CMPSC 461: Programming Language Concepts, Fall 2025
Assignment 4

Prof. Suman Saha

Due: 11:59 PM, October 17, 2025

General Instructions:
You need to submit your homework to **Gradescope**. Do **every problem on a separate page and mark** them before submitting. **If the questions are not marked** or are submitted with incorrect page-to-question mapping, the question will be **deducted partial points**. Make sure your name and PSU ID are legible on the first page of your assignment.

You are required to submit your assignments in typed format, please follow the latex/doc template (which can be found on Canvas) for the homework submission. Furthermore, please note that no handwritten submissions (of any form on paper or digital) will be accepted.

**(Kindly refer to the syllabus for late submission and academic integration policies.)

Assignment Specific Instructions:

1. The sample examples provided in the questions below are just for your reference and do not cover every possible scenario your solution should cover. Therefore, it is advised that you think through all the corner cases before finalizing your answer.
2. Students are expected to answer the questions in a way that shows their understanding of the concepts rather than just mentioning the answers. The rubric does contain partial points to encourage brief conceptual explanations.

Problem 1: Parameter Passing Exercise [20 pts]

Consider the following pseudocode:

```
x = 3;  y = 2;  z = 5;

function update(a, b) {
   a = a + y;
   y = b + x;
   b = a - z;
   x = b + y + a;
   z = x + a - b;
}
```

1. **[16 pts]** For each of the cases below, write down the values of x, y, and z after the following calls to update(), For each line in the function also write down what is the value of the LHS. If necessary, assume that output arguments are copied back to parameters in the left-to-right order.

   (a) update(x,y) where all parameters are passed by value.

   (b) update(x,z) where all parameters are passed by reference.

   (c) update(y,z) where all parameters are passed by value-result.

   (d) update(y,x) where all parameters are passed by reference.

2. **[4 pts]** Passed-by-result parameter passing scheme is not applicable for the above program – justify this statement.

Solution

1. (a) 10, 5, 15
   a = 3, b = 2 [values of x, y are copied to a, b respectively]
   a = a + y = 3 + 2 = 5
   y = b + x = 2 + 3 = 5
   b = a - z = 5 - 5 = 0
   x = b + y + a = 0 + 5 + 5 = 10
   z = x + a - b = 10 + 5 - 0 = 15

   (b) 15, 10, 30
   Here, a and b are aliases of x and z respectively.
   a = a + y ⇒ x = 3 + 2 = 5
   y = b + x ⇒ y = 5 + 5 = 10
   b = a - z ⇒ z = 5 - 5 = 0
   x = b + y + a ⇒ x = 0 + 10 + 5 = 15
   z = x + a - b ⇒ z = 15 + 15 - 0 = 30

(c) 11, 4, -1
a = 2, b = 5 [values of y, z are copied to a, b respectively]
a = a + y = 2 + 2 = 4
y = b + x = 5 + 3 = 8
b = a - z = 4 - 5 = -1
x = b + y + a = -1 + 8 + 4 = 11
z = x + a - b = 11 + 4 - (-1) = 16
Finally, values copied back (value-result):
y = a = 4, z = b = -1

(d) 13, 6, 6
Here, a and b are aliases of y and x respectively.
a = a + y ⇒ y = 2 + 2 = 4
y = b + x ⇒ y = 3 + 3 = 6
b = a - z ⇒ x = 6 - 5 = 1
x = b + y + a ⇒ x = 1 + 6 + 6 = 13
z = x + a - b ⇒ z = 13 + 6 - 13 = 6

2. The statement is correct. In the pass-by-result scheme, parameters are expected to be initialized by the callee before any use. However, in this program, the initial values of a and b are used by the callee before any modifications occur, which makes pass-by-result inapplicable.

Problem 2: Parameter Passing II [4 + 4 = 8 pts]

Like Call by Name parameter-passing mechanism, *Call by Macro Expansion* technique also uses a lazy evaluation technique. It is widely used in C, C++, etc. Unlike function calls, macros do not have runtime overhead because they are expanded at compile-time (before actual code execution). Macro expansion works in the following way:

- **No evaluation:** The literal text of each macro argument is substituted for the corresponding parameter in the macro's body.
- **No evaluation:** The resulting macro body is then textually inserted into the program where the macro was called.
- **Evaluation:** The expanded macro code is executed in the caller's environment, meaning the variables and expressions are evaluated as if they were part of the original calling code.

Below is a C code with Macros:

```
#define add(a,b) a + b
#define square(x) x * x
int main() {
    int x = 2;
    printf("%d\n", square(add(x,1)));
    }
```

1. **[4 pts]** Show the fully expanded macro. What output does it produce and why?

2. **[4 pts]** How would you fix the issue, Note you cannot change the parameter passing technique.

Note: Check related exercises from this link to learn more about macro expansion (collected from the optional reading material section of Module Week 6).

Solution

1. the preprocessor performs textual substitution before compilation. Substituting macros step by step:

   $$square(add(x, 1)) \Rightarrow add(x, 1) * add(x, 1)$$

   But because there are no parentheses around the macro parameters, the expansion actually becomes:

   $$(x + 1 * x + 1)$$

   According to C operator precedence, multiplication has higher priority than addition. So the expression evaluates as:

   $$2 + (1 * 2) + 1 = 5$$

   Therefore, the output printed is: 5

2. To fix the issue, parentheses should be placed around both the parameters and the entire macro body:

```
#define add(a,b) ((a) + (b))
#define square(x) ((x) * (x))
```

Problem 3: Recursion [4 + 4 = 8 pts]

Consider the following two recursive function implementations:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

```
def power_tail(base, exp, acc = 1):
    if exp == 0:
        return acc
    else:
        return power_tail(base, exp - 1, acc * base)
```

1. [**4 pts**] Rewrite the **factorial** function in a **tail-recursive** form. You may use an accumulator or a helper function.

2. [**4 pts**] Rewrite the **power_tail** function in a **non-tail-recursive** form.

Solution

1. Tail-recursive version of **factorial**:

```
def factorial_tail(n, acc = 1):
    if n == 0:
        return acc
    else:
        return factorial_tail(n - 1, acc * n)
```

2. Non-tail-recursive version of **power_tail**:

```
def power(base, exp):
    if exp == 0:
        return 1
    else:
        return base * power(base, exp - 1)
```

Problem 4: Exception Handling [4 + 10 = 14 pts]

Consider the following Java program that demonstrates exception propagation across multiple methods. The execution begins with `main`:

```
public class SimpleExceptionFlow {

    static class CustomException extends Exception {
        public CustomException(String msg) {
            super(msg);
        }
    }

    public static void main(String[] args) {
        try {
            A();
            System.out.println("End of main");
        } catch (CustomException e) {
            System.out.println("Catch in main: " + e.getMessage());
        }
    }

    public static void A() throws CustomException {
        try {
            B();
        } catch (CustomException e) {
            System.out.println("Caught in A: " + e.getMessage());
            throw new CustomException("From A");
        } finally {
            System.out.println("Finally in A");
        }
    }

    public static void B() throws CustomException {
        try {
            C();
        } catch (CustomException e) {
            System.out.println("Caught in B: " + e.getMessage());
            throw new CustomException("From B");
        }
    }

    public static void C() throws CustomException {
        throw new CustomException("Thrown in C");
    }
}
```

1. **[4 pts]** Write the exact output produced by the above program.
2. **[10 pts]** Modify the program so that:
   (a) The `main` method first calls `B()` (which calls `C()` and propagates its exception),

(b) Then main calls `A()`, which directly throws an exception, catches it and throws to calling function.The final output should appear exactly as follows:

```
Caught in B: Thrown in C
Catch in main: From B
Catch in A: Direct exception from A
Finally in A
Catch in main: From A
```

Solution

1. Output of the original program:

```
Caught in B: Thrown in C
Caught in A: From B
Finally in A
Catch in main: From A
```

2. One possible modified version:

```
public class SimpleExceptionFlow {

    static class CustomException extends Exception {
        public CustomException(String msg) {
            super(msg);
        }
    }

    public static void main(String[] args) {
        try {
            B();
        } catch (CustomException e) {
            System.out.println("Catch in main: " + e.getMessage());
        }

        try {
            A();
        } catch (CustomException e) {
            System.out.println("Catch in main: " + e.getMessage());
        }
    }

    public static void A() throws CustomException {
        try {
            throw new CustomException("Direct exception from A");
        } catch (CustomException e) {
            System.out.println("Catch in A: " + e.getMessage());
            throw new CustomException("From A");
        } finally {
            System.out.println("Finally in A");
        }
    }
```

```
    public static void B() throws CustomException {
        try {
            C();
        } catch (CustomException e) {
            System.out.println("Caught in B: " + e.getMessage());
            throw new CustomException("From B");
        }
    }

    public static void C() throws CustomException {
        throw new CustomException("Thrown in C");
    }
}
```