

Name: \_\_\_\_\_

USC ID: \_\_\_\_\_

## CS 410 Final Exam Fall 2000 [Bono]

December 14, 2000

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

	value	score
Problem 1	10 pts.	
Problem 2 & 3	10 pts.	
Problem 4	9 pts.	
Problem 5	12 pts.	
Problem 6	10 pts.	
Problem 7	15 pts.	
Problem 8	12 pts.	
Problem 9	12 pts.	
Problem 10	10 pts.	
<b>TOTAL</b>	100 pts.	

### Problem 1 [10 pts.]

Fill in the blanks and multiple choice.

**Part A.** In type checking rules the *object environment* maps

\_\_\_\_\_ to \_\_\_\_\_.

**Part B.** In rules for operational semantics the *environment* maps

\_\_\_\_\_ to \_\_\_\_\_.

**Part C.** Circle the choice that is always true for an *ambiguous grammar* for a language.

1. undecidable which strings are in the language
2. has multiple leftmost derivations for some sentences
3. substrings of some sentences are also sentences in the language
4. the language has an infinite number of sentences

**Part D.** A shift-reduce parser performs what kind of derivation?

\_\_\_\_\_.

**Part E.** Eliminating left recursion from a grammar solves what problems in a recursive descent parser? \_\_\_\_\_

**Problem 2 [5 pts.]**

**Part A [6].** Do Thompson's construction on the following regular expression ( $\epsilon$  = epsilon):

$ab^* \mid (c|\epsilon)$

**Problem 3 [5 pts.]**

Write a complete grammar for a language that has an infinite number of sentences (use upper case for nonterminals, lower case for terminals):

**Problem 4 [9 pts.]**

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

```
int main ()
{
    . . .
    int a;      /* (i) */
    void sort(...)
    {
        void swap()
        {
            . . . a . . . /* (j) */
            . . .
        }
        void compare() { . . . }
        . . . /* body of sort */
    }
    void output() { . . . }
    . . . /* body of main */
}
```

**Part A.** Show what the values of the access links would be at the points in the program reflected by the each of the two following contents of the call stack (stack grows downwards):

**A1.**

main
sort
swap
sort
compare
output

**A2.**

main
sort
sort
sort
swap

**Part B.** Assuming we are using access links, describe informally how the generated code would get to the variable  $a$  in the reference on line (j) (refers to variable defined at line (i)).

**Problem 5 [12 pts.]**

Consider the following basic block. Assume all of the programmer variables, a through e, are live at the end of the basic block, and the temporary variables, t1 through t5, are not live at the end of the basic block.

```
t1 = 3 * a;
t2 = t1 + 10;
a = t2;
t3 = b * a;
d = t3;
t4 = 3 * a;
c = t4;
t5 = b * a;
e = t5;
```

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

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

**Problem 6 [10 pts.]**

Eliminate left recursion from the following grammar (nonterminals are upper case, terminals lower case).

```
A → A B C
   | d e f
   | g f
   | A f g
B → B C
   | B g
   | e e
C → d C C
   | e f
```

### Problem 7 [15 pts.]

The C *continue* statement may only be used inside a loop. Write semantic rules to check this rule given the following simplified C-subset grammar. In particular, write rules for the synthesized boolean attribute, *legal*, which is true for a statement or statement list if it (and all of its subparts) obeys this rule, and false otherwise. Also, recall that expressions (E, below), and DeclS cannot contain statements.

Hint: use *inherited* boolean attribute *inloop* which signifies that this statement occurs in the context of a loop. We have written the top-level assignment to this inherited attribute (and the rest of the first rule) for you.

Note: subscripts are just used to distinguish different instances of a nonterminal.

Following while statement is legal:      Following if statement is illegal (not inside a loop):

```

while (i < 10) {
  if (3 < m) {
    if (y > x) continue;
  }
  i++;
}

main() {
  if (3 < m) {
    if (y > x) continue;
  }
}

```

```

FuncBody → { DeclS SList }   { SList.inloop = false;
                               FuncBody.legal = SList.legal; }

```

S → E ;

| continue ;

| { SList }

| while (E) S<sub>1</sub>

| if (E) S<sub>1</sub>

SList → ε

| SList<sub>1</sub> S

### Problem 8 [12 pts.]

This question concerns the following type-checking rule for Cool case expressions:

$$\frac{\begin{array}{c} O, M \mid - e_0 : T_0 \\ O[T_1/x_1], M \mid - e_1 : T_1' \\ \vdots \\ O[T_n/x_n], M \mid - e_n : T_n' \end{array}}{O, M \mid - \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \dots x_n : T_n \Rightarrow e_n; \text{ esac} : \text{lub}_{1 \leq i \leq n} (T_i')}$$

**Part A [2].** Describe any restrictions on the type of  $e_0$ .

**Part A [2].** Describe any restrictions on the relationship between  $T_i$  and  $T_i'$ .

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

**Part A [3].** Is the following case expression legal? If it is, describe the semantics. If not, explain or show the part of the type rule above that disallows it.

```

case x of
  x : T => expr;
esac;

```

**Part A [3].** The *types* of the variables declared in each branch must be distinct, but this information is not in the type rule itself (it's mentioned in a separate sentence in CoolAid). The type rule also says nothing about whether the *names* of the declared variables in each branch have to be distinct. Do they?  E.g., is the following case expression legal? Explain why or why not.

```

case x of
  y : T1 => expr1;
  y : T2 => expr2;
esac;

```

### Problem 9 [12 pts.]

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

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

class B inherits A {
  b : Int;
  b2 : Int;
  g() : Int { . . . }
};

class C inherits B {
  c : Int;
  f() : Int { . . . }
  h() : Int { . . . }
};
```

In your answers to the following questions, assume that A is the base class (not inherited from Object). Assume dispatch tables for A, B, and C are at addresses `A_dispTab`, `B_dispTab`, `C_dispTab`, respectively.

**Part A.** Show the layout of the dispatch tables for classes A, B, and C. You may denote the address of the function `bar` defined in class `Foo` as `Foo.bar`.

**Part B.** Show the layout of the objects of classes A, B, and C.

Circle and label your answers.

### Problem 10 [10 pts.]

Write a routine, `cgen`, to generate code for a `repeat-until` loop. Here is what it looks like:

```
repeat
  e1
until e2;
```

The repeat until loop repeatedly evaluates `e1` until `e2` is true (it's like a do-while, but with the opposite test). This construct is a statement, thus does not return any value.

Assumptions about generating code:

- the code generation scheme will be similar to the one we discussed in lecture for a stack-machine:
  - There is a frame pointer `$fp`
  - temporaries are pushed on the stack, not put in the frame
  - the results of expressions are always left in `$a0`
  - evaluating an expression leaves the stack pointer, `$sp`, with the same value it had before evaluating the expression
  - 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 `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;
```

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

(Space given for your answer on the next page)

**Problem 10 (cont.)**

`cgen(repeat e1 until e2) =`