ASSIGNMENT 7
04/04/25

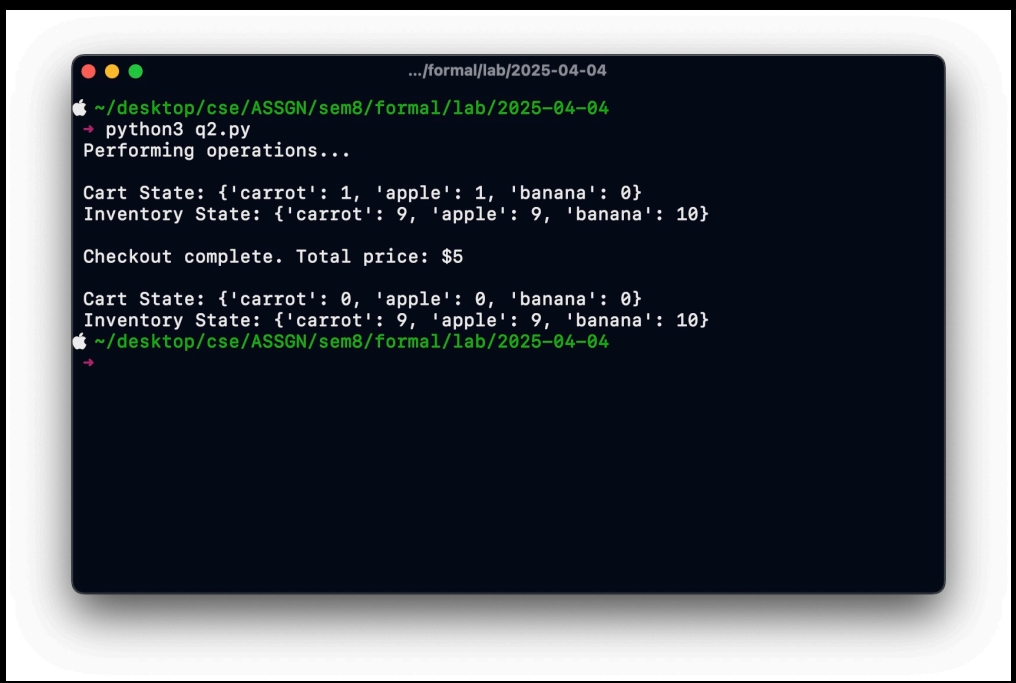
NAME : SHRESTH SONKAR
REGNO : 20214272
GROUP : CS8D
TOPIC : FORMAL METHOD LAB
CODE : CS-18201

```python
# Q1 Write Z notation specifications for a library
management system and validate the specifications in
python

class Library:
    def __init__(self):
        self.books = set()
        self.members = set()
        self.borrowed = {}

    def add_book(self, book):
        if book in self.books:
            raise ValueError(f"Book '{book}' already
exists.")
        self.books.add(book)

    def register_member(self, member):
        if member in self.members:
            raise ValueError(f"Member '{member}'
already registered.")
        self.members.add(member)

    def borrow_book(self, book, member):
        if book not in self.books:
            raise ValueError(f"Book '{book}' does not
exist.")
        if member not in self.members:
            raise ValueError(f"Member '{member}' is not
registered.")
        if book in self.borrowed:
            raise ValueError(f"Book '{book}' is already
borrowed by '{self.borrowed[book]}'.")
        self.borrowed[book] = member

    def return_book(self, book):
        if book not in self.borrowed:
            raise ValueError(f"Book '{book}' was not
borrowed.")
        del self.borrowed[book]

    def print_status(self):
        print("Library Status:")
        print(f"  Books   : {sorted(self.books)}")
        print(f"  Members : {sorted(self.members)}")
        print(f"  Borrowed: {self.borrowed}")
```

```
        print()


if __name__ == "__main__":
    lib = Library()
    lib.add_book("Book1")
    lib.add_book("Book2")

    lib.register_member("Alice")
    lib.register_member("Bob")

    lib.print_status()
    lib.borrow_book("Book1", "Alice")
    lib.print_status()

    try:
        lib.borrow_book("Book1", "Bob")
    except ValueError as e:
        print("Error:", e)

    lib.return_book("Book1")
    lib.print_status()

    try:
        lib.return_book("Book2")
    except ValueError as e:
        print("Error:", e)
```

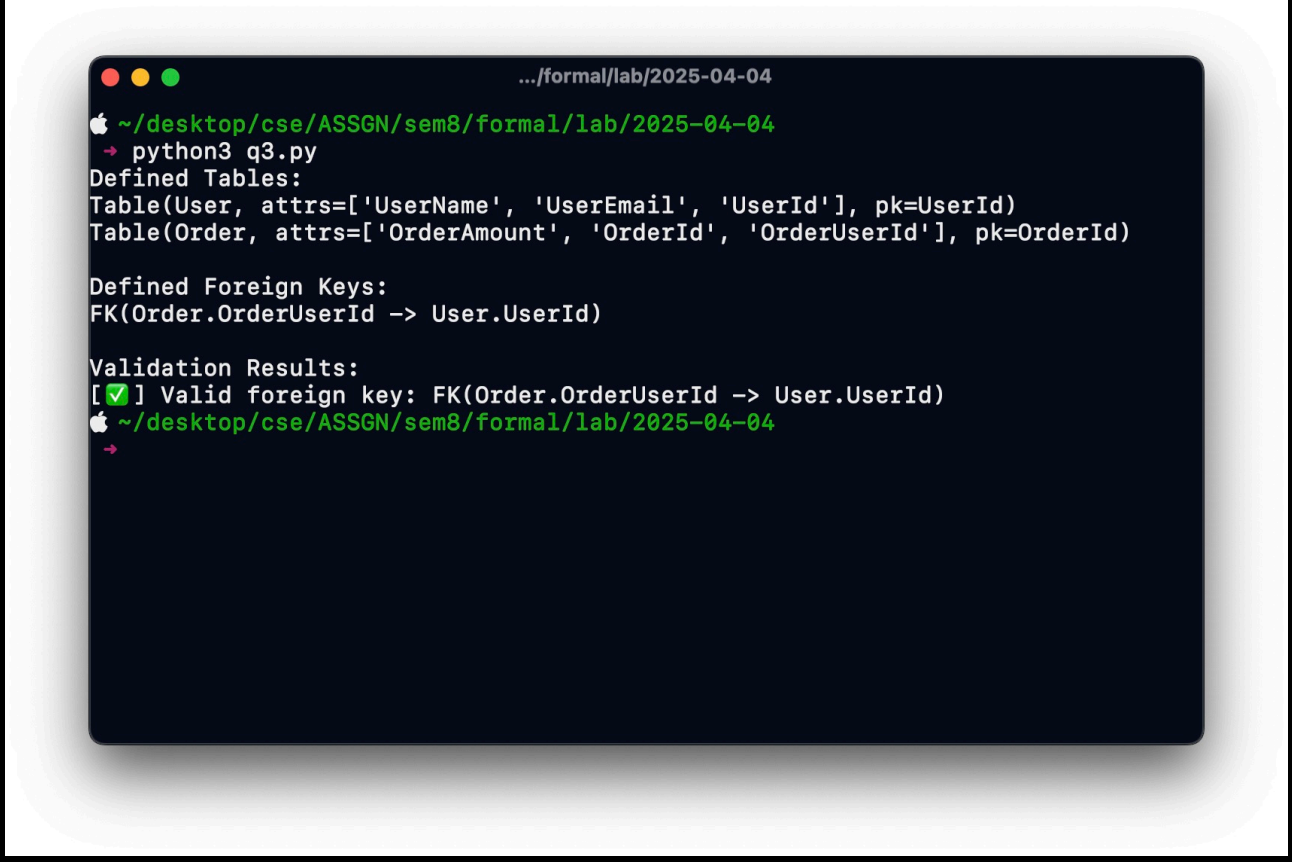
```python
# Q2 Implement a B-Method specification for a basic e-commerce checkout system and verify consistency in python

class ECommerceCheckout:
    def __init__(self, all_items, prices):
        self.all_items = set(all_items)
        self.prices = dict(prices)
        assert self.all_items == set(self.prices.keys()), "Prices must cover all items"

        self.cart = {item: 0 for item in self.all_items}
        self.inventory = {item: 10 for item in self.all_items}  # assume 10 units of each item

    def check_invariant(self):
        for item in self.all_items:
            assert self.cart[item] >= 0, f"Invariant violated: cart has negative quantity of {item}"
            assert self.inventory[item] >= 0, f"Invariant violated: inventory has negative quantity of {item}"
            total_available = self.cart[item] + self.inventory[item]
            assert total_available <= 10, f"Invariant violated: total for {item} exceeds initial stock (10)"

    def add_item(self, item):
        if item in self.all_items and self.inventory[item] > 0:
            self.cart[item] += 1
            self.inventory[item] -= 1
        else:
            raise ValueError(f"Cannot add item '{item}': invalid or out of stock")
        self.check_invariant()

    def remove_item(self, item):
        if item in self.all_items and self.cart[item] > 0:
            self.cart[item] -= 1
            self.inventory[item] += 1
        else:
```

```
            raise ValueError(f"Cannot remove item
'{item}': not in cart")
        self.check_invariant()

    def checkout(self):
        total = sum(self.cart[item] * self.prices[item]
for item in self.cart)
        print(f"\n Checkout complete. Total price: $
{total}")
        self.cart = {item: 0 for item in
self.all_items}
        self.check_invariant()

    def print_state(self):
        print("\n Cart State:", self.cart)
        print(" Inventory State:", self.inventory)


if __name__ == "__main__":
    items = ['apple', 'banana', 'carrot']
    prices = {'apple': 2, 'banana': 1, 'carrot': 3}

    checkout_system = ECommerceCheckout(items, prices)

    print(" Performing operations...")
    checkout_system.add_item('apple')
    checkout_system.add_item('banana')
    checkout_system.add_item('carrot')
    checkout_system.remove_item('banana')
    checkout_system.print_state()
    checkout_system.checkout()
    checkout_system.print_state()
```



```python
# Q3 Use Alloy Analyzer to formally specify and analyze
a simple database schema for correctness.

class Attribute:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f"Attr({self.name})"


class Table:
    def __init__(self, name, attributes, primary_key):
        self.name = name
        self.attrs = set(attributes)
        self.pk = primary_key
        if primary_key not in self.attrs:
            raise ValueError(f"Primary key
'{primary_key}' not in attributes of table '{name}'")

    def __repr__(self):
        return f"Table({self.name}, attrs={[a.name for
a in self.attrs]}, pk={self.pk.name})"


class ForeignKey:
    def __init__(self, from_table, from_attr, to_table,
to_attr):
        self.from_table = from_table
        self.from_attr = from_attr
        self.to_table = to_table
        self.to_attr = to_attr

    def is_valid(self):
        return (self.from_attr in
self.from_table.attrs) and (self.to_attr in
self.to_table.attrs)

    def __repr__(self):
        return f"FK({self.from_table.name}.
{self.from_attr.name} -> {self.to_table.name}.
{self.to_attr.name})"


def define_schema():
```

```
    user_id = Attribute("UserId")
    user_name = Attribute("UserName")
    user_email = Attribute("UserEmail")

    order_id = Attribute("OrderId")
    order_user_id = Attribute("OrderUserId")
    order_amount = Attribute("OrderAmount")

    user_table = Table("User", [user_id, user_name,
user_email], user_id)
    order_table = Table("Order", [order_id,
order_user_id, order_amount], order_id)

    fk = ForeignKey(order_table, order_user_id,
user_table, user_id)

    return user_table, order_table, [fk]


def validate_schema():
    user_table, order_table, foreign_keys =
define_schema()

    print("Defined Tables:")
    print(user_table)
    print(order_table)
    print("\nDefined Foreign Keys:")
    for fk in foreign_keys:
        print(fk)

    print("\nValidation Results:")
    for fk in foreign_keys:
        if not fk.is_valid():
            print(f"[❌] Invalid foreign key: {fk}")
        else:
            print(f"[✅] Valid foreign key: {fk}")


if __name__ == "__main__":
    validate_schema()
```

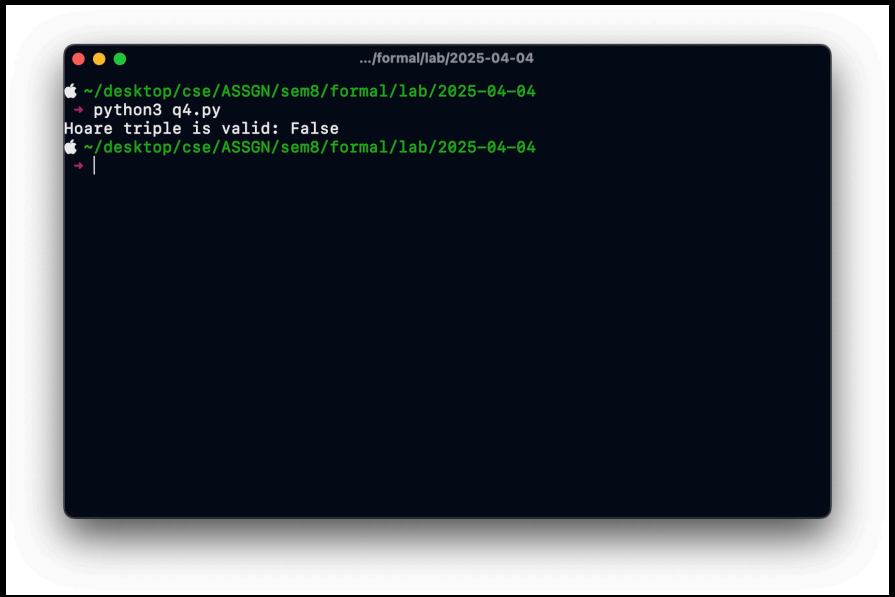



```python
# Q4 Develop a Python-based Hoare Logic verifier for simple imperative programs

from dataclasses import dataclass
from typing import Union


@dataclass
class Expr:
    pass


@dataclass
class Var(Expr):
    name: str


@dataclass
class Const(Expr):
    value: int


@dataclass
class BinOp(Expr):
    op: str  # '+', '-', '*'
    left: Expr
    right: Expr
```

```python
@dataclass
class BoolExpr:
    pass


@dataclass
class RelOp(BoolExpr):
    op: str
    left: Expr
    right: Expr


@dataclass
class Cmd:
    pass


@dataclass
class Skip(Cmd):
    pass


@dataclass
class Assign(Cmd):
    var: str
    expr: Expr


@dataclass
class Seq(Cmd):
    first: Cmd
    second: Cmd


@dataclass
class If(Cmd):
    cond: BoolExpr
    then_cmd: Cmd
    else_cmd: Cmd


@dataclass
class While(Cmd):
```

```
    cond: BoolExpr
    body: Cmd
    invariant: 'Assertion'


@dataclass
class Assertion:
    expr: str


def expr_to_str(e: Expr) -> str:
    if isinstance(e, Var):
        return e.name
    elif isinstance(e, Const):
        return str(e.value)
    elif isinstance(e, BinOp):
        return f"({expr_to_str(e.left)} {e.op}
{expr_to_str(e.right)})"
    else:
        return "?"


def bool_expr_to_str(b: BoolExpr) -> str:
    if isinstance(b, RelOp):
        return f"{expr_to_str(b.left)} {b.op}
{expr_to_str(b.right)}"
    else:
        return "?"


def verify(assertion_pre: Assertion, cmd: Cmd,
assertion_post: Assertion) -> bool:
    if isinstance(cmd, Skip):
        return assertion_pre.expr ==
assertion_post.expr

    elif isinstance(cmd, Assign):
        substituted =
assertion_post.expr.replace(cmd.var,
f"({expr_to_str(cmd.expr)})")
        return assertion_pre.expr == substituted

    elif isinstance(cmd, Seq):
        print("Sequential composition requires a
manually provided intermediate assertion.")
```

```
        return False

    elif isinstance(cmd, If):
        return
(verify(Assertion(f"({assertion_pre.expr}) and
({bool_expr_to_str(cmd.cond)})"),
                        cmd.then_cmd, assertion_post)
and

verify(Assertion(f"({assertion_pre.expr}) and (not
({bool_expr_to_str(cmd.cond)}))"),
                        cmd.else_cmd, assertion_post))

    elif isinstance(cmd, While):
        inv = cmd.invariant
        cond_str = bool_expr_to_str(cmd.cond)
        body_ok = verify(Assertion(f"({inv.expr}) and
({cond_str})"), cmd.body, inv)
        post_ok = inv.expr ==
f"({assertion_post.expr})"
        return body_ok and post_ok

    else:
        print("Unknown command type.")
        return False


if __name__ == "__main__":
    pre = Assertion("x == 0")
    cmd = Assign("x", BinOp("+", Var("x"), Const(1)))
    post = Assertion("x == 1")

    result = verify(pre, cmd, post)
    print("Hoare triple is valid:", result)
```



```python
# Q5 Model and verify preconditions, postconditions,
and invariants for a bank account system using formal
methods in py

class BankAccount:
    def __init__(self, initial_balance: float):
        assert isinstance(initial_balance, (int,
float)), "Initial balance must be a number"
        assert initial_balance >= 0, "Initial balance
must be non-negative"
        self.balance = initial_balance
        self._check_invariant()

    def _check_invariant(self):
        assert self.balance >= 0, f"Invariant violated:
balance is {self.balance}"

    def deposit(self, amount: float):
        assert amount > 0, "Deposit amount must be
positive"
        old_balance = self.balance

        self.balance += amount

        assert self.balance == old_balance + amount,
"Postcondition failed: Incorrect deposit logic"
        self._check_invariant()

    def withdraw(self, amount: float):
        assert amount > 0, "Withdrawal amount must be
positive"
        assert self.balance >= amount, "Insufficient
balance"
        old_balance = self.balance

        self.balance -= amount
        assert self.balance == old_balance - amount,
"Postcondition failed: Incorrect withdrawal logic"
        self._check_invariant()

    def get_balance(self) -> float:
        assert self.balance >= 0, "Postcondition
failed: balance should be non-negative"
        self._check_invariant()
        return self.balance
```

```python
if __name__ == "__main__":
    account = BankAccount(100)

    account.deposit(50)
    print("Balance after deposit:",
account.get_balance())  # Expected: 150

    account.withdraw(30)
    print("Balance after withdrawal:",
account.get_balance())  # Expected: 120
```