Queens College of CUNY
Department of Computer Science
Programming Languages
(CSCI 316)
Winter 2026

Assignment #9
"Subprogram Scoping Simulation"
Due: January 17, 2026

Introduction:

In this assignment, we simulate the "call stack" of subroutines to illustrate the state of variables and parameters passed by value and by reference. We are also interested in contrasting **Static (Lexical) Scoping** in which variable lookup follows the static link (parent) chain vs. **Dynamic Scoping** io which variable lookup follows the call chain.

The following two tables may be helpful:

Aspect	**Caller**	**Parent**
Definition	The specific subprogram that initiates the execution of another subprogram at a given point during runtime.	The subprogram whose code block immediately and statically contains the definition of a nested subprogram in the source code.
Relationship	A dynamic relationship. It changes with each invocation. The same subprogram can be called by many different callers.	A static (lexical) relationship. It is fixed at compile time by the source code structure. A subprogram has only one static parent.
Purpose	Control flow and execution stack management. The caller is suspended while the called subprogram executes, and control returns to the caller upon completion.	Scoping and variable access (referencing environment) in languages that allow nested subprograms. A nested subprogram can typically access the non-local variables declared in its static parent's scope.
Mechanism	The connection is managed by a dynamic link (return address and saved execution status) on the call stack.	The connection is managed by a static link (a pointer to the parent's activation record instance) in languages that use static scoping with nested functions.

Feature	**Static (Lexical) Scoping**	**Dynamic Scoping**

Determination	Compile-time (based on code's physical nesting).	Runtime (based on call stack order).
Variable Lookup	Finds the closest enclosing declaration in the source code.	Finds the most recently active declaration in the call stack.
Predictability	Highly predictable; easier for humans to reason about.	Less predictable; can change with different call sequences.
Implementation	Compiler uses symbol tables to map names to scopes.	Interpreter/runtime uses linked lists/stack for bindings.
Example	In f() calling g(), g's x is found in f's scope (if nested).	In f() calling g(), g's x could be from f or an outer scope, depending on who called f.
Usage	Most modern languages (C, Java, Python, etc.).	Older Lisps, some shell

Submissions:

In the Google form, please submit:

- Assignment09.py (source code - MODIFIED FOR DYNAMIC SCOPING)
- ~~Assignment09.py (console output for both static and dynamic scoping)~~

Tasks:

[1] Use the code below to illustrate "static (aka lexical) scoping". Understand the goals and structure.

[2] Modify the code to use "dynamic scoping" that is, when resolving a variable name, look in the current frame, and if not found, follow the dynamic link (`caller`) chain instead of the static link (`parent`)

```
from dataclasses import dataclass

# ---------- core runtime ----------
@dataclass
class Cell:
    value: int

@dataclass
class Frame:
    name: str
    parent: "Frame | None"   # static link (lexical parent)
    caller: "Frame | None"   # dynamic link (caller)
    env: dict                # name -> Cell

class Runtime:
    def __init__(self, debug: bool = False):
        self.debug = debug
        self.stack = [Frame("GLOBAL", None, None, {})]

    def _dbg(self, msg: str) -> None:
        if self.debug:
            print(f"[DBG] {msg}")
            print(self.dump_stack())

    def dump_stack(self) -> str:
        lines = ["STACK (top last):"]
        for i, fr in enumerate(self.stack):
            env_view = {k: v.value for k, v in fr.env.items()}
            lines.append(
                f"  [{i}] {fr.name} "
                f"dyn={fr.caller.name if fr.caller else None} "
                f"stat={fr.parent.name if fr.parent else None} "
                f"env={env_view}"
            )
        return "\n".join(lines)

    def top(self) -> Frame:
        return self.stack[-1]

    def declare(self, name: str, value: int) -> None:
        self.top().env[name] = Cell(int(value))
        self._dbg(f"declare {name}={value} in {self.top().name}")

    def lookup_cell(self, name: str) -> Cell:
        fr = self.top()
        hops = 0
        while fr is not None:
            if name in fr.env:
                if self.debug:
                    self._dbg(f"lookup {name} found in {fr.name} (static hops={hops})")
                return fr.env[name]
            fr = fr.parent  # static chain lookup
            hops += 1
        raise NameError(f"Undefined name: {name}")

    def get(self, name: str) -> int:
        return self.lookup_cell(name).value

    def set(self, name: str, value: int) -> None:
        cell = self.lookup_cell(name)
        old = cell.value
        cell.value = int(value)
        self._dbg(f"set {name}: {old} -> {value}")

    def push(self, fr: Frame) -> None:
        self.stack.append(fr)
        self._dbg(f"push frame {fr.name}")

    def pop(self) -> None:
        fr = self.stack.pop()
        self._dbg(f"pop frame {fr.name}")

    def call(self, fn, arg_cells: dict, parent: Frame, byref: set[str] = set()):
        """
        fn(rt): executes using current frame
        arg_cells: mapping param -> Cell (from caller)
        parent: static link for callee (lexical parent)
        byref: which params are passed by reference; others are by value
        """
        caller = self.top()
        fr = Frame(fn.__name__, parent, caller, {})

        # Bind parameters into the callee frame
        for pname, caller_cell in arg_cells.items():
            if pname in byref:
                fr.env[pname] = caller_cell              # alias
            else:
                fr.env[pname] = Cell(caller_cell.value)  # copy

        self._dbg(
            f"CALL {fn.__name__} "
            f"args={{" + ", ".join(
                f"{k}={'&' if k in byref else ''}{v.value}" for k, v in arg_cells.items()
            ) + "}} "
            f"static_link={parent.name if parent else None} "
            f"dynamic_link={caller.name if caller else None}"
        )

        self.push(fr)
        fn(self)
        self.pop()

# ---------- "program" built with subprograms ----------
def inner(rt: Runtime):
    # return a + b + x (resolved via static scope)
    rt.declare("_ret", rt.get("a") + rt.get("b") + rt.get("x"))

def outer(rt: Runtime):
    # param: a
    rt.declare("b", 10)
    rt.declare("x", 5)
    # inner is lexically nested in outer, so its static link should be the current outer
frame
    rt.call(inner, {"x": rt.lookup_cell("x")}, parent=rt.top())
    # forward return (store into current frame for easy retrieval by main)
    rt.declare("_ret", rt.get("_ret"))

def inc(rt: Runtime):
    # param: p
    rt.set("p", rt.get("p") + 1)

def main(rt: Runtime):
    # outer(7) -> 22
    rt.declare("a", 7)
    rt.call(outer, {"a": rt.lookup_cell("a")}, parent=rt.top())
    print(rt.get("_ret"))

    # pass-by-value vs pass-by-reference
    rt.declare("x", 10)

    # by value: formal p is a copy
    rt.call(inc, {"p": rt.lookup_cell("x")}, parent=rt.top(), byref=set())
    print("x after by-value inc:", rt.get("x"))  # 10

    # by reference: formal p aliases caller's x
    rt.call(inc, {"p": rt.lookup_cell("x")}, parent=rt.top(), byref={"p"})
    print("x after by-ref inc:", rt.get("x"))    # 11

if __name__ == "__main__":
    # Flip debug=True to see stack/lookup/call traces
    rt = Runtime(debug=False)
    main(rt)
```