CS 352 – Compilers: Principles and Practice
Final Examination, 05/03/04

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. Describe the program representations that are generated as *output* from each of the following compiler phases:

   (a) scanning (lexical analysis)
   Answer:

   tokens

   (b) parsing (syntactic analysis)
   Answer:

   abstract syntax trees (ASTs)

   (c) type checking (semantic analysis)
   Answer:

   annotated ASTs, symbol tables

   (d) translation
   Answer:

   intermediate code trees (IR), frame layout

   (e) canonicalization
   Answer:

   IR statements

   (f) code generation
   Answer:

   assembly language instructions

   (g) control flow analysis
   Answer:

   control flow graph (CFG)

   (h) data flow analysis (liveness)
   Answer:

   interference graph

   (i) graph coloring register allocation
   Answer:

   colored interference graph

   (j) code emission
   Answer:

   assembly language

2. (Runtime management; 20%) Consider the MiniJava program given below:

```
class List {
    public static void main (String[] args) {
        List list;
        list = new List().init(0, list);
        list = new List().init(1, list);
        list = new List().init(2, list);
        list = new List().init(3, list);
        list.print();
    }
    int value;
    List next;
    List init(int value, List next) {
        this.value = value;
        this.next = next;
        return this;
    }
    void printint(int i) {
        if (i > 0) { this.printint(i/10); System.out.write(i-i/10*10+'0'); }
    }
    void printvalue() {
        int i = this.value;
        if (i == 0) System.out.write('0');
        else if (i < 0) { System.out.write('-'); this.printint(-i); }
        else this.printint(i);
    }
    void print() {
        this.printvalue();
        System.out.write(' ');
        if (this.next != null) this.next.print();
    }
}
```

(a) (5%) What does this program do?

Answer:

It prints out the string “3 2 1 0”.

(b) (15%) Show a diagram of MIPS stack frames *at the point* where `printvalue` has just returned to method `print` after writing the character ’0’ to standard output. Indicate where *all* the local and heap variables are (assume all local variables are stored 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:

```
       +---------+
main:  | args  --+--------------------> array of String
       | list  --+--------------------> (3, -> (2, -> (1, -> (0, null))))
       | ...     |                       ^      ^      ^      ^
       +---------+                       |      |      |      |
       | this  --+-----------------------+      |      |      |
print: | ...     |                              |      |      |
       +---------+                              |      |      |
       | this  --+------------------------------+      |      |
print: | ...     |                                     |      |
       +---------+                                     |      |
       | this  --+-------------------------------------+      |
print: | ...     |                                            |
       +---------+                                            |
       | this  --+--------------------------------------------+
print: | ...     |
       +---------+
```

3. (ASTs, IR trees, canonicalization, instruction selection; 35%) Consider the following MiniJava classes:

```
class Number {
    int i;
}
class Integer extends Number {
    int i;
    int init(int i, int j) {
        return this.i = i + j;
    }
}
```

(a) (5%) Draw the MJ abstract syntax tree (AST) for the method `init`.

Answer:

(b) (10%) Draw an intermediate code tree for the method `init`. Assume that the word size of the target machine is 4 bytes and that any result is returned in (MIPS register) temporary `$v0`. You may use named temporaries for each of the local variables or formal parameters in the method (*eg*, parameter `i` would be allocated to temporary `i`). 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.**

Answer:

```
MOVE(TEMP $v0,
     ESEQ(MOVE(MEM(ADD(TEMP this, CONST 4)),
               ESEQ(MOVE(TEMP t0, ADD(TEMP i, TEMP j)), TEMP t0)),
          TEMP t0))
```

(c) (10%) Transform the tree code from your answer to 3b into *canonical* trees (*ie*, a straight-line list of tree *statements* containing no SEQ/ESEQ nodes).

Answer:

```
MOVE(TEMP t1, ADD(TEMP this, CONST 4))
MOVE(TEMP t0, ADD(TEMP i, TEMP j))
MOVE(MEM(TEMP t1), TEMP t0)
MOVE(TEMP $v0, TEMP t0)
```

(d) (10%) Generate MIPS instructions from your answer to 3c, using “maximal munch”. Circle the tiles and number 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).

Answer:

```
[2 = MOVE(TEMP t1, [1 = ADD(TEMP this, CONST 4) ] ) ]
[4 = MOVE(TEMP t0, [3 = ADD(TEMP i, TEMP j) ] ) ]
[5 = MOVE(MEM(TEMP t1), TEMP t0) ]
[6 = MOVE(TEMP $v0, TEMP t0) ]
```

```
        add t2 this 4
        move t1 t2
        add t3 i j
        move t0 t3
        sw t0 (t1)
        move $v0 t0
```

4. (Control flow graphs, liveness analysis, register allocation; 35%) The following program has been compiled for a machine with 2 registers:

   - r_1: a callee-save register
   - r_2: a caller-save argument/result register

```
      $a \leftarrow r_1$
      $b \leftarrow r_2$
      $r_2 \leftarrow b$
      $r_2 \leftarrow f(r_2)$         ($call\ to\ function\ f$)
      $c \leftarrow r_2$
      if $b > N$ goto $L_1$   ($N\ is\ a\ constant$)
      $d \leftarrow b - c$
      goto $L_2$
$L_1$:  $d \leftarrow b$
      $c \leftarrow b + d$
$L_2$:  $b \leftarrow c * d$
      $r_2 \leftarrow b$
      $r_1 \leftarrow a$
      return                 (uses $r_1$, $r_2$)
```

   (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:

```
     a := r1
     b := r2
     r2:= b
     r2:= f(r2)
     c := r2
     if b > N goto L1 -----+
        |                  |
        v                  v
     d := b-c        L1: d := b
     goto L2             c := b + d
        |                  |
        v                  |
L2:  b := c*d <------------+
     r2 := b
     r1 := a
     return
```

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

		Def	Use	LiveOut
	$a \leftarrow \mathtt{r}_1$			
	$b \leftarrow \mathtt{r}_2$			
	$\mathtt{r}_2 \leftarrow b$			
	$\mathtt{r}_2 \leftarrow f(\mathtt{r}_2)$			
	$c \leftarrow \mathtt{r}_2$			
	if $b > N$ goto L_1			
	$d \leftarrow b - c$			
	goto L_2			
L_1 :	$d \leftarrow b$			
	$c \leftarrow b + d$			
L_2 :	$b \leftarrow c * d$			
	$\mathtt{r}_2 \leftarrow b$			
	$\mathtt{r}_1 \leftarrow a$			
	return			

Answer:

		Def	Use	LiveOut
	$a \leftarrow \mathtt{r}_1$	a	$\mathtt{r}_1$	$a\mathtt{r}_2$
	$b \leftarrow \mathtt{r}_2$	b	$\mathtt{r}_2$	ab
	$\mathtt{r}_2 \leftarrow b$	$\mathtt{r}_2$	b	$ab\mathtt{r}_2$
	$\mathtt{r}_2 \leftarrow f(\mathtt{r}_2)$	$\mathtt{r}_2$	$\mathtt{r}_2$	$ab\mathtt{r}_2$
	$c \leftarrow \mathtt{r}_2$	c	$\mathtt{r}_2$	abc
	if $b > N$ goto L_1		b	abc
	$d \leftarrow b - c$	d	bc	acd
	goto L_2			acd
L_1 :	$d \leftarrow b$	d	b	abd
	$c \leftarrow b + d$	c	bd	acd
L_2 :	$b \leftarrow c * d$	b	cd	ab
	$\mathtt{r}_2 \leftarrow b$	$\mathtt{r}_2$	b	$a\mathtt{r}_2$
	$\mathtt{r}_1 \leftarrow a$	$\mathtt{r}_1$	a	$\mathtt{r}_1\mathtt{r}_2$
	return		$\mathtt{r}_1\mathtt{r}_2$	

(c) (5%) 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**.

	$\mathtt{r}_1$	$\mathtt{r}_2$	*a*	*b*	*c*	*d*
a						
b						
c						
d						

Answer:

	$\mathtt{r}_1$	$\mathtt{r}_2$	*a*	*b*	*c*	*d*
a	○	×		×	×	×
b		×	×		×	○
c		○	×	×		×
d			×	○	×	

(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.

Answer:

i. Initial adjacency table:

	$\mathtt{r}_1$	$\mathtt{r}_2$	*a*	*b*	*c*	*d*
a	○	×		×	×	×
b		×	×		×	○
c		○	×	×		×
d			×	○	×	

ii. Coalesce *d* with *b* using George criterion (all of *d*'s neighbors already conflict with *b*):

	$\mathtt{r}_1$	$\mathtt{r}_2$	*a*	*bd*	*c*
a	○	×		×	×
bd		×	×		×
c		○	×	×	

iii. Coalesce c with r_2:

	$\mathtt{r}_1$	$\mathtt{r}_2c$	a	bd
a	$\circ$	$\times$		$\times$
bd		$\times$	$\times$	

iv. Cannot coalesce a, cannot freeze high-degree a, so potential spill a (fewest uses/defs):

	$\mathtt{r}_1$	$\mathtt{r}_2c$	bd
bd		$\times$	

v. Simplify bd

vi. Select $bd \equiv \mathtt{r}_1$

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

		Def	Use	LiveOut
	$\mathcal{M}[loc_a] \leftarrow \mathtt{r}_1$		$\mathtt{r}_1$	$\mathtt{r}_2$
	$b \leftarrow \mathtt{r}_2$	b	$\mathtt{r}_2$	b
	$\mathtt{r}_2 \leftarrow b$	$\mathtt{r}_2$	b	$b\mathtt{r}_2$
	$\mathtt{r}_2 \leftarrow f(\mathtt{r}_2)$	$\mathtt{r}_2$	$\mathtt{r}_2$	$b\mathtt{r}_2$
	if $b > N$ goto L_1		b	$b\mathtt{r}_2$
	$b \leftarrow b - \mathtt{r}_2$	b	$b\mathtt{r}_2$	$b\mathtt{r}_2$
	goto L_2			$b\mathtt{r}_2$
L_1 :	$\mathtt{r}_2 \leftarrow b + b$	$\mathtt{r}_2$	b	$b\mathtt{r}_2$
L_2 :	$b \leftarrow \mathtt{r}_2 * b$	b	$b\mathtt{r}_2$	b
	$\mathtt{r}_2 \leftarrow b$	$\mathtt{r}_2$	b	$\mathtt{r}_2$
	$\mathtt{r}_1 \leftarrow \mathcal{M}[loc_a]$	$\mathtt{r}_1$		$\mathtt{r}_1\mathtt{r}_2$
	return		$\mathtt{r}_1\mathtt{r}_2$	

viii. New adjacency table:

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

ix. Simplify b

x. Select $b \equiv \mathtt{r}_1$

xi. Resulting code:

```
      $\mathcal{M}[loc_a] \leftarrow \mathtt{r}_1$
      $\mathtt{r}_1 \leftarrow \mathtt{r}_2$
      $\mathtt{r}_2 \leftarrow \mathtt{r}_1$
      $\mathtt{r}_2 \leftarrow f(\mathtt{r}_2)$
      if $\mathtt{r}_1 > N$ goto $L_1$
      $\mathtt{r}_1 \leftarrow \mathtt{r}_1 - \mathtt{r}_2$
      goto $L_2$
$L_1$ : $\mathtt{r}_2 \leftarrow \mathtt{r}_1 + \mathtt{r}_1$
$L_2$ : $\mathtt{r}_1 \leftarrow \mathtt{r}_2 * \mathtt{r}_1$
      $\mathtt{r}_2 \leftarrow \mathtt{r}_1$
      $\mathtt{r}_1 \leftarrow \mathcal{M}[loc_a]$
      return
```