CS 352 – Compilers: Principles and Practice
Final Examination, 05/02/05

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 four (4) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature:

1. (Compiler phases; 10%) The MiniJava compiler you worked on this semester manipulates several representations of a program as it is compiled. The initial input is a MiniJava source program. For each of the following program representations name the compiler passes (there may be more than one!) that take that representation as *input*:

   (a) program source text

   Answer:

   scanning (lexical analysis)

   (b) tokens

   Answer:

   parsing (syntactic analysis)

   (c) abstract syntax trees (ASTs)

   Answer:

   type checking (semantic analysis)
   translation

   (d) intermediate code *trees*

   Answer:

   canonicalization

   (e) intermediate code *statements* (*ie*, tuples)

   Answer:

   instruction selection (code generation)

   (f) assembly language instructions

   Answer:

   control flow analysis

   (g) control flow graph (CFG)

   Answer:

   data flow (liveness) analysis

   (h) interference graph

   Answer:

   graph coloring register allocation

   (i) colored interference graph + assembly language instructions

   Answer:

   code emission

2. (Runtime management; 25%) Consider the MiniJava program given below:

```
class Tree {
    public static void main (String[] a)
        throws java.io.IOException
    {
        Tree root = null;
        for (Tree n = Tree.readint(); n != null; n = Tree.readint())
            root = n.insert (root);
        if (root != null) root.print ();
        System.out.println("");
    }

    int value;
    Tree left, right;
    Tree insert(Tree root) {
        if (root == null) return this;
        if (this.value <= root.value)
            root.left = this.insert (root.left);
        else
            root.right = this.insert(root.right);
        return root;
    }
    void print() {
        System.out.write('(');
        if (this.left != null) this.left.print();
        Tree.printint(this.value);
        if (this.right != null) this.right.print();
        System.out.write(')');
    }

    static Tree readint () throws java.io.IOException {
        Tree result = null;
        int buffer = System.in.read();
        while (buffer == ' ')
            buffer = System.in.read();
        if (buffer >= '0' && buffer <= '9') {
            result = new Tree();
            result.value = 0;
            while (buffer >= '0' && buffer <= '9') {
                result.value = result.value * 10 + buffer - '0';
                buffer = System.in.read();
            }
        }
        return result;
    }
    static void printint(int i) {
        if (i > 0) {
            Tree.printint(i/10);
            System.out.write(i-i/10*10+'0');
        }
    }
}
```

(a) (5%) Given input:

```
45 33 24 22 22 10
```

what output does this program produce?

Answer:

It prints out:

```
((((((10)22)22)24)33)45)
```

(b) (20%) Assuming the same input, show a diagram of PowerPC stack frames *at the point* where `printint` has returned to `print` having printed out the string “10” to standard output. Indicate where *all* the local and heap variables are (assume all local variables are stored in memory in the activation records, not in registers, and that all arguments are passed in the stack, not in registers); show the value of all integer variables, as well as variables containing references to the heap, and the object they refer to.

Answer:

```
         STACK                        HEAP
       +---------+
main:  | a     --+-------------------> array of String
       | root  --+--------------+---------------------> 45
       | ...     |              |                      /  \
       +---------+              | +-----------------> 33  null
print: | this  --+--------------+ |                  /  \
       | ...     |                | +-------------> 24  null
       +---------+                | |              /  \
print: | this  --+----------------+ | +---------> 22  null
       | ...     |                  | |          /  \
       +---------+                  | | +-----> 22  null
print: | this  --+------------------+ | |      /  \
       | ...     |                    | | +-> 10  null
       +---------+                    | | |
print: | this  --+--------------------+ | |
       | ...     |                      | |
       +---------+                      | |
print: | this  --+----------------------+ |
       | ...     |                        |
       +---------+                        |
print: | this  --+------------------------+
       | ...     |
       +---------+
```

3. (IR trees, canonicalization, instruction selection; 25%) In this question you may assume that the word size of the target machine is 4 bytes and that any result is returned in (PowerPC register) temporary `r3`. You may use named temporaries for each of the local variables or formal parameters in the method (*eg*, parameter `b` would be allocated to temporary `_b`). For unnamed intermediate results use numbered temporaries (`t0`, `t1`, etc.). **Do not worry about checking for run-time errors such as array index out of bounds or null pointer dereference.**

   Consider the following MiniJava program:

```
class A {
    String name;
    A init() { this.name = "A"; return this; }
    String name() { return this.name; }
}
class B extends A {
    String name;
    B init() { super.init(); this.name = "B"; return this; }
    String name() { return this.name; }
}
class Main {
    static String Aname(A a) { return a.name(); }
    static String Bname(B b) { return b.name(); }
    public static void main (String[] args) {
        A a = new A().init();
        System.out.println(a.name);
        System.out.println(Main.Aname(a));

        B b = new B().init();
        System.out.println(b.name);
        System.out.println(Main.Bname(b));

        a = b;
        System.out.println(a.name);
        System.out.println(Main.Aname(a));
    }
}
```

(a) (5%) What output does this program produce?

Answer:

```
A
A
B
B
A
B
```

(b) (10%) Draw an intermediate code tree for each of the following methods:

i. (2%) Method `name` in class `A` (*ie*, `A.name`).
Answer:

```
MOVE(TEMP r3, MEM(TEMP _this, 0))
```

ii. (2%) Method `name` in class `B` (*ie*, `B.name`).
Answer:

```
MOVE(TEMP r3, MEM(TEMP _this, 4))
```

iii. (3%) Method `Aname` in class `Main` (*ie*, `Main.Aname`).
Answer:

```
MOVE(TEMP r3, CALL(MEM(MEM(TEMP _a, -4), 4), TEMP _a))
```

iv. (3%) Method `Bname` in class `Main` (*ie*, `Main.Bname`).
Answer:

```
MOVE(TEMP r3, CALL(MEM(MEM(TEMP _b, -4), 4), TEMP _b))
```

(c) (10%) For the following, generate PowerPC instructions using “maximal munch”, circling the tiles and numbering them *in the order that they are “munched”* (*ie*, the order that instructions are emitted for the tile). Leave the temporary names in place (do not assign registers).

i. (4%) Generate instructions for the tree of Question 3(b)ii.
Answer:

```
[2 = MOVE(TEMP r3, [1 = MEM(TEMP _this, 4)])]
```

```
1: lwz t0,4(_this)
2: mr r3,t0
```

ii. (6%) Generate instructions for the tree of Question 3(b)iv.
Answer:

```
[4 = MOVE(TEMP r3,
          [3 = CALL([2 = MEM([1 = MEM(TEMP _b, -4)], 4)],
                    TEMP _b)])]
```

```
1: lwz t0,-4(_b)
2: lwz t1,4(t0)
3: mr r3,_b
   mtctr t1
   bctrl
4: mr r3,r3
```

4. (Control flow graphs, liveness analysis, register allocation; 40%) The following program has been compiled for a machine with 2 registers:

- $\mathtt{r}_1$: a callee-save register
- $\mathtt{r}_2$: a caller-save argument/result register

```
      s ← r1
      a ← r2
      i ← a
      i ← i − 1
      if i < 0 goto L1
L0 :  a ← i
      r2 ← a
      call f              (use r2, def r2)
      i ← i − 1
      if i ≥ 0 goto L0
L1 :  r1 ← s
      return              (use r1, r2)
```

(a) (5%) Draw the control flow graph for this program, with nodes that are the program's *basic blocks* (*ie*, not individual instructions) and with edges representing the flow of control between the basic blocks.

Answer:

```
    s  := r1
    a  := r2
    i  := a
    i  := i - 1
    if i < 0 goto L1 -----+
       |                  |
       v                  |
L0: a  := i <-----------+ |
    r2 := a             | |
    call f              | |
    i  := i - 1         | |
    if i >= 0 goto L0 --+ |
       |                  |
       v                  |
L1: r1 := s <-------------+
    return
```

(b) (10%) Annotate each *instruction* with the variables/registers live-out at that instruction.

		Def	Use	LiveOut
	$s \leftarrow \mathtt{r}_1$			
	$a \leftarrow \mathtt{r}_2$			
	$i \leftarrow a$			
	$i \leftarrow i-1$			
	if $i<0$ goto L_1			
L_0 :	$a \leftarrow i$			
	$\mathtt{r}_2 \leftarrow a$			
	call f			
	$i \leftarrow i-1$			
	if $i \geq 0$ goto L_0			
L_1 :	$\mathtt{r}_1 \leftarrow s$			
	return			

Answer:

		Def	Use	LiveOut
	$s \leftarrow \mathtt{r}_1$	s	$\mathtt{r}_1$	$s\mathtt{r}_2$
	$a \leftarrow \mathtt{r}_2$	a	$\mathtt{r}_2$	$sa\mathtt{r}_2$
	$i \leftarrow a$	i	a	$si\mathtt{r}_2$
	$i \leftarrow i-1$	i	i	$si\mathtt{r}_2$
	if $i<0$ goto L_1		i	$si\mathtt{r}_2$
L_0 :	$a \leftarrow i$	a	i	sai
	$\mathtt{r}_2 \leftarrow a$	$\mathtt{r}_2$	a	$si\mathtt{r}_2$
	call f	$\mathtt{r}_2$	$\mathtt{r}_2$	$si\mathtt{r}_2$
	$i \leftarrow i-1$	i	i	$si\mathtt{r}_2$
	if $i \geq 0$ goto L_0		i	$si\mathtt{r}_2$
L_1 :	$\mathtt{r}_1 \leftarrow s$	$\mathtt{r}_1$	s	$\mathtt{r}_1\mathtt{r}_2$
	return		$\mathtt{r}_1\mathtt{r}_2$	

(c) (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 interferes with a pre-colored node; 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. **Remember that nodes that are move-related should not interfere if their live ranges overlap only starting at the move and neither is subsequently redefined**.

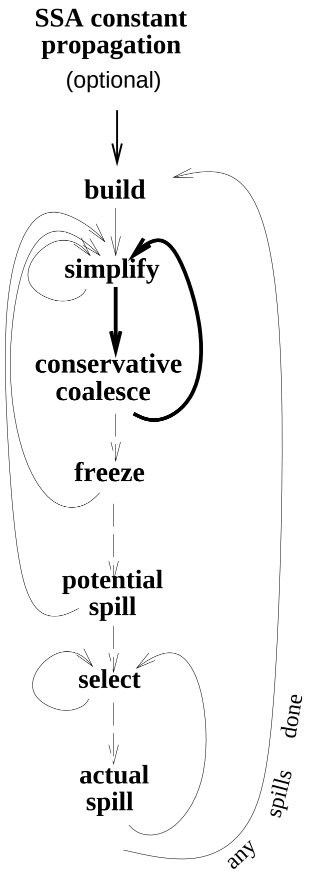
	$\mathtt{r}_1$	$\mathtt{r}_2$	s	i	a
s					
i					
a					

Answer:

	$\mathtt{r}_1$	$\mathtt{r}_2$	s	i	a
s	$\circ$	$\times$		$\times$	$\times$
i		$\times$	$\times$		$\circ$
a		$\circ$	$\times$	$\circ$	

(d) (15%) Show the steps of a coalescing graph-coloring register allocator as it assigns registers to the variables in the program. Use the *George criterion* for coalescing nodes: node *a* can be coalesced with node *b* only if all significant-degree (*ie*, degree $>= K$) neighbors of *a* already interfere with *b*. Show the final program, noting any redundant moves.

The flow diagram for iterated register coalescing is included here for your reference.



Answer:

i. Coalesce *a* with $\texttt{r}_2$ (or *a* with *i*) using George criterion (all of *a*'s neighbors already conflict with $\texttt{r}_2$):

	$\texttt{r}_1$	$\texttt{r}_2a$	*s*	*i*
s	○	×		×
i		×	×	

ii. Cannot coalesce *s* since *i* does not interfere with $\texttt{r}_1$; cannot freeze *a* since it is high-degree; so potential spill *s* (fewest uses/defs)

	$\texttt{r}_1$	$\texttt{r}_2a$	*i*
i		×	

iii. Simplify *i*

iv. Select $i \equiv \mathtt{r}_1$

	$\mathtt{r}_1$	$\mathtt{r}_2 a$	i
i		$\times$	

v. No color for $s \Rightarrow$ actual spill.
Rewrite code, retaining coalescences from before spill:

		Def	Use	LiveOut
	$\mathcal{M}[loc_s] \leftarrow \mathtt{r}_1$		$\mathtt{r}_1$	$\mathtt{r}_2$
	$i \leftarrow \mathtt{r}_2$	i	$\mathtt{r}_2$	$i\mathtt{r}_2$
	$i \leftarrow i - 1$	i	i	$i\mathtt{r}_2$
	if $i < 0$ `goto` L_1		i	$i\mathtt{r}_2$
L_0 :	$\mathtt{r}_2 \leftarrow i$	$\mathtt{r}_2$	i	$i\mathtt{r}_2$
	`call` f	$\mathtt{r}_2$	$\mathtt{r}_2$	$i\mathtt{r}_2$
	$i \leftarrow i - 1$	i	i	$i\mathtt{r}_2$
	if $i \geq 0$ `goto` L_0		i	$i\mathtt{r}_2$
L_1 :	$\mathtt{r}_1 \leftarrow \mathcal{M}[loc_s]$	$\mathtt{r}_1$		$\mathtt{r}_1\mathtt{r}_2$
	return		$\mathtt{r}_1\mathtt{r}_2$	

vi. New adjacency table:

	$\mathtt{r}_1$	$\mathtt{r}_2$	i
i		$\times$	

vii. Simplify i

viii. Select $i \equiv \mathtt{r}_1$

ix. Resulting code:

```
      M[loc_s] ← r1
      r1 ← r2
      r1 ← r1 − 1
      if r1 < 0 goto L1
L0 :  r2 ← r1
      call f
      r1 ← r1 − 1
      if r1 ≥ 0 goto L0
L1 :  r1 ← M[loc_s]
      return
```