

Name: _____

USC loginid (e.g., ttrojan): _____

CS 410 Final Exam
Fall 2005 [Bono]
December 7, 2005

There are 12 problems on the exam, with 110 points total available. There are 11 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!

	value	score
Problem 1	12 pts.	
Problems 2-5	10 pts.	
Problem 6	11 pts.	
Problem 7	10 pts.	
Problem 8	10 pts.	
Problem 9	10 pts.	
Problem 10	12 pts.	
Problem 11	10 pts.	
Problem 12	25 pts.	
TOTAL	110 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 x, int y)
{
    x--;
    y--;
    A[1] = 8;
}

main()
{
    int k = 1;
    A[0] = 3;
    A[1] = 6;
    foo(k, A[k]);
    cout << k << 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 [2 pts.]

Suppose the following is the complete set of items for one state in the DFA to recognize viable prefixes for a SLR parser (lower case letters are terminals, upper case letters non-terminals):

[A → C . y B]
[B → z D .]

What would have to be true to get a shift-reduce conflict in this state?

Problem 3 [4 pts.]

In structuring the code for type checking an expression we normally have one function per AST expression node type.

Part A. What information is passed *into* such a type-check routine (i.e., that's not already in the AST node)?

Part B. What information is passed *back* from (or computed by) such a type-check routine?

Problem 4 [2 pts.]

Give an example of a compile-time error that could only be detected during semantic analysis.

Problem 5 [2 pts.]

What common feature of object-oriented languages requires type-specific information to be stored with objects at run-time?

Problem 6 [11 points]

Consider the language of zero or more *comma-separated expressions*, which is the syntax for parameter lists in Espresso and many other languages. Here are four sample sentences in the language (we're treating **expr** as a terminal for the purposes of this problem):

ϵ **expr** **expr , expr** **expr , expr , expr**

where ϵ is the empty string.

Part A [3]. Suppose someone proposes the following grammar for the language:

```
list →  $\epsilon$ 
      | expr
      | list , expr
```

Explain in what way and why the grammar given does not denote the exact language described.

Part B [5]. Write a correct grammar for the language described.

Part C [3]. Write a regular expression for the language described.

Problem 7 [10 pts.]

Find the FIRST and FOLLOW sets for each of the non-terminals in the following grammar.

Note: ϵ denotes, epsilon, the empty string

$S \rightarrow a B S$

| $B C$

$B \rightarrow b C$

| $b B$

| ϵ

$C \rightarrow B x$

| $C y$

| z

Problem 8 [10 pts.]

Consider the following grammar:

$$\begin{aligned} S &\rightarrow a S d \\ &\quad | a B d \\ B &\rightarrow b B c \\ &\quad | b c \end{aligned}$$

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. 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 9 [10 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, t_1 through t_8 , are not 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. Generate an improved 3AC sequence from your DAG.

```
t1 = i * 2;  
t2 = a[t1];  
t3 = i * 2;  
t4 = a[t3];  
t5 = t2 + t4;  
y = t5;  
t6 = y + 10;  
t7 = t6 / m;  
z = t7;
```

Problem 10 [12 pts.]

Part A [4]. 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) $a := b * 10;$

live-out(1) =

(2) $c := a / b;$

live-out(2) =

(3) $d := e + 7;$

live-out(3) =

(2) $e := e + 1;$

live-out(4) = $\{e, c, d\}$

Part B [5]. Show the register interference graph for this code sequence.

Part C [3]. 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 11 [10 pts.]

Write a code-generation routine for a new infix operator **min**. Min returns the minimum of its two integer operands. Here is an example of its use

```
int minVal = (4*y+7) min (x+g);  
cout << (10 min 20); // prints 10
```

Refer to the additional handout given with this exam for assumptions about the generated code and the assembly language syntax to use.

```
cgen( e1 min e2 ) =
```

Problem 12 [25 pts.]

Note: the question and answers for this problem use the assumptions about the generated code and the assembly language syntax given on the additional handout.

Suppose our Espresso compiler uses a calling sequence similar to the one discussed in class, except with some differences described here. This problem concerns writing the code to set up, access, and clean up the activation record under this scheme. Calling convention to use:

- In this scheme, like the one discussed in class, the callee saves and restores the `$fp` and `$ra`, as well as setting up the current value of the `$fp`.
- The `$fp` will point one word beyond the last parameter pushed (i.e., at a lower memory address, since memory grows downwards).
- Keep the “this” pointer in register `$s0`. This will allow for quicker access to fields in the routine. `$s0` will be a callee saved register, just as `$fp` and `$ra` are.

So, under this scheme, the following is the code to access “this”:

```
cgen(this) =      $a0:=$s0
```

- assume Espresso does not have local variables
- the rest of the calling convention is shown below. This is the caller part of the call sequence:

```
cgen ( e0 . f (e1, e2, ..., en) ) =  
cgen(e0)  
push $a0      # push object reference, i.e., callee's "this"  
cgen(e1)  
push $a0      # evaluate and push parameters  
cgen(e2)  
push $a0  
...  
cgen(en)  
push $a0  
jal T0.f      # where T0 is e0's type
```

Part A. Write the `cgen` code for the prologue part of a function definition, assuming the caller code and the calling conventions described above. The prologue is the “callee” part of the call sequence. You will not be showing *all* the `cgen` code for the function definition, just the first part that comprises the prologue.

```
cgen (A.f(x1, x2, ..., xn) { ... }) =
```

Problem 12 (cont.)

Part B. Show the activation record layout. I.e., what's in it right after your prologue from part A executes. Also show the current values of $\$sp$ and $\$fp$ at that point. In your drawing show larger addresses at the top (as we did in all our stack diagrams in class).

Part C. Give the `cgen` code to access the i th, formal parameter of n parameters x_1 through x_n :

`cgen (xi) =`

Part D. Show the `cgen` code for the epilogue (i.e., return sequence) for a function definition, such that the whole return sequence is handled by the callee (i.e, the `cgen` code for the call, given on the previous page, is complete, ending with the “jal” instruction). The epilogue is only part of `cgen` for a function def (another part of which we did in Part B). You can use the MIPS instruction “jr reg” to return: it jumps to the address stored in reg.

```
cgen (A.f(x1, x2, ..., xn) { ... }) =  
  <prologue would go here...>  
  <cgen for function body and return expr would go here...>  
  # add code for epilogue below this point
```

Part E. (*fill in the blanks with numbers*) When we do *not* store “this” in a register it takes _____ MIPS instructions to access a field in Espresso. Instead, using $\$s0$, it takes _____ MIPS instructions to access a field in Espresso.