

Compilers

Fall 2009

Control-Flow Analysis

Sample Exercises and Solutions

Prof. Pedro Diniz

Departamento de Engenharia Informática (DEI)
Instituto Superior Técnico / Tagus Park
Av. Prof. Dr. Cavaco Silva
2780-990 Porto Salvo
Portugal

Problem1: Control-Flow Graph, Dominators and Natural Loops

For the program below:

1. Determine the corresponding control flow graph.
2. Determine the dominators of each node in the CFG
3. Show the immediate dominator tree
4. Identify the set of nodes in each natural loop. Are the loops nested? Why

```

00:  i = 0;
01:  while (i < N) do
02:      for(i=0; j < i; j++)
03:          if(a[i] < a[j]) then
04:              tmp = a[j];
05:              a[j] = a[i]
06:              a[i] = tmp;
07:          else
08:              if(a[i] == a[j]) then
09:                  continue;
10:              else
11:                  break;
12:      end for
13:      i = i + 1;
14:  end while

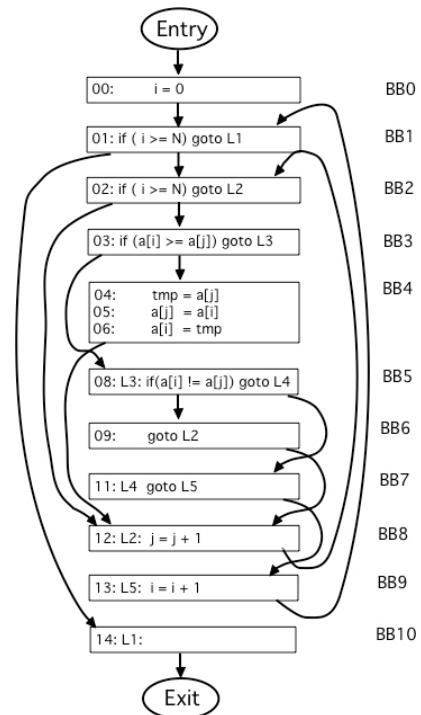
```

Note: Assume there are two *entry* and *exit* nodes corresponding to the entry point and exit points of the code segment, which would correspond to the prologue and epilogue sections of the procedure's generated code.

Answers:

1. A possible control flow graph is as show on the right.
2. Node Dominators

0	{0}
1	{0, 1}
2	{0, 1, 2}
3	{0, 1, 2, 3}
4	{0, 1, 2, 3, 4}
5	{0, 1, 2, 3, 5}
6	{0, 1, 2, 3, 5, 6}
7	{0, 1, 2, 3, 5, 7}
8	{0, 1, 2, 3, 8}
9	{0, 1, 2, 3, 5, 7, 9}
10	{0, 1, 10}



3. Immediate dominator tree edges: 0->1, 1->2, 1->10, 2->3, 3->4, 3->5, 3->8, 5->6, 5->7, 7->9

4. Natural loops

Back edge	Nodes in natural loop
9->1	{2, 3, 5, 7, 9}
8->2	{1, 2, 3, 4, 5, 6, 8}

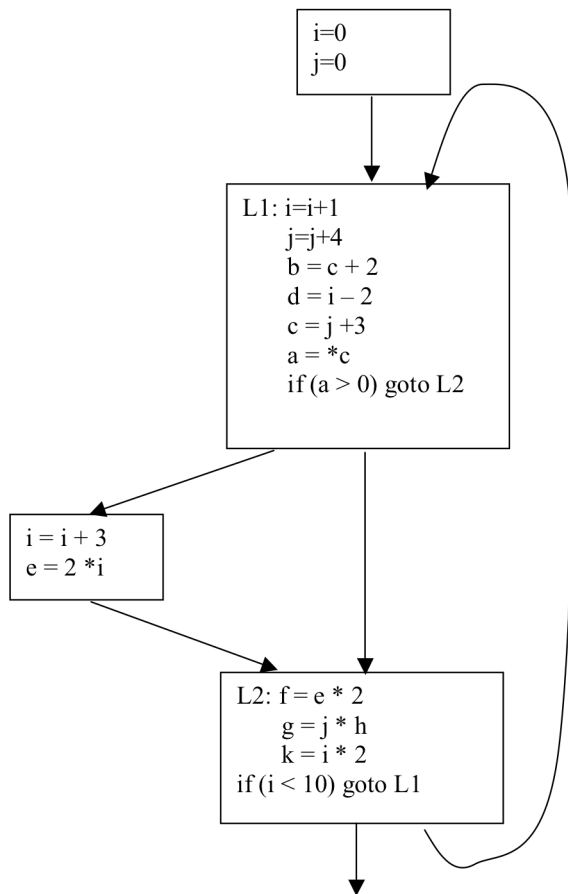
They are nested because they have different headers, and natural loops with different headers can either be nested or disjoint. They are not disjoint, so they must be nested.

Problem 2: Loop Induction Variables and Loop Unrolling

For the program depicted below do the following:

1. Find the induction variables in this program. Show the results after applying strength reduction and loop test replacement.
2. Show the resulting code from applying loop unrolling with an unrolling factor of 2.

Note: The “a = *c” assigns the contents of the memory location pointed to by c to a



Answers:

Assume there is no alias among the variables, so h is not changed within the loop. Basic induction variables: $\{i, j\}$ Induction variable families: $\{\{i, d, e, k\}, \{j, g, c\}\}$ (We can consider h as a constant within the loop, and as a result, g will be an induction variable in the same family with j) Introduce $s1$ for d , $s2$ for e and k , $s3$ for g , $s4$ for c , and eliminate basic induction variables i and j because they are only used for calculation of other induction variables and loop test. After strength reduction and loop test replacement:

$s1 = -2$	$\text{if } a > 0 \text{ goto L2}$
$s2 = 0$	
$s3 = 0$	$s1 = s1 + 3$
$t3 = 4 * h$	$s2 = s2 + 6$
$s4 = 3$	$e = s2$
L1: $s1 = s1 + 1$	L2: $f = e * 2$
$s2 = s2 + 2$	$g = s3$
$s3 = s3 + t3$	$k = s2$
$s4 = s4 + 4$	$\text{if } s1 < 8 \text{ goto L1}$
$b = c + 2$	
$d = s1$	$i = d + 2$
$c = s4$	$j = c - 3$
$a = *c$	
1) After loop unrolling:	
$s1 = -2$	L3: $f = e * 2$
$s2 = 0$	$g = s3$
$s3 = 0$	$k = s2$
$t3 = 8 * h$	
$s4 = 3$	$\text{if } s1 < 4 \text{ goto L1}$
$\text{if } s1 \geq 4 \text{ goto L4}$	L4: $s1 = s1 + 1$
L1: $s2 = s2 + 2$	$s2 = s2 + 2$
$s4 = s4 + 4$	$s3 = s3 + t3$
$c = s4$	$s4 = s4 + 4$
$a = *c$	$b = c + 2$
$\text{if } a > 0 \text{ goto L2}$	$d = s1$
	$c = s4$
$s1 = s1 + 3$	$a = *c$
$s2 = s2 + 6$	$\text{if } a > 0 \text{ goto L5}$
$e = s2$	
L2: $s1 = s1 + 2$	$s1 = s1 + 3$
$s2 = s2 + 2$	$s2 = s2 + 6$
$s3 = s3 + t3$	$e = s2$
$b = c + 2$	
$d = s1$	L5: $f = e * 2$
	$g = s3$
$s4 = s4 + 4$	$k = s2$
$c = s4$	$\text{if } s1 < 8 \text{ goto L4}$
$a = *c$	
$\text{if } a > 0 \text{ goto L3}$	$i = d + 2$
	$j = c -$
$s1 = s1 + 3$	
$s2 = s2 + 6$	
$e = s2$	

Problem 3: Loops and Loop Optimizations

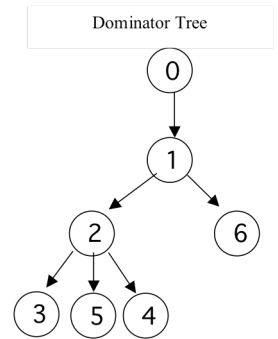
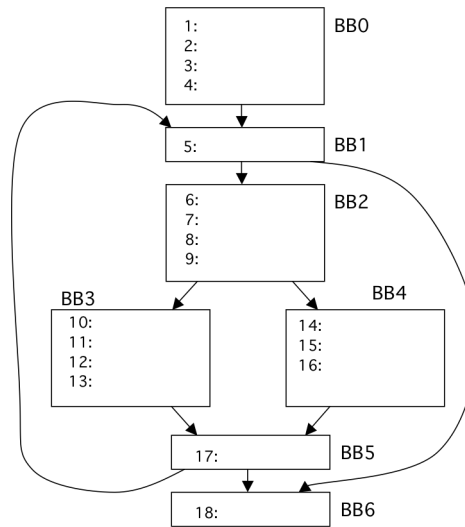
Consider the code depicted below for a function with integer local variables i , a , b , c , d and an integer input parameters p and n .

```

int i, a, b, c, d;

1:    i = 0;
2:    a = 1;
3:    b = i * 4;
4:    c = a + b;
5: L1: if(i > n) goto L2
6:    c = a + 1;
7:    i = i + 1;
8:    b = i * 4;
9:    if (c <= p) goto L3
10:   e = 1;
11:   c = e - b;
12:   a = e;
13:   goto L4
14: L3: d = 2;
15:   c = d + b;
16:   a = d;
17: L4: goto L1
18: L2: return;

```



For this code fragment determine:

1. The Control Flow Graph (CFG).
2. The dominator tree and the loops (identify the back edges).
3. Use constant propagation to the statements in the body of the loop identified in (2).
4. Are the statements in line 8 and 16 loop invariants? Explain and describe where can they be moved to if they can in fact be moved outside any of the loops.

Answers:

1. See the CFG on the left above.
2. See the dominator tree on the right above.
3. Back edge is (5,1) since head dominates its tail. The natural loop is (1,2,3,4,5).
4. Could remove the statement $a = p1$ to the head of the loop. This is only valid because a is not live at the end of the loop so even if the original loop were never to execute and since the head of the loop would execute once this transformation would still be correct.

Problem 4: Algebraic Transformations

For the code below apply the following code transformations: constant propagation, constant folding, copy-propagation, dead-code elimination and strength reduction.

```

L0:    t1 = t1 + 1
      t2 = 0
      t3 = t1 * 8 + 1
      t4 = t3 + t2
      t5 = t4 * 4
      t6 = *t5
      t7 = FP + t3
      *t7 = t2
      t8 = t1
      if (t8 > 0) goto L1
L1:    goto L0
L2:    t1 = 1
      t10 = 16
      t11 = t1 * 2
      goto L1

```

Answers:

The code in red is unreachable and can thus be eliminated. The goto L1 in the end is a goto to another goto instruction and thus can be converted into a goto L0 instruction.

<pre> L0: t1 = t1 + 1 t2 = 0 t3 = t1 * 8 + 1 t4 = t3 + t2 t5 = t4 * 4 t6 = *t5 t7 = FP + t3 *t7 = t2 t8 = t1 if (t8 > 0) goto L0 L1: goto L0 L2: t1 = 1 t10 = 16 t11 = t1 * 2 goto L1 </pre>	<pre> L0: t1 = t1 + 1 t2 = 0 t3 = (t1 << 3) + 1 t4 = t3 t5 = (t4 << 2) t6 = *t5 t7 = FP + t3 *t7 = t2 t8 = t1 if (t8 > 0) goto L0 </pre> <p>after symbolic propagation for (t2 = 0) and strength reduction of the various multiplications by 4 and 8.</p>	<pre> L0: t1 = t1 + 1 t2 = 0 t3 = (t1 << 3) + 1 t4 = t3 t5 = (t3 << 2) t6 = *t5 t7 = FP + t3 *t7 = t2 t8 = t1 if (t1 > 0) goto L0 </pre> <p>after copy propagation; now we can eliminate the variables t8 and t4 and t2 as they are never used in this code.</p>
--	---	--

after dead-code elimination

In reality you can also see that t1 is now loop invariant and so is the sub-expression $(t1 \ll 3) + 1$ which can now be hoisted outside the loop. Along the same reasoning t5 is also loop invariant and so is everything else. This is a very uninteresting example.

Problem 5: Control-Flow Graph Analysis and 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:

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

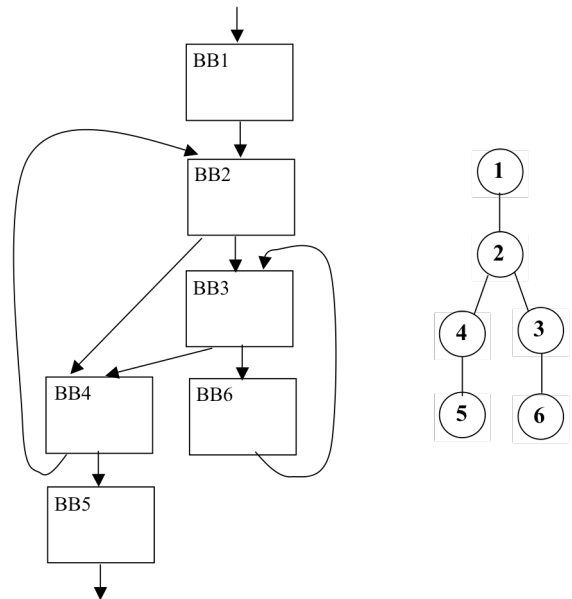
Answers:

- 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 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, 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: Control-Flow Graph Analysis and 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

```

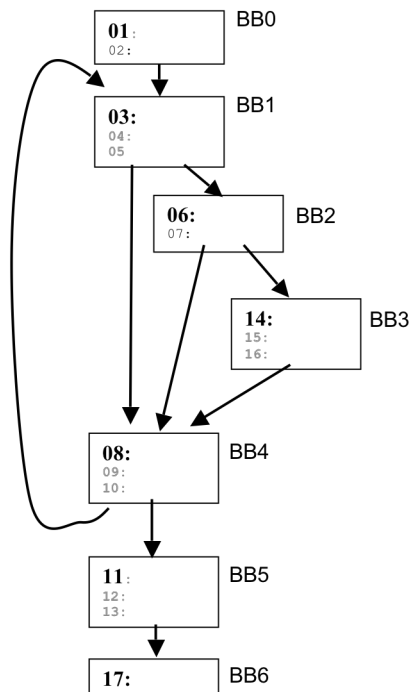
For the code shown on the left, determine the following:

- The basic blocks of instructions identifying the instructions that constitutes each basic block. Clearly identify the leader instruction of each basic block.
- The control-flow graph (CFG) and the corresponding dominator tree.
- For each variable, its *use-def* chains.
- 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.
- 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?

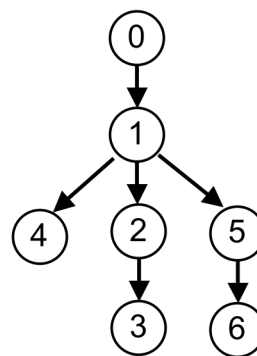
Answers:

- The basic blocks of instructions identifying the instructions that constitutes each basic block. Clearly identify the leader instruction of each basic block.
- 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

- c) 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}

```

- 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 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 = {4,8}, 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 = {5,9}, 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: {1,2,3,4,5,6,7,8,9} 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) 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.