Thomas Nguyen
Professor Vu
COEN 171

Homework #6

Problem Set

7.9 Show the order of evaluation of the following expressions by parenthesizing all subexpressions and placing a superscript on the right parenthesis to indicate order.

a. a * b – 1 + c

$(((a * b)^1 - 1)^2 + c)^3$

b. a * (b – 1) / c mod d

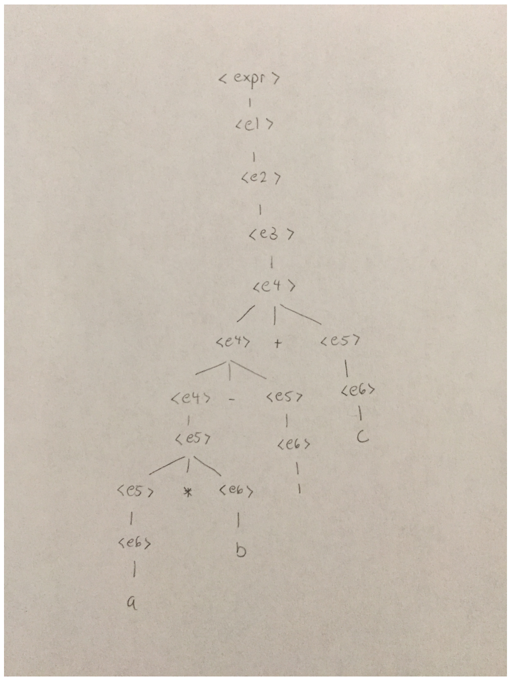
$(((a * (b - 1)^1)^2 / c)^3 \text{ mod } d)^4$

7.11 Write a BNF description of the precedence and associativity rules defined for the expressions in Problem 9. Assume the only operands are the names a,b,c,d, and e

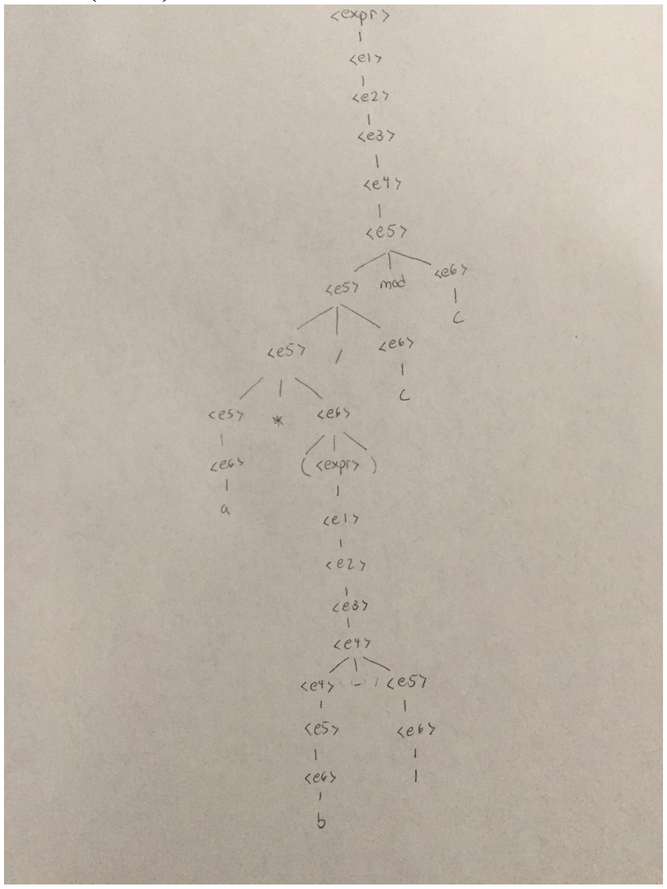
```
<expr> -> <expr> or <e1>
        | <expr> xor <e1>
        | <e1>
<e1> -> <e1> and <e2>
        | <e2>
<e2> -> <e2> = <e3>
        | <e2> /= <e3>
        | <e2> < <e3>
        | <e2> <= <e3> | <e2> > <e3> | <e2> >= <e3> | <e3>
<e3> -> <e4>
<e4> -> <e4> + <e5>
        | <e4> - <e5>
        | <e4> & <e5>
        | <e4> mod <e5>
        | <e5>
<e5> -> <e5> * <e6>
        | <e5> / <e6>
        | not <e5>
        | <e6>
<e6> -> a | b | c | d | e | const | ( <expr> )
```

7.12 Using the grammar of Problem 11, draw parse trees for the expression of Problem 9

a. a * b – 1 + c



b. a * (b – 1) / c mod d

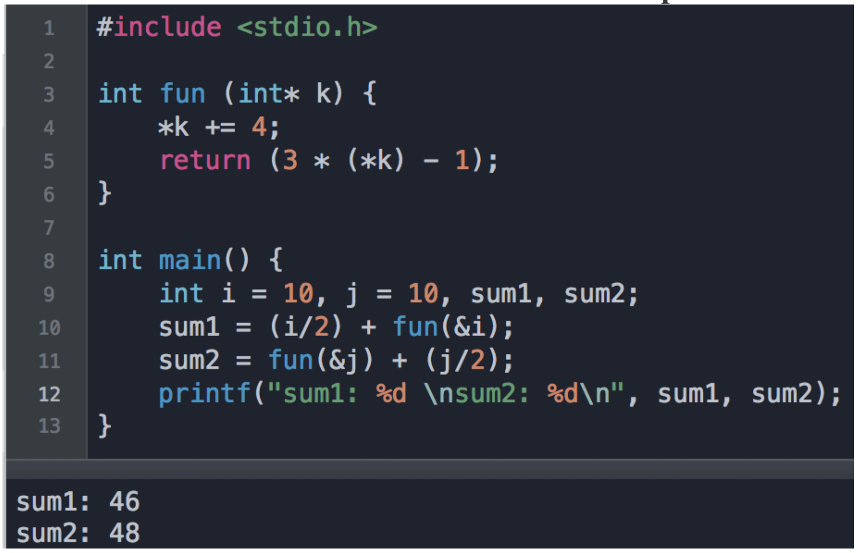


7.13 What are the values of sum1 and sum2
a. operands in the expressions are evaluated left to right?
Sum1 = 10/2 + (3 * 14 – 1) = 46
Sum2 = (3 * 14 – 1) + 14/2 = 48

b. operands in the expressions are evaluated right to left?
Sum1 = 14/2 + (3 * 14 – 1) = 48
Sum2 = (3 * 14 – 1) + 10/2 = 46

7.15 Explain why it is difficult to eliminate functional side effects in C
Because functions only return one data value. However, often times we might want to return more than one data value, and that is the purpose for passing by reference. The very nature of passing by reference will cause side effects, which is difficult to avoid.

<u>**Programming Exercises**</u>
7.1 Run the code given in Problem 13 (in the Problem Set) on some system that supports C to determine the values of sum1 and sum2. Explain the Results.



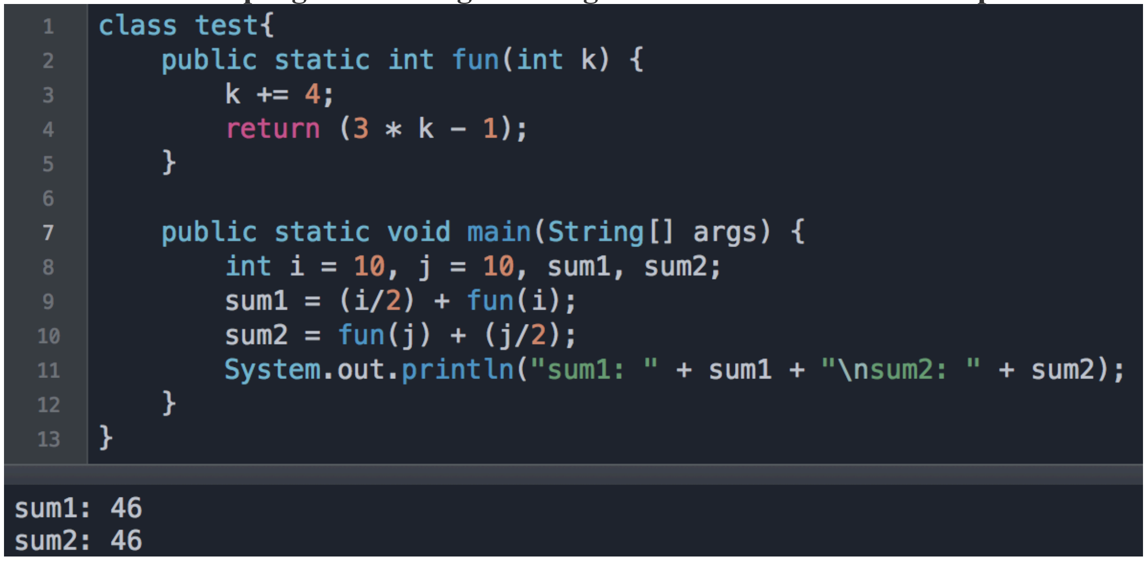
```c
#include <stdio.h>

int fun (int* k) {
    *k += 4;
    return (3 * (*k) - 1);
}

int main() {
    int i = 10, j = 10, sum1, sum2;
    sum1 = (i/2) + fun(&i);
    sum2 = fun(&j) + (j/2);
    printf("sum1: %d \nsum2: %d\n", sum1, sum2);
}
```

```
sum1: 46
sum2: 48
```

In C, the operands are evaluated from left to right. Therefore:
Sum1 = 10/2 + (3 * 14 – 1) = 46
Sum2 = (3 * 14 – 1) + 14/2 = 48

In sum2, we saw the side effects of evaluating fun(&j) before again using the value of j.

7.2 Rewrite the program of Programming Exercise 1 in Java and compare the results.



```
class test{
    public static int fun(int k) {
        k += 4;
        return (3 * k - 1);
    }

    public static void main(String[] args) {
        int i = 10, j = 10, sum1, sum2;
        sum1 = (i/2) + fun(i);
        sum2 = fun(j) + (j/2);
        System.out.println("sum1: " + sum1 + "\nsum2: " + sum2);
    }
}
```

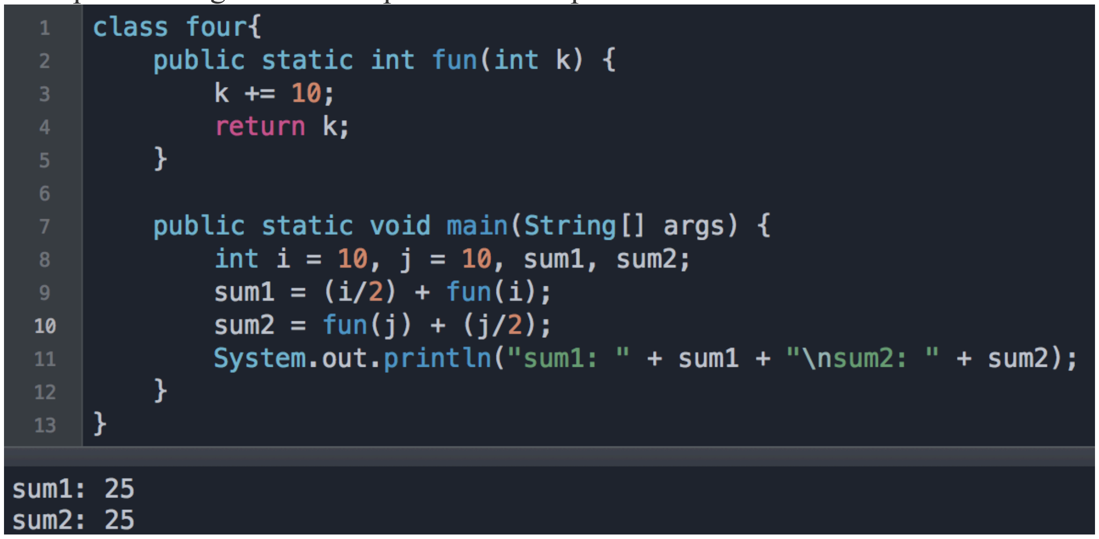
```
sum1: 46
sum2: 46
```

Java only allows passing by value and not by reference. Therefore:
Sum1 = 10/2 + (3 * 14 – 1) = 46
Sum2 = (3 * 14 – 1) + 10/2 = 46

In Java, the function “fun” will have no side effects.

7.4 Write a Java program that exposes Java’s rule for operand evaluation order when one of the operands is a method call.

I wasn’t entirely sure how this question was different from the previous, but I have written an example that might make the previous example more obvious.

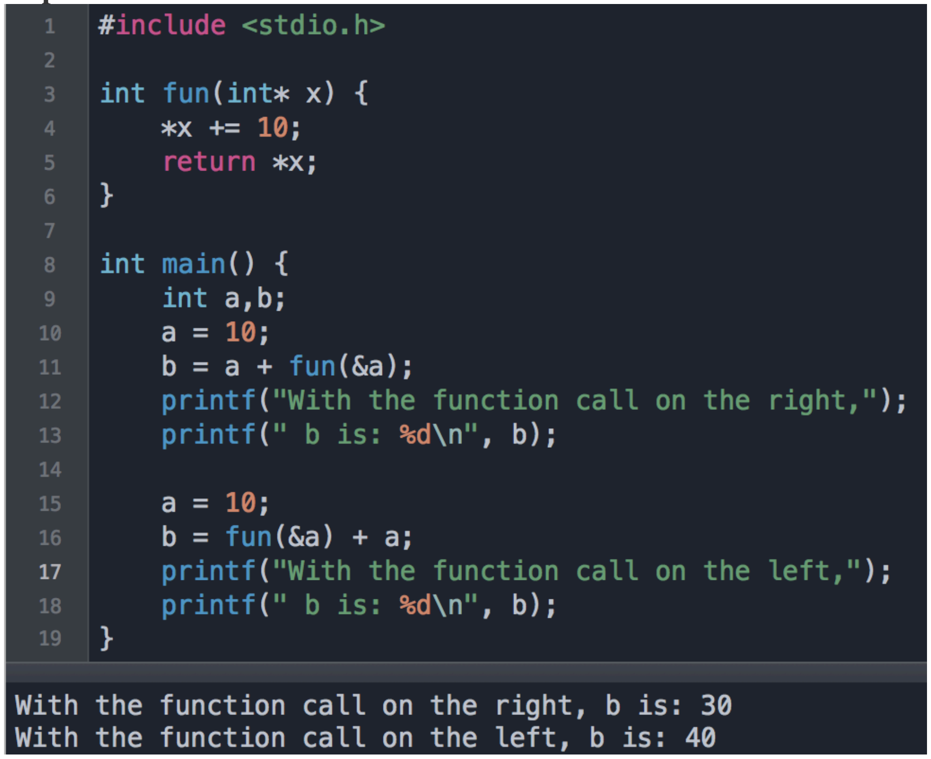


```
class four{
    public static int fun(int k) {
        k += 10;
        return k;
    }

    public static void main(String[] args) {
        int i = 10, j = 10, sum1, sum2;
        sum1 = (i/2) + fun(i);
        sum2 = fun(j) + (j/2);
        System.out.println("sum1: " + sum1 + "\nsum2: " + sum2);
    }
}
```

```
sum1: 25
sum2: 25
```

Again, Java does not allow pass by reference, so when a method is an operand, associativity rules still apply. In the above example:
Sum1 = (10/2) + (10 + 10) = 25
Sum2 = (10 + 10) + (10/2) = 25

7.8 Write a C program that has the following statements … and define fun to add 10 to a. Explain the results.



```c
#include <stdio.h>

int fun(int* x) {
    *x += 10;
    return *x;
}

int main() {
    int a,b;
    a = 10;
    b = a + fun(&a);
    printf("With the function call on the right,");
    printf(" b is: %d\n", b);

    a = 10;
    b = fun(&a) + a;
    printf("With the function call on the left,");
    printf(" b is: %d\n", b);
}
```

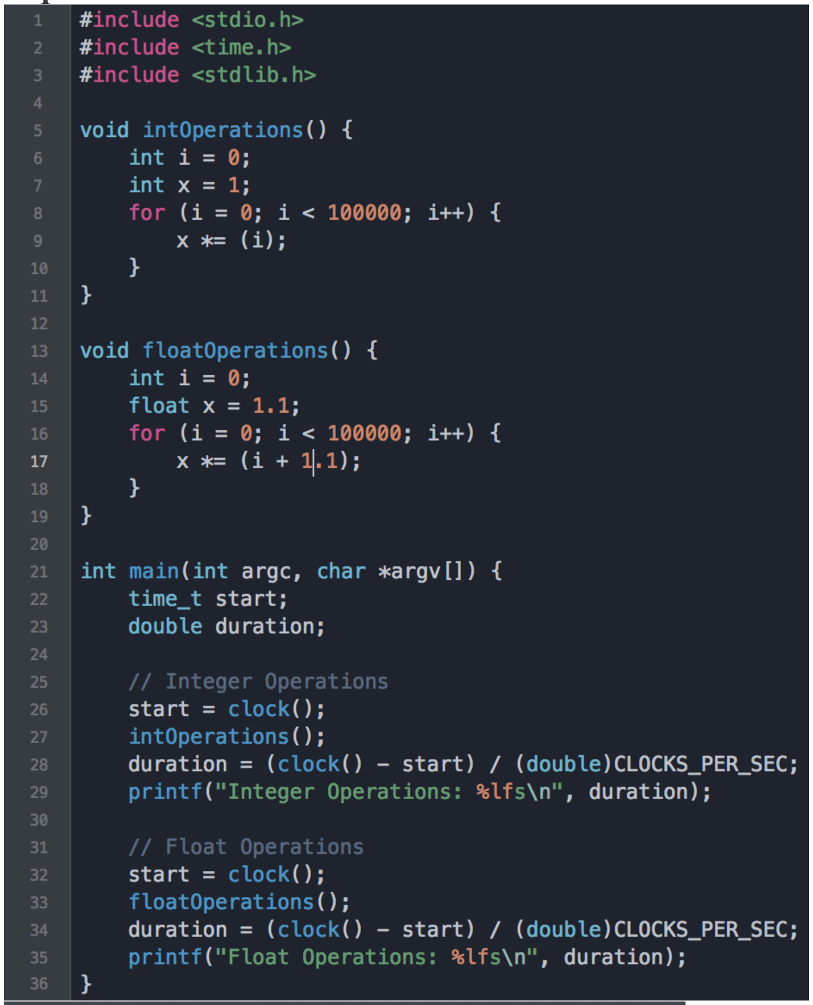
```
With the function call on the right, b is: 30
With the function call on the left, b is: 40
```

In C, the operands are evaluated from left to right. Therefore:

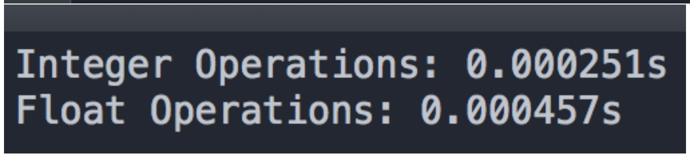
1) b = 10 + (10 + 10) = 30
2) b = (10 + 10) + (10 + 10) = 40

These are the side effects of passing by reference. All operations in parentheses shown in the explanation above are due to the function "fun".

7.9 Write a program in either Java, C++, or C# that performs a large number of floating-point operations and an equal number of integer operations and compare the time required.



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

void intOperations() {
    int i = 0;
    int x = 1;
    for (i = 0; i < 100000; i++) {
        x *= (i);
    }
}

void floatOperations() {
    int i = 0;
    float x = 1.1;
    for (i = 0; i < 100000; i++) {
        x *= (i + 1.1);
    }
}

int main(int argc, char *argv[]) {
    time_t start;
    double duration;

    // Integer Operations
    start = clock();
    intOperations();
    duration = (clock() - start) / (double)CLOCKS_PER_SEC;
    printf("Integer Operations: %lfs\n", duration);

    // Float Operations
    start = clock();
    floatOperations();
    duration = (clock() - start) / (double)CLOCKS_PER_SEC;
    printf("Float Operations: %lfs\n", duration);
}
```



```
Integer Operations: 0.000251s
Float Operations: 0.000457s
```