Signature _____ **Name** _____

Login Name _____ **Student ID** _____

Midterm
CSE 131
Spring 2008

Page 1	_____	**(28 points)**
Page 2	_____	**(21 points)**
Page 3	_____	**(20 points)**
Page 4	_____	**(18 points)**
Page 5	_____	**(26 points)**
Page 6	_____	**(16 points)**
Subtotal	_____	**(129 points)**
Page 6 Extra Credit	_____	**(8 points)**
Total	_____	

1. Which of the following would be correct if we wanted to add the minus sign (-) as an operator with higher precedence than the current plus sign (+)? _____ (2 pts)

A

```
Expr :: =   Expr T_PLUS Expr1
      |     Expr1
      ;

Expr1 ::=   Expr1 T_MINUS Designator
      |     Designator
      ;
```

B

```
Expr :: =   Expr Op Designator
      |     Designator
      ;

Op ::=      T_PLUS
      |     T_MINUS
      ;
```

C

```
Expr :: =   Expr T_MINUS Expr1
      |     Expr1
      ;

Expr1 ::=   Expr1 T_PLUS Designator
      |     Designator
      ;
```

D

```
Expr :: =   Expr Op Designator
      |     Designator
      ;

Op ::=      T_MINUS
      |     T_PLUS
      ;
```

Using the Right-Left rule write the C definition of a variable named `kashmir` that is a pointer to a function that takes one argument, an array where each element is a pointer to float, and returns a pointer to an array of 5 elements where each element is of type pointer to a pointer to a struct zeppelin. (10 pts)

Write a valid Reduced-C program to perform a simple Project I pointer and dynamic memory test. Define two global integer pointers named `ptr1` and `ptr2`. In main(), define a local integer variable named `x`. In main(), set `ptr1` to point to `x`. Dynamically allocate an integer appropriately such that `ptr2` points to this allocated memory. Then assign the integer value in `x` to the memory location pointed to by `ptr2` without directly using the variable name `x`. Be sure to free the memory you dynamically allocated before returning. (16 points)

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. Equivalence Error
B. Addressability Error
C. Assignability Error
D. No 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 : int main()
{
  float a;
  int b;
  float[5] c;
  int[5] d;
  float * e;

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

  foo1( a );                _____
  foo1( e );                _____
  foo1( b );                _____

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

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

  foo4( &a );               _____
  foo4( e );                _____
  foo4( (float *) &b );     _____

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

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 7. (2 pts each)

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

```
_____________________ a ;

_____________________          /* b requires two lines of Reduced-C */

_____________________ b ;      /* to properly define it */
```

```
    d = 5.5;

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

```
_____________________ a ;

_____________________ b ;

_____________________ c ;

_____________________ d ;
```

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

```
_____________________ a ;

_____________________ b ;

_____________________ c ;
```

---

4. Consider the following struct definitions. Specify the size of each struct on a typical RISC architecture (like ieng9) or -99 if it is an illegal definition. (6 pts)

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

Size _____

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

Size _____

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

Size _____

Using Reduced-C syntax from Project I, define a pointer to an array of 3 floats named `foo` such that `(*foo)[2]` is a valid 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)` ? _____

5. Given the following definitions: (4 pts)

```
int a;
float b;
```

Give an example of a converting type cast using only variables `a` and `b` defined above and any appropriate type cast(s).

Give an example of a non-converting type cast using only variables `a` and `b` defined above and any appropriate type cast(s) and any appropriate operators.

With regard to the following C/Reduced-C definition: (22 pts)

```
float x;
```

What type is `&x` ? _____

Is the expression `&x` addressable ? _____ /* In other words, can we say `&&x` ? */

What type is `*(long long *) &x` ? _____

Is the above expression addressable? _____ /* In other words, can we say `&*(long long *)&x` ? */

What type is `(struct fubar *) &x` ? _____

Is the above expression addressable? _____ /* In other words, ... you get the idea! */

What is the type of the expression `x + 5` ? _____

Is the above expression addressable? _____

If `&x` is 0x8000, what value does `&x + 3` represent? _____ /* Yes, this is in hexadecimal! */

Is the above expression addressable? _____

Is the expression `*(&x + 3)` addressable? _____

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. 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. (16 pts)

```
struct foo {
   char   a[5];
   double b;
   float  c;
   struct fubaz {
      double y;
      short  z;
   }      d;
   int    e[3];
};

struct foo fubar;
```

low memory

fubar:

high memory

What is the `offsetof( struct foo, e[1] )`? _____

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

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

If you rearranged the order of the struct members in struct foo to minimize padding, what would be the size of this modified struct? _____

How many bytes are saved by rearranging the struct members to minimize padding? _____

Does it matter whether you arrange the struct members in struct foo from largest to smallest data type or smallest to largest data type to minimize padding? _____

Extra Credit (8 points)

What gets printed by the following C program?

```
#include <stdio.h>

int
main()
{
  char a[] = "CSE131";
  char *p = a;

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

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

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

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

  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 -- 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