CS 502 – Compiling and Programming Systems
Final and Qualifying Examination, 5/3/11

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 (*ie*, your grade will be the percentage of your answers that are correct).

This exam is **open book, open notes**. You are free to refer to any book or other study materials you bring to the exam room.

You have **2 hours** to complete all five (5) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature:

1. (LALR parsing; 20%) Consider the following grammar:

$$
\begin{aligned}
S' &\rightarrow S \\
S &\rightarrow aAd \mid bBd \mid aBe \mid bAe \\
A &\rightarrow c \\
B &\rightarrow c
\end{aligned}
$$

   (a) (10%) Is this grammar LR(1)? Why or why not?

   (b) (5%) Is it LALR(1)? Why or why not?

   (c) (5%) Suppose we have an LR(1) grammar (i.e., one whose LR(1) items produce no parsing action conflicts). Explain why the LALR(1) table construction for any LR(1) grammar cannot introduce shift-reduce conflicts.

2. (Run-time management; 20%) Consider the following M3 program as it executes:

```
1  MODULE List;
2  TYPE
3    T = OBJECT
4      head: INTEGER;
5      tail: T;
6    METHODS
7      merge(t: T): T := Merge;
8    END;
9  PROCEDURE Cons(head: INTEGER; tail: T): T =
10   VAR t := NEW(T);
11   BEGIN
12     t.head := head;
13     t.tail := tail;
14     RETURN t;
15   END Cons;
16 PROCEDURE Merge(this, other: T): T =
17   BEGIN
18     IF other = NIL THEN
19       RETURN this;
20     ELSIF this.head < other.head THEN
21       RETURN Cons(this.head, other.merge(this.tail));
22     ELSE
23       RETURN Cons(other.head, this.merge(other.tail));
24     END;
25   END Merge;
26 VAR
27   t1 := Cons(1, Cons(3, NIL));
28   t2 := Cons(2, Cons(4, NIL));
29   t := t1.merge(t2);
30 BEGIN
31 END List.
```

(a) (15%) Show a diagram of MIPS stack frames at the point when this program is executing line 19 of `Merge` where `other`=**`NIL`**. 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 each variable containing a reference to the heap, and the object it refers to.

(b) (5%) Show a diagram of MIPS stack frames at the point when this program has finished executing line 29 after assigning to `t`. 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 each variable containing a reference to the heap, and the object it refers to.

3. (Liveness analysis and register allocation; 25%) 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 := r1
    b := r2
    r2 := b
    call f
    c := r2
    if b > N goto L1                    (* N is a constant *)
    d := b - c
    goto L2
L1:
    d := b
    c := b + d
L2:
    b := c * d
    r2 := b
    r1 := a
    return
```

   (a) (5%) Perform liveness analysis for this program annotating each *instruction* with the temporaries/registers live-out at that instruction.

   (b) (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; you must treat them as having infinite degree. 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.

	r_1	r_2	a	b	c	d
a						
b						
c						
d						

   (c) (15%) Perform iterated coalescing register allocation for this program, showing the steps in detail (you will want to copy and amend the above adjacency table at each step to keep track of the interference graph as it changes), and the final code for the program. Use the George criterion for coalescing.

4. (Loop optimization; 15%) Consider the following C code fragment:

```
int i, j;
int A[50,100];
int B[50], Y[50];
int X[100];

for (i = 50; --i >= 0; ) {
  Y[i] = B[i];
  for (j = 100; --j >= 0; )
    Y[i] += A[i,j] * X[j];
}
```

   (a) (Control flow graphs; 5%) Draw a control flow graph for the code fragment, with basic blocks as nodes, and making addressing code explicit. For example, the array access `A[i,j]` should be written `*(A+100*i+j)`.

   (b) (Factoring invariants; 5%) Using your previous answer, identify loop invariant expressions and factor loop invariants.

   (c) (Strength reduction; 5%) Using your previous answer, identify loop induction expressions and loop induction variables, then perform strength reduction. Do this *only* for the inner loop.

5. (Run-time management, data-flow analysis; 20%) Consider the difference between 'boxed' and 'unboxed' data. Boxed data is allocated explicitly on the heap whereas unboxed data is allocated statically (for globals) or dynamically on the stack (for locals). For example, in M3, a programmer will write:

```
VAR v: T;
```

to allocate an unboxed (local or global) value of type `T`, and

```
VAR v := NEW(REF T);
```

to allocate a boxed (heap) value of type `T`.

Now, consider the case when the boxed value is manipulated, but its pointer never *escapes* the procedure that allocated it. For example:

```
PROCEDURE foo(): T =
VAR v := NEW(REF T);
BEGIN
  v^ := ...;
  doSomething(v^); (* no escape even if the target of v is passed by reference *)
  RETURN v^;
END foo;
```

The programmer could just as easily write the following equivalent code:

```
PROCEDURE foo(): INTEGER =
VAR v: T;
BEGIN
  v := ...;
  doSomething(v);
  RETURN v;
END foo;
```

This would probably be more efficient because the storage for `v` can be allocated automatically in the stack frame for `foo`. Unfortunately, languages like Java do not allow unboxed objects or arrays, requiring that they always be allocated using `new`. Yet, modern Java virtual machines support automatic unboxing of objects and arrays whenever possible. Underlying this optimization is a data flow analysis called *escape analysis* which determines when a reference to locally-allocated storage may escape.

Your task is to devise an intra-procedural escape analysis for heap-allocated data in M3. State your assumptions carefully. For simplicity, you should assume that any statement that may store a reference to a heap variable causes that reference to escape. For example, the assignment `p.f := v` in the following program may cause the reference held by the variable `v` to escape:

```
TYPE P = REF RECORD f: REF T END;
PROCEDURE bar(p: P) =
VAR v := NEW(REF T);
BEGIN
```

```
  p.f := v;
END bar;
```

In devising your analysis be sure to answer each of the following questions:

(a) What information does your data flow analysis propagate?

(b) Is your analysis forward flow or backward flow?

(c) What statements generate flow information and what statements kill flow information?

(d) How do you handle flow merges? Is the flow any-path or all-paths?

(e) Finally, describe how you might improve your analysis to take account of the fact that an assignment `p.f := v` need not cause the reference `v` to escape if `p` itself does not escape. For example:

```
1 PROCEDURE baz() =
2 VAR
3   p := NEW(REF RECORD f: REF T END);
4   v := NEW(REF T);
5 BEGIN
6   p.f := v;
7 END baz;
```

does not lead to the boxed `T` value allocated on line 4 escaping because the target of `p` itself doesn't escape.