used. For variable "d" the value is immediately redefined in BB3 without any possibility of being used.

e) The live variable analysis is a backward data-flow problem as we propagate the information about a future use of a variable backward to specific points of the program. If there is a definition at a specific point backward the solution kills all other uses and resets the information associated with that variable. As with many other iterative formulations of data-flow analysis problems termination is guaranteed by the fact that the lattice, in this case the set of variables, has finite cardinality or length. The flow-function, in this case set-union is monotonic.

Problem 3:

In this problem you will develop and show the application of the *Def-Use* or *Reaching Definitions* analysis problem to the code of a given procedure. The Reaching Definitions Data-Flow Analysis problem seeks to determine for the use of each scalar variable (or temporary register), which are the definitions of that same variable that can reach it. This information is crucial in defining possible constant assignment and is also the basis for the SSA intermediate representation format.

Formally, a definition *d* for a variable *v* at program point *p* to reach an execution point *q* there must exists an execution path from *p* to *q* along which the value of *v* is not written.

a. [10 points] Formalize the reaching definition problem as an iterative data-flow analysis problem showing the equations for the *gen* and *kill* abstractions as well as the initialization of the values for each basic block and statement. In this section you need to determine if this is a forward or backward data flow problem.
b. [30 points] Apply your data flow problem formalization to the code depicted in problem 1 showing the final result of the IN and OUT sets of reaching definitions for each basic block in the code (for this you need to find out the basic blocks in problem 1 first). Show the application of the flow equations inside each of the basic blocks as well as the local propagation after the data-flow solution at the CFG is done. You thus need to first compute the aggregate basic-block functions; solve the data-flow problem at the CFG Basic Block level and then propagate the solution found for each basic block to each of the individual basic block instructions. Be clear about the initialization of the data-flow problem.
c. [20 points] Use the information from b.) to perform constant propagation and loop invariant detection as well as to support the conditions for induction variable recognition by looking at the values of the various definitions that reach a given use of the variable.

Solution:

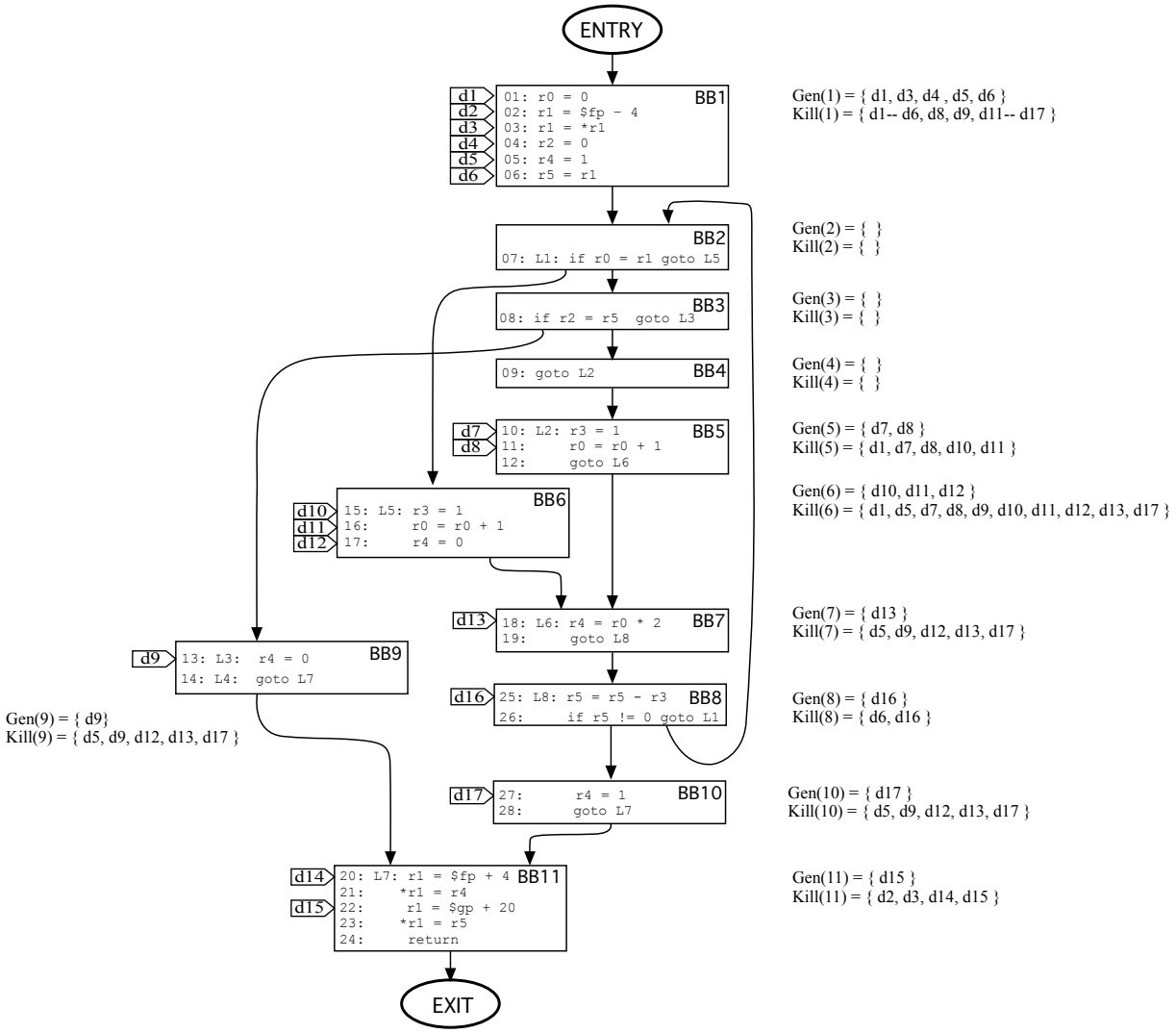
a) [10 points] A possible formulation for this problem is:

Domain: Set of all variable definitions numbered d_1 through d_k where k is number of assignments.
Direction: Forwards as the flow functions define the output of a basic block with respects to its input.
Initial values: Empty set for all Out sets and In set of entry node of CFG is also empty.
Functions:
Gen(n) = { d_i | if the definition d_i is in the basic block n}, i.e. the downward-exposed definitions in n.
Kill(n) = { all definitions d_i | for variable v_k assigned in n d_i is also assigns v_k in another BB }
Flow: $Out(n) = Gen(n) \cup (In(n) - Kill(n))$
Meet: $In(n) = \cup_{s \text{ in } pred(n)} Out(s)$,

b) [30 points] Assuming all basic blocks n are initialized with Out(n) as the empty set at the first iteration we compute the In(n) and Out(n) sets for all basic blocks until a fixed-point is reached. We use a topological sorting order of the CFG block (thus ignoring the back edges) for the application of the flow and meet function, in the following sorting order: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}. The table below depicts the evolution of the values for the In(n) and Out(n) for each basic block and the figure below depicts the labels associated with each statement and each variable along with the Gen and Kill sets of each basic block.

	Iter.	BB1	BB2	BB3	BB4	BB5	BB6	BB7	BB8	BB9	BB10	BB11
Gen		1,3,4,5,6	∅	∅	∅	7,8	10,11,12	13	16	9	17	5
Kill		1,2,3,4,5,6,8,9, 11,12,13,14,15,1617	∅	∅	∅	1,7,8,10,11	1,5,7,8,9,10, 11,12,13,17	5,9,12,13,17	6,16	5,9,12,13,17	5,9,12,13,17	2,3,14,15
In	0	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
Out		∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
In	1	∅	1,3,4,5,6	1,3,4,5,6	1,3,4,5,6	1,3,4,5,6	1,3,4,5,6	3,4,5,6,7,8, 10, 11,12	3,4,6,7,8, 10,11,13	1,3,4,5,6	3,4,7,8,10, 11,13,16	1,3,4,6,7,8,9, 10,11,16,17
Out		1,3,4,5,6	1,3,4,5,6	1,3,4,5,6	1,3,4,5,6	3,4,5,6,7,8	3,4,6,10,11,12	3,4,6,7,8,10,11, 13	3,4,7,8,10, 11,13,16	1,3,4,6,9	3,4,7,8,10, 11,16,17	1,4,6,7,8,9,10, 11,15,16,17
In	2	∅	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8 ,10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	3,4,5,6,7,8,10, 11,12,13,16	3,4,6,7,8,10, 11,13,16	1,3,4,5,6,7,8, 10,11,13,16	3,4,7,8,10 11,13,16	1,3,4,6,7,8,9, 10,11,16,17
Out		1,3,4,5,6	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	3,4,5,6,7,8, 13,16	3,4,6,10,11,12, 16	3,4,6,7,8,10,11, 13,16	3,4,7,8,10, 11,13,16	1,3,4,6,7,8,9, 10,11,16	3,4,7,8,10, 11,16,17	1,4,6,7,8,9,10, 11,15,16,17
In	3	∅	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8 ,10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	3,4,5,6,7,8,10, 11,13,16	3,4,6,7,8,10, 11,13,16	1,3,4,5,6,7,8, 10,11,13,16	3,4,7,8,10, 11,13,16	1,3,4,6,7,8,9, 10,11,16,17
Out		1,3,4,5,6	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	1,3,4,5,6,7,8, 10,11,13,16	3,4,5,6,7,8, 13,16	3,4,6,10,11, 13, 16	3,4,6,7,8,10, 11, 13,16	3,4,7,8,10, 11,13,16	1,3,4,6,7,8,9, 10,11,16	3,4,7,8,10, 11,16,17	1,4,6,7,8,9,10, 11,15,16,17

The table above shows the values for the Gen and Kill sets for each of the blocks. These abstract values already have into account the control-flow inside each of the basic blocks. For instance, and regarding the variable r1 in BB1 only the second definition d3 reaches the end of that basic block as *d2* is masked by *d3* and so only *d3* is downwards-exposed as that is the meaning of the Gen set in this particular data-flow problem.



To propagate the values of the In sets inside each basic block, we need to apply the same data-flow equations, but now at the granularity of the instruction. As an interesting example, in basic block BB8 the set of definitions that reach the statement in line 26 is In(8), excluding all the definitions of

the variables r5 but then including definition *d16*. Given that In(8) = { d3, d4, d6, d7, d8, d10, d11, d13, d16}, the definitions that reach line 26 are In(8) - { d6 , d16} + { d16 } = { d3, d4, d7, d8, d10, d11, d13, d16 }

c) [30 points] The results of the Reaching-Definitions data-Flow Analysis problem yield for basic block BB8 the solution at its input as: In(8) = { d3, d4, d6, d7, d8, d10, d11, d13, d16}. For the variables r3 we the two reaching definitions are {d7, d10}. Based on this information, a compiler could determine that in fact both these definitions are compile-time constants, in this case identical and assuming the value 1. Given that the head of the loop dominates both the basic blocks (BB5 and BB6) where these definitions are made, and BB8 post-dominates the two basic blocks BB5 and BB6 then it is correct to replace the use of r3 by the value 1. In addition both definitions would then become dead code as there would be no uses of r3 in the code and the two definition statements in lines 10 and 15 and the variable r3 itself could simply be removed.