CS 352 – Compilers: Principles and Practice
Final Examination, 12/11/05

Instructions: Read carefully through the whole exam first and plan your time. Note the relative weight of each question and part (as a percentage of the score for the whole exam). The total points is 100 (your grade will be the percentage of your answers that are correct).

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

You have **two hours** to complete all **nine** (9) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature:

1. (Runtime management: 25%) Consider the following MiniJava program:

```
class List {
    public static void main (String[] a) {
        List list1 = List.cons(1, List.cons(3, null));
        List list2 = List.cons(2, List.cons(4, null));
        list1.merge(list2);
        System.out.println("");
    }

    int head;
    List tail;
    static List cons(int head, List tail) {
        List list = new List();
        list.head = head;
        list.tail = tail;
        return list;
    }
    void merge (List other) {
        if (other == null) {
            for (List l = this; l != null; l = l.tail)
                Int.print(l.head);
        } else if (this.head < other.head) {
            Int.print(this.head);
            other.merge(this.tail);
        } else {
            Int.print(other.head);
            this.merge(other.tail);
        }
    }
}

class Int {
    static void f(int i) {
        if (i > 0) { Int.f(i/10); System.out.write(i-i/10*10+'0'); }
    }
    static void print(int i) {
        if (i < 0) { System.out.write('-'); i = -i; }
        if (i == 0) System.out.write('0'); else Int.f(i);
        System.out.write(' ');
    }
}
```

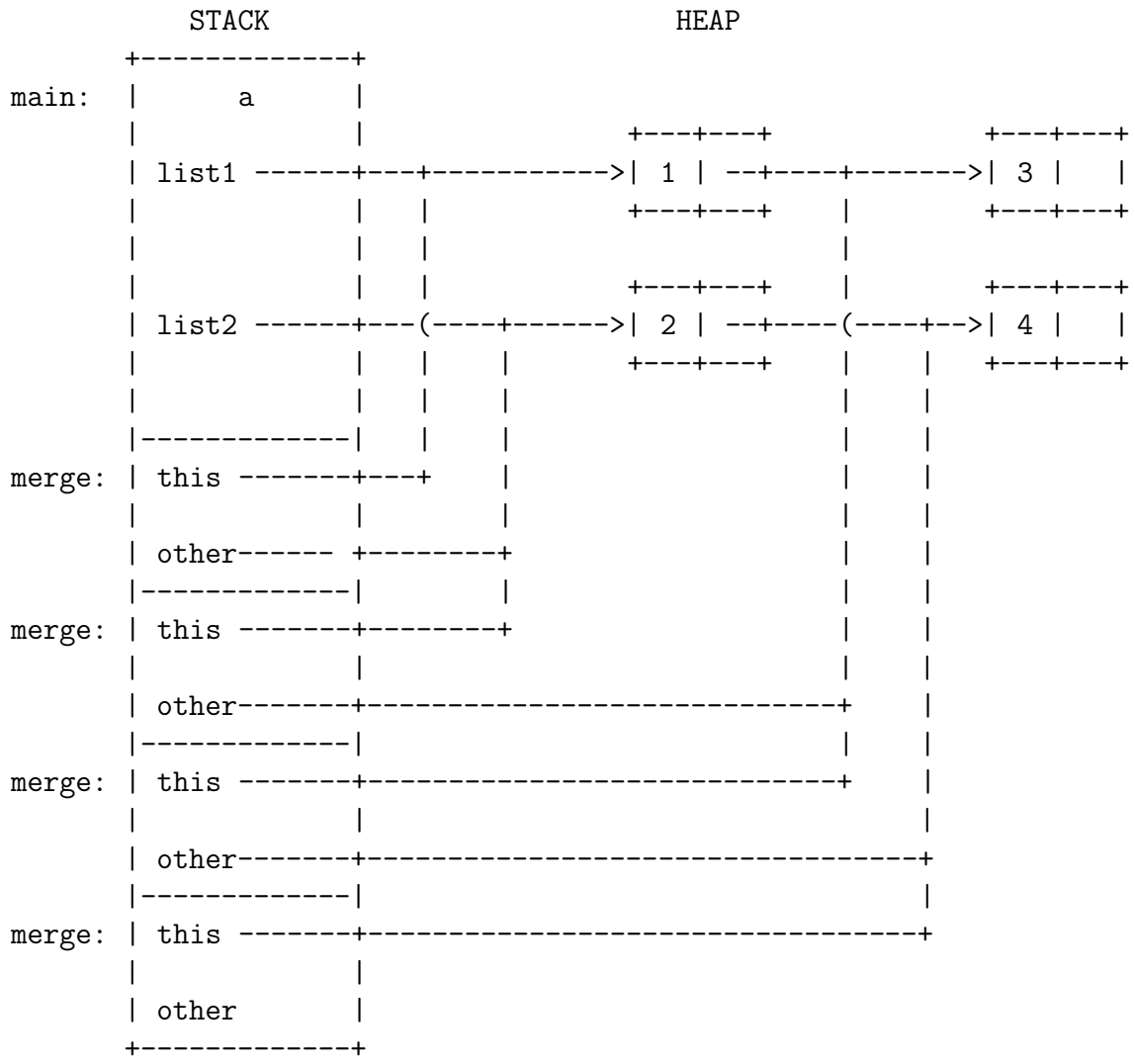
(a) (5%) What output does this program produce when the `List.main` method runs?

Answer:

1 2 3 4

(b) (20%) Show a diagram of PowerPC stack frames *at the point* in `List.merge` where `Int.print` is called with the value `4`. Show where *all* the local variables and heap variables are (assume that all local variables are stored in memory in the stack, not in registers); show the value of all integer variables in the stack and heap, as well as variables containing references to the heap, and the object they refer to.

Answer:



The remaining questions refer to the following Java class:

```
class Fact {
    static int fact (int i) {
        if (i > 1) return Fact.fact(i-1) * i;
        return 1;
    }
}
```

2. (IR trees; 10%) Draw an intermediate code tree for the method `Fact.fact`. Assume that the word size of the target machine is 4 bytes, and that the result is returned in register temporary `$a0` (as on the PowerPC). You may use named temporaries for each of the local variables or formal parameters in the method (e.g., parameter `i` would be represented as temporary `i`). For unnamed intermediate results use numbered temporaries (`t.1`, `t.2`, etc.). For labels use numbered labels (`L.1`, `L.2`, etc.).

   Answer:

```
SEQ(
 SEQ(
  SEQ(
   SEQ(
    BGT(TEMP i, CONST 1, L.4, L.3),
    SEQ(
     SEQ(
      LABEL L.4,
      SEQ(
       MOVE(
        TEMP $a0,
        MUL(CALL(NAME Fact.fact, SUB(TEMP i, CONST 1)), TEMP i)),
       JUMP(NAME L.0))),
     JUMP(NAME L.3))),
   LABEL L.3),
  SEQ(
   MOVE(TEMP $a0, CONST 1),
   JUMP(NAME L.0))),
 LABEL L.0)
```

3. (Canonical trees; 10%) Transform the tree code from your answer to Question 2 into *canonical* trees (i.e., a straight-line sequence of tree *statements* containing no SEQ/ESEQ nodes).

   Answer:

```
BGT(TEMP i, CONST 1, L.4, L.3)
LABEL L.4
MOVE(TEMP t.1, SUB(TEMP i, CONST 1))
MOVE(TEMP t.2, CALL(NAME Fact.fact, TEMP t.1))
MOVE(TEMP $a0, MUL(TEMP t.2, TEMP i))
JUMP(NAME L.0)
JUMP(NAME L.3)
LABEL L.3
MOVE(TEMP $a0, CONST 1)
JUMP(NAME L.0)
LABEL L.0
```

4. (Trace scheduling; 10%) Trace schedule the *basic blocks* of your answer to Question 3 so that each conditional jump is followed immediately by its false target.

   Answer:

```
LABEL L.5
BGT(TEMP i, CONST 1, L.4, L.3)
LABEL L.3
MOVE(TEMP $a0, CONST 1)
JUMP(NAME L.0)
LABEL L.4
MOVE(TEMP t.1, SUB(TEMP i, CONST 1))
MOVE(TEMP t.2, CALL(NAME Fact.fact, TEMP t.1))
MOVE(TEMP $a0, MUL(TEMP t.2, TEMP i))
JUMP(NAME L.0)
LABEL L.6
JUMP(NAME L.3)
LABEL L.0
```

5. (CFGs; 10%) MJ generates the following instructions for this program:

```
        mr i,$a0
L.5:
        cmpwi i,1
        bgt L.4
L.3:
        li t.3,1
        mr $a0,t.3
        b L.0
L.4:
        subi t.4,i,1
        mr t.1,t.4
        mr $a0,t.1
        bl Fact.fact
        mr t.2,$a0
        mullw t.5,t.2,i
        mr $a0,t.5
        b L.0
L.6:
        b L.3
L.0:
```

Identify the *basic blocks* in this code, and draw its control flow graph (CFG) having nodes which are the *basic blocks* and edges representing control flow among them. Remember that a *call* is not treated as a branch in the CFG. For each *basic block*, summarize the temporaries/registers used (before they are defined) and defined by the block.

Answer:

```
0: i <- $a0 ; goto 1
        mr i,$a0

1: <- i ; goto 3 2
   L.5:
        cmpwi i,1
        bgt L.4

2: t.3 $a0 <- ; goto 5
   L.3:
        li t.3,1
        mr $a0,t.3
        b L.0

3: t.4 t.1 $a0 t.2 t.5 <- i ; goto 5
   L.4:
```

```
        subi t.4,i,1
        mr t.1,t.4
        mr $a0,t.1
        bl Fact.fact
        mr t.2,$a0
        mullw t.5,t.2,i
        mr $a0,t.5
        b L.0

4: <- ; goto 2
   L.6:
        b L.3

5: <- $a0 ; goto
   L.0:
```

6. (Liveness analysis; 10%) Assume we are compiling to a target architecture similar to the PowerPC, as we did in the project, but having only the following general-purpose registers:

   - `$a0`: a caller-saved argument/result register
   - `$s0`: a callee-save register

   Draw the CFG for the individual *instructions* in the code below, having nodes which are the *instructions* and edges representing control flow among them. Compute *liveness* information for each *instruction* in the code, as described in class, by tracing from uses back to definitions, being careful to propagate along both edges at a merge point in the CFG. Remember that a call instruction uses its argument registers and defines all argument/result registers. I have already labeled the variables/registers *used* and *defined* by each instruction as "`def <- use`". Moves are marked as "`def <= use`".

```
                          DEF <- USE

      mr i,$a0            i <= $a0
L.5:
      cmpwi i,1           <- i

      bgt L.4             <- : goto L.4 L.3
L.3:
      li t.3,1            t.3 <-

      mr $a0,t.3          $a0 <= t.3

      b L.0               <- : goto L.0
L.4:
      subi t.4,i,1        t.4 <- i

      mr t.1,t.4          t.1 <= t.4

      mr $a0,t.1          $a0 <= t.1

      bl Fact.fact        $a0 <- $a0

      mr t.2,$a0          t.2 <= $a0

      mullw t.5,t.2,i     t.5 <- t.2 i

      mr $a0,t.5          $a0 <= t.5

      b L.0               <- : goto L.0
L.6:
      b L.3               <- : goto L.3
L.0:
```

Answer:

		DEF <- USE	LIVEIN
	mr i,$a0	i <= $a0	$a0
L.5:	cmpwi i,1	<- i	i
	bgt L.4	<- : goto L.4 L.3	i
L.3:	li t.3,1	t.3 <-	t.3
	mr $a0,t.3	$a0 <= t.3	$a0
	b L.0	<- : goto L.0	$a0
L.4:	subi t.4,i,1	t.4 <- i	i
	mr t.1,t.4	t.1 <= t.4	i t.4
	mr $a0,t.1	$a0 <= t.1	i t.1
	bl Fact.fact	$a0 <- $a0	i $a0
	mr t.2,$a0	t.2 <= $a0	i $a0
	mullw t.5,t.2,i	t.5 <- t.2 i	i t.2
	mr $a0,t.5	$a0 <= t.5	t.5
	b L.0	<- : goto L.0	$a0
L.6:	b L.3	<- : goto L.3	$a0
L.0:			$a0

7. (Interference graphs; 10%) Fill in the following adjacency table representing the interference graph for the program; an entry in the table should contain an $\times$ if the variable in the left column interferes with the corresponding variable/register in the top row. Since machine registers are pre-colored, we choose to omit adjacency information for them. Naturally, you must still record if a non-precolored node (variable) interferes with a pre-colored node (register); the columns for pre-colored nodes are there for that purpose.

   Also, record the *unconstrained* move-related nodes in the table by placing an $\circ$ in any empty entry where the variable in the left column is the source or target of any move involving the variable/register in the top row. **Nodes that are move-related should not interfere if their live ranges overlap only starting at the move and neither is subsequently redefined**.

	`$a0`	`$s0`	`i`	`t.1`	`t.2`	`t.3`	`t.4`	`t.5`
`i`								
`t.1`								
`t.2`								
`t.3`								
`t.4`								
`t.5`								

Answer:

	`$a0`	`$s0`	`i`	`t.1`	`t.2`	`t.3`	`t.4`	`t.5`
`i`	$\times$			$\times$	$\times$		$\times$	
`t.1`	$\circ$		$\times$				$\circ$	
`t.2`	$\circ$		$\times$					
`t.3`	$\circ$							
`t.4`			$\times$	$\circ$				
`t.5`	$\circ$							

8. (Register allocation; 10%) The register allocator determines the following register assignments: `i->$s0`, `t.1->$a0`, `t.2->$a0`, `t.3->$a0`, `t.4->$a0`, `t.5->$a0`. Write the resulting assembly code program, eliminating any redundant move instructions, in between the method prologue/epilogue below: ls1

```
Fact.fact:
__framesize=48
        mflr $zt
        stw $zt,8($sp)
        stmw $s0,-4($sp)
        stwu $sp,-__framesize($sp)
```

Answer:

```
        mr $s0,$a0
    L.5:
            cmpwi $s0,1
            bgt L.4
    L.3:
            li $a0,1
    #       mr $a0,$a0
            b L.0
    L.4:
            subi $a0,$s0,1
    #       mr $a0,$a0
    #       mr $a0,$a0
            bl Fact.fact
    #       mr $a0,$a0
            mullw $a0,$a0,$s0
    #       mr $a0,$a0
            b L.0
    L.6:
            b L.3
    L.0:

        addi $sp,$sp,__framesize
        lmw $s0,-4($sp)
        lwz $zt,8($sp)
        mtlr $zt
        blr
```

9. (Register allocation; 5%) Temporary `i` is allocated to callee-save register `$s0`. Why does `i` end up in this register? What does the prolog/epilog code do to permit this assignment? What would happen if the machine had no callee-save registers?

   Answer:

   `i` has a lifetime that spans the recursive call to `Fact.fact`, so it needs to go in a callee-save register `$s0` or be spilled. As a result, `$s0` must be saved and restored in the prolog. If there were no callee-save registers, `i` would have to spill to memory, and be loaded/stored on each use/definition.