Signature ____________________ **Name** ________________________

Login Name __________________ **Student ID** ____________________

Midterm
CSE 131
Spring 2009

Page 1	___________	**(14 points)**
Page 2	___________	**(48 points)**
Page 3	___________	**(30 points)**
Page 4	___________	**(14 points)**
Page 5	___________	**(18 points)**
Page 6	___________	**(15 points)**
Subtotal	___________	**(139 points)**
Page 7 **Extra Credit**	___________	**(14 points)**
Total	___________	

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

```
Stmt ::=   Des AssignOp Des T_SEMI {: System.out.println("1"); :}
     ;

Des  ::=   T_PLUSPLUS {: System.out.println("3"); :} Des {: System.out.println("17"); :}
     |     T_STAR {: System.out.println("5"); :} Des {: System.out.println("19"); :}
     |     Des2 {: System.out.println("7"); :}
     ;

Des2 ::=   Des2 {: System.out.println("9"); :} T_PLUSPLUS {: System.out.println("21"); :}
     |     Des3 {: System.out.println("11"); :}
     ;

Des3 ::=   T_ID {: System.out.println("13"); :}
     ;

AssignOp ::=  T_ASSIGN {: System.out.println("15"); :}
     ;
```

What is the output when parsing the follow statement (you should have 14 lines/numbers in your output):

```
x = ++*ptr++;
```

Output

Which operator in which production has higher precedence in the above grammar:
the T_PLUSPLUS in Des production or the T_PLUSPLUS in Des2 production?

____

Is this the pre-increment or the post-increment operator? __________________

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

A – Parser (Semantic Analysis)	E – Scanner (Lexical Analysis)
B – Target language file (for ex., prog.s)	F – Parser (Syntax Analysis)
C – Source language file (for example, prog.c)	G – Intermediate Representation(s)
D – Code generation (for ex., Assembly)	

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

Machine-specific code improvements typically can occur immediately before, during, and/or immediately after which phase? ______

Machine-independent improvements typically can occur immediately before, during, and/or immediately after which phase? ______

Using Reduced-C syntax, define an array of an array of floats with dimensions 7x4 named bar such that `bar[6][3] = 42.24;` is a valid expression. This will take two lines of code.

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) Modifiable L-val B) Non-Modifiable L-val C) R-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()
```

Specify the sizes of the various data types listed for the following Compiler Models

	ILP-32	LP-64	LLP-64
int	______	______	______
long	______	______	______
long long	______	______	______
pointer	______	______	______

3. Given the following C++ definitions:

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

```
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 fubar that is an array of 3 elements where each element is a pointer to a function that takes a pointer to a struct Foo as a single parameter and returns a pointer to an array of 8 elements where each element is a pointer to a pointer to a struct Baz. (9 points)

4. State whether constant folding can be performed by the compiler according to this quarter's Reduced-C spec in the following Reduced-C statements (**Y** or **N**)

```
function : void foo()
{
  const int a = 5;
  int b = 3;

  const int c = a + 10;      ______

  int[53 + c] d;             ______

  b = d[d[2] + c];           ______

  d[-2 + (a * b)] = c;       ______

  int e = d[a + c];          ______

  d[5 - 2 + c] = e;          ______

  e = d[13 + b];             ______

  e = d[e + a];              ______
}
```

Fill in the blanks of the following Reduced-C program with correct types to test if your fix to the scoping bug present in the starterCode works correctly. If the scoping bug is fixed, this program should compile without error. If the bug is not fixed, this program should generate an assignment error at the line x = y;

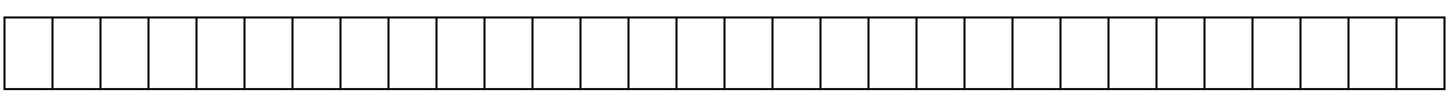
```
_______  x;      // global x

function : int main() {

  _______  x;    // local x

  int y;

  x = y;         // If fixed in Phase 0, this line will not cause an error!
                 // If not fixed in Phase 0, this line will cause an error!
  return 0;
}
```

Given the C array declaration

C
```
      int a[3][3];
```

Mark with an **A** the memory location(s) where we would find

```
      a[1][2]
a:
```



`low memory` `high memory`

Each box represents a byte in memory.

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. Circle the page number below for an extra 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;
  short  b;
  double c;
  int    d;
};

struct fubar {
  short      e;
  struct foo f;
  char       g[7];
  short      h;
};

struct fubar fubaz;
```

low memory

fubaz:

high memory

What is the `sizeof( struct fubar )`? _____ What is the `offsetof( struct fubar, g[1] )`? _____

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

What is the resulting type of the following expression?

`* (char *) & ( ( (struct fubar *) & fubaz.f ) -> e )` _______________

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

`fubaz.`_______________

6. Given the following C program:

```
#define X 3
#define Y 5

int   a[X][Y];
int * b[X];

int main()
{
  int i;

  for ( i = 0; i < X; i++ )
    b[i] = malloc( sizeof(int) * Y );

  return 0;
}
```

Match the following expressions with the corresponding type (think type equivalence) from the list A-P. Use type equivalence rules, not assignability.

```
b[1]         _____
b            _____
&a[0]        _____
**b          _____
a[1]         _____
a            _____
*(a + 2)     _____
*b[1]        _____
```

```
A. int
B. int *
C. int[5]
D. int[3]
E. int[3][5]
F. int[5][3]
G. int (*)[3]
H. int (*)[5]
I. int (*)[3][5]
J. int (*)[5][3]
K. int* [3]
L. int* [5]
M. int* [3][5]
N. int* [5][3]
O. int* (*)[3]
P. int* (*)[5]
```

Fill in the blanks to make the array expression below equivalent to the following pointer expression. Note: You cannot use negative numbers in the array expression!

```
*(*(a + 1) - 2) is equivalent to a[___][___]
```

We can access the underlying data associated with `a` and `b` (as defined in the program above) using the same array or pointer expressions. However their underlying structure is different for each other.

What is the total number of bytes allocated to the entire data structure for `a`? ______
What is the total number of bytes allocated to the entire data structure for `b` including any memory dynamically allocated and associated with and reachable by `b`? ______

Assume we want to add a traversal pointer to more efficiently traverse the array `a` above. How would you define and initialize this traversal pointer?

```
____________________ ptr = _______________________ ;
```

Using this traversal pointer, write a pointer expression (with no array brackets []) to access the same array element as `a[2][3]`

____

Extra Credit (14 points)

What gets printed when the following C program is executed?

```
#include <stdio.h>

int
main()
{
  char a[] = "ABYSS";
  char *p = a;

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

  printf( "%c", *(p+2) = p[3] - 1 );          ______

  printf( "%c", *++p = 3["DINO"] );            ______

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

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

  printf( "%d", ++p - a );                     ______

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

  return 0;
}
```

```
A portion of the Operator Precedence Table

Operator                 Associativity

++ postfix increment     L to R
-- postfix decrement
[] array element
--------------------------------------
*  indirection           R to L
++ prefix increment
-- prefix decrement
&  address-of
--------------------------------------
*  multiplication        L to R
/  division
%  modulus
--------------------------------------
+  addition              L to R
-  subtraction
--------------------------------------
      .
      .
      .
--------------------------------------
=  assignment            R to L
```

Scratch Paper

Scratch Paper