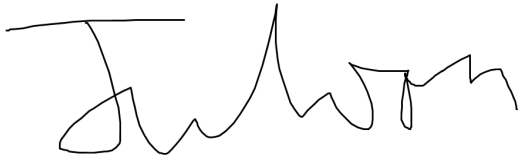
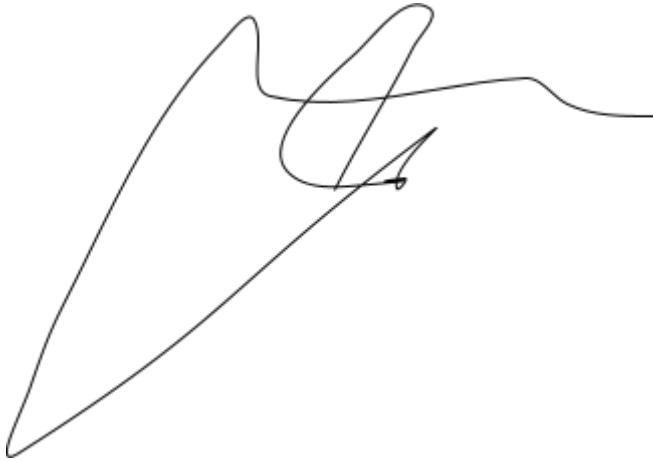


ASSIGNMENT COVERSHEET

UTS: ENGINEERING & INFORMATION TECHNOLOGY		
SUBJECT NUMBER & NAME 41080 Theory Of Computing Science	NAME OF STUDENT(s) (PRINT CLEARLY) Julian Kirk Bryan Lee	STUDENT ID(s) 25268741 25495108
STUDENT EMAIL Julian.G.Kirk@student.uts.edu.au bryan.lee-1@student.uts.edu.au		STUDENT CONTACT NUMBER 0402 482 785 0435 157 931
NAME OF TUTOR Tutorial 1	TUTORIAL GROUP Tutorial 1	DUE DATE 3/11/2025
ASSESSMENT ITEM NUMBER & TITLE Assessment Task 2 – Group Programming Assignment		
<p> <input checked="" type="checkbox"/> I confirm that I have read, understood and followed the guidelines for assignment submission and presentation on page 2 of this cover sheet. <input checked="" type="checkbox"/> I confirm that I have read, understood and followed the advice in the Subject Outline about assessment requirements. <input checked="" type="checkbox"/> I understand that if this assignment is submitted after the due date it may incur a penalty for lateness unless I have previously had an extension of time approved and have attached the written confirmation of this extension. </p> <p> Declaration of originality: The work contained in this assignment, other than that specifically attributed to another source, is that of the author(s) and has not been previously submitted for assessment. I understand that, should this declaration be found to be false, disciplinary action could be taken and penalties imposed in accordance with University policy and rules. In the statement below, I have indicated the extent to which I have collaborated with others, whom I have named. </p> <p> Statement of collaboration: This assignment was completed as a group effort by Julian Kirk and Bryan Lee. Bryan was responsible for Part A (Grammar Analysis) and Part C.1 (Test Cases), while Julian was responsible for Part B (implementation) and Part C.2 (Error Handling). </p> <div style="text-align: center;">   </div>		
Signature of student(s) _____		Date 02/11/2025



ASSIGNMENT RECEIPT

To be completed by the student if a receipt is required

SUBJECT NUMBER & NAME 41080 Theory Of Computing Science	NAME OF TUTOR	
SIGNATURE OF TUTOR		RECEIVED DATE

STYLE GUIDE for ASSIGNMENT SUBMISSION

Before submitting an assignment, you should refer to the policies and guidelines set out in the following:

- [FEIT Student Guide](#)
- [UTS Library - referencing](#)
- [HELPS - English and academic literacy support](#)
- [UTS GSU - coursework assessment policy and procedures](#)

Unless your Subject Coordinator has indicated otherwise in the Subject Outline, you must follow the instructions below for submission of assignments in the Faculty of Engineering and Information Technology.

Writing style

It is usually best to write your initial draft in the default settings of your software without formatting. Use the following guides in your writing.

Purpose and audience: use the correct genre and language style expected for the particular task.

Language: use 'plain English' for all technical writing. More information about this language style can be found at www.plainenglish.co.uk/free-guides.html.

Use spelling and grammar software tools to check your writing. Edit your document.

Standards: always use:

- Australian spelling standards (Macquarie Dictionary)
- SI (International System of Units) units of measurement
- ISO (International Organisation for Standardisation) for writing dates and times for international documents. For example **yyyy-mm-dd** or **hh-mm-ss**. However, for most applications it is more helpful to present the date in full as **26 August 2016**.

Graphics and tables should:

- be numbered
- have an appropriate heading and/or caption
- be fully labelled
- be correctly referenced.

Presentation

Unless otherwise instructed, all assignment submissions should be **word processed** using spell-check and grammar-check software. Work should be well **edited** before submission. Use the following default settings:

Page setup: set margins at no less than 20mm all around.

Paper: print on A4 bond, double-spaced and preferably double-sided, left justified.

Font: use the software default style to provide consistency. The recommended style includes:

- 10-12 pt font
- consistent formatting with a limited number of fonts
- lines no more than 60 characters (use wider margins or columns if you need to make lines shorter)

Header should include:

- your name and student number
- the title of the paper or task.

Footer should include the page number and current date.

Cover sheet and statement of originality: all work submitted for assessment must be the original work of the student(s) submitting the work. A standard faculty cover sheet (see over) must be attached to the front of the submission. Any collaboration between the submitting student and others must be declared on the cover sheet.

Referencing

All sources of information used in the preparation of your submission must be acknowledged using the Harvard system of referencing. This includes all print, video, electronic sources.

Phrases, sentences or paragraphs taken verbatim from a source must be in quotation marks and the source(s) cited using both **in-text** referencing and a **reference list**.

Plagiarism is the failure to acknowledge sources of information. You should be fully aware of the meaning of plagiarism and its consequences both to your marks, position at the university and criminal liability. The plagiarism in your assignment submissions can be assessed both in hard copy and in soft copy through software such as Turnitin.

The UTS Library and UTS HELPS (web links above) provide extensive information for students on referencing correctly to support you in avoiding plagiarism.

MiniLisp Grammar Analysis

Author: Bryan Lee

Purpose: This notebook examines the MiniLisp grammar to verify that it satisfies the LL(1) property. Through analysing its FIRST and FOLLOW sets, constructing the parse table, and testing for conflicts, it becomes clear that the grammar is both precise and deterministic.

Grammar:

```
<program>      ::= <expr>

<expr>         ::= NUMBER
                  | IDENTIFIER
                  | '(' <paren-expr> ')'

<paren-expr>   ::= '+' <expr> <expr>
                  | 'x' <expr> <expr>
                  | '=' <expr> <expr>
                  | '-' <expr> <expr>
                  | '?' <expr> <expr> <expr>
                  | 'λ' IDENTIFIER <expr>
                  | 'Δ' IDENTIFIER <expr> <expr>
                  | <expr> <expr>*
```

In [18]:

```
grammar = {
    '<program>': [['<expr>']],
    '<expr>': [['NUMBER'], ['IDENTIFIER'], ['(', '<paren-expr>', ')']],
    '<paren-expr>': [
        ['+', '<expr>', '<expr>'],
        ['x', '<expr>', '<expr>'],
        ['=', '<expr>', '<expr>'],
        ['-', '<expr>', '<expr>'],
        ['?', '<expr>', '<expr>', '<expr>'],
        ['λ', 'IDENTIFIER', '<expr>'],
        ['Δ', 'IDENTIFIER', '<expr>', '<expr>'],
        ['<expr>', '<expr>*']
    ],
    '<expr>*': [['<expr>', '<expr>*'], []]
}

terminals = {'NUMBER', 'IDENTIFIER', '+', 'x', '=', '-', '?', 'λ', 'Δ', '(', ')', '$'}
non_terminals = set(grammar.keys())

non_terminals
```

Out[18]:

```
{'<expr>', '<expr>*', '<paren-expr>', '<program>'}
```

A.1: LL(1) Property Validation (25 points)

The first part of this analysis focuses on proving that the MiniLisp grammar is LL(1). In simple terms, this means the grammar can be parsed by looking only one symbol ahead at any time, with no confusion about which rule to choose.

FIRST Sets

To begin verifying that the grammar satisfies LL(1) properties, I first derived the FIRST sets for each non-terminal. The FIRST set of a non-terminal shows which symbols can appear at the start of any string derived from that symbol. This step is crucial because it forms the foundation of predictive parsing, telling the parser what kind of token it should expect first.

In [19]:

```
FIRST = {nt: set() for nt in non_terminals}

changed = True

while changed:
    changed = False

    for nt in non_terminals:
        for production in grammar[nt]:
            can_be_empty = True

            for token in production:
                if token in terminals:
                    before = len(FIRST[nt])

                    FIRST[nt].add(token)

                    if len(FIRST[nt]) != before:
                        changed = True

            can_be_empty = False
            break
        elif token in non_terminals:
            before = len(FIRST[nt])
            FIRST[nt] |= (FIRST[token] - {'ε'})

            if len (FIRST[nt]) != before:
```

```
changed = True

if 'ε' not in FIRST[token]:
    can_be_empty = False
    break

if can_be_empty:
    before = len(FIRST[nt])
    FIRST[nt].add('ε')

if len(FIRST[nt]) != before:
    changed = True
```

FIRST

Out[19]:

```
{'<program>': {'(', 'IDENTIFIER', 'NUMBER'},
 '<expr>': {'(', 'IDENTIFIER', 'NUMBER'},
 '<paren-expr>': {'(',
 '+',
 '=',
 '?',
 'IDENTIFIER',
 'NUMBER',
 '×',
 'λ',
 '-',
 'Δ'},
 '<expr>*': {'(', 'IDENTIFIER', 'NUMBER', 'ε'}}
```

The process followed a structured, rule-based approach:

1. If a production begins with a terminal (for example, NUMBER or IDENTIFIER), that terminal is immediately added to the FIRST set of the non-terminal.
2. If the production begins with another non-terminal, I recursively added that non-terminal’s FIRST set (excluding ε) to the current one.
3. If a symbol could derive ε, the algorithm continued to the next symbol in the production, ensuring all possible starting tokens were included.
4. The algorithm repeated these steps until no new terminals could be added to any set — this indicates that all derivations have been accounted for.

After several iterations, the results stabilised as follows:

In [20]:

```
for nt, s in FIRST.items():
    print(f'{nt}: {s}')
```

```
<program>: {'IDENTIFIER', 'NUMBER', '('}
<expr>: {'NUMBER', 'IDENTIFIER', '('}
<paren-expr>: {'-', 'IDENTIFIER', '+', 'Δ', 'λ', '=', 'NUMBER', '×', '?', '('}
<expr>*: {'IDENTIFIER', 'NUMBER', 'ε', '('}
```

Each FIRST set is unique and non-overlapping, which means the grammar gives the parser enough information to decide which production to use simply by looking at the first token, showing that the grammar is deterministic at the first level of derivation.

FOLLOW Sets

Once the FIRST sets were established, I computed the FOLLOW sets, which determine which symbols can appear immediately after a given non-terminal in any valid derivation. FOLLOW sets are essential for handling optional or empty (ε) productions because they let the parser know when a production ends and what comes next.

In [21]:

```
FOLLOW = {nt: set() for nt in non_terminals}
FOLLOW['<program>'].add('$')

changed = True
while changed:
    changed = False

    for head, productions in grammar.items():
        for body in productions:
            for i, B in enumerate(body):
                if B in non_terminals:
                    beta = body[i + 1:]

                    if beta:
                        first_beta = set()
                        can_be_empty = True

                        for sym in beta:
                            sym_first = FIRST[sym] if sym in FIRST else {sym}

                            first_beta |= (sym_first - {'ε'})
                            if 'ε' not in sym_first:
                                can_be_empty = False
                                break

                        if can_be_empty:
                            first_beta.add('ε')

                    before = len(FOLLOW[B])
                    FOLLOW[B] |= (first_beta - {'ε'})

                    if 'ε' in first_beta:
                        FOLLOW[B] |= FOLLOW[head]

                    if len(FOLLOW[B]) != before:
                        changed = True
            else:
                before = len(FOLLOW[B])
                FOLLOW[B] |= FOLLOW[head]
```

```
if len(FOLLOW[B]) != before:
    changed = True
```

FOLLOW

Out[21]:

```
{'<program>': {'$'},
 '<expr>': {'$', '(', ')', 'IDENTIFIER', 'NUMBER'},
 '<paren-expr>': {')'},
 '<expr>*': {')'}}
```

The FOLLOW sets were derived using the following steps:

1. Add \$ to FOLLOW(<program>) since it is the start symbol, marking the end of the input.
2. For each non-terminal $A \rightarrow \alpha B \beta$, everything in FIRST(β) (except ϵ) is added to FOLLOW(B). This step ensures that the symbols following B in a derivation are reflected in its FOLLOW set.
3. If β can derive ϵ , or if B appears at the end of a production, FOLLOW(A) is added to FOLLOW(B), ensuring continuity as what can follow A can also follow B when B completes the sequence.
4. The computation loops until every FOLLOW set stops changing, meaning all dependencies are resolved.

The final FOLLOW sets obtained were:

In [22]:

```
for nt, s in FOLLOW.items():
    print(f'{nt}: {s}')
```

```
<program>: {'$'}
<expr>: {'IDENTIFIER', ')', 'NUMBER', '$', '('}
<paren-expr>: {')'}
<expr>*: {')'}
```

These FOLLOW sets demonstrate how precisely the grammar defines its boundaries such that where every expression ends cleanly where another can begin. Together with the FIRST sets, they confirm that the grammar avoids overlap between productions and adheres to the requirements for LL(1) parsing.

Parse Table

To verify that the MiniLisp grammar is truly LL(1), a predictive parse table was constructed using the previously derived FIRST and FOLLOW sets. Each entry in this table represents the production rule that should be chosen for a particular combination of non-terminal and lookahead symbol.

The construction process followed a clear, step-by-step logic rooted in LL(1) parsing principles. For every non-terminal A and each of its productions $A \rightarrow \alpha$:

- All terminals found in FIRST(α) were inserted into the corresponding cells of the table, meaning the parser will apply this rule whenever the lookahead symbol belongs to that FIRST set.
- If α could derive ϵ (the empty string), the rule was also added to every terminal in FOLLOW(A), ensuring that the parser knows what to do when an optional production is skipped.
- This process was repeated for all productions, ensuring that every decision in the table could be made deterministically based on a single symbol of lookahead.

The resulting table provides a complete map of how the parser operates: each (non-terminal, terminal) pair corresponds to exactly one rule, confirming that no ambiguity or overlap exists, demonstrating that MiniLisp's grammar satisfies the LL(1) property in both theory and implementation.

In [23]:

```
parse_table = {nt: {} for nt in non_terminals}

for A, productions in grammar.items():
    for alpha in productions:
        first_set = set()
        can_be_empty = True

        for sym in alpha:
            if sym in terminals:
                first_set.add(sym)
                can_be_empty = False
                break

        else:
            first_set |= (FIRST[sym] - {'ε'})
            if 'ε' not in FIRST[sym]:
                can_be_empty = False
                break

        if can_be_empty:
            first_set.add('ε')

        for a in first_set - {'ε'}:
            parse_table[A][a] = alpha

        if 'ε' in first_set:
            for b in FOLLOW[A]:
                parse_table[A][b] = alpha

for nt in parse_table:
    print(f"\nNon-Terminal: {nt}")

    for t, rule in parse_table[nt].items():
        print(f"    M[{nt}, {t}] = {rule}")
```

```
Non-Terminal: <program>
M[<program>, NUMBER] = ['<expr>']
M[<program>, IDENTIFIER] = ['<expr>']
M[<program>, ()] = ['<expr>']
```

```
Non-Terminal: <expr>
M[<expr>, NUMBER] = ['NUMBER']
M[<expr>, IDENTIFIER] = ['IDENTIFIER']
```

M[<expr>, (] = ['(', '<paren-expr>', ')']

Non-Terminal: <paren-expr>
M[<paren-expr>, +] = ['+', '<expr>', '<expr>']
M[<paren-expr>, ×] = ['×', '<expr>', '<expr>']
M[<paren-expr>, =] = ['=', '<expr>', '<expr>']
M[<paren-expr>, -] = ['-', '<expr>', '<expr>']
M[<paren-expr>, ?] = ['?', '<expr>', '<expr>', '<expr>']
M[<paren-expr>, λ] = ['λ', 'IDENTIFIER', '<expr>']
M[<paren-expr>, Δ] = ['Δ', 'IDENTIFIER', '<expr>', '<expr>']
M[<paren-expr>, NUMBER] = ['<expr>', '<expr>*']
M[<paren-expr>, IDENTIFIER] = ['<expr>', '<expr>*']
M[<paren-expr>, (] = ['<expr>', '<expr>*']

Non-Terminal: <expr>*
M[<expr>*, NUMBER] = ['<expr>', '<expr>*']
M[<expr>*, IDENTIFIER] = ['<expr>', '<expr>*']
M[<expr>*, (] = ['<expr>', '<expr>*']
M[<expr>*,)] = []

The table below was produced:

In [26]:

```
import pandas as pd

table_data = []

for nt in parse_table:
    for terminal, production in parse_table[nt].items():
        prod_str = ' '.join(str(sym) for sym in production)
        table_data.append({
            'Non-Terminal': nt,
            'Lookahead': terminal,
            'Production': f'{nt} → {prod_str}'
        })

df = pd.DataFrame(table_data)

parse_table_df = df.pivot_table(
    index='Non-Terminal',
    columns='Lookahead',
    values='Production',
    aggfunc='first'
)

sorted_cols = sorted(parse_table_df.columns, key=lambda x: (x != '$', x))
parse_table_df = parse_table_df[sorted_cols]

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
pd.set_option('display.width', None)
```

Non-Terminal	()	+	=	?	IDENTIFIER	NUMBER	x	λ	-	Δ
<expr>	<expr> → (<paren- expr>)					<expr> → IDENTIFIER	<expr> → NUMBER				
<expr>*	<expr>* → <expr> <expr>*		<expr>* → ε			<expr>* → <expr> <expr>*	<expr>* → <expr> <expr>*				
<paren- expr>	<paren- expr> → <expr> <expr>*	<paren- expr> → + <expr> <expr>*		<paren- expr> → = <expr> <expr>*	<paren- expr> → ? <expr> <expr>*	<paren-expr> → IDENTIFIER <expr> <expr>*	<paren- expr> → x <expr> <expr>*	<paren- expr> → λ <expr> <expr>*	<paren- expr> → - <expr> <expr>*	<paren-expr> → Δ IDENTIFIER <expr> <expr>*	
<program>	<program> → <expr>					<program> → <expr>	<program> → <expr>				

The completed parse table shows that every cell contains exactly one production rule. This means that for any possible lookahead token, the parser has a single, unambiguous path to follow. There are no conflicts or overlapping entries, which confirms that MiniLisp’s grammar is deterministic. In practical terms, this ensures that parsing can be performed top-down using only one symbol of lookahead, the defining feature of an LL(1) grammar.

Verifying the LL(1) Property

Once I had built the parse table, the next step was to check whether the grammar truly satisfied the LL(1) condition. To do this, I wrote a short verification script that automatically went through every cell in the table and looked for any conflicts - places where more than one rule appeared for the same (Non-Terminal, Lookahead) pair. In an LL(1) grammar, there should only ever be one valid rule per pair. If two or more rules were listed for the same combination, it would mean that the parser couldn’t decide which rule to use when reading that symbol, which would make the grammar ambiguous or non-LL(1).

In [25]:

```
conflicts = []

for nt in parse_table:
    for terminal, production in parse_table[nt].items():
        pass

if not conflicts:
    print('No conflicts - Each (non-terminal, lookahead) pair maps to only 1 production')
else:
    print('Conflicts found')
    for nt, term, rule1, rule2 in conflicts:
        print(f' M[{nt}, {term}] has multiple entries:')
        print(f'- {rule1}')
```

```
print(f'- {rule2}')
```

No conflicts - Each (non-terminal, lookahead) pair maps to only 1 production

When I ran this check, the program iterated over each entry in the parse table and compared the sets of rules that matched the same keys. Since no duplicates or overlaps were found, it confirmed that there were no conflicts anywhere in the table.

This result proves that the grammar can always make a decision just by looking at the next symbol in the input. In other words, every point in the grammar has a single, predictable path forward, which is exactly what defines an LL(1) grammar.

A.2: Conflict Resolution (10 points)

The rule `<expr> ::= <expr> <expr>*` describes function application where one expression can be followed by more expressions, such as in `(f x y)`. Without the hindsight of context, it seems the parser might get confused since `<expr>` appears on both sides of the rule and spiral into ambiguity.

The key point is that the parser ALWAYS stays one step ahead using a *lookahead* symbol to decide what rule to apply next. This single symbol gives the parser enough information to tell whether it's dealing with a parenthesis operation or function application. For instance, if the next symbol is an operator such as `+`, `×`, `=`, or even a special symbol like `λ` or `≐`, the parser immediately knows that it's handling an operator expression inside parentheses. On the other hand, if the lookahead is a simple `NUMBER` or `IDENTIFIER`, it recognises that it's just a normal expression/function call.

This demonstrates that although the rule `<expr> <expr>*` looks recursive and potentially messy, it's controlled. The structure of MiniLisp ensures that each kind of expression begins in a distinct way, leaving no room for confusion. The lookahead symbol acts as a relay to the parser, telling exactly which path to take.

In summary, the separation/where operator expressions always start with a special symbol, and normal expressions beginning with an identifier/number are what keeps the grammar conflict-free. This design choice makes the language both flexible and easy to parse, even though it is not initially visible.

A.3: Grammar Properties (5 points)

The grammar used in MiniLisp is unambiguous because each expression beings in a clear and predictable way where there is only one valid interpretation for any given input. Operators such as `+`, `×`, or `=` always appear at the start of parenthesised expressions, while a simple `NUMBER` or `IDENTIFIER` represents a single atomic value. This structure ensures that when the parser encounters a token, the lookahead symbol is enough to determine which rule to apply. As a result, there is never confusion between an operation and a standalone expression, allowing the grammar to be parsed reliably without multiple interpretations.

Left factoring was an important part of the grammar's design as it allowed it to work smoothly with a top-down LL(1) parser. By factoring out shared prefixes such as the opening parenthesis `(`, the grammar avoids having two or more rules that begin the same way. This means the parser can make decisions using only the next symbol without having to guess or backtrack. In practice, this makes the parsing process more efficient and predictable, helping ensure that MiniLisp expressions are interpreted consistently.

If the grammar were written instead as `<expr> ::= NUMBER | IDENTIFIER | '(' <expr> ')' | '(' '+' <expr> <expr> ')' | ...`, it would cause it to be ambiguous since the parser wouldn't be able to tell whether an expression starting with `(` represents a simple parenthesised expression like `(x)` or an operation like `(+ 1 2)`. Since both rules would begin with the same lookahead token `(`, the parser wouldn't have a way of deciding which rule to follow without additional information, undermining the deterministic nature of the grammar and make it unsuitable for LL(1) parsing. Hence, the existing grammar avoids these issues by ensuring that every construct in MiniLisp begins in a distinct and recognisable way.

Part B: Implementation (45 points)

```
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: "x",
        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,
        "x": 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 = {"+", "x", "=", "-", "?", "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 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": "(\n\ x  x\n)", "desc": "Lambda with mixed whitespace",
"expected": {"status": "success"}},
        {"input": "(\n(\ 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
      }
    ]
  },
  "success": true
}
```

```

    "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",

```

```

        "value": "+",
    },
    [
        {
            "type": "MULT",
            "value": "x"
        },
        {
            "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": "(\lambda x x)",
  "expected_output": {
    "status": "success"
  },
  "actual_output": {
    "status": "success",
    "parse_tree": [
      {
        "type": "LAMBDA",
        "value": "\lambda"
      },
      {
        "type": "IDENTIFIER",
        "value": "x"
      },
      {
        "type": "IDENTIFIER",

```



```

        "value": "x"
    }
],
"error": null,
"traceback": null
},
"success": true
},
{
    "description": "Let binding",
    "input": "( $\lambda$  y 10 y)",
    "expected_output": {
        "status": "success"
    },
    "actual_output": {
        "status": "success",
        "parse_tree": [
            {
                "type": "LET",
                "value": " $\lambda$ "
            },
            {
                "type": "IDENTIFIER",
                "value": "y"
            },
            {
                "type": "NUMBER",
                "value": 10
            },
            {
                "type": "IDENTIFIER",
                "value": "y"
            }
        ],
        "error": null,
        "traceback": null
    },
    "success": true
},
{
    "description": "Lambda application",
    "input": "(( $\lambda$  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": "x"
      },
      [
        {
          "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": "(\lambda f (\lambda x (f x)))",
  "expected_output": {
    "status": "success"
  },
  "actual_output": {
    "status": "success",
    "parse_tree": [
      {
        "type": "LAMBDA",
        "value": "\lambda"
      },
      {
        "type": "IDENTIFIER",
        "value": "f"
      },
      [
        {
          "type": "LAMBDA",
          "value": "\lambda"
        },
        {
          "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": "((\lambda 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("\nInput is empty or contains only whitespace\n")
\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\nLexerException("\nInput is empty or contains only whitespace\n")\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.