Signature _____ **Name _____**

Login Name _____ **Student ID _____**

Midterm
CSE 131
Spring 2010

Page 1	**_____ (21 points)**
Page 2	**_____ (28 points)**
Page 3	**_____ (32 points)**
Page 4	**_____ (18 points)**
Page 5	**_____ (19 points)**
Page 6	**_____ (8 points)**
Subtotal	**_____(126 points = 100%)**
Page 7 Extra Credit	**_____ (9 points)**
Total	**_____**

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

```
Expr ::=   Des AssignOp Expr {: System.out.println("00"); :}
     |     Des {: System.out.println("0"); :}
     ;

Des  ::=   T_STAR {: System.out.println("1"); :} Des {: System.out.println("2"); :}
     |     T_PLUSPLUS {: System.out.println("3"); :} Des {: System.out.println("4"); :}
     |     T_AMPERSAND {: System.out.println("5"); :} Des {: System.out.println("6"); :}
     |     Des2 {: System.out.println("7"); :}
     ;

Des2 ::=   Des2 {: System.out.println("8"); :} T_PLUSPLUS {: System.out.println("9"); :}
     |     Des3 {: System.out.println("10"); :}
     ;

Des3 ::=   T_ID {: System.out.println("11"); :}
     ;

AssignOp ::=  T_ASSIGN {: System.out.println("12"); :}
     ;
```

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 postincrement operator have left-to-right associativity or right-to-left associativity? _____

If variable `z` is defined to be type `int`, 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

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

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

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

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

void foo ( struct S2 &b ) { }

struct S1 a;
```

a call to `foo( a )` passing in `a` as the actual argument will cause a compile error. Why?

Fix the function call `foo( a )` below to pass `a` to `foo()` without causing a C++ compile error.

```
foo( ____________________________________________ a );
```

Using Reduced-C syntax, define an array of an array of bools with dimensions 8x4 named bar such that `bar[7][2] = true;` 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. */ }
structdef R1 { int a; float b; };
float[9] a;
R1       b;
R1 *     c;
int *    d;
```

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

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

```
void foo1( int & a ) { ... }
void foo2( int a )   { ... }
int  foo3()          { ... }

int x;
float y;
int *ptr;
```

For each of the following function calls, 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 from the box to the right.

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

```
foo1( *&y );                ____
foo1( (int)&*ptr );         ____
foo1( *&ptr );              ____
foo1( 42 );                 ____
foo1( *(int *)&y );         ____
foo1( foo3() );             ____
foo1( ptr );                ____
foo1( *ptr );               ____
foo1( *ptr++ );             ____
foo1( *++ptr );             ____
foo1( ++*ptr );             ____
foo1( ++*ptr++ );           ____
foo1( *&x );                ____
```

```
foo2( ++*ptr++ );           ____
foo2( *&x );                ____
foo2( *&y );                ____
foo2( (int)&*ptr );         ____
foo2( *&ptr );              ____
foo2( 42 );                 ____
foo2( ptr );                ____
foo2( *ptr );               ____
foo2( *ptr++ );             ____
foo2( *++ptr );             ____
foo2( ++*ptr );             ____
```

Using the Right-Left rule write the C definition of a variable named fubaz that is a pointer to a 2-d array of 5 rows by 11 columns where each element is a pointer to a function that takes a pointer to a double as a single parameter and returns a pointer to a pointer to an array of 20 elements where each element is a pointer to a struct fubaz.

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

Size _____

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

Size _____

```
structdef FOO3 {
  FOO3  *a;
  float  b;
  function : void bar( FOO3 &x)
  {
    FOO3 *y;
  }
  int   *c;
  int[2] d;
};
```

Size _____

State whether constant folding can be performed by the compiler according to this quarter's Reduced-C spec in the following Reduced-C statements (**T** or **F**)

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

  const int c = a + 10;     _____

  int[53 + c] d;            _____

  d[-2 + (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;         _____
}
```

How did your group test the sizeof operator to make sure it was working correctly?

Using only the following C variable declarations:

```
int a;
float b;
int *c;
```

Give an example assignment stmt using a non-converting type cast (underlying bit pattern does not change).

Give an example assignment stmt using a converting type cast (underlying bit pattern changes).

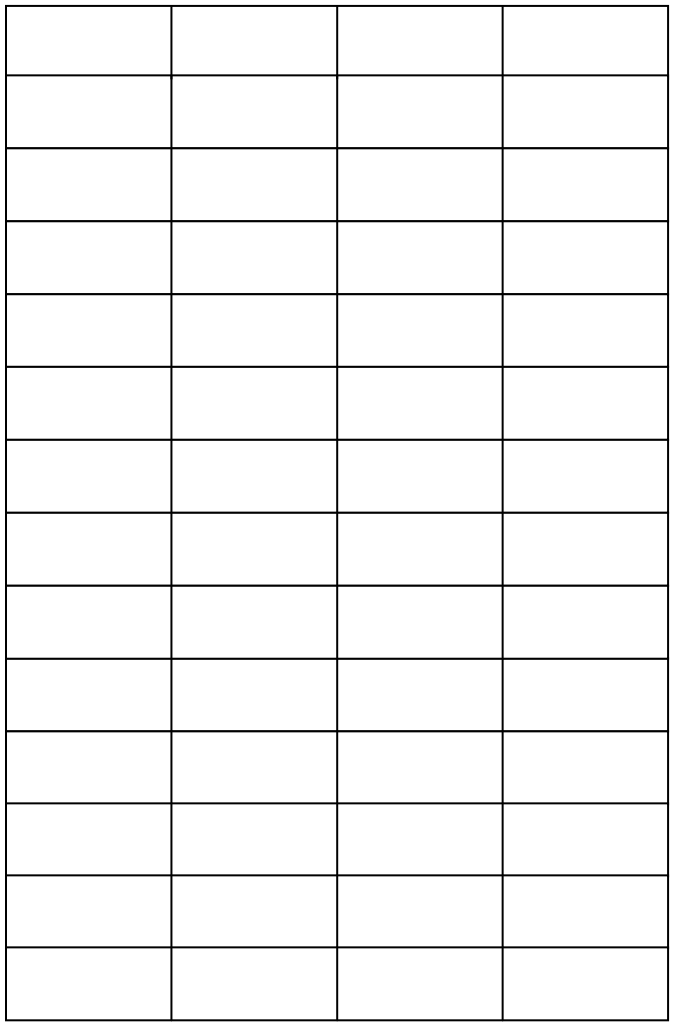
5. Show the memory layout of the following C struct definition taking into consideration the **SPARC** data type memory alignment restrictions discussed in class. Fill bytes in memory with the appropriate struct member/field name. For example, if member/field name `p` takes 4 bytes, you will have 4 `p`'s 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;
  short  b[2];
  double c;
  int    d;
};

struct fubar {
  int        e[2];
  short      f[2];
  char       g[11];
  struct foo h;
  short      i[2];
};

struct fubar fubaz;
```

low memory
fubaz:



high memory

What is the `sizeof( struct fubar )`? _____ What is the `offsetof( struct fubar, h.d )`? _____

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?

```
( (struct fubar *) ((& fubaz.h.c) + 1) ) -> f[1] _________________
```

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

```
fubaz._________________
```

6. Given the following Reduced-C structdef:

```
structdef MYSTRUCT {
  int x;

  function : int foo()
  {
    float x;

   /* Body of foo() */
  }

  function : void bar()
  {
   /* Body of bar() */
  }

};
```

Write a simple assignment statement that could occur in the body of foo() using both of the variables named x such that an error will not occur according to this quarter's spec. You must use both x variables in this simple assignment statement. No casts or other operators other than = and . (dot).

Write a simple assignment statement that could occur in the body of foo() using both of the variables named x such that an error will occur according to this quarter's spec. You must use both x variables in this simple assignment statement. No casts or other operators other than = and . (dot).

Write the appropriate error-free code in the body of bar() to call the function foo() and assign the return value of foo() to the first x variable [the x defined to be an int].

_____ analysis deals with verifying correct structure of a program.

_____ analysis deals with verifying correct meaning of a program.

What are the 3 ways in C/C++ to subvert the typing system as discussed in class.

1)

2)

3)

Extra Credit

What gets printed when the following C program is executed?

```
#include <stdio.h>

int
main()
{
  char a[] = "Foundations of Computer ";
  char b[] = "Science";
  char *p1 = a + 1;
  char *p2 = b;

  printf( "%c", *p2++ );    ______

  printf( "%c", p2[2] );    ______

  printf( "%c", *++p1 );    ______

  printf( "%c", *(p1 = p1 + 4) - 1 );    ______

  printf( "%c", p1[4] );    ______

  printf( "%c", *++p2 );    ______

  printf( "%c", *--p2 );    ______

  printf( "%c", *p2 - 2 );    ______

  printf( "%c", p1[-3] -2 );    ______

  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