CMPSC 461: Programming Language Concepts, Fall 2025
Assignment 3

Prof. Suman Saha

Due: 11:59 PM, September 26, 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: Storage, Scope, and Lifetime [$4 + 4 + 2 = 10$ pts]

Consider the following C++-like pseudocode:

```
const int A = 100;

void process(int* B) {
    int C = *B + 5;
    *B = C * 2;
}

int main() {
    static int D = 0;
    int E = 20;
    int* F = new int(50);

    process(F);

    if (E > 10) {
        int G = 5;
        D += G;
    }

    delete F;
    return 0;
}
```

(A) Storage Allocation [4 marks]. For each item below, classify its storage allocation mechanism using the slide terminology: **static object**, **object on stack**, or **object on heap**.

- `A`
- `B`
- `C`
- `D`
- `E`
- `F`
- The integer object created by `new int(50)`
- `G`

(B) Scope and Lifetime [4 marks]. For each of the same items, describe its **scope** and **lifetime**.

(C) Scope vs Lifetime of `D` [2 marks]. Comment specifically on the scope and lifetime of `D`. how do they differ?

Problem 2: Static Scoping and Symbol Tables [$4 + 3 + 3 = 10$ pts]

Consider the following pseudo-code, which allows nested subroutines.

```
int x = 1;
int y = 2;

void D(int z) {
    int y = z * 2;
    print(y + x);
}

void A() {
    int x = 10;

    void B(int z) {
        int y = z + x;
        print(y);

        void C() {
            int x = y + 5;
            D(x);
        }

        C();
        print(x);
    }

    B(5);
    print(x);
}

void main() {
    A();
    print(y);
}
```

(A) **(4 pts):** Draw the hierarchical symbol tables for all relevant scopes, assuming the language uses static scoping.

(B) **(3 pts):** What is the program's output under static scoping?

(C) **(3 pts):** For each function (`A`, `B`, `C`, `D`, and `main`), explain how every use of variables `x` and `y` is resolved under static scoping. Refer to your symbol table hierarchy from Part A to describe the search process.

Problem 3: Nested Scopes, Recursion, and Links [$4 + 4 + 4 + 3 = 15$ pts]

Consider the following pseudo-code, which allows nested subroutines and uses recursion.

```
int x = 100;

void helper() {
    print(x);
}

void outer() {
    int x = 50;

    void recur(int n) {
        if (n > 0) {
            int x = n;
            recur(n - 1);
        } else {
            helper(); // Base case of recursion
        }
    }

    recur(2);
    print(x);
}

void main() {
    outer();
    print(x);
}
```

(A) **(4 pts)** What does the program print if the language uses **static scoping**? Provide a step-by-step trace of the output.

(B) **(4 pts)** What does the program print if the language uses **dynamic scoping**?

(C) **(4 pts)** Draw a diagram of the **runtime stack** at the exact moment the `helper()` function is called. For each frame on the stack, show the function name and its static and dynamic links.

(D) **(3 pts)** Referring to your stack diagram, briefly explain how the `helper()` function resolves the binding for variable `x` under both **static** and **dynamic** scoping rules.

Problem 4: Deep vs. Shallow Binding and Closures [$5 + 5 + 5 = 15$ pts]

Consider the following pseudo-code, which returns a function reference from a subroutine. Assume the language uses **dynamic scoping**.

```

typedef void (*FuncPtr)();

void worker() {
    print(val);
}


FuncPtr setup() {
    int val = 50;
    return worker;
}

void recursive_executor(int n, FuncPtr F) {
    if (n > 0) {
        int val = n * 10;
        recursive_executor(n - 1, F);
    } else {
        F();
    }
}

void main() {
    FuncPtr my_func;
    my_func = setup();
    recursive_executor(2, my_func);
}
```

- **(A)** **(5 pts):** What is the output if the language uses **shallow binding**? Explain your answer by tracing the variable resolution from the call site of the function.
- **(B)** **(5 pts):** What is the output if the language uses **deep binding**? Explain what referencing environment is captured when the function reference is created and how it is used.
- **(C)** **(5 pts):** Based on this program, what is a **closure**? Which binding rule (deep or shallow) requires it, and what is its primary necessity as demonstrated by this example?