

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

USC loginid: \_\_\_\_\_

**CS 410 Final Exam**  
**Fall 2002 [Bono]**  
December 11, 2002

There are 9 problems on the exam, with 95 points total available. There are 10 pages to the exam, including this one; make sure you have all of them. There is also a one-page handout on assumptions for code generation. 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	8 pts.	
Problem 2	10 pts.	
Problem 3	11 pts.	
Problem 4	6 pts.	
Problem 5	12 pts.	
Problem 6	13 pts.	
Problem 7	10 pts.	
Problem 8	10 pts.	
Problem 9	15 pts.	
<b>TOTAL</b>	95 pts.	

## Problem 1 [8 pts.]

A. A top-down parser performs what kind of derivation? \_\_\_\_\_

B. When running a SLR(1) parser, suppose on top of the stack we have a handle for the production

$$A \rightarrow \alpha$$

Give an expression that describes the inputs for which we can reduce by this production.

\_\_\_\_\_

C. The Unix tool `bison` takes \_\_\_\_\_ as input and produces

\_\_\_\_\_ as output.

D. D. An abstract machine to recognize whether a string matches a particular regular expression

is called a \_\_\_\_\_.

**Problem 2 [10 pts.]**

Consider the language that is the set of nested begin-end pairs.

Here are some example strings, and whether they are part of the language or not. Note that the empty string, and sequences of begin-end pairs are not part of the language.

<u>sentence</u>	<u>is in language?</u>
begin end	yes
begin begin end end	yes
begin begin end	no
<epsilon>	no (i.e., empty string is not allowed)
end	no
begin end begin end	no (i.e., no sequences of nested pairs allowed)

**Part A.** Circle the most specific language class this language belongs in:

regular languages

context free languages

**Part B.** If the language is regular, write a regular expression for it, otherwise write a context free grammar for it:

### Problem 3 [11 pts.]

Consider a new behavior of your Cool compiler such that it will generate a warning to the user when he/she declares a variable of the same name as a usable variable that has already been declared in an outer scope. (Sometimes users do this by mistake.) Warnings are just notices to the user: programs with warnings only are still correct programs. Here is an example correct Cool program fragment annotated with line numbers and the warnings that would get generated from it:

```
(1) class Foo is
(2)   i : Int;
(3)   func(i : Int, j : Int) : Int is
(4)     let j : Int in
(5)       i + j
(6)     end
(7)   end;
(8) end;
```

Warning: i defined on line 3 hides i defined on line 2

Warning: j defined on line 4 hides j defined on line 3

**Part A [2].** The code to detect and generate this warning would be part of what phase of your Cool compiler

---

**Part B [2].** What data structure(s) would be involved in detecting this warning condition

---

**Part C [3].** Briefly describe how you would detect the warning condition:

**Part D [2].** We could also solve this problem using semantic attributes (Note: this would make more sense with a different compiler design than ours). If we were to do this, circle the type of semantic attribute that would be involved:

*( inherited / synthesized )*

**Part E [2].** What data would the semantic attribute represent:

---

## Problem 4 [6 pts.]

Show what the output of the following program would be under the two following assumptions.

**Part A.** Assume we're using *static scoping*:

Program output:

**Part B.** Assume we're using *dynamic scoping*:

Program output:

---

```
int a = 0;

void f(int a)
{
    cout << "f: " << a << endl;
    g(15);
}

void g(int c)
{
    cout << "g: " << a << endl;
}

int main()
{
    int b = 1;
    f(b);
    g(b);
}
```

## Problem 5 [12 pts.]

Show the values printed by the following program assuming various parameter-passing schemes given below:

```
int a = 0;

void foo (int x, int y)
{
    x = 3;
    y = a;
}

main ()
{
    int b = 20;
    foo (a, b);
    cout << a << " " << b << endl;
}
```

**Part A.** Call-by-value

**Part B.** Call-by-reference

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

**Part D.** Call-by-name

### Problem 6 [13 pts.]

Consider the following basic block. Assume all of the programmer variables, a through d, are live at the end of the basic block, and the temporary variables, t1 through t6, are not live at the end of the basic block. Label and circle your final answers.

t1 = a \* b;

t2 = c - d;

t3 = t1 + t2;

x = t3;

t4 = a \* b;

t5 = c - d;

t6 = t4 + t5;

b = t6;

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

**Part B.** Generate an improved 3AC sequence from your DAG.

**Problem 7 [10 pts.]**

Consider generating code for a variable reference in Cool. (Note: the AST node type corresponding to this kind of expression is `object_class`.) With the exception of a reference to the special variable `self`, the code that gets generated will have the following form:

```
lw $a0, c(reg)    # load into $a0 the word c bytes away
                  # from the address in reg
```

**Part A.** What will `reg` be? State what its value represents

**Hint:** there are multiple cases to consider: describe each one.

**Part B.** What does `c` represent? Again, describe this in terms of the cases described in part A.

### **Problem 8 [10 pts.]**

Currently in Cool we are allowed to override method definitions in subclasses: that is, in an inheritance hierarchy we can have different method definitions for a given method declaration. Consider a change to the Cool language so that in the new version we are NOT allowed to override method definitions. Thus, for any class, there is only one method definition for any particular method declaration in this class and subclasses.

**Part A.** Describe how you'd implement the code generation for *dispatch* in this new version of Cool.

**Part B.** This change would also have an impact on the runtime data structures. Which of them would change, and how and why?

## Problem 9 [15 pts.]

Write a code-generation routine for a new infix operator **max**. Max returns the maximum of its two integer operands. Here's are a few examples of its use:

```
int maxVal = (3*x+5) max (y/g(x+4));  
cout << (3 max 5);    -- prints 5
```

Refer to the additional handout for details on the stack machine we'll be using and on pseudo-MIPS notation. Furthermore, you may assume:

- all results will be 32-bit integers (NOTE: this is different than operations in Cool)
- we'll show the input of the routine to be the source code itself; you can call `cgen` recursively on parts of the expression to generate code for those parts
- you may use the function `newLabel()` to generate unique labels (returns a `string`).
- output your code by using `<<` statements to `cout`

```
cgen( e1 max e2 ) =
```