CSCI 565 - Compiler Design

Spring 2013

Second Test - Solution

May 1, 2013 at 3.30 PM in Room RTH 115

Duration: 2h 30 min.

Please label all pages you turn in with your name and student number.

Name: ______________________________ **Number:** ______________

Grade:

Problem 1 [20 points]:

Problem 2 [35 points]:

Problem 3 [25 points]:

Problem 4 [20 points]:

Total:

Instructions:

1. This is a closed book Exam.
2. The test booklet contains four (5) pages including this cover page.
3. Clearly label all pages you turn in with your name and student ID number.
4. Append, by stapling or attaching your answer pages.
5. Use a black or blue pen (not a pencil).

Problem 1. Run-Time Environments and Data Layout [20 points]

Consider the following PASCAL source program shown below.

```
01: program main(input,  output);
02:      var track[0..3]: integer;
03:      var id: integer;
04:     function P1(a: integer) : integer;
05:       var cnt : integer;
06:       begin (* P1 *)
07:          id := a;
08:          if(a > 0)
09:            cnt := P1(a-1);
10:         else
11:            track[id] = a;
12:         P1 := cnt;
13:       end;
14:      procedure P2(b: integer);
15:       begin (* P2 *)
16:         id := 0;
17:         P1(b)
18:       end;
19:      procedure P3(c: integer);
20:        begin (* P3 *)
21:          id := 3;
22:          P2(c)
23:        end;
24:     procedure P4(d: integer);
25:       begin (* P4 *)
26:         id := 4;
27:         P3(d);
28:       end;
29:  begin (* main *)
30:    P4(1)
31:  end.
```

Questions:

a. [05 points] Show the call tree for this particular program and discuss for this particular code if the Activation Records (ARs) for each of the procedures P1 through P4 can be allocated statically or not. Explain why or why not.

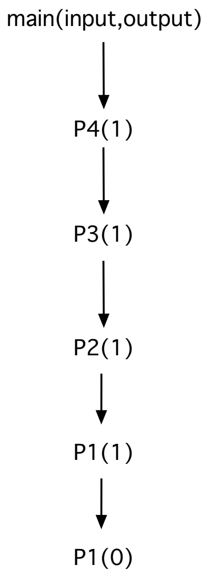
b. [10 points] Assuming you are using a stack to save the activation records of all the function's invocations, draw the contents of the stack when the control reaches the line in the source code labeled as "11+" i.e., before the program executes the return statement corresponding to the invocation call at this line. For the purpose of indicating the return addresses include the designation as "N+" for a call instruction on line N. For instance, then procedure P2 invokes the procedure P1 in line 17, the corresponding return address can be labeled as "17+" to indicate that the return address should be immediately after line 17. Indicate the contents of the global and local variables to each procedure as well as the links in the AR. Use the AR organization described in class indicating the location of each procedure's local variable in the corresponding AR.

c. [05 points] For this particular code do you need to rely on the *Access Links* on the AR to access non-local variables? Would there be a substantial advantage to the use of the *Display* mechanism?

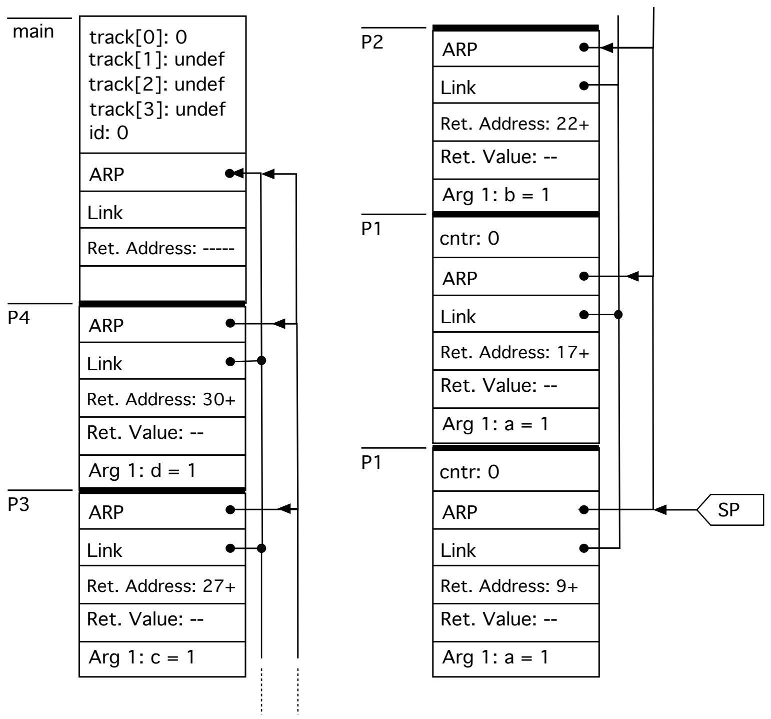
Answers:

a. [05 points] Given that there are recursive function calls (in this particular case P1), we cannot in general allocate the activation records of all these functions in a global region of the storage. However, in this particular case we observe that the recursive calls are confined to a very small portion of the call graph as P1 only calls itself. As such we could in principle allocate all the remainder procedures statically and use a simpler stack for the activations of P1.

b. [10 points] The figure below depicts both the call tree and the stack configuration when the execution reaches the line 11+ in the source program.



(a) Call Tree



(b) Stack Layout

c. [05 points] Given that these are only accesses to local variables within each procedure or global variables (which are allocated in a specific static data section) and there are no accesses to other procedure's local variables, there is no need to use the Access Links (*Access*) in the Activation Records (*AR*) and hence no need to use and maintain the display access mechanism.

Problem 2. Control-Flow Analysis [35 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:         t1 = t0
04:         t0 = $fp - 8
05:         t0 = *t0
06:         t2 = t0
07:         t3 = 0
08:         t4 = 0
09:         t5 = t1
10:  L0:    if t2 = 16 goto L2
11:         goto L1
12:  L1:    if t1 <= 0 goto L2
13:         t2 = t2 + t1
14:         t3 = 4 * t2
15:         t4 = t3 - 1
16:         goto L4
17:  L4:    t6 = 0
18:         t0 = $fp - 12
19:         t0 = *t0
20:         if t0 > 0 goto L3
21:         t5 = t5 + 1
22:         *t5 = 0
23:         goto L5
24: L3:     t6 = 1
25: L5:     goto L0
26: L2:     t0 = $fp - 16
27:         *t0 = t4
28:         t0 = $fp - 20
29:         *t0 = t3
30:         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.

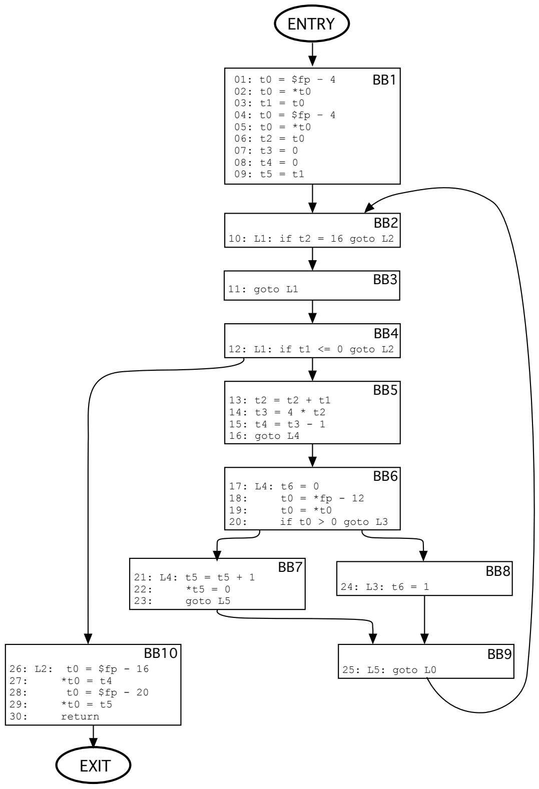
b. [10 points] Dominator tree and the natural loops in this code along with the corresponding back edge(s).

c. [15 points] Loop invariants instructions and induction variables for the loops identified in (b). Include in your answer elements from Data-Flow Analysis (such as reaching definition or live variable analysis) that could help a compiler to identify these opportunities, i.e., explain how a compiler could make use of the information uncovered by those analyses to determine that these specific cases of instructions are loop invariant and that these specific variables are induction variables.

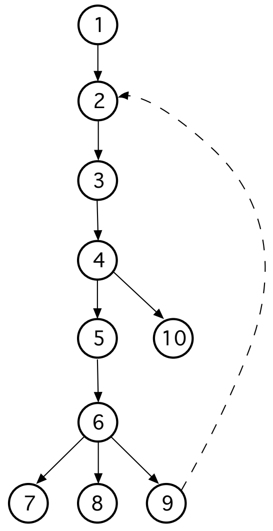
Answers:

a. [10 points] See figure below on the left.

b. [10 points] See figure below on the right.



```
01: t0 = $fp - 4
02: t0 = *t0
03: t1 = t0
04: t0 = $fp - 4
05: t0 = *t0
06: t2 = t0
07: t3 = 0
08: t4 = 0
09: t5 = t1

10: L1: if t2 = 16 goto L2

11: goto L1

12: L1: if t1 <= 0 goto L2

13: t2 = t2 + t1
14: t3 = 4 * t2
15: t4 = t3 - 1
16: goto L4

17: L4: t6 = 0
18:     t0 = *fp - 12
19:     t0 = *t0
20:     if t0 > 0 goto L3

21: L4: t5 = t5 + 1
22:     *t5 = 0
23:     goto L5

24: L3: t6 = 1

25: L5: goto L0

26: L2:  t0 = $fp - 16
27:     *t0 = t4
28:      t0 = $fp - 20
29:     *t0 = t5
30:     return
```



back edge = { (9, 2) }
loop = { 2, 3, 4, 5, 6, 7, 8, 9 }

a) Control-Flow Graph (CFG)

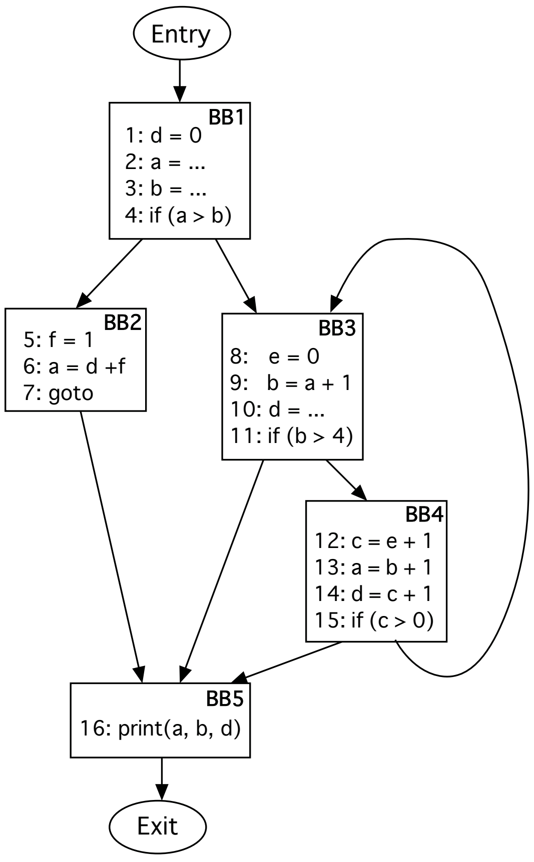
b) Dominator Tree, Back-edge and Natural Loop

c. [15 points] There are two potential cases of loop invariant code in lines 17 and 24 regarding variable t6 as in these statements the values assigned to t6 are constant. There are however two assignment in two different basic blocks neither of which post-dominates the exit of the loop. As such we cannot move the assignment "t6 = 0" (in the block that dominates the second assignment) to the header of the loop. Notice that in fact these two assignments are in fact "dead" instructions, as t6 is never used in this code.

As to induction variables we observe the sequence in instructions 13, 14 and 15. In instructions 13 "t2 = t2 + t1" the use of t1 is in fact loop invariant - the value of t1 does not change within the loop. This means that from the perspective of the loop, t1 is a constant and thus t2 is a basic induction variable. Subsequently, t3 and thus t4 will be derived inductions variables with coefficient tuples (t2, 4, 0) and (t2, 4, -1) respectively.

Problem 3. Register Allocation [25 points]

Consider the following control flow graph (CFG) fragment below where it is assumed all scalar variables a through f are dead on procedure exit.



Questions:

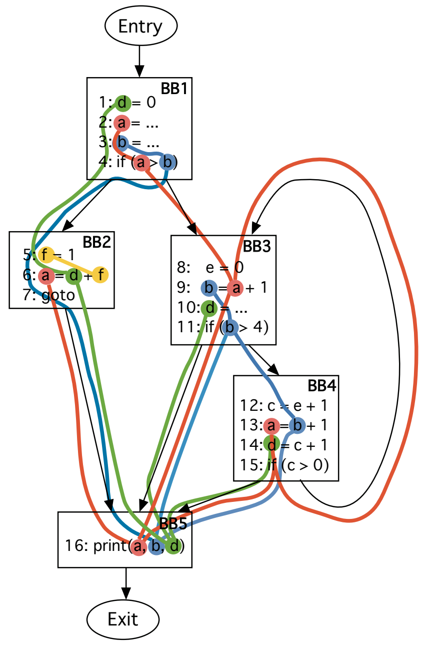
a. [10 points] Determine the live ranges and the corresponding webs for the variables a, b, c, d, e and f. Show the webs for the variables defined in terms of the line numbers depicted next to each instruction. Explain the way you combine the *def-use* chains for the variable 'a' to form the web for that variable. You do not need to be as specific for the other variables, so present only the corresponding webs.

b. [05 points] Derive the interference graphs (or table) for these variables using the refined interference described in class (this is the "second" more sophisticated definition that takes into account the read and write of each specific variable/temporary). Explain in detail the interference (or lack thereof) between the webs corresponding to the variables 'a' and 'f'.

c. [10 points] Can you color the resulting interference graphs with 3 colors? Why or why not? Use the graph-coloring algorithm described in class for coloring this graph and assign specific registers to the various webs. Describe where spill code would have to be included and why so that you could color the graph with 3 colors using the same algorithm (not by manual inspection).

Answers:

a. [10 points] The graph below illustrates the various live ranges for the variables 'a', 'b', 'd' and 'f'. On the right-hand-side we have the live ranges for each variable defined in terms of the line numbers and the corresponding interference graph for the webs that can be generated from these ranges. For variable 'a' there are some ranges, namely: range #1 = {2,3,4,8,9,10,11,16} as this corresponds to a definition in line {2} that propagates to the many uses in lines {4}, {9} and {16}; range #2 - {13, 14, 15, 16} as this corresponds to the definition in line {13} that propagates directly to line {16} and then range #3 = {13, 14, 15, 8, 9} as this corresponds to the definition in line {13} that propagates along the back edge of the loop to the uses in 9 (and eventually to 16).

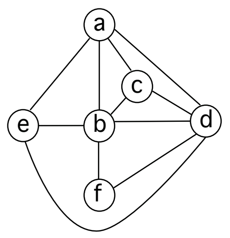


```
1: d = 0
2: a = ...
3: b = ...
4: if (a > b)

5: f = 1
6: a = d + f
7: goto

8:  e = 0
9:  b = a + 1
10: d = ...
11: if (b > 4)

12: c = e + 1
13: a = b + 1
14: d = c + 1
15: if (c > 0)

16: print(a, b, d)
```

Live Range Webs for Variables 'a', 'b', 'd' and 'f'

Var	Def-Use Chains
a	{2, 3, 4, 8, 9, 10, 11, 16} ; {13, 14, 15, 16} ; {13, 14, 15, 8, 9 }
b	{3, 4, 5, 6, 7, 16} ; {9, 10, 11, 16} ; {9, 10, 11, 12, 13, 14, 15, 16}
c	{12, 13, 14, 15}
d	{1, 2, 3, 4, 5, 6, 7, 16} ; {10, 11, 16} ; 14, 15, 16}
e	{8, 9, 10, 11, 12}
f	{5, 6}

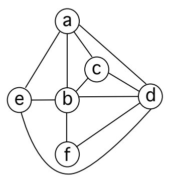
DU chains for variables 'a' through 'f'



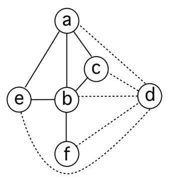
Interference graph

b. [05 points] This is shown above on the right-hand side (lower side). Notice that the web associated with variable

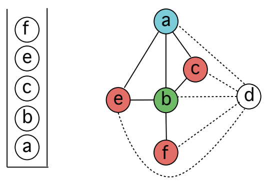
c. [10 points] No, we cannot color this graph with 3 colors as the subset of nodes in this graph corresponding to the variables {a,b,c,d} form a 4-clique. As such we remove the node 'd' from the interference graph, as this is the node with the largest degree. We then use the heuristic graph coloring algorithm described in class by pushing onto a stack the nodes in the sequence as shown below and proceed to color them as they are being "popped" out of the stack. The assignment of color is as shown to the right.



(a) Original Interference graph



(b) Removing node with the highest degree



(c) Pushing order into the stack: a, b, c, e, f and coloring during 'pop' operations using 'red', 'green' and 'blue'.

Name: ________________

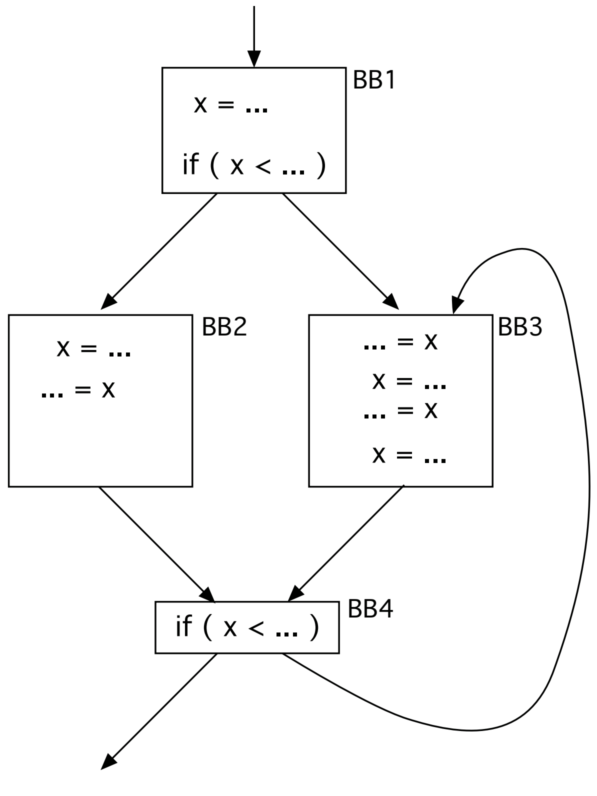
Problem 4: Iterative Data-Flow Analysis [20 points]

There are several compiler passes that rely on the information about which variables are defined and used. Register allocation is such a case whose information can be derived directly from live variable analysis or by *def-use* (defined-used) chains. In this problem you are asked to describe the *def-use* (DU) data-flow analysis in detail and discuss its use to derive information for program enhancing transformations.

Questions:

a. [10 points] Formulate the *def-use* iterative data-flow analysis problem indicating the structure of the lattice, the meet operator; the transfer functions and the initialization values for the nodes in the program assumed to be the nodes in the CFG corresponding to the program's basic blocks. Justify the choice of initial value in terms of precision and safety of the initial and the resulting final solution it leads to.

b. [10 points] Using the formulation you have developed in a. above apply it to the CFG structure depicted below where you can assume that the initial values for the data-flow abstractions are empty and that they do not change over the various iterations of your algorithm. Moreover, assume there are no definitions to the x variable other than the ones depicted here. Show the intermediate values of the IN and OUT abstractions for each basic block.



Answers:

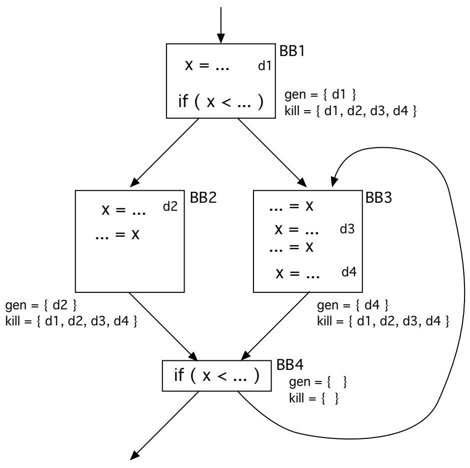
a. [15 points] We use the same formulation as described in class using the common abstractions of In and Out to represent the definitions that reach the input and the output of each basic block in the program's CFG. As such the lattice will consist of the set of definitions for each variable and it closed under the subset relationship. The top element of the lattice (T) is the set of all definitions and the bottom element (⊥), and safest, is the empty set. The Gen set for each basic block is the set of definitions that are downward exposed or downward available and correspond to the definition set in the current basic block for variables that are not later redefined in the same basic block. The Kill set for each basic block corresponds to the definition whose variables are defined in the basic block. Note that the Kill set does include definition in other basic blocks as well as the definition in the basic lock to which it corresponds. The meet operator is in this case the set union and the transfer function for the basic block is defined by the equation

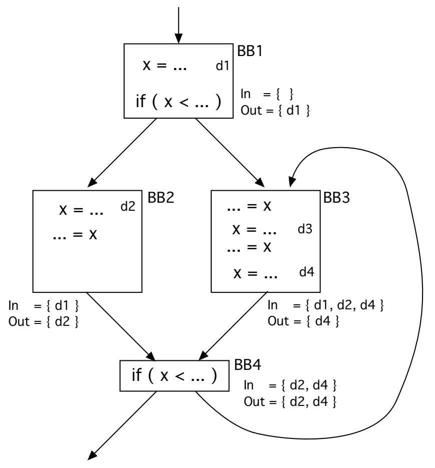
$$\text{In}(n) = \bigcup \text{Out}(p) \text{ for all predecessors nodes p of n}$$
$$\text{Out}(n) = \text{Gen}(n) \cup (\text{In}(n - \text{Kill}(n)))$$

The lattice being the power-set of the definition in the program is of finite height (as well as width) and the set-union is commutative, distributive and associative. This means that not only does the iterative work-list algorithm does converge, but that also the MOP is the same solution as the MFP computed by this algorithm.

As to the initialization, all sets In and Out should be initialized to empty sets. This is the safest assumption to the reaching definitions where we claim that no definition reaches any point of the program.

b. [10 points] The figure below depicts the various values for the iterative data-flow analysis algorithm for this DU-chain problem for the inputs CFG.





	Iteration 0		Iteration 1		Iteration 2		Iteration 3	
BB	In	Out	In	Out	In	Out	In	Out
1	{ }	{ }	{ }	{ d1 }	{ }	{ d1 }	{ }	{ d1 }
2	{ }	{ }	{ d1 }	{ d2 }	{ d1 }	{ d2 }	{ d1 }	{ d2 }
3	{ }	{ }	{ d1 }	{ d4 }	{ d1 }	{ d4 }	{ d1, d2, d4 }	{ d4 }
4	{ }	{ }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }

Order = BB1 BB2, BB3, BB4