Compilers (L.EIC026)

Spring 2022

Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Second Test

June 17, 2022 from 14:30 to 16:30

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

Name: _____ Number: _____

Grade:

Problem 1 [10 points]:
Problem 2 [40 points]:
Problem 3 [20 points]:
Problem 4 [30 points]:

Total:

INSTRUCTIONS:

1. **This is a close-book and close-notes exam**.
2. Please identify all the pages where you have answers that you wish to be graded. Also make sure to label each of the additional pages with the problem you are answering.
3. Use black or blue ink. No pencil answers allowed.
4. Staple or bind additional answer sheets together with this document to avoid being misplaced or worse, lost. Make sure this cover page is stapled at the front.
5. Please avoid laconic answers so that we can understand you understood the concepts being asked.

Problem 1. Run-Time Environments [10 points]

Many modern imperative programming languages support the abstraction of procedures and functions. To support the execution of procedures, compilers make use of activation records or AR that capture data related to the execution of procedures and functions. In this context, answer the following questions:

a. [05 points] What information is typically stored at run-time and why?
b. [05 points] Where are ARs allocated and why?

Answers:

a. The Activation Records tracks the execution environment of a procedure (or function) and thus must capture among other value, the return address and the frame pointer of the enclosed procedure (i.e., the procedure that immediately called the currently-executing procedure). In addition, and for procedures that allow for nested procedures (as is the case of PASCAL), the activation record also capture the Access Link, which allows an executing environment to access non-local variables, i.e., variables that are local to other active procedures (elsewhere on the stack) but are visible from a scoping standpoint. In addition, the activation records also capture the usual actual parameters of the procedure as local variables.

b. For programming languages that allow for recursion, the activation records are "allocated" on the stack as their activation and deactivation follows a LIFO order. For older languages where at most a single activation of a given procedure is possible, there is no need to use a stack "allocation" and instead a single static location can be used for each activation of a procedure. Notice however, that a stack is still needed (possibly spread across the static locations) to save and restore the return address of each procedure.

Problem 2. Control-Flow Analysis and Register Allocation [40 points]

Consider the three-address code below for a procedure with input/output arguments `p0` and `p1` and using several temporary variables, named `t0` through `t4`.

```
01:      t1 = 0
02:      t0 = p1
03:      t2 = 0
04:      t1 = p2
05:      if (t1 > 0) goto L1
06:      t1 = t0
07:      t0 = 1
08:      t2 = 0
09: L1:  t1 = t0
10:      t4 = t0
11:      if (t2 > t4) goto L2
12:      t2 = t2 + 1
13:      goto L1
14: L2:  t0 = p1
15:      t3 = t0
16:      t1 = t2 + t3
17:      ret t1
```

Questions:
For this code determine the following:

a. [05 points] Basic blocks and the corresponding control-flow graph (CFG) indicating for each basic block the corresponding line numbers of the code above.

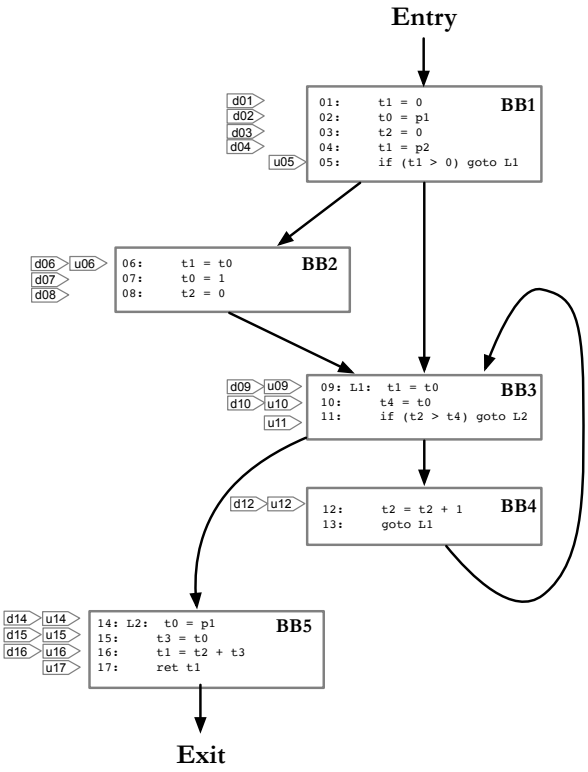
b. [05 points] Dominator tree and the natural loops in this code (if any) along with the corresponding back edge(s).

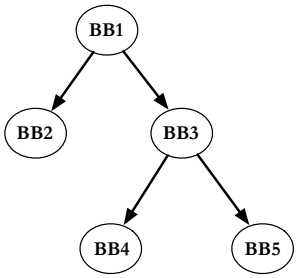
c. [20 points] Determine the live ranges for the variables `t0`, `t1`, `t2`, `t3` and `t4`, the corresponding webs and interference graph, for the refined notion of interference discussed in class. Assume that you do not need registers for the parameters p0 and p1 and assume that on exit of the last basic block (the one ending with the return instruction) `t2` is live but the remainder temporaries are dead on exit of the procedure. In this analysis you should ignore the parameter variables `p0` and `p1`.

d. [10 points] Can you color the resulting interference graphs with 3 colors? If you cannot, suggest a source code transformation that will reduce the connectivity of the corresponding interference graph so that it becomes 3-colorable. If you can use 3 colors, just show a possible color assignment as you do not need to show the coloring that would result if you were to follow the graph coloring algorithm described in class.

Answers:

a) and b) The CFG and the corresponding dominator tree are as shown below. The only back edge, whose "head" dominates its "tail" is edge (3,4) and the corresponding natural loop is composed of the basic blocks 3 and 4.

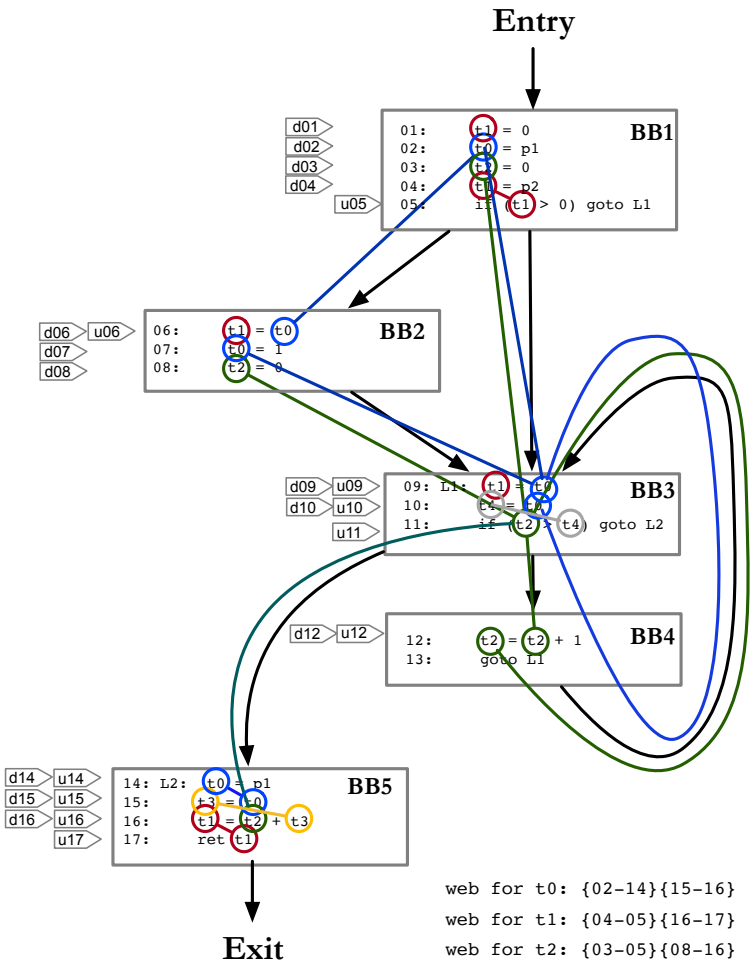




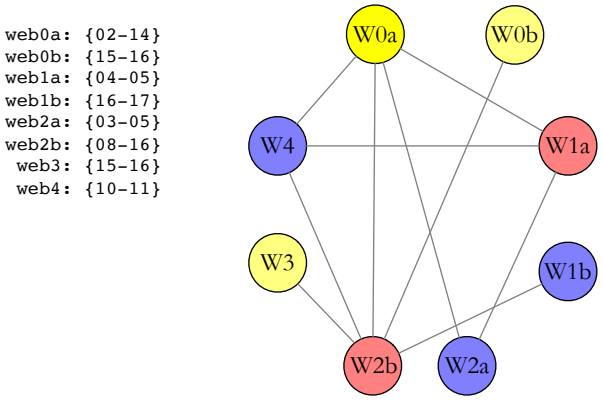
Control-Flow Graph Dominator Tree

This figure also indicates the definition points and use points so that DU-chains, and thus webs, can more easily derived. For instance, variable `t5` is defined at instruction `01`, where we have a designation of `d01`. The particular value is used in instruction at line 11, thus making use of the use `u11`.

c) The live ranges and the corresponding webs are shown below.

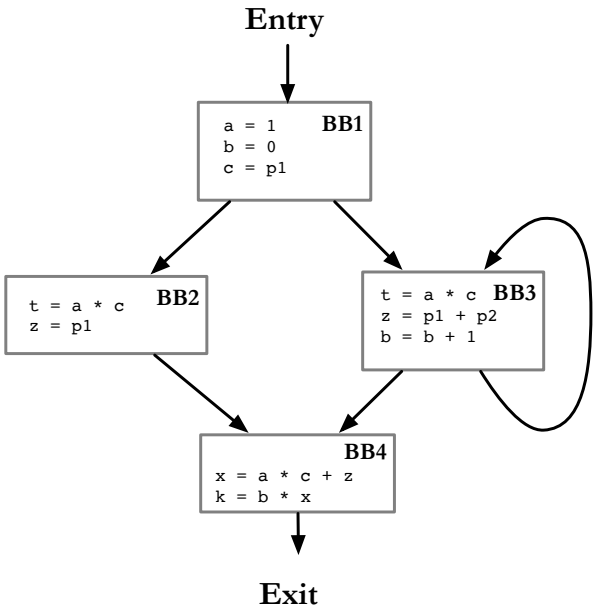


The figure below depicts the interference graph for the temporaries `t0` through `t4` using the refined notion of interference as discussed in class. As it can be seen there are several cliques of size 3, so clearly it can be colored using 3 colors.



Problem 3. Code Optimization [20 points]

Consider the snippet 3-address code below using temporary variables `a`, `b`, `c`, `t`, `z`, `x` and `k` as well as the argument value `p1` and `p2` (used via the corresponding procedure's parameters).



Questions:

a. [10 points] Identify opportunities for the application of constant propagation, available expression and propagation, algebraic simplification and strength reduction.

b. [10 points] Identify opportunities for loop invariant code motion (LICM), as well as induction variables (i.e., variables that increase or decrease in value at every iteration of the loop). Consider also, redundant variable elimination (i.e., variables that can be removed by copy propagation).

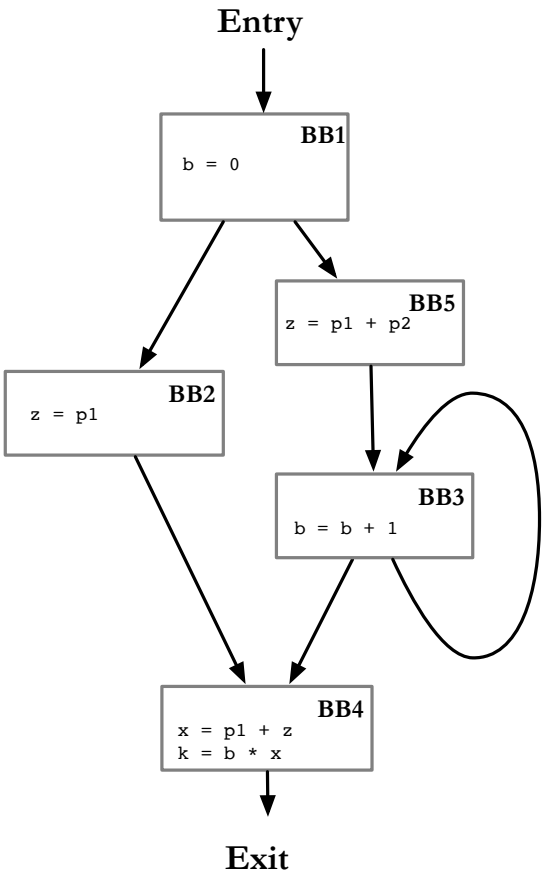
In both cases, reasons about the legality of the code transformation with respect to the control-flow and the notion of dominance.

Answers:

a. [10 points] The assignment to `a` in the very first line is a constant that can be propagated to all its uses, since it is the only assignment to a and in a basic block that dominates all the uses in the other basic blocks. The expression (`a*c`) can therefore be simplified as (`c`), and since all definitions of `c` are in BB1, the value of `p1` can be propagate to all its uses.

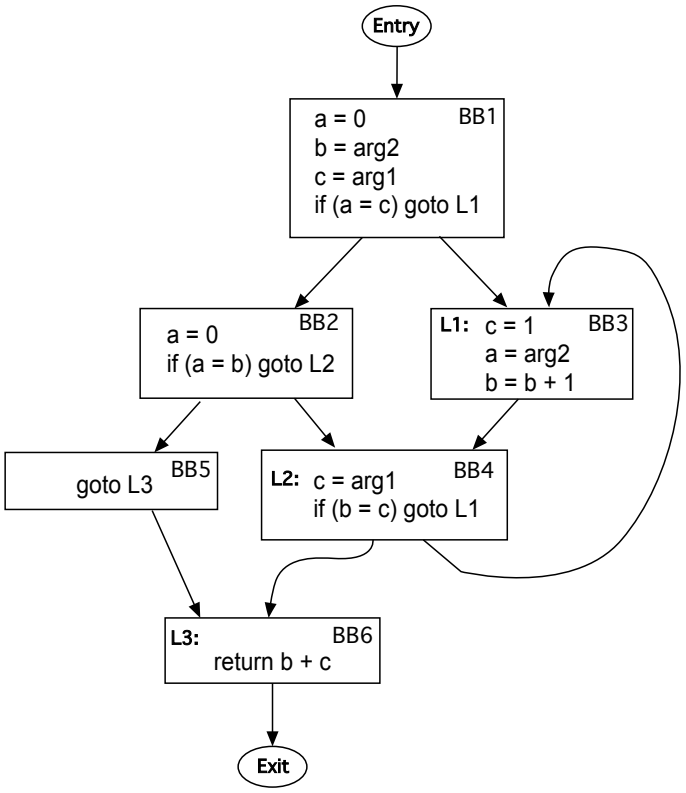
b. [10 points] Clearly, the computation of the expression (`a*c`) inside the loop that consists of basic block BB3 is loop invariant. Similarly, the assignment to `z` is also loop invariant. Both could be moved to a preheader of the loop as shown by `BB5` below. Notice that this is independent of the application of other transformations such as constant propagation as they do not affect by the insertion of the loop's pre-header block. The only induction variable is `b` (which is also called basic induction variable).

The snippet code below illustrates the application of theses transformations.



Problem 4: Iterative Data-Flow Analysis [30 points]

Your task is to formalize and apply the Copy-Propagation data flow analysis for a problem where we want to determine which pairs of variables "bound" by an assignment of the form `a = b` at program point `p` reach a given program point `q` if along any program path neither or `a` nor `b` are redefined. For instance, the assignment `b = arg2` reaches BB2 and thus the predicate (`a = b`) can be rewritten as (`a = arg2`).



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. [05 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?

e. [10 points] Describe transformations and optimizations that can leverage the information uncovered by this analysis <u>in general</u> and <u>in particular to the example code provided</u>. Provide a couple of examples to illustrate your points.

Answers:

a. [05 points] At each program point we will keep track of which pairs of variables are bound to the same value at run time. i.e., we have an unordered set of tuples of the form {u,v}. The initial values will be the empty set.

b. [05 points] The direction of the data flow problem is forward. At each program point we are asking if the binding of two variables is still value. As such as progress forward until there is one assignment to either of the variables that affect each tuple. As such the flow of information is naturally forward along the control flow.

c. [05 points] At the confluence of control-flow paths we intersect the sets as we need to ensure that along all paths the binding of two variables will hold. Regarding the equations they are as shown below reflecting the forward-looking nature of this data-flow problem.

$$OUT = Gen \cup (IN - Kill)$$
$$Gen = \{ \langle v,u \rangle \mid v = u \text{ is the statement } \}$$
$$Kill = \{ \langle v,u \rangle \mid \text{LHS var. of an assignment stmt. is either v or u } \}$$

d. [05 points] The Gen can be accomplished by a single forward pass on the instructions of a basic block where we just keep track at each instructions what variables are assigned on the LHS of the assignment and are not subsequently killed by other assignments. The Kill set is simply the merge of all the variables that show up on the LHS of the assignments.

e. [10 points] This fact that we know that two variables will hold the same value will allow use to replace one of them, in the hope that the corresponding variable (and consequently the instructions that make the assignments, become dead. In addition, and when combined with constant propagation this can lead to even more dead-code elimination. In this particular example the assignment to the c variable is basic block BB1 and BB3 are dead. The only association that reaches the use of c in BB4 for example is the assignment `c = arg1`. As such the predicate in this BB4 can be rewritten as `if (b = arg1) goto L1`. Notice also that the two associations that reach BB6 for `c` are the same {c, arg1} and as a result the return statement can be rewritten as `return b+arg1`. After this the references to `c` can simply be removed from the code