

CSCI 565 - Compiler Design

Spring 2014

Second Test Solution

Problem 1. Run-Time Environments and Data Layout [15 points]

Consider the following skeleton of source program shown below for a lexically scoped language that allows for dynamic arrays as is the case with the array *v* in the blocked region labeled as B1.

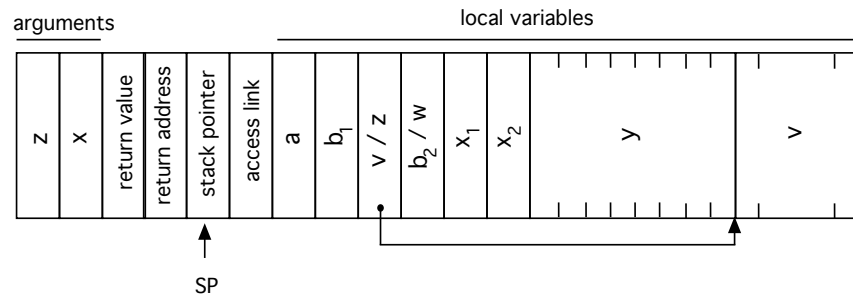
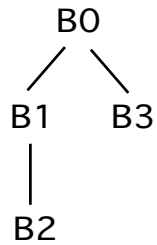
```
procedure F (x: integer, z: integer) {  
  B0: {  
    integer a, b;  
    ... assign value to a  
  B1: {  
    integer v(a), b, x;  
  B2: {  
    integer x, y(8);  
    ...  
  }  
  B3: {  
    integer z, w;  
    ...  
  }  
}
```

Questions:

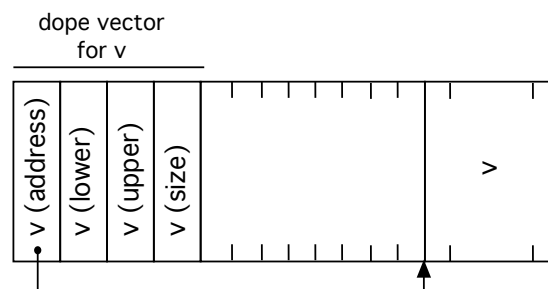
- [10 points] Describe the layout of the activation record for the procedure *F* with the various fields for input arguments and local variables. Discuss which sections of the AR can be reused and why.
- [5 points] Describe which run-time data structure need to be created and maintained to support the access to the variable-sized array *v* in B1. How is the initialization of these data structure carried out?

Answers:

- [10 points] A possible layout of the activation records for the procedure *F* is as shown below. Given that the block scopes B1 and B3 are not simultaneously active, they can share the storage of the AR for the corresponding local variables, namely { *v(a)*, *b*, *x* } and { *z*, *w* }. The figure below uses the subscript of the block for the local variables with the same name to distinguish them from each other.



- b. [5 points] In order to maintain the size and indexing information (lower bound index value and upper bound index value) the compiler needs to create at the activation record level a dope vector for v . Given that it needs to maintain the lower bound, upper bound, size and first address, the size of this vector is known at compile time and needs to be filled as part of the prologue sequences of instruction of the blocked scope it is active in.



Problem 2. Control-Flow Analysis [45 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	14:	goto L5
02:	t0 = *t0	15: L2:	t1 = t1 - 1
03:	t2 = \$fp - 8	16:	if t0 > 16 goto L3
04:	t2 = *t2	17:	goto L4
05:	t1 = 0	18: L3:	t0 = t0 + 1
06:	if t0 < 16 goto L2	19:	if t0 < 16 goto L1
07:	goto L1	20:	goto L2
08: L1:	t1 = t1 + 1	21: L4:	goto L6
09:	t2 = 4 * t1	22: L5:	t1 = 1
10:	t3 = \$gp + t2	23:	goto L6
11:	t4 = *t3	24: L6:	t0 = \$fp - 4
12:	t5 = t4 / t1	25:	*t0 = t1
13:	if t5 >= 0 goto L3	26:	return

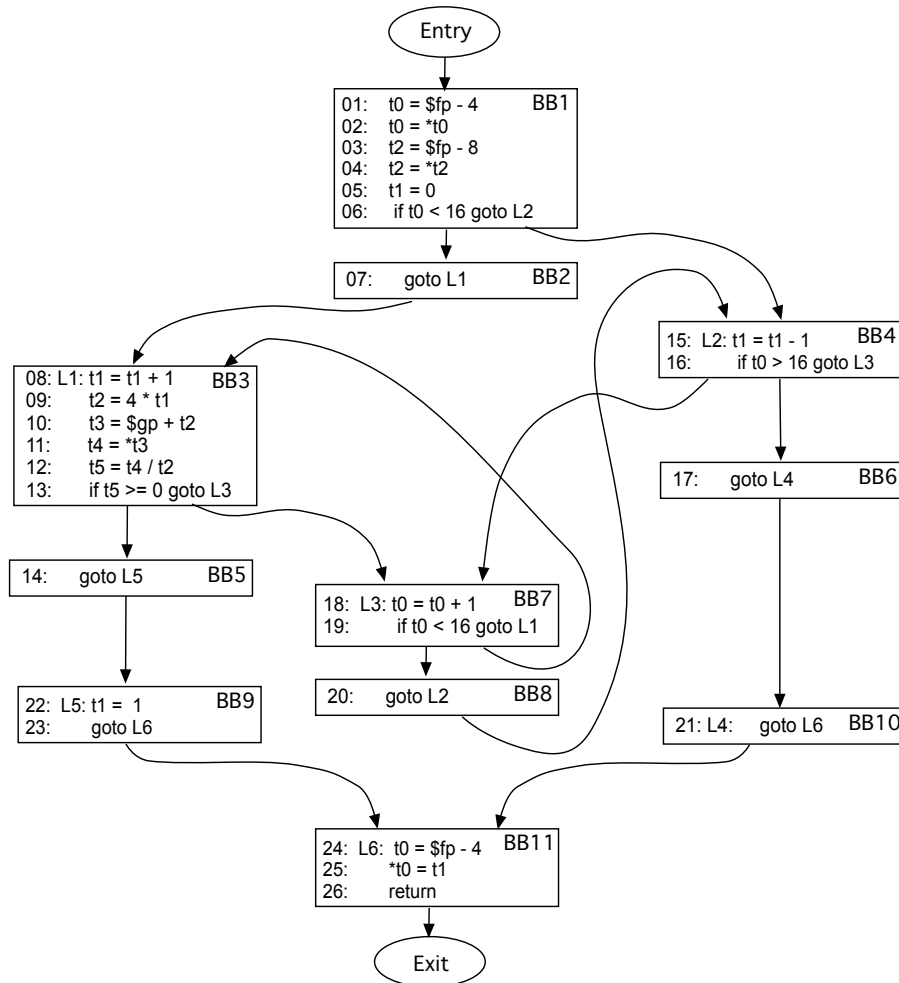
Questions:

For this code determine the following:

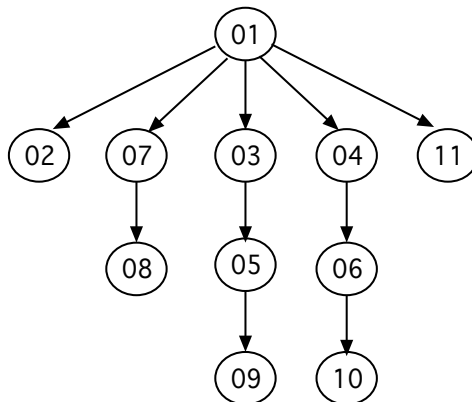
- [10 points] Basic blocks and the corresponding control-flow graph (CFG) indicating for each basic block the corresponding line numbers of the code above.
- [10 points] Dominator tree and the natural loops in this code (if any) along with the corresponding back edge(s).
- [10 points] Determine the live 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.
- [05 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.
- [10 points] Can you color the resulting interference graphs with 2 colors? Why or why not? Present a coloring assignment for 3 colors. If, however, you were to use only two registers describe where spill code could be included.

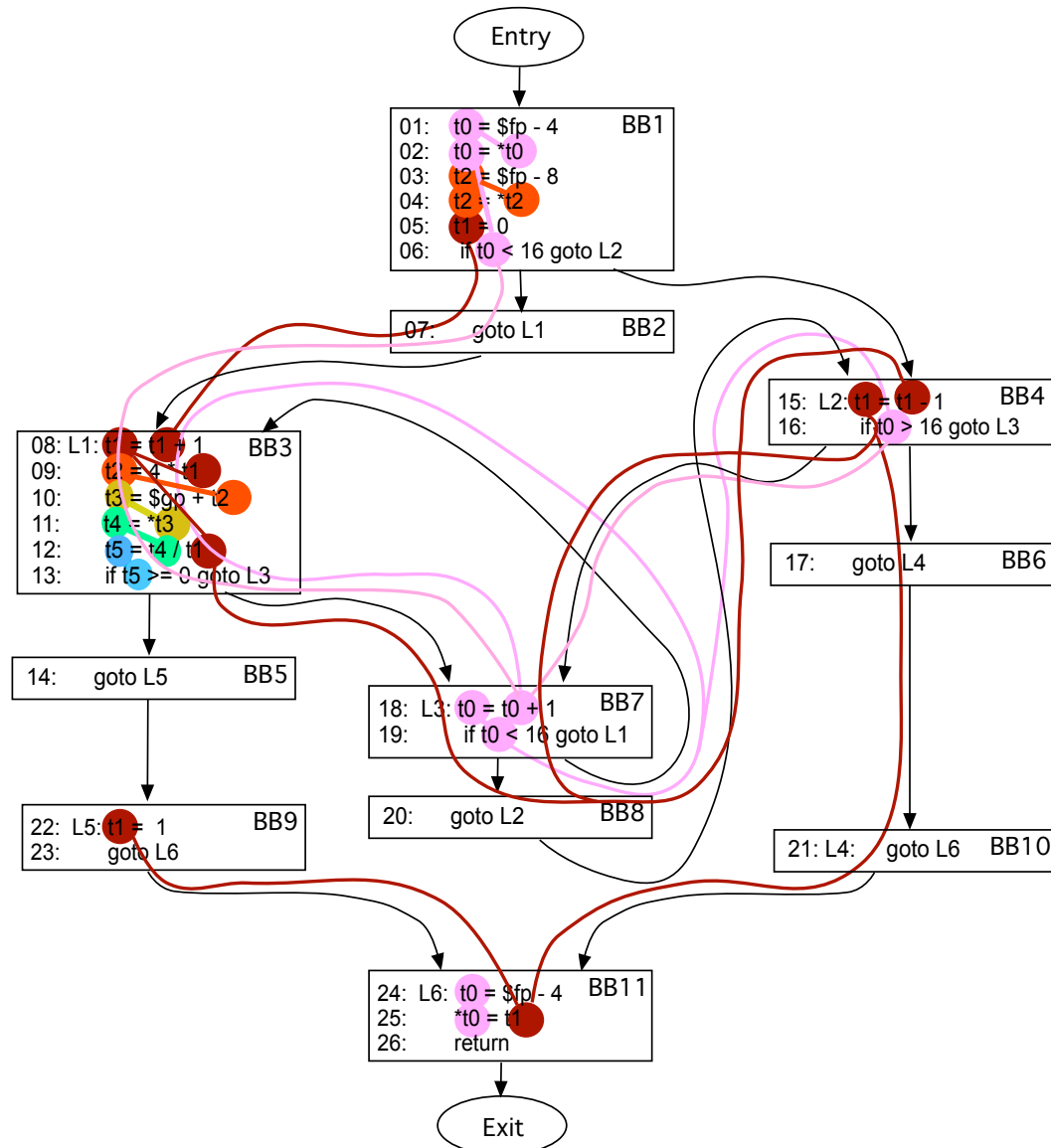
Answers:

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



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





- c. [10 points] Only the temporary variables t_0 and t_1 have long-lived ranges as shown below. The variables t_2 , t_3 , t_4 , and t_5 are used mostly in basic block BB3 and in the case of t_2 only in basic block BB1 where the values of t_2 are 'dead'. Below is the list of individual webs for the distinct values.

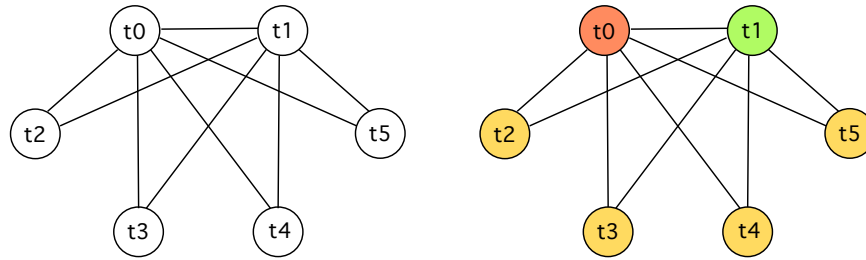
```

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 = {11,12}
t5 = {12,13}

```

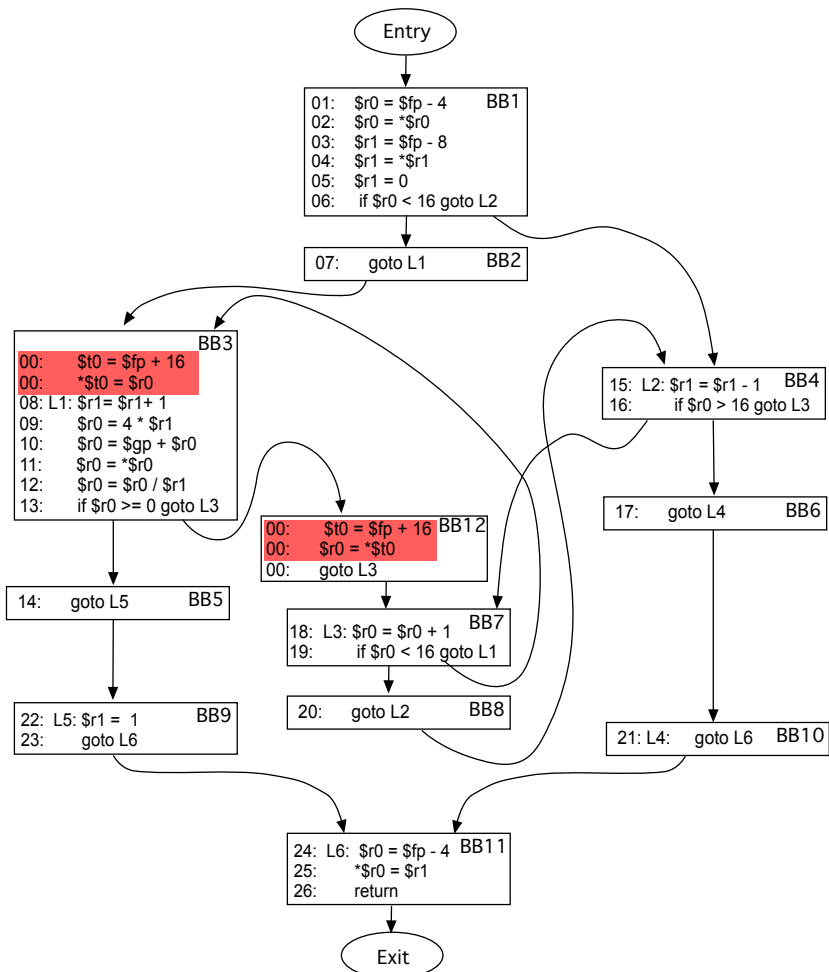
As can be observed none of the temporary variables t_2 , t_3 , t_4 , and t_5 interfere.

- d. [05 points] The interference graph is as shown below (left). The webs for the variables t_1 and t_2 interfere in lines 08 and 09 only as the value of t_1 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 3 in this graph we cannot color it with less than 3 colors. As it is apparent, t_0 and t_1 will have two colors and the remainder nodes will all share the same color, hence the same register.

In addition, and as the web for t_0 has a 'long' hiatus where the value of t_0 is not being used in BB3, I would include a spill code for t_0 at the beginning of BB3 restoring it on exit of BB3. This in itself is a bit of a complication given that t_5 is being used for the conditional test. As a result we would have to create a new basic block just for the instruction that restores t_0 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 $\$t_0$ not used for the 'regular' computation.



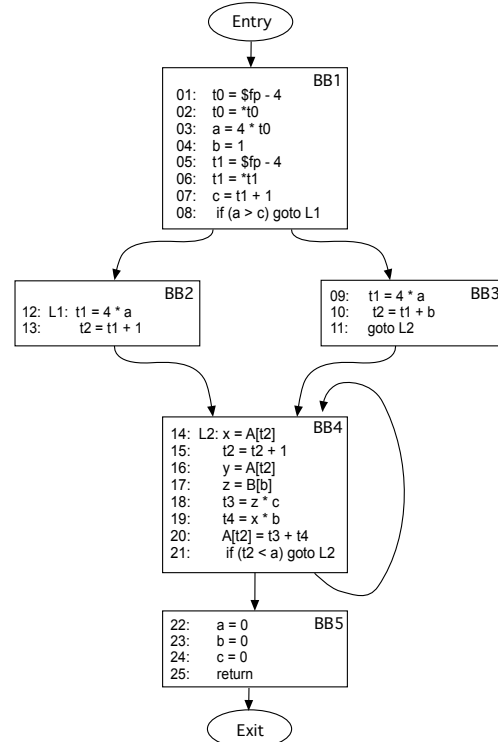
Problem 3. Code Optimization [25 points]

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

```

**Questions:**

- [15 points] Identify opportunities for the application of constant propagation, common-sub-expression elimination (CSE) algebraic simplification, strength reduction.
- [10 points] Identify and exploit opportunities for loop invariant code motion (LICM) identifying basic induction variables of the loop(s) in this code.

Answers:

- [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 propagated to line 10 (as BB1 dominates BB3) and also to line 10 as b is a loop invariant expression. There is the simple opportunity for strength-reduction by having $4 * a$ replaced by $(a < 2)$ in lines 09 and 12.
- [10 points] The computation $B[b]$ which is the same as $B[1]$ is loop invariant. Regarding induction variables there is the statement $t2 = t2 + 1$ which is the single assignment to $t2$ and 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.

Problem 4: Iterative Data-Flow Analysis [35 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
```

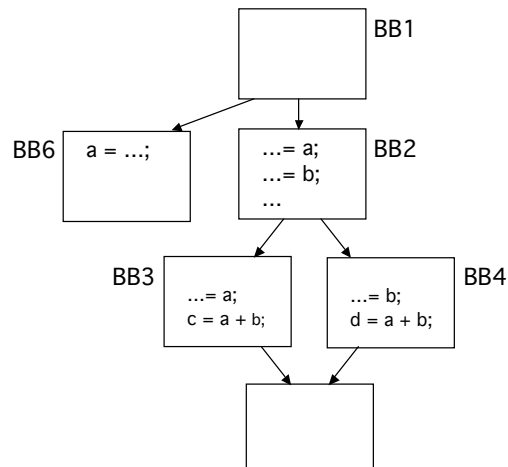
The cost of this transformation is having to store the value in a register for a longer period of time; the benefit is that the program is sped up if the condition $(a > 0)$ holds.

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.



Name: _____

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. [10 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. [10 points] How do you construct the transfer function of a basic block?

Answers:

- 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. [10 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. [10 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.