CSCI 565 - Compiler Design

Spring 2015

Second Test Solution

Problem 1. Control-Flow Analysis [50 points]

Consider the three-address code below for a procedure with input/output arguments passed on the Activation Record on the stack.

```
01:        t0 = $fp - 4
02:        t0 = *t0
03:        t2 = $fp - 8
04:        t2 = *t2
05:        t1 = 0
06:        if t0 < 16 goto L2
07:        goto L1
08:  L1:   t1 = t1 + 1
09:        t2 = 4 * t1
10:        t3 = $gp + t2
11:        t4 = *t3
12:        t5 = t4 / t1
13:        if t5 >= 0 goto L3
14:        goto L5
15:  L2:   t1 = t1 - 1
16:        if t0 > 16 goto L3
17:        goto L6
18:  L3:   t0 = t0 + 1
19:        if t0 < 16 goto L1
20:        goto L2
21:  L5:   t1 = t4
22:        goto L6
23:  L6:   t0 = $fp - 4
24:        *t0 = t1
25:        return
```

Questions:
For this code determine the following:

a. [10 points] Basic blocks and the corresponding control-flow graph (CFG) indicating for each basic block the corresponding line numbers of the code above.

b. [10 points] Dominator tree and the natural loops in this code (if any) along with the corresponding back edge(s).

c. [10 points] Determine the live ranges using the "second" more sophisticated definition that takes into account the read and write of each specific variable/temporary, and the corresponding webs for the variables t0, t1, t2, t3, t4 and t5. Explain the way you combine the *def-use* chains for the variable t0 and t1 to form the webs for those variables. You do not need to be as specific for the other variables, so present only the corresponding webs.

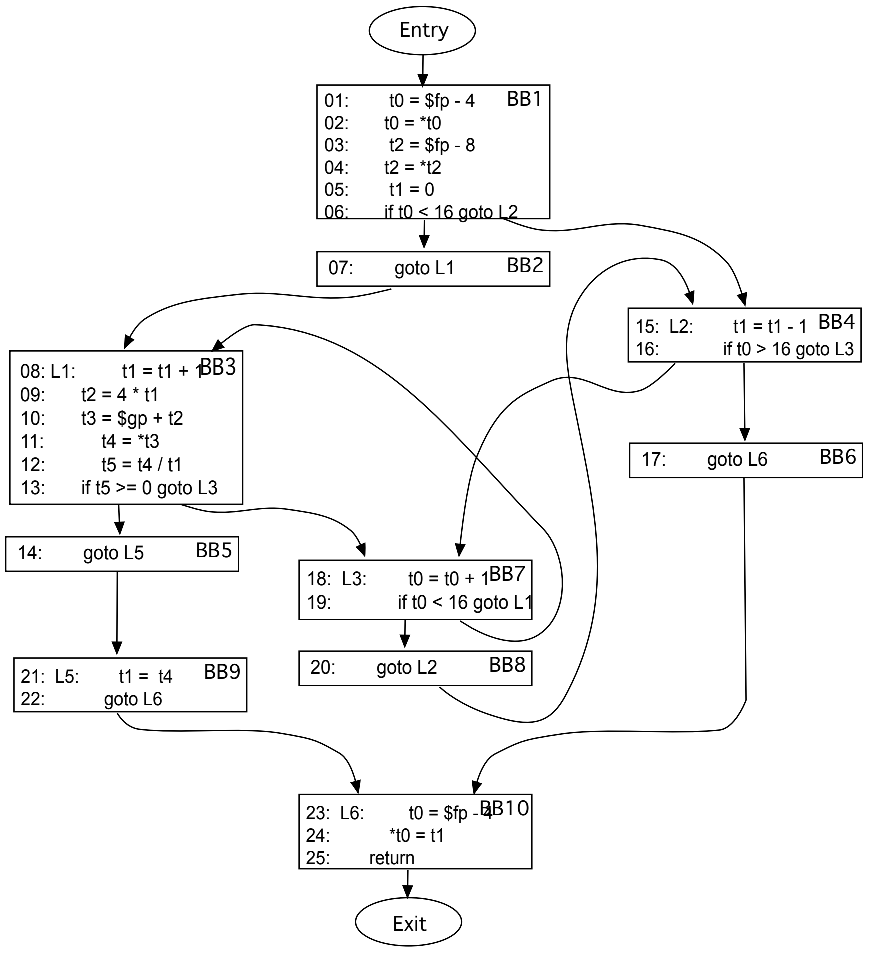
d. [10 points] Derive the interference graphs (or table) for these variables. Explain in detail the interference (or lack thereof) between the webs corresponding to the variables t2 and t3.

e. [10 points] Can you color the resulting interference graphs with 3 colors? Why or why not? Present a coloring assignment for 4 colors. If, however, you were to use only two registers describe where spill code could be included.

Name: ________________

Solution:

a. [10 points] The CFG is as shown below.



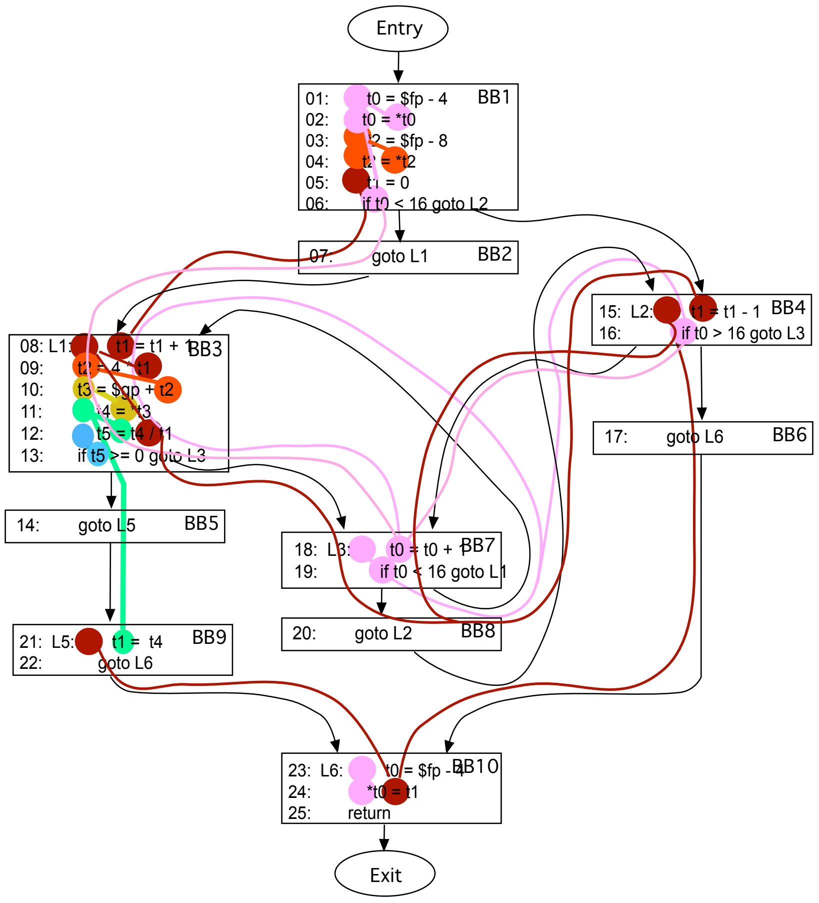
```
01:      t0 = $fp - 4
02:     t0 = *t0
03:      t2 = $fp - 8
04:     t2 = *t2
05:      t1 = 0
06:     if t0 < 16 goto L2

07:     goto L1

08: L1:        t1 = t1 + 1
09:      t2 = 4 * t1
10:      t3 = $gp + t2
11:          t4 = *t3
12:          t5 = t4 / t1
13:      if t5 >= 0 goto L3

14:      goto L5

15:  L2:        t1 = t1 - 1
16:            if t0 > 16 goto L3

17:      goto L6

18:  L3:       t0 = t0 + 1
19:            if t0 < 16 goto L1

20:      goto L2

21:  L5:       t1 =  t4
22:          goto L6

23:  L6:       t0 = $fp - 4
24:           *t0 = t1
25:       return
```

b. [10 points] This CFG has no natural loops given than there are no edges in the CFG whose basic blocks pointed to by their 'heads' dominate the basic blocks dominates by their 'tails'. Still there are two loops in this CFG sharing a basic block, namely { 3, 7 } and { 4, 7, 8 }.

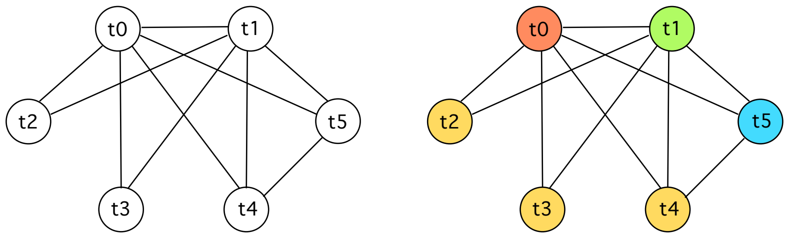
$$\begin{tikzcd}
 & & 01 \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
02 & 07 \arrow[d] & 03 \arrow[d] & 04 \arrow[d] & 10 \\
 & 08 & 05 \arrow[d] & 06 & \\
 & & 09 & &
\end{tikzcd}$$

c. [10 points] Only the temporary variables t0 and t1 have long-lived ranges as shown below. The variables t2, t3, t4, and t5 are used mostly in basic block BB3 and in the case of t2 only in basic block BB1 where the values of t2 are 'dead'. Below is the list of individual webs for the distinct values.

```
t0 = {1,2}{2-6-13, 18, 19, 20, 15, 16}{24,25}
t1 = {5-8}{8-13,18-20,15,17,21,24,25}
t2 = {3,4}{9,10}
t3 = {10,11}
t4 = {11-14, 21}
t5 = {12,13}
```



d. [10 points] The interference graph is as shown below (left). The webs for the variables t1 and t2 interfere in lines 08 and 09 only as the value of t1 needs to be preserved along the path from the definition in line 08 to its use in line 12.

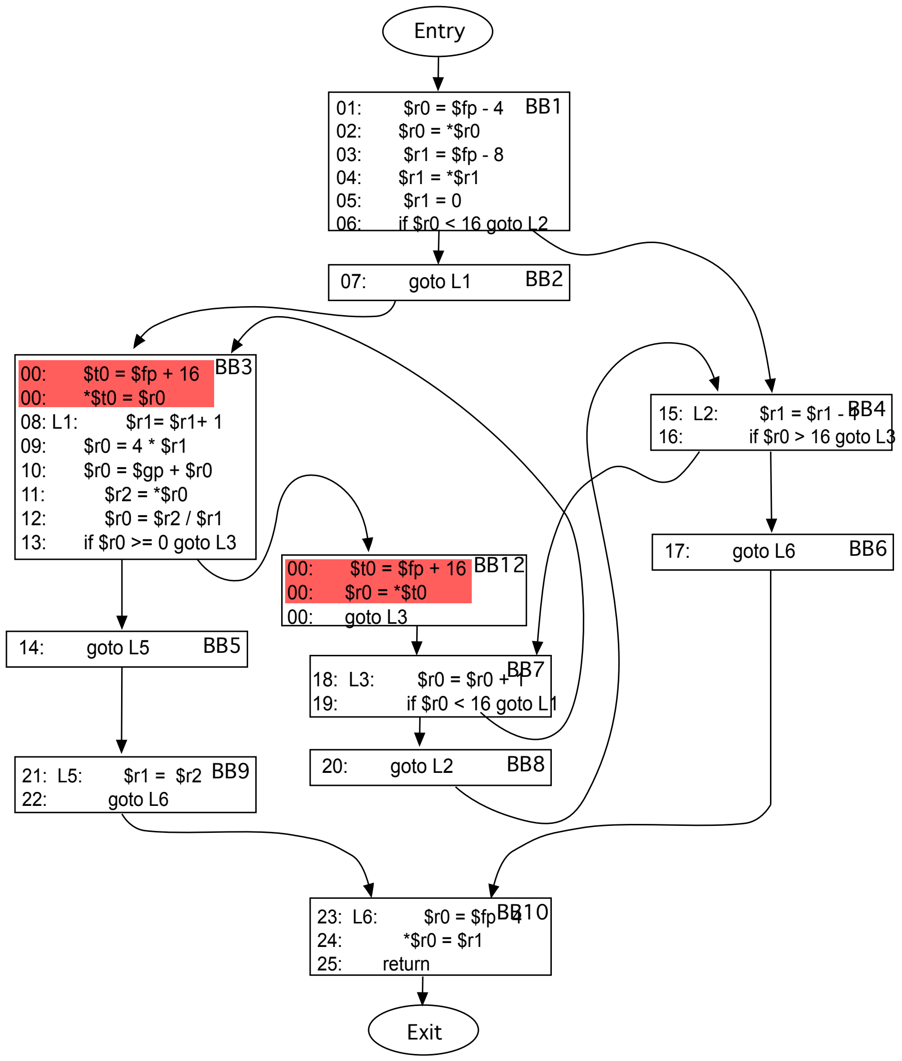


e. [10 points] Given that we have various cliques of size 4 in this graph we cannot color it with less than 4 colors. As it is apparent, t0, t1 and t5 will have three colors and the remainder nodes will all share the same color, hence the same register.

In addition, and as the web for `t0` has a 'long' hiatus where the value of `t0` is not being used in BB3, I would include a spill code for `t0` at the beginning of BB3 restoring it on exit of BB3. This in itself is a bit of a complication given that `t5` is being used for the

Name: ________________

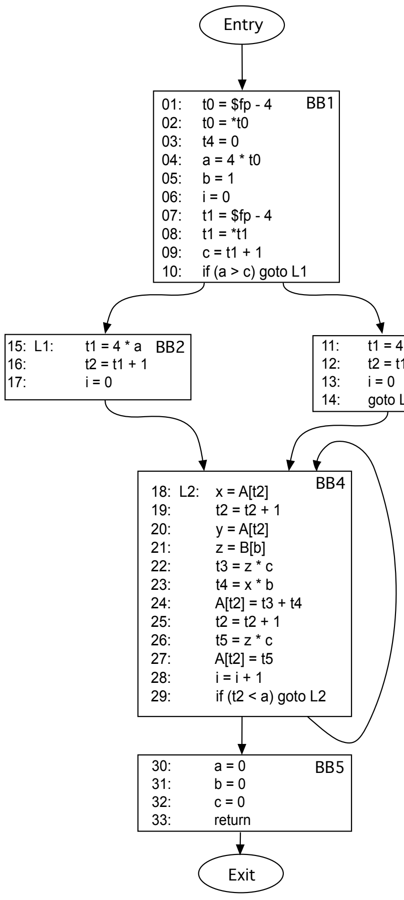
conditional test. As a result we would have to create a new basic block just for the instruction that restores t0 as shown in the revised CFG shown below. Note also that the saving and restoring of the register is made with the help of an auxiliary register $t0 not used for the 'regular' computation.



Problem 2. Code Optimization [15 points]

Consider the three-address code below (left) and the corresponding CFG (right) for a procedure with input/output arguments passed on the Activation Record (AR) on the stack. Explicit register variables are indicated with the $ sign. All other scalar variable references are assumed to be local variables to the procedure. The translation of the accesses to local variable and array variables (A and B) are not yet translated into accesses to the corresponding fields of the AR.

```
01:          t0 = $fp - 4
02:          t0 = *t0
03:          t4 = 0
04:          a = 4 * t0
05:          b = 1
06:          i = 0
07:          t1 = $fp - 4
08:          t1 = *t1
09:          c = t1 + 1
10:          if (a > c) goto L1
11:          t1 = 4 * a
12:          t2 = t1 + b
13:          i = 0
14:          goto L2
15:  L1:     t1 = 4 * a
16:          t2 = t1 + 1
17:          i = 0
18:  L2:     x = A[t2]
19:          t2 = t2 + 1
20:          y = A[t2]
21:          z = B[b]
22:          t3 = z * c
23:          t4 = x * b
24:          A[t2] = t3 + t4
25:          t2 = t2 + 1
26:          t5 = z * c
27:          A[t2] = t5
28:          i = i + 1
29:          if (t2 < a) goto L2
30:          a = 0
31:          b = 0
32:          c = 0
33:          return
```



Questions:

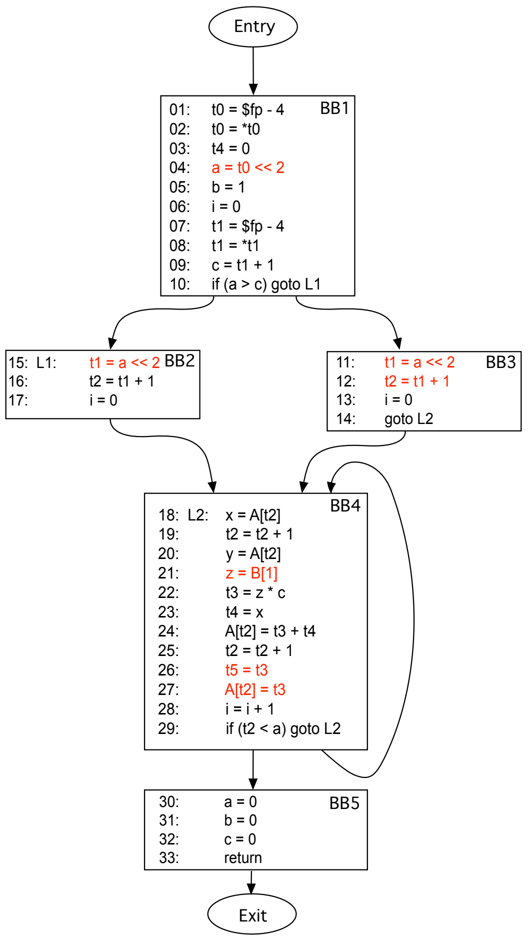
a. [15 points] Identify opportunities for the application of constant propagation, common-sub-expression elimination (CSE) algebraic simplification, strength reduction.

b. [10 points] Identify and exploit opportunities for loop invariant code motion (LICM) identifying basic induction variables of the loop(s) in this code. Furthermore, perform dead code elimination taking into account the results of live-variable analysis.

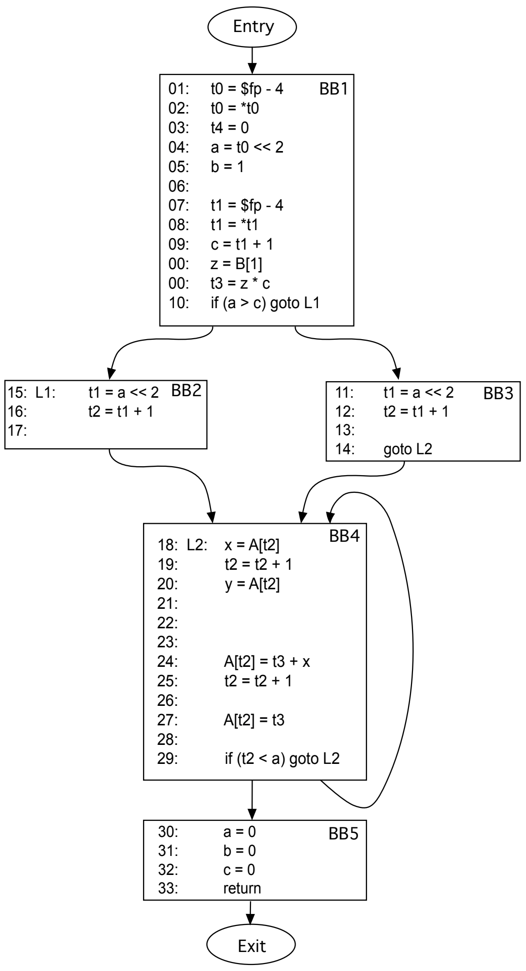
Solution:

a. [15 points] The computation of the offset of the arguments in lines 01 and 05 can be optimized by using an intermediate register to hold the offset computation. The assignment `b = 1` can be forward propagates to lines 12 and 21 (as BB1 dominates BB3 and BB4. There is the simple opportunity for strength-reduction by having `4*a` replaced by `(a<<2)` in lines 09 and 12. In BB4 there is a redundant expression computation `z*c` saved in t3 which can be reused and copy-propagated to t5 on line 27, thus eliminating lines 26. Lastly, the assignments to the variable `i` on lines 13 and 17 are redundant as they are dominated by the assignment on line 06 in BB1. Figure below (left) depicts these transformations.

b. [10 points] The computation `B[b]` which is the same as `B[1]` is loop invariant. Similarly, the computation of `z*c` is also loop invariant. Regarding induction variables there is the statement `t2 = t2 + 1` as well as `i = i + 1` which are the single assignment to `t2 and i` and are both executed in the sole basic block of the loop, hence dominating all its uses and the exit of the loop. In addition and once `b`'s value is propagate the variable `b` can be eliminated as its value is dead. Figure below (right) depicts these transformations.



```
BB1
01:    t0 = $fp - 4
02:    t0 = *t0
03:    t4 = 0
04:    a = t0 << 2
05:    b = 1
06:    i = 0
07:    t1 = $fp - 4
08:    t1 = *t1
09:    c = t1 + 1
10:    if (a > c) goto L1

BB2
15: L1:    t1 = a << 2
16:        t2 = t1 + 1
17:        i = 0

BB3
11:    t1 = a << 2
12:    t2 = t1 + 1
13:    i = 0
14:    goto L2

BB4
18: L2:  x = A[t2]
19:      t2 = t2 + 1
20:      y = A[t2]
21:      z = B[1]
22:      t3 = z * c
23:      t4 = x
24:      A[t2] = t3 + t4
25:      t2 = t2 + 1
26:      t5 = t3
27:      A[t2] = t3
28:      i = i + 1
29:      if (t2 < a) goto L2

BB5
30:      a = 0
31:      b = 0
32:      c = 0
33:      return
```



```
BB1
01:    t0 = $fp - 4
02:    t0 = *t0
03:    t4 = 0
04:    a = t0 << 2
05:    b = 1
06:
07:    t1 = $fp - 4
08:    t1 = *t1
09:    c = t1 + 1
00:    z = B[1]
00:    t3 = z * c
10:    if (a > c) goto L1

BB2
15: L1:    t1 = a << 2
16:        t2 = t1 + 1
17:

BB3
11:    t1 = a << 2
12:    t2 = t1 + 1
13:
14:    goto L2

BB4
18: L2:  x = A[t2]
19:      t2 = t2 + 1
20:      y = A[t2]
21:
22:
23:
24:      A[t2] = t3 + x
25:      t2 = t2 + 1
26:
27:      A[t2] = t3
28:
29:      if (t2 < a) goto L2

BB5
30:      a = 0
31:      b = 0
32:      c = 0
33:      return
```

Problem 3: Iterative Data-Flow Analysis [25 points]

Your task is to devise a data flow analysis for a problem, which we will call ***anticipation***. In the following, we motivate the analysis and explain the information you need to derive. The goal of common sub-expression elimination is to speed up program execution by eliminating redundant calculations. There is a more aggressive variation of this optimization known as partial redundancy elimination. Consider the following example:

```
if a > 0
  d = b + c
e = b + c
```

The expression (b + c) is evaluated twice if (a > 0) and once otherwise. That is, the second occurrence of (b + c) is partially redundant, since it may already have been computed. We can improve the program by inserting an earlier computation, which makes both occurrences completely redundant, transforming the code to the following:

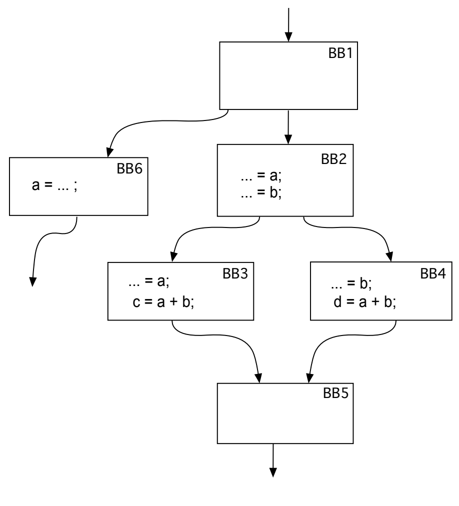
```
t = b + c
if a > 0
  d = t
e = t
```

The cost of this transformation is having to store the value in a register for a longer period of time; the benefit is that the program is sped up if the condition (`a > 0`) holds.

Your goal is to solve the Anticipation Analysis data-flow problem.
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.



Questions:

Describe your approach to anticipation analysis by answering the following questions:

a. [05 points] What is the set of values in the lattice and the initial values?
b. [05 points] What is the direction of the problem, backwards or forward and why?
c. [10 points] What is the meet function for this data-flow problem, i.e., the GEN and KILL and the equations the iterative approach needs to solve?
d. [05 points] How do you construct the transfer function of a basic block based on the GEN and KILL at the instruction level or another algorithmic method?

Solution:

a. [05 points] The set of values is simply the (ordered) set of expressions in the program, where each expression is uniquely identified by an integer and the same textual expression can have as many instances as they occur in the program. Points of the lattice are related by sub-set inclusion having a top element being the set of all set expressions in the program and has bottom element the empty set.

b. [05 points] This problem can be formulated as a backwards iterative data-flow analysis problem. We work backwards against the control-flow repeatedly checking if all paths emanating from the current program point `p` still require the evaluation of a given expression `e`. At the input of a basic block an expression `e` is anticipated if it is evaluated in that basic block and between the input and the evaluation point there are no definitions of either its operands. In other words, the expression is *upwards exposed*. This is an analysis scenario similar to the Live Variables problem.

c. [10 points] Given that we are requiring an expression to be anticipated in all paths emanating from the current program point `p`, the meet function is the set intersection. In the backwards formulation of this problem we can define the meet function as follows:

GEN = { set of expressions used in the current basic block whose operands are not previously defined in the same basic block}
KILL = { set of expressions with variable v on the RHS for which there is an assignment to v in the current basic block }

$$IN(b) = GEN(b) - (OUT(b) - KILL(b))$$
$$OUT(b) = \bigcap_{s \in succ(b)} IN(s)$$

d. [05 points] We can trace each expression backwards against the control flow. If any of its operands is defined, then we kill the expression and do not include it in the GEN set. For all LHS variables of assignment statements we include in the KILL set the expression number that have that variable as an operand. As with the Live Variable analysis problem we can also perform this 'local' computation in a forward fashion.

Regarding the snippet example shown above, the assignment to the variable `a` in BB6 would 'kill' the expression (`a+b`) in BB3 and BB4 despite this expression being anticipated at the beginning of BB2.