Compilers (L.EIC026)

Spring 2023

Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Second Test with Solution

June 05, 2023 from 17:30 to 19:30

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

Name: ______________________________ **Number:** ________

Grade:

Problem 1 [10 points]:
Problem 2 [20 points]:
Problem 3 [30 points]:
Problem 4 [20 points]:
Problem 5 [20 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:

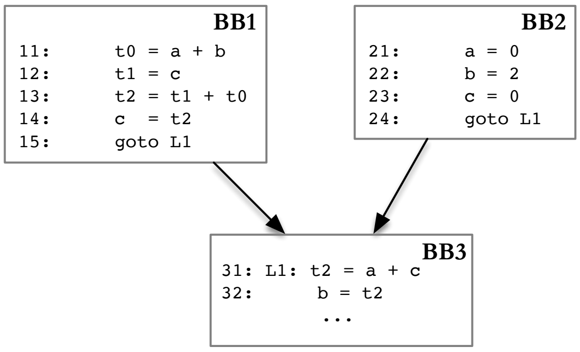
a. [05 points] Why do languages that support recursion need to have their ARs allocated on a stack?
b. [05 points] Where are the procedure local variables allocated in memory. Explain mechanisms to access them at run time and the connection between the information in the symbol table and this mechanism.

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. The local variables are typically saved in specific offset off of the stack point (sp) in the AR. To access them the compiler needs to generate an indirect access where it uses a register (typically, the stack pointer – sp) and uses a constant value off of that point. These offsets are derived by the compiler when parsing the code of the procedure and saved in the symbol table so that each local variable is known by its offset in the current AR. Notice that in the case of recursive function multiple instances of the AR will host the multiple instances of the procedure local variables, corresponding to each active invocation.

Problem 2. Local Register Allocation [20 points]

Consider the snippet of a CFG depicted below corresponding to a procedure that uses 3 procedure local variables, named `a`, `b` and `c` as well as temporary variables `t0` through `t2`. The local variables are located at offsets, respectively, 0, 4, and 8 off of the stack pointer register and the temporary variables need not to be saved on the AR as their values are never reused across basic blocks. In terms of memory operations to load and store the value of each variable, you are expected to use a simple instructions of the form r_x = offset(sp) for a load operation, and offset(sp) = r_x for a store operation. As a specific example, to load variable a into register r0, you would use the instruction r0 = 0(sp).



```
BB1
11:      t0 = a + b
12:      t1 = c
13:      t2 = t1 + t0
14:      c  = t2
15:      goto L1
```

```
BB2
21:      a = 0
22:      b = 2
23:      c = 0
24:      goto L1
```

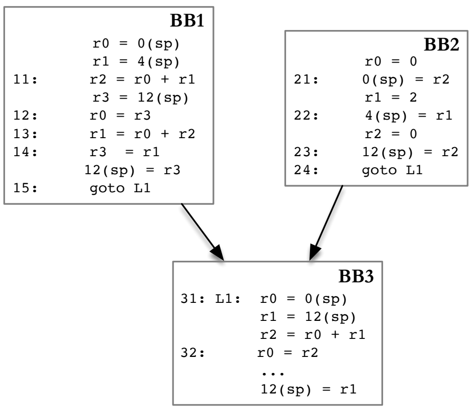
```
BB3
31: L1: t2 = a + c
32:      b = t2
         ...
```

For each basic blocks BB1 through BB3 derive the simple local register allocator using 4 registers, `r0` through `r3` and assuming that at end of every block all modified local variables are to be stored back to the procedure's AR. Comment on the issue of local register allocation across basic blocks. Show how it can be improved in this specific case, by trying to maximize the register reuse across basic blocks so that you can minimize memory operations (loads and stores).

Solution:

The local register allocation, greedily allocates available registers to variables and evicts from registers variables that are going to be used farthest into the future. At the end of each basic block any variable that has been updated needs to be save in memory, in this case in the corresponding slot of the AR.

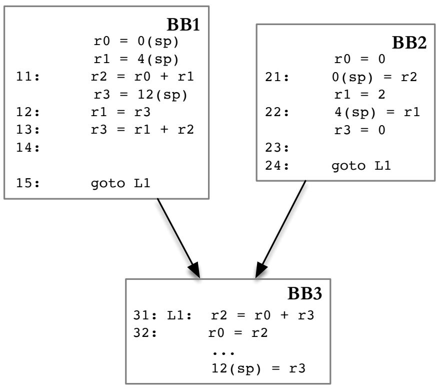
As such, the picture on the left below shows the local register allocation and the various load and store operations. As can be seen at the end of BB1 we have variable `c` allocated in register r3 and on BB2 we have the same variable `c` allocated to register r2. A smatter allocator would attempt to reconcile the allocation of both variables `a` and `c` across basic blocks as shown on the right below, thereby eliminating 4 memory access operations in BB1, BB2 and BB3.



```
BB1
         r0 = 0(sp)
         r1 = 4(sp)
11:      r2 = r0 + r1
         r3 = 12(sp)
12:      r0 = r3
13:      r1 = r0 + r2
14:      r3  = r1
        12(sp) = r3
15:      goto L1
```

```
BB2
         r0 = 0
21:      0(sp) = r2
         r1 = 2
22:      4(sp) = r1
         r2 = 0
23:      12(sp) = r2
24:      goto L1
```

```
BB3
31: L1:  r0 = 0(sp)
         r1 = 12(sp)
         r2 = r0 + r1
32:      r0 = r2
         ...
         12(sp) = r1
```



```
BB1
         r0 = 0(sp)
         r1 = 4(sp)
11:      r2 = r0 + r1
         r3 = 12(sp)
12:      r1 = r3
13:      r3 = r1 + r2
14:

15:      goto L1
```

```
BB2
         r0 = 0
21:      0(sp) = r2
         r1 = 2
22:      4(sp) = r1
         r3 = 0
23:
24:      goto L1
```

```
BB3
31: L1:  r2 = r0 + r3
32:      r0 = r2
         ...
         12(sp) = r3
```

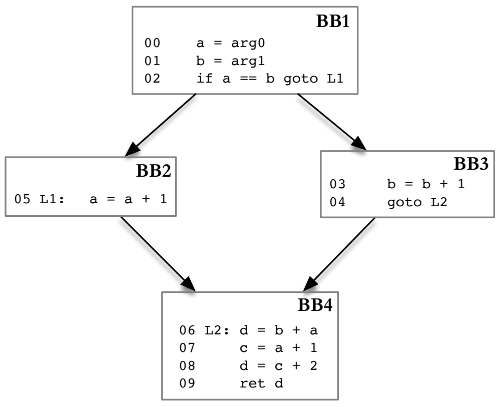
Problem 3. Control-Flow Graph and Global Register Allocation [30 points]
Consider the 3-address code below corresponding to a function with 2 arguments and local variables a, b, c and d.

```
00       a = arg0
01       b = arg1
02       if a == b goto L1
03       b = b + 1
04      goto L2
05 L1:  a = a + 1
06 L2:  d = b + a
07      c = a + 1
08      d = c + 2
09      ret d
```

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. [15 points] Determine the live ranges for the variables `a`, `b`, `c`, and `d`, the corresponding webs and interference graphs, for the 2 interference notions discussed in class. Recall, that in the "more refined' interference notion, one could have two variables being used on the same line but if one web ends at that line and corresponds to a read operation, and the other web begins at that line and corresponds to a write operation, then the 2 webs do not interfere at that line.
c. [10 points] What is the minimum number of registers you can use related to each of the two interference graphs? Why?

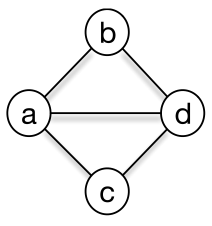
Solution:
a) The basic blocks and the corresponding CFG is depicted below (on the left).

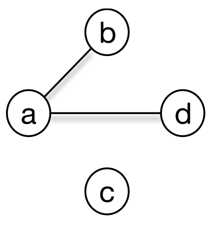


```
BB1
00    a = arg0
01    b = arg1
02    if a == b goto L1

BB2
05 L1:   a = a + 1

BB3
03     b = b + 1
04     goto L2

BB4
06 L2: d = b + a
07     c = a + 1
08     d = c + 2
09     ret d
```

```
a: {0,1,2,3,4,5,6,7}
b: {1,2,3,4,5,6}
c: {7,8}
d: {6,7,8,9}
```

b) For each of the variables we indicate (above, on the right) the instructions, using the corresponding line numbers, where the variable is alive. Recall, that a variable is alive at instruction X if it is used at that particular instruction of use at another instruction Y possibly executed later during the program's execution.

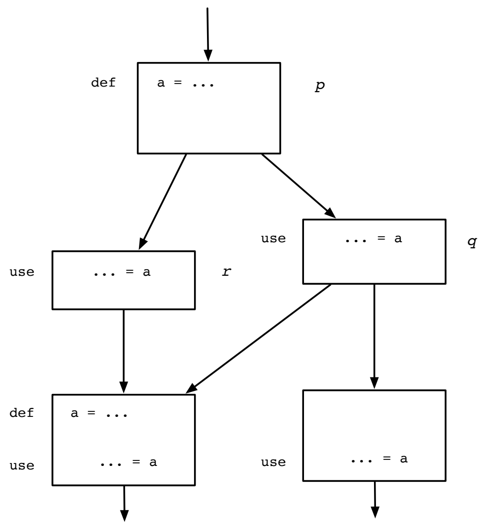




c) Clearly, the first interference graph (left), with more edges requires 3 colors, since it has at least one clique of size 3, whereas the second interference graph (right) requires only 2 colors.

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

In class we discussed the Live-Variable Analysis problem, where one variable is said to be live at program point *p*, if its value is used along some control-flow path from *p*. In other word, the value of the variable at p can be used in another point *q* without being possibly redefined. In the example below, we say that the variable *a* is live at *p* because there is a control-flow path starting at *p* where the current value of *a* is still possibly used (in this case at *q*). Conversely, the same variable is dead at *r* since its value is no longer used beyond that point as the variable *a* is redefined in all the control-flow paths beyond *r* into the following basic block.



Regarding this data-flow analysis problem answer the following:

a. [05 points] What is the set of values in the corresponding 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 basic block transfer function for this data-flow problem, i.e., the GEN and KILL and the equations the iterative approach needs to solve? At control-flow merge points, what is the meet operator, and why?
d. [05 points] What would be the preferred order of processing of the basic blocks in this particular data-flow analysis problem? Explain the rationale.

Solution:

a. Lattice values consist of unordered sets of variables. All blocks initialize their IN values to the empty set.
b. A possible formulation of this problem defines the input values as a function of the output values of each basic block, so the direction is backwards. The information that a variable is still live can be propagated backwards in the control flow revealing that in the forward direction a given variable is still needed. This need is killed or eliminated at the point where the variable is defined. Before that definition, the variable is in fact dead. On merge of control-flow paths (backwards, of course) a variable is alive if at least in one of the control-flow paths the variable is alive, which suggests that the merge function is the union of the IN sets of the successors of a basic block.
c. As suggested in the description, the transfer function (either at the basic block level or at the instruction level) is defined by the two equations below.

$$\mathrm{OUT}(B) = \cup_{s\ \mathrm{succ}(B)}\ \mathrm{IN}(s)$$
$$\mathrm{IN}(B) = \mathrm{Use}(B) \cup (\mathrm{OUT}(B) - \mathrm{Def}(B))$$

The merge is the Union, since the problem clearly states that a variable is alive if there is at least one path along which the value of a given definition of the variable can still be used. Notice, that by initializing the IN values to the empty sets and using the Union function, the solutions of the equations can only grow and are limited by the size of the universe set that includes all the variables in the program under consideration.
d. In this particular case, and since it is a backwards problem, the preferred order would be to visit the nodes in reverse pre-order since this order ensures that for every node, all its successors are computed beforehand

Problem 5. Code Optimization [20 points]

Consider the following excerpt of code where you can assume the variables b and N to be defined prior to the execution of this code section. Also assume that the variable x is live at the end of the execution of this code section.

```
int x, acc = 0;
for(int i=0; i < N; i++) {
   x = b + 1;
   acc = acc + x + i;
}
```

For this code:

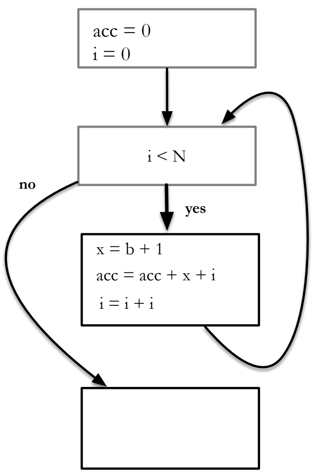
a. [10 points] Derive the corresponding CFG and basic blocks and identify opportunities for loop invariant code motion (LICM). Where can you move the loop invariant code? Justify.

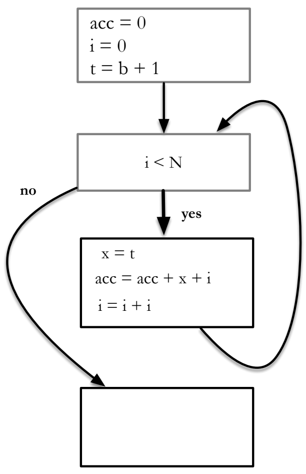
b. [05 points] Can you remove the code associated with the computation of variable `x`? Why or why not.

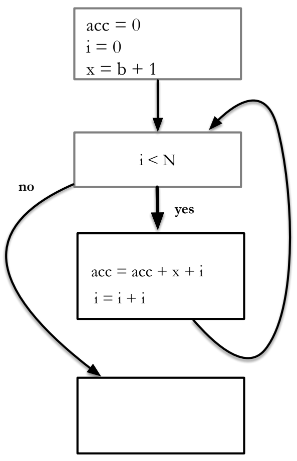
c. [05 points] Under which circumstances are the transformations you have carried out profitable? Explain. Can you think of other transformations you could make to this code to simplify the transformed code?

Solution:

a) A possible CFG realization is as shown below (left).







There are opportunities for LICM. First, the sub-expression b+1 is loop invariant, so it can be move outside the loop, say to the first basic block as that block dominates all the blocks of the loop. This is depicted in the figure above (center). Second the variable x is itself loop invariant and can also be moved outside the loop resulting in the code above (right). Notice, however, that we are assuming that the loop is executed at least once, as otherwise, moving the computation of the variable x to the preheader of the loop would result in a code with modified externally visible behavior.

b) Regarding removing `x`, since we are told that the variable `x` is alive, we need to save the last and only computation associated with `x`. Since the basic block where `x` is defined does not dominator the exit of the loop, we cannot move the code and hence cannot apply this transformation. However, and given that `x` was not well defined in case N is 0, it is acceptable to make a transformation.

c) Clearly, only when N > 1 is this profitable since we have moved some computation outside the loop and executed exactly once. If N is zero, then we worsen the transformed code. Given the simplicity of the code, in essence an arithmetic progression the entire could be transformed to compute the symbolic expression: `N * b + (N*(N+1))/2` thus eliminating the loop entirely.