Login Name _____ **Name** _____

Student ID _____ **Signature** _____

Final
CSE 131B
Spring 2007

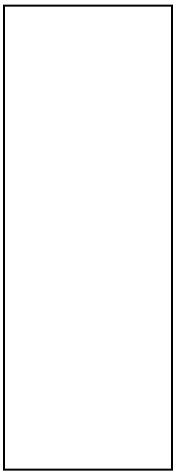
Page 1	_____	**(28 points)**
Page 2	_____	**(21 points)**
Page 3	_____	**(31 points)**
Page 4	_____	**(27 points)**
Page 5	_____	**(23 points)**
Page 6	_____	**(38 points)**
Page 7	_____	**(20 points)**
Page 8	_____	**(22 points)**
Subtotal	_____	**(210 points)**
Page 9 Extra Credit	_____	**(12 points)**
Total	_____	

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

```
Stmt ::=       Designator T_ASSIGN Expr T_SEMI
               {: System.out.println("5"); :}
     ;

Expr ::=       Expr {: System.out.println("4"); :} AddOp {: System.out.println("3"); :} Designator
     |         Designator {: System.out.println("2"); :}
     ;

Designator ::= T_ID {: System.out.println("1"); :}
     ;

AddOp ::=      T_MINUS {: System.out.println("-"); :}
     ;
```

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

```
x := y - z;
```



Given the following Oberon code, write the equivalent SPARC Assembly code that should be generated for the CIN and COUT statements according to this quarter's spec. Give code only for the CIN and COUT statements.

```
FUNCTION foo() : INTEGER;
  VAR f : FLOAT;
BEGIN

  CIN >> f;

  COUT << f;

  RETURN 0;
END foo;
```

/* Assume variable `f` is located at %fp – 4 */
/* Code for `COUT << f;` */

/* Code for `CIN >> f;` */

2. In object-oriented languages like Java, determining which overloaded method code to bind to (to execute) is done at run time rather than at compile time (this is known as dynamic dispatching or dynamic binding). However, the name mangled symbol denoting a particular method name is determined at compile time. Given the following Java class definitions, specify the output of each print() method invocation.

```
class Lois {
    public void print(Lois p) {
        System.out.println("Lois_1");
    }
}

class Peter extends Lois {
    public void print(Peter l) {
        System.out.println("Peter_1");
    }

    public void print(Lois p) {
        System.out.println("Peter_2");
    }

}
public class Overloading_Final_Exam {
    public static void main (String [] args) {
        Peter chris = new Peter();
        Lois stewie = new Peter();
        Lois meg = new Lois();

        chris.print(chris);                          _____________
        chris.print(stewie);                         _____________
        chris.print(meg);                            _____________

        ((Lois)chris).print(chris);                  _____________
        ((Lois)chris).print(stewie);                 _____________
        ((Lois)chris).print(meg);                    _____________

        meg.print(chris);                            _____________
        meg.print(stewie);                           _____________
        meg.print(meg);                              _____________

        ((Peter)stewie).print(chris);                _____________
        ((Peter)stewie).print(stewie);               _____________
        ((Peter)stewie).print(meg);                  _____________

        stewie.print(chris);                         _____________
        stewie.print(stewie);                        _____________
        stewie.print(meg);                           _____________
    }
}
```

Consider the following struct definitions. Specify the size of each struct on a typical RISC architecture (like ieng9) or -1 if it is an illegal definition.

```
struct foo {
  int a;
  double b;
  struct foo c;
  short d[8];
};
```

Size _____

```
struct foo {
  int a;
  double b;
  struct foo c[2];
  short d[8];
};
```

Size _____

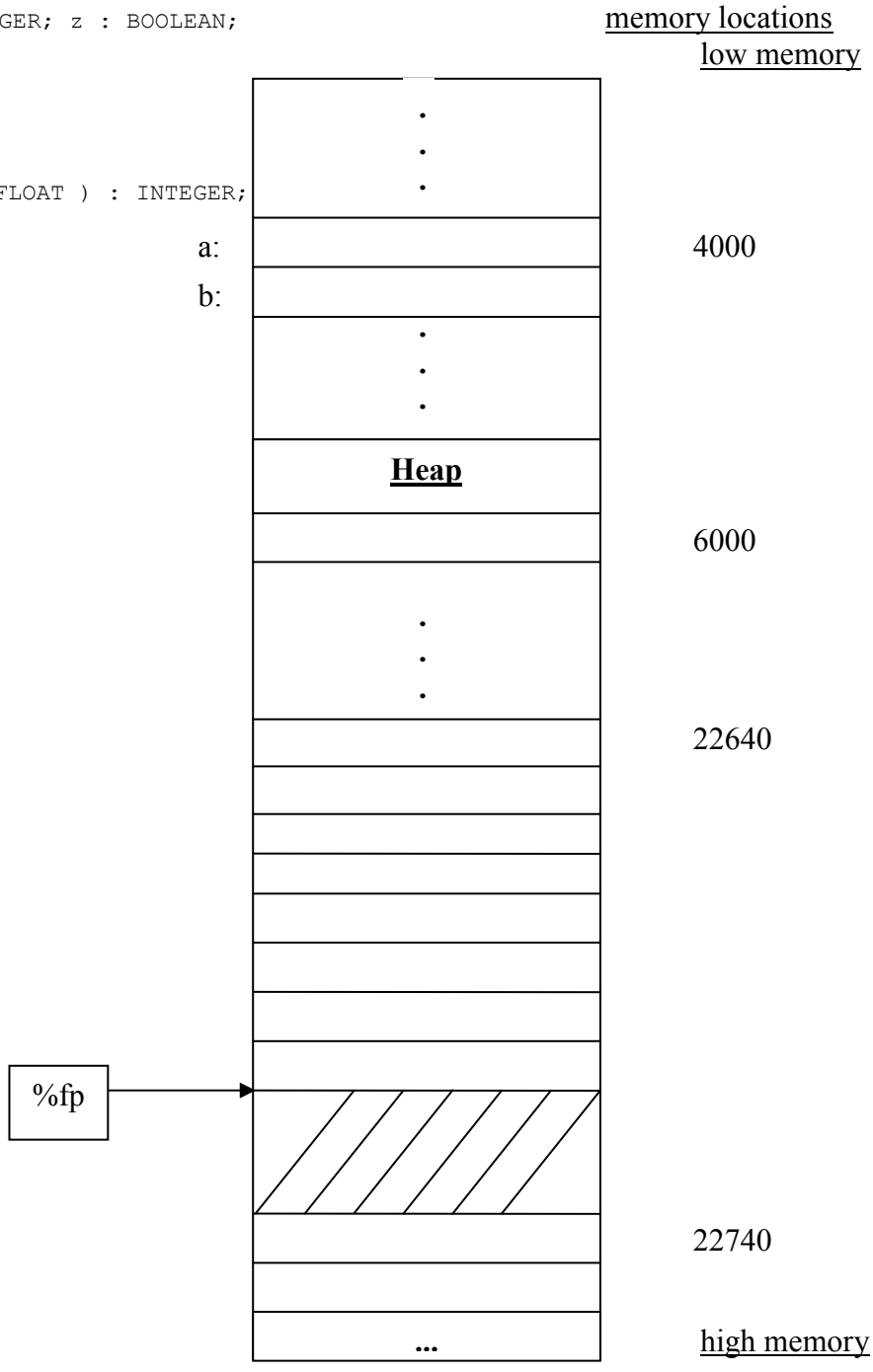
```
struct foo {
  int a;
  double b;
  struct foo *c;
  short d[8];
};
```

Size _____

3. In your Project 2, how did you (and your partner if you had a partner) handle local stack space allocation greater than 4K? Be specific how your project implemented this!

Given the following Oberon program and a real compiler's code gen as discussed in class, fill in the values of the global and local variables and parameters in the run time environment when the program reaches the comment `(* HERE *)`.

```
ALIAS rec1 = RECORD
              x : FLOAT; y : INTEGER; z : BOOLEAN;
             END;

VAR a : FLOAT;
VAR b : INTEGER;

FUNCTION f( i : INTEGER; REF z : FLOAT ) : INTEGER;
    VAR a2 : ARRAY 2 OF rec1;
    VAR  k : INTEGER;
    VAR  j : POINTER TO INTEGER;
BEGIN
    NEW(j);
    z := -55.5;
    a2[1].x := 18.25;
    a2[0].x := -9.5;
    a2[0].z := FALSE;
    a2[1].y := -95;
    j^ := 420;
    a2[1].z := TRUE;
    a2[0].y := i;
    k := -123;
    i := -79;

(*** HERE ***)
    FREE(j);
    RETURN 0;
END f;

BEGIN
    f( b, a );
END.
```



4. Given the following SPARC Assembly code, write the equivalent code in either C or Oberon. There is one parameter named `a1` and one local variable named `x1`. You should be able to determine their types based on the code. **No gotos!** **No if or if-else!** Use only the language constructs that were part of this quarter's project.

```
/* SPARC Assembly */

    .section ".text"

foo:

    save  %sp, -(92+4)&-8, %sp

    set   420, %l0
    st    %l0, [%fp - 4]

    ld    [%fp - 4], %l0
    ld    [%i0], %l1
    cmp   %l0, %l1
    bl    .L1
    nop

.L2:
    ld    [%i0], %l2
    add   %l2, 5, %l2
    st    %l2, [%i0]

    ld    [%fp - 4], %l0
    ld    [%i0], %l1
    cmp   %l0, %l1
    bge   .L2
    nop

.L1:
    ld    [%i0], %l3
    ld    [%fp - 4], %l4
    sub   %l3, %l4, %l5

    mov   %l5, %i0
    ret
    restore
```

(* Equivalent C or Oberon code *)

Using the Right-Left rule (which follows the operator precedence rules) write the definition of a variable named bar that is an array of 8 elements where each element is a pointer to a function that takes a pointer to a struct fubar as a single parameter and returns a pointer to an array of 3 elements where each element is a pointer to a pointer to a struct Baz. (9 points)

5. What is the output of the following Oberon program when executed?

```
VAR x, y : INTEGER;
VAR diff : INTEGER;

FUNCTION foo() : INTEGER;
 VAR r1, r2 : RECORD a, b, c, d, e : FLOAT; END;
BEGIN
 x := ${INTEGER} r1&;
 y := ${INTEGER} r2&;
 diff := x - y;
 IF diff < 0 THEN diff := -diff; END;
 COUT << "local: " << diff << NL;
 RETURN 0;
END foo;

BEGIN
 x := ${INTEGER} x&;
 y := ${INTEGER} y&;
 diff := x - y;
 IF diff < 0 THEN diff := -diff; END;
 COUT << "global: " << diff << NL;
 RETURN foo();
END.
```

Output

Given the following Oberon program:

```
ALIAS int = INTEGER;
ALIAS float = FLOAT;
ALIAS iptr = POINTER TO int;
ALIAS fptr = POINTER TO float;

VAR ip : iptr;
VAR fp : fptr;

BEGIN
 NEW(fp);
 CIN >> fp^;
 ip := ${iptr} fp;
 COUT << fp^ << " --> " << ip^ << NL;
 FREE(ip);
 RETURN 0;
END.
```

What is the line `ip := ${iptr} fp;` doing?

Is this a converting or a non-converting type cast?

What is the `COUT` line doing?

Is there a memory leak? Why or why not?

Fill in the blanks in the following C program fragment to match the output of the above Oberon program.

```
int main( void ) {
  float floatVar;

  /* Code to read a float into floatVar */

  printf( "%____ --> %____\n", ___________, ____________________________________ );

  return 0;
}
```

6. Consider the following Oberon program fragment.

```
VAR x : INTEGER;          (* assume x is located at %g6 - 4 - set  globals, %g6 *)

FUNCTION ptrFubar (a : POINTER TO INTEGER) : INTEGER;
       BEGIN
              ...
              RETURN 0;
       END ptrFubar;

FUNCTION fubar (a : INTEGER) : INTEGER;
       BEGIN
              ...
              RETURN 0;
       END fubar;

FUNCTION snafu (REF a : INTEGER) : INTEGER;
       BEGIN
              ...
              RETURN 0;
       END snafu;

FUNCTION foo (a : INTEGER; REF b : INTEGER) : INTEGER;
       VAR y : INTEGER;    (* assume y is located at %fp - 4 *)
       BEGIN
              (* XXXXX *)
              RETURN 0;
       END foo;
```

If you make different function calls to ptrFubar(), snafu(), and fubar() at the point in function foo() marked XXXXX, different SPARC assembly instructions are needed to correctly pass the different actual arguments to these functions. For example, if you were to call fubar(a); at the line marked XXXXX in foo(), you would need to generate the correct instruction(s) to pass the actual argument `a` to fubar() matching the parameter passing mode specified in the definition of fubar() taking into account the parameter passing mode specified in foo() for the formal parameter `a`.

The possible instructions you are allowed to use are:

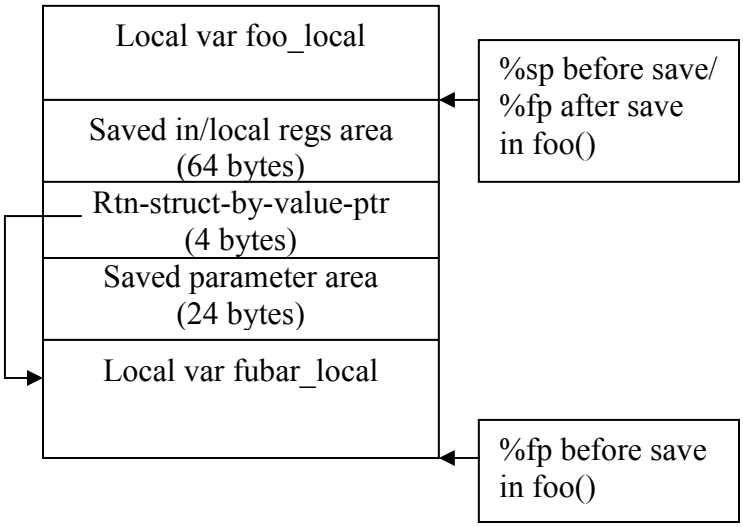
```
A) ld    [%fp + 68], %o0          I) st    %i0, [%fp - 4]
B) ld    [%fp + 72], %o0          J) st    %i0, [%g6 - 4]
C) ld    [%o0], %o0               K) st    %i1, [%fp - 4]
D) ld    [%o1], %o0               L) st    %i1, [%g6 - 4]
E) ld    [%g6 - 4], %o0           M) st    %i0, [%fp + 68]
F) add   %g6, -4, %o0             N) add   %fp, +68, %o0
G) ld    [%fp - 4], %o0           O) st    %i1, [%fp + 72]
H) add   %fp, -4, %o0             P) add   %fp, +72, %o0
```

Specify the letter(s) matching the instruction(s) needed to correctly pass the argument in the following function calls. If more than one instruction is needed, specify the letters in the correct order (for example, if the instruction tagged with the letter N should be executed first followed by the instruction tagged with the letter D, your answer should be ND). Note there are no optimized handling of mov's of the %i regs into %o regs.

```
ptrFubar(x&);  ______        fubar(x);  ______        snafu(x);  ______

ptrFubar(y&);  ______        fubar(y);  ______        snafu(y);  ______

ptrFubar(a&);  ______        fubar(a);  ______        snafu(a);  ______

ptrFubar(b&);  ______        fubar(b);  ______        snafu(b);  ______
```

7. Returning a struct/record by value in the SPARC calling convention can be implemented by setting up local stack space for the return value in the caller's stack space, setting a pointer to this stack space at %sp + 64 (4 bytes between the 64 bytes preallocated for saving the in and local registers and the 24 bytes preallocated for the parameters of the next function call), and having the callee use this pointer (available to the callee at %fp + 64 after the callee's save instruction) to copy (memcpy) the struct being returned from the callee's stack space into the caller's stack space. Let's do it! You may want to fill in fubar() code before foo().

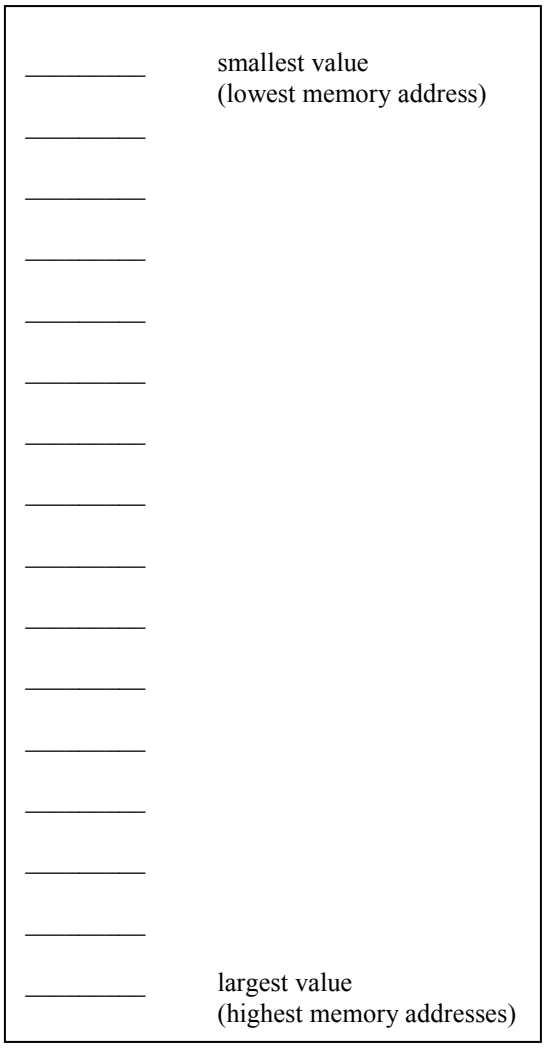
```
/*
   C Code -
   You can imagine something equivalent
   in Oberon.
*/

typedef struct {
  int a;
  int b[4];
  int c;
} rec1;

rec1 foo()
{
  rec1 foo_local;

/* Code manipulating the
   struct fields of foo_local */

  return foo_local;
}

int fubar()
{
  rec1 fubar_local;

  fubar_local = foo();

/* Code accessing the
   struct fields of fubar_local */

  return 0;
}
```




```
    /* Partial SPARC Assembly */

    .section ".text"
foo:
    ! Save space for local var foo_local

    save    %sp, -(92 + ______) & -8, %sp

    ! Code manipulating foo_local fields ...

    ! Set up return of foo_local by value
    ! using memcpy( to, from, n );

    ! Get the address of where to copy
    ! this local struct into the caller's
    ! stack space per above

    ld      _________________, %o0

    ! Calculate the address of foo_local
    ! struct in this stack frame

    add     _____________, _____________, %o1

    ! Specify # of bytes to copy

    set     _________, %o2

    ! copy foo_local struct into fubar_local

    call    ______________________
    nop

    ret
    restore

fubar:

    ! Save space for local var fubar_local

    save    %sp, -(92 + ______) & -8, %sp

    ! Place address of fubar_local in
    ! return-struct-by-value space per above

    add     ____________, ____________, %l0

    st      %l0, ____________________

    ! Call foo - no args - returns rec1

    call    foo
    nop

    ! Access returned struct in fubar_local

    mov     %g0, %i0
    ret
    restore
```

8. Given the following program, specify the order of the output lines when run and sorted by the address printed with the %p format specifier on a Sun SPARC Unix system. For example, which line will print the lowest memory address, then the next higher memory address, etc. up to the highest memory address?

```
#include <stdio.h>
#include <stdlib.h>

void foo1( int *, int ); /* Function Prototype */
void foo2( int, int * ); /* Function Prototype */

int main( int argc, char *argv[] ) {

   int a;
   int b = 420;

   foo2( a, &argc );

/* 1 */ (void) printf( "1: argv --> %p\n", &argv );
/* 2 */ (void) printf( "2: foo1 --> %p\n", foo1 );
/* 3 */ (void) printf( "3: b --> %p\n", &b );
/* 4 */ (void) printf( "4: argc --> %p\n", &argc );
/* 5 */ (void) printf( "5: a --> %p\n", &a );
}

void foo1( int *c, int d ) {

   int e = 404;
   static int f = 37;
   int g;

/* 6 */ (void) printf( "6: f --> %p\n", &f );
/* 7 */ (void) printf( "7: d --> %p\n", &d );
/* 8 */ (void) printf( "8: malloc --> %p\n", malloc(50) );
/* 9 */ (void) printf( "9: e --> %p\n", &e );
/* 10 */ (void) printf( "10: c --> %p\n", &c );
/* 11 */ (void) printf( "11: g --> %p\n", &g );

}

void foo2( int h, int *i ) {

   int j = 101;
   int k;
   static int l;

   foo1( &l, h );

/* 12 */ (void) printf( "12: l --> %p\n", &l );
/* 13 */ (void) printf( "13: k --> %p\n", &k );
/* 14 */ (void) printf( "14: i --> %p\n", &i );
/* 15 */ (void) printf( "15: h --> %p\n", &h );
/* 16 */ (void) printf( "16: j --> %p\n", &j );

}
```



How can you reduce the stack space required for a large number of local variables of different types on a typical RISC architecture with no optimization compiler flags turned on?

What is the name of the compiler optimization that would change a multiply by a power of 2 into an equivalent left shift?

Variables declared to be _____ will not be optimized by the compiler.

9. Extra Credit (12 points total extra credit)

What gets printed when this program is executed?

```
#include <stdio.h>

int main( void )
{
  char *a = "BuildOnACommit!";
  char *p = a + 4;

  printf( "%c\n", *&*p-- );           ______

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

  printf( "%c\n", *a++ );             ______

  printf( "%c\n", *(p = p + 3) + 1 ); ______

  printf( "%c\n", *(a+2) );           ______

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

  return 0;
}
```

Give the equivalent C expression for each of the following without using array access square brackets ([ ]) or the address of (&) operator.

```
a[0]       _______________

&a[2]      _______________

&a[0]      _______________

a[2]       _______________
```

Tell me something you learned in this class that is extremely valuable to you and that you think you will be able to use for the rest of your computer science career. (1 point if serious; you can add non-serious comments also)

Crossword Puzzle (next page) (1 point)

Scratch Paper

Scratch Paper