

Name: _____

SSN: _____

CS 410 Final Exam
Spring 2002 [Bono]
April 30, 2002

There are 11 problems on the exam, with 105 points total available. There are 12 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 ID number at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Problem 1&2	15 pts.	
Problem 3	20 pts.	
Problem 4	6 pts.	
Problem 5&6	8 pts.	
Problem 7 & 8	14 pts.	
Problem 9	12 pts.	
Problem 10	14 pts.	
Problem 11	16 pts.	
TOTAL	105 pts.	

Problem 1 [5 pts.]

For each of the following language rules (numbered 1-5) for the programming language C, name the least powerful machine that can be used to check it. Give each answer as one of a, b, or c below:

- a. FSA
- b. PDA (the book calls this a stack automaton)
- c. Turing machine (this is equivalent to a program written in a general purpose language)

Language rules:

1. matching { } pairs
2. matching types of actual parameters to formal parameters
3. checking that string constants are properly terminated
4. checking which variable declaration applies for a particular variable reference
5. checking that identifiers contain no illegal characters

Problem 2 [10 pts.].

Circle all that are true about a bottom-up parser:

- (a) its actions correspond to a leftmost derivation
- (b) its actions correspond to a rightmost derivation in reverse
- (c) it is usually table-driven
- (d) it is usually recursive-descent
- (e) it is also called a predictive parser
- (f) it may be a LALR(1) parser
- (g) it may be a LL(1) parser
- (h) it Applies Thompson's construction
- (i) it is usually generated by another program
- (j) it usually takes a grammar as input

Problem 3 [20 pts.]

Create the SLR parse tables for the following grammar for postfix expressions. Show your work. In particular, you must show (1) the augmented grammar, (2) the DFA for recognizing viable prefixes, including the set of items that each state corresponds to, and (3) the action and goto tables

$$(1) \quad E \rightarrow E E +$$

$$(2) \quad E \rightarrow E E -$$

$$(3) \quad E \rightarrow \mathbf{id}$$

Problem 4 [6 pts.]

Part A. Construct a short, but complete, C++ program that includes one call to `foo` such that using pass *by reference* with formal parameter `x` produces different output than using pass *by value-return* with `x` (i.e., for the same call in the same program). Write your answer on the lower part of this page, starting with the code given there.

Part B. Show the output of the program you gave for Part A assuming each of the two parameter-passing techniques:

- a. Output using call by reference:

 - b. Output using call by value-return:
-

```
void foo (int x
```

Problem 5 [2 pts.]

Circle the correct word or phrase from each pair or triple to complete the sentence:

1. Most modern general-purpose programming languages use [*static* / *dynamic*] scoping
2. A DAG can be used to represent [*basic block* / *flow graph* / *both*].

Problem 6 [6 pts.]

Fill in the blanks.

1. What programming feature necessitates *access links* or *a display*?

2. Dynamic storage allocation is necessary to support what two common programming language features?
 - a. _____
 - b. _____
3. State the status (live vs. dead) of each of the variables shown below *right before* we execute the following 3AC statement:

x := y + z ← what's true here

x _____

y _____

z _____

Problem 7 [5 pts.]

Assume we have the following C-style array declaration,

```
xType foo[10][20];
```

and furthermore suppose that the size of an **xType** is **XSize** bytes, and that two-dimensional arrays are stored in *column-major* order.

Give an expression for the correct byte offset to access **foo[i][j]** (note: this doesn't have to be written using 3AC, just express it mathematically):

Problem 8 [9 pts.]

For each of the following data structures, say whether it exists at compile-time (C) or run-time (R):

- (a) call stack
- (b) grammar stack
- (c) semantic stack
- (d) symbol table
- (e) dispatch table
- (f) abstract syntax tree
- (g) activation record
- (h) heap
- (i) display

Problem 9 [12 pts.]

Consider the following for loop:

```
for (int i = 0; i < n; i += 2) {  
    A[i] = B[i] + C[i];  
}
```

Suppose we generated the following unoptimized 3AC (three-address code) for the loop:

```
    i := 0;  
loop:  
    if (i >= n) goto after;  
    t1 := i * 4;  
    t2 := b[t1];  
    t3 := i * 4;  
    t4 := c[t3];  
    t5 := t2 + t4;  
    t6 := i * 4;  
    a[t6] := t5;  
    i := i + 2;  
    goto loop;  
after:
```

This problem concerns optimizing this code. Note: the optimizations you apply will be "by hand", rather than using compiler algorithms (e.g., no DAG creation necessary).

Part A. If possible, apply *common-subexpression elimination* to the 3AC above, or explain why we can't apply that optimization.

Part B. If possible, apply the *loop version of reduction in strength* to the most recent version of the 3AC (i.e., the code for the previous part if you gave some, otherwise the original code), or explain why we can't apply that optimization.

Part C. If possible, apply *code motion* to the most recent version of the 3AC, or explain why we can't apply that optimization.

Part D. If possible, apply *loop unrolling* to the most recent version of the 3AC, or explain why we can't apply that optimization.

(space for your answers is provided on the next page)

Problem 9 [cont.]

Write and label your answers to parts **A-D** on this page. Here's the original code again:

```
    i := 0;
loop:
    if (i >= n) goto after;
    t1 := i * 4;
    t2 := b[t1];
    t3 := i * 4;
    t4 := c[t3];
    t5 := t2 + t4;
    t6 := i * 4;
    a[t6] := t5;
    i := i + 2;
    goto loop;
after:
```

Problem 10 [14 pts.]

Suppose we are designing an activation record for a language whose functions take the following form, shown by example:

```
def foo(x, y, z) = {var a, b, c; funcBodyExpr }
```

More details about what this function means:

- all values are integers (that's why there are no types on the variable declarations; also, no objects in this language). Integers are 4 bytes.
- **x**, **y**, and **z** are parameters and **a**, **b**, and **c** are local variables
- expressions do not introduce any new variables
- expressions may be function calls, including recursive calls
- We'll be using a stack machine: see additional handout for details on this and pseudo-MIPS.

Part A. Show a possible layout for the activation record for the example function shown above (there's more than one correct answer). In your answer, show where the stack pointer and frame pointer are pointing after constructing the activation record. Show the stack growing downward (i.e., towards the bottom of the page).

Part B. Show the code that would be generated for a reference to **c**, from the example function shown, assuming you are using the activation record you gave in part A. (Your answer should consist of one or more complete MIPS or pseudo-MIPS instructions.)

Part C Show the code that would be generated for a reference to **y**, from the example function shown, assuming you are using the activation record you gave in part A. (Your answer should consist of one or more complete MIPS or pseudo-MIPS instructions.)

Problem 11 [16 pts.]

Consider the Java operator *instanceof* which tests if the run-time type of an object conforms to a particular class. It returns a boolean: true if the run-time type of the object *does* conform to the given type, and false otherwise. In Java this allows for safe downcasting. Here's an example of its use in Java (assume Circle is a subclass of Shape):

```
Shape s;  
.  
.  
.  
if (s instanceof Circle) {  
    return (Circle) s.getRadius(); // getRadius is not defined  
}                                     // for other kinds of Shapes  
else {  
    return 0;  
}
```

Part A [4]. Write a Cool expression equivalent to the if-else statement above (assume the necessary definitions of Shape, Circle, s, and getRadius exist). One difference: instead of using "return", the *value* of expression you write be the same as what was *returned* in the original version. Note: your answer will *not* use instanceof, since Cool doesn't have this feature.

Part B [4]. Consider adding **instanceof** to Cool (same semantics as described above). Here's the syntax:

`expr → expr instanceof Type`

Using logic rules of inference, write a type-rule for instanceof. To remind you of the notation, we'll give you the rule for let-with-init. (Hint: instanceof has a much simpler rule.)

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

Problem 11 [cont.]

Part C [8]. Write a code-generation routine for **instanceof**. Here again is the description of what instanceof does: it tests if the run-time type of an object conforms to a particular class. It returns a boolean: true if the run-time type of the object *does* conform to the given type, and false otherwise.

More precisely, here is the operational semantic rule for **instanceof** in Cool:

$$\boxed{\begin{array}{l} so, E, S \vdash e : v, S_1 \\ v = X(\dots) \\ v_1 = \begin{cases} \text{Bool (true), if } X \leq T \\ \text{Bool (false), otherwise} \end{cases} \\ \hline so, E, S \vdash e \text{ instanceof } T : v_1, S_1 \end{array}}$$

Refer to the additional handout for details on the stack machine we'll be using and on pseudo-MIPS notation. Furthermore, you may assume:

- we already generated type tags for the classes by doing a *preorder* traversal on the inheritance hierarchy. **Hint:** In this scheme, **Object** has type tag 0; if the tag for type T has value v, then the tags for all types in the subtree rooted at T have values greater or equal to v.
- the first word in every object is its type tag.
- 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 `char*`).
- output your code by using `<<` statements to `cout`

(space to write your answer is provided on the next page)

Problem 11 [cont.]

Part C (cont.)

Note: If there are any additional assumptions you need to make (e.g., about what values are where), please state them.

`cgen(e instanceof T) =`