PPL Assignment

IIT2022029
Yash Ganeriwal
Section - A

1. Give an example of static scoping in languages that allow nested subprograms and show how static scoping is implemented in such languages with the help of this example.
Ans)
Static Scoping (Lexical Scoping):
Definition:

- Static scoping is a scoping mechanism where the binding of a variable is determined by the **program text** (lexical structure) and is **independent of the run-time call stack**.
- It's based on the nesting of subprograms (functions or blocks) within each other.

How It Works:

- Variables are resolved based on their **lexical context** (where they appear in the code).
- The scope of a variable is determined by its **surrounding block or function**.
- The compiler or interpreter can determine variable bindings at compile time.

Some programming languages that support static (lexical) scoping and allow nesting of subprograms:

1. Python: Python uses static scoping (lexical scoping) for variable resolution. Nested functions can access variables from their enclosing scope.
2. JavaScript: JavaScript also employs lexical scoping. Functions defined within other functions (closures) have access to variables in their outer scope.

Sample Python Code to show static scoping as python allows nested subprograms :

```
x = 10
def f():
    return x

def g():
    x = 20
    return f()

print(g())

# Output: 10
```

In this example:

- The global variable `x` is assigned the value `10`.

- Function `f()` always returns the value of the global `x`.
- Function `g()` has its own local variable `x` (with value `20`), but it still calls `f()`.
- The output of `g()` is `10`, demonstrating static scoping.
- The binding of `x` is determined by the program text, not the runtime call stack.

2. Implement a recursive C/C++ program for the tower of Hanoi problem.Now, for this program, what is the maximum number of activation record instances witnessed in the runtime stack? Also, for this state of the runtime stack, provide the values of the parameters (of the recursive function) in all the activation record instances.

Ans)

C++ code :

```
#include <iostream>
void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 1) {
    std::cout << "Move disk 1 from rod " << source << " to rod " <<
   destination << std::endl;
        return;
    }
    towerOfHanoi(n - 1, source, destination, auxiliary);
    std::cout << "Move disk " << n << " from rod " << source << " to rod " <<
destination << std::endl;
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int n = 3; // Number of disks
    towerOfHanoi(n, 'A', 'B', 'C');  // A, B and C are names of rods
    return 0;
}
```

For ’n’ disks, in the Tower of Hanoi problem, the maximum number of activation record instances will be `2^n - 1`. This result can be proved using mathematical induction.

For `n = 3`, the maximum number of activation record instances will be `2^3 - 1 = 7`.

The values of the parameters for each activation record instance:

1. **towerOfHanoi(3, 'A', 'B', 'C'):** This is the initial call with three disks on rod A, with rod B as auxiliary, and rod C as the destination.

2. **towerOfHanoi(2, 'A', 'C', 'B'):** In this call, the function is recursively called with two disks, moving the top two disks from rod A to rod B, using rod C as auxiliary.
3. **towerOfHanoi(1, 'A', 'B', 'C'):** This call moves the bottom disk from rod A to rod C using rod B as auxiliary.
4. **towerOfHanoi(1, 'A', 'C', 'B'):** Now, the function is called to move the disk from rod C to rod B using rod A as auxiliary.
5. **towerOfHanoi(2, 'B', 'A', 'C'):** This call is for moving the two disks from rod B to rod C, using rod A as auxiliary.
6. **towerOfHanoi(1, 'B', 'C', 'A'):** Moves the bottom disk from rod B to rod A using rod C as auxiliary.
7. **towerOfHanoi(1, 'B', 'A', 'C'):** Finally, the last disk is moved from rod A to rod C using rod B as auxiliary.

3. Discuss in brief the Flynn's classification with an example.
Ans)
Flynn's classification is a system proposed by Michael J. Flynn in 1966 and extended in 1972 to categorise computer architectures based on the number of instructions and data items that can be manipulated simultaneously. Let's delve into the four major categories of Flynn's classification:

Single-instruction, single-data (SISD) systems:

- SISD represents a **uniprocessor machine** capable of executing a **single instruction** on a **single data stream**.
- In SISD, machine instructions are processed **sequentially**, and these computers are commonly known as **sequential computers**.
- Most conventional computers, including the **IBM PC** and workstations, follow the SISD architecture.
- All instructions and data to be processed are stored in **primary memory**.
- The processing speed in the SISD model is limited by the rate at which the computer can transfer information internally.
-

Single-instruction, multiple-data (SIMD) systems:

- An SIMD system is a **multiprocessor machine** capable of executing the **same instruction** on **multiple CPUs**, but operating on **different data streams**.
- These machines are well-suited for **scientific computing**, especially tasks involving vector and matrix operations.
- In SIMD systems, data elements are organised into vectors, which can be divided into multiple sets (N-sets for N PE systems). Each processing element (PE) processes one data set.
- A dominant representative of SIMD systems is **Cray's vector processing machine**.

Multiple-instruction, single-data (MISD) systems:

- An MISD computing system is a **multiprocessor machine** capable of executing **different instructions** on **different PEs**, but all of them operate on the **same dataset**.
- Although not widely used, MISD systems perform different operations on the same data set.
- An example of an MISD operation could be calculating `Z = sin(x) + cos(x) + tan(x)` where different operations (sin, cos, and tan) are applied to the same input data.

Multiple-instruction, multiple-data (MIMD) systems:

- An MIMD system is a **multiprocessor machine** capable of executing **multiple instructions** on **multiple data sets**.
- Each PE in the MIMD model has separate instruction and data streams.
- Machines built using the MIMD model are versatile and can handle a wide range of applications.
- Examples of MIMD systems include **parallel clusters** and **distributed computing environments**.

In summary, Flynn's classification provides a framework for understanding the organization of computer systems based on how instructions and data are processed simultaneously.

4. Suppose two tasks, A and B, must use the shared variable Buf_Size. Task A adds 2 to Buf_Size, and task B subtracts 1 from it. Assume that such arithmetic operations are done by the three-step process of fetching the current value, performing the arithmetic, and putting the new value back. In the absence of competition synchronization, what sequences of events are possible and what values result from these operations?
Assume that the initial value of Buf_Size is 6.
Ans)

In the absence of competition synchronization, where two tasks operate on a shared variable without coordination, various sequences of events can occur. The possible scenarios and resulting values for the shared variable `Buf_Size are :`

Add Completes Before Subtract:

- Task A (addition) completes before Task B (subtraction).
- Sequence: A fetches the current value (6), adds 2, and puts back the new value (8). Then B fetches the updated value (8), subtracts 1, and puts back the new value (7).
- Result: `Buf_Size` becomes 7.

Subtract Completes Before Add:

- Task B (subtraction) completes before Task A (addition).
- Sequence: B fetches the current value (6), subtracts 1, and puts back the new value (5). Then A fetches the updated value (5), adds 2, and puts back the new value (7).
- Result: `Buf_Size` becomes 7.

Race Condition (Interleaved Execution):

- Task A and Task B execute concurrently, leading to interleaved steps.
  - Possible interleaving:
  - A fetches the current value (6).
  - B fetches the same current value (6).
  - A adds 2 and puts back the new value (8).
  - B subtracts 1 and puts back the new value (7).
- Result: `Buf_Size` becomes 7.

Race Condition (Different Interleaving):

- Another possible interleaving:
- B fetches the current value (6).
- A fetches the same current value (6).
- B subtracts 1 and puts back the new value (5).
- A adds 2 and puts back the new value (7).
- Result: `Buf_Size` becomes 7.

Interrupted Operations:
Either Task A or Task B gets interrupted mid-operation.

- If A is interrupted after fetching (6) but before adding 2,
- `Buf_Size` remains 6.
- If B is interrupted after fetching (6) but before subtracting 1,
- `Buf_Size` remains 6.

In summary, without proper synchronization, the final value of `Buf_Size` can be either 5, 6, or 7, depending on the order of execution and potential interruptions.

5. Consider the Readers writers problem, a classic synchronization problem. In this problem, there is a shared resource which can be accessed by multiple processes. There are two types of processes: the reader and the writer. Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource. When a writer is writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.
Provide a solution to the above problem using semaphores.

One possible solution to the Readers-Writers problem using semaphores involves using two semaphores: one for controlling access by writers and another for controlling access by readers. Additionally, we use a mutex (binary semaphore) to ensure mutual exclusion when updating shared variables.

```
// Initialize semaphores

semaphore mutex = 1 // Binary semaphore for mutual exclusion
semaphore wrt = 1   // Semaphore for controlling access by writers
int read_count = 0  // Counter for tracking the number of readers
```

```
// Writer process
writer() {
    wait(wrt);          // Wait until no writer is writing
    write_data();       // Write data to the shared resource
    signal(wrt);        // Release the writer semaphore
}


// Reader process
reader() {
    wait(mutex);        // Ensure mutual exclusion when updating read_count
    read_count++;       // Increment the reader count
    if (read_count == 1) {
        wait(wrt);      // If this is the first reader, wait for no writers
    }
    signal(mutex);      // Release the mutex
    // Read from the shared resource
    read_data();
    wait(mutex);        // Ensure mutual exclusion when updating read_count
    read_count--;       // Decrement the reader count
    if (read_count == 0) {
        signal(wrt);    // If this is the last reader, signal writers to write
    }
    signal(mutex);      // Release the mutex
}
```

Explanation:

- The `mutex` semaphore is used to ensure mutual exclusion when updating the `read_count` variable to prevent race conditions.
- The `wrt` semaphore controls access by writers. Writers must wait until there are no readers or writers accessing the shared resource before writing.
- The `read_count` variable keeps track of the number of readers currently accessing the shared resource. Writers must wait until there are no readers accessing the resource before writing.
- wait(semaphore):
  - `wait(mutex)`: Acquires the mutex semaphore, ensuring mutual exclusion when updating shared variables.
  - `wait(wrt)`: Waits until the `wrt` semaphore becomes available, indicating that no writer is currently writing to the shared resource.
- signal(semaphore):
  - `signal(mutex)`: Releases the mutex semaphore, allowing other processes to acquire it.
  - `signal(wrt)`: Signals that the writer has finished writing, allowing other writers or readers to access the shared resource.
- Reader Process (reader()):
  - Upon entry, a reader process acquires the mutex semaphore to update the `read_count` variable.
  - If it's the first reader, it waits for the `wrt` semaphore to ensure no writers are currently writing.

- After reading from the shared resource, the reader releases the mutex semaphore to allow other processes to update `read_count`.
- Before exiting, the reader process again acquires the mutex semaphore to update `read_count` and signals writers if it's the last reader.

- Writer Process (writer()):
  - Upon entry, a writer process waits for the `wrt` semaphore to ensure no other writer or reader is currently accessing the shared resource.
  - It then proceeds to write to the shared resource.
  - After writing, the writer signals the `wrt` semaphore to allow other writers or readers to access the shared resource.

6. Write a program using scheme to:

A. Compute Product of All Elements in an Integer List:

code 1 :

```
(Define (prod list_1 ans) (
 If (null? list_1)
        (ans)
        ( prod (Cdr list_1) (* ans (Car list_1) ) ) ) )

(Define result (product-list '(2 3 4 5) 1))

(display "Product of elements in the list: ")
(display result)
```

Code 2 :

```
(define (product-list lst)
  (if (null? lst)
      1
      (* (car lst) (product-list (cdr lst)))))

(display "Product of elements in the list: ")
(display (product-list '(2 3 4 5))) ; Example input list
(newline)
```

B. Compute GCD of Two Numbers:

```
(define (gcd a b)
  (if (< a b)
      (gcd b a)          ; Swap a and b if a < b
      (if (= b 0)
          a
          (gcd b (remainder a b)))))

(display "GCD of 98 and 56 is: ")
(display (gcd 98 56))
(newline)
```

C. Compute reverse of a list

Code 1 :

```
(define (rev_list list_1 list_2)
        if(null? list_1)
                list_2
                (rev_list (cdr list_1) (cons (car list_1) list2)))
)
```

Code 2 :

```
(define (reverse-list lst)
  (if (null? lst)
      '()
      (append (reverse-list (cdr lst)) (list (car lst)))))

(display "Reversed list: ")
(display (reverse-list '(1 2 3 4 5)))
```

D. Compute the length of a list

```
(define (len_of list_1)
  (define (len_cal list_1 ans)
    (if (null? list_1)
        ans
        (if (list? (car list_1))
            (len_cal (cdr list_1) (+ ans (len_cal (car list_1) 0)))
            (len_cal (cdr list_1) (+ ans 1))
        )
    )
  )
  (len_cal list_1 0)
)
```

E. Compute the nth number of the fibonacci series

```
(  define (fib n)
        ( if ( <= n 1)
                 n
                (+ (fib (- n 1)) (fib (- n 2)))
)
(display (fib 7))
```

F. Insert a number into the correct position of a sorted list

```
(define (insert-sorted num lst)
  (cond
    ((null? lst) (list num))
    ((< num (car lst)) (cons num lst))
    (else (cons (car lst) (insert-sorted num (cdr lst))))))

(display "Sorted list after inserting 7: ")
(display (insert-sorted 7 '(1 3 5 9)))
```

G. Use the above function (in f) to sort a given unsorted list.

```
(define (insert-sorted num lst)
  (cond
```

```
    ((null? lst) (list num))
    ((< num (car lst)) (cons num lst))
    (else (cons (car lst) (insert-sorted num (cdr lst) ) ) )
   )
)

(define (sort-list lst)
  (if (null? lst)
      '()
      (insert-sorted (car lst) (sort-list (cdr lst) ) )
  )
)

(display "Sorted list after inserting 7: ")
(display (insert-sorted 7 '(1 3 5 9)))
(newline)

(display "Sorted list: ")
(display (sort-list '(9 1 5 3 7)))
(newline)
```