

Name: _____

USC loginid: _____

CS 410 Final Exam
Spring 2003 [Bono]
May 7, 2003

There are 9 problems on the exam, with 100 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!

	value	score
Problem 1	14 pts.	
Problem 2	10 pts.	
Problem 3	8 pts.	
Problem 4	6 pts.	
Problem 5	10 pts.	
Problem 6	15 pts.	
Problem 7	16 pts.	
Problem 8	11 pts.	
Problem 9	10 pts.	
TOTAL	100 pts.	

Problem 1 [14 pts.]

Part A [2]. Apply the *constant folding* optimization to the following code (Note: the starting code as well as the result are/may be given in C, not 3AC):

```
x = 3*5*y + 5*y;
```

Part B [1]. Name a different optimization we could we apply to the original code shown in part A.

Part C [1]. Are the optimizations in parts A and B considered *local* optimizations or *global* optimizations? _____

Part D [1]. Circle the choice that best describes the language defined by the following grammar

$A \rightarrow B y B$
 $B \rightarrow x \mid x B$

- a. $x^+ y x^+$
- b. $x^* y x^*$
- c. an even number of x 's
- d. zero or more x 's followed by a y , followed by the same number of x 's again

Part E [2]. Name the type of parameter passing mode whose semantics are closest to macro expansion:

Part F [2]. Name a type of parameter passing mode that is available in C++ but not in Cool.

Part G [1]. What part of memory are the Cool **dispatch tables** stored in (circle the correct response):

code / heap / stack / static memory

Part H [4]. What part of memory are the Cool objects stored in? Hint: they are in more than one section. Explain which objects are where. code / heap / stack / static memory

Problem 2 [10 pts.]

Part A [8]. Use subset construction to build a DFA equivalent to the NFA below. Show your work. Note: ϵ denotes epsilon.

	a	b	ϵ
1	{ }	{ }	{ 2, 6 }
2	{ 3 }	{ }	{ }
3	{ }	{ 4 }	{ }
4	{ 5 }	{ }	{ }
5	{ }	{ }	{ 2, 6 }
6	{ }	{ }	{ }

Part B [2]. Give a regular expression for the language accepted by the NFA above (and your DFA).

Problem 3 [8 pts.]

The following grammar suffers from the dangling-else problem.

Part A [6]. If we attempt to build an SLR parser with the grammar, we'll get a shift-reduce error. Show exactly where and how this error occurs. I.e., show the relevant sets of items in the DFA, and the details and rationale for the shift and for the reduce. You do not have to build the whole DFA or parse tables to answer this question.

Part B [2]. If we want every **else** to match the closest **if** (normal C/C++, etc. interpretation), should we choose *shift* or *reduce* at the point of the conflict? _____

- 1) $S \rightarrow \text{if (expr) } S$
- 2) $S \rightarrow \text{if (expr) } S \text{ else } S$
- 3) $S \rightarrow \text{other}$

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 b = 100;

void f()
{
    b = b + 50;
    cout << "f: " << b << endl;
}

void g()
{
    int b = 5;
    f();
}

int main()
{
    f();
    g();
    cout << "main: " << b << endl;
}
```

Problem 5 [10 pts.]

Below is a grammar for a subset of cool expressions. Write semantic rules to compute the total number of variables defined in the expression: call the attribute **E.num**. Note: You do not have to be concerned with whether variable names are unique or not or which ones are in scope at the same time: you are just counting the number of variable definitions. You may not use any global variables in your answer. Here is an example cool expression with the result of parsing with the semantic rules (Note that the actual input string is shown, not the sequence of tokens generated):

```
let a : Int <- 0 in
  a + 3 + y.foo( ) +
    let b : Int <- 12 in
      case a + b of
        x : Int => x;
        obj : Object => 0;
      esac
    end -- inner let
end -- outer let
```

For the whole expression: **E.num** = 4
(vars defined in the example are
a, b, x, and obj)

Note: in the grammar below punctuation and bold words are terminals. Uppercase letters are nonterminals (E = expression; L = branch-list; B = branch). Subscripts are just to differentiate different instances of the same nonterminal in one production. We started the first semantic rule for you below.

$E \rightarrow E_1 . \mathbf{id} () \quad \{ E.num =$

| **let id : type** <- E₁ **in** E₂ **end**

| E₁ + E₂

| **id**

| **intConst**

| **case** E₁ **of** L **esac**

L -> B L₁

| B

B -> **id : type** => E ;

Problem 6 [15 pts.]

Consider the following basic block. Assume all of the programmer variables, w through z , are live at the end of the basic block, and the temporary variables, t_1 through t_6 , are not live at the end of the basic block. Label and circle your final answers.

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

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

$t_1 = 3 * x;$

$t_2 = 10 - y;$

$t_3 = t_2 / z;$

$t_4 = t_1 + t_3;$

$x = t_4;$

$t_5 = 10 - y;$

$t_6 = 3 * x;$

$t_7 = t_5 * t_6;$

$w = t_7;$

Problem 7 [16 pts.]

Consider the following classes from a Cool program:

```
class A is
  x : Int;
  f() : Int is
    x
  end;
end;

class B inherits A
  is
  z : Int;
  g () : Int is
    x      -- (1)
  end;
end;

class C inherits B
  is
  f() : Int is
    33
  end;
end;
```

In your answer to the following problems assume that classes are not implicitly derived from Object (so in this example, A has no superclass). Assume dispatch tables for A, B and C are at addresses A_dispTab, and B_dispTab, and C_dispTab respectively.

Part A [6]. Show the **layout for objects** of type A, B and C and label them as such. Also include the header *contents* for each object. Hint: the first word of the header is the class tag; you can just write in classtag where this value would go.

Part B [6]. Show the *contents* of the **dispatch tables** for types A, B and C.

Part C [4]. Suppose we call function *g* on an object whose runtime type is C. Using pseudo-MIPS and the other assumptions given in the additional handout, show the code generated for the expression labeled **(1)** in method *g* (in class definition B above). If necessary, state any additional assumptions you make about what values are where.

Problem 8 [11 pts.]

This question concerns the following type-checking rule for Cool let-with-init expressions:

$$\frac{\begin{array}{l} O, M \vdash e_1 : T_1 \\ T_1 \leq T_0 \\ O[T_0/x], M \vdash e_2 : T_2 \end{array}}{O, M \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 \text{ end} : T_2}$$

Part A [2]. Give an **English** description of any restrictions on the type of e_1 . Or say “none” if there are none.

Part B [2]. Give an **English** description of any restrictions on the type of e_2 . Or say “none” if there are none.

Part C [2]. Give an **English** description of the inferred type for the whole let expression (i.e., the one denoted by T_2 above).

Part D [2]. The symbol O , in type rules, including the rule above, denotes a function that maps _____ to _____.

Part E [3]. The symbol M , in type rules, including the rule above, denotes a function that maps _____ and _____ (i.e., a pair) to _____.

Problem 9 [10 pts.]

Write a code-generation routine for a new infix operator `&&`. `&&` is a short-circuit **and**, such as C/C++ has. It's like a logical `and`, but doesn't evaluate its second argument if the first one is false. This is useful for situations like the one given in this example:

```
// search returns loc of target in arr, or ARRAY_MAX if target  
// does not exist in arr  
int search (int arr[], int target) {  
    int loc = 0;  
    // doesn't try to access arr[loc] when loc is off the end of array  
    while ((loc < ARRAY_MAX) && (arr[loc] != target)) {  
        loc++;  
    }  
    return loc;  
}
```

Like in C `&&` will always return a 1 (true) or 0 (false) value. Also, like in C, its arguments may be integer or boolean-valued expressions, such that we interpret non-zero values to be true, and zero to be false.

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 && e2) =`