Login name ________________ **Name** ______________________

Signature __________________ **Student ID** __________________

Final
CSE 131B
Spring 2004

Page 1	___________	**(25 points)**
Page 2	___________	**(24 points)**
Page 3	___________	**(32 points)**
Page 4	___________	**(24 points)**
Page 5	___________	**(28 points)**
Page 6	___________	**(26 points)**
Page 7	___________	**(22 points)**
Page 8	___________	**(23 points)**
Subtotal	___________	**(204 points)**
Page 9 Extra Credit	___________	**(10 points)**
Total	___________	

1a. Consider the following C/C++ program:

```
#include <stdio.h>

int main()
{
  int y = 1;
  int x[5];
  int i;

  for ( i = 0; i <= 5; ++i )
    x[i] = i;

  printf( "%d\n", y );    /* Print the value of y */
  printf( "%p\n", &y );   /* Print the address of y */

  return 0;
}
```

Explain why the first printf() statement prints the value 5 when compiled with some compiler on some architecture (it does on SPARC with cc and CC)? (5 points)

Explain why the first printf() statement prints the value 1 when compiled with the same compiler on the same architecture as above but with the second printf() statement removed? (5 points)

1b. C/C++ structs are assignable only if they are equivalent types. Two structs that are not equivalent cannot be assigned even with a cast. Why? (5 points)

How can you get around this limitation? Give a specific example. (5 points)

Why is this OKAY? (5 points) [Hint: Discuss types of equivalence used in your answers]

2. The following C function is one way to calculate whether a year is a leap year. In particular, it uses several constructs that were part of Project 2 Code Generation: if-else conditional, modulus operator, boolean expressions containing equality checks, and logical AND and OR expressions with short-circuiting. Using your hard-earned code generation talents from Project 2, translate this into SPARC assembly. Nothing fancy, no optimizations, just perform a direct translation keeping in mind short-circuiting semantics. (20 points)

```
int
leapyear( unsigned int year )
{
  if ( (year % 400 == 0) || ((year % 4 == 0) && (year % 100 != 0)) )
    return 1;
  else
    return 0;
}
```

Given the array declaration

C	**Oberon-like**
`int a[3][2];`	`VAR a : ARRAY 3, 2 OF INTEGER`

Mark with an **A** the memory locations where we would find (4 points)

```
      a[2][0]                             a[2,0]
a:
```


```
low memory                                                                high memory
```

3. Why is passing and returning references or pointers to structs/objects as arguments and return values usually recommended over passing/returning structs/objects by value in most languages? (4 points)

What gets printed? (28 points)

```
VAR x : INTEGER;
VAR y : BOOLEAN;

PROCEDURE foo1( a : BOOLEAN; VAR b : INTEGER ) : INTEGER;
  VAR i : INTEGER;
  VAR j : BOOLEAN;
BEGIN
  i := b;
  b := 77;
  j := a;
  a := FALSE;

  OUTPUT x, " ", y;     ____________

  OUTPUT a, " ", b;     ____________

  OUTPUT i, " ", j;     ____________

  RETURN i;
END foo1;

PROCEDURE foo( a : INTEGER; VAR b : BOOLEAN );
  VAR i, j : INTEGER;
BEGIN
  i := a;
  a := 66;
  j := foo1( b, a );
  b := TRUE;

  OUTPUT x, " ", y;     ____________

  OUTPUT a, " ", b;     ____________

  OUTPUT i, " ", j;     ____________
END foo;

BEGIN
  x := 55;
  y := FALSE;

  foo( x, y );

  OUTPUT x, " ", y;     ____________
END.
```

4. Identify where each of the following program parts live in the Java runtime environment as discussed in class. (12 points)

```
public class Foo {
  private static Foo a;                        a           _________________

  private int b;                               b           _________________

  public Foo() {                               Foo()       _________________

    a = this;                                  this        _________________
    ++b;
  }
                                               main()      _________________

  public static void main( String[] args ) {   args        _________________

    Foo c = new Foo();                         c           _________________

    int d;                                     d           _________________

    c = new Foo();           where c is pointing           _________________
    c.method( d );
  }
                                               method()    _________________

  private void method( int e ) {               e           _________________

    int f;                                     f           _________________
    f = e;
  }
}
```

Using the Right-Left rule (which follows the operator precedence rules) write the definition of a variable named foo that is a pointer to an array of 9 elements where each element is a pointer to a function that takes a pointer to a struct Pub as the single parameter and returns a pointer to a 7x19 2-D array where each element is a pointer to a struct Fubar. (6 points)

Regarding type checking, reference (VAR) parameters require the actual arguments to be __________________

and ____________________ to the formal parameter type while value parameters require the actual arguments

to be ___________________ to the formal parameter type. (6 points)

5. Given the following program, order the printf() lines so that the values that are printed when run on a Sun SPARC Unix system are displayed from smallest value to largest value and label the corresponding C/C++ Runtime area. (20 points)

```
void foo( int, int * ); /* Function Prototype */

int a;

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

    static int b = 420;
    int c = 404;

    foo( argc, &c );

/*  1 */ (void) printf( "a --> %p\n", &a );
/*  2 */ (void) printf( "b --> %p\n", &b );
/*  3 */ (void) printf( "c --> %p\n", &c );
/*  4 */ (void) printf( "argc --> %p\n", &argc );
/*  5 */ (void) printf( "malloc --> %p\n", malloc(50) );
/*  6 */ (void) printf( "foo --> %p\n", foo );
}

void foo( int d, int *e ) {

    int f = 911;
    int g;

/*  7 */ (void) printf( "d --> %p\n", &d );
/*  8 */ (void) printf( "e --> %p\n", &e );
/*  9 */ (void) printf( "f --> %p\n", &f );
/* 10 */ (void) printf( "g --> %p\n", &g );
}
```

smallest value (low memory)	Runtime Area
___________	___________
___________	___________
___________	___________
___________	___________
___________	___________
___________	___________
___________	___________
___________	___________
___________	___________
___________	___________
largest value (high memory)	

Why do compilers typically allocate space for arguments in the Runtime Stack even when they pass them in registers? (4 points)

Object-oriented languages allow operator, function, and constructor overloading. In these languages, the function name is not always a unique identifier, since you can have multiple related definitions. For lookup purposes, the compiler must construct a distinct identifier for each function. Sometimes, such overloaded functions will have different return types as well. How would you create distinct identifiers for such functions? (4 points)

6. Given the following Oberon program, emit the **unoptimized** SPARC assembly language code that should be generated for the two procedures foo1() and foo(). Assume no optimizations – treat each instruction separately without any knowledge of any previously computed/loaded/stored values that may still be in a register from a previous instruction. Draw a line between each group of assembly language instructions that represent the emitted code generated for each instruction and label them with the instruction number. All local variables must be allocated and accessed on the Stack (do not map them directly into a local register). (26 points)

```
PROCEDURE foo1( VAR x : INTEGER; VAR y : INTEGER ) : INTEGER;
  VAR i, j : INTEGER;        (**** Local Stack variables - Do not initialize to zero ****)
BEGIN
  j := y + 5;           (* 1 *)
  i := x;               (* 2 *)
  y := j - 10;          (* 3 *)
  RETURN x;             (* 4 *)
END foo1;

PROCEDURE foo( a : INTEGER; VAR b : INTEGER );
  VAR i, j : INTEGER;        (**** Local Stack variables - Do not initialize to zero ****)
BEGIN
  i := foo1( a, b );    (* 5 *)
  b := foo1( i, j );    (* 6 *)  (* Don't worry about any local variables not initialized *)
END foo;

BEGIN
  (* ... *)
END.
        .global foo, foo1, main

        .section ".text"
foo1:                                              foo:
```

7. 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 `p0`'s, `p1`'s, `p2`'s, etc. Place an `X` in any bytes of padding. Structs are padded so the total size is evenly divisible by the most strict alignment requirement of its members. (11 points)

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

struct foo fubar;
```

`fubar:` `low memory`

Generate SPARC assembly code for the 3 labeled statements below. Assume the two local variables have been allocated on the Stack with an appropriate `save` instruction. Draw a line between and label the group of translated instructions for each labeled line. (11 points)

```
PROCEDURE P();
  VAR i : INTEGER;
  VAR ptr : POINTER TO INTEGER;
BEGIN
  NEW( ptr );            (* 1 *)

  ptr^ := 420;           (* 2 *)

  i := ptr^;             (* 3 *)
END P;

BEGIN
END.
```

8. Give the order of the phases of compilation in a typical compiler as discussed in class (8 points)

A – Machine-specific code improvement (optional)
B – Scanner (lexical analysis)
C – Parser (Semantic analysis/intermediate code gen.)
D – Parser (syntax analysis)
E – Machine-independent code improvement (optional)
F – Target code generation
G – Source language (for example, C)
H – Target language (for ex., assembly)

______ –> ______ –> ______ –> ______ –> ______ –> ______ –> ______ –> ______

Give the order of the typical C compilation stages and on to actual execution as discussed in class (8 points)

B – loader
G – cpp (C preprocessor)
D – ld (Linkage Editor)
E – as (assember)
C – exe/a.out (executable image)
H – Source file
F – ccomp (C compiler)
A – Program Execution

gcc ______ –> ______ –> ______ –> ______ –> ______ –> ______ –> ______ –> ______

How did you implement the OUTPUT function in your Project 2? (5000 points)

Tell me something you learned in this class that is extremely valuable and that you think you will be able to use for the rest of your programming/computer science career. (2 points)

9. Extra Credit (10 points)

Given the following ANSI/ISO C/C++ variable definitions, identify which expressions will produce a static semantic compiler error.

A) No compiler error
B) Compiler error

```
int
main()
{
  char * s = "CSE 131B Rocks!";

  char * p1 = &s[4];
  const char * p2 = s;
  char * const p3 = s + 7;
  const char * const p4 = &*(s + 9);

  p1 = s;                             ______

  *p1 = 'A';                          ______

  p2 = s;                             ______

  *p2 = 'A';                          ______

  p3 = s;                             ______

  *p3 = 'A';                          ______

  p4 = s;                             ______

  *p4 = 'A';                          ______

  *&p1[9] = *(s + 1);                 ______

  ((char *) p4) = (char *) p3;        ______

}
```

Note: cc, CC, and g++ report these as errors; gcc reports these as warnings! :-/

Scratch Paper

Scratch Paper