SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

Mathematical Language Compiler

Guided by:

Mr. H. Karthikeyan

Assistant Professor, Department of Networking and Communications

Submitted By:

Prakhranshu Singh (RA2011030010217)

Sooraj Tomar (RA2011030010224)

Pankaj Bhattarai (RA2011030010230)

Table of Contents

Aim: - ***To develop a mathematical language compiler to parse and give the result of mathematical queries given in plain English as well as mathematical format.***

ABSTRACT: -

We live in the age of personal virtual assistants where they aid us in our day-to-day tasks. Such incredibly smart Artificial Intelligence is based on robust and efficient coding which enables these assistants to do what we ask them to do. There are several modules which these assistants use, one of which can be a Mathematical Language Compiler based on user's input.

The module as a whole shall provide output/result of an input basic mathematical problem that a user may face in their day to day lives.

Requirements to run the script:

- Laptop/Computer with minimum specifications as:1.6 GHz or faster processor and 1 GB RAM.
- Python 3.1 or higher
- Python IDE such as Jupyter Notebook, VS Code, PyCharm etc.
- Ply 3.11 or higher
- word2number module

Modules:

The following modules shall be implemented in the code:

- **mylexer.py** – Performs lexical analysis.

  Identifies tokens - such as keywords, operators, operands - that is, values, their types such as integer or floating point etc. and the mathematical operations to be done on those values. Also, throws an error message in case of an illegal or unidentified token.

- **myparser.py** – Performs parsing of the tokenized input.

  Identifies the operations to be done on the tokenized input and parses the given mathematical query to check if the operation is legal or not, throws an error in case of illegal query (like division by zero). Parsing done in LALR (1).

- **project.py** – Controls the entire processing as a main program.

  The user interacts with this program. The program taken in the user input, sends it to the lexical analyser whose tokenized output is sent to the parser after which the parsed query is evaluated and the appropriate output is given to the user.

Type of Parsing Method used: LALR (1)

LALR Parser is lookahead LR parser. The “LR” stands for **L**eft to Right parsing **R**ight Most Derivation Method. The “(1)” means that the parser uses a lookahead value of 1. It is the most powerful parser which can handle large classes of grammar. The size of CLR parsing table is quite large as compared to another parsing table. LALR reduces the size of this table. LALR works similar to CLR. The only difference is, it combines the similar states of CLR parsing table into one single state.

Algorithm:

Step 1: Start

Step 2: Get user input

Step 3: Pass input to lexical analyser

Step 4: If all valid tokens, send tokenized input to parser

Step 5: Else output appropriate erroneous input message

Step 6: Parse the tokens

Step 6: If valid query, pass it to processor in main module

Step 7: Else output appropriate erroneous input message

Step 8: Give output of valid processed query

Step 9: Stop

Flowchart:

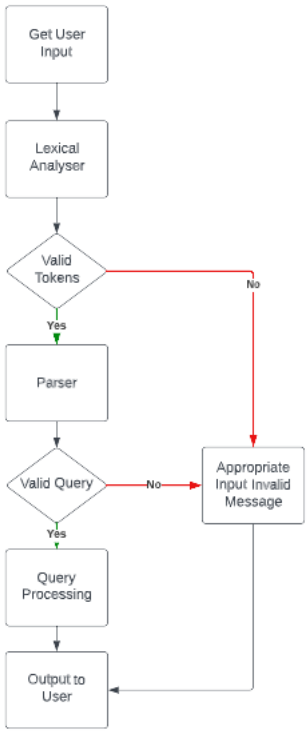

Figure 1: Program Processing Flowchart

Code:

mylexer.py

```
 import ply.lex as lex
tokens = (
    'INT',
    'FLOAT',
    'PLUS',
    'MINUS',
    'DIV',
    'MUL',
    'POWER',
    'LEFTPAR',
    'RIGHTPAR'
)


def MyLexer():
    t_ignore = ' '

    t_PLUS = r'\+'
    t_MINUS = r'\-'
    t_MUL = r'\*'
    t_DIV = r'\/'
    t_POWER = r'p'
    t_LEFTPAR = r'\('
    t_RIGHTPAR = r'\)'

    def t_FLOAT(t):
        r'\d+\.\d+'
```

```
    t.value = float(t.value)
    return t

def t_INT(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_error(t):
    print("illegal char:", t.value[0])
    t.lexer.skip(1)
return lex.lex()
```

myparser.py

```
from mylexer import MyLexer, tokens
import ply.yacc as yacc


lexer = MyLexer()


def MyParser():
    def p_E(p):
        '''E : E PLUS T
             | E MINUS T
        '''
        if p[2] == '+':
            p[0] = p[1] + p[3]
        elif p[2] == '-':
            p[0] = p[1] - p[3]

    def p_E_T(p):
        'E : T'
        p[0] = p[1]

    def p_T(p):
        '''T : T MUL F
             | T DIV F
        '''
        if p[2] == '*':
            p[0] = p[1] * p[3]
        elif p[2] == '/':
            try:
```

```
            p[0] = p[1] / p[3]
        except ZeroDivisionError:
            print("NO ZEROOOOO!!!!")

def p_T_F(p):
    'T : F'
    p[0] = p[1]

def p_F(p):
    '''F : G POWER F
         | MINUS G
    '''
    if p[1] == '-':
        p[0] = -p[2]
    elif p[2] == 'p':
        p[0] = p[1] ** p[3]

def p_F_G(p):
    'F : G'
    p[0] = p[1]

def p_G(p):
    '''G : NUM
         | LEFTPAR E RIGHTPAR
    '''
    if p[1] == '(':
        p[0] = p[2]
    else:
        p[0] = p[1]
```

```
def p_NUM(p):
    '''NUM : INT
           | FLOAT
    '''
    p[0] = p[1]

def p_error(p):
    print("syntax error!")

return yacc.yacc()
```

project.py

```
from myparser import MyParser
from word2number import w2n

if __name__ == "__main__":
    parser = MyParser()
    while True:
        # getting input and parse while 'ctrl+d' pressed
        try:
            s = input('Input Exp >>>> ')
            op=['+','-','/','*']
            operator=''
            for o in op:
                if o in s:
                    operator=o
                    break
            splitted = s.split(operator)
            # print(splitted[0],splitted[1])
            exp1=splitted[0].strip()
            exp2=splitted[1].strip()
            try:
                if not exp1.isnumeric():
                    exp1=w2n.word_to_num(exp1)
                if not exp2.isnumeric():
                    exp2=w2n.word_to_num(exp2)
                # print(exp1,exp2)
                s=str(exp1)+operator+str(exp2)
                # print(s)
            except Exception as e:
                pass
```

```
except EOFError:
    break
if not s:
    continue
result = parser.parse(s)
print(f"Result Is -> { {result} }\n")
```

Output:

```
PS E:\SRM\Semester VI\Compiler Design\project\Compiler-project\code> python .\Project.py
Input Exp >>>> two million + three
2000000+3
Result Is -> {2000003}
```

Figure 2: Input given in plain English

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  SQL CONSOLE  GITLENS  COMMENTS
PS E:\SRM\Semester VI\Compiler Design\project\Compiler-project\code> python .\Project.py
Input Exp >>>> two+2
Result Is -> {4}

Input Exp >>>> 2+two
Result Is -> {4}

Input Exp >>>> 2+2
Result Is -> {4}

Input Exp >>>> four million+5
Result Is -> {4000005}
```

Figure 3: Inputs in mixed and pure mathematical format

```
Input Exp >>>> two - two
Result Is -> {0}

Input Exp >>>> five-two
Result Is -> {3}

Input Exp >>>> five*seven
Result Is -> {35}

Input Exp >>>> five/two
Result Is -> {2.5}
```

Figure 4: Another set of inputs

Future Updates:

The way in which the program is developed has made it possible to incorporate further updates in the future. The user shall be able to given speech inputs as well as receive speech outputs. The complexity of the mathematical inputs can be increased by incorporating into the code the ability to solve those complex mathematical equations. The speech component can further be diversified to accept different languages.

Applications:

This program can be incorporated with several services. It can be used in a Web App where users can submit mathematical problems to get solutions to them (when the complex mathematical solutions update is installed). It can also be used from simple calculators to smart virtual assistants which can use this functionality to answer this specific subset (mathematical queries) of the user's queries.

Result: ***Thus, the mathematical language compiler was successfully developed and its results as well as potential applications were also discussed.***