

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

USC ID: \_\_\_\_\_

## CS 410 Final Exam Spring 2000 [Bono]

May 4, 2000

There are 11 problems on the exam, with 100 points total available. There are 11 pages to the exam, including this one; make sure you have all of them. 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 ID number at the top of the exam. Please read over the whole test before beginning. Good luck!

	<b>value</b>	<b>score</b>
Problem 1	14 pts	
Problem 2 & 3	9 pts.	
Problem 4	11 pts.	
Problem 5	9 pts.	
Problem 6 & 7	10 pts.	
Problem 8	7 pts.	
Problem 9	15 pts.	
Problem 10	10 pts.	
Problem 11	15 pts.	
<b>TOTAL</b>	100 pts.	

## Problem 1 [14 pts.]

Fill in the blanks to complete the following sentences.

**Part A.** The \_\_\_\_\_ stage of a compiler can be written as a DFA.

**Part B.** Abstract machines for recognizing languages that can be denoted with regular expressions are called \_\_\_\_\_.

**Part C.** A top-down parser that is not table-driven is called \_\_\_\_\_.

**Part D.** Parsers we studied this semester that work by doing a rightmost derivation in reverse are called \_\_\_\_\_.

**Part E.** Location-oriented (a.k.a., position-oriented) generations of boolean expressions is useful for generating code for what C/C++ feature? (hint: something that Cool does not have)  
\_\_\_\_\_.

**Part F.** Access links or displays are necessary if your language has \_\_\_\_\_.

**Part G.** Inherited semantic attributes allow you to pass information down a \_\_\_\_\_.

**Problem 2 [4 pts.]**

Explain why programming languages that do not allow recursion do not need a call stack. Your explanation should include what the data is that would have been on the stack, and where it goes.

**Problem 3 [5 pts.]**

Write a regular expression for strings of digits over the alphabet  $\{1, 2, 3, 4\}$ , such that they appear in non-decreasing order. Your language may include the empty string.

Sample strings in the language:

3

14

111222

1234

## Problem 4 [11 pts.]

**Part A [6].** The application programmer can apply many of the optimizations we discussed directly to the source code. You can do three such optimizations to the following C++ code fragment (i.e., such that the resulting code is also C++). Show the optimized version of the code -- you can mark your changes right in or alongside the code (including crossing out anything that goes away in the new version).

```
while (i < 10) {  
  
    cin >> a;  
  
    cout << b * 10 / a;  
  
    d = 12 * 3 / a;  
  
    d = d * 8;  
  
}
```

**Part B [3].** Write the letters a, b, and c above next to your three optimizations from part A. For each of a, b, and c, give the name of the optimization you performed:

a. \_\_\_\_\_

b. \_\_\_\_\_

c. \_\_\_\_\_

**Part C [1].** Suppose you are an application programmer who just made the changes from part A to your program. Give a scenario where the original program would be just as fast as the changed version. (Hint: we discussed this as a justification for programmers not optimizing their code).

**Part D [1].** Give another justification for programmers not optimizing their code.

## Problem 5 [9 pts.]

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

```
void Swap (int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}

main ()
{
    int i;
    int A[2];

    i = 0;
    A[0] = 1;
    A[1] = 5;
    Swap (i, A[i]);
    cout << i << A[0] << A[1];
}
```

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

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

**Part C.** Call-by-name

### Problem 6 [4 pts.]

Complete the following two sentences using the word "per" in your answer. E.g.  
"3 buildings per year"

**Part A.** The space overhead for doing dynamic dispatch in Cool or C++ is

**Part B.** The time overhead for doing dynamic dispatch in Cool or C++ is

### Problem 7 [6 pts.]

Consider the following inheritance hierarchy in Cool.

```
class Foo is
  f ( ) : Object is . . . end;
end;

class Bar inherits Foo is
  f ( ) : Object is . . . end;
  b ( ) : Object is . . . end;
end;
```

**Part A [3].** Why does the following expression not compile? (Hint: it has no syntax errors)

```
let x : Foo <- new Bar in
  begin
    . . .
    x.b( );  -- line (i)
  end
end
```

**Part B [3].** Replace line (i) in the above let with an equivalent expression which does compile. Write your replacement below.

### Problem 8 [7 pts.]

Construct a (possibly ambiguous) context-free grammar that accepts arithmetic expressions with terminal symbols in the set  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +\}$  such that the language described by the grammar includes only expressions that evaluate to *even* numbers. Hint: define nonterminals *Odd* and *Even*. Think recursively.

A few examples of sentences:

4 + 3                      not in the language

4                              in the language

4 + 4 + 10                in the language

## Problem 9 [15 pts.]

**Part A [12].** Give the FOLLOW sets for each of the non-terminals (A-D and S) in the following grammar. Show your work.

Note: S is the start symbol, the lower-case letters are terminals, and  $\epsilon$  denotes the empty string.

$S \rightarrow A m r$

| p B

$A \rightarrow D B C$

| A S

$B \rightarrow a D x C y$

|  $\epsilon$

$C \rightarrow f g$

| h B

$D \rightarrow b D$

|  $\epsilon$

**Part B [3].** Fill in the blanks in the following sentence. Hint: There is more than one kind of parser that uses follow sets. Answer the question for one of them to receive full credit.

Follow sets tell us when to choose \_\_\_\_\_

in a \_\_\_\_\_ parser.

## Problem 10 [10 pts.]

**Part A [8].** Find the liveness and next-use information for the variables appearing in a statement for each statement in 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.

I have left room underneath each statement for you to attach the information for that statement.

(1)  $t1 = b * 10;$

(2)  $t2 = a - c;$

(3)  $t3 = 3 * t2;$

(4)  $t4 = t1 + t3;$

(5)  $t5 = t1 * t2;$

(6)  $a = t4 - t5;$

**Part B [2].** Liveness information, such as in the above example, is used (circle one):

- a. to help eliminate statements when doing a DAG-based transformation
- a. to help create a flow graph
- b. to help determine when registers may be reused in final code generation
- c. to do peephole optimization

## Problem 11 [15 pts.]

Write a routine, `cgen`, to generate code for a **for** loop with the following syntax:

```
for id = e1 to e2 do e3
```

Semantics of the **for** loop:

$e_1$  is evaluated giving an integer  $i_1$ ;  $e_2$  is evaluated giving an integer  $i_2$ ; and then  $e_3$  is evaluated with  $id = x$  for each  $i_1 \leq x \leq i_2$  in order. If  $i_1 > i_2$  then the **for** terminates without evaluating  $e_3$ . The `id` is a new variable that hides any definitions in outer scopes; the scope of `id` is  $e_3$ . This construct is a statement, thus does not return any value.

Assumptions about generating code:

- you may assume that the iteration variable `id` is stored at offset `idloc` in the frame.
- the code generation scheme will be similar to the one we discussed in lecture for a stack-machine:
  - There is a frame pointer `$fp`
  - temporaries are pushed on the stack, not put in the frame
  - the results of expressions are always left in `$a0`
  - evaluating an expression leaves the stack pointer, `$sp`, with the same value it had before evaluating the expression
  - 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 `char*`).
- output your code by using `<<` statements to `cout`.
- syntax of the assembly language to generate (we'll call it pseudo-MIPS). It has operations like MIPS, but allows you to take operands directly from memory:

```
x := y           For loading, storing, and moving between registers
x := y op z      Where op is one of +, -, *, /, %, "and", "or"
x := op y        Where op is one of -, "not"
goto L           jump to label L
if y relop z goto L      Conditional jump to label L, where relop
                        is one of <, >, =, !=, >=, <=
push y           shorthand for: 0($sp) := y; $sp := $sp-4;
pop              shorthand for: $sp := $sp + 4;
```

in the above statements:

"x", "y", and "z" may be one of:  
a register  
offset(reg)

and "y" and/or "z" may also be constant values

## Problem 11 (cont.)

Part of the previous page reprinted here for your convenience:

Semantics of the **for** loop:

$e_1$  is evaluated giving an integer  $i_1$ ,  $e_2$  is evaluated giving an integer  $i_2$ , and then  $e_3$  is evaluated with  $id = x$  for each  $i_1 \leq x \leq i_2$  in order. If  $i_1 > i_2$  then the **for** terminates without evaluating  $e_3$ . The  $id$  is a new variable that hides any definitions in outer scopes; the scope of  $id$  is  $e_3$ . This construct is a statement, thus does not return any value.

$\text{cgen}(\text{for } id = e_1 \text{ to } e_2 \text{ do } e_3) =$