

Name: _____

SSN: _____

CS 410 Final Exam
Fall 2001 [Bono]
December 11, 2001

There are 10 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 ID number at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Problem 1&2	16 pts.	
Problem 3	9 pts.	
Problem 4	12 pts.	
Problem 5	6 pts.	
Problem 6	10 pts.	
Problem 7	10 pts.	
Problem 8	14 pts.	
Problem 9	8 pts.	
Problem 10	15 pts.	
TOTAL	100 pts.	

Problem 1 [4 pts.]

Consider the process of constructing a DAG from the 3AC for a basic block, and then subsequently generating new 3AC from the resulting DAG. Describe (or name) two types of code optimizations that result from the transformation described.

1. _____
2. _____

Problem 2 [12 pts.]

Circle all that are true about the compiler tool `bison`:

1. Generates a parser whose actions correspond to a leftmost derivation
2. Generates a table-driven parser
3. Generates a recursive-descent parser
4. Generates a predictive parser (i.e., LL(1))
5. Takes a grammar as input
6. Generates a LALR(1) parser
7. Generates a bottom-up parser
8. Applies Thompson's construction
9. Takes regular expressions as input
10. Generates a top-down parser
11. Takes a source code program as input (e.g., written in Cool)
12. Generates a parser whose actions correspond to a rightmost derivation in reverse

Problem 3 [9 pts.]

The frame pointer is a register that points at the current stack frame.

Part A. What is the frame pointer used for?

Part B. What program operation could overwrite the frame pointer?

Part C. How can we make sure the value doesn't get lost when doing the operation mentioned in part B?

Problem 4 [12 pts.]

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

```
int A[2];
int k;

void foo (int x, int y, int z)
{
    z = 0;
    y = 1;
    A[k] = 7;
    x++;
}

main ()
{
    A[0] = 2;
    A[1] = 4;
    k = 1;
    foo (A[1], A[k], k);
    cout << A[0] << " " << A[1] << " " << k;
}
```

Part A. Call-by-value

Part B. Call-by-reference

Part C. Call-by-value-return

Part D. Call-by-name

Problem 5 [6 pts.]

Show what the output of the following program would be under the two following assumptions. Besides the scoping rules, assume normal C++ program semantics. Reminder: the x defined in the if in function f goes out of scope when we exit the block it is declared in (i.e., body of the if).

Part A. Assume we're using *static scoping*:

Program output:

Part B. Assume we're using *dynamic scoping*:

Program output:

```
int x = 5;
void f(int y)
{
    cout << x << " ";
    if (x < 100) {
        int x = 3000;
        g(y);
    }
    x = x-y;
}

void g(int x)
{
    x--;
    cout << x << " ";
}

int main()
{
    int x = 2;
    f(x);
    cout << x << " ";
}
```

Problem 6 [10 pts.]

Consider the following Cool-like classes (function bodies elided):

```
class A {
    a: Int;
    g() : Int { . . . }
    h() : Int { . . . }
};

class B inherits A {
    b: Int;
    j() : Int { . . . }
};

class C inherits A {
    c : Int;
    g() : Int { . . . }
    k() : Int { . . . }
};
```

Consider the following dispatch expression that uses this inheritance hierarchy:

```
let x : A <- new D in
  x.g();
end
```

Part A [3]. At *compile* time, to generate code for the dispatch, we find the offset for method g in the method-offset table for what class?

Part B [4]. At runtime, *where* does the code for the dispatch get the information about which dispatch table to use?

Part C [3]. At runtime we use the dispatch table for which class?

Problem 7 [10 pts.]

If we attempt to build an SLR parser with the grammar below for sets of matched parentheses, 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.

- 1) $S \rightarrow S S$
- 2) $S \rightarrow (S)$
- 3) $S \rightarrow ()$

Problem 8 [14 pts.]

Part A [10]. 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 t3, 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) $t1 = b * c;$

(2) $t2 = a / 7;$

(3) $t3 = t1 * t2;$

(4) $a = t3 / d;$

(5) $b = b + 1;$

Part B [4]. For each of the following variables, give its liveness and next use information upon *entering* the basic block above.

a

b

c

d

Problem 9 [8 pts.]

For the grammar below, write semantic rules for computing the semantic attribute $S.len$, which is the number of tokens in the sentence generated from S .

A few notes about the grammar:

- terminals are lower case; nonterminals are upper case; other identifiers are nonterminals; ϵ denotes the empty string
- subscripts are just used to distinguish different instances of a nonterminal

$S \rightarrow a S_1 a$

$S \rightarrow B$

$B \rightarrow \epsilon$

$B \rightarrow c B_1$

Problem 10 [15 pts.]

Write a routine, `cgen`, to generate code for a different version of the Cool **while** expression. This version of while looks syntactically like the old version, except *it will return the number of iterations that were executed*. Here is the reminder of the syntax:

```
while e1 loop e2 pool
```

New while loop semantics: e_1 is evaluated before each iteration of the loop. if e_1 is false, the loop terminates and the number of iterations is returned. If it is true, the body of the loop is evaluated and the process repeats. Thus, if we never enter the loop the value of the while is zero, if we go through the loop only one time before exiting, the value is one, etc.

Assumptions about generating code (room for your answer appears on the following page):

- 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
 - 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
 - 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
- 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 := $sp + 4;
top              shorthand for: 4($sp)
```

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 10 (cont.)

Hint: make sure it obeys the stack-machine convention as described on the previous page.

`cgen(while e1 loop e2 pool) =`