Name: ______________________________

USC ID:______________________

CS 410 Final Exam
Spring 2001 [Bono]

May 1, 2001

There are 8 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 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	12 pts.	
Problem 3	15 pts.	
Problem 4a	5 pts.	
Problem 4bc	16 pts.	
Problem 5	6 pts.	
Problem 6	16 pts.	
Problem 7	10 pts.	
Problem 8	10 pts.	
TOTAL	100 pts.	

Problem 1 [10 pts.]

Fill in the blanks.

Part A. Abstract machines for recognizing languages that can be denoted with regular expressions are called ________________________________.

Part B. Thompson's construction is an algorithm to create a

____

from a ____________________________________.

Part C. Subset construction is an algorithm to create a

____

from a ____________________________________.

Part D. ______________________ is an example of a tool that uses the above algorithms to automatically create a ________________________________ from a

__________________________.

Part E. A top-down parser performs what kind of derivation?

______________________________.

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

________________________________ to ____________________________________.

Problem 2 [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 = 1;
   y = 9;
   A[k] = 5;
   x++;
}

main ()
{
   A[0] = 3;
   A[1] = 7;
   k = 0;
   foo (A[k], A[1], 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 3 [15 pts.]

Consider the following basic block. Assume all of the programmer variables, A, B, i, j, and m, are live at the end of the basic block, and the temporary variables, t1 through t9, are not live at the end of the basic block.

Part A. Show the DAG representation for this basic block (circle and label your answer).

Part B. Generate an improved 3AC sequence from your DAG (circle and label your answer).

Part C. The C source code that resulted in the original 3AC below referenced A[i][j]. A is stored in row-major order. For each of the following questions give a number as an answer or say "can't tell", if the answer can't be determined from the code given.

How many rows are in A?
How many columns are in A?
What is the size (in bytes) of one element in A?

```
t1 = 10 *  i;
t2 = t1 +  j;
t3 =  4 * t2;
t4 = A[t3];
t5 = 10 *  i;
t6 = t5 +  j;
t7 =  4 * t6;
t8 =  B[t7];
t9 = t4 + t8;
m  = t9;
```

Problem 4 [21 pts.]

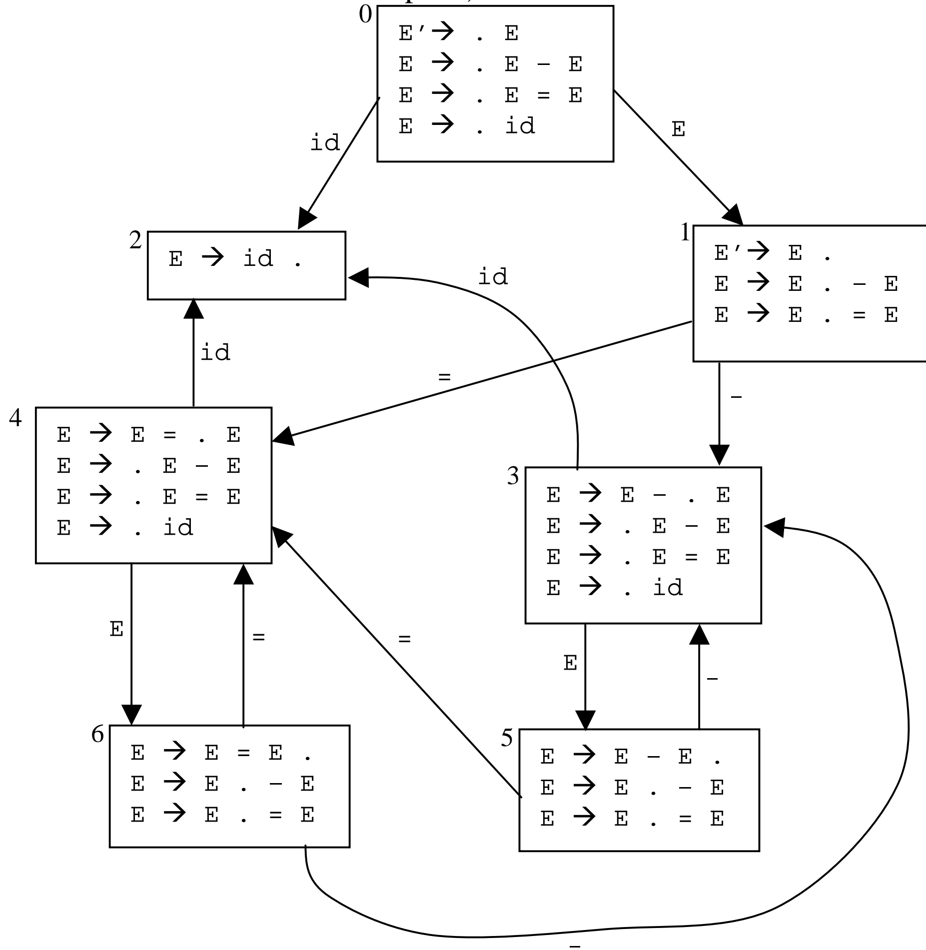
Consider the following expression grammar (nonterminals are upper case, terminals lower case):

```
E → E - E
  | E = E
  | id
```

Part A [5]. Show that the grammar is ambiguous.

(problem continues on the following page)

Part B [8]. The following is the DFA to recognize viable prefixes for an SLR parser for an augmented version of the ambiguous grammar from the previous page. Explain where all the shift-reduce and/or reduce-reduce conflicts occur (for what states, on what inputs, and involving what rules for the reduce parts).



Part C [8]. Suppose we want the following interpretation for expressions parsed with the grammar:

- \- (minus) expressions are left-associative (e.g., `a-b-c` is interpreted as `(a-b)-c`
- = (assignment) expressions are right-associative (e.g., `a=b=c` is interpreted as `a=(b=c)`
- minus has higher precedence than =

For each of the conflicts you got for part B, give a default action that the parser can take to get the interpretation given above. Your answer should be of the form

action[*state*, *input*] = reduce *rule* or

action[*state*, *input*] = shift

for each conflict, such that you will give the values for *state*, *input*, and *rule*.

Problem 5 [6 pts.]

In the following partial Cool program for each of the expressions with a box around it tell us in what part of memory its value is to be found at run-time. Your answer for each will be one of the following (you can use the numbers to identify them in your answer if you wish):

1. code segment
2. static memory
3. stack
4. heap

Note: recall that all Cool values are pointers. We are asking where certain pointers are, not where the things they point to are.

Note2: assume we don't keep things in registers, or do other optimizations (in particular assume none of the code below has been optimized away).

You can put your answers next to the line of code the expression appears on.

```
class A is
  a : Int <- [3] ;
  b : Int <- [a] *  27;
  func(c : Int) : Int is
    let d : Int in
      begin
       [d];
       [b];
       [c];
       [a * 2]+ 10 / b ;
      end
    end  -- of let
   end; -- of func
end;
```

Problem 6 [16 pts.]

One of the rules for statements in C is that case-labeled statements and default-labeled statements are only allowed to appear inside of switch statements. Because other statements besides this are also allowed to appear in switch statements, this rule is not context-free: i.e., it can't be described with the grammar itself. Furthermore, a statement labeled with a case or default may not just be one level down from the switch, but may also be nested within several other statements that appear inside the switch (see second example below). Here are some examples of legal and illegal statement lists for a function body:

```
void foo() {
  while (expr) {      /* EX 1: not legal */
      case 3: expr;             /* a case-labeled statement */
  }                             /* not inside a switch */
}

void bar() {
  switch ( expr ) {  /* Ex 2: legal */
    while (expr) {
      if (expr)
          expr;
      else
          default : expr;     /* a default-labeled statement */
    }                         /* inside a switch */
  }
}
```

(problem continued next page)

Problem 6 (cont.)

You are going to write semantic rules for the following simplified partial C grammar such that it computes the synthesized boolean attribute `legal`, which is true for a statement or statement list if it and all of its subparts obeys the aforementioned rule about case and default placement, false otherwise.

A few notes about the grammar:

- terminals consist of punctuation and bold identifiers; other identifiers are nonterminals; ε denotes the empty string
- subscripts are just used to distinguish different instances of a nonterminal
- since variable declarations and expressions can't contain statements inside of them, we can treat them like terminals for the purposes of this problem (and we use terminals **`decls`**, **`expr`**, and **`constExpr`** respectively, below)

Hint: use *inherited* boolean attribute `insw`, which signifies that this statement appears somewhere inside of a switch statement. We have written the top-level assignment to this inherited attribute (and the rest of the first rule) for you.

FuncBody → { **decls** SL } *{SL.insw = false;*
FuncBody.legal = SL.legal; }

SL → ε

| S SL_1

S → **expr ;**

| **switch** (**expr**) S_1

| **default** : S_1

| **case constExpr** : S_1

| **while** (**expr**) S_1

| { SL }

Problem 7 [10 pts.]

The least upper bound (LUB) of two Cool types is defined as the least type that they both conform to. Also, in case you forgot the direction of the conformance relationship, for all classes, C, C conforms to Object (C ≤ Object).

Write a C++ function LUB, which returns the type that is the LUB of the two types given. For the purposes of this question, a type will be represented by its name stored in a `string` variable. `string` is a class in the standard C++ library. You can compare two strings with `==`, assign one to another with `=`, compare them with or copy them from C string literals. For example:

```
string s = "Hello";
```

Furthermore, assume you have at your disposal the following auxiliary functions (i.e., you can call them – don't write them):

- **`string parentOf(string C);`**
  Returns the name of the type C inherits from (i.e., C's superclass) in the cool program we're compiling. For class "Object", returns the empty string (note: for empty `string`, `s`, `s.size() == 0`).
- **`boolean conformsTo(string T1, string T2);`**
  Returns true iff T1 conforms to T2 (i.e, T1 ≤ T2).

```
// PRECONDITION: A and B are valid types in the cool program we're compiling
string LUB(string A, string B)
```

Problem 8 [10 pts.]

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

```
repeat
  e1
until e2
```

The repeat-until loop always evaluates the body of the loop (i.e., e_1) at least once, and then evaluates the test (i.e., e_2). When 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. Note: Unlike the Cool while expression, repeat-until has an interesting value.

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 8 (cont.)

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

cgen(**repeat e_1 until e_2**) =