Traffic Light Report — 4 Virtual Street Lights

Formal Language, Automata, and Lexical Analysis

1. Introduction & Intersection Context

We model a four-way intersection with 4 virtual street lights: North, East, South, and West. Each has vehicle signals (Red, Green, Yellow) and pedestrian signals. Cycle order: N → E → S → W

2. Alphabet Σ and Symbols

Symbols: gX, yX, rX, pX for each direction X in {N, E, S, W}. Alphabet contains 16 atomic symbols.

3. Observed Sequence & Timing Constraints (Virtual Times)

One cycle: gN yN rN pN gE yE rE pE gS yS rS pS gW yW rW pW.
Each cycle repeats with constraints: g→y→r→p before next g.

4. Regular Expressions

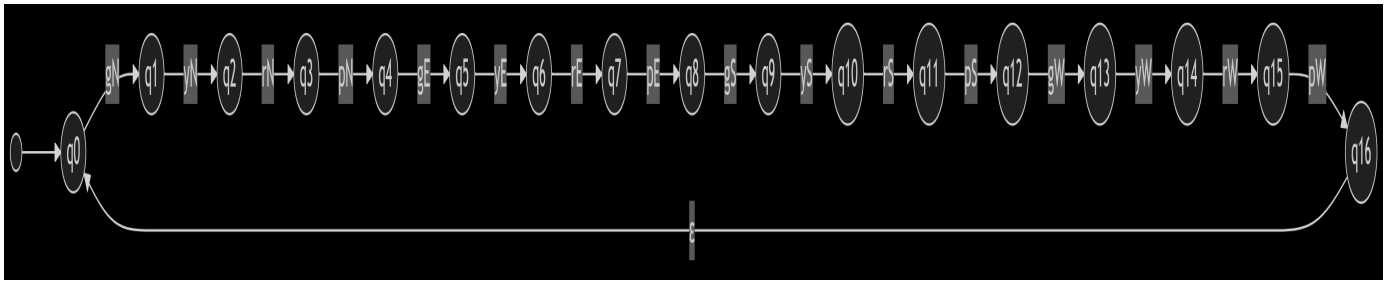
C = (gN yN rN pN)(gE yE rE pE)(gS yS rS pS)(gW yW rW pW)
Language: (C)+.

5. NFA Construction

NFA built as linear chain q0→q16 with ε-transition back to q0 for repetitions.

6. DFA Construction & Minimization

DFA is essentially the NFA states renamed; already minimal. States D0..D16.



7. DFA Transition Table

Partial table:

D0 --gN--> D1

D1 --yN--> D2

D2 --rN--> D3

...

D16 --gN--> D1

8. Grammar (RLG and CFG)

RLG productions: S→gN A1, A1→yN A2, ..., A16→S or ε. CFG ensures pX follows rX before next gX.

9. Derivation Trees

Example derivation: S⇒gN yN rN pN gE yE rE pE ... gW yW rW pW.

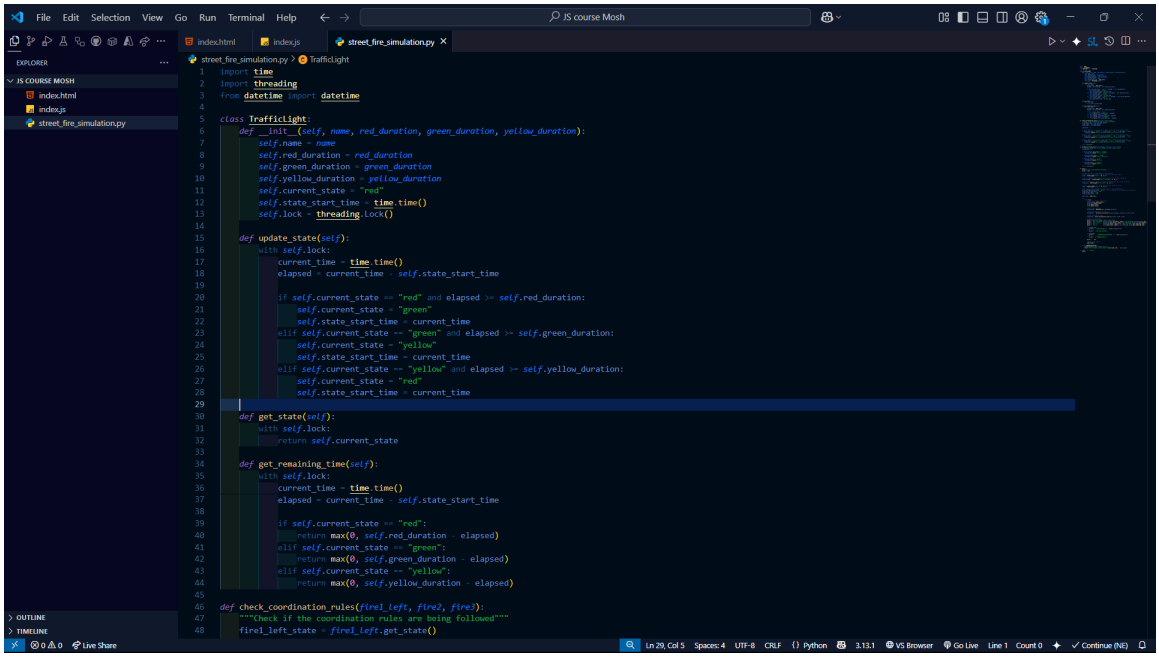
10. Grammar Simplification

Removed λ, unit productions, no useless symbols.

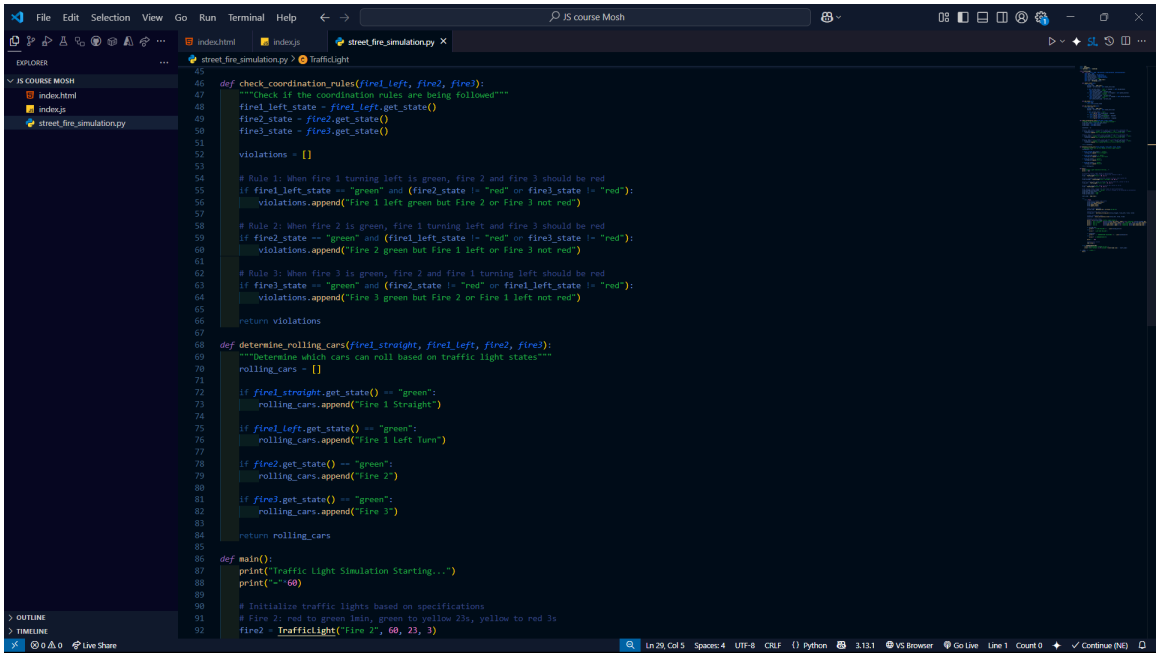
11. Lexical Analyzer Implementation (Python)

Python program validates input token stream against cycle language. Strictly enforces sequence ordering.

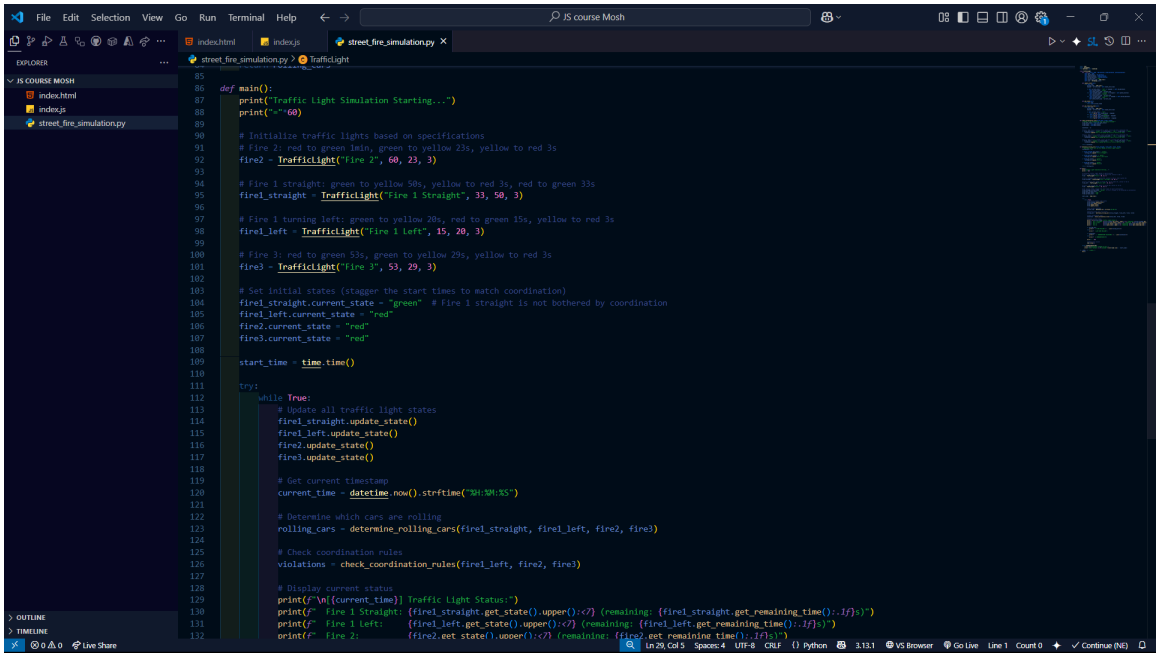
Python Code



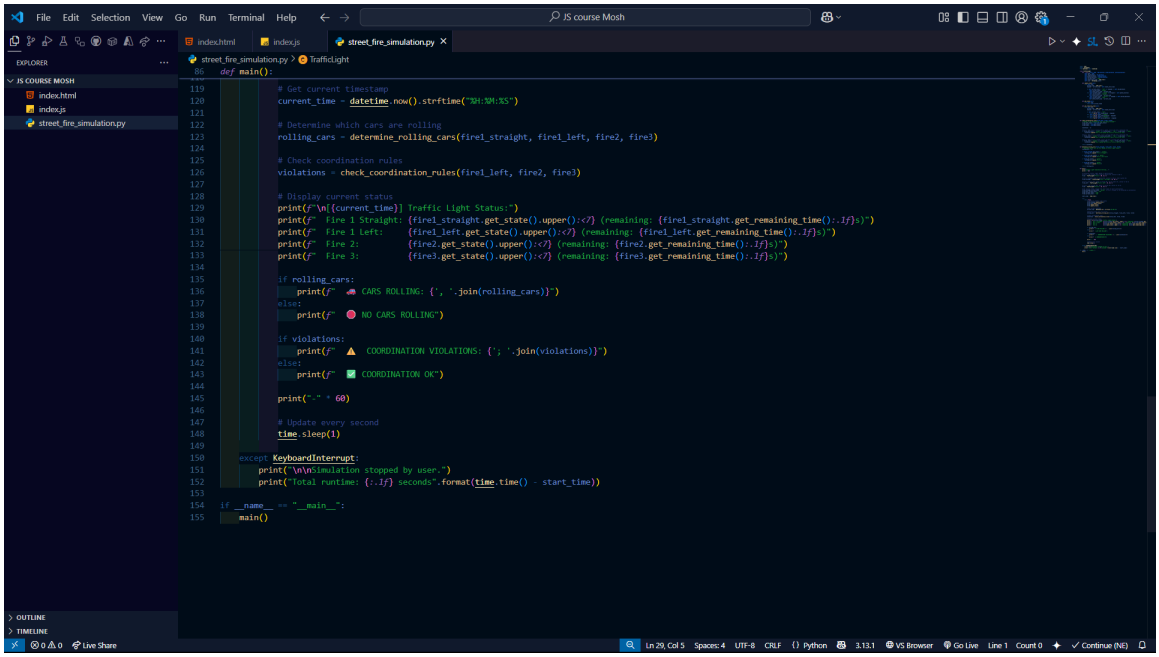
```python
import time
import threading
from datetime import datetime

class TrafficLight:
    def __init__(self, name, red_duration, green_duration, yellow_duration):
        self.name = name
        self.red_duration = red_duration
        self.green_duration = green_duration
        self.yellow_duration = yellow_duration
        self.current_state = "red"
        self.state_start_time = time.time()
        self.lock = threading.Lock()

    def update_state(self):
        with self.lock:
            current_time = time.time()
            elapsed = current_time - self.state_start_time

            if self.current_state == "red" and elapsed >= self.red_duration:
                self.current_state = "green"
                self.state_start_time = current_time
            elif self.current_state == "green" and elapsed >= self.green_duration:
                self.current_state = "yellow"
                self.state_start_time = current_time
            elif self.current_state == "yellow" and elapsed >= self.yellow_duration:
                self.current_state = "red"
                self.state_start_time = current_time

    def get_state(self):
        with self.lock:
            return self.current_state

    def get_remaining_time(self):
        with self.lock:
            current_time = time.time()
            elapsed = current_time - self.state_start_time

            if self.current_state == "red":
                return max(0, self.red_duration - elapsed)
            elif self.current_state == "green":
                return max(0, self.green_duration - elapsed)
            elif self.current_state == "yellow":
                return max(0, self.yellow_duration - elapsed)

def check_coordination_rules(fire1_left, fire2, fire3):
    """Check if the coordination rules are being followed"""
    fire1_left_state = fire1_left.get_state()
```



```python
def check_coordination_rules(fire1_left, fire2, fire3):
    """Check if the coordination rules are being followed"""
    fire1_left_state = fire1_left.get_state()
    fire2_state = fire2.get_state()
    fire3_state = fire3.get_state()

    violations = []

    # Rule 1: When fire 1 turning left is green, fire 2 and fire 3 should be red
    if fire1_left_state == "green" and (fire2_state != "red" or fire3_state != "red"):
        violations.append("Fire 1 left green but Fire 2 or Fire 3 not red")

    # Rule 2: When fire 2 is green, fire 1 turning left and fire 3 should be red
    if fire2_state == "green" and (fire1_left_state != "red" or fire3_state != "red"):
        violations.append("Fire 2 green but Fire 1 left or Fire 3 not red")

    # Rule 3: When fire 3 is green, fire 2 and fire 1 turning left should be red
    if fire3_state == "green" and (fire2_state != "red" or fire1_left_state != "red"):
        violations.append("Fire 3 green but Fire 2 or Fire 1 left not red")

    return violations

def determine_rolling_cars(fire1_straight, fire1_left, fire2, fire3):
    """Determine which cars can roll based on traffic light states"""
    rolling_cars = []

    if fire1_straight.get_state() == "green":
        rolling_cars.append("Fire 1 Straight")

    if fire1_left.get_state() == "green":
        rolling_cars.append("Fire 1 Left Turn")

    if fire2.get_state() == "green":
        rolling_cars.append("Fire 2")

    if fire3.get_state() == "green":
        rolling_cars.append("Fire 3")

    return rolling_cars

def main():
    print("Traffic Light Simulation Starting...")
    print("=" * 60)

    # Initialize traffic lights based on specifications
    # Fire 2: red to green 1min, green to yellow 23s, yellow to red 3s
    fire2 = TrafficLight("Fire 2", 60, 23, 3)
```



```python
def main():
    print("Traffic Light Simulation Starting...")
    print("="*60)

    # Initialize traffic lights based on specifications
    # Fire 2: red to green 1min, green to yellow 23s, yellow to red 3s
    fire2 = TrafficLight("Fire 2", 60, 23, 3)

    # Fire 1 straight: green to yellow 50s, yellow to red 3s, red to green 33s
    fire1_straight = TrafficLight("Fire 1 Straight", 33, 50, 3)

    # Fire 1 turning left: green to yellow 20s, red to green 15s, yellow to red 3s
    fire1_left = TrafficLight("Fire 1 Left", 15, 20, 3)

    # Fire 3: red to green 53s, green to yellow 29s, yellow to red 3s
    fire3 = TrafficLight("Fire 3", 53, 29, 3)

    # Set initial states (stagger the start times to match coordination)
    fire1_straight.current_state = "green"  # Fire 1 straight is not bothered by coordination
    fire1_left.current_state = "red"
    fire2.current_state = "red"
    fire3.current_state = "red"

    start_time = time.time()

    try:
        while True:
            # Update all traffic light states
            fire1_straight.update_state()
            fire1_left.update_state()
            fire2.update_state()
            fire3.update_state()

            # Get current timestamp
            current_time = datetime.now().strftime("%H:%M:%S")

            # Determine which cars are rolling
            rolling_cars = determine_rolling_cars(fire1_straight, fire1_left, fire2, fire3)

            # Check coordination rules
            violations = check_coordination_rules(fire1_left, fire2, fire3)

            # Display current status
            print(f"\n[{current_time}] Traffic Light Status:")
            print(f"  Fire 1 Straight: {fire1_straight.get_state().upper():<7} (remaining: {fire1_straight.get_remaining_time():.1f}s)")
            print(f"  Fire 1 Left:     {fire1_left.get_state().upper():<7} (remaining: {fire1_left.get_remaining_time():.1f}s)")
            print(f"  Fire 2:          {fire2.get_state().upper():<7} (remaining: {fire2.get_remaining_time():.1f}s)")
```



```python
            print(f"  Fire 3:          {fire3.get_state().upper():<7} (remaining: {fire3.get_remaining_time():.1f}s)")

            if rolling_cars:
                print(f"  🚗 CARS ROLLING: {', '.join(rolling_cars)}")
            else:
                print(f"  🔴 NO CARS ROLLING")

            if violations:
                print(f"  ⚠️  COORDINATION VIOLATIONS: {'; '.join(violations)}")
            else:
                print(f"  ✅ COORDINATION OK")

            print("-" * 60)

            # Update every second
            time.sleep(1)

    except KeyboardInterrupt:
        print("\n\nSimulation stopped by user.")
        print("Total runtime: {:.1f} seconds".format(time.time() - start_time))

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

12. Test Cases

Valid 1: gN yN rN pN gE yE rE pE gS yS rS pS gW yW rW pW

Valid 2: Two concatenated cycles

Invalid 1: Missing pN

Invalid 2: Partial cycle length mismatch

13. Conclusion

We designed automata, grammars, and a Python lexer for a 4-way intersection model. Report included REs, NFA→DFA, RLG, CFG, derivations, simplification, code, and test cases.