

Name: _____

USC loginid: _____

CS 410 Final Exam
Spring 2004 [Bono]
May 5, 2004

There are 12 problems on the exam, with 100 points total available. There are 12 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 loginid at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Probs 1 - 3	9 pts.	
Problem 4	10 pts.	
Problem 5	8 pts.	
Problem 6	15 pts.	
Problem 7	8 pts.	
Problem 8	9 pts.	
Problem 9	8 pts.	
Problem 10	8 pts.	
Problem 11	15 pts.	
Problem 12	10 pts.	
TOTAL	100 pts.	

Problem 1 [5 pts.]

For each of the run-time values listed below, put a letter next to it corresponding to where in memory it goes using the following key:

H = heap
S = stack
M = static memory

- a. local variables
- b. global variables
- c. dispatch tables
- d. attributes of self
- e. saved register values

Problem 2 [2 pts.]

Apply the *common subexpression elimination* optimization to the following 3AC:

```
t1 := 3 * y;  
t2 := t1 * m;  
t3 := 3 * y;  
x := t2 + t3;
```

Problem 3 [2 pts.]

Apply the *code motion* optimization to the following C code:

```
while (x < 10) {  
    j = m * 5 - x;  
    cout << j;  
    x++;  
}
```

Problem 4 [10 pts.]

Use subset construction to build a DFA equivalent to the NFA below. Show your work. Note: ϵ is the epsilon symbol.

	a	b	ϵ
1	{ }	{ 3 }	{ 2 }
2	{ 2 }	{ }	{ 4 }
3	{ 3 }	{ 4 }	{ }
<u>4</u>	{ 3 }	{ }	{ }

Problem 5 [8 pts.]

Consider the following ambiguous expression grammar over the alphabet $\{@, \#, \text{id}\}$:

$$\begin{aligned} E &\rightarrow E @ E \\ &| E \# E \\ &| \text{id} \end{aligned}$$

Suppose the @ and # operators have the following properties:

- @ is left associative and has higher precedence
- # is right associative and has lower precedence

For each of the four expressions below, show the correct parse tree such that the operator precedence and associativity rules are obeyed.

Expr1: $\text{id} @ \text{id} @ \text{id}$

Expr2: $\text{id} \# \text{id} \# \text{id}$

Expr3: $\text{id} \# \text{id} @ \text{id}$

Expr4: $\text{id} @ \text{id} \# \text{id}$

Problem 6 [15 pts.]

Consider the following grammar over the alphabet $\{@, \#, \mathbf{id}\}$:

$$\begin{aligned} E &\rightarrow E @ E \\ &| E \# E \\ &| \mathbf{id} \end{aligned}$$

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. Attempt to 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 7 [8 pts.]

Use the following type-checking rule for Cool case expressions to answer the questions below. Note: the \sqcup symbol denotes the *least upper bound*.

$$\frac{\begin{array}{l} O, M \vdash e_0 : T_0 \\ O[T_1/x_1], M \vdash e_1 : T_1' \\ \vdots \\ O[T_n/x_n], M \vdash e_n : T_n' \end{array}}{O, M \vdash \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \dots x_n : T_n \Rightarrow e_n; \text{ esac} : \sqcup_{1 \leq i \leq n} T_i'}$$

Part A [2]. What is the scope of x_i for any i , in $[1..n]$?

Part B [2]. Are there any restrictions on the type of e_0 ? If so, what are they?

Part C [2]. Are there any restrictions on the relationship between T_i and T_i' ? If so, what are they?

Part D [2]. What is the overall type of the following case expression? Assume the following inheritance relationships among user-defined classes: B inherits A; C inherits A. Also assume the compile-time type of v is B.

```
case v of
  x : A => 500;
  y : B => "hello";
  z : C => new C;
esac;
```

Problem 8 [9 pts.]

Consider the following Cool class hierarchy.

```
class A is
  . . .
  mA1() : Int is ... end;
  mA2() : Int is ... end;
end;

class B inherits A is
  . . .
  mA1() : Int is ... end;
  mB1() : Int is ... end;
end;

class C inherits B is
  . . .
  mC1() : Int is ... end;
end;
```

Assuming the classes above are defined, consider the following dispatch expression on method `mA1`:

```
let x : A <- new C in
  x.mA1()
end
```

Part A [2]. At compile-time, to generate code for the dispatch, the compiler finds the offset for method `mA1` in the symbol table for what class?

Part B [3]. At run-time, where does the code for the dispatch get the information about which dispatch table to use?

Part C [2]. At run-time it uses the dispatch table for which class?

Part D [2]. At run-time it calls the method `mA1` defined in which class?

Problem 9 [8 pts.]

This question concerns scoping and accessing of non-local variables in languages with nested functions. Consider the following program in a language that looks like C, but that also allows nested functions, using the most-closely-nested rule:

```
int main() {
    int f() {
        int x; // var inside f
        int h() { // func inside f
            int y;
            . . .
            ... x ... (**)
            . . .
        }
        int i() {
            . . . <----- Is it legal to call h from here? _____
        }
        //body of f . . . <----- Is it legal to access y from here? _____
    }
    int g() {
        . . . <----- Is it legal to call h from here? _____
    }
    // body of main . . .
}
```

Part A [3]. Answer the questions in bold shown in with the code above.

Part B [5]. Consider generating code for the reference to `x` labeled `(**)` in the code above. Can we get the value of `x` via a MIPS expression such as `const($fp)` (where `$fp` is the frame pointer register)?

If so, elaborate on what `const` represents and/or how it is computed; if not, explain why not.

Problem 10 [8 pts.]

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

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

main ()
{
    int A[2];
    int x = 0;

    A[0] = 5;
    A[1] = 8;
    foo (x, A[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 11 [15 pts.]

Part A [5]. 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 $\{a, i\}$ are live after the last statement.

live-in(1) =

(1) $m := i + b;$

live-out(1) =

(2) $a := 3 * m;$

live-out(2) =

(3) $i := i + 1;$

live-out(3) = $\{a, i\}$

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

Part C [3]. Give an optimal register assignment for the variables using the graph given in part B (you may show it by labeling the vertices on the graph above).

Part D [2]. Doing a register assignment on such a graph is an example of an NP-complete problem called the

_____ problem.

Problem 12 [10 pts.]

Write a C++ member function `genCode`, to generate 3-address code for a new Cool expression, **repeat-until**, which repeats some action until some condition is met. Here is its syntax:

```
expr → repeat
      expr1
      until expr2
```

The repeat-until loop always evaluates the body of the loop (i.e., `expr1`) at least once, and then evaluates the condition (i.e., `expr2`). If the test is true, the loop exits *using the value of the loop body on the last iteration as the value of the loop*, otherwise the loop is repeated. Hint: repeat-until is an Expression, not a Statement: its value is described in italics above.

Note: While this construct is Cool-like in that it is an expression and contains only expressions, we will use the usual assumptions that (1) the values of expressions are always integers; and (2) where such values are used in tests we interpret non-zero as true, and zero as false. (In this way it differs from real Cool, where values are object pointers).

Other assumptions about generating code:

- `newTemp()` generates unique temporary names (returns a string).
- `newLabel()` generates unique labels (returns a string).
- There is a polymorphic `genCode()` function defined for all AST node types.
- All Expression node types have a string field called `place` for storing the name that will hold its value. `genCode` on the expression initializes this field. `place` can be accessed via the member function `getPlace()`.
- you may use the function `emit`, which takes a variable number of arguments, to generate 3AC statements. Here is an example of its use, where the parts in double-quotes are literal, and `myTemp` is a string variable:

```
emit(myTemp, ":", myTemp, "+27;");
```

- Syntax of the 3AC for the generated code:

```
x := y          Copy y's value into x
```

```
x := y op z     Where op is one of +, -, *, /, %, or, and
```

```
x := op y       Where op is one of -, not
```

```
goto L         Jump to label L
```

```
x := y[i]      Set x to the value in the mem. loc. i bytes beyond y
```

```
x[i] := y      Give the mem loc i bytes beyond x the value in y
```

```
if y relop z goto L    Conditional jump to label L, where relop is one of <, >, =,
                       !=, >=, <=
```

where the operands can be names or integer constants (except those used as arrays), and the result must be a name. For operands of logical operations, assume non-zero is true, and zero is false.

(Don't put your answer here. Problem continued next page.)

Problem 12 (cont.)

Here is the AST structure for this construct:

```
class Repeat : Expression {
protected:
    Expression *body;
    Expression *cond;
public:
    virtual void genCode(); // you write this function
    . . .
};
```

```
void Repeat::genCode()
{
```