Student ID _____ **Name** _____

Login Name _____ **Signature** _____

Midterm
CSE 131
Winter 2014

Page 1	_____	**(21 points)**
Page 2	_____	**(36 points)**
Page 3	_____	**(28 points)**
Page 4	_____	**(16 points)**
Page 5	_____	**(18 points)**
Page 6	_____	**(20 points)**
Subtotal	_____	**(139 points = 100%)**
Page 7 Extra Credit	_____	**(7 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 Expr {: System.out.println("Z"); :}
     |     Des {: System.out.println("A"); :}
     ;

Des  ::=   T_STAR {: System.out.println("B"); :} Des {: System.out.println("C"); :}
     |     T_AMPERSAND {: System.out.println("D"); :} Des {: System.out.println("E"); :}
     |     T_PLUSPLUS {: System.out.println("F"); :} Des {: System.out.println("G"); :}
     |     Des2 {: System.out.println("H"); :}
     ;

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

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

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

What is the output when parsing the follow expression (you should have 18 lines/numbers in your output):

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

Output

____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____

In the above grammar, does the post-increment operator have left-to-right associativity or right-to-left associativity? _____

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

_____ `y;` _____ `x;`

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

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

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

Given the C following definitions (same rules in our Reduced-C compiler)

```
struct S1 { int a; };
struct S2 { int a; };

struct S1 a;
struct S2 b;
```

indicate whether each of the following statements will cause an equivalence compiler error or not.

A) Error
B) No Error

```
_____ a = b;                        _____ a.a = b.a;
_____ a = (struct S1)b;             _____ (struct S2)a = b;
_____ a = *(struct S1 *)b;          _____ *(struct S2 *)a = b;
_____ a = *(struct S1 *)&b;         _____ *(struct S2 *)&a = b;
```

Using Reduced-C syntax, define an array of an array of ints with dimensions 8x4 named bar such that `bar[7][3] = 42;` 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) R-val B) Modifiable L-val C) Non-Modifiable L-val

```
function : int * foo() { /* Function body not important. */ }
structdef R1 { int a; float b; };
float[9] a;
R1       b;
R1 *     c;
int *    d;
```

```
____ b.a++      ____ *d+1     ____ &b          ____ (int)a[3]        ____ c->a % b.a
____ foo()      ____ &a[2]    ____ (R1 *)foo() ____ a[1] = *foo()    ____ ++*d++
____ *foo()     ____ a[7]     ____ *&b         ____ a                ____ *d++
```

3. Given the following Reduced-C code and list of statements, indicate for each numbered statement the type of error that should be reported according to the Project I spec for this quarter (which is similar to C++ rules). Use the letters associated with the available errors in the box on the right, or choose A for No error.

```
int * [5] a;
const int b = 5;
bool c, d;
function : int foo(){ /* ... */ return 0; }
```

A) No error.
B) Operand to ++ is not a modifiable L-value.
C) Type of index expr in array not equivalent to int.
D) Left-hand operand is not assignable
(not a modifiable L-value).
E) Non-addressable argument to address-of operator.

```
____ 1) b = 3;

____ 2) &&a[0];

____ 3) a[b-2] = &b;

____ 4) (int)c = 4;

____ 5) &foo();

____ 6) *a[foo()] = b;

____ 7) a[2] = (int *) &c;

____ 8) ++b;

____ 9) (c = d) = true;

____ 10) c = d = true;

____ 11) (*a[0])++ = *a[1];

____ 12) *a[0]++ = *a[1];
```

State whether constant folding can be performed by the compiler according to this quarter's Reduced-C spec in the following Reduced-C statements

A) No constant folding
B) Constant folding

```
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;         ______

  b = d[e + a];             ______

  e = d[13 + b];            ______
}
```

Using the Right-Left rule write the C definition of a variable named foo that is a pointer to a 2-d array of 3 rows by 14 columns where each element is a pointer to a function that takes a pointer to a pointer to a long as a single parameter and returns a pointer to an array of 5 elements where each element is a pointer to a struct bar.

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 FOO1
{
  int    a;
  float  b;
  FOO1  *c;
  int[2] d;
};
```

Size _____

```
structdef FOO2
{
  int     a;
  float   b;
  int    *c;
  FOO2[2] d;
};
```

Size _____

```
structdef FOO3
{
  FOO3   a;
  float  b;
  int   *c;
  int[2] d;
};
```

Size _____

Identify whether each of the following will cause an underlying bit pattern change.

```
int a = 5;
float b = -4.20;
int * ptr1;
float * ptr2;
void foo( float x, float & y ) { /* … */ }
```

A) Yes – Underlying bit pattern change
B) No – No underlying bit pattern change

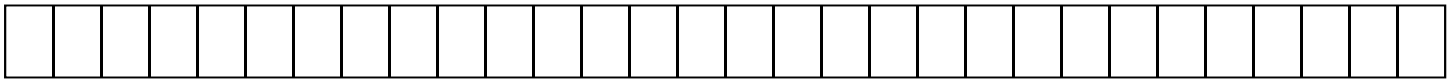
```
a = (int) b;                _____
b = a;                      _____
ptr1 = (int *) & b;         _____
foo( a, b );                _____
a = * (int *) & b;          _____
ptr2 = (float *) ptr1;      _____
```

Given the C array declaration

```
      C
int b[4][2];
```

Mark with a **B** all the memory location(s) where we would find the array element

```
      b[1][1]
b:
```



```
low memory                                                    high memory
```

Each box represents a byte in memory.

Given the following Reduced-C code below, fill in the blanks of the compile error that should be reported according to this quarter's Project I spec. Use the letters associated with the words in the box below.

```
typedef float F1;
typedef F1 F2;
typedef int I1;
typedef I1 I2;

I1 x;
I2 y;
F2 z;
```

A. I1	B. I2
C. int	D. F1
E. F2	F. float
G. 5-hour Energy	H. Sleep
I. equivalent	J. modifiable
K. addressable	L. assignable

```
x = z = y; // Compile error reported here. Assume this stmt is inside a function.
```

Value of type ____ not ____ to variable of type ____ .

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. 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;
  char   b[4];
  double c;
  short  d;
};

struct fubar {
  char       e[5];
  int *      f;
  struct foo g;
  char       h[3];
  char       i;
};

struct fubar fubaz;
```

low memory
fubaz:

high memory

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

What is the `sizeof( struct fubar )`? _____ What is the `offsetof( struct fubar, g.b[3] )`? _____

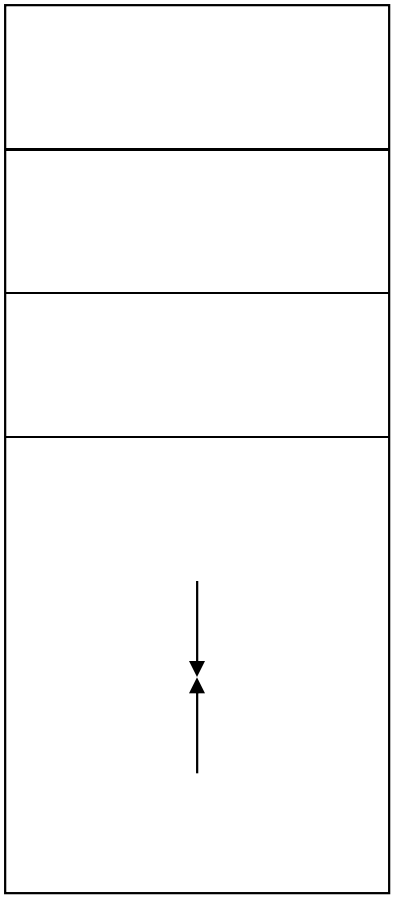
What is the resulting type of the following expression?

```
* (char *) (& ( ((struct foo *) fubaz.e )->d ) + 2) _________________
```

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

```
fubaz._________________
```

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 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)

_____ –> _____ –> _____ –> _____ –> _____ –> _____ –> _____

Extra Credit

What gets printed when the following C program is executed?

```
#include <stdio.h>

int
main()
{
  char a[] = "Rohan";
  char *ptr = a;

  printf( "%c\n", *(ptr++ + 2) + 1 );           _____

  printf( "%c\n", *(a+2) = *++ptr + 5 );        _____

  printf( "%c\n", *++ptr );                     _____

  printf( "%c\n", *++ptr + 9 );                 _____

  printf( "%c\n", *a = ptr[-sizeof(ptr)] + 1);  _____

  printf( "%d\n", --ptr - a );                  _____

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

  return 0;
}
```

A portion of the C Operator Precedence Table

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

Hexadecimal - Character

00	NUL	01	SOH	02	STX	03	ETX	04	EOT	05	ENQ	06	ACK	07	BEL
08	BS	09	HT	0A	NL	0B	VT	0C	NP	0D	CR	0E	SO	0F	SI
10	DLE	11	DC1	12	DC2	13	DC3	14	DC4	15	NAK	16	SYN	17	ETB
18	CAN	19	EM	1A	SUB	1B	ESC	1C	FS	1D	GS	1E	RS	1F	US
20	SP	21	!	22	"	23	#	24	$	25	%	26	&	27	’
28	(	29	)	2A	*	2B	+	2C	,	2D	-	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	‘	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{	7C	|	7D	}	7E	~	7F	DEL

Scratch Paper

Scratch Paper