Signature _____ **Name _____**

Login Name _____ **Student ID _____**

Midterm
CSE 131
Winter 2009

Page 1	**_____ (25 points)**
Page 2	**_____ (21 points)**
Page 3	**_____ (22 points)**
Page 4	**_____ (21 points)**
Page 5	**_____ (14 points)**
Page 6	**_____ (15 points)**
Subtotal	**_____(118 points)**
Page 6 Extra Credit	**_____ (7 points)**
Total	**_____**

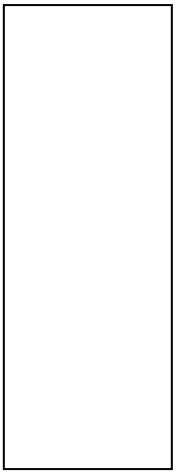
1. Given the following CUP grammar snippet (assuming all other Lexing and terminals are correct): (7 pts)

```
Stmt ::=        Designator T_ASSIGN Expr T_SEMI
                {: System.out.println("A"); :}
     ;

Expr ::=        Expr MulOp {: System.out.println("B"); :} Designator
     |          Designator {: System.out.println("C"); :}
     ;

Designator ::= T_ID {: System.out.println("D"); :}
     ;

MulOp ::=       T_STAR {: System.out.println("E"); :}
     ;
```



What is the output on the screen when the follow statement is given as input:

```
a = b * c;
```

Using the Right-Left rule (which follows the operator precedence rules) write the definition of a variable named foo that is a 2-d array of 3 rows by 5 columns where each element is a pointer to an array of 7 elements where each element is a pointer to a function that takes a pointer to a pointer to a float as a single parameter and returns a pointer to an array of 9 elements where each element is a pointer to a struct bar. (10 pts)

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**) (8 pts)

```
function : void foo()
{
  const int a = 5;
  int b = 17;

  const int c = a + 10;     ______

  int[3 + c] d;             ______

  d[13 + (a * b)] = c;      ______

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

  int e = d[a + c];         ______

  e = d[13 + b];            ______

  e = d[e + a];             ______

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

2. Given the following Reduced-C program and following the Project I spec for parameter passing type checking, for each function call determine if a semantic error will occur (and which kind of error). (21 pts)

A. No Error
B. Equivalence Error
C. Addressability Error
D. Assignability Error

```
function : void foo0 ( float[5] & x ) { /* ... */ }
function : void foo1 ( float * & x ) { /* ... */ }
function : void foo2 ( int x ) { /* ... */ }
function : void foo3 ( float & x ) { /* ... */ }
function : void foo4 ( float x ) { /* ... */ }
function : void foo5 ( float * x ) { /* ... */ }

function : void main()
{
  float a;
  int b;
  int[5] c;
  float[5] d;
  float * e;

  foo0( c );                _____
  foo0( d );                _____

  foo1( e );                _____
  foo1( &a );               _____
  foo1( (float *) &b );     _____

  foo2( a );                _____
  foo2( b );                _____

  foo3( b );                _____
  foo3( a );                _____
  foo3( e );                _____
  foo3( *&a );              _____
  foo3( a + b );            _____

  foo4( e );                _____
  foo4( b );                _____
  foo4( a );                _____

  foo5( &b );               _____
  foo5( (float *) &b );     _____
  foo5( &a );               _____
  foo5( c );                _____
  foo5( d );                _____
  foo5( e );                _____
}
```

3. The types in Reduced-C variable definitions are often unnecessary in the sense that it may be possible to infer variables' types and detect type errors simply from their use. For each of the following program fragments, find a set of types that makes it legal, and write a Reduced-C definition for each variable. If there is more than one possible type, choose only one. If there is none, write "NONE". Assume all arrays are of size 4. (2 pts each)

```
if ( a && (b != c) )
    a = b;
```

_____ a ;

_____ b ;

_____ c ;

```
b = (*a)[b];
```

_____ /* a requires two lines of Reduced-C */

_____ a ; /* to properly define it */

_____ b ;

```
a = 5.5;

if( d != b )
    d = (c % b) / a;
```

_____ a ;

_____ b ;

_____ c ;

_____ d ;

What is an advantage of defining multi-dimensional arrays as contiguous memory locations as opposed to allocating the equivalent as arrays of arrays in C/C++, especially when needing to access each array element in order.

4. Consider the following struct definitions. Specify the size of each struct on a typical RISC architecture (like ieng9) or 0 if it is an illegal definition. (6 pts)

```
struct foo {
  int a;
  double b;
  struct foo c;
  char d[4];
};
```

Size _____

```
struct foo {
  int a;
  double b;
  struct foo *c[3];
  char d[4];
};
```

Size _____

```
struct foo {
  int a;
  double b;
  struct foo c[2];
  char d[4];
};
```

Size _____

Using Reduced-C syntax, define an array of array of int named `foo` with dimensions 7 x 13 (7 rows, 13 cols) such that `foo[6][12]` is a valid index expression. This will take two lines of code. (4 pts)

Assume the following Reduced-C definitions are correct: (8 pts)

```
structdef RECA
{
   int * ptr;
};

structdef RECB
{
  RECA * ptr;
};

RECB * ptr;
```

a) What type is `ptr->ptr` ? _____

b) What type is `*(ptr->ptr->ptr)` ? _____

c) What type is `*(ptr->ptr)` ? _____

d) What type is `(*(*ptr).ptr).ptr` ? _____

From our Reduced-C spec, name a construct which uses (3 pts)

loose name equivalence _____

strict name equivalence _____

structural equivalence _____

5. According to this quarter's Reduced-C grammar, what two constructs must be uppercase symbols?

_____ _____

Given the following CUP grammar rules

```
Expr1 ::=           Expr1 T_OP1 Expr2
    |               Expr2
    ;

Expr2 ::=           Expr3 T_OP2 Expr2
    |               Expr3
    ;
```

Which operator has higher precedence (OP1 or OP2)? _____

What is the associativity of the OP1? _____

What is the associativity of the OP2? _____

According to the Project 1 spec, the address-of operator requires its operand to be _____

and the result of this expression is not _____ and not _____, or in other

words the result is a(n) _____ (what is the proper term).

According to the Project 1 spec, what is the compile-time size of the following in this Reduced-C program?

```
structdef REC { int a, b; float c; };

int[4] a;
REC b;

int x;

function : void foo( int[4] & p1, REC * p2 ) {

  x = sizeof( p1 );  // should be ________

  x = sizeof( p2 );  // should be ________
}

function : void main() {

  x = sizeof( b );   // should be ________

  x = sizeof( a );   // should be ________

  foo( a, &b );
}
```

In a call to a struct member function like

```
structPtr->foo();
```

what does the `this` keyword in the function `foo()` refer to? Be specific using the above expression.

6. Show the memory layout of the following C struct/record definition taking into consideration the **SPARC** data type memory alignment restrictions discussed in class. Fill bytes in memory with the appropriate struct/record 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 it's member names (e.g. p.a, p.b). Place an `X` in any bytes of padding. Structs and unions are padded so the total size is evenly divisible by the most strict alignment requirement of its members.

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

struct fubar {
  float d;
  int   e;
  char  f[5];
  struct foo g;
  short h;
};

struct fubar fubaz;
```

low memory
fubaz:

high memory

What is the `offsetof( struct fubar, g.b )`? _____

What is the `sizeof( struct fubar )`? _____

What is the resulting type of the following expression (pure beauty)?

```
* (char *) & ( ( (struct foo *) & fubaz ) -> b ) _____
```

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

```
fubaz._____
```

Extra Credit (7 points)

What gets printed when the following C program is executed?

```
#include <stdio.h>

int
main()
{
  char a[] = "CSE030 Rolls!";
  char *p = a + 2;

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

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

  printf( "%c", ++p[2] );                  ______

  p = p + 4;

  printf( "%c", *++p = a[11] + 2 );        ______

  p++;

  printf( "%c", a[10] = *++p - 7 );        ______

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

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

  return 0;
}
```

A portion of the Operator Precedence Table

Operator	Associativity
++ postfix increment -- postfix decrement	L to R
* indirection ++ prefix increment -- prefix decrement & address-of	R to L
* multiplication / division % modulus	L to R
+ addition - subtraction	L to R
.	
= assignment	R to L

Scratch Paper

Scratch Paper