CSCI 565 - Compiler Design

Spring 2010

Final Exam - Solution

May 07, 2010 at 1.30 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 [25 points]:
Problem 2 [25 points]:
Problem 3 [50 points]:

Total:

Instructions:

1. This is a closed-book Exam.
2. The Exam booklet contains four (4) pages.
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 pen (not a pencil) to answer your questions.

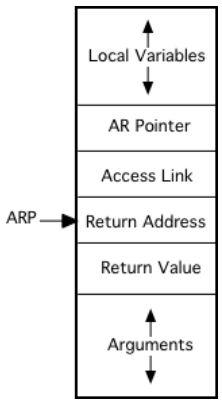
Problem 1: Run-Time Environment and Storage Organization [25 points]

Under the assumption that the AR are allocated on the stack with the individual layout as shown below, and given the PASCAL code on the right-hand-side answers the following questions:

(a) [10 points] Draw the set of ARs on the stack when the program reaches the end of the execution of the line 20 in procedure P2. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access links and clearly label the values of local variables and parameters in each AR.

(b) [10 points] Outline the code using three address instructions for the statement "x := x - 1" in line 20. Make explicit use of the access link. Assume assuming the current value of ARP is stored in the register named ARP.

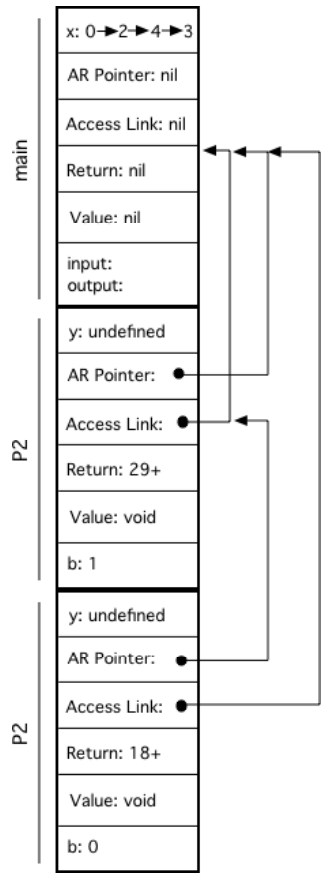
(c) [05 points] Discuss if the AR for the procedure P3 needs to be allocated on the stack. Under which conditions is it possible for a procedure/function not has its AR allocated on the stack? Where can it be allocated if not on the stack?




```
01: program main(input,  output);
02:    var x : integer;
03:    procedure P1(a: integer));
04:      var x: integer;
05:      begin
06:         x := 1;
07:         if (a <> 0) then
08:           begin
09:              P1(a-1);
10:              P2(a);
11:           end
12:       end;
13:      procedure P2(b: integer);
14:       var y: integer;
15:       begin
16:          x := x + 2;
17:          if(b <> 0) then
18:               P2(b-1);
19:          else
20:               x := x – 1;
21:       end;
22:      procedure P3(c: integer);
23:       begin
24:         writeln(c);
25:       end
26:  begin (* main *)
27:     x := 0;
28:     P1(0);
29:     P2(1);
30:  end.
```

Solution:

a) The layout of the ARs on the stack is as shown below. Note the sequence of value transition for the global variable "x" at the main AR as this is the only variable "x" in scope for the statements in lines 16 and 20.



b) Assuming that negative offset with respect to the ARP correspond to field above the ARP in the figure depicted for a) we would have the following sequence of three address instructions:

```
01:     t1  = ARP + offset_AL  // getting the address of this AR's AL
02:     t2  = *t1              // getting the AL value itself.
03:     t3  = t2 - offset_x    // getting the address of local variable x on main's AR
04:     t4  = *t3              // getting the value of x
05:     t5  = t4 - 1           // decrementing the value of x by 1
06:     *t3 = t5               // storing the value back at its address in the AR.
```

Notice that in this sequence we are reusing the address of x in t3. A less astute implementation would have to invoke a sequence of instruction similar to the instructions in lines 01 through 03 to obtain the address after the computation in instruction 05.

c) For procedures that are not involved in cycles in the call graph we can allocate their AR statically in a global area of memory. This is the case with P3. In some languages the procedure's local data can outlive the invocation of the procedure itself. In these particular cases the AR need to be allocated on the heap should the language allow recursion as well.

Problem 2: Register Allocation [25 points]

Consider the following three-address format representation of a computation using scalars and arrays A, B and a procedure `F(int a, int b)`. For this particular exercise assume the values of the arguments for the procedure F are passed on a stack using the instructions `putparam` (push) and `getparam` (pop).

```
01:   t1 = i
02:   t2 = A[t1]
03:   t3 = j
04:   t4 = B[t3]
05:   putparam t2
06:   putparam t4
07:   call F,2
08:   t5 = i
09:   t6 = t5 + 1
```

a) [05 points] Assume the arrays A and B are globally allocated at a specific offset of a Global Register Pointer (GRP), respectively offset_A and offset_B. Similarly for the scalar variables this program manipulates. To access the arrays in instructions in lines 02 and 04 you need to use the GRP. As such modify the code to make the access to the arrays explicit taking into account the fact that each integer uses 4 bytes of memory. Also load the value of the scalar variables using the offset of the scalar from the Activation Record Pointer (ARP).

b) [15 points] Show the assignment of variables to registers before the execution of each instruction, and using the bottom-up register allocation algorithm for 3 registers, rewrite the code in this procedure. Assume that variable `i` is live outside this basic block and after the call to F.

c) [05 points] Do you think a global register allocation algorithm would do better than the bottom-up register allocator for this particular segment of code? Why or why not?

Solution:

a) [05 points] The modified code has to take into account the fact that you need to access the local variables via the Activation Record Pointer (ARP) and the arrays via the Global Register Pointer (GRP). A revised version of the code is shown below. We use lines numbers and comments to help you understand the mapping.

```
01: t1 = ARP + offset_i    // getting the address of i
    t1 = *t1               // loading the value of i in t1
02: t1 = t1 * 4            // computing the offset of A[i]
    t2 = GRP + offset_A    // getting the base address of A
    t2 = t2 + t1           // computing the offset of address of A[i]
    t2 = *t2               // loading of A[i]
03: t3 = ARP + offset_j    // getting the address of j
    t3 = *t3               // loading the value of j in t3
    t3 = t3 * 4            // computing the offset of B[i]
    t4 = GRP + offset_B    // getting the base address of A
04: t4 = t3 + t4           // computing the offset of address of B[j]
    t4 = *t4               // loading of B[j]
05: putparam t2
06: putparam t4
07: call F,2
08: t5 = ARP + offset_i    // getting the address of i
    t5 = *t5               // loading the value of i in t5
09: t6 = t5 + 1            // calculation in line 09
```

b) [15 points] The code below illustrates the resulting code using the bottom-up register allocator.

```
01: r0 = ARP + offset_i    // r0 = t1;    r1 = empty; r2 = empty
    r0 = *r0               // r0 = t1;    r1 = empty; r2 = empty
02: r0 = r0 * 4            // r0 = t1;    r1 = empty; r2 = empty
    r1 = GRP + offset_A    // r0 = t1;    r1 = t2;    r2 = empty
    r1 = r1 + r0           // r0 = empty; r1 = t2;    r2 = empty
    r1 = *r1               // r0 = empty; r1 = t2;    r2 = empty
03: r0 = ARP + offset_j    // r0 = t3;    r1 = t2;    r2 = empty
    r0 = *r0               // r0 = t3;    r1 = t2;    r2 = empty
    r0 = r0 * 4            // r0 = t3;    r1 = t2;    r2 = empty
    r2 = GRP + offset_B    // r0 = t3;    r1 = t2;    r2 = t4
04: r2 = r0 + r2           // r0 = t3;    r1 = t2;    r2 = t4
    r2 = *r2               // r0 = empty; r1 = t2;    r2 = t4
05: putparam r1            // r0 = empty; r1 = t2;    r2 = t4
06: putparam r2            // r0 = empty; r1 = empty; r2 = t4
07: call F,2               // r0 = empty; r1 = empty; r2 = empty
08: r0 = ARP + offset_i    // r0 = t5;       r1 = empty; r2 = empty
    r0 = *r0               // r0 = t5;       r1 = empty; r2 = empty
09: r1 = r0 + 1            // r0 = t5;       r1 = t6;    r2 = empty
```

Notice that in the last instruction we kept the value of t5 in r0 as the corresponding program variable, `i`, is alive and hence its value is used next in addition to the use on line 09.

(c) [05 points] For this particular code the number of register is 3 which is strictly necessary given the calculations of the array addresses and the fact that we need to hold the values in register for parameters while we compute the value of the second argument. If we were to compute the inference web we would find out that there is a clique of size 3 and hence we would have to have at least 3 registers. The global graph-based algorithm would find we need at least 3 registers without spilling which is exactly what we got with this bottom-up register allocator.

Problem 3: Analysis and Optimization [50 points]

For the code shown below, determine the following:

```
01        a = 1
02        b = 0
03 L0:    a = a + 1
04        b = p + 1
05         if (a > b) goto L3
06 L1:    a = 3
07         if (b > a) goto L2
08        b = b + 1
09         goto L1
10 L2:    a = b
11         b = p + q
12         if (a > b) goto L0
13 L3:    t1 = p * q
14        t2 = t1 + b
15         return t2
```

a) [10 points] The basic blocks of instructions and the control-flow graph (CFG) and the CFG dominator tree.

b) [10 points] Identify the natural loops and determine if they are nested or not (explain why or why not).

c) [10 points] The live variables at the end of each basic block. A variable is said to be live at the end of a given basic block or instruction if its value is used beyond that basic block or instructions. You do not need to determine the live variables before and after each basic block and justify your answer for the value presented for the basic block containing instruction at line 11.

d) [10 points] Indicate the solution for the Reaching-Definitions (RD) Data-flow Analysis problem for the variables b only. You do not have to show all the passes of your solution. Simply show the final results of this analysis by indicating for each of the uses, which definition reaches it. Indicate each use as u_x and each definition as d_x where the subscript x indicates the line number for which the use/definition corresponds to.

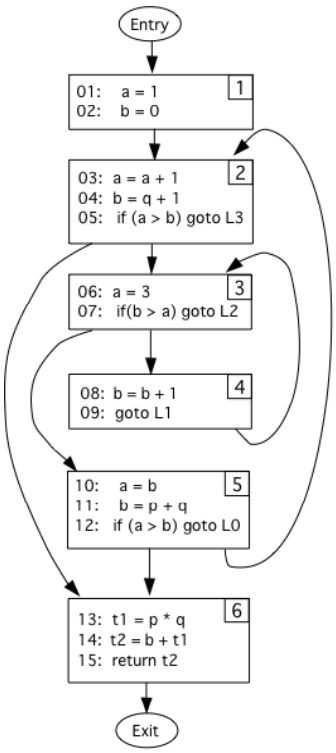
e) [10 points] Indicate which expressions are loop invariant and where to could they be moved in the code. Discuss the profitability of your transformation.

***Solution**:*

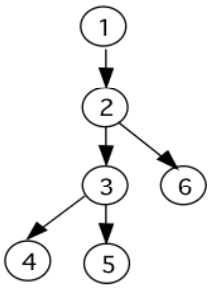
a) We indicate the instructions in each basic block and the CFG and dominator tree on the RHS.

BB1: {01, 02}
BB2: {03, 04, 05}
BB3: {06, 07}
BB4: {08, 09}
BB5: {10, 11, 12}
BB6: {13, 14, 15}

Basic Blocks



Control-Flow Graph (CFG)



Dominator Tree

b) There are two edges, whose basic blocks pointed to by their heads dominate the basic blocks at their tails. These edges are (4,3) and (5,2). Tracing back the nodes of the CFG backwards (against the flow of control) we get the natural loop composed by the basic blocks {3, 4} and {2, 3, 4, 5} respectively. As the first set of nodes is a subset of the nodes in the second set, the first loop is nested within the second loop.

c) At the end of each basic block we can inspect the code and ask if a given variable is still used after the control flow leaves that basic block. The results of this inspection are depicted below:

BB1: {a, p, q}
BB2: {b, p, q}
BB3: {b, p, q}
BB4: {b, p, q}
BB5: {a, b, p, q}
BB6: { }

For BB5 the live variable solution at the exit of this basic block has {a, b, p, q} as there exists a path after BB5 where all the variables are being used. For example variable "a" is used in the basic block BB2 as it is read in the assignment "a = a + 1". All other variables are used in BB6 as part of the return expression calculation for instance.

d) For variable b we have 4 definitions namely $\{d_2, d_4, d_8, d_{11}\}$ and 6 uses, namely $\{u_5, u_7, u_8, u_{10}, u_{12}, u_{13}\}$. The table below illustrates which definitions reach which uses. For completeness we also present the table for variable a.

RD_b	d_2	d_4	d_8	d_{11}
u_5		✓		
u_7		✓	✓	
u_8		✓	✓	
u_{10}		✓	✓	
u_{12}				✓
u_{13}		✓		✓

RD_a	d_1	d_3	d_6	d_{10}
u_3	✓			✓
u_5		✓		
u_7			✓	
u_{12}				✓

From this representation it becomes clear that definition d_2 for variable b is useless in the sense that is never used. The corresponding code on line 2 can thus be eliminated.

e) There are two expressions that are loop invariant, namely the expression "3" on line 6 and the expression "p+q" on line 11. While in the first case the constant can be propagate to the only place where it is used on line 7, in the second case the situation is more delicate. For the second case the expression "p+q" can be moved to the basic block BB2 where it is saved as a temporary variable, say t = p + q. This temporary expression is then used in the expression in line 11 as "b = t". Notice however that moving this computation to basic block BB2 raises the issue that it is always executed even if the outer loop is never executed. In practice this is seldom an issue as loops tend to be executed many times.