Part B: Implementation (45 points)

```python
from enum import Enum, auto
import argparse

class LexerException(Exception):
    pass

class ParseException(Exception):
    pass

class State(Enum):
    START_OR_SPACE = auto()
    NUMBER = auto()
    IDENTIFIER = auto()
    SINGLE_CHARACTER_TOKEN = auto()
    ERROR = auto()

class TokenType(Enum):
    NUMBER = auto()
    IDENTIFIER = auto()
    PLUS = auto()
    MINUS = auto()
    MULT = auto()
    EQUALS = auto()
    CONDITIONAL = auto()
    LAMBDA = auto()
    LET = auto()
    LPAREN = auto()
    RPAREN = auto()


class Token:
    def __init__(self, tokenType, value):
        if tokenType is None:
            # For single character tokens - can only have one token type
            self.tokenType = self.getTokenTypeForSingleCharacterToken(value)
            self.value = value
        else:
            self.tokenType = tokenType
            self.value = value
    def __str__(self):
        strings = {
            TokenType.NUMBER: str(self.value),
            TokenType.IDENTIFIER: str(self.value),
            TokenType.PLUS: "+",
            TokenType.MINUS: "-",
            TokenType.MULT: "×",
            TokenType.EQUALS: "=",
            TokenType.CONDITIONAL: "?",
            TokenType.LAMBDA: "λ",
            TokenType.LET: "≜",
            TokenType.LPAREN: "(",
            TokenType.RPAREN: ")"
        }
        return strings.get(self.tokenType, None)

    def getTokenTypeForSingleCharacterToken(cls, character):
        mapping = {
            "+": TokenType.PLUS,
            "-": TokenType.MINUS,
            "×": TokenType.MULT,
            "=": TokenType.EQUALS,
            "?": TokenType.CONDITIONAL,
            "λ": TokenType.LAMBDA,
            "≜": TokenType.LET,
            "(": TokenType.LPAREN,
            ")": TokenType.RPAREN
        }
        return mapping.get(character, None)

    def __repr__(self):
        return str(self)

    def __eq__(self, other):
        if not isinstance(other, Token):
            return False
        elif self.tokenType != other.tokenType:
            return False
        elif self.tokenType == TokenType.NUMBER:
                return self.value == other.value
        elif self.tokenType == TokenType.IDENTIFIER:
                return self.value == other.value
        else:
            return True

    def __ne__(self, other):
        return not self == other


class Lexer:
    def __init__(self):
        self.numbers = {"0","1","2","3","4","5","6","7","8","9"}

        self.lowerCaseIdentifiers =
{"a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","
t","u","v","w","x","y","z"}
        self.identifiers = self.lowerCaseIdentifiers.union({x.upper() for x in
self.lowerCaseIdentifiers})

        self.operators = {"+", "×", "=", "-", "?", "λ", "≜"}
        self.formattingCharacters = {"(", ")"}

        self.singleCharacterTokenCharacters =
self.operators.union(self.formattingCharacters)

        self.whiteSpaceCharacters = {" ", "\t", "\n"}

        self.alphabet =
self.numbers.union(self.identifiers).union(self.singleCharacterTokenCharacters
).union(self.whiteSpaceCharacters)

    def createTransitionFunctions(self):
        allNonErrorStates = [State.START_OR_SPACE, State.NUMBER,
State.IDENTIFIER, State.SINGLE_CHARACTER_TOKEN]

        transitions = {}

        ## Identifiers
        for letter in self.identifiers:
            transitions[(State.IDENTIFIER, letter)] = State.IDENTIFIER
            transitions[(State.NUMBER, letter)] = State.ERROR
            transitions[(State.START_OR_SPACE, letter)] = State.IDENTIFIER
            transitions[(State.SINGLE_CHARACTER_TOKEN, letter)] =
State.IDENTIFIER
```

```
        # Numbers
        for digit in self.numbers:
            transitions[(State.NUMBER, digit)] = State.NUMBER
            transitions[(State.IDENTIFIER, digit)] = State.ERROR
            transitions[(State.START_OR_SPACE, digit)] = State.NUMBER
            transitions[(State.SINGLE_CHARACTER_TOKEN, digit)] = State.NUMBER

        # Single character tokens
        for character in self.singleCharacterTokenCharacters:
            transitions[(State.SINGLE_CHARACTER_TOKEN, character)] =
State.SINGLE_CHARACTER_TOKEN
            transitions[(State.NUMBER, character)] =
State.SINGLE_CHARACTER_TOKEN
            transitions[(State.IDENTIFIER, character)] =
State.SINGLE_CHARACTER_TOKEN
            transitions[(State.START_OR_SPACE, character)] =
State.SINGLE_CHARACTER_TOKEN

        for state in allNonErrorStates:
            transitions[(state, " ")] = State.START_OR_SPACE

        return transitions

    def GetTokensForString(self, input):
        transitionFunctionDictionary = self.createTransitionFunctions()

        allTokens = []

        numberBuffer = ""
        identifierBuffer = ""

        currentState = State.START_OR_SPACE

        for character in input:
            if character not in self.alphabet:
                raise LexerException(f"Character '{character}' is not in the
alphabet")

            newState = transitionFunctionDictionary.get((currentState,
character), State.ERROR)

            if newState == State.ERROR:
                raise LexerException(f"Transition ({currentState},
'{character}') is not valid")

            # Tokens are added when states change
            if (newState != currentState):
                if currentState == State.NUMBER:
                    allTokens.append(Token(TokenType.NUMBER,
int(numberBuffer)))
                    numberBuffer = ""
                elif currentState == State.IDENTIFIER:
                    allTokens.append(Token(TokenType.IDENTIFIER,
identifierBuffer))
                    identifierBuffer = ""

            currentState = newState

            # Tokenize any leftover buffer
            if currentState == State.SINGLE_CHARACTER_TOKEN:
                allTokens.append(Token(None, character))
            elif currentState == State.NUMBER:
                numberBuffer += character
            elif currentState == State.IDENTIFIER:
                identifierBuffer += character

        if currentState == State.NUMBER:
            allTokens.append(Token(TokenType.NUMBER, int(numberBuffer)))
        elif currentState == State.IDENTIFIER:
            allTokens.append(Token(TokenType.IDENTIFIER, identifierBuffer))

        return allTokens

class Parser:
    @classmethod
    def Parse(cls, tokens):
        parsingTable = {
            '<program>': {
                'NUMBER': ['<expr>'],
                'IDENTIFIER': ['<expr>'],
                'LPAREN': ['<expr>']
            },
            '<expr>': {
                'NUMBER': ['NUMBER'],
                'IDENTIFIER': ['IDENTIFIER'],
                'LPAREN': ['LPAREN', '<paren-expr>', 'RPAREN']
            },
            '<paren-expr>': {
                'PLUS': ['PLUS', '<expr>', '<expr>'],
                'MULT': ['MULT', '<expr>', '<expr>'],
                'EQUALS': ['EQUALS', '<expr>', '<expr>'],
                'MINUS': ['MINUS', '<expr>', '<expr>'],
                'CONDITIONAL': ['CONDITIONAL', '<expr>', '<expr>', '<expr>'],
                'LAMBDA': ['LAMBDA', 'IDENTIFIER', '<expr>'],
                'LET': ['LET', 'IDENTIFIER', '<expr>', '<expr>'],
                'NUMBER': ['<expr>', '<expr>*'],
                'IDENTIFIER': ['<expr>', '<expr>*'],
                'LPAREN': ['<expr>', '<expr>*']
            },
            '<expr>*': {
                'NUMBER': ['<expr>', '<expr>*'],
                'IDENTIFIER': ['<expr>', '<expr>*'],
                'LPAREN': ['<expr>', '<expr>*'],
                'RPAREN': []
            }
        }

        parseTreeStack = [[]] # Initial nested list is needed so that
"parseTreeStack[-1]" can always be used to the get the latest parse tree
container in the stack

        terminals = {'NUMBER', 'IDENTIFIER', 'PLUS', 'MULT', 'EQUALS',
'MINUS', 'CONDITIONAL', 'LAMBDA', 'LET', 'LPAREN', 'RPAREN', '$'}
        nonTerminals = set(parsingTable.keys())

        stack = ['$', '<program>']

        inputTokensIndex = 0
        parenthesisDepth = 0

        while len(stack) > 0:
            top = stack[-1]
            currentTokenType = tokens[inputTokensIndex].tokenType.name if
inputTokensIndex < len(tokens) else '$'


            if top == '$' or currentTokenType == '$':
                stack.pop()
                if parenthesisDepth > 0:
                    raise ParseException(f"Missing {parenthesisDepth} closing
parentheses")
            elif (top in terminals):
                if top == currentTokenType:
                    stack.pop()

                    # Since the only situation where a non-terminal is
expanded to more than one terminal is when expanding the non terminal <paren-
expr>,
                    # and <paren-expr> can ONLY appear when surrounded by
parentheses, we can safely use the parenthesis to manage the parse tree depth/
structure.
                    if (currentTokenType == 'LPAREN'):
                        parenthesisDepth += 1

                        # Add new parse tree container to the stack - will be
put into its 'parent' tree container when RPAREN is encountered
                        newParseTreeSublist = []
                        parseTreeStack.append(newParseTreeSublist)
                    elif (currentTokenType == 'RPAREN'):
                        parenthesisDepth -= 1
                        if parenthesisDepth < 0:
                            raise ParseException(f"Unmatched closing
parenthesis at position: {inputTokensIndex}")

                        # Removes the current parse tree container from the
stack and appends it to the 'parent' parse tree container
                        completedParseTreeSublist = parseTreeStack.pop()
                        parseTreeStack[-1].append(completedParseTreeSublist)
                    else:
                        parseTreeStack[-1].append(tokens[inputTokensIndex])

                    inputTokensIndex += 1
                else:
                    if currentTokenType == '$':
                        raise ParseException(f"Unexpected end of input.
Expected: {top}")
                    elif top == 'RPAREN':
                        raise ParseException(f"Expected closing parenthesis,
but found: {currentTokenType}. Wrong number of arguments")
                    else:
                        raise ParseException(f"Expected {top}, found:
{currentTokenType}")
```

```
            elif top in nonTerminals:
                # Check if there is a valid production rule for the current
non-terminal and token
                productionRules = parsingTable.get(top,
None).get(currentTokenType, None)
                if productionRules is not None:
                    stack.pop()
                    if len(productionRules) == 0: # Only case is: ('<expr>*',
')') -> []
                        pass
                    else:
                        for symbol in reversed(productionRules):
                            stack.append(symbol)
                else:
                    if currentTokenType == '$':
                            raise ParseException(f"Unexpected end of input
while parsing: {top}")
                    elif top == '<paren-expr>':
                        raise ParseException(f"Invalid expression inside
parentheses: {currentTokenType}")
                    else:
                        raise ParseException(f"Unexpected {currentTokenType}
while parsing: {top}")

        result = parseTreeStack[0]
        return result[0] if len(result) == 1 else result

class MiniLispAnalyser:
    @classmethod
    def Analyse(cls, input):
        lexer = Lexer()

        tokens = lexer.GetTokensForString(input)
        parseTree = Parser.Parse(tokens)
        return parseTree

def main():
    parser = argparse.ArgumentParser(description='MiniLisp Lexer and Parser')
    parser.add_argument('mode', choices = ['Lexer', 'Analyser'], help = 'Mode
to run: Lexer (lexer only), Analyser (full analysis)')
    parser.add_argument('input', help = 'MiniLisp expression to process')
    args = parser.parse_args()

    if args.mode == 'Lexer':
        tokens = Lexer.GetTokensForString(args.input)
        print("Tokens:", tokens)
    elif args.mode == 'Analyser':
        parseTree = MiniLispAnalyser.Analyse(args.input)
        print("Parse Tree:", parseTree)

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

Part C: Testing and Validation (15 points)

Test Cases

```
import json
import traceback
from typing import Any, Union

from Implementation import MiniLispAnalyser, Token, TokenType

def token_to_json(t: Token) -> dict[str, Any]:
    return {
        "type": t.tokenType.name,
        "value": t.value if t.tokenType in {TokenType.NUMBER,
TokenType.IDENTIFIER} else str(t)
    }

def serialize_tree(node: Union[Token, list[Any]]) -> Any:
    if isinstance(node, Token):
        return token_to_json(node)
    if isinstance(node, list):
        return [serialize_tree(n) for n in node]
    return node

def run_test_case(input_str: str, description: str, expected: dict[str, Any])
-> dict[str, Any]:
    actual = {
        "status": None,
        "parse_tree": None,
        "error": None,
        "traceback": None
    }

    try:
        result = MiniLispAnalyser.Analyse(input_str)
        actual["parse_tree"] = serialize_tree(result)
        actual["status"] = "success"
    except Exception as e:
        actual["status"] = "error"
        actual["error"] = str(e)
        actual["traceback"] = traceback.format_exc()

    passed = False
    if expected.get("status") == "success":
        passed = (actual["status"] == "success")
    else:
        if actual["status"] == "error":
            substr = expected.get("error_contains")
            passed = True if not substr else (substr in (actual["error"] or
""))

    return {
        "description": description,
        "input": input_str,
        "expected_output": expected,
        "actual_output": actual,
        "success": passed
    }

def main() -> None:
    tests = [
        # Positive tests
        {"input": "42", "desc": "NUMBER literal", "expected": {"status":
"success"}},
        {"input": "x", "desc": "IDENTIFIER literal", "expected": {"status":
"success"}},
        {"input": "(+ 2 3)", "desc": "Simple addition", "expected": {"status":
"success"}},
        {"input": "(× x 5)", "desc": "Multiplication with identifier",
"expected": {"status": "success"}},
        {"input": "(+ (× 2 3) 4)", "desc": "Nested arithmetic", "expected":
{"status": "success"}},
        {"input": "(? (= x 0) 1 0)", "desc": "Conditional expression",
"expected": {"status": "success"}},
        {"input": "(λ x x)", "desc": "Lambda identity", "expected": {"status":
"success"}},
        {"input": "(≜ y 10 y)", "desc": "Let binding", "expected": {"status":
"success"}},
        {"input": "((λ x (+ x 1)) 5)", "desc": "Lambda application",
"expected": {"status": "success"}},
        {"input": "(× (+ 1 2) (− 5 3))", "desc": "Mixed operators with unicode
minus", "expected": {"status": "success"}},
        {"input": "(λ f (λ x (f x)))", "desc": "Higher-order lambda",
"expected": {"status": "success"}},
        {"input": "(− 7 2)", "desc": "Unicode minus operator (U+2212)",
"expected": {"status": "success"}},

        # Negative tests
        {"input": "(+ 1)", "desc": "Too few args for +", "expected":
{"status": "error"}},
        {"input": "(+ 1 2 3)", "desc": "Too many args for +", "expected":
{"status": "error", "error_contains": "Expected closing parenthesis"}},
        {"input": "@", "desc": "Invalid character", "expected": {"status":
"error", "error_contains": "Character '@' is not in the alphabet"}},
        {"input": ")", "desc": "Unmatched closing parenthesis", "expected":
{"status": "error"}},
        {"input": "(? 1 2)", "desc": "Too few args for ?", "expected":
{"status": "error"}},
        {"input": "(- 7 2)", "desc": "ASCII hyphen-minus should be rejected",
"expected": {"status": "error", "error_contains": "Character '-' is not in the
alphabet"}},
        {"input": "((λ x (+ x 1)) 5", "desc": "Missing closing parenthesis",
"expected": {"status": "error"}, "error_contains": "Missing closing
parenthesis"},

        # Whitespace tolerance (positive)
        {"input": "   42   ", "desc": "Leading/trailing spaces around number",
"expected": {"status": "success"}},
        {"input": "( + 2 3 )", "desc": "Spaces between parens, operator, and
operands", "expected": {"status": "success"}},
        {"input": "(+    2    3)", "desc": "Multiple spaces between tokens",
"expected": {"status": "success"}},
        {"input": "(\n+ \n2\t3\n)", "desc": "Newlines and tabs between
tokens", "expected": {"status": "success"}},
        {"input": "(\nλ  x   x\n)", "desc": "Lambda with mixed whitespace",
"expected": {"status": "success"}},
        {"input": "(\n(λ   x   (+  x    1))\t  5\n)", "desc": "Nested
application with whitespace", "expected": {"status": "success"}},

        # Whitespace-only or empty input (negative)
        {"input": "   \n\t  ", "desc": "Whitespace-only input", "expected":
{"status": "error"}},
        {"input": "", "desc": "Empty input", "expected": {"status": "error"}},
    ]

    results = [run_test_case(t["input"], t["desc"], t["expected"]) for t in
tests]

    out_path = "test_results.json"
    with open(out_path, "w", encoding="utf-8") as f:
        json.dump(results, f, indent=2, ensure_ascii=False)
    print(f"Wrote {len(results)} test results to {out_path}")

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

Test Output

```
[
  {
    "description": "NUMBER literal",
    "input": "42",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": {
        "type": "NUMBER",
        "value": 42
      },
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "IDENTIFIER literal",
    "input": "x",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": {
        "type": "IDENTIFIER",
        "value": "x"
      },
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Simple addition",
    "input": "(+ 2 3)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "PLUS",
          "value": "+"
        },
        {
          "type": "NUMBER",
          "value": 2
        },
        {
          "type": "NUMBER",
          "value": 3
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Multiplication with identifier",
    "input": "(× x 5)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "MULT",
          "value": "×"
        },
        {
          "type": "IDENTIFIER",
          "value": "x"
        },
        {
          "type": "NUMBER",
          "value": 5
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Nested arithmetic",
    "input": "(+ (× 2 3) 4)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "PLUS",
```

```json
          "value": "+"
        },
        [
          {
            "type": "MULT",
            "value": "×"
          },
          {
            "type": "NUMBER",
            "value": 2
          },
          {
            "type": "NUMBER",
            "value": 3
          }
        ],
        {
          "type": "NUMBER",
          "value": 4
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Conditional expression",
    "input": "(? (= x 0) 1 0)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "CONDITIONAL",
          "value": "?"
        },
        [
          {
            "type": "EQUALS",
            "value": "="
          },
          {
            "type": "IDENTIFIER",
            "value": "x"
          },
          {
            "type": "NUMBER",
            "value": 0
          }
        ],
        {
          "type": "NUMBER",
          "value": 1
        },
        {
          "type": "NUMBER",
          "value": 0
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Lambda identity",
    "input": "(λ x x)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "LAMBDA",
          "value": "λ"
        },
        {
          "type": "IDENTIFIER",
          "value": "x"
        },
        {
          "type": "IDENTIFIER",
          "value": "x"
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Let binding",
    "input": "(≙ y 10 y)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "LET",
          "value": "≙"
        },
        {
          "type": "IDENTIFIER",
          "value": "y"
        },
        {
          "type": "NUMBER",
          "value": 10
        },
        {
          "type": "IDENTIFIER",
          "value": "y"
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Lambda application",
    "input": "((λ x (+ x 1)) 5)",
    "expected_output": {
        "status": "success"
      },
      "actual_output": {
        "status": "success",
        "parse_tree": [
          [
            {
              "type": "LAMBDA",
              "value": "λ"
            },
            {
              "type": "IDENTIFIER",
              "value": "x"
            },
            [
              {
                "type": "PLUS",
                "value": "+"
              },
              {
                "type": "IDENTIFIER",
                "value": "x"
              },
              {
                "type": "NUMBER",
                "value": 1
              }
            ]
          ],
          {
            "type": "NUMBER",
            "value": 5
          }
        ],
        "error": null,
        "traceback": null
      },
      "success": true
    },
    {
      "description": "Mixed operators with unicode minus",
      "input": "(× (+ 1 2) (- 5 3))",
      "expected_output": {
```

```
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "MULT",
          "value": "×"
        },
        [
          {
            "type": "PLUS",
            "value": "+"
          },
          {
            "type": "NUMBER",
            "value": 1
          },
          {
            "type": "NUMBER",
            "value": 2
          }
        ],
        [
          {
            "type": "MINUS",
            "value": "-"
          },
          {
            "type": "NUMBER",
            "value": 5
          },
          {
            "type": "NUMBER",
            "value": 3
          }
        ]
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
{
  "description": "Higher-order lambda",
  "input": "(λ f (λ x (f x)))",
  "expected_output": {
    "status": "success"
  },
  "actual_output": {
    "status": "success",
    "parse_tree": [
      {
        "type": "LAMBDA",
        "value": "λ"
      },
      {
        "type": "IDENTIFIER",
        "value": "f"
      },
      [
        {
          "type": "LAMBDA",
          "value": "λ"
        },
        {
          "type": "IDENTIFIER",
          "value": "x"
        },
        [
          {
            "type": "IDENTIFIER",
            "value": "f"
          },
          {
            "type": "IDENTIFIER",
            "value": "x"
          }
        ]
      ]
    ],
    "error": null,
    "traceback": null
  },
  "success": true
},
  {
    "description": "Unicode minus operator (U+2212)",
    "input": "(− 7 2)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "MINUS",
          "value": "-"
        },
        {
          "type": "NUMBER",
          "value": 7
        },
        {
          "type": "NUMBER",
          "value": 2
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Too few args for +",
    "input": "(+ 1)",
    "expected_output": {
      "status": "error"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Unexpected RPAREN while parsing: <expr>",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 295, in Analyse\n    parseTree =
Parser.Parse(tokens)\n  File \"/Users/bryanlee/Documents/University/Theory Of
Computing Science/Assignments/minilisp-parser/Implementation.py\", line 284,
in Parse\n    raise ParseException(f\"Unexpected {currentTokenType} while
parsing: {top}\")\nImplementation.ParseException: Unexpected RPAREN while
parsing: <expr>\n"
    },
    "success": true
  },
  {
    "description": "Too many args for +",
    "input": "(+ 1 2 3)",
    "expected_output": {
      "status": "error",
      "error_contains": "Expected closing parenthesis"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Expected closing parenthesis, but found: NUMBER. Wrong number
of arguments",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 295, in Analyse\n    parseTree =
Parser.Parse(tokens)\n  File \"/Users/bryanlee/Documents/University/Theory Of
Computing Science/Assignments/minilisp-parser/Implementation.py\", line 265,
in Parse\n    raise ParseException(f\"Expected closing parenthesis, but found:
{currentTokenType}. Wrong number of arguments\")
\nImplementation.ParseException: Expected closing parenthesis, but found:
NUMBER. Wrong number of arguments\n"
    },
    "success": true
  },
  {
    "description": "Invalid character",
    "input": "@",
    "expected_output": {
      "status": "error",
      "error_contains": "Character '@' is not in the alphabet"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Character '@' is not in the alphabet",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 294, in Analyse\n    tokens =
lexer.GetTokensForString(input)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 153, in GetTokensForString\n    raise
LexerException(f\"Character '{character}' is not in the alphabet\")
\nImplementation.LexerException: Character '@' is not in the alphabet\n"
    },
    "success": true
  },
  {
    "description": "Unmatched closing parenthesis",
    "input": ")",
    "expected_output": {
      "status": "error"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Unexpected RPAREN while parsing: <program>",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 295, in Analyse\n    parseTree =
Parser.Parse(tokens)\n  File \"/Users/bryanlee/Documents/University/Theory Of
Computing Science/Assignments/minilisp-parser/Implementation.py\", line 284,
in Parse\n    raise ParseException(f\"Unexpected {currentTokenType} while
parsing: {top}\")\nImplementation.ParseException: Unexpected RPAREN while
parsing: <program>\n"
    },
    "success": true
  },
  {
    "description": "Too few args for ?",
    "input": "(? 1 2)",
    "expected_output": {
      "status": "error"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Unexpected RPAREN while parsing: <expr>",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 295, in Analyse\n    parseTree =
Parser.Parse(tokens)\n  File \"/Users/bryanlee/Documents/University/Theory Of
Computing Science/Assignments/minilisp-parser/Implementation.py\", line 284,
in Parse\n    raise ParseException(f\"Unexpected {currentTokenType} while
parsing: {top}\")\nImplementation.ParseException: Unexpected RPAREN while
parsing: <expr>\n"
    },
    "success": true
  },
  {
    "description": "ASCII hyphen-minus should be rejected",
    "input": "(- 7 2)",
    "expected_output": {
      "status": "error",
      "error_contains": "Character '-' is not in the alphabet"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Character '-' is not in the alphabet",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 294, in Analyse\n    tokens =
lexer.GetTokensForString(input)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 153, in GetTokensForString\n    raise
LexerException(f\"Character '{character}' is not in the alphabet\")
```

```
\nImplementation.LexerException: Character '-' is not in the alphabet\n"
    },
    "success": true
  },
  {
    "description": "Missing closing parenthesis",
    "input": "((λ x (+ x 1)) 5",
    "expected_output": {
      "status": "error"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Missing 1 closing parentheses",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 295, in Analyse\n    parseTree =
Parser.Parse(tokens)\n  File \"/Users/bryanlee/Documents/University/Theory Of
Computing Science/Assignments/minilisp-parser/Implementation.py\", line 236,
in Parse\n    raise ParseException(f\"Missing {parenthesisDepth} closing
parentheses\")\nImplementation.ParseException: Missing 1 closing
parentheses\n"
    },
    "success": true
  },
  {
    "description": "Leading/trailing spaces around number",
    "input": "   42   ",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": {
        "type": "NUMBER",
        "value": 42
      },
      "error": null,
      "traceback": null
    },
      "success": true
    },
    {
      "description": "Spaces between parens, operator, and operands",
      "input": "( + 2 3 )",
      "expected_output": {
        "status": "success"
      },
      "actual_output": {
        "status": "success",
        "parse_tree": [
          {
            "type": "PLUS",
            "value": "+"
          },
          {
            "type": "NUMBER",
            "value": 2
          },
          {
            "type": "NUMBER",
            "value": 3
          }
        ],
        "error": null,
        "traceback": null
      },
      "success": true
    },
    {
      "description": "Multiple spaces between tokens",
      "input": "(+    2    3)",
      "expected_output": {
        "status": "success"
      },
      "actual_output": {
        "status": "success",
        "parse_tree": [
          {
            "type": "PLUS",
            "value": "+"
          },
          {
```

```
          "type": "NUMBER",
          "value": 2
        },
        {
          "type": "NUMBER",
          "value": 3
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Newlines and tabs between tokens",
    "input": "(\n+ \n2\t3\n)",
    "expected_output": {
      "status": "success"
    },
    "actual_output": {
      "status": "success",
      "parse_tree": [
        {
          "type": "PLUS",
          "value": "+"
        },
        {
          "type": "NUMBER",
          "value": 2
        },
        {
          "type": "NUMBER",
          "value": 3
        }
      ],
      "error": null,
      "traceback": null
    },
    "success": true
  },
  {
    "description": "Lambda with mixed whitespace",
    "input": "(\nλ  x   x\n)",
      "expected_output": {
        "status": "success"
      },
      "actual_output": {
        "status": "success",
        "parse_tree": [
          {
            "type": "LAMBDA",
            "value": "λ"
          },
          {
            "type": "IDENTIFIER",
            "value": "x"
          },
          {
            "type": "IDENTIFIER",
            "value": "x"
          }
        ],
        "error": null,
        "traceback": null
      },
      "success": true
    },
    {
      "description": "Nested application with whitespace",
      "input": "(\n(λ   x   (+  x    1))\t  5\n)",
      "expected_output": {
        "status": "success"
      },
      "actual_output": {
        "status": "success",
        "parse_tree": [
          [
            {
              "type": "LAMBDA",
              "value": "λ"
            },
            {
              "type": "IDENTIFIER",
              "value": "x"
            },
            [
          {
            "type": "PLUS",
            "value": "+"
          },
          {
            "type": "IDENTIFIER",
            "value": "x"
          },
          {
            "type": "NUMBER",
            "value": 1
          }
        ]
      ],
      {
        "type": "NUMBER",
        "value": 5
      }
    ],
    "error": null,
    "traceback": null
  },
  "success": true
},
{
  "description": "Whitespace-only input",
  "input": "   \n\t  ",
  "expected_output": {
    "status": "error"
  },
  "actual_output": {
    "status": "error",
    "parse_tree": null,
    "error": "Input is empty or contains only whitespace",
    "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 294, in Analyse\n    tokens =
lexer.GetTokensForString(input)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 140, in GetTokensForString\n    raise
```

```
LexerException(\"Input is empty or contains only whitespace\")
\nImplementation.LexerException: Input is empty or contains only whitespace\n"
    },
    "success": true
  },
  {
    "description": "Empty input",
    "input": "",
    "expected_output": {
      "status": "error"
    },
    "actual_output": {
      "status": "error",
      "parse_tree": null,
      "error": "Input is empty or contains only whitespace",
      "traceback": "Traceback (most recent call last):\n  File \"/Users/
bryanlee/Documents/University/Theory Of Computing Science/Assignments/
minilisp-parser/test_cases.py\", line 29, in run_test_case\n    result =
MiniLispAnalyser.Analyse(input_str)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 294, in Analyse\n    tokens =
lexer.GetTokensForString(input)\n  File \"/Users/bryanlee/Documents/
University/Theory Of Computing Science/Assignments/minilisp-parser/
Implementation.py\", line 140, in GetTokensForString\n    raise
LexerException(\"Input is empty or contains only whitespace\")
\nImplementation.LexerException: Input is empty or contains only whitespace\n"
    },
    "success": true
  }
]
```

How to run our parser

Implementation

The implementation python file takes a few arguments to run, in sequential order:

- Mode: Lexer or Analyser mode. Lexer mode runs only the lexer and returns tokens while Analyser mode runs both the lexer and parser and returns the parse tree.
- Input: The MiniLisp string to lex or parse

Example commands:

```
python Implementation.py Lexer "(+ x y)"
python Implementation.py Analyser "(+ x y)"
```

Test Cases

To run the existing test cases, run the test cases python file without arguments.

New test cases can be testing manually using the implementation python file or added to the test case list in the test case python file, which just runs the implementation.