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?

Solution

(A) Storage Allocation

- `A`: Static object

- `B`: Object on stack (parameter in `process`)
- `C`: Object on stack (local in `process`)
- `D`: Static object (function-local `static` in `main`)
- `E`: Object on stack (local in `main`)
- `F`: Object on stack (pointer variable in `main`)
- `new int(50)`: Object on heap
- `G`: Object on stack (block-local in the `if`-block)

(B) Scope and Lifetime

- `A`: **Scope**= global (file/namespace) scope. **Lifetime**= whole program execution.
- `B`: **Scope** = function scope (`process`). **Lifetime** = active during call to `process`.
- `C`: **Scope** = function scope (`process`). **Lifetime** = active during call to `process`.
- `D`: **Scope** = function scope (`main`). **Lifetime** = whole program execution (static object).
- `E`: **Scope** = function scope (`main`). **Lifetime** = active during call to `main`.
- `F`: **Scope** = function scope (`main`). **Lifetime** = active during call to `main` (the pointer variable itself).
- **Heap object `new int(50)`**: **Scope** = no lexical name of its own; accessed via `F`. **Lifetime** = arbitrary: begins at `new` (line 11) and ends at `delete` (line 20).
- `G`: **Scope** = block scope (inside the `if`-block). **Lifetime** = active while that block executes.

(C) Scope vs Lifetime of `D`

- **Scope**: limited to the body of `main` (function scope); `D` is not visible outside `main`.
- **Lifetime**: because it is declared `static`, its object exists for the entire program execution (static object).
- **Why this shows the difference**: `D` has <u>local scope</u> but a <u>program-long lifetime</u>, illustrating that scope and lifetime are distinct concepts.

CMPSC461 Fall 2025

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.

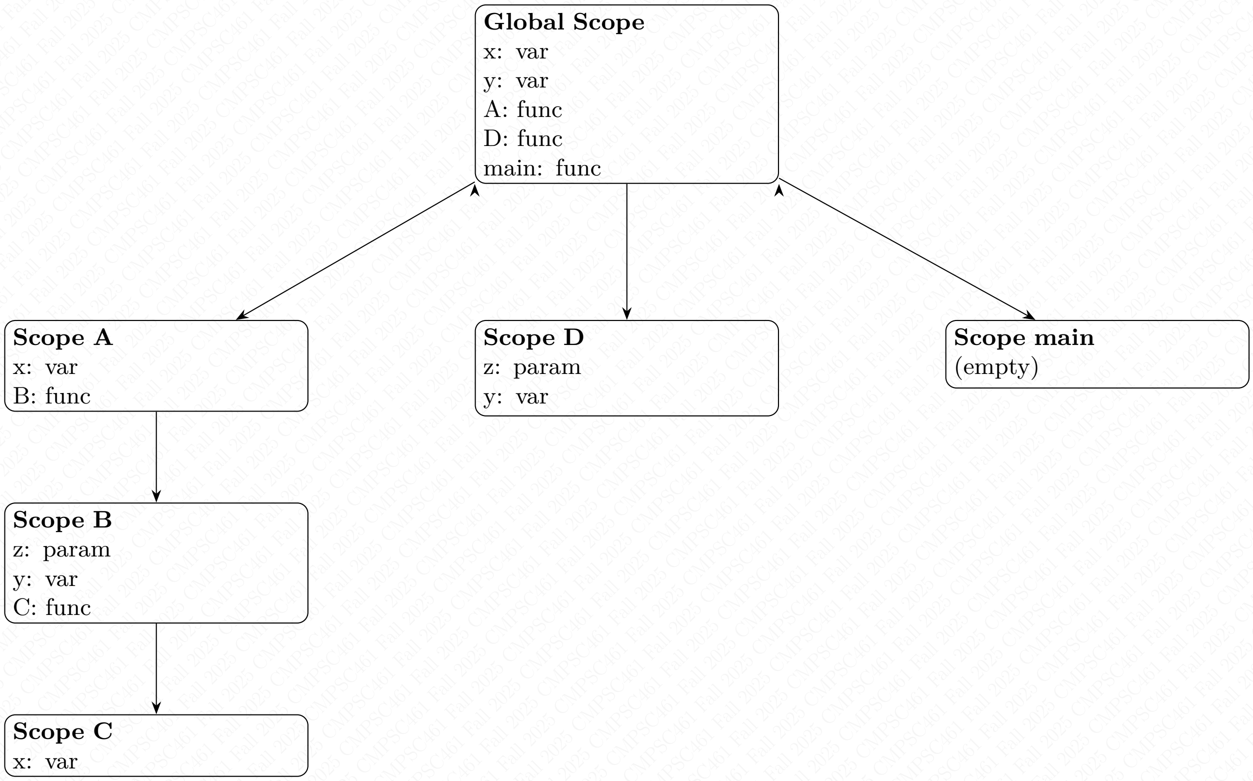
(B) **(3 pts):** What is the program's output under static scoping?

(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.

Solution

(A) **Hierarchical Symbol Tables (Static Scoping)**

The symbol tables are structured based on the lexical (textual) nesting of the code.



(B) Static Scoping Output

Output: ‘15 41 10 10 2‘

(C) Variable Resolution Under Static Scoping

- **In `main`**:
  - The use of `y` in `print(y)` searches scope `main` (not found), then its parent **Global**, where it is found.
- **In `A`**:
  - The use of `x` in `print(x)` searches scope `A` and is found **locally**.
- **In `B`**:
  - In `y = z + x`, the use of `x` searches scope `B` (not found), then its parent **A**, where it is found.
  - In `print(y)`, the use of `y` searches scope `B` and is found **locally**.
  - In `print(x)`, the use of `x` searches scope `B` (not found), then its parent **A**, where it is found.
- **In `C`**:
  - In `x = y + 5`, the use of `y` searches scope `C` (not found), then its parent **B**, where it is found.
- **In `D`**:
  - In `print(y + x)`, the use of `y` searches scope `D` and is found **locally**.
  - The use of `x` searches scope `D` (not found), then its parent **Global**, where it is found.

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.

Solution

(A) **Static Scoping Output and Trace**

Output: ‘100 50 100‘

- `main` calls `outer`, which calls `recur(2)`, which calls `recur(1)`, which calls `recur(0)`, which finally calls `helper()`.

CMPSC461 Fall 2025

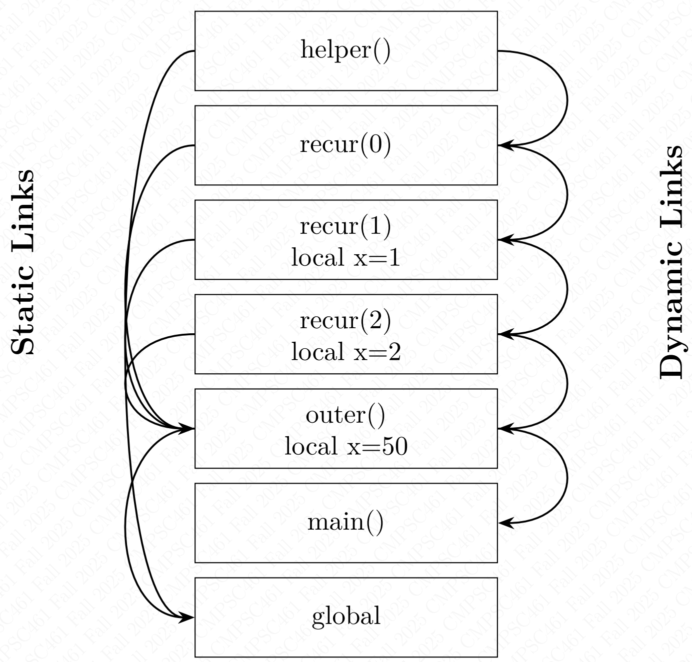
- In `helper()`, `print(x)` resolves `x` based on its lexical (static) parent, which is the **Global** scope. The global `x` is 100. **Prints 100**.
- Execution returns to `outer()`, and `print(x)` resolves to the `x` local to `outer`. **Prints 50**.
- Execution returns to `main()`, and `print(x)` resolves to the **Global** `x`. **Prints 100**.

(B) Dynamic Scoping Output

Output: ‘1 50 100‘

(C) Runtime Stack Diagram

At the moment `helper()` is called, the stack contains frames for all active calls. Static links point to the frame of the lexically enclosing scope, while dynamic links point to the caller's frame.



(D) Variable Resolution Explanation for `x` in `helper()`

- **Static Scoping:** The resolution follows the **static link**. From the `helper` frame, the static link points to its lexical parent's environment (the Global scope, represented by the `main` frame in the diagram). The search finds the global `x`, whose value is **100**.
- **Dynamic Scoping:** The resolution follows the **dynamic links** down the call stack. From the `helper` frame, the search checks its caller, `recur(0)` (no `x`), then its caller, `recur(1)`. The frame for `recur(1)` contains a local binding for `x` with the value **1**. The search stops there.

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?

Solution

(A) **Shallow Binding Output**

Output: ‘10‘

Explanation: Shallow binding uses the environment of the **call site**.

- The function `worker` (referenced by `F`) is called from the base case of the recursion, inside `recursive_executor(0)`.

- At this point, the call stack is `main -> recursive_executor(2) -> recursive_executor(1) -> recursive_executor(0) -> worker`.
- The search for `val` begins at the top of the stack. It checks `worker` (none), then its caller `recursive_executor(0)` (none).
- It proceeds to the next frame, `recursive_executor(1)`, where it finds a local `val` with a value of `1 * 10 = 10`. The search stops and this value is printed.

(B) Deep Binding Output

Output: ‘50‘

Explanation: Deep binding uses the environment from when the function reference was **create***.

- The function reference `my_func` is created and assigned inside the `main` function by the call to `setup()`.
- When `setup()` executes, deep binding creates a **closure**, which is a pair containing the function pointer for `worker` and a reference to the environment of `setup()`. This environment contains the local variable `val = 50`.
- This closure is what gets passed to `recursive_executor`.
- When `worker` is finally called, it uses its captured environment (the one from `setup()`) to resolve `val`, printing **50**.

(C) Closures

- **What is a closure?** A closure is a pair consisting of a pointer to a function and a reference to its **referencing environment**. This environment is where the function was defined or the reference to it was created.
- **Which binding rule needs it? Deep binding** requires closures to implement its behavior.
- **What is its necessity?** The necessity of a closure is to preserve a referencing environment that would otherwise be destroyed. In this program, the function reference is created inside `setup()`. After `setup()` returns, its activation record is popped from the stack and its local `val` is destroyed. Without the closure capturing and preserving that environment, deep binding would be impossible, as there would be no way to access the value `50`.