Compilers

Spring 2023

Data-Flow Analysis and Its Applications

Sample Exercises and Solutions

Prof. Pedro C. Diniz

Informatics Engineering Department
Faculty of Engineering of the University of Porto
pedrodiniz@fe.up.pt

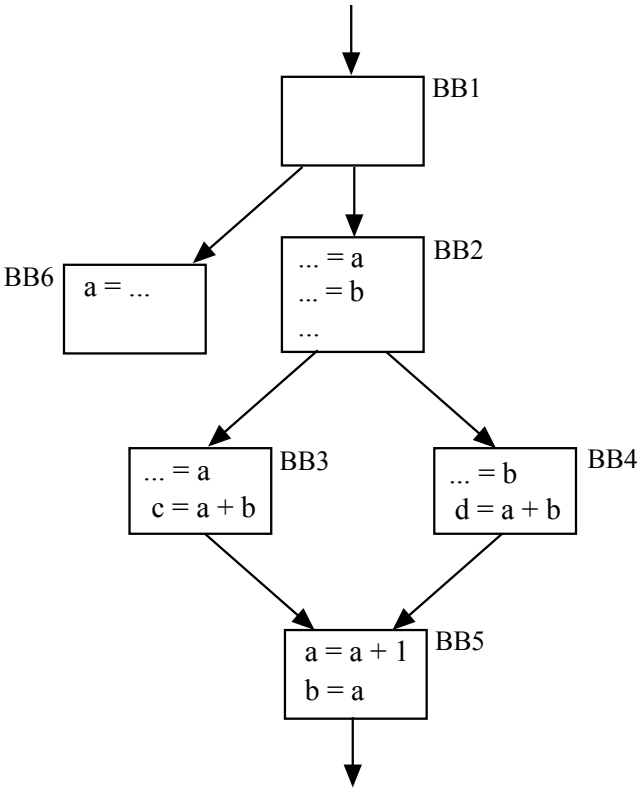
Problem 1

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.



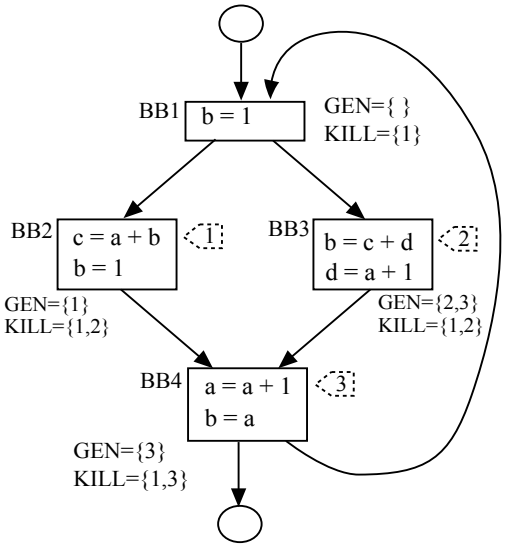
Describe your approach to anticipation analysis by explaining the following:

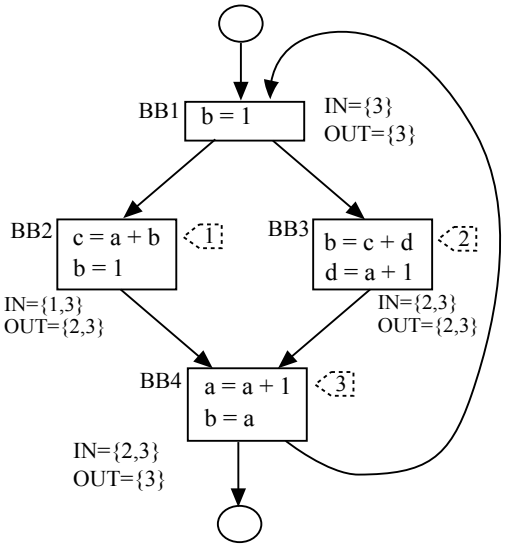
1. Direction of the problem, backwards or forward and why?
2. Your representation format and the initial values based on the suggested representation?
3. The definition and rationale for the GEN and KILL sets.
4. The equations that the iterative approach needs to solve.

Solution:

1. This is essentially the inverse version of the Available Expressions data-flow analysis 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

Consider the code shown below, in three-address instruction format.

```
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
17 L4:     return
```

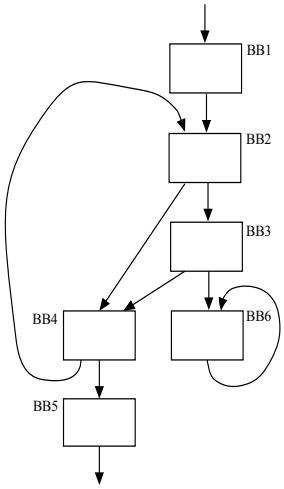
For the code shown above, determine the following:

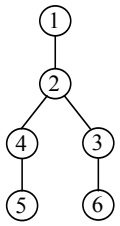
a) The basic blocks of instructions.
b) The control-flow graph (CFG)
c) For each variable, its corresponding *du*-chain.
d) The live variables at the end of each basic block. You do not need to determine the live variables before and after each basic block and justify your answer for the value presented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

Solution:

a) b) We indicate the instructions in each basic block, the CFG and the 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}
BB7: {17}
```





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

```
a: {d01, u03, u04, u08, u09, u14}, {d11, u12}
b: {d02, u03, u06, u14, u8},{d08, u11, u03, u06, u08, u14}, {d12}
c: {d03, u04, u05, u09}
d: {d04, u05, u06},{d06, u07, u11, u12},{d14}
e: {d09, u10, u15},{d14, u15, u14}
```

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 are 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, e}
```

For BB6 the live variable solution at the exit of this basic block has {a, b, e} as for variable "d" and "c" 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. It is a bit of an extreme case since `BB6` is an infinite-loop and hence dead code.

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. 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. 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. 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:

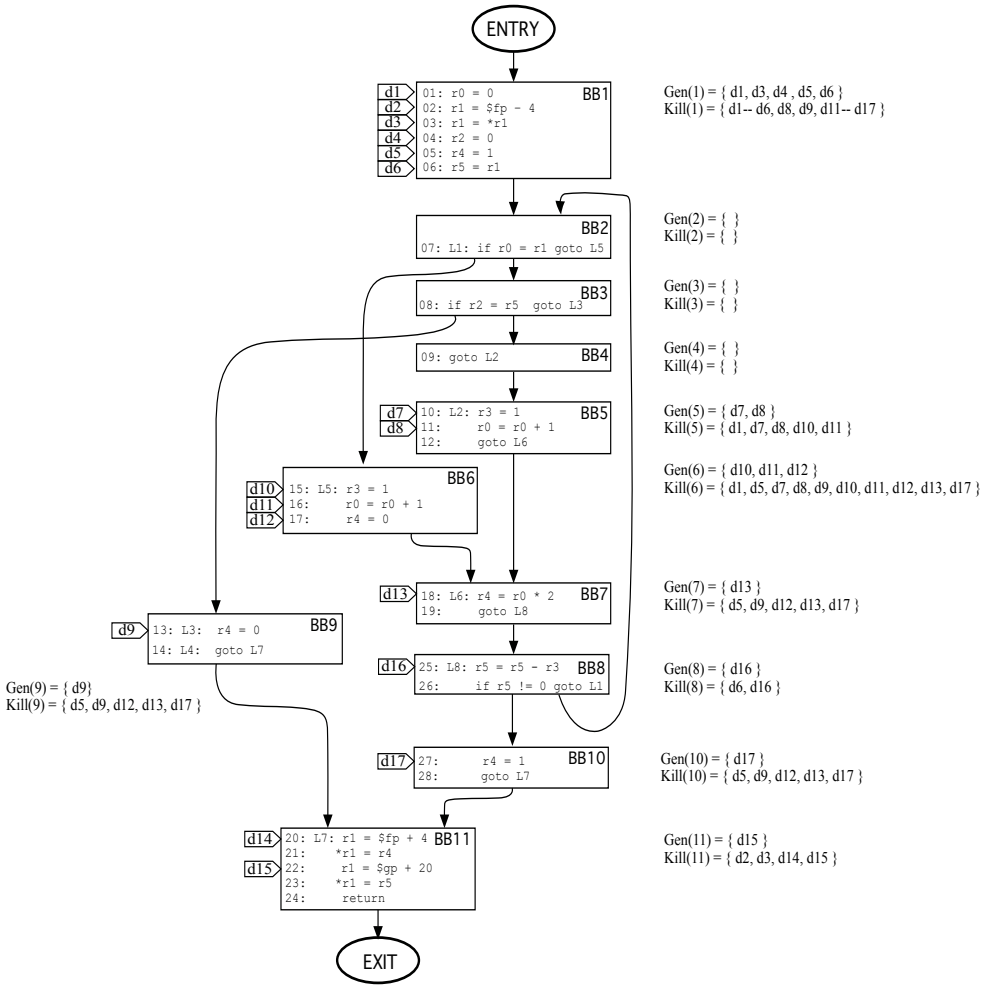
a) A possible formulation for this problem is:

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) 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) 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.

Problem 4

```
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 L1
17 L4:  return
```

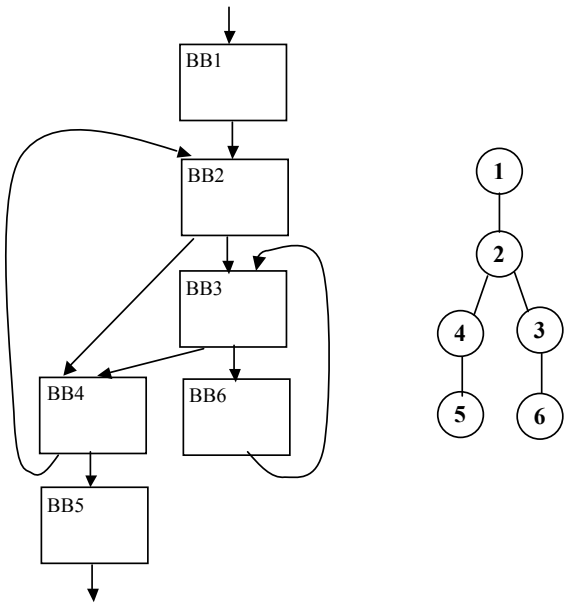
For the code shown above, determine the following:

a) The basic blocks of instructions and the control-flow graph (CFG).
b) 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 but justify your answer for the value presented for the basic block containing instructions at line 6 and 7.
c) Is the Live-Variables 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?

Solution:

a) We indicate the instructions in each basic block and the CFG and dominator tree on the RHS.

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



b) 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" definition at the instruction 1, denoted by a1 reaches the use points at the instructions 3, 4, 8, 9 and 14, hence the notation {d1, u3, u4, u8, u9, u14}. 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, u14}
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}
e: {d14, u6}
```

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 are 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, d, e}
```

For BB6 the live variable solution at the exit of this basic block has {a, b, c, d, e} as there exists a path after BB6 where all the variables are being used. For example, variable "e" is used in the basic block following the L1 label and taking the jump to L3 to BB6 again where is used before being redefined.

c) 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 5

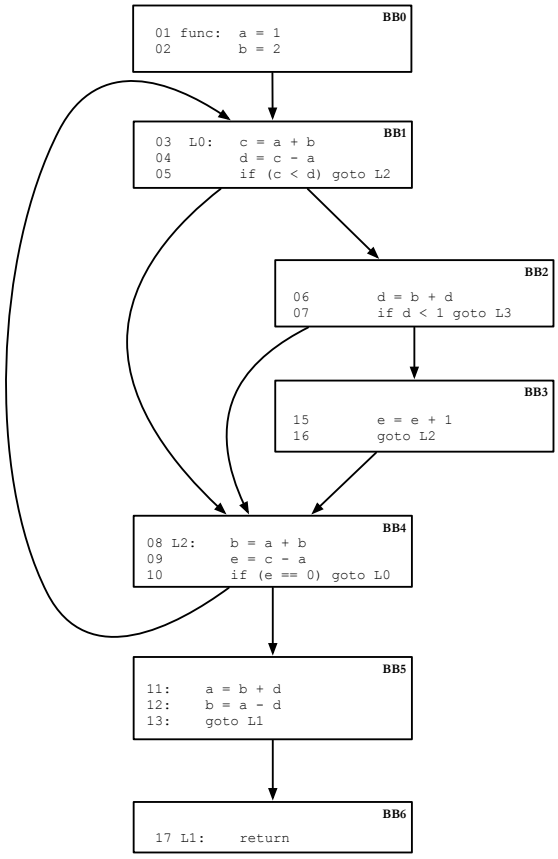
For the code shown on the left, determine the following:

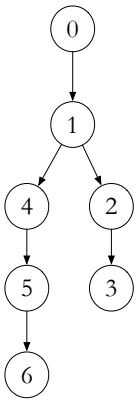
```
01 func:  a = 1
02        b = 2
03 L0:    c = a + b
04        d = c - a
05        if c < d goto L2
06        d = b + d
07        if d < 1 goto L3
08 L2:    b = a + b
09        e = c - a
10        if e == 0 goto L0
11        a = b + d
12        b = a - d
13        goto L1
14 L3:    d = a + b
15  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

a) The basic blocks of instructions identifying the instructions that constitutes each basic block. Clearly identify the leader instruction of each basic block.
b) The control-flow graph (CFG) and its dominator tree.
c) For each variable, determine its *use-def* chains.
d) The set of available expressions at the beginning of each basic block. You do not need to compute the DFA solution at each step of the iterative formulation but rather argue that your solution is correct by explaining the specific value of Available-Expressions DFA problem.
e) Is the Available Expressions DFA a forward or backwards data-flow analysis problem and why?
f) What does guarantee its termination when formulated as a data-flow analysis iterative problem?

Solution:

a) The basic blocks of instructions identifying the instructions that constitutes each basic block. Clearly identify the leader instruction of each basic block.
b) The control-flow graph (CFG) and the corresponding dominator tree.
See the answer to both these questions below.





c) For each variable we indicate for a given definition "d" its uses "u" as the line numbers in which they occur.

```
a: {d1, u3, u4, u8, u9} {d11, u12} {d1, u3, u4, u14}
b: {d2, u3, u6, u8} {d8, u11} {d8, u3, u6, u8}
c: {d3, u4, u5, u9}
d: {d4, u5, u11, u12} {d6, u7, u11, u12}
e: {d9, u10, u15} {d15}
```

d) You are not asked to compute the DFA solution at each step of the iterative formulation but rather argue that your solution is correct by explaining the specific Available Expressions DFA problem. Justify your answer by showing the GEN and KILL sets of each of the basic blocks.

The set of expressions in the program are outlined below on the left. On the right we have for each Basic Block `BB0` through `BB6` we have the following GEN and KILL sets.

```
(1): (a+b) line 3        BB0: gen = { },     kill = {1,2,3,5,6,7,8,9}
(2): (c-a) line 4        BB1: gen = {1,2},   kill = {2,3,5,7,9}
(3): (b+d) line 6        BB2: gen = {3},     kill = {3,5,9}
(4): (e+1) line 15       BB3: gen = {8},     kill = {3,4,5,9}
(5): (a-d) line 12       BB4: gen = {6,7},   kill = {1,3,4,6,8,9}
(6): (a+b) line 8        BB5: gen = {5},     kill = {1,2,3,5,6,7,8,9}
(7): (c-a) line 9        BB6: gen = { },     kill = { }
(8): (a+b) line 14
(9): (b+d) line 11
```

Applying the iterative fixed-point computation with set intersection corresponding to this problem's DFA formulation as described in class we would arrive at the following final solution for the set of available expression at the input of each basic block:

```
BB0: { } by definition
BB1: {4}
BB2: {1,2,4}
BB3: {1,2,3,4}
BB4: {2,4}
BB5: {2,4}
BB6: {4,5,9}
```

Recall that for each basic block the input is the intersection of the outputs of all its predecessors and the output of each basic block is computed by the data-flow equation: Out = gen + (In – kill).

e) This DFA problem is a forward data-flow problem as the expression become un-available as soon as one of its operands is redefined. At the beginning no expressions are available as they are generated as soon as the expression that defined them are encountered. A given expression is only available at a specific execution point is all path leading to that specific point have that expression available. As such the meet function is the set intersection.

f) With an initial solution of the set of all available expressions (except for the initial node whose initial solution is empty) and applying the set intersection operation, given that the number of expressions is finite the iterative application of the set intersection will eventually converge. Hence the termination of the fixed-point computation is guaranteed.

Problem 6

Consider the three-address instruction code below:

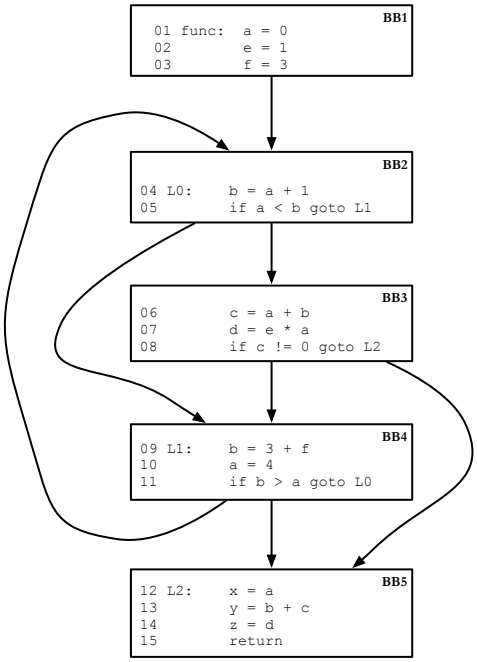
```
01 func:  a = 0
02        e = 1
03        f = 3
04 L0:    b = a + 1
05        if a < b goto L1
06        c = a + b
07        d = e * a
08        if c != 0 goto L2
09 L1:    b = 3 + f
10        a = 4
11        if b > a goto L0
12 L2:    x = a
13        y = b + c
14        z = d
15        return
```

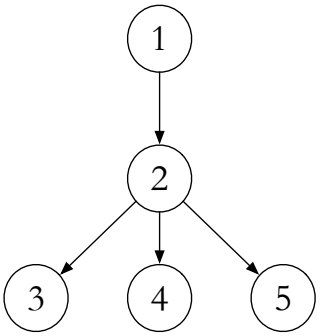
For this code determine the following:

a. The set of basic blocks and the corresponding control-flow graph and dominator tree.
b. The DU-chains and Reaching Definitions for all variables `a`, `b`, `c`, `d`, `e` and `f`. Ignore in this analysis the variables `x`, `y` and `z`.
c. Determine a new representation of this code using SSA form representation.
d. Are there any opportunities for constant propagation? How would you detect them from either the information on dominance or via the SSA representation of the program? Which do you think would be the easiest to implement in a compiler? Explain.

Solution:

a. The code defines 5 basic blocks as shown below (left) and the corresponding dominator tree (right).

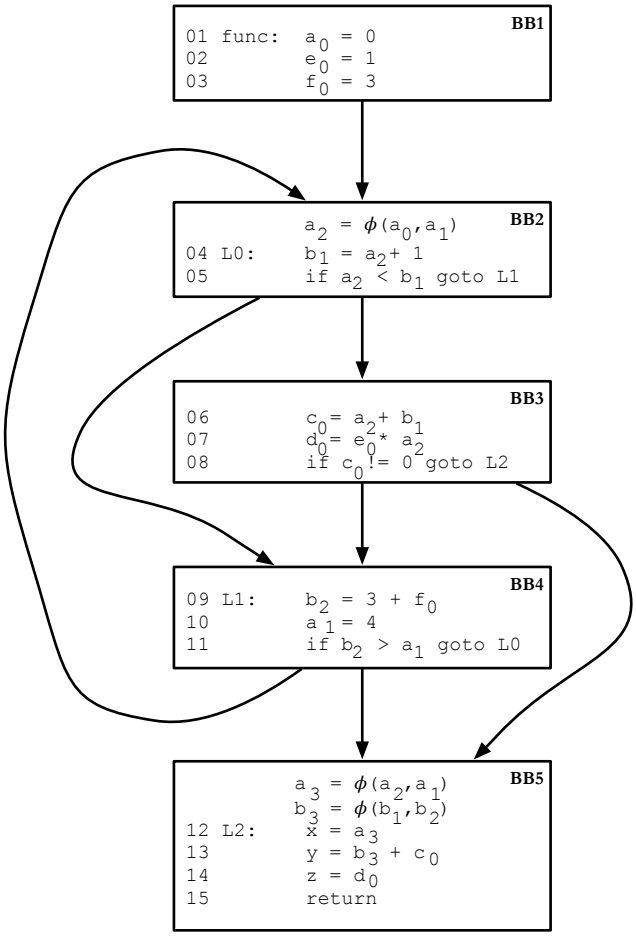




b. The *def-use* chains are shown below, where we indicate for each variable definition the line of the code where the variable is defined by an index. We denote the variable uses accordingly. For each variable definition d_i we indicate which uses u_j make use of it.

Variable	Def-Use Chains
a	$d_1 \rightarrow u_4, u_5, u_6, u_7, u_{12}$
	$d_{10} \rightarrow u_2, u_3, u_4, u_5, u_{11}, u_{12}$
b	$d_4 \rightarrow u_5, u_6, u_{13}$
	$d_9 \rightarrow u_{11}, u_{13}$
c	$d_6 \rightarrow u_8, u_{13}$
d	$d_7 \rightarrow u_{14}$
e	$d_2 \rightarrow u_7$
f	$d_3 \rightarrow u_9$

c. The program in SSA form is depicted below:



d. The program in SSA form makes it very clear which definition reach which uses and therefore make it trivial the application of copy-propagation as well as constant propagation. In this particular example on line `09` we can replace the instance of `f` by its single assignment to the constant value `3`. Similarly, the on line `07` we can replace the instance of `e` by its single assignment to the constant value `1`. Then, with the application of constant propagation we can even simplify the control-flow by observing that the predicate on line `11` is constant and always `true`.

Problem 5 Data-Flow Analysis

```
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 L1
17 L4: return
```

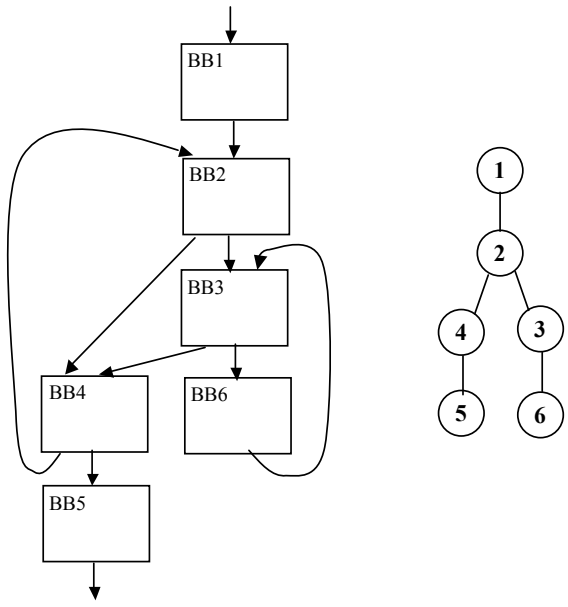
For the code shown above, determine the following:

f) The basic blocks of instructions and the control-flow graph (CFG).
g) 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.
h) 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?

***Solution**:*

a) We indicate the instructions in each basic block and the CFG and dominator tree on the RHS.

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



b) 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" 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, u15}
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}
e: {d14, u6}
```

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 are 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, d, e}
```

For BB6 the live variable solution at the exit of this basic block has `{a, b, c, d, e}` as there exists a path after BB6 where all the variables are being used. For example, variable "`e`" is used in the basic block following the L1 label and taking the jump to L3 to BB6 again where is used before being redefined.

c) 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 6: Data-Flow Analysis

```
01 func:  a = 1
02        b = 2
03 L0:    c = a + b
04        d = c - a
05        if c < d goto L2
06        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 L1
14 L3:    d = a + b
15        e = e + 1
16        goto L2
17 L1:    return
```

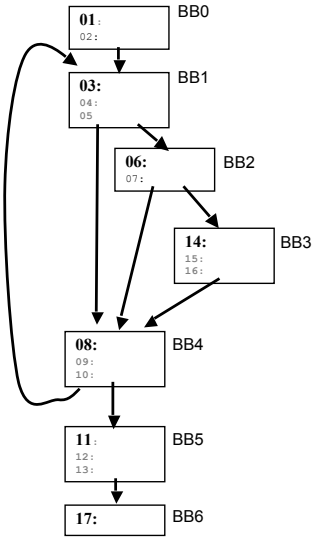
For the code shown on the left, determine the following:

g) The basic blocks of instructions identifying the instructions that constitutes each basic block. Clearly identify the leader instruction of each basic block.
h) The control-flow graph (CFG) and the corresponding dominator tree.
i) For each variable, its *use-def* chains.
j) The set of available expressions at the beginning of each basic block. You do not need to compute the DFA solution at each step of the iterative formulation but rather argue that your solution is correct by explaining the specific Available Expressions DFA problem.
k) Is the Available Expressions DFA a forward or backward data-flow analysis problem? Why and what does guarantee its termination when formulated as a data-flow analysis iterative problem?

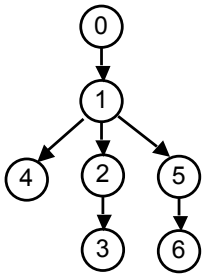
Solution:

g) The basic blocks of instructions identifying the instructions that constitutes each basic block. Clearly identify the leader instruction of each basic block.
h) The control-flow graph (CFG) and the corresponding dominator tree.

See the answer to both these questions below where the leader of each basic block is in bold font.



Control-Flow Graph



Dominator Tree

i) For each variable, its *use-def* chain.
For each variable we indicate for a given definition "d" its uses "u" as the line numbers in which they occur.

```
a: {d1, u3, u4, u8, u9} {d11, u12} {d1, u3, u4, u14}
b: {d2, u3, u6, u8} {d8, u11} {d8, u3, u6, u8}
c: {d3, u4, u5, u9}
d: {d4, u5, u11, u12} {d6, u7, u11, u12}
e: {d9, u10, u15} {d15}
```

j) The set of available expressions at the beginning of each basic block. You do not need to compute the DFA solution at each step of the iterative formulation but rather argue that your solution is correct by explaining the specific Available Expressions DFA problem. Justify your answer by showing the GEN and KILL sets of each of the basic blocks.

The set of expressions in the program are outlined below on the left. On the right we have for each Basic Block BB0 through BB6 we have the following GEN and KILL sets.

(1): (a+b) line 3
(2): (c-a) line 4
(3): (b+d) line 6
(4): (e+1) line 15
(5): (a-d) line 12
(6): (a+b) line 8
(7): (c-a) line 9
(8): (a+b) line 14
(9): (b+d) line 11

BB0: gen = { }, kill = {1,2,3,5,6,7,8,9}
BB1: gen = {1,2}, kill = {2,3,5,7,9}
BB2: gen = {3}, kill = {3,5,9}
BB3: gen = {8}, kill = {3,4,5,9}
BB4: gen = {6,7}, kill = {1,3,4,6,8,9}
BB5: gen = {5}, kill = {1,2,3,5,6,7,8,9}
BB6: gen = { }, kill = { }

Applying the iterative fixed-point computation with set intersection corresponding to this problem's DFA formulation as described in class we would arrive at the following final solution for the set of available expression at the input of each basic block:

BB0: { } by definition
BB1: {4}
BB2: {1,2,4}
BB3: {1,2,3,4}
BB4: {2,4}
BB5: {2,4}
BB6: {4,5,9}

Recall that for each basic block the input is the intersection of the outputs of all its predecessors and the output of each basic block is computed by the data-flow equation out = gen + (in – kill).

k) Is the Available Expressions DFA a forward or backward data-flow analysis problem? Why and what does guarantee its termination when formulated as a data-flow analysis iterative problem?

This DFA problem is a forward data-flow problem as the expression become un-available as soon as one of its operands is redefined. At the beginning no expressions are available as they are generated as soon as the expression that defined them are encountered. A given expression is only available at a specific execution point is all path leading to that specific point have that expression available. As such the meet function is the set intersection. With an initial solution of the set of all available expressions (except for the initial node whose initial solution is empty) and applying the set intersection operation, given that the number of expressions is finite the iterative application of the set intersection will eventually converge. Hence the termination of the fixed-point computation is guaranteed.

Problem 7: Optimizations

Consider the 3-address code depicted below already with specific register allocation and assignments.

```
01:      r0 = 0
02:      r1 = $fp - 4
03:      r1 = *r1
04:      r2 = 0
05:      r4 = 1
06:      r5 = r1
07: L1:  if r0 = r1 goto L5
08:      if r2 = r5 goto L3
09:      goto L2
10: L2:  r3 = 1
11:      r0 = r0 + 1
12:      goto L6
13: L3:  r4 = 0
14: L4:  goto L7
15: L5:  r3 = 1
16:      r0 = r0 + 1
17:      r4 = 0
18: L6:  r4 = r0 * 2
19:      goto L8
20: L7:  r1 = FP + 4
21:      *r1 = r4
22:      r1 = $gp + 20
23:      *r1 = r5
24:      return
25: L8:  r5 = r5 - r3
26:      if r5 != 0 goto L1
27:     r4 = 1
28:      goto L7
```

For this code fragment determine:

a. [10 points] The corresponding Control Flow Graph (CFG) identifying the basic blocks with their instructions.

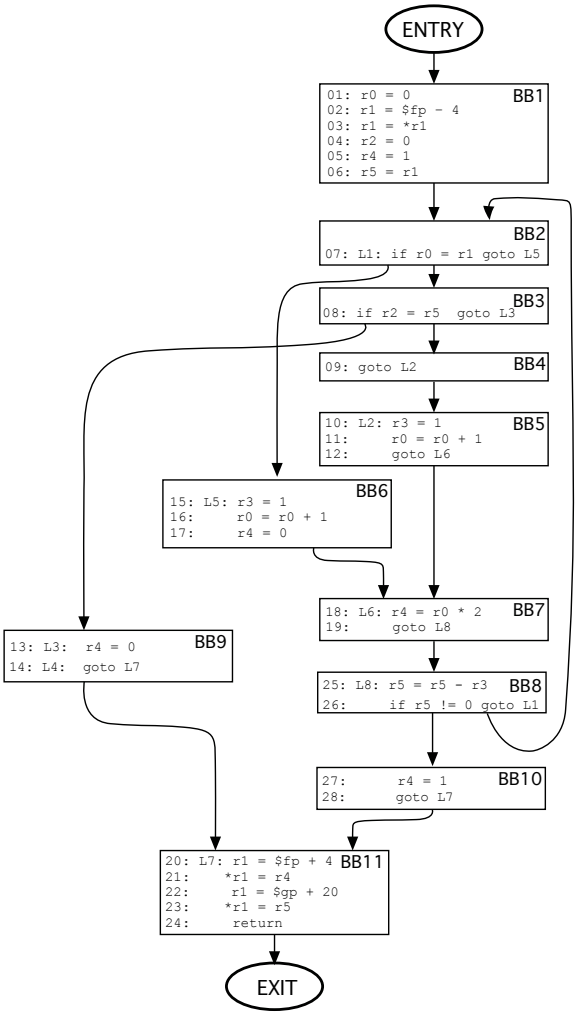
b. [10 points] The dominator tree and the natural loops (identify the back edges). The dominator tree is a simple relationship between nodes. A node in this tree is directly connected to the nodes it immediately dominates.

c. [10 points] Check and transform the code taking advantage of loop-invariant code motion arguing about the correctness of your transformations.

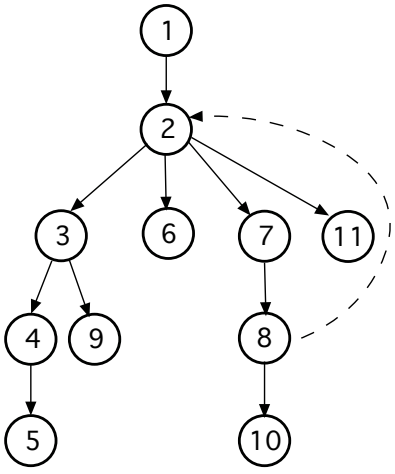
d. [10 points] Check and transform the code taking advantage of induction variables in this loop arguing about the correctness of your transformations.

Solution:

a) The control-flow graph is as shown below on the left-hand-side.
b) The dominator tree is shown on the right-hand side. The natural loop induced by the back-edge (8-2). Tracing backwards this edge from the basic block BB8 towards BB2 we encounter the basic blocks {2, 3, 4, 5, 6, 7, 8}
c) There are really no loop-invariant instructions in the only loop in this code. The only candidate instruction which one could move to the head of the loop would be the instructions "r3 = 1" in the basic blocks BB5 and BB6. Unfortunately, none of these basic blocks by themselves dominate the basic blocks where there are uses of the variable r3 in BB8. Similarly, we can see that the assignment " r4 = 0" in BB6 would be a candidate for loop invariant code motion. Again, this statement does not dominate the exit of the loop (BB2). In fact that are other assignment statement to variable r4.



Control-Flow Graph (CFG)



loop = { 2, 3, 4, 5, 6, 7, 8 }

Dominator Tree, Back-edge and Natural Loop

d) There are some opportunities for induction variables in this loop, which are not trivial to grasp, and for which the help of data-flow analysis (as seen in the next problem) are required. There is an increment on r0 by 1 at every iteration of the loop although the increment occurs in different basic blocks. At each iteration of the loop, however, only one of them is executed. This means that r0 is a *basic induction variable*. As such the variable r4

is a *derived* induction variable with increment by 2. The fact that we have an assignment to r4 in basic block BB6 is irrelevant as that assignment is dead in the sense that is immediately overwritten in BB7.