

Name: \_\_\_\_\_

USC loginid (e.g., ttrojan): \_\_\_\_\_

## CS 410 Final Exam Fall 2004 [Bono]

Dec 8, 2004

There are 10 problems on the exam, with 100 points total available. There are 12 pages to the exam, including this one; make sure you have all of them. In addition there is a separate one-page handout that goes with the exam. If you need additional space to write any answers, you may use the backs of pages (just direct us to look there).

Put your name and USC loginid at the top of the exam. Please read over the whole test before beginning. Good luck!

	<b>value</b>	<b>score</b>
Problem 1	16 pts.	
Problem 2	5 pts.	
Problem 3	10 pts.	
Problem 4	8 pts.	
Problem 5	6 pts.	
Problem 6	5 pts.	
Problem 7	13 pts.	
Problem 8	12 pts.	
Problem 9	10 pts.	
Problem 10	15 pts.	
<b>TOTAL</b>	100 pts.	

## **Problem 1 [16 pts.]**

Short-answer problems

**Part A.** Regular expressions are used in what phase of compilation?

**Part B.** Give a reason why someone might implement a LL(1) parser over an LALR(1) parser?

**Part C.** Loop optimizations are a general subcategory of

- a. local optimizations
- b. global optimizations
- c. peephole optimizations

**Part D.** We wrote semantic rules (a.k.a. semantic actions) in our compiler project to do what?

**Part E.** When writing a semantic rule, we associate it with what?

**Part F.** Did our compiler project involve handcrafting a DFA? \_\_\_\_\_  
If so, why, if not, why not?

**Part G.** Our symbol table data structure consisted of a stack of lookup structures (our implementation used the STL map as the lookup structure). The stack part of this is necessary for implementing what common feature of programming languages?

**Part H.** What feature(s) of Espresso (and Java) prevents us from writing a one-pass compiler for the language?

**Part J.** True or False: A statement-by-statement scheme is a good way to produce efficient final code from three-address code.

**Problem 2 [5 pts.]**

The optimization called *constant propagation* is just like copy propagation but is done with constants instead of variables.

**Part A.** Apply *only* the constant propagation optimization to the following 3AC sequence. You may write your changes directly in the code.

```
a := 0;
b := 5 + m;
if (a = 0) goto Label
write(y);
t1 := 3 * x
write(t1);
Label: t2 := b - 1;
```

**Part B.** Once we apply constant propagation to the code above it makes another optimization available. In the space below show the version of the code after the second optimization is applied:

### **Problem 3 [10 pts.]**

For each of the following run-time entities, state what part of memory it is stored in at run time. Some of these things could be kept in registers, but for the purposes of this problem, assume the values are kept in memory.

The possible parts of memory are: Code (**C**), Static data (**SD**), Stack (**S**), Heap (**H**). You may use the abbreviations given.

- a. method tables
- b. inherited fields
- c. temporaries
- d. return address
- e. methods
- f. display
- g. access link
- h. integer constants
- i. local variables
- j. register values overwritten by every method

## Problem 4 [8 pts.]

**Part A.** Suppose we are compiling a language that has nested functions and we are using a *display* to manage non-local references. Furthermore, assume the label for the location of the display is **display**.

Using the assumptions given on the separate handout, show the pseudo-MIPS code that would be generated to access a non-local variable using the display. Assume that from function  $f$  we are accessing variable  $x$  that's defined in function  $g$ .

Your answer will be a sequence of one or more MIPS instructions, not the `genCode` function to generate those instructions.

In general, state any additional assumptions you make. In particular, for places where your generated code would use constants, use symbols to stand in for those constants, and explain what each one stands for.

**Part B.** When we use a display what is the additional value we need to store in each activation record? (You can describe this in terms of an activation record for a function  $f$ .)

## Problem 5 [6 pts.]

Consider the following piece of a C++ program. Assume `ints` are one word.

```
struct Foo {
    int x;
    int y;
};

struct Bar {
    int x;
    int y;
    int z;
};

void f1(Foo a, Bar b) // pass by value
{ . . .
}

void f2(Foo &a, Bar &b) // pass by reference
{ . . .
}
```

**Part A.** How much space in words is needed in `f1`'s stack frame for all the parameters total? (pass by value) Briefly explain what the space is used for.

**Part B.** How much space in words is needed in `f2`'s stack frame for all the parameters total? (pass by reference) Briefly explain what the space is used for.

**Part C.** How much space in words is needed in `f3`'s stack frame for all the parameters total if `f3`, has parameters `Foo` and `Bar`, but uses call *by value-return*. Briefly explain what the space is used for.

**Problem 6 [5 pts.]**

Show that the following grammar is ambiguous.

$A \rightarrow B B$

$B \rightarrow B x$

$\quad | x$

### Problem 7 [13 pts.]

Here is the same grammar again:

$A \rightarrow B B$

$B \rightarrow B x$

$\quad | \quad x$

**Part A.** Do any grammar transformations necessary to make the grammar suitable for LL(1) parsing. Circle and label each of the resulting grammar(s) with the transformation(s) you applied (or say “same” for that part if no transformations were necessary).

**Part B.** Show the first and follow sets for each of the non-terminals in the grammar you ended up with in part A.

**Part C.** Attempt to create the LL(1) parse table for the final grammar you gave in part A. Show the complete table: if this grammar is not LL(1), that is if there are conflicts, show all the values that could go in a table entry.

Circle and label your answers to each of the parts.

### Problem 8 [12 pts.]

Here is the same grammar, but now augmented:

$$0) A' \rightarrow A$$

$$1) A \rightarrow B B$$

$$2) B \rightarrow B x$$

$$3) B \rightarrow x$$

If we attempt to build an SLR parser with the grammar, we'll get a conflict. Show exactly where and how this error occurs. I.e., show the relevant sets of items in the DFA and the details and rationale for the conflict. You are not required to (but you may) build the whole DFA or parse tables for this problem.

## Problem 9 [10 pts.]

**Part A [X].** For the following code sequence show the set of variables that are live at the end of each statement  $i$  (called *live-out*( $i$ )) and what's live before the first statement (*live-in*(1)) assuming it is known that  $\{a, b\}$  are live after the last statement.

*live-in*(1) =

(1)  $b := j + 7;$

*live-out*(1) =

(2)  $a := a + x;$

*live-out*(2) =  $\{a, b\}$

**Part A.** Name two variables in the code sequence that would have to be stored in different registers.

**Part B.** Name two variables in the code sequence that could be stored in the same register (or say *none* if there are none).

**Part C.** What does it mean for variables to be *live* at a certain point in the program?

## Problem 10 [15 pts.]

Consider a version of the C `switch` statement with the following syntax:

```
switch expr0 {
  case value1 : stmt1;
  case value2 : stmt2;
  . . .
  case valuen : stmtn;
  default: stmtn+1 // default branch is optional
}
```

The differences between this switch and the real C/C++ switch are that this one is required to have at least one non-default case; the branch values and `expr0` must be type `int`; only one of the cases will be executed (doesn't fall through to the other ones); and only one value may be listed for each statement.

Write a C++ member function `genCode`, below, to generate pseudo-MIPS code for this switch statement, assuming the fields given below. Note: for simplicity, we're representing a list of statements as a bare bones STL vector; in the project it was wrapped in another class.

Here is the AST structure for this construct:

```
class Switch : public Stmt {
  Expr *expr;           // select a branch based on this expr
  vector<int> branchVals;
  vector<Stmt*> branchStmts;
  // implementation invariant: branchVals.size() == branchStmts.size()
  Stmt *defBranchStmt; // optional - may be NULL
public:
  virtual void genCode();
  . . .
}
```

Assumptions about generating code:

- There is a polymorphic `genCode` function defined for all AST types. For the purposes of this problem we're ignoring issues of passing around the environment.
- output your code by using `<<` statements to `cout`. Don't worry about formatting the instructions
- A global variable `lgen` of type `NameGenerator` is available for generating unique labels. The call `lgen.gen()` generates the next unused label (returns type `string`).
- See the additional handout given with this exam for assumptions about the generated code and the assembly language syntax to use.

Review of STL vector syntax. Suppose you have vector `v`:

- `v[i]` access/modify element `i` of `v` if it exists
- `v.push_back(e)` adds element `e` to the end of the vector
- `v.size()` returns the number of elements in the vector
- `v.empty()` to test whether a vector is empty (returns a `bool`)

(Problem continued next page.)

## Problem 10 (cont.)

Here is the AST structure for this construct shown again:

```
class Switch : public Stmt {
    Expr *expr;           // select a branch based on this expr
    vector<int> branchVals;
    vector<Stmt*> branchStmts;
    // implementation invariant: branchVals.size() == branchStmts.size()
    Stmt *defBranchStmt; // optional - may be NULL

public:
virtual void genCode();
    . . .
}
```

---

```
void Switch::genCode()
{
```