

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

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

**CS 410 Final Exam**  
**Spring 2006 [Bono]**  
May 3, 2006

There are 8 problems on the exam, with 132 points total available. There are 13 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	12 pts.	
Problem 2	16 pts.	
Problem 3	10 pts.	
Problem 4A	5 pts.	
Problem 4B	6 pts.	
Problem 4C	20 pts.	
Problem 4D	10 pts.	
Problem 5	15 pts.	
Problem 6	13 pts.	
Problem 7	10 pts.	
Problem 8	15 pts.	
<b>TOTAL</b>	132 pts.	

**Problem 1 [12 pts.]**

Show the values printed by the following program assuming various parameter-passing schemes given below to be used for the parameters to `foo`:

```
int A[2];

void foo(int a, int b)
{
    b--;
    a++;
}

main()
{
    int x = 1;
    A[0] = 3;
    A[1] = 5;
    foo(A[x], x);
    cout << x << A[0] << A[1] << endl;
}
```

**Part A.** call by value

**Part B.** call by reference

**Part C.** call by value-return

**Part D.** call by name

## Problem 2 [16 pts.]

Short answer.

1. Subset construction creates a \_\_\_\_\_ from a \_\_\_\_\_.
2. Thompson's construction creates a \_\_\_\_\_ from a \_\_\_\_\_.
3. Bottom-up parsers do what kind of derivations?
4. Top-down parsers do what kind of derivations?
5. This semester in our Espresso project we wrote semantic rules to do what specific task?
6. The input to flex is \_\_\_\_\_
7. We discussed three abstract machines this semester: PDA, DFA, and Turing machine.
  - a. Of the machines given above: name the least powerful abstract machine that is required to accomplish each of the compiler tasks below:
    - i. lexical analysis \_\_\_\_\_
    - ii. parsing \_\_\_\_\_
    - iii. semantic analysis \_\_\_\_\_
  - b. Which of the abstract machines named is the least powerful needed to accomplish each of the following tasks:
    - i. check that C/C++ `continue` statements are only used within loops.
    - ii. recognize valid U.S. phone numbers
    - iii. check if the parentheses are matched in a string of arbitrarily nested parentheses

### Problem 3 [10 pts.]

Suppose a bison-generated parser is running and is about to reduce by the following production, which has the given associated semantic action:

```
Formals : Formals ',' Formal          { $$ = $1 + 1; }
```

Note: The semantic rule in this example is to compute the number of formals in a list of formal parameters (the other RHS's and semantic actions for `Formals` are not shown here).

**Part A.** Show the current state of the data structures used while running this parser *right before* the reduce and semantic action happens. You won't know the complete contents of the data structures: just indicate what you do know. For items whose value you don't know exactly, just use variables, and state what they represent. Note: you do not have to show the parse tables, since they do not change as we run the parser.

**Part B.** Show the contents of the same data structures *right after* the reduce and semantic action.

## Problem 4

Consider the following grammar-grammar. That is, it is a grammar to specify the syntax of grammars themselves. The grammars being described use the syntax of grammars in bison. To help avoid confusion, we'll use the following terminology for this problem:

- the **BNF grammar** is the grammar below, it has start symbol `gmr`.
- **BNF** is the language being described by the BNF grammar
- a **BNF sentence** is a sentence in BNF. These sentences start with the terminal `nt`.

We indicate all nonterminals in BNF with the terminal symbol `nt`, and all terminals with the terminal symbol `t`. The exact terminal or nonterminal in BNF would be represented as a semantic value of the `t` or `nt`: the lexical analyzer would handle that transformation. The other terminal symbols in the BNF grammar are: `';`, `':'` and `|`

Note: the  $\epsilon$  (epsilon) below is used to denote the empty string in the BNF grammar: it is not a terminal in BNF sentences (i.e., bison grammars): recall that  $\epsilon$  on the right-hand side (RHS) of a production in bison is given by an empty RHS.

```
gmr → prod
    | gmr prod
```

```
prod → nt ':' RHS optRHSList ';'

```

```
RHS → symbol RHS
    | ε

```

```
symbol → t
        | nt

```

```
optRHSList → '|' RHS optRHSList
           | ε

```

**Part A [5 pts].** Here is an example BNF sentence. This BNF sentence is an example of a grammar whose language includes the empty string.

```
nt : t t
    |
    ;

```

Show a derivation of the above string using the BNF grammar. You may either show your derivation as a sequence of steps or a parse tree (you can use the space to the right of the BNF grammar).

## Problem 4 (cont.)

**Part B [6 pts].** Is BNF regular? If so, write a regular expression for the BNF language as defined by the BNF grammar. If not, give an argument why it is not. More information on the form of your answer: If you choose *yes*, you may write it as a regular definition, i.e., using variable names, which you have to define as other regular expressions, to stand in for parts of the regular expression, like we did in flex. If you choose *no*, your answer does not have to be a formal proof: you can give an informal argument in terms of the need for a stack to recognize strings in the language.

Here is the BNF grammar once again:

```
gmr → prod
      | gmr prod
```

```
prod → nt ':' RHS optRHSList ';'
      | nt
```

```
RHS → symbol RHS
      | ε
```

```
symbol → t
          | nt
```

```
optRHSList → '|' RHS optRHSList
             | ε
```

### Problem 4 (cont.)

[20 pts] Create a LL(1) parse table for the BNF language. You will do this by answering the parts given in this and the next page.

Here is the BNF grammar once again:

$$\text{gmr} \rightarrow \text{prod} \\ | \text{gmr prod}$$
$$\text{prod} \rightarrow \text{nt} \text{ ':' } \text{RHS} \text{ optRHSList} \text{ ';'}$$
$$\text{RHS} \rightarrow \text{symbol} \text{ RHS} \\ | \epsilon$$
$$\text{symbol} \rightarrow \text{t} \\ | \text{nt}$$
$$\text{optRHSList} \rightarrow \text{'|'} \text{RHS} \text{ optRHSList} \\ | \epsilon$$

**Part C1.** Do any grammar transformations necessary to make the BNF 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).

**Problem 4 (cont.)**

**Part C2.** Show the first and follow sets for each of the non-terminals in the grammar you ended up with in part C1.

**Part C3.** Create the LL(1) parse table for the final grammar you gave in part C1. 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.

## Problem 4 (cont.)

**Part D [10 pts].** Write semantic rules for the BNF grammar which determine if the BNF sentence being parsed has any epsilon-productions. Recall that in the bison syntax, epsilon-productions are not denoted with an epsilon, but rather, with an empty right-hand-side. The BNF sentence we did a derivation for in Problem 4A was an example of a grammar that *does* have an epsilon production (so `gmr.empty` would be true for that grammar).

To be more specific, use the boolean semantic attribute `gmr.empty`, which is true iff the BNF sentence has *any* productions with an empty RHS. Note: such BNF sentences do not necessarily denote languages that contain the empty string, for example, `gmr.empty` is true for the following BNF sentence for a grammar that denotes the language  $\{ xyz, xz \}$ . In the following, A and OptY are **nt**'s and x, y, and z are **t**'s.

```
A      :  x OptY z ;
OptY   :  | y ; /* The y is optional: the first RHS is empty */
```

---

```
gmr → prod
```

```
      | gmr1 prod
```

```
prod → nt ':' RHS optRHSList ';' 
```

```
RHS → symbol RHS1
```

```
      | ε
```

```
symbol → t
```

```
        | nt
```

```
optRHSList → '|' RHS optRHSList1
```

```
          | ε
```

### Problem 5 [15 pts.]

Consider the following basic block. Assume all of the temporary variables,  $t_1$  through  $t_8$ , *are not* live at the end of the basic block, and all of the other variables *are* live at the end of the basic block. *Label and circle your final answers for each part.*

**Part A.** Show the DAG representation for this basic block.

**Part B.** Using the heuristic ordering discussed in class, which can yield more efficient final code, assign a valid topological ordering to all the interior nodes in the graph for use in code generation. Number the nodes, 1, 2, etc., such that smaller numbered nodes correspond to instructions that will be generated earlier than those with larger numbers.

**Part C.** Generate an improved 3AC sequence from your DAG using the ordering you gave in part B.

```
t1 = x - y;  
t2 = z * 4;  
t3 = t1 * t2;  
t4 = a + m;  
t5 = t3 / t4;  
b = t5;  
t6 = z * 4;  
t7 = a + m;  
t8 = t6 / t7;  
c = t8;
```

## Problem 6 [13 pts.]

**Part A.** 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  $\{e, c, d\}$  is the set of variables live after the last statement.

$live-in(1) =$

(1)  $x := a + b;$

$live-out(1) =$

(2)  $y := x + 10;$

$live-out(2) =$

(3)  $z := a - b;$

$live-out(3) =$

(4)  $m := y / z;$

$live-out(4) =$

(5)  $z := m * 5;$

$live-out(5) = \{z, a, b\}$

**Part B.** Show the register interference graph for this code sequence.

**Part C.** Give a register assignment for the variables using a minimal amount of registers (assume you have as many as you need). Show your answer by annotating your graph from part B with numbers denoting registers (e.g., 1, 2, ...).

### Problem 7 [10 pts.]

Consider the following faulty code generation code:

```
cgen(e1 + e2) =  
    cgen(e1);  
    $t1 := $a0;  
    cgen(e2);  
    $a0 := $t1 + $a0;
```

**Part A.** Explain what the problem is with the code. Give specific situations where the generated code will fail.

**Part B.** Write a correct version of the code. Refer to the additional handout given with this exam for assumptions about the generated code and the assembly language syntax to use.

```
cgen( e1 + e2 ) =
```

## Problem 8 [15 pts.]

Consider the following classes for a C++ program.

```
class X {
    int a;
    int b;
public:
    virtual void f();
    virtual void g();
    . . .
};

class Y : public X {
    int c;
    int d;
public:
    virtual void g();
    virtual void h();
    virtual void j();
    . . .
};

class Z : public X {      // Notice that Z is NOT a subclass of Y
    int e;
public:
    virtual void f();
    virtual void j();
    . . .
};
```

**Part A.** Recall that to do code generation for virtual function binding correctly we need some run-time data structures called method tables. Show the contents of the method tables for classes X, Y, and Z. Assume we are using the following naming scheme:

- o method table for class Foo is located at address denoted by label Foo\_methTab
- o method bar defined in class Foo is located at address denoted by label Foo.bar

**Part B.** Show the layout of X, Y, and Z objects, including the information necessary to do virtual function binding.