CS352 - Compilers: Principles and Practice
Final Exam

"The final annihilation of the lifeform known as Man. Let the Exam begin..."

2003 Dec 15

Instructions: Read carefully through the entire exam first, and plan your time accordingly. Note the relative weights of each segment, as a percentage of the total exam score.

This exam is **closed book, closed notes**. You may *not* refer to any books or other materials during the exam.

Write your answers on this exam. You may use both sides of the page.

When you are done, present your completed exam and your student identification to the instructor or proctors at the head table. If leaving before the exam period is concluded, please leave as quietly as possible as a courtesy to your neighbors.

Name:

Student Number:

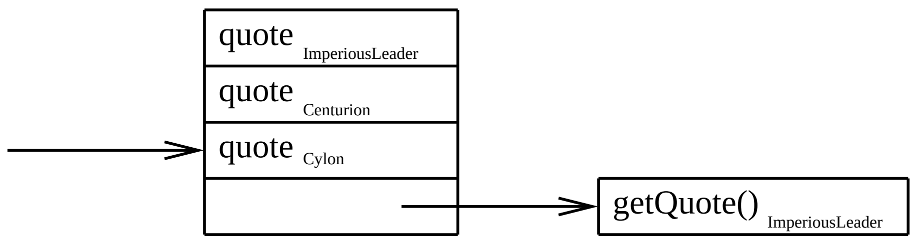
Signature:

Consider the following code for Q1 and Q2:

```
class Cylon
{
  String quote = "Death to the Galactica.";
  public String getQuote() {   return quote;   }
}

class Baltar extends Cylon
{
  String quote = "Possibly a Cylon welcoming committee?";
  public String getQuote() {   return quote;   }
}

class Centurion extends Cylon
{
  String quote = "By your command.";
  public String getQuote() {   return quote;   }
}

class ImperiousLeader extends Centurion
{
  String quote = "Let the attack begin.";
  public String getQuote() {   return quote;   }

  public static void main(String[] args)
  {
    // Code snippet for Q1a:
    Cylon redeye  = new Cylon();
    Cylon traitor = new Baltar();
    Cylon lucifer = new Centurion();
    Cylon leader  = new ImperiousLeader();
    int x, y, z;    x = 2; y = 3; z = 4;

    // Code snippet for Q1b:
    String quote = leader.getQuote();

    // Code snippet for Q2:
    x = y + z * 8;

  }
}
```

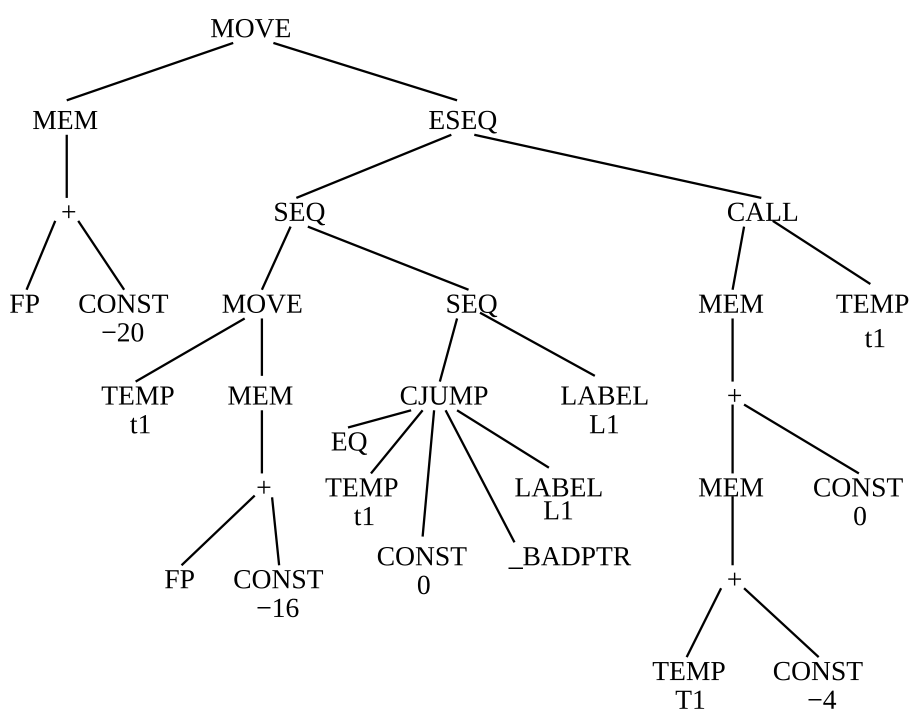
1. (OBJECT LAYOUT and INTERMEDIATE TRANSLATION: 20%)

   (a) (10%) Draw the object layout that the MJ compiler would use to represent the runtime object of class `ImperiousLeader` (or `IL` for short) allocated above. Label the locations and types of each member field, the vtable, and the methods in the vtable. Include an arrow pointing to the record which corresponds to the reference stored in local variable `leader`.



   (b) (10%) Give the Intermediate Tree representation of the line of code that calls the `getQuote()` method above.

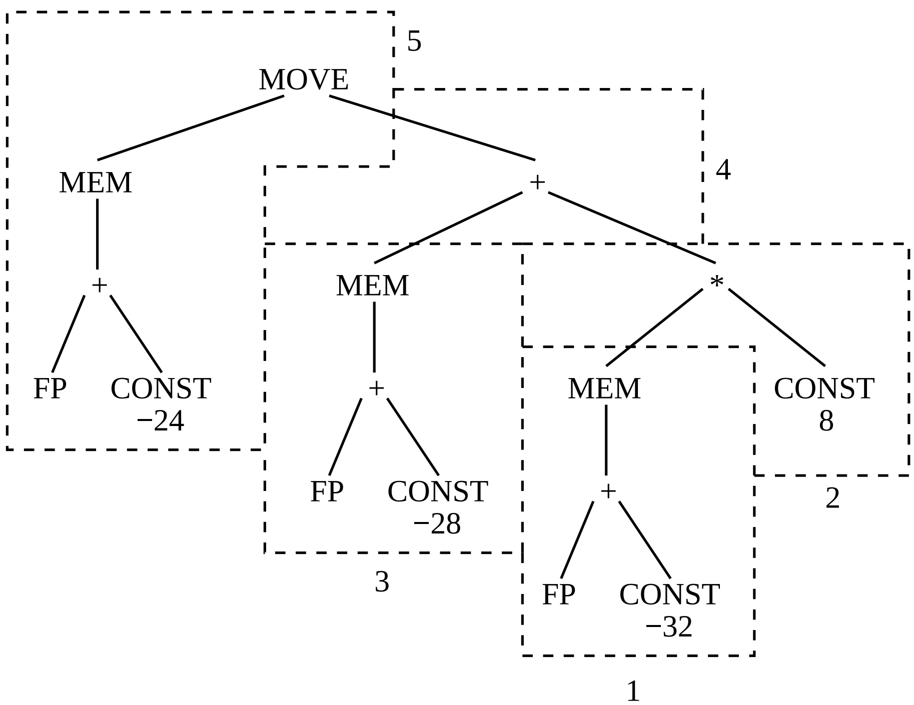
   - Assume a wordsize of 4 bytes, and the existence of a "`_BADPTR`" routine for handling null pointer errors. (Yes, you need to check for a null pointer.)
   - Further assume that locals `leader` and `quote` are stored in the frame at `MEM(FP + CONST(-16))` and `MEM(FP + CONST(-20))`, respectively.
   - Use your answer to part A to determine record offsets. You should distinguish temporaries and labels, but the order of numbering (i.e. temporary "`t1`" vs. temporary "`t2`") does not matter.



2. (INTERMEDIATE TRANSLATION and INSTRUCTION SELECTION: 20%)

(a) (10%) Give the Intermediate Tree representation of the line of code, "`x = y + z * 8;`" above. Assume that `x`, `y`, and `z` are stored in the frame at offsets -24, -28, and -32 from the frame pointer, respectively.

Give an optimal tiling of the IR tree you have drawn, using tiles that will allow you to select appropriate Mips instructions for this code. Number the tiles in your diagram to indicate the order in which they will emit instructions.



(b) (10%) Emit Mips instructions for your tiled tree. Wherever appropriate, use *shift* opcodes in place of more expensive operations, and indexed addressing modes instead of multiple equivalent opcodes.

Some typical Mips opcode templates are given below for your reference:

`add`	R_d	R_{s1}	R_{s2}	add	`sub`	R_d	R_{s1}	R_{s2}	subtract
`mulo`	R_d	R_{s1}	R_{s2}	multiply	`div`	R_d	R_{s1}	R_{s2}	divide
`srl`	R_d	R_s	I_{16}	shift right	`sll`	R_d	R_s	I_{16}	shift left
`lw`	R_d	$I_{16}(R_b)$		load word	`sw`	R_s	$I_{16}(R_b)$		store word
`la`	R_d	label		load address	`jal`	label			jump

```
lw   t0,  -32(fp)
sll  t0,  3
lw   t1,  -28(fp)
add  t2,  t0,  t1
sw   t2,  -24(fp)
```

Consider the following familiar code for Q3:

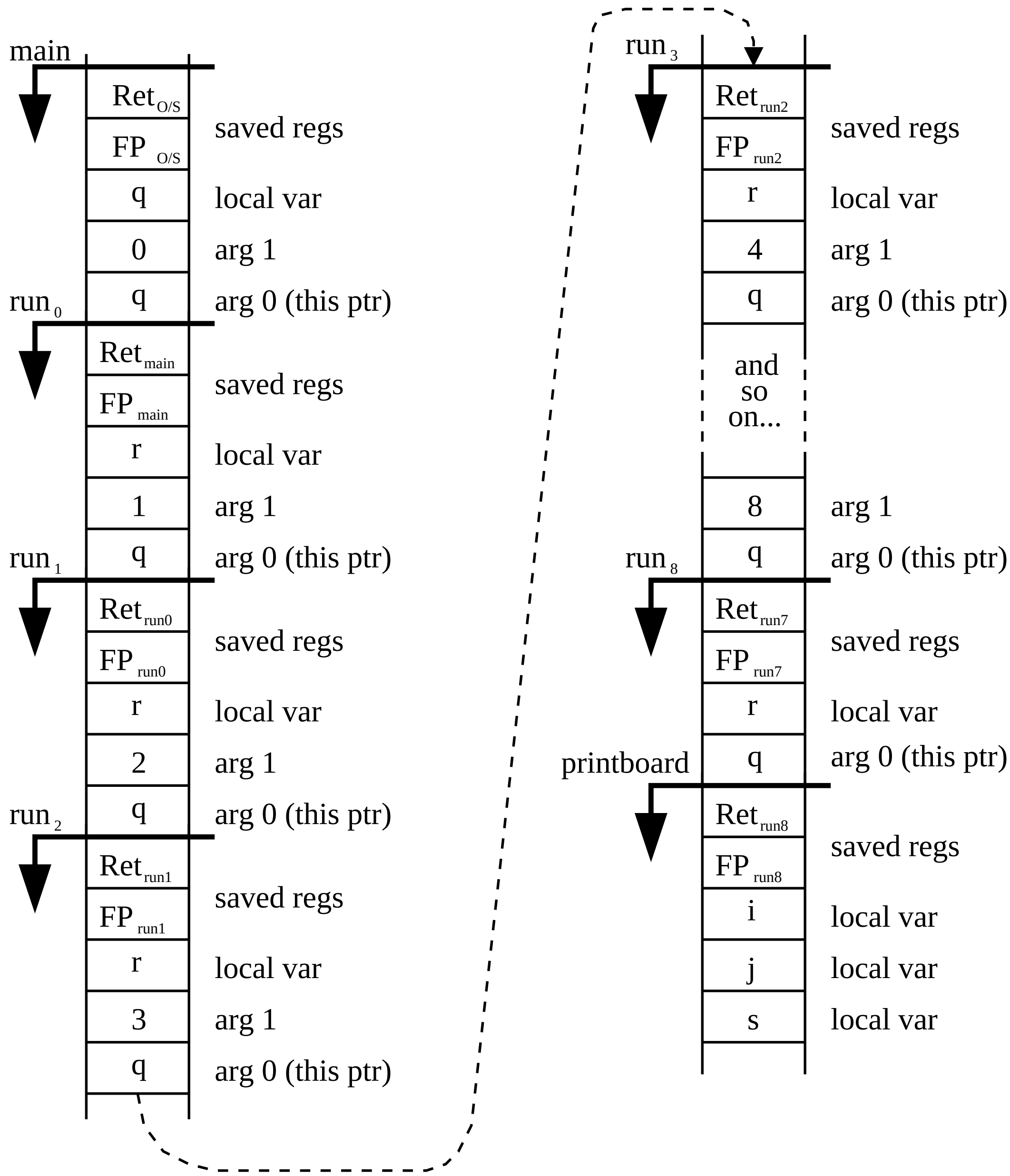
```
class Queens
{   public static void main (String[] a)
    {   Queens q = new Queens().init();    q.run(0);   }

    int N;
    int[] row, col, diag1, diag2;

    Queens init()
    {   int n = 8;                          this.N = n;
        this.row = new int[n];              this.col = new int[n];
        this.diag1 = new int[n+n-1];        this.diag2 = new int[n+n-1];
        return this;
    }
    void run (int c)
    {   if (c == this.N) this.printboard();
        else
        {   int r = 0;
            while (r < this.N)
            {   if (this.row[r] == 0 && this.diag1[r+c] == 0
                                     && this.diag2[r+7-c] == 0)
                {
                    this.row[r]       = 1;   this.diag1[r+c]   = 1;
                    this.diag2[r+7-c] = 1;   this.col[c]       = r;
                    this.run(c+1);
                    this.row[r]       = 0;   this.diag1[r+c]   = 0;
                    this.diag2[r+7-c] = 0;
                }
                r = r+1;
    }   }   }
    void printboard()
    {   int i = 0;
        while (i < this.N)
        {   int j = 0;
            while (j < this.N)
            {   String s;
                if (this.col[i] == j) s = " Q"; else s = " .";
                System.out.print(s);            j = j+1;
            }
            System.out.println();               i = i+1;
        }
        System.out.println();
}   }
```

3. (ACTIVATION RECORDS: 15%)

Draw the stack of activation records for this program at the point where `printboard()` is executing. Assume:

- an Intel-like architecture with all locals and formals stored in memory, (as opposed to in registers, like Mips,)
- the return address automatically at the top of each stack frame, and
- all methods treated as non-leaf methods.

Carefully label arguments, locals, return addresses, and frame pointers. You may omit ignored, empty, or unknown words allocated in the frames. You may assume the result value is returned in a register.



4. (BASIC BLOCKS and TRACES: 15%)

Consider the program below:

```
1  m := 0
2  v := 0
3  if v > n goto 15
4  r := v
5  s := 0
6  if r < v goto 9
7  v := v + 1
8  goto 3
9  x := M[r]
10 s := s + x
11 if s < m goto 13
12 m := s
13 r := r + 1
14 goto 6
15 return m
```

(a) (10%) Break the program into basic blocks, expressed in fragments of Intermediate Representation ("`Tree`") form. For example, the "`return m`" statement can be translated into "`LABEL(L15), MOVE(RV, m)`"

```
LABEL(L1),  MOVE(m, CONST(0)),  MOVE(v, CONST(o)),  JUMP(L3)
LABEL(L3),  CJUMP(LT, v, n, L15, L4)
LABEL(L4),  MOVE(r, v),  MOVE(s, 0),  JUMP(L6)
LABEL(L6),  CJUMP(LT, r, v, L9, L7)
LABEL(L7),  MOVE(v, BINOP(PLUS, v, 1)),  JUMP(L3)
LABEL(L9),  MOVE(x, MEM(r)),  MOVE(s, BINOP(PLUS, s, x)),
            CJUMP(LT, s, m, L13, L12)
LABEL(L12), MOVE(m, s),  JUMP(L13)
LABEL(L13), MOVE(r, BINOP(PLUS, r, 1)),  JUMP(L6)
LABEL(L15), MOVE(RV, m)
```

(b) (5%) Generate a minimal set of traces for your basic blocks from part A using the algorithm discussed in lecture and in the text. Start with label `L1`, and clearly indicate which blocks (they can be referred to by label only, for brevity) appear in each trace.

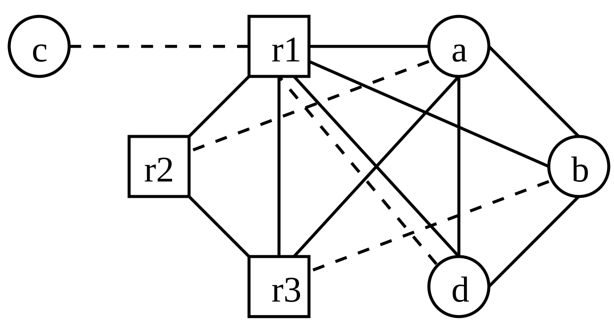
```
Trace 0:  L1, L3, L4, L6, L7
Trace 1:  L15
Trace 2:  L9, L12, L13
```

5. (LIVENESS ANALYSIS and REGISTER ALLOCATION: 30%)

(a) (5%) The code below has been compiled for a three-register architecture. Register `r1` is a caller-save argument/result register. Registers `r2` and `r3` are callee-save.

Perform liveness analysis and clearly indicate the *live out* set calculated for each line of code. (You may calculate other sets, but only the *live out* sets will be graded.) :

		def	use	live in	liveout
	`a := r2`	$\{a\}$	$\{r2\}$	$\{r1, r2, r3\}$	$\{a, r1, r3\}$
	`b := r3`	$\{b\}$	$\{r3\}$	$\{a, r1, r3\}$	$\{a, b, r1\}$
	`c := r1`	$\{c\}$	$\{r1\}$	$\{a, b, r1\}$	$\{a, b\}$
	`r1 := 5`	$\{r1\}$	$\{\}$	$\{a, b\}$	$\{a, b, r1\}$
	`call f1`	$\{r1\}$	$\{r1\}$	$\{a, b, r1\}$	$\{a, b, r1\}$
	`d := r1`	$\{d\}$	$\{r1\}$	$\{a, b, r1\}$	$\{a, b, d\}$
`L1:`	`r1 := d`	$\{r1\}$	$\{d\}$	$\{a, b, d\}$	$\{a, b, d, r1\}$
	`call f2`	$\{r1\}$	$\{r1\}$	$\{a, b, d, r1\}$	$\{a, b, d, r1\}$
	`d := d + r1`	$\{d\}$	$\{d, r1\}$	$\{a, b, d, r1\}$	$\{a, b, d\}$
	`r1 := d`	$\{r1\}$	$\{d\}$	$\{a, b, d\}$	$\{a, b, d, r1\}$
	`if d < 10 goto L1`	$\{\}$	$\{d\}$	$\{a, b, d, r1\}$	$\{a, b, d, r1\}$
	`r3 := b`	$\{r3\}$	$\{b\}$	$\{a, b, r1\}$	$\{a, r1, r3\}$
	`r2 := a`	$\{r2\}$	$\{a\}$	$\{a, r1, r3\}$	$\{r1, r2, r3\}$
	`return` (r1,r2,r3 live out)	$\{\}$	$\{r1, r2, r3\}$	$\{r1, r2, r3\}$	$\{r1, r2, r3\}$



(b) (5%) Show the interference graph for the code above by filling in the table below. Put an "X" wherever an interference edge exists in the graph, and an "O" wherever a move relationship exists between two nodes. If there are any constrained moves, put in both the X and the O.

	r1	r2	r3	a	b	c	d
a	X	O	X		X		X
b	X		O	X			X
c	O						
d	XO			X	X		

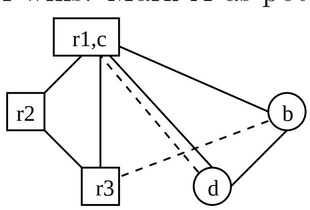
(c) (15%) Show the steps for register allocation with conservative coalescing, using the algorithm discussed in the textbook and in lectures. Use the Briggs criteria for conservative coalescing and "(uses + defs) / degree" as a spill priority heuristic.

You need not show all of your work, but you must clearly indicate each step of the algorithm. (i.e., "Build", "Coalesce x and y", or "Simplify z".)

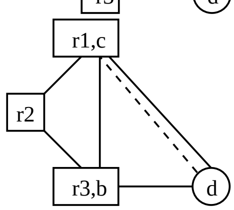
Build. Coalesce C with R1. Cannot perform any other moves, must mark potential spill.

	(	uses	+	defs	)	/	degree	priority
a	(	1	+	1	)	/	4	0.5
b	(	1	+	1	)	/	3	0.66
c	(	0	+	1	)	/	0	∞
d	(	4	+	2	)	/	3	2

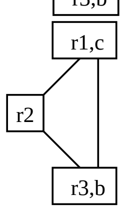
A wins. Mark A as potential spill.



Coalesce B with R3.



Freeze and Simplify D.

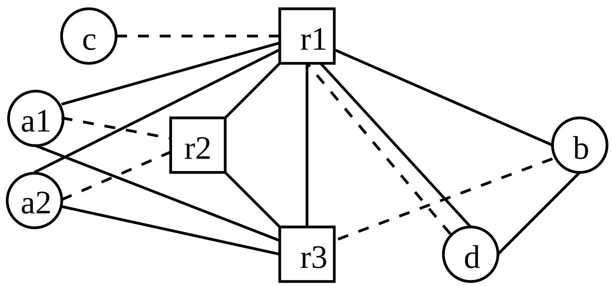


Select. No color left for A;
potential spill becomes actual spill.

Insert code for spilling A:

```
a := r2  →  a1 := r2
            M[a_loc] := a1

r2 := a  →  a2 := M[a_loc]
            r2 := a2
```

Recalculate liveness, rebuild interference graph.



Coalesce A1 with R2, and A2 with R2.

Coalesce C with R1.

Coalesce B with R3 (as before spill).

Freeze and Simplify D (as before spill).

Select Colors:

$r1$	$r2$	$r3$	$a1$	$a2$	b	c	d
1	2	3	2	2	3	1	2

(d) (5%) Show the final register allocated code with any redundant moves eliminated. (Be sure to include the code for any actual spills.)

```
                         // r2 := r2
    M[a_loc] := r2
                         // r3 := r3
                         // r1 := r1
    r1 := 5
    call f1
    r2 := r1
L1: r1 := r2
    call f2
    r2 := r2 + r1
    r1 := r2
    if r2 < 10 goto L1
                         // r3 := r3
    r2 := M[a_loc]
                         // r2 := r2
    return
```