CS2349 – Spring 2021 — Project

Kanishk Singh

Introduction:

Through this project I would try get a better understanding of Lexical analysers and Parsers through an implementation to make a functional calculator which takes in algebraic expressions as input and outputs the solution value of the expression.

The essential usage of a lexer is to take an input stream of characters and try to break the stream into smallest possible functional chunks according to predefined rules. These chunks, called `tokens`, are then taken in by the parser as its input. Lexer acts as a pre-processor for the parser and just recognises `regular language`.

The parser takes the input from the lexer in form of tokens, and recognises `context-free language` give the grammar rules. It gives out a `parse tree` which signifies the relation between the tokens ingested and gives a hierarchical structure.

The main usage of these two are in Programming Language Compilers to take the code, put it through checks depending on the language definition, and then make sense it through defined logic.

Lexer:

The lexer implemented in the project takes help of a lexical analyzer generator called `LEX`. With LEX, we define regular expression rules in form of variables (for particular characters in the input) and functions (for operable data types such as numbers and/or for when specific code needs to be executed at the encounter of a regular expression) for a list of acceptable tokens that we want for our lexer to accept. LEX reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by LEX. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream. Here, the Python port of the original LEX (written for C) is used from the PLY library.

```
# list of acceptable tokens
tokens = (
'INT',          # integers
'FLOAT',        # decimal numbers
'POWER',        # exponemt
'PLUS',         # addition
'MINUS',        # subtraction
'MULTIPLY',     # multiplication
'DIVIDE',       # division
'BRACKETOP',    # bracket open
'BRACKETCL',    # bracket close
)
```

Defining the list of acceptable tokens.

```
# defining regex for the various tokens
t_POWER     = r'\^'
t_PLUS      = r'\+'
t_MINUS     = r'\-'
t_MULTIPLY  = r'\*'
t_DIVIDE    = r'\/'
t_BRACKETOP = r'\('
t_BRACKETCL = r'\)'

def t_FLOAT(t):
    r'\d+\.\d+'
    t.value = float(t.value)    # changing token value data type to float from string
    return t

def t_INT(t):
    r'\d+'
    t.value = int(t.value)    # changing token value data type to int from string
    return t

t_ignore = " \t"    # regex token for whitespace which is ignored
```

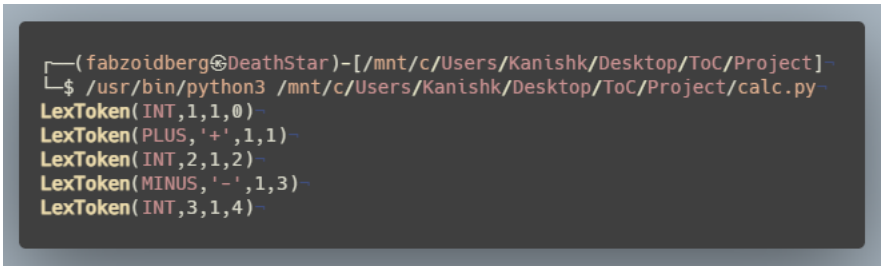
Defining regex rules for the various token types.

The `t_` prefix is used to define token type with a regular expression rule. Each matching regular expression is given this defined token type. The characters not belonging to the defined regular language are skipped and printed as illegal characters.

```
# handling unexpected characters
def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)
```

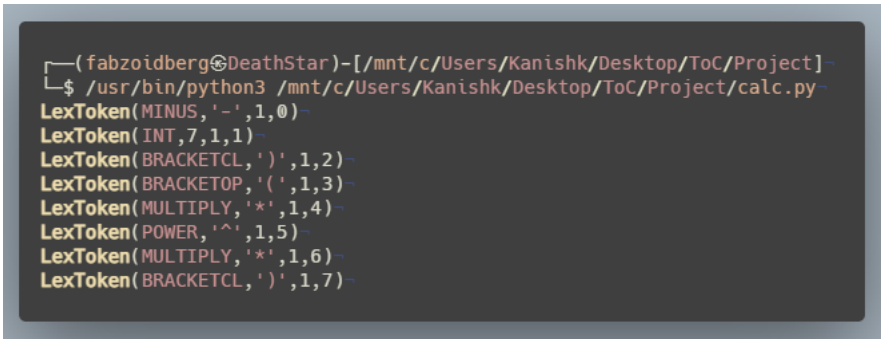
Error handling.

This lexer returns `LexToken` objects with `token.type`, `token.value`, `token.lineno`, and `token.lexpos` as attributes for the type of token, value of the token, line number the token was on, and position of the token relative to the beginning of the input text respectively.



Lexer output for `1+2-3`.

Lexer output for `1_1/0=3`.



Lexer output for `-7)(*^*)`.

As seen above in the last output, the lexer does not reject anything as the input string only had valid tokens but arithmetically makes no sense. For that we use the parser.

Parser:

The parser implemented in this project takes help of `YACC`: Yet Another Compiler-Compiler. YACC is a Look-ahead, Left-to-right, Rightmost derivation producer with 1 look-ahead token. YACC takes the stream of tokens from LEX and uses pre-defined Backus normal form grammar that describe how those tokens are assembled. For ambiguous grammar the parser already looks ahead one token and we also define precedence rules for various tokens. Whenever there is a match found for a grammar rule, the corresponding function is executed. It creates an object with the various terms

at separate indices with the 0th index storing the result of the function operations. Here, the Python port of the original YACC (written for C) is used from the PLY library.

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULTIPLY', 'DIVIDE'),
    ('left', 'POWER'),
    ('right', 'NEGATIVE')
)
```

Defining the precedence rules as operator associativity followed by operators; listed in lowest to highest precedence from top to bottom.

```
Grammar

Rule 0     S' -> calc
Rule 1     calc -> expression
Rule 2     calc -> empty
Rule 3     expression -> expression POWER expression
Rule 4     expression -> expression PLUS expression
Rule 5     expression -> expression MINUS expression
Rule 6     expression -> expression MULTIPLY expression
Rule 7     expression -> expression DIVIDE expression
Rule 8     expression -> BRACKETOP expression BRACKETCL
Rule 9     expression -> MINUS expression
Rule 10    expression -> INT
Rule 11    expression -> FLOAT
Rule 12    empty -> <empty>
```

Grammar used.

```
def p_exp(p):
    '''
    expression : expression POWER expression
               | expression PLUS expression
               | expression MINUS expression
               | expression MULTIPLY expression
               | expression DIVIDE expression
    '''
    p[0] = (p[2], p[1], p[3])
```

Defining context-free grammar.

The `p_` prefix is used to define functions with grammar rules that trigger it. The above code snippet is responsible for construction of the (flattened) `Parse Tree` which is an ordered, rooted tree that

represents the syntactic structure of a string according to the context-free grammar. We merely take the operator and place it as the root and the operands as the nodes and this is repeated for all the sub-trees. The tokens not belonging to the defined context-free grammar are printed as error and if the entered expression has valid tokens but do not follow grammar rules, appropriate error message is printed.

```
# handling unexpected expression
def p_error(p):
    if p:
        print(f"Error before {p.value}")
    else:
        print("Illegal or incomplete expression")
```

Error handling.

The flattened parse tree is what we will use to calculate the final value of the input expression but for now we get the `calculator(p)` function to print the flattened parse tree.

```
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: 1+2-3
('-', ('+', 1, 2), 3)
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: 1+2--3
('-', ('+', 1, 2), -3)
```

Handling negative numbers.

```
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: 1_1/0=3
Illegal character '_'
Error before 1
Illegal character '='
None
```

Here `None` states no parse tree was built.

```
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: -7)(*^*)
Error before )
None
```

The lexer did not throw any error for this input but parser catches the invalid expression.

```
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: 3^(5*6-9)/3
('/', ('^', 3, ('-', ('*', 5, 6), 9)), 3)
```

Parse tree for `3^(5*6-9)/3`.

```
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: 9^3*6-5
('-', ('*', ('^', 9, 3), 6), 5)
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: 9^3*(6-5)
('*', ('^', 9, 3), ('-', 6, 5))
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/Project/calc.py
enter expression: 9^(3*6-5)
('^', 9, ('-', ('*', 3, 6), 5))
```

Different trees built with differing bracket placements.

We will further use these flat parse trees to evaluate the expressions.

Computing result:

With the flattened parse tree, we make a recursive function and look which operator is at the $0th$ index of the tuple and accordingly operate on the other members of the tuple. This recursively goes on to evaluate all the subtrees.

```
def calculator(p):
    if type(p) is tuple:  # check if parse tree created
        if p[0] == '^':
            return calculator(p[1]) ** calculator(p[2])
        if p[0] == '+':
            return calculator(p[1]) + calculator(p[2])
        if p[0] == '-':
            return calculator(p[1]) - calculator(p[2])
        if p[0] == '*':
            return calculator(p[1]) * calculator(p[2])
        if p[0] == '/':
            return calculator(p[1]) / calculator(p[2])
    else:
        return p
```

Evaluating function.

```
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/ToC_Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/ToC_Project/CalcCLI.py
enter expression: 9^3*6-5
Flat parse tree: ('-', ('*', ('^', 9, 3), 6), 5)
Result: 4369
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/ToC_Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/ToC_Project/CalcCLI.py
enter expression: 9^3*(6-5)
Flat parse tree: ('*', ('^', 9, 3), ('-', 6, 5))
Result: 729
┌──(fabzoidberg㉿DeathStar)-[/mnt/c/Users/Kanishk/Desktop/ToC/ToC_Project]
└─$ /usr/bin/python3 /mnt/c/Users/Kanishk/Desktop/ToC/ToC_Project/CalcCLI.py
enter expression: 9^(3*6-5)
Flat parse tree: ('^', 9, ('-', ('*', 3, 6), 5))
Result: 2541865828329
```

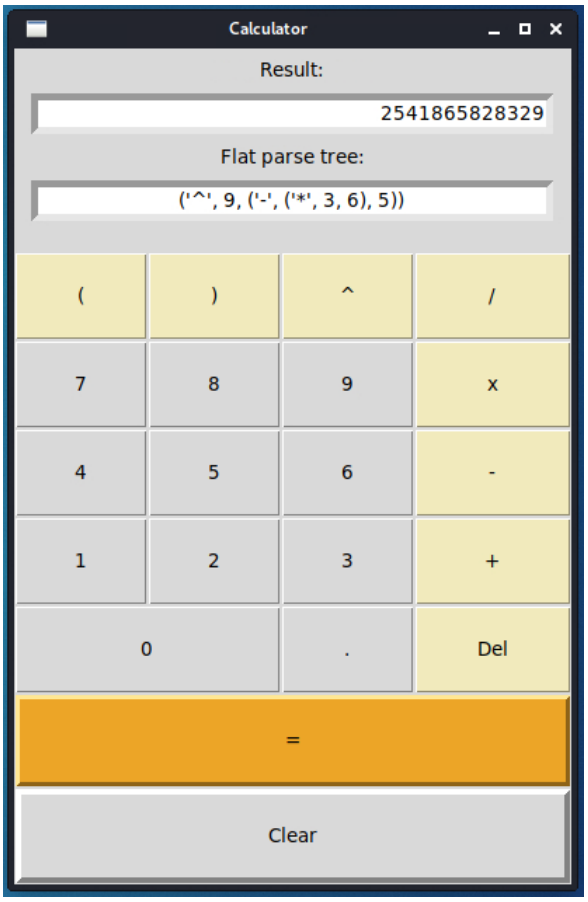
The program returns the parse tree and the result of the expression entered.

The CLI only version is available in the `CalcCLI.py` file.

Graphical User Interface:

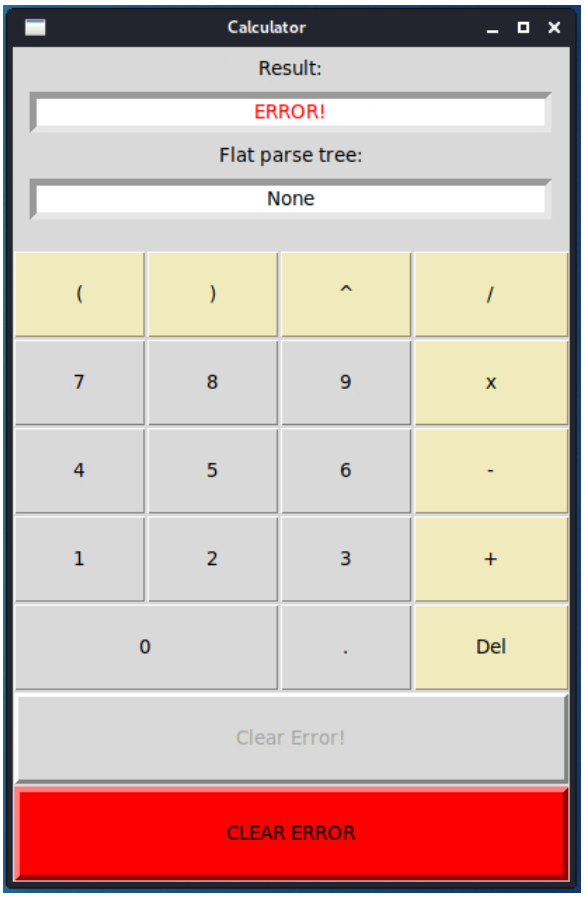
With the same base as the CLI version, the GUI is made using `TkInter` python library.

The calculator GUI.



Sample output.

The GUI takes input from keyboard as well as clickable buttons and returns the equivalent result and the parse tree for that expression. The error reporting is not as verbose as in the CLI version. The text from the input field is directly fed to the parser object on the click of '=' which then calls the lexer and the generated tokens are then again used by the parser to determine the language membership and build the parse tree.



Error reporting.

The same error is thrown for both illegal characters and illegal expressions. The program still puts out additional error information in the terminal while the GUI is running. This GUI version is available in the `CalcGUI.py` file.

Conclusion:

In the making of this project, the majority of the effort went towards understanding the functioning of a Lexical Analyzer and a Parser. LEX and YACC helped a lot in simplifying the process of constructing the lexer and parser but the understanding of the functioning helped a lot in understanding how PLY works and how to use it. This is a very simple implementation of the lexer parser duo which are immensely important in computing but gives a good idea of its basics in this implementation.

The Project is available on GitHub Repository and references for PLY Documentation and TkInter Documentation.