Login Name ____________________ **Name** ________________________

Signature ______________________ **Student ID** ________________________

Midterm
CSE 131
Winter 2011

Page 1	___________	**(23 points)**
Page 2	___________	**(19 points)**
Page 3	___________	**(30 points)**
Page 4	___________	**(32 points)**
Page 5	___________	**(15 points)**
Page 6	___________	**(24 points)**
Subtotal	___________	**(143 points = 100%)**
Page 7 Extra Credit	___________	**(9 points)**
Total	___________	

This exam is to be taken by yourself with closed books, closed notes, no electronic devices.
You are allowed one side of an 8.5"x11" sheet of paper handwritten by you.

1. Given the following CUP grammar snippet (assuming all other Lexing and terminals are correct):

```
Expr ::=   Des AssignOp {: System.out.println("Z"); :} Expr {: System.out.println("A"); :}
     |     Des {: System.out.println("B"); :}
     ;

Des  ::=   T_STAR {: System.out.println("C"); :} Des {: System.out.println("N"); :}
     |     T_PLUSPLUS {: System.out.println("D"); :} Des {: System.out.println("M"); :}
     |     T_AMPERSAND {: System.out.println("E"); :} Des {: System.out.println("L"); :}
     |     Des2 {: System.out.println("F"); :}
     ;

Des2 ::=   Des2 {: System.out.println("G"); :} T_PLUSPLUS {: System.out.println("K"); :}
     |     Des3 {: System.out.println("H"); :}
     ;

Des3 ::=   T_ID {: System.out.println("I"); :}
     ;

AssignOp ::=  T_ASSIGN {: System.out.println("J"); :}
     ;
```

What is the output when parsing the follow expression (you should have 20 lines/letters in your output):

```
x = *y = z++
```

Output

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

In the above grammar, which has higher precedence: pre-increment operator or post-increment operator or do they have the same precedence? ______________________

If variable `z` is defined to be type `short *`, what types must variables `y` and `x` be defined for this expression to be semantically correct?

______________________ `y;` ______________________ `x;`

2. Give the order of the phases of compilation in a typical C compiler as discussed in class

0 – Scanner (Lexical Analysis)
1 – Parser (Semantic Analysis)
2 – Parser (Syntax Analysis)
3 – Target language file (for ex., prog.s)
4 – Source language file (for example, prog.c)
5 – Intermediate Representation(s)
6 – Code generation (for ex., Assembly)

______ –> ______ –> ______ –> ______ –> ______ –> ______ –> ______

In RC (and C/C++), we do not support the assignment of an entire array to another array (of the same type) using the assignment operator. However, we do support assignment of an entire struct instance to another struct instance of the same type. Using this fact, fill in the template for the code below, which allows arrays to piggy-back on a struct type to simulate entire-array assignments that are semantically and logically correct.

```
structdef INTARR5 { int [5] a; };
int [5] a, b;
function : void foo()
{
     //  a = b would be a semantic error, but....

     ______ _____________________ ______ a _______ = ______ _____________________ ______ b _______;
}
```

Using Reduced-C syntax, define an array of an array of floats with dimensions 3x11 named bar such that `bar[2][10] = 42.42;` is a valid expression. This will take two lines of code.

In RC, the following code produces no errors:

```
typedef int MONTH;
typedef int DAY;
typedef int YEAR;
MONTH mm;
DAY dd = mm;
YEAR yy = dd;
```

The Project I specs define a particular equivalence rule to be used for types created using typedefs. How would you change that rule so that the assignments in the last two lines of the code above produce an assignability error? State what the current rule is and what you would change it to.

Current equivalence rule for typedefs:

How would you change the equivalence rule for typedefs to produce an assignability error in the last two lines

3. Given the following C++ definitions (similar to Reduced-C):

```
float foo1( float & a ) { int b; return b; }
float foo2( float a )   { int b; return b; }
float foo3( float * a ) { int b; return b; }

float x;
int y;
float z[5];
```

For each of the following statements, indicate the type of error (if any) that should be reported (using the Project I spec for this quarter which is similar to the C++ rules). Use the letters associated with the available errors in the box below.

A) Arg passed to reference param is not a modifiable L-val
B) Argument not equivalent to reference param
C) Argument not assignable to value param
D) No Error

```
x = foo1( 4.2 );                 _____
x = foo1( y );                   _____
x = foo1( x );                   _____
x = foo1( foo2( x ) );           _____
x = foo1( *(float *)&y );        _____
x = foo1( (float)y );            _____
x = foo1( z[2] );                _____
x = foo1( x + y );               _____

x = foo2( x );                   _____
x = foo2( 4.2 );                 _____
x = foo2( foo2( x ) );           _____
x = foo2( &x );                  _____
x = foo2( *(float *)&y );        _____
x = foo2( y );                   _____
x = foo2( x + y );               _____

x = foo3( z );                   _____
x = foo3( &y );                  _____
x = foo3( &x );                  _____
x = foo3( (float *)&y );         _____
x = foo3( foo2( x ) );           _____
x = foo3( &z[2] );               _____
```

Using the Right-Left rule (which follows the operator precedence rules) write the C definition of a variable named foo that is an array of 5 elements where each element is a pointer to a function that takes a pointer to a pointer to a struct Baz as a single parameter and returns a pointer to an array of 11 elements where each element is a pointer to a pointer to a struct Fubar. (9 points)

4. Consider the following struct definitions in Reduced-C (similar to C/C++). Specify the size of each struct on a typical RISC architecture (like ieng9) or 0 if it is an illegal definition.

```
structdef FOO3 {
  FOO3 *a;
  float b;
  function : void bar( FOO3 &x)
  {
    FOO3 *y;
  }
  int  *c;
  int   d;
};
```

Size _______

```
structdef FOO2 {
  int   a;
  float b;
  function : void bar()
  {
    int x = *this.d;
  }
  FOO2 *c;
  int  *d;
};
```

Size _______

```
structdef FOO1 {
  int    a;
  float  b;
  function : void bar()
  {
    FOO1 *x;
  }
  FOO1   c;
  int[2] d;
};
```

Size _______

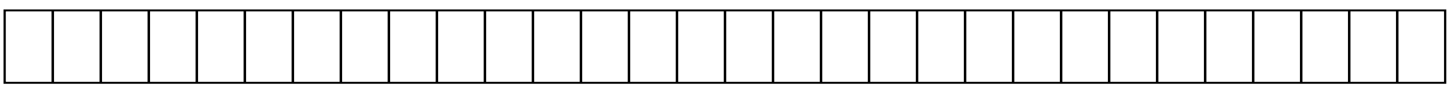
Given the C array declaration

C

```
    int a[4][2];
```

Mark with a **C** the memory location(s) where we would find

```
    a[1][0]
a:
```



```
low memory                                                   high memory
```

Each box represents a byte in memory.

Modifiable L-vals, Non-Modifiable L-vals, R-vals

Using the Reduced-C Spec (which closely follows the real C language standard), given the definitions below, indicate whether each expression evaluates to either a

A) Non-Modifiable L-val B) R-val C) Modifiable L-val

```
function : int * foo() { /* Function body not important. */ }
float[9] a;
float x;
const float y = 5.5;
float *p = &x;
```

____ foo()	____ 4.2	____ *foo()	____ (int)x	____ *(int *)p
____ p	____ a[2]	____ (int *)&x	____ *(int *)&x	____ (float *)foo()
____ x	____ *p	____ **&p	____ y	____ *foo() * y
____ &x	____ a	____ &*p	____ *p - y	____ *(float *)foo()
____ x++	____ ++x	____ a[2]++	____ &a[0]	____ ++*foo()

5. Show the memory layout of the following C struct definition taking into consideration the **SPARC** data type memory alignment restrictions discussed in class. Fill bytes in memory with the appropriate struct member/field name. For example, if member/field name `p` takes 4 bytes, you will have 4 `p`'s in the appropriate memory locations. If the member/field is an array, use the name followed by the index number. For example, some number of `p[0]`s, `p[1]`s, `p[2]`s, etc. If the member/field is a struct, use the member name followed by its member names (e.g. p.a, p.b). Place an `X` in any bytes of padding. Underline this sentence for a point. Structs and unions are padded so the total size is evenly divisible by the most strict alignment requirement of its members.

```
struct foo
{
  char   a;
  double b;
};

struct bar
{
  short      c[2];
  char       d[3];
  double     e;
  struct foo f[2];
};

struct bar oof;
```

`low memory`
`oof:`

`high memory`

What is the `sizeof( struct bar )`? _____ What is the `offsetof( struct bar, f[1].b )`? _____

If `struct bar` had been defined as `union bar` instead, what would be the `sizeof(union bar)`? _____

What is the resulting type of the following expression?

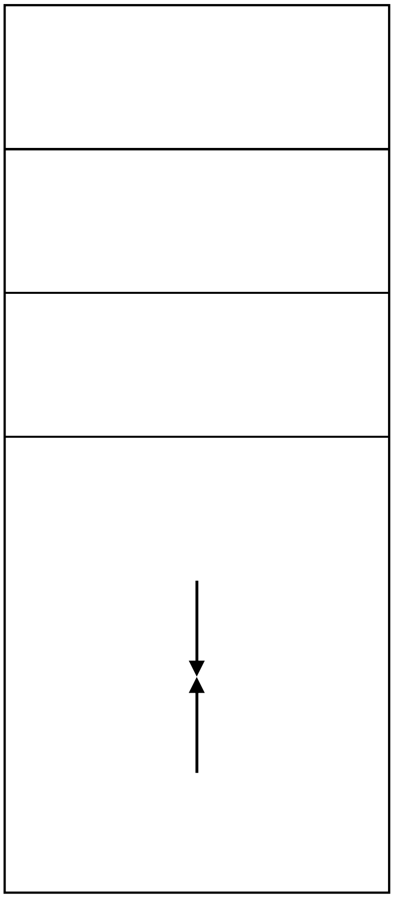
`( ((struct foo *) &oof) + 2 ) -> b` _______________

Write the equivalent expression that directly accesses this value/memory location without all the fancy casting and `&` operators.

`oof.`________________

6. Fill in the names of the 5 main areas of the C Runtime Environment as laid out by most Unix operating systems (and Solaris on SPARC architecture in particular) as discussed in class. Then state what parts of a C program are in each area.

low memory



high memory

Which 3 areas of the above are essentially determined at compile time and are part of an executable image (for example, an a.out or .exe file)?

1) ____________________________

2) ____________________________

3) ____________________________

Give the order of the typical C compilation stages and on to actual execution as discussed in class

1 – prog.exe/a.out (Executable image)
2 – Object file (prog.o)
3 – Loader
4 – as (Assember)
5 – Program Execution
6 – cpp (C preprocessor)
7 – Assembly file (prog.s)
8 – ccomp (C compiler)
9 – Source file (prog.c)
10 – ld (Linkage Editor)
11 – Segmentation Fault (Core Dump) / General Protection Fault

gcc ____ –> ____ –> ____ –> ____ –> ____ –> ____ –> ____ –> ____ –> ____ –> ____ –> ____

Extra Credit

What gets printed when the following C program is executed?

```
#include <stdio.h>

int
main()
{
  char a[] = "ANKUR^2";
  char *p = a;

  printf( "%c\n", *p++ );                      ______

  printf( "%c\n", *(p+2) = ++p[1] );           ______

  printf( "%c\n", *++p = 2["BRUCE"] );         ______

  printf( "%c\n", ++*++p );                    ______

  printf( "%c\n", p[1] = p[-3] );              ______

  printf( "%c\n", *++p + 1 );                  ______

  printf( "%c\n", (p+1)[0] = *(p-1) );         ______

  printf( "%c\n", (++p)[1] = **&a );           ______

  printf( "%s\n", a );                         ______________________

  return 0;
}
```

```
A portion of the C Operator Precedence Table

Operator                Associativity

++ postfix increment    L to R
-- postfix decrement
[] array element
() function call
--------------------------------------
*  indirection          R to L
++ prefix increment
-- prefix decrement
&  address-of
sizeof size of type/object
(type) type cast
--------------------------------------
*  multiplication       L to R
/  division
%  modulus
--------------------------------------
+  addition             L to R
-  subtraction
--------------------------------------
      .
      .
      .
--------------------------------------
=  assignment           R to L
```

Scratch Paper

Scratch Paper