

Name: _____

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

CS 410 Final Exam
Spring 2005 [Bono]
May 4, 2005

There are 8 problems on the exam, with 120 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!

	value	score
Problem 1	15 pts.	
Problem 2	13 pts.	
Problem 3	11 pts.	
Problem 4	14 pts.	
Problem 5	20 pts.	
Problem 6a	2 pts.	
Problem 6bc	20 pts.	
Problem 6d	3 pts.	
Problem 7	12 pts.	
Problem 8	10 pts.	
TOTAL	120 pts.	

Problem 1 [15 pts.]

Short-answer problems. 2 pts each, except where noted.

Part A. Briefly describe the relationship between DFAs and flex.

Part B. Briefly describe the relationship between DFAs and lexical analyzers.

Part C. Briefly describe the relationship between regular expressions and flex.

Part D. Briefly describe the relationship between regular expressions and lexical analyzers.

Part E [5 pts]. Write a **grammar** for the language denoted by the following regular expression:

$a^* b^+ a^*$

Part F. Using your answer to the last problem, show a derivation of the following string:

bba

Problem 2 [13 pts.]

More short-answer problems. For parts A and B you may want to refer to the extra handout for further information on pseudo-MIPS and on the `NameGenerator`.

Part A [5]. For Espresso code generation, suppose `this` is stored at offset `offthis` from `$fp`. Show the generated code for a `VarRef` on field `f`. Your answer will be a sequence of one or more pseudo-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 [2]. Consider using fixed labels (for locations in static memory) for **local** variable locations in Espresso. So, for example, you might assign the label “L32” (generated by the `NameGenerator`) to the local variable `x` in method `y`. Give a scenario where the code we generate this way will fail (i.e., will produce different results than would be defined by the Espresso language semantics).

Part C [2]. Name or describe two optimizations that can result from the DAG transformation of a basic block.

Part D [4]. Give an example of a **loop optimization** that is performed by compilers. Do this by showing before and after source code (C++) where you have performed the same optimization by hand.

Code before opt:

Code after opt:

Problem 3 [11 pts.]

Suppose C allowed nested functions. Consider the following program in the modified language:

```
int main(...)
{
    . . .
    void d(...)
    { . . . /* body of d */
    }
    void f(...)
    {
        void g(...)
        { . . . /* body of g */
        }
        void h(...)
        { . . . /* body of h */
        }
        . . . /* body of d */
    }
    . . . /* body of main */
}
```

Part A. Given the following call stack (shown growing downwards), where a stack frame is shown by the name of the function it's for, show the values of the access links at this point in the program execution:

main
d
f
g
h
d

Part B. Now assume we are using a display instead of access links. For the same point in the run, show the current value of the display. Also show the saved value(s) in stack frames necessary to restore the display once we return from each call.

main
d
f
g
h
d

Problem 4 [14 pts.]

Consider the following classes for a C++ program.

```
class Foo {
public:
    virtual void meth1();
    virtual void meth2();
    . . .
};

class Bar : public Foo {
public:
    virtual void meth1();
    virtual void meth3();
    . . .
};

class Blob : public Bar {
public:
    virtual void meth2();
    virtual void meth4();
    . . .
};
```

Part A [10]. 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 `Foo`, `Bar`, and `Blob`. Assume we are using the following naming scheme:

- method table for class `X` is located at address denoted by label `X_methTab`
- method `y` defined in class `X` is located at address denoted by label `X.y`

Part B [1]. For the following method call, circle the exact entry in the exact table that is used when we run the generated code. That is, circle the correct entry from your answer to part A.

```
Bar *a = new Blob();
a->meth3();
```

Part C [3]. If we change the above code to the code below (the part that changed is shown in italics), it will no longer compile. What is the error and what part of the compiler would catch the error?

```
Foo *a = new Blob();
a->meth3();
```

Problem 5 [20 pts.]

Consider the following unambiguous grammar for reverse polish notation (postfix) arithmetic expressions:

$$\begin{aligned} E &\rightarrow E E + \\ &| E E - \\ &| \text{id} \end{aligned}$$

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 6 [25 pts.]

This problem concerns implementing the `for` construct using a `while`.

Part A [2]. Show C++ code equivalent to the following `for` loop, but using a `while` loop instead.

```
for (i=0; i<n; i++) {  
    cout << i;  
}
```

Part B [10]. Consider adding Java's `for` construct to Espresso, but implementing it in terms of `while`, as you did above. Given the following grammar for `for`, draw a diagram of the AST that would be constructed.

```
stmt  $\rightarrow$  for (stmt1; expr; stmt2) stmt3
```

Notes:

- The subscripts in the grammar above are to distinguish different instances of the same non-terminal.
- This is not the exact Java syntax, it's changed to simplify things here.
- Refer to the additional handout, which summarizes some of the AST constructors we used this semester.
- The type of the semantic value of the non-terminal, `stmt` is `Stmt*`, and for `expr`, it's `Expr*`.

Here's an example of what we have in mind, showing the same thing for a `+` expression. Here, and in your answer, leaves can stand in for semantic values of the non-terminals on the RHS). Also, you can show list ASTs as an arrays of pointers.



(You'll want to refer to part B to do part C; so I recommend putting both answers on the next page. We've left ample space for you there.)

Problem 6 (cont.)

Part B (cont) [10]. (Question repeated here.) Consider adding Java's `for` construct to Espresso, but implementing it in terms of `while`, as you did above. Given the following grammar for `for`, draw a diagram of the AST that would be constructed.

```
stmt → for (stmt1; expr; stmt2) stmt3
```

Part C [10]. Here's the same grammar for `for`, but now shown in bison syntax. Write the semantic action to build the tree you drew above.

Reminder of bison syntax by example (production and action for +):

```
expr : expr '+' expr    { $$ = new Plus($1, $3); }
```

```
stmt : FOR '(' stmt ';' expr ';' stmt ')' stmt
```

Problem 6 (cont.)

Part D [3]. Briefly describe changes to any *other* parts of your Espresso compiler necessary to add `for` implemented using `while` (you already wrote the changes to the parser).

Your answer should be a few sentences long at most. Just because of page formatting, we happened to end up with way too much space here. Please don't write a novel.

Problem 7 [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 $\{a, b\}$ is the set of variables live after the last statement.

live-in(1) =

(1) $n := x * 7;$

live-out(1) =

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

live-out(2) =

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

live-out(3) =

(2) $a := n + z;$

live-out(4) = $\{a, b\}$

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 8 [10 pts.]

Suppose we add the C/Java *conditional expression* to Espresso. Here is an example of its use to compute the maximum of two numbers:

```
int max = (i > j)? i : j;
```

The general syntax and run-time semantics are:

```
expr1 ? expr2 : expr3
```

expr₁ is a Boolean expression. If it is true, the value of the conditional expression as a whole is the value of expr₂ otherwise it is the value of expr₃. Any one time evaluating the conditional expression only one of expr₂ and expr₃ get evaluated.

Write a C++ member function `genCode`, below, to generate pseudo-MIPS code for the conditional expression, assuming the AST structure given below.

```
class Cond : public Expr {
    Expr *cond;           // condition
    Expr *trueExpr;
    Expr *falseExpr;
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.
- See the additional handout given with this exam for assumptions about the generated code and the assembly language syntax to use.

(Space for your answer provided on the next page.)

Problem 10 (cont.)

Here is the AST structure for this construct shown again:

```
class Cond : public Expr {
    Expr *cond;           // condition
    Expr *trueExpr;
    Expr *falseExpr;
public:
    virtual void genCode();
    . . .
}
```

```
void Cond::genCode()
{
```