CONSTRUCTION COMPILER LAB TERMINAL (Q1-5)

SUBMITTED BY : FAJAR AAMIR SHEIKH

REGISTRATION NO: SP22-BCS-031

SUBMITTED TO : SIR BILAL BUKHARI

SUBMISSION DATE: 18TH JUNE2025

QUESTION NO 1:

Briefly explain your project

Project Overview

- A **Mini Compiler** is a simplified tool that mimics the functioning of a real-world compiler for educational or prototype purposes.
- It takes a small, structured subset of a programming language as input and translates it into an intermediate representation or detects syntax/semantic errors.
- This project helps students understand the key phases of compilation such as lexical analysis, syntax parsing, and semantic validation.

Key Objectives

- To develop a tool that can read and analyze input code.
- To simulate the basic phases of a compiler.
- To identify and display syntax or semantic errors.
- To provide intermediate output such as token streams or parse trees.
- To enhance understanding of compiler design concepts.

Phases of Compilation (Mini Compiler)

Phase	Description
Lexical Analysis	Tokenization using finite automata or regex
Syntax Analysis	Grammar rules via parsers (LL(1), Bottom-Up)
Semantic Analysis	Type checks, undeclared variables, etc.
Intermediate Code	Generation of simplified pseudo-code
Symbol Table	Tracks variables, types, scopes

Advantages

- Demonstrates **compiler phases in isolation**.
- Encourages **modular design** and clear thinking about language rules.
- Provides **error diagnostics** for code input.

Limitations

- Only supports a **small subset** of language grammar.
- Not capable of **full code optimization** or **machine code generation**.

QUESTION NO 2:

Explain any 2 analysis functionalities along with screenshots (function code +output)

Functionality 1: Lexical Analysis (Token Generation) Purpose:

Breaks input source code into valid tokens: keywords, identifiers, literals, operators, etc.

CODE:

```
using System;
using System.Text.RegularExpressions;

0 references
class SyntaxAnalyzer
{
    1 reference
    public static void ParseIfStatement(string code)
    {
        string pattern = @"^if\s*\(\s*[a-zA-Z_]\w*\s*[><=]\s*\d+\s*\)\s*\{\s*[a-zA-Z_]\w*\s*=\s*[a-zA-
        if (Regex.IsMatch(code, pattern))
        {
            Console.WriteLine(" Syntax is valid: IF statement is correct.");
        }
        else
        {
            Console.WriteLine(" Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
```

```
        else
        {
            Console.WriteLine(" Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
        Console.WriteLine("Enter an if-statement (e.g., if (x > 5) { x = x + 1; }):");
        string input = Console.ReadLine();

        ParseIfStatement(input);
    }
}
```

OUTPUT:

```
Microsoft Visual Studio Debu

=== Tokens ===
[1] => int
[Keyword] => int
[Identifier] => x
[Operator] => =
[Number] => 10
[Punctuation] => ;
[1] => if
[Keyword] => if
[Punctuation] => (
[Identifier] => x
[Operator] => >
[Number] => 5
[Punctuation] => )
[Punctuation] => {
[Identifier] => x
[Operator] => =
[Identifier] => x
[Operator] => +
[Number] => 1
[Punctuation] => ;
[Punctuation] => }

C:\Users\HP\source\repos\ConsoleApp3\Q2\Q2\bin\Debug\net8.0\Q2.exe (process 9732) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically clo
ging stops.
Press any key to close this window . . .
```

<u>Functionality 2: Syntax Analysis</u> (Simple Parser - IF Statement Validation) Purpose:

Validate structure of simple if statements using regular expression parsing logic.

CODE:

```
using System;
using System.Text.RegularExpressions;

0 references
class SyntaxAnalyzer
{
    1 reference
    public static void ParseIfStatement(string code)
    {
        string pattern = @"^if\s*\(\s*[a-zA-Z_]\w*\s*[><=]\s*\d+\s*\)\s*\{\s*[a-zA-Z_]\w*\s*=\s*[a-zA-

        if (Regex.IsMatch(code, pattern))
        {
            Console.WriteLine("☑ Syntax is valid: IF statement is correct.");
        }
        else
        {
            Console.WriteLine("❌ Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
```

```
        else
        {
            Console.WriteLine("❌ Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
        Console.WriteLine("Enter an if-statement (e.g., if (x > 5) { x = x + 1; }):");
        string input = Console.ReadLine();

        ParseIfStatement(input);
    }
}
```

OUTPUT:

```
Microsoft Visual Studio Debu

Enter an if-statement (e.g., if (x > 5) { x = x + 1; }):
if (x > 5) { x = x + 1; }
? Syntax is valid: IF statement is correct.
```

QUESTION NO 3:

For any given input give detail of how you arrive at the output.(attach relevant code segements and give screenshot of input and output)

1. Lexical Analysis (Tokenization)

The Lexer class reads the characters and converts them into meaningful **tokens**

```
minicompiler2 | MiniCompiler.Program | Main()

public Token NextToken()
{
    while (char.IsWhiteSpace(Current)) _pos++;

    int start = _pos;

    if (char.IsLetter(Current))
    {
        while (char.IsLetterOrDigit(Current)) _pos++;
        string word = _input.Substring(start, _pos - start);
        return new Token(word == "int" ? TokenType.Keyword : TokenType.Identifier, word, star
    }

    if (char.IsDigit(Current))
    {
        while (char.IsDigit(Current)) _pos++;
        return new Token(TokenType.Number, _input.Substring(start, _pos - start), start);
    }

    if ("=+-*/;".Contains(Current))
    {
        return new Token(TokenType.Operator, _input[_pos++].ToString(), start);
    }
```

```
            if ("=+-*/;".Contains(Current))
            {
                return new Token(TokenType.Operator, _input[_pos++].ToString(), start);
            }

            if (Current == '\0')
                return new Token(TokenType.EOF, "", _pos);

            throw new Exception($"Lexical Error at position {_pos}: Invalid character '{Current}'");
        }
    }
```

OUTPUT:

```
--- Compilation Start ---

########## Line 1 ##########
[Input] int x = 5;

+------------------+
| Lexical Analysis |
+------------------+
| Keyword: int     |
| Identifier: x    |
| Operator: =      |
| Number: 5        |
| Operator: ;      |
+------------------+
```

These are the tokens extracted from the input line.

2. Syntax Analysis

The Parser.ParseAssignment() method expects a strict format:

```
    1 reference
    public void ParseAssignment()
    {
        Eat(TokenType.Keyword);                // int
        VariableName = _current.Value;
        Eat(TokenType.Identifier);             // x
        Eat(TokenType.Operator);               // =
        VariableValue = _current.Value;
        Eat(TokenType.Number);                 // 5
        Eat(TokenType.Operator);               // ;
    }
}

1 reference
```

OUTPUT:

```
+------------------------------+
| Syntax Analysis              |
+------------------------------+
| Parsed assignment: int x = 5; |
+------------------------------+
```

The parser confirms that the syntax is correct and extracts:

- VariableName = x

- VariableValue = 5

3. Semantic Analysis & Symbol Table

The SymbolTable class:

- Declares the variable x as int
- Checks that x is declared before usage
- Ensures correct type usage

```
// Phase 3 & 7: Semantic Analysis + Symbol Table
symbolTable.Declare(parser.VariableName, "int");
symbolTable.Check(parser.VariableName, "int");
PrintBox("Semantic Analysis & Symbol Table", new List<string> {
    $"Variable '{parser.VariableName}' declared as 'int'",
    $"Type check passed for '{parser.VariableName}'"
});
```

OUTPUT:

```
+-----------------------------------+
| Semantic Analysis & Symbol Table |
+-----------------------------------+
| Variable 'x' declared as 'int'    |
| Type check passed for 'x'         |
+-----------------------------------+
```

4. Optimization (Constant Folding)

The Optimizer does basic constant folding:

```
1 reference
public class Optimizer
{
    1 reference
    public string ConstantFold(string expr)
    {
        if (expr == "2 + 3") return "5"; // Example folding
        return expr;
    }
}
```

OUTPUT:

```
+-------------------------+
| Optimization            |
+-------------------------+
| No optimization applied |
+-------------------------+
```

5. Intermediate Code Generation (IR)

The IRGenerator generates two lines of intermediate code:

```
1 reference
public class IRGenerator
{
    1 reference
    public List<string> Generate(string id, string value)
    {
        return new List<string> {
            $"t1 = {value}",
            $"{id} = t1"
        };
    }
}
```

OUTPUT:

```
+-----------------------------+
| Intermediate Code Generation |
+-----------------------------+
| t1 = 5                      |
| x = t1                      |
+-----------------------------+
```

Intermediate Representation (IR) helps abstract away hardware for later stages.

6. Target Code Generation

The TargetCodeGenerator translates it into target-level pseudo code:

```
1 reference
public class TargetCodeGenerator
{
    1 reference
    public List<string> Generate(string id, string value)
    {
        return new List<string> {
            $"LOAD {value}",
            $"STORE {id}"
        };
    }
}
```

OUTPUT:

```
+------------------------+
| Target Code Generation |
+------------------------+
| LOAD 5                 |
| STORE x                |
+------------------------+

Line compiled successfully ?
```

Line Compiled successfully.

QUESTION NO 4:

What Challenges Did You Face During the Project?

Developing a mini compiler, even for a limited scope, involved several challenges across different phases. These challenges were both **technical** and **logical**, especially since the compiler mimics the behavior of real-world compilers in a simplified form.

1. Lexical Analysis Complexity

- **Challenge:** Designing a clean and minimal lexer that correctly identifies keywords, identifiers, numbers, operators, and handles invalid characters.
- **Issue Faced:** At first, unexpected characters caused the program to crash due to lack of error handling in Lexer.NextToken().
- **How Resolved:** Introduced throw new Exception() to catch and display meaningful error messages for invalid characters.

2. Syntax Validation Logic

- **Challenge:** Ensuring only valid assignment statements like int x = 5; are accepted.
- **Issue Faced:** Incorrect ordering of tokens (e.g., missing ; or misplacing =) was hard to debug.
- **How Resolved:** Implemented a strict Eat(TokenType) method and added detailed position-based error messages to help identify syntax issues.

3. Symbol Table & Semantic Checks

- **Challenge:** Handling redeclaration and type mismatches using a Dictionary.
- **Issue Faced:** Accidentally allowed redeclaration of variables multiple times.
- **How Resolved:** Added logic to SymbolTable.Declare() to check for existing entries and throw a semantic error if redeclared.

4. No User Input During Syntax Phase

- **Challenge:** Syntax analyzer directly used tokens without re-prompting the user.
- **Issue Faced:** Users expected the compiler to ask for corrections, but the syntax analyzer didn't allow input correction.
- **How Resolved:** This was accepted as a limitation in a mini compiler. Compilation stops with a helpful error message instead.

5. Manual Intermediate & Target Code Generation

- **Challenge:** Designing logic to convert parsed variables into intermediate and target code without a real backend.
- **Issue Faced:** Mapping every variable to temporary values (t1, etc.) while keeping it readable.
- **How Resolved:** Simplified the IR and target code formats to readable strings:
  - IR → t1 = value, x = t1 ○

    Target → LOAD value, STORE x

6. Debugging and Testing Multiple Lines

- **Challenge:** Allowing users to enter 5 lines of code and parsing each independently.
- **Issue Faced:** If one line failed, the whole compilation process stopped.
- **How Resolved:** Used a loop to compile each line separately, displaying errors line-byline, while allowing others to proceed.

7. Optimization Limitations

- **Challenge:** Implementing general-purpose constant folding.
- **Issue Faced:** Only hardcoded case (2 + 3) worked.
- **How Resolved:** Left as a placeholder for future extension with expression evaluation logic.

QUESTION NO 5:

Design a Domain-Specific Language (DSL) in C# to define and generate gameplay elements like police units, criminal waves, backup support, and city levels for a dynamic police shooter game.

DSL Specification

Purpose

This Domain-Specific Language (DSL) defines the gameplay elements of a police shooter game: police units, criminal waves, backup support, zones, and objectives.

DSL Keywords & Syntax

Keyword	Description	Syntax Example
level	Level title	level Downtown Standoff
zone	Game map location (Downtown, Suburbs, Industrial)	zone Downtown
difficulty	Level difficulty (Easy, Medium, Hard)	difficulty Medium
time	Time limit in minutes	time limit 15
unit	Police unit definition	unit SWAT 2 150 80 Rifle
backup	Backup unit configuration	backup Helicopter 60 OnDemand
objective	Game or wave objective	objective Defeat all criminals
wave	Start of a new criminal wave	wave 1
criminals	Criminal group in a wave	criminals Gangster 4 80 50 Shotgun
trigger	Trigger condition to spawn wave	trigger Immediate

Example DSL Script

```
level Downtown Standoff
zone Downtown
difficulty Medium
time limit 15

unit Patrol 4 100 60 Pistol
unit SWAT 2 150 80 Rifle

backup SWAT 30 AutoWhenLow
backup Helicopter 60 OnDemand

objective Defeat all criminals
objective Protect civilians
```

```
wave 1
criminals Thief 8 50 30 Pistol
criminals Gangster 4 80 50 Shotgun
trigger Immediate
objective Stop initial assault

wave 2
criminals Gangster 6 100 60 Rifle
trigger PreviousDefeated
objective Secure area
```

Parser Implementation

- Function: Parse(string script)
- Responsibility: Parses a DSL script and builds an in-memory Level object.

Key Elements Parsed:

- Level → Name, Zone, Difficulty, Time
- Units → PoliceUnit objects with stats
- Backups → Backup records with arrival delay and condition
- Objectives → Global objectives

- Waves → Multiple criminal groups, each with a trigger and objective

Example Object Created:

```
new PoliceUnit(UnitType.SWAT, 2, 150, 80, Weapon.Rifle)
```

Interpreter

The interpreter executes the game logic based on the parsed Level object:

Key Functions:

- Load() – Initializes level data
- Run() – Runs the game loop
- SpawnWave() – Displays a wave when triggered
- Attack() – Simulates attacking criminal waves
- CallBackup() – Adds backup units to player's team
- Update() – Reduces health of both sides each round

Input Commands:

Key	Action
A	Attack
B	Call Backup
H	Show Help
X	Surrender

Game Prototype

While the current prototype is console-based, the structure allows easy expansion to a 2D or 3D game engine like Unity or Godot:

- Units and Waves are data-driven.
- Level design is fully dynamic via DSL.
- Could be plugged into a Unity prefab spawner.

liceDSLGame | PoliceShooterDSL.Zone | Suburbs

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace PoliceShooterDSL
{
    7 references
    public enum UnitType { Patrol, SWAT, Helicopter, K9 }
    2 references
    public enum CriminalType { Thief, Gangster, Sniper, Boss }
    7 references
    public enum Weapon { Pistol, Shotgun, Rifle, SniperRifle, MachineGun }
    3 references
    public enum Zone { Downtown, Suburbs, Industrial }
    3 references
    public enum Difficulty { Easy, Medium, Hard }

    4 references
    public record PoliceUnit(UnitType Type, int Count, int Health, int Accuracy, Weapon Weapon)
    {
        0 references
        public override string ToString() => $"{Count} {Type}s (H:{Health} A:{Accuracy}% W:{Weapon})"
    }
```

liceDSLGame | PoliceShooterDSL.Parser | Parse(string script)

```
    }
    2 references
    public record CriminalGroup(CriminalType Type, int Count, int Health, int Aggression, Weapon Weap
    {
        0 references
        public override string ToString() => $"{Count} {Type}s (H:{Health} A:{Aggression}% W:{Weapon}
    }
    3 references
    public record Backup(UnitType Type, int Delay, string Condition)
    {
        0 references
        public override string ToString() => $"{Type} (Arrives in {Delay}s)";
    }
    5 references
    public record Wave(int Number, List<CriminalGroup> Groups, string Trigger, string Objective);

    4 references
    public record Level(string Name, Zone Zone, Difficulty Difficulty, int Minutes,
        List<PoliceUnit> Units, List<Backup> Backups, List<string> Objectives, List<Wave> Waves);
    1 reference
    public static class Parser
    {
        1 reference
```

PoliceDSLGame | PoliceShooterDSL.Parser | Parse(string script)

```
1 reference
31 public static Level Parse(string script)
32 {
33     var lines = script.Split('\n').Select(l => l.Trim()).Where(l => l.Length > 0 && !l.Starts
34     var level = new Level("", Zone.Downtown, Difficulty.Medium, 10,
35         new(), new(), new(), new());
36     Wave currentWave = null;
37
38     foreach (var line in lines)
39     {
40         var parts = line.Split(' ', StringSplitOptions.RemoveEmptyEntries);
41         if (parts.Length == 0) continue;
42
43         switch (parts[0].ToLower())
44         {
45             case "level": level = level with { Name = string.Join(" ", parts[1..]) }; break;
46             case "zone": level = level with { Zone = Enum.Parse<Zone>(parts[1]) }; break;
47             case "difficulty": level = level with { Difficulty = Enum.Parse<Difficulty>(parts
48             case "time": level = level with { Minutes = int.Parse(parts[2]) }; break;
49             case "unit":
50                 level.Units.Add(new(Enum.Parse<UnitType>(parts[1]), int.Parse(parts[2]),
51                 int.Parse(parts[3]), int.Parse(parts[4]), Enum.Parse<Weapon>(parts[5]))); bre
52             case "backup":
```

PoliceDSLGame | PoliceShooterDSL.Parser | Parse(string script)

```
42
43         switch (parts[0].ToLower())
44         {
45             case "level": level = level with { Name = string.Join(" ", parts[1..]) }; break;
46             case "zone": level = level with { Zone = Enum.Parse<Zone>(parts[1]) }; break;
47             case "difficulty": level = level with { Difficulty = Enum.Parse<Difficulty>(parts
48             case "time": level = level with { Minutes = int.Parse(parts[2]) }; break;
49             case "unit":
50                 level.Units.Add(new(Enum.Parse<UnitType>(parts[1]), int.Parse(parts[2]),
51                 int.Parse(parts[3]), int.Parse(parts[4]), Enum.Parse<Weapon>(parts[5]))); bre
52             case "backup":
53                 level.Backups.Add(new(Enum.Parse<UnitType>(parts[1]),
54                 int.Parse(parts[2]), parts.Length > 3 ? parts[3] : "")); break;
55             case "objective": level.Objectives.Add(string.Join(" ", parts[1..])); break;
56             case "wave":
57                 currentWave = new(int.Parse(parts[1]), new(), "", "");
58                 level.Waves.Add(currentWave); break;
59             case "criminals" when currentWave != null:
60                 currentWave.Groups.Add(new(Enum.Parse<CriminalType>(parts[1]),
61                     int.Parse(parts[2]), int.Parse(parts[3]), int.Parse(parts[4]),
62                     Enum.Parse<Weapon>(parts[5]))); break;
63             case "trigger" when currentWave != null:
64                 currentWave = currentWave with { Trigger = string.Join(" ", parts[1..]) }; br
```

PoliceDSLGame | PoliceShooterDSL.Parser | Parse(string script)

```
}
2 references
public class Game
{
    private readonly Random rnd = new();
    private Level level;
    private List<PoliceUnit> units = new();
    private int waveIndex = 0;
    private readonly List<Backup> calledBackup = new();

    1 reference
    public void Load(Level level)
    {
        this.level = level;
        units = new(level.Units);
        waveIndex = 0;
        calledBackup.Clear();

        Console.WriteLine($"=== {level.Name} ===\nZone: {level.Zone}\nDifficulty: {level.Difficul
        Console.WriteLine("\nUnits: " + string.Join(", ", units));
        Console.WriteLine("\nObjectives:\n- " + string.Join("\n- ", level.Objectives));
        Console.WriteLine("\nPress any key to start...");
        Console.ReadKey();
```

PoliceDSLGame | PoliceShooterDSL.Game | ShouldSpawn(Wave wave)

```
{
    while (!IsOver())
    {
        Console.Clear();
        Console.WriteLine($"Wave: {waveIndex}/{level.Waves.Count}\nUnits: {units.Count}\n");

        if (waveIndex < level.Waves.Count && ShouldSpawn(level.Waves[waveIndex]))
            SpawnWave(level.Waves[waveIndex++]);

        ProcessInput();
        Update();
    }
    Console.WriteLine(units.Count > 0 ? "=== MISSION COMPLETE ===" : "=== MISSION FAILED ==="
}
1 reference
private bool IsOver() => waveIndex >= level.Waves.Count || units.Count == 0;
1 reference
private bool ShouldSpawn(Wave wave) => waveIndex == 0 || wave.Trigger.Contains("Defeated");
1 reference
private void SpawnWave(Wave wave)
{
    Console.WriteLine($"\n=== WAVE {wave.Number} ===\nObjective: {wave.Objective}");
    wave.Groups.ForEach(g => Console.WriteLine($"- {g}"));
```

PoliceDSLGame | PoliceShooterDSL.Game | Attack()

```
115         }
            1 reference
116         private void ProcessInput()
117         {
118             Console.Write("\n(A)ttack (B)ackup (H)elp: ");
119             switch (Console.ReadKey().Key)
120             {
121                 case ConsoleKey.A: Attack(); break;
122                 case ConsoleKey.B: CallBackup(); break;
123                 case ConsoleKey.H: ShowHelp(); break;
124             }
125         }
            1 reference
126         private void Attack()
127         {
128             if (waveIndex == 0) return;
129             var wave = level.Waves[waveIndex - 1];
130             var damage = units.Sum(u => u.Accuracy * u.Count / 10);
131             wave.Groups.ForEach(g => g = g with { Health = Math.Max(0, g.Health - damage / wave.Group
132             Console.WriteLine($"Dealt {damage} damage!");
133             Console.ReadKey();
134         }
            1 reference
135         private void CallBackup()
```

PoliceDSLGame | PoliceShooterDSL.Game | Attack()

```
134         }
            1 reference
135         private void CallBackup()
136         {
137             var available = level.Backups.Except(calledBackup).ToList();
138             if (available.Count == 0) return;
139
140             Console.WriteLine("\nAvailable: " + string.Join(", ", available.Select((b, i) => $"{i + 1
141             if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int choice) && choice <= availa
142             {
143                 var backup = available[choice - 1];
144                 calledBackup.Add(backup);
145                 units.Add(new(backup.Type, backup.Type == UnitType.Helicopter ? 1 : rnd.Next(2, 5),
146                     100, 70 + rnd.Next(20), backup.Type switch
147                     {
148                         UnitType.SWAT => Weapon.Rifle,
149                         UnitType.Helicopter => Weapon.MachineGun,
150                         _ => Weapon.Pistol
151                     }));
152                 Console.WriteLine($"\n{backup.Type} backup arrived!");
153                 Console.ReadKey();
154             }
155         }
156
```

```
PoliceDSLGame    PoliceShooterDSL.Game    ShowHelp()
            Console.ReadKey();
        }
        1 reference
        private void Update()
        {
            if (waveIndex == 0) return;
            var wave = level.Waves[waveIndex - 1];
            var damage = wave.Groups.Sum(g => g.Aggression * g.Count / 10);
            units.ForEach(u => u = u with { Health = Math.Max(0, u.Health - damage / units.Count) });
            units.RemoveAll(u => u.Health <= 0);
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main()
        {
            const string script = @"
                level Downtown Standoff
                zone Downtown
                difficulty Medium
                time limit 15

                unit Patrol 4 100 60 Pistol
                unit SWAT 2 150 80 Rifle
```

```
PoliceDSLGame    PoliceShooterDSL.Game    ShowHelp()
                unit Patrol 4 100 60 Pistol
                unit SWAT 2 150 80 Rifle

                backup SWAT 30 AutoWhenLow
                backup Helicopter 60 OnDemand

                objective Defeat all criminals
                objective Protect civilians

                wave 1
                criminals Thief 8 50 30 Pistol
                criminals Gangster 4 80 50 Shotgun
                trigger Immediate
                objective Stop initial assault

                wave 2
                criminals Gangster 6 100 60 Rifle
                trigger PreviousDefeated
                objective Secure area";

            var level = Parser.Parse(script);
            new Game().Load(level);
            new Game().Run();
        }
    }
}
```

OUTPUT:

```
C:\Users\HP\source\repos\Co

=== Downtown Standoff ===
Time elapsed: 0.0/15 minutes
Current Wave: None/3

YOUR UNITS:
- 4 PatrolOfficer(s) - Health: 100, Accuracy: 60%, Weapon: Pistol
- 2 SWAT(s) - Health: 150, Accuracy: 80%, Weapon: AssaultRifle

AVAILABLE COMMANDS:
A - Attack enemies
B - Request backup
H - Help
X - Surrender

=== INCOMING WAVE 1 ===
OBJECTIVE: Stop the initial assault
- 8 Thief(s) appeared!
  Armed with: Pistol
  Threat level: 30/100
- 4 Gangster(s) appeared!
  Armed with: Shotgun
  Threat level: 50/100

Press any key to continue...

Enter your command: A

Your units attack for 40 total damage!
8 Thiefs took 20 damage. Remaining health: 30
4 Gangsters took 20 damage. Remaining health: 60

Press any key to continue...
```

```
=== Downtown Standoff ===
Time elapsed: 0.2/15 minutes
Current Wave: 1/3

YOUR UNITS:
- 4 PatrolOfficer(s) - Health: 78, Accuracy: 60%, Weapon: Pistol
- 2 SWAT(s) - Health: 128, Accuracy: 80%, Weapon: AssaultRifle

ENEMY FORCES:
- 8 Thief(s) - Health: 30
- 4 Gangster(s) - Health: 60

AVAILABLE COMMANDS:
A - Attack enemies
B - Request backup
H - Help
X - Surrender

Enter your command:
```

PRESS H-HELP

```
=== Downtown Standoff ===
Time elapsed: 0.0/15 minutes
Current Wave: None/3

YOUR UNITS:
- 4 PatrolOfficer(s) - Health: 100, Accuracy: 60%, Weapon: Pistol
=== POLICE SHOOTER - HELP ===

COMMANDS:
A - Attack: All your units will attack the current wave of enemies
B - Backup: Call for reinforcements (if available)
H - Help: Show this help screen
X - Surrender: Give up the mission

GAME MECHANICS:
- Each attack does damage based on your units' accuracy and count
- Enemies will counterattack after your turn
- Some backup arrives automatically when your health is low
- Complete all waves within the time limit to win

Press any key to return to game...
```

PRESS X-SURRENDER

```
ENEMY FORCES:
- 8 Thief(s) - Health: 30
- 4 Gangster(s) - Health: 60

AVAILABLE COMMANDS:
A - Attack enemies
B - Request backup
H - Help
X - Surrender

Enter your command: x
You have surrendered!

=== ENEMY COUNTERATTACK ===
8 Thiefs attack for 24 total damage!
4 Gangsters attack for 20 total damage!
```