**Student ID** ____________________ **Name** ________________________

**Login Name** ____________________ **Signature** ________________________

# Final
# CSE 131
# Spring 2009

| | | |
|---|---|---|
| **Page 1** | ____________ | **(21 points)** |
| **Page 2** | ____________ | **(29 points)** |
| **Page 3** | ____________ | **(36 points)** |
| **Page 4** | ____________ | **(27 points)** |
| **Page 5** | ____________ | **(23 points)** |
| **Page 6** | ____________ | **(38 points)** |
| **Page 7** | ____________ | **(11 points)** |
| **Page 8** | ____________ | **(19 points)** |
| **Page 9** | ____________ | **(13 points)** |
| **Page 10** | ____________ | **(17 points)** |
| **Page 11** | ____________ | **(23 points)** |
| **Subtotal** | ____________ | **(257 points)** |
| **Page 12 Extra Credit** | ____________ | **(18 points)** |
| **Total** | ____________ | |

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

```
Stmt ::=   Des AssignOp Des T_SEMI {: System.out.println("1"); :}
     ;

Des  ::=   T_STAR {: System.out.println("2"); :} Des {: System.out.println("3"); :}
     |     T_PLUSPLUS {: System.out.println("4"); :} Des {: System.out.println("5"); :}
     |     T_AMPERSAND {: System.out.println("6"); :} Des {: System.out.println("7"); :}
     |     Des2 {: System.out.println("8"); :}
     ;

Des2 ::=   Des2 {: System.out.println("9"); :} T_PLUSPLUS {: System.out.println("10"); :}
     |     Des3 {: System.out.println("11"); :}
     ;

Des3 ::=   T_ID {: System.out.println("12"); :}
     ;

AssignOp ::=  T_ASSIGN {: System.out.println("13"); :}
     ;
```

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

<u>Output</u>

```
*++x = &*y++;
```

Does the above grammar agree with the C/C++ operator precedence? _________

Does the above grammar agree with the C/C++ operator associativity? _________

If variable `y` is defined to be type `int *`, what type must variable `x` be defined to be for this statement to be semantically correct? _____________

**2.** Given the following Reduced-C code fragment:

```
function : int foo( int & x, int * y, int z ) { /* Body of code not important for this question */ }

function : int main()
{
  int a;
  int b = -17;
  int c = b;

  a = foo( a, &b, c );

  return c;
}
```

Complete the SPARC Assembly language statements that might be emitted by a compliant Reduced-C compiler from this quarter for function main(). Allocate, store, and access all local variables on the Stack.

```
        .section _________

        .global _________
        .align 4

_______:
        set     _________________, %g1

        save    ________, %g1, ________

        /* Initialize the local variables */

        set     _________, %o0

        st      %o0, ______________                 ! int b = -17;

        ld      ______________, %o0

        st      %o0, ______________                 ! int c = b;

        /* Set up the 3 actual arguments to foo() */

        _______ ______________________________, %o0 ! large blank can be one or two operands

        _______ ______________________________, %o1

        _______ ______________________________, %o2

        call    foo                                 ! Call function foo()

        __________

        st      ___________, [%fp - 16]             ! Save return value into local temp1

        /* Copy saved return value stored in temp1 into local var a */

        _______ [%fp - 16], _____________________

        _______ __________________________________  ! a = foo( ... );

        /* return c; */

        ld      ______________, ______________

        ___________

        ___________

        MAIN_SAVE = -(92 + _______) ________ ________  ! Save space for 3 local vars + 1 temp
```

**3.** 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 Larry {
    public void print(Larry l) {
        System.out.println("Larry 1");
    }
}
```

```
class Curly extends Larry {
    public void print(Curly c) {
        System.out.println("Curly 1");
    }

    public void print(Larry l) {
        System.out.println("Curly 2");
    }
}
```

```
class Moe extends Curly {
    public void print(Moe m) {
        System.out.println("Moe 1");
    }

    public void print(Curly c) {
        System.out.println("Moe 2");
    }

    public void print(Larry l) {
        System.out.println("Moe 3");
    }
}
```

Now <u>remove</u> the entire `print(Larry l) {}` method in `class Curly` and <u>remove</u> the entire `print(Curly c) {}` method in `class Moe`. Specify the output of each print() method <u>with these changes</u> below.

```
public class Overloading_Final_Exam {
    public static void main (String [] args) {

        Larry  stooge1 = new Moe();
        Larry  stooge2 = new Larry();
        Larry  stooge3 = new Curly();
        Curly  stooge4 = new Moe();
        Curly  stooge5 = new Curly();
        Moe    stooge6 = new Moe();

        ( (Moe) stooge1).print( (Curly) stooge6 );

        ( (Larry) stooge2).print( (Moe) stooge1 );

        ( (Curly) stooge3).print( (Moe) stooge4 );

        ( (Moe) stooge4).print( (Larry) stooge5 );

        ( (Curly) stooge5).print( (Larry) stooge6 );

        ( (Moe) stooge6).print( (Moe) stooge4 );


        stooge1.print( (Curly) stooge6 );

        stooge2.print( (Moe) stooge1 );

        stooge3.print( (Moe) stooge4 );

        stooge4.print( (Larry) stooge5 );

        stooge5.print( (Larry) stooge6 );

        stooge6.print( (Moe) stooge4 );


        stooge1.print( stooge6 );

        stooge2.print( stooge1 );

        stooge3.print( stooge4 );

        stooge4.print( stooge5 );

        stooge5.print( stooge6 );

        stooge6.print( stooge4 );
    }
}
```

**4.** In your Project 2, how did you (and your partner if you had a partner) handle code gen for the address-of operator with an <u>Expression</u> that results in a modifiable l-val? For example, `&*ptr` or `&a[i]` or `&mystruct.a` . Note this question is <u>not</u> asking about handling the address-of operator with an identifier. Be specific how your project implemented this!

Using Reduced-C syntax, first define a struct S with members of type int, float, and pointer to struct S named a, b, and ptr, respectively. Then define a variable named `fubar` which is an array of an array (with dimensions 5x9) of pointers to struct S such that `fubar[4][1]->ptr = fubar[1][8];` is a valid expression. This will take more than one line of code.

For each of the following make no assumptions of what may be above or below each window of instructions unless otherwise stated. Use virtual register notation.

Change the following into three instructions that is an improvement over a single multiply instruction

```
r1 = r4 * 126            ______________

                         ______________

                         ______________
```

Optimize the following. Assume all registers except r2 are not needed (not alive) after the last statement. Note: Memory access (ld/st) is only between registers and memory.

```
r4 = x
r7 = x
r5 = r4
r4 = r7 - r5
r3 = 4                        /* x represents a memory location */
x = r3
r2 = r3 * 5
r7 = r2 + r4
x = r7
r2 = x
```

**5.** What gets printed in the following C++ program (just like Reduced-C without "function : " in front of each function definition)? If a value is unknown/undefined or otherwise cannot be determined by the code given, put a question mark ('?') for that output. Hint: Draw stack frames!

```
int a = 23;
int b = 34;
int c = 45;

void fubar( int * x, int & y, int z )
{
   ++*x;
   ++y;
   ++z;
}

void foo1( int & d, int e, int * f )
{
   ++d;
   ++e;
   ++*f;

   cout << a << endl;    ______
   cout << b << endl;    ______
   cout << c << endl;    ______
   cout << d << endl;    ______
   cout << e << endl;    ______
   cout << *f << endl;   ______
   fubar( &d, d, d );
   fubar( &e, e, e );
   fubar( f, *f, *f );
   cout << a << endl;    ______
   cout << b << endl;    ______
   cout << c << endl;    ______
   cout << d << endl;    ______
   cout << e << endl;    ______
   cout << *f << endl;   ______
}

int main()
{
   foo1( a, b, &c );
   cout << a << endl;    ______
   cout << b << endl;    ______
   cout << c << endl;    ______
   return 0;
}
```

Using the Right-Left rule write the C definition of a variable named fubar that is a pointer to a 2-d array of 5 rows by 8 columns where each element is a pointer to a function that takes a pointer to a pointer to a float as a single parameter and returns a pointer to an array of 11 elements where each element is a pointer to a struct fubaz.

**6.** Using the load/load/compute/store and internal static variable paradigms recommended in class and discussion sections, complete the SPARC Assembly language statements that might be emitted by a compliant Reduced-C compiler from this quarter for function foo(). Store all formal params on the Stack.

```
int foo( int *x, int y, int & z )
{
  static int c = z;

  *x = c - y;

  return z;
}
```

```
        .section      "__________"

        .global       __________
        .align        4
foo:
        set     foo.SAVE, %g1
        save    %sp, %g1, %sp

        st      %i0, _____________

        st      %i1, _____________

        st      ________, [%fp + 76]

        .section      "__________"
        .align        4
.foo_c:
        .skip         4
.foo_c_flag:
        .skip         4

        .section      "__________"

! Check if internal static var c has
! already been initialized

        set     ____________________, %o0
        ld      [%o0], %o0

        cmp     _________, _________

        ______  .L1           ! skip init
        nop

! Init internal static var c for 1st time

        ld      _____________, %o0

        ______  _____________, %o0

        st      %o0, [%fp - 4] ! tmp1 = z
        ld      [%fp - 4], %o0

        set     ____________________, %o1

! c = z
        ______  ________, __________

! set flag to skip all further inits
        set     ______, %o0

        set     .foo_c_flag, %o1

        st      _______, ___________
```

```
.L1:

! Perform *x = c - y; block

! c - y
        set     _____________, %o0

        ______  [%o0], %o0          ! c

        ld      _____________, %o1 ! y

        ______  %o0, %o1, %o0       ! c - y

! tmp2 <- (c - y)
        st      _________, [%fp - 8]

! previous result from tmp2
        ld      [%fp - 8], %o0

! get param x
        ld      _____________, %o1

! *x = c - y; (store tmp2 into *x)

        _______ %o0, ____________


! return z;
        ld      ____________, %o0

        ld      ____________, %o0

        ______  %o0, _________

        _________

        _________

! save space for 2 temporaries on stack

foo.SAVE = -(92 + _____) _____ ______
```

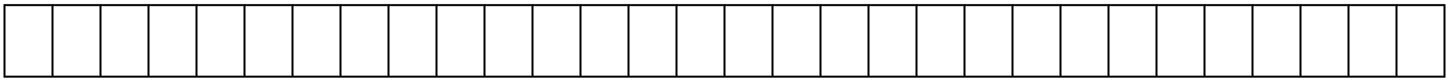
**7.** Given the C array declaration

**C**

```
int a[4][2];
```

Mark with an **A** the memory location(s) where we would find

```
     a[3][0]
a:
```



```
low memory                                                                    high memory
```

Each box represents a byte in memory.

Which of the following would be correct if we wanted to add the divide sign (/) as an operator with higher precedence than the current multiplication sign (*)? _____

**A**

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

Op ::=       T_SLASH
     |       T_STAR
     ;
```

**C**

```
Expr :: =    Expr T_STAR Expr1
     |       Expr1
     ;

Expr1 ::=    Expr1 T_SLASH Designator
     |       Designator
     ;
```

**B**

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

Op ::=       T_STAR
     |       T_SLASH
     ;
```

**D**

```
Expr :: =    Expr T_SLASH Expr1
     |       Expr1
     ;

Expr1 ::=    Expr1 T_STAR Designator
     |       Designator
     ;
```

What is the output of the following C++ program (similar to this quarter's Reduced-C spec)?

```
void foo( int x )
{
   static int y = x + 1;

   cout << y-- << endl;

   if ( x <= 2 && y >= 0 )
       foo( y + 1 );
}

int main()
{
  foo( 1 );
  return 0;
}
```

**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 and Linux 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 a = 42;

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

   int b;
   double c;

   foo2( a, &b );

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

void foo1( int *d, int e ) {

   struct foo {int a; int b;} f;
   int g;

/*  6 */ (void) printf( "6: f.b --> %p\n", &f.b );
/*  7 */ (void) printf( "7: d --> %p\n", &d );
/*  8 */ (void) printf( "8: e --> %p\n", &e );
/*  9 */ (void) printf( "9: f.a --> %p\n", &f.a );
/* 10 */ (void) printf( "10: foo2 --> %p\n", foo2 );
/* 11 */ (void) printf( "11: g --> %p\n", &g );
}

void foo2( int h, int *i ) {

   static int j[3];
   int k = 411;

   foo1( i, k );

/* 12 */ (void) printf( "12: j[1] --> %p\n", &j[1] );
/* 13 */ (void) printf( "13: h --> %p\n", &h );
/* 14 */ (void) printf( "14: a --> %p\n", &a );
/* 15 */ (void) printf( "15: i --> %p\n", &i );
/* 16 */ (void) printf( "16: j[0] --> %p\n", &j[0] );
/* 17 */ (void) printf( "17: k --> %p\n", &k );
}
```

_________ smallest value (lowest memory address)

_________

_________

_________

_________

_________

_________

_________

_________

_________

_________

_________

_________

_________

_________

_________

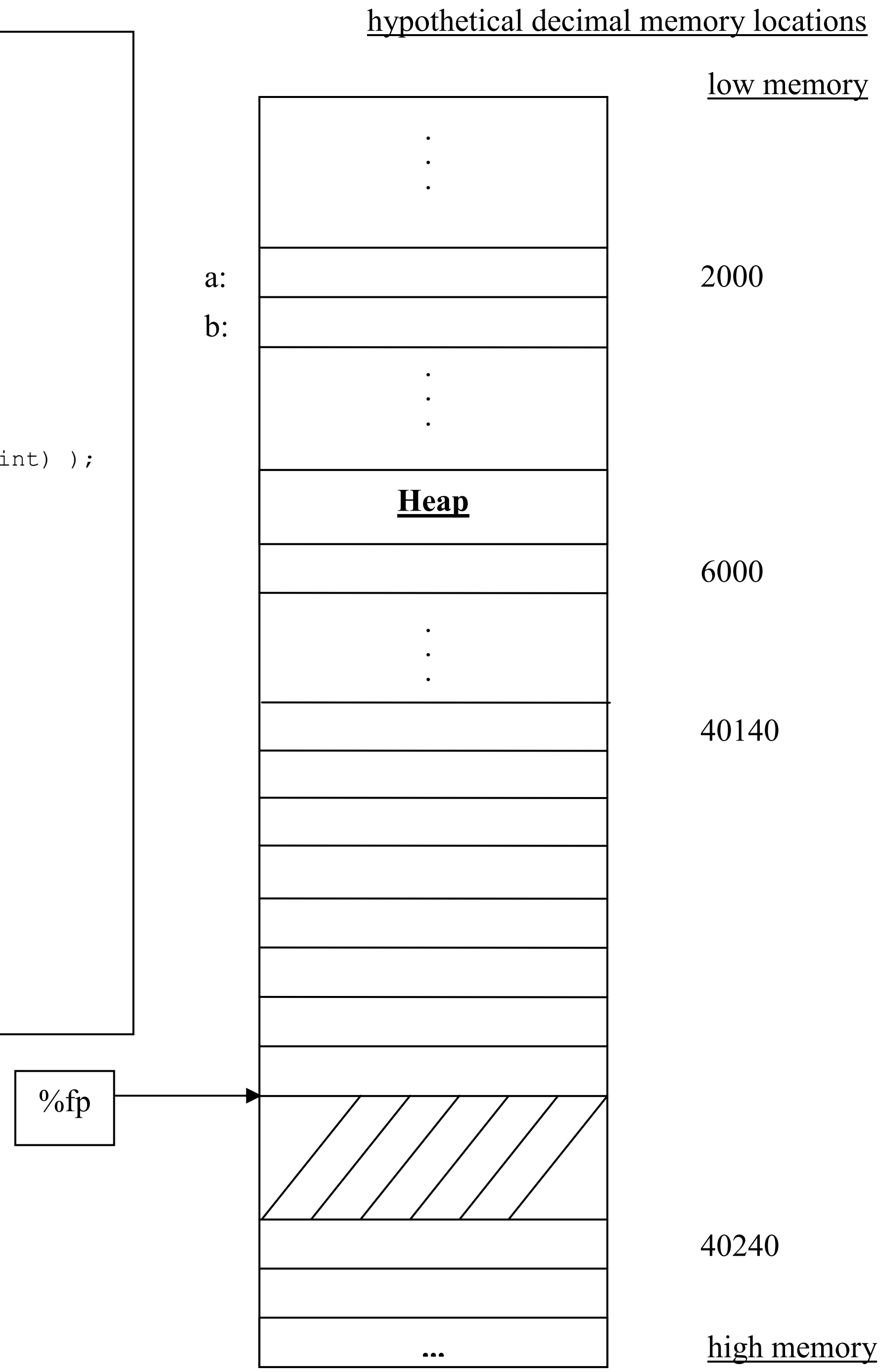
_________ largest value (highest memory addresses)

Who shot Mr. Burns? ____________________________________

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

**9.** Given the following C++ program (whose semantics in this case is similar to our Reduced-C) 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 for the SPARC architecture when the program reaches the comment `/* HERE */`. Do not add any unnecessary padding.

```
struct fubar {
  int   a;
  int * b;
  float c;
};

int   a;
float b;

void foo( float & f, int i ) {
  int   var1;
  int * var2;
  struct fubar var3[2];

  var1 = 123;
  var2 = (int *) calloc( 1, sizeof(int) );
  f = 98.6;
  var3[0].c = b;
  var3[1].a = i + 3;
  var3[1].b = &var3[1].a;
  i = -99;
  var3[0].a = a;
  var3[0].b = &i;
  var3[1].c = f;
  *var2 = var1 - 3;

  /* HERE */

  free( var2 );
}

int main() {
  foo( b, a );

  return 0;
}
```



**10.** Identify <u>where</u> each of the following program parts live in the Java runtime environment as discussed in class.

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

  private int b;                                    b            __________________

  public Foo() {                              code for Foo()     __________________

    a = this;                                       this         __________________
    ++b;
  }
                                              code for main()    __________________

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

    Foo c = new Foo();                              c            __________________

    int d = 5;                                      d            __________________

    c = new Foo();                 where c is pointing           __________________
    c.method( d );
  }
                                              code for method()  __________________

  private void method( int e ) {                    e            __________________

    int f;                                          f            __________________
    f = e;
  }
}
```

Given the following definitions

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

void foo ( struct S1 &a ) { }

struct S2 b;
```

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

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

```
foo( ___________________________________________ b );
```

**11.** Pick one of the following numbers to answer the questions below related to most calling conventions.

1) Prologue (in callee) 2) Epilogue (in callee) 3) Pre-Call (in caller) 4) Post-Return (in caller)

| | |
|---|---|
| _____ Allocates space for return value | _____ Restores caller-save registers |
| _____ Copies actual arguments into argument space | _____ Saves registers in callee-save scheme |
| _____ Allocates space for actual arguments | _____ Saves %pc into the return address location |
| _____ Stores return value into return value location | _____ Retrieves saved return address for return |
| _____ Allocates space for local variables & temps | _____ Performs initialization of local variables |
| _____ Saves registers in caller-save scheme | _____ Restores callee-save registers |
| _____ Retrieves return value from return value location | _____ Deallocates argument space |
| _____ Copies params passed in regs to param stack space | _____ Deallocates local variable & temps space |

Many experienced programmers prefer to use pre-increment/pre-decrement to perform a stand-alone inc/dec of a variable. For example, `++i;` or `for ( i = 0; i < SIZE; ++i )`

Why might a pre-increment/pre-decrement be preferred for these seasoned programmers? Think in terms of code gen from your compiler.

Given the following C type definitions

```
struct foo {               struct fubar {
  short  a;                  int        e;
  char   b;                  char       f[6];
  double c;                  struct foo g;
  int    d;                  int        h;
};                         };

struct fubar fubaz;
```

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

`* (int *) & ( ( (struct fubar *) & fubaz.g.c ) -> g )` _______________

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

`fubaz.`_______________

## 12. Extra Credit

What gets printed when this program is executed?

```
#include <stdio.h>

int
main()
{
  char a[] = "10019";
  char *ptr = a;

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

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

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

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

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

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

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

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

  return 0;
}
```

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)

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|
```

A portion of the Operator Precedence Table

```
Operator                Associativity

++ postfix increment    L to R
-- postfix decrement
[] array element
--------------------------------------
*  indirection          R to L
++ prefix increment
-- prefix decrement
&  address-of
--------------------------------------
*  multiplication       L to R
/  division
%  modulus
--------------------------------------
+  addition             L to R
-  subtraction
--------------------------------------
      .
      .
      .
--------------------------------------
=  assignment           R to L
```

**Scratch Paper**

## Scratch Paper