CSCI 565 - Compiler Design

Spring 2012

Second Test - Solution

April 25, 2012 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 [20 points]:

Problem 4 [25 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 p[1..2]: integer;
03:     var z: integer;
04:    procedure P1(a: integer);
05:      begin
06:         z := a+1;
07:         writeln(p[z])
08:      end;
09:     procedure P2(b: integer);
10:      begin (* P2 *)
11:        z := 0;
12:        P3(b)
13:      end;
14:    procedure P3(c: integer);
15:     begin (* P3 *)
16:       p[c] := z;
17:       P1(0);
18:     end;
19: begin (* main *)
20:   P2(1)
21: end.
```

Questions:

a. [05 points] Discuss for this particular code if the Activation Records (ARs) for each of the procedures P1 through P3 can be allocated statically or not. Explain why or why not.

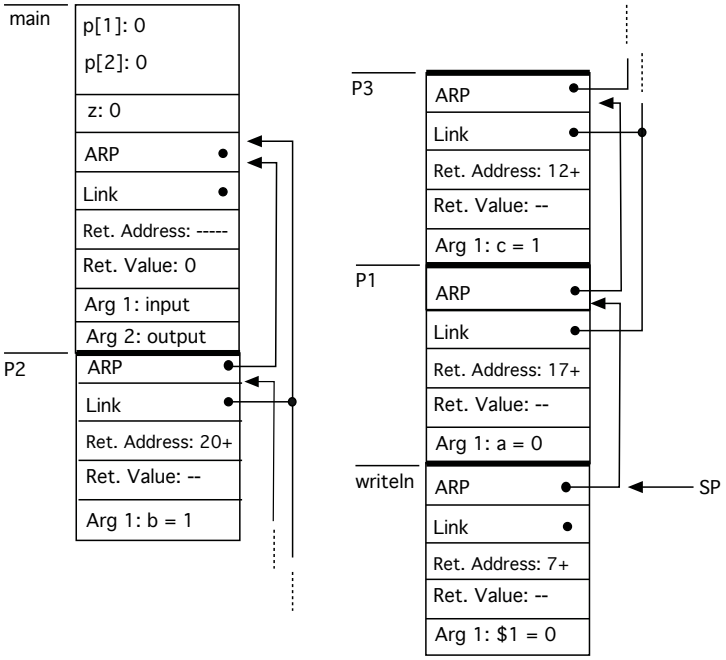
b. [10 points] Assuming you are using a stack to save the activation records of all the function's invocations, draw the contents of the stack when the control reaches the line in the source code labeled as "7+" i.e., before the program executes the return statement corresponding to the invocation call 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 P3 in line 12, the corresponding return address can be labeled as "12+" to indicate that the return address should be immediately after line 12. 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 thee be a substantial advantage to the use of the *Display* mechanism?

Answers:

a. [05 points] Given that there are no recursive function calls, we can allocate the activation records of all these functions in a global region of the storage.

b. [10 points] See stack layout below.



c. [05 points] Given that all the procedures are nested within the main procedure (as in C) there is really no performance advantage in the use of *Displays*. In fact, there is no need to use the *Display* at all given that in addition to the local variables (inside each procedure) there are no variables hidden in enclosing procedures.

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:          t2 = 0
08:          t5 = 0
09:          t6 = 0
10:  L0:     if t2 = 16 goto L2
11:          goto L1
12:  L1:     if t1 > 0 goto L3
13:          t2 = t2 + 2
14:          goto L4
15:  L3:     t2 = t2 - 1
16:          goto L4
17:  L4:     nop
18:          t0 = $fp - 12
19:          t0 = *t0
20:          t4 = t2 * 4
21:          t0 = t0 + t2
22:          t3 = *t0
23:          if t3 <= t5 goto L5
24:          t5 = t3
25:          t6 = t2
26:  L5:     goto L0
27:  L2:     t0 = $fp - 16
28:          *t0 = t6
29:          t0 = $fp - 20
30:          *t0 = t5
31:          return
```

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.

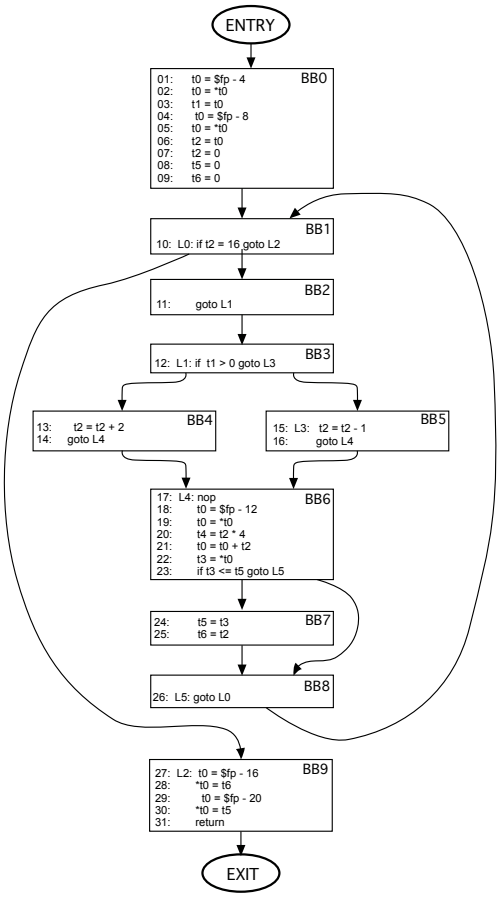
b. [10 points] Dominator tree and the natural loops in this code along with the corresponding back edge(s).

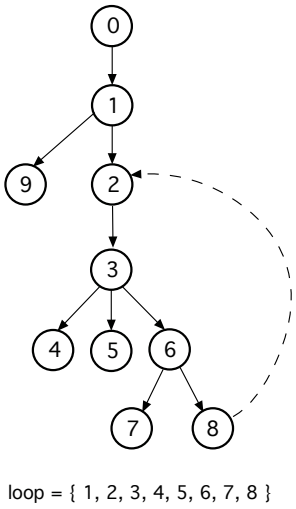
c. [10 points] Loop invariants instructions and induction variables for the loops identified in b.

d. [05 points] Suggest code transformations that will increase the number and type of loop invariant and loop induction variables in the transformed code.

Answers:

a. [10 points] See figure below (left).
b. [10 points] See figure below (right).





c. [10 points] Variable t1 and consequently the predicate (t1 > 0) are loop invariant, which means that only one of the branch target is taken, but it is unknown which one is taken. The computation in lines 18 and 19 is also loop invariant as it corresponds to the value of an argument of the function.

There is a single potential loop induction variable, t2, but it is incremented by 2 and decremented by 1 depending on the direction the loop is taken, so unless there is a transformation to the code this is really not an induction variable.

d. [05 points] The natural code transformation is to replicate the body of the loop for the two branches of the loop in essence moving the predicate in line 12 out of the loop and creating two loops. There will be no control-flow inside each of these loops and then there will be an induction variable for each of these loops, one variable with increment 2 and another with a decrement 1.

Problem 3. Register Allocation [20 points]

Under the liveness assumptions stated above answer the following questions:

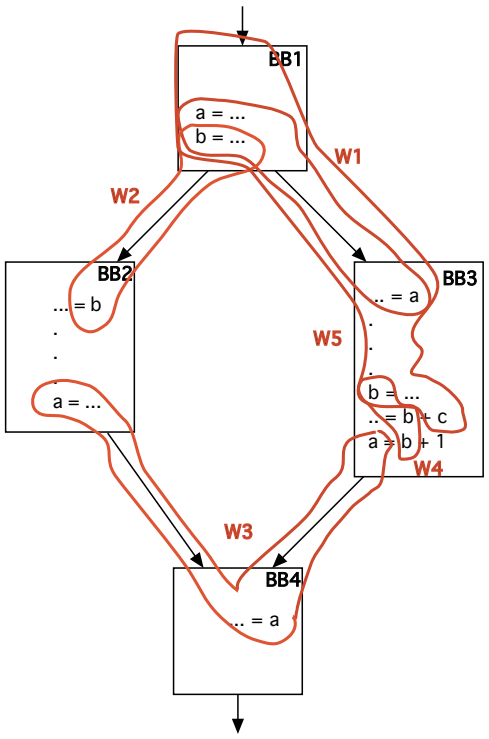
a. [10 points] Determine the live ranges and the corresponding webs for the variables a, b and c.

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).

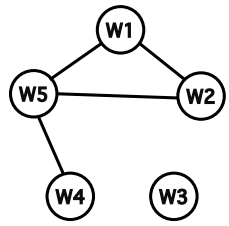
c. [05 points] Can you color the resulting interference graphs with 2 colors? Why or why not? If not suggest a way to split one of the webs so that the resulting interference graph is 2-colorable.

Answers:

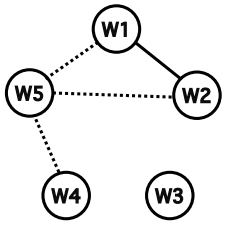
a. [10 points] See figure below (left).
b. [05 points] See figure below (right).



Live Ranges and Webs



Interference graph
(requires 3 colors)



Interference graph
removing register
for variable c
(requires 2 colors)

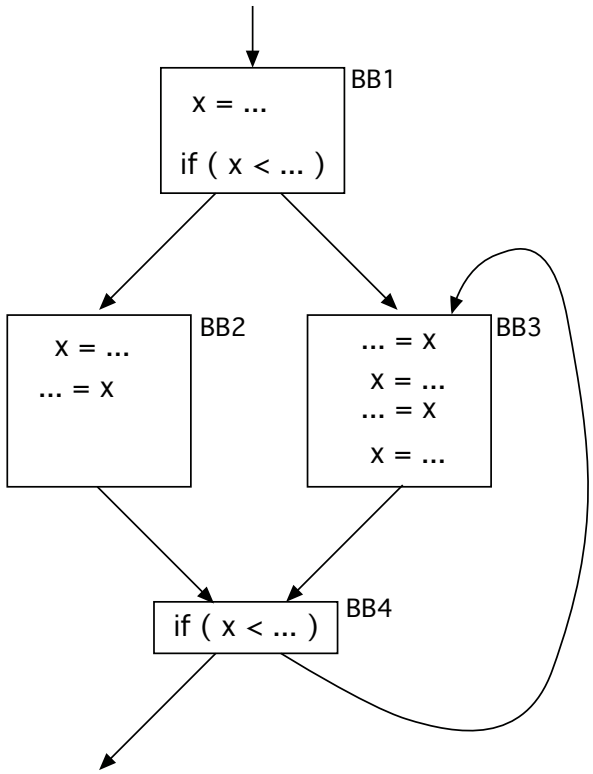
c. [05 points] The initial interference graph shown on the right (top) has a clique of size 3, so a 2-coloring is not possible. A simple strategy is to remove the web associated with variable c as this one has the longest un-used span of its live range. In terms of interference graph this means that we simply remove the edges associated with node W5 resulting in a simplified interference graph as shown on the right (bottom), which can be trivially colored using 2 colors.

Problem 4: Iterative Data-Flow Analysis [25 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. Specifically address the following questions.

a. [15 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

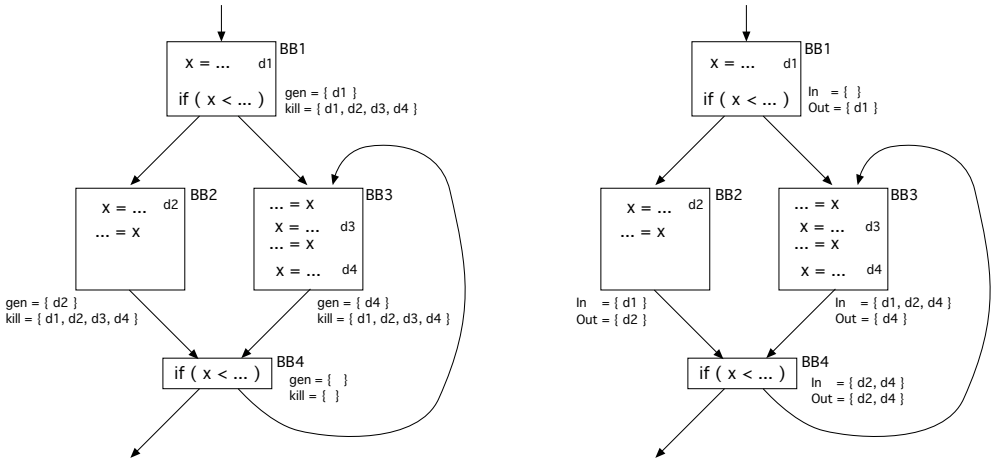
set for each basic block is the set of definitions that are downward exposed or downward available and correspond to the definition set in the current basic block for variables that are not later redefined in the same basic block. The Kill set for each basic block corresponds to the definition whose 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

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

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.



BB	Iteration 0 In	Iteration 0 Out	Iteration 1 In	Iteration 1 Out	Iteration 2 In	Iteration 2 Out	Iteration 3 In	Iteration 3 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