MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Laboratory Work 3:

Lexer & Scanner.

Course: Formal Languages & Finite Automata

Student: Gusev Roman

Group: FAF-222

Professor: Cojuhari Irina

University Assistant: Crețu Dumitru

Chișinău, 2024

Content

Theory

- **Definitions:**
  - **Lexer** - The lexical analyzer defines how the contents of a file are broken into tokens, which is the basis for supporting custom language features [1];
  - **Token** - A lexical token is a string with an assigned and thus identified meaning.
  - **Lexeme** - A lexeme is only a string of characters known to be of a certain kind.

Objectives

- Understand what lexical analysis [2] is;
- Get familiar with the inner workings of a lexer/scanner/tokenizer;
- Implement a sample lexer and show how it works.

Implementation Description

- For the start, I decided to implement a Lexer for the PBL DSL that me and the team I work with decided to develop. During the work on this Laboratory Work, I decided to implement an example Lexer that I found in some guides on YouTube [3]. Specifically, I followed the structure of the presented in the video Lexer, but for the extension I used only my knowledge.
- The Lexer I developed pass through the entire Input symbol by symbol until the End of File, and transform each combination of the Input into Lexemes. The mathematical expressions are not required to be separated by Whitespace and may be written in both, for example "2+2" and "2 + 2", forms, and will be tokenized correctly. For the "words" type Lexemes, they are required to be separated by whitespace. This is done in order to greedily tokenize them and eliminate cases in inputs such as "Formula1", that may be tokenized like:

```
FORMULA_TOKEN : Formula
NUMBER: 1
```

and tokenize that as a single word, that might give theoretically output as:

```
IDENTIFIER: Formula1
```

This was done in order to have a defined structure for the Tokens, that will lead to a whitespace-sensitive Language.

- First thing developed was the Tokens themselves. I have several Tokens that I decided to hold in Enum classes, and that have been used in the Lexer to classify them.

```python
from enum import Enum

class OperatorType(Enum):
    MULTIPLY = "*"
    DIVISION = "/"
    ADDITION = "+"
    SUBTRACTION = "-"
    POWER = "**"

class ComparatorType(Enum):
    EQUALS = "=="
    NOT_EQUALS = "!="
    LESS_THAN = "<"
    GREATER_THAN = ">"
    LESS_THAN_OR_EQUALS = "<="
    GREATER_THAN_OR_EQUALS = ">="

class FileType(Enum):
    CSV = "csv"
    TEXT = "txt"
    JSON = "json"
    EXCEL = "excel"
    CONSOLE = "console"

class ImageType(Enum):
    JPG = "jpg"
    PNG = "png"

class PlotType(Enum):
    GRAPH = "graph"
    BAR = "bar"
    PIE = "pie"
    PLOT = "plot"
```

```
    HIST = "hist"

class VariableType(Enum):
    FORMULA = "Formula"
    DATA = "Data"
    DATASET = "dataset"
    NAME = "name"

class ExportToType(Enum):
    EXPORT_TO_IMAGE = "ExportToImage"
    EXPORT_TO_FILE = "ExportToFile"

class VisualizationType(Enum):
    VISUALIZE_DATA = "VisualData"
    VISUALIZE_FORMULA = "VisualFormula"
```

- At the same time, I started with the Token class, that will describe what type of Lexeme I have in input. Here I defined the Constructor for the Token, where it gets the type/kind of the Token (type is a reserved keyword in python therefore I used kind), and the line of the input - whole input of the User. Also, it gets the string, that is the Token itself. But there may be a case when the String returned is empty, so it is replaced with "EOF" - End of File Token:

```
class Token:
    def __init__(self, kind, line):
        self.kind = kind
        self.line = line
        self.locale, self.string = line.new_locale()

        # If retrieved string is empty, then set it as "EOF",
           otherwise - stays the same retrieved string
        self.string = self.string or "EOF"
    ...
```

- After that, Token class has 2 other methods in it - "__repr__" that will be the representation of the Token in a visual human-readable form for data structures such as Lists, because they call specifically

this method. Tokens will be present in a list, therefore this method is required.

- At the same time, 2-nd method - "mark", will be used to make the visual representation of the Incorrect Tokens during the evaluation. This method will be explained in depth further.

```
class Token:
    ...
    def __repr__(self):
        return f"{self.kind}: '{self.string}'"

    def mark(self):
        self.line.mark(self)
```

- Next Class developed for the Lexer implementation was "SourceLine", that is used to hold the input from the user and will hold also a lot of useful functionalities that will be called by the Lexer class and will track the Tokens based on their index in the input. First of all, I developed the Constructor, that will hold the Input from the user, will hold some locale variables, a list of 2 integers, that will be the position of the START of the Token and END of the Token, that was done because some Tokens are multi-symbol Tokens, and Marked, that will hold the wrong portion of the Token, in case that this Token is invalid:

```
class SourceLine:
    def __init__(self, line):
        self.line = line  # Actual line from the User Input
        self.locale = [0, 0]  # Track START and END of a Token
        self.marked = [len(line), -1]
    ...
```

- After that, I have a method to check if the Input from the User was processed entirely. This is done by the comparison check between the locale variable and the length of the input - if the SourceLine locale passed the length of the string input or is equal to this length, then the input was tokenized entirely:

```
class SourceLine:
    ...
    # Checks if line was processed entirely
    def finished(self):
        return self.locale[1] >= len(self.line)
```

```
    ...
```

- After that, I developed the method to actually take and adjust the state of the variables in order to progress in the Token components further. This method will peek the next symbol and will increment the locale variables - specifically the END position if the Input is not processed fully, otherwise will stay the same. Also, this method returns the symbol that was peeked:

```
class SourceLine:
    ...
    # Adjusts the states and takes one symbol at a time
    def take(self):
        symbol = self.next()
        self.locale[1] += 0 if self.finished() else 1

        return symbol
    ...
```

- Here is presented a method to Ignore some Symbols from the Input, such as Whitespaces or others. This is done by setting the START Position/Index of the Token to END Position/Index - they will become the same and will be after the Ignored Symbols:

```
class SourceLine:
    ...
    # Ignores current Locale for example spaces
    def ignore(self):
        self.locale[0] = self.locale[1]
    ...
```

- After that, I have a method to retrieve the entire Multi-Symbol Token. This is done by String Slicing. This method will Slice the Input String and will retrieve the Token between START and END Indices:

```
class SourceLine:
    ...
    # Takes MultiCharacter Symbol
    def taken(self):
        return self.line[self.locale[0]:self.locale[1]]
    ...
```

- Here I present the method that will update the Locale variables in order to assign it correctly those START and END Positions for the new Token that will be created. This method was featured in the Constructor for the Token class. This is done by the assignment of the Locale variables and the actual Token string when the Token is constructed itself. After the that the locales that are in the class SourceLine are reset to the END of the current Token:

```
class SourceLine:
    ...
    def new_locale(self):
        locale, taken = self.locale.copy(), self.taken()
        self.locale[0] = self.locale[1]
        return locale, taken
    ...
```

- Here I have 2 methods that are related to Error handling, i.e., will show where the error in the String is. Given the Token, that has its own Locale variables, it will compare them with the marked variables. Marked are initially set outside the Token range, and for the START Position of the mistake Symbol, it will take the minimum value between START of the Token and the outside bound of the marked variable, and for the END - it is the maximum value. After that, it will append to a string called "marks", that will hold the marks, and in a for loop it will add the "^̂" marks that will indicate where the mistake is between the boundaries of the marked values:

```
class SourceLine:
    ...
    def mark(self, token):
        self.marked[0] = min(token.locale[0], self.marked[0])
        self.marked[1] = max(token.locale[1], self.marked[1])

    # Creates Error Marks
    def get_marks(self):
        marks = "  "

        for i in range(len(self.line) + 1):
            between = self.marked[0] <= i < self.marked[1]
```

```
            marks += "^" if between or self.marked[0] == i else " "

        return marks
```

- After that, I have developed a new class for Error handling, i.e., will be the class for the actual LanguageError, of type RuntimeError, that will get the marks of the Token where error was obtained and will output them, with the actual name of the Error that will be assigned when subclasses of that LanguageError class will be created:

```
class LanguageError(RuntimeError):
    def __init__(self, component, message):
        component.mark()
        self.line = component.line
        self.message = message

    def __str__(self):
        return f"{self.line.get_marks()}\n{type(self).__name__} : {
            self.message}"
```

- For the actual Lexer, I created a new python file with the same name - Lexer. First of all, I imported all the previously created files and also created a class for the LexerError, that is a subclass of the LanguageError, that will be used to create more specific Errors further:

```
from Error import LanguageError
from Token import Token
from Tokens import FileType, VariableType, ExportToType, ImageType,
    PlotType, VisualizationType

class LexerError(LanguageError):
    pass
...
```

- After that, I created the class for the Lexer, that has a Constructor in the beginning, that will hold the actual Input of the User, that was converted to the SourceLine class - this will provide the extensive use of the methods that were developed in that class:

```
...
```

```
class Lexer:
    def __init__(self):
        self.line = None
    ...
```

- Here is another method from Lexer class, that returns a new Token with specified Kind/Type and with the input String from the User. This method will be used the most amongst other methods, because by this method, new Tokens will be created from Lexemes, based on some criteria of matching patterns further when analyzing the Input String:

```
...
class Lexer:
    ...
    def new_token(self, kind):
        return Token(kind, self.line)
    ...
```

- Here is another method from Lexer class, that is called once to start the process of Tokenization. First, I take the Input String and assign it to the stored Input String variable that is in the scope of a Lexer object. Afterward, I create a list of Tokens, that will store all the Tokens. In a while loop, I iterate over the Input String until it is finished, i.e., until the End of File. Also, I call the method to ignore the Whitespaces. At the end of each iteration, the list of tokens is appended with new Token that. And at the end, I add the last Token - "EOF":

```
...
class Lexer:
    ...
    def make_tokens(self, line):
        self.line = line
        tokens = []

        while not self.line.finished():
            self.ignore_spaces()
            if self.line.finished():
                break
```

```
            tokens += [self.make_token()]

        tokens += [self.new_token("EOF")]
        return tokens
    ...
```

- Here is the method to ignore Whitespaces. It is an iteration based method, that will go over the Input String until it encounters a non-whitespace character. In the loop, will be called 2 methods - to take new element (will update the variables in the scope of the SourceLine object) and will ignore each whitespace:

```
...
class Lexer:
    ...
    def ignore_spaces(self):
        while self.line.next().isspace():
            self.line.take()
            self.line.ignore()
    ...
```

- Here is the method to make Tokens. Each next Symbol from the Input will be checked to match any of the Token Type. Specifically, in this code snippet is shown how Non-Word Lexemes are matched to a Token Type, such as: LPAREN or RPAREN Tokens (parenthesis Tokens) or NUMBER Tokens and so on. They all call their own specific method of Tokenization:

```
...
class Lexer:
    ...
    def make_token(self):
        if self.line.next() in "()":
            return self.make_parenthesis()

        if self.line.next() in "[]":
            return self.make_brackets()
```

```
        if self.line.next() == ";":
            return self.make_semicolon()

        if self.line.next() == ",":
            return self.make_colon()

        if self.line.next() in "<>=":
            return self.make_comparison()

        if self.line.next() in "|_^":
            return self.make_punctuator()

        if self.line.next() in "~+-=\%":
            return self.make_operator()

        if self.line.next() in "0123456789.":
            return self.make_number()

        if self.line.next() in "\#":
            return self.make_comment_line()

        if self.line.next() in "/":
            return self.make_comment_block()

        if self.line.next() == '"':
            return self.make_path()
        ...
```

- Here is the method to make Tokens. Each next Symbol from the Input will be checked to match any of the Token Type. Specifically, in this code snippet is shown how Word Lexemes are matched to a Token Type, such as: FORMULA Token (keyword - Formula) or VISULIZATION_TYPE Token (VisualData or VisualFormula). They all call their own specific method of Tokenization. In the end, if a Lexeme is not a Token, then raise a LexerError, with "Unrecognized Symbol!" message:

```
...
class Lexer:
    ...
    def make_token(self):
        if self.line.next().isalpha():
            if self.line.next() == "F":
                return self.make_keyword_token("Formula")
            elif self.line.next() == "D":
                return self.make_keyword_token("Data")
            elif self.line.next() == "R":
                return self.make_read_from()
            elif self.line.next() == "E":
                return self.make_export_to()
            elif self.line.next() == "V":
                return self.make_visualization()
            else:
                return self.make_ID()

        self.line.take()
        raise LexerError(self.new_token("?"), "Unrecognized Symbol!
           ")
        ...
```

- Next, I will list some of the methods of Tokenization, not all, because essentially they are pretty similar between each other, and the main difference is that some of them perform more checks and takes more symbols.
- Here I provided the Tokenization of VISUALIZATION_TYPE Tokens. First of all, I iterate over the Lexeme until it is finished, and check if it is in the VisualizationType Enum class. If it is, then create a new Token of this Type. Otherwise - raise a LexerError with the message that maybe User meant one of 2 available VisualizatioType Tokens:

```
...
class Lexer:
    ...
```

```
    def make_visualization(self):
        while self.line.next().isalnum() and not self.line.finished
           ():
            self.line.take()

        if self.line.taken() in VisualizationType:
            return self.new_token("VISUALIZATION_TYPE")

        raise LexerError(self.new_token("VISUALIZATION_TYPE"), "
           Maybe you meant 'VisualData' or 'VisualFormula' instead?
           ")
```

- Here I provided the Tokenization of COMPARISON_OPERATORS Tokens. First of all, I take the first character of the Lexeme and check if the next element is an Equal Symbol, then it might be ">=" "<=" or "==" Comparison Operator, and create a new COMPARISON_OPERATOR Token. Otherwise, I check if the current symbol is one of ">" or "<", and if it is not, then I create a ASSIGN_OPERATOR Token, because the symbol is "=", in the other case, then it is still a COMPARISON_OPERATOR Token:

```
...
class Lexer:
    ...
    def make_comparison(self):
        operator = self.line.take()  # Take the first character of
           the comparison operator

        # Check if the next character is "=" to determine if it's a
            complete comparison operator
        if self.line.next() == "=":
            operator += self.line.take()
            return self.new_token("COMPARISON_OPERATOR")

        # If the next character is not "=", treat the standalone
           character as an assignment operator
```

```
        if operator == ">" or operator == "<":
            return self.new_token("COMPARISON_OPERATOR")
        else:
            return self.new_token("ASSIGN_OPERATOR")
    ...
```

- Here I provided the Tokenization of COMMENT_BLOCK Tokens. In the Language I had, there are two Commenting possibilities:
  1. **COMMENT_LINE**: using "", that will comment the symbols after this symbol,
  2. **COMMENT_BLOCK**: using "/" and "/" that will comment all from inside.

  First of all, I take the first character of the Lexeme and check if the next one is "*" and then take all the elements until it finds "/" symbols. In case that the comment block was not closed, it will raise an error with the message that says that the comment block should be enclosed.

```
...
class Lexer:
    ...
    def make_comment_block(self):
        self.line.take()
        if self.line.next() == "*":
            self.line.take()
            while True:
                if self.line.next() == "*":
                    self.line.take()
                    if self.line.next() == "/":
                        self.line.take()
                        return self.new_token("COMMENT_BLOCK")
                elif self.line.finished():
                    raise LexerError(self.new_token("COMMENT_BLOCK"
                        ), "Block comment not terminated")
                else:
                    self.line.take()
        else:
            return self.make_operator()
```

```
    ...
```

- Here I provided the Tokenization of Keyword Tokens. In the Language I had, there are multiple keywords:
  1. **FILE_TYPE**: csv, txt, json, excel, console,
  2. **IMAGE_TYPE**: png, jpg,
  3. **VARIABLE_TYPE**: Data, Formula, dataset, name,
  4. **PLOT_TYPE**: graph, pie, bar, hist, plot,
  5. **ID**: any string that starts with a letter.

  First of all, I take the whole Lexeme, and check if they are in one of the mentioned Token Types Enum classes. If they are not, then it is an ID Token = Identifier.

```
...
class Lexer:
    ...
    def make_ID(self):
        while self.line.next().isalnum() and not self.line.finished
           ():
            self.line.take()

        if self.line.taken().lower() in FileType:
            return self.new_token("FILE_TYPE")

        if self.line.taken().lower() in ImageType:
            return self.new_token("IMAGE_TYPE")

        if self.line.taken().lower() in PlotType:
            return self.new_token("PLOT_TYPE")

        if self.line.taken() in VariableType:
            return self.new_token("VARIABLE_TYPE")

        if self.line.taken() == "range":
            return self.new_token("RANGE")
```

```
        return self.new_token("ID")
    ...
```

- There are more Tokens and, therefore, Tokenization Methods, but I listed some of the most important and most unique ones, because there are also methods that are only 2 lines of code length and are similar with the ones above.
- In the Main class, I decided to give User the opportunity to choose between 3 methods of Input:
  1. Manual line by line (will check each line instantly):

```
...
def main():
    ...
    elif type_input.lower() == "1":
        print("Enter lines of code. Type 'exit' to finish.")
        while True:
            line = input("> ")
            if line == "exit":
                break

            try:
                source_line = SourceLine(line)
                tokens = lexer.make_tokens(source_line)

                print("TOKENS:")
                for token in tokens:
                    print(token)

                json_object = json.dumps(convert(tokens), indent
                   =4)

                with open("./Laboratory-Work-3-Lexer/
                   ExamplePrograms/Tokenized_Manual_Input.json",
                    "w") as outfile:
```

```
                    outfile.write(json_object)
            except LanguageError as error:
                print(error)
    ...
```

2. Manual all lines ot once (will check the lines altogether):

```
...
def main():
    ...
    if type_input.lower() == "c":
      lines = ""
      print("Enter lines of code. Type 'exit' to finish.")
      while True:
          line = input("> ")
          if line == "exit":
              break

          lines += line + "\n"

      try:
          source_line = SourceLine(lines)
          tokens = lexer.make_tokens(source_line)

          print("TOKENS:")
          for token in tokens:
              print(token)

          json_object = json.dumps(convert(tokens), indent=4)

          with open("./Laboratory-Work-3-Lexer/ExamplePrograms/
             Tokenized_Manual_Input.json", "w") as outfile:
              outfile.write(json_object)
      except LanguageError as error:
```

```
            print(error)
    ...
```

3. File (will tokenize contents of a .txt file). Additionally, here I decided to export the results of the Tokenization in a json file that will hold every Token by their Type and information of the placement - Index by the variable of Locales:

```
...
def main():
    ...
      elif type_input.lower() == "f":
      file_path = select_file()

      if not file_path:
          print("No file selected.")
          return

      with open(file_path) as f:
          lines = f.read()

      try:
          source_line = SourceLine(lines)
          tokens = lexer.make_tokens(source_line)

          print("TOKENS:")
          for token in tokens:
              print(token)

          json_object = json.dumps(convert(tokens), indent=4)

          with open(
                  f"./Laboratory-Work-3-Lexer/ExamplePrograms/
                     Tokenized_{os.path.splitext(os.path.
                     basename(file_path))[0]}.json",
```

```
                    "w") as outfile:
                outfile.write(json_object)
        except LanguageError as error:
            print(error)
    ...
```

- Also, by the use of Tkinter libray, I provide User to choosem manually a .txt File from the Explorer in order to tokenize its insides:

```
...
def select_file():
    root = tk.Tk()
    root.withdraw()
    root.attributes("-topmost", True)

    file_path = filedialog.askopenfilename(initialdir="./
       Laboratory_Work_3-Lexer/ExamplePrograms",
                                           title="Select a file")

    return file_path
...
```

- As I mentioned, I made a conversion into a JSON File in order to display the results of the Tokenization. In this method I iterate over the list of Tokens, and separate them by the ": " symbol that delimits the Lexeme from the Type. Additionaly, I create a Dictionary that will hold the following structure:

```
{
Type :
    {
    Lexeme :
        [
        START_POS,
        END_POS
        ]
    }
```

```
}
```

- Here is the method:

```
...
def convert(lst):
    res_dict = {}
    for i in range(0, len(lst)):
        key = lst[i].kind
        value = lst[i].string
        locale = lst[i].locale
        if key in res_dict:
            res_dict[key].update({value: locale})
        else:
            res_dict[key] = {value: locale}
    return res_dict
```

Conclusions / Results

I present here one output for the Laboratory Work nr.3.

First part of the console output is the general information about the laboratory work, student and group:

```
Laboratory Work 3 - Lexer
Student: Gusev Roman
Group: FAF-222
```

After that, user is asked to choose the method of input they want:

```
MY LEXER:
Choose the type of the input:
F - File
C - Console all lines together
L - Console line-by-line
```

After that is the prompt for user:

```
Your choice: ___
```

I will provide example of inputs for each type of Input (Each Line from the console starts with ">" symbol to show where to start):

- Manual line-by-line:

```
Enter lines of code. Type 'exit' to finish.
> Data data = ReadFrom("/path1/file1")
TOKENS:
VARIABLE_TYPE: 'Data'
ID: 'data'
ASSIGN_OPERATOR: '='
READ_FROM: 'ReadFrom'
LPAREN: '('
PATH: '"/path1/file1"'
RPAREN: ')'
EOF: 'EOF'
```

```
> Formula formula 1
TOKENS:
VARIABLE_TYPE: 'Formula'
ID: 'formula'
NUMBER: '1'
EOF: 'EOF'
```

- Manual line-by-line:

```
    Enter lines of code. Type 'exit' to finish.
    > Data data = ReadFrom("/path1/file1")
    TOKENS:
    VARIABLE_TYPE: 'Data'
    ID: 'data'
    ASSIGN_OPERATOR: '='
    READ_FROM: 'ReadFrom'
    LPAREN: '('
    PATH: '"/path1/file1"'
    RPAREN: ')'
    EOF: 'EOF'
    > Formula formula 1
    TOKENS:
    VARIABLE_TYPE: 'Formula'
    ID: 'formula'
    NUMBER: '1'
    EOF: 'EOF'
```

and the JSON Contents (Note that each input will modify the JSON Contents):

```
{
    "VARIABLE_TYPE": {
        "Formula": [
            0,
            7
        ]
```

```
    },
    "ID": {
        "formula": [
            8,
            15
        ]
    },
    "NUMBER": {
        "1": [
            16,
            17
        ]
    },
    "EOF": {
        "EOF": [
            17,
            17
        ]
    }
}
```

- Manual all lines at once:

```
Enter lines of code. Type 'exit' to finish.
> Formula 1
> Data csv
> # alpha
> exit
TOKENS:
VARIABLE_TYPE: 'Formula'
NUMBER: '1'
VARIABLE_TYPE: 'Data'
FILE_TYPE: 'csv'
COMMENT_LINE: '# alpha'
```

```
EOF: 'EOF'
```

and the JSON Contents:

```
{
  "VARIABLE_TYPE": {
      "Formula": [
          0,
          7
      ],
      "Data": [
          10,
          14
      ]
  },
  "NUMBER": {
      "1": [
          8,
          9
      ]
  },
  "FILE_TYPE": {
      "csv": [
          15,
          18
      ]
  },
  "COMMENT_LINE": {
      "# alpha": [
          19,
          26
      ]
  },
  "EOF": {
```

```
      "EOF": [
          27,
          27
      ]
  }
}
```

- File input:
  - Here are the contents of the file:

```
# This is Table Data read from TXT File;
Data tableData = ReadFrom("/path1/folder1/file.txt");
VisualData(console) dataset=(tableData);
ExportToFile("/path1/") dataset = (tableData) name = (new.txt);
# Operators
1.11 / 1
1 // 2
1.1 * 1
1 ** 2
1 + 1
1 - 1
1 == 1
2 >= 1
3 <= 1
1 = 2
1 < 2
2 > 3

Formula formulaData = formula[x**2 + x*2 + y];
VisualFormula(formulaData) range=(1,2);
ExportToImage("/path2/") graph(formulaData) name=(newGraph.png);
/* Line 1
Data excelData = ReadFrom("/path2/folder1/file.xlss");
Line 3 */
```

– Console Output:

```
TOKENS:
COMMENT_LINE: '# This is Table Data read from TXT File;'
VARIABLE_TYPE: 'Data'
ID: 'tableData'
ASSIGN_OPERATOR: '='
READ_FROM: 'ReadFrom'
LPAREN: '('
PATH: '"/path1/folder1/file.txt"'
RPAREN: ')'
SEMICOLON: ';'
VISUALIZATION_TYPE: 'VisualData'
LPAREN: '('
FILE_TYPE: 'console'
RPAREN: ')'
VARIABLE_TYPE: 'dataset'
ASSIGN_OPERATOR: '='
LPAREN: '('
ID: 'tableData'
RPAREN: ')'
SEMICOLON: ';'
EXPORT_TO_TYPE: 'ExportToFile'
...
```

and the JSON Contents are presented in the link [4]:

As a conclusion to this Laboratory Work nr.3, I can say that I accomplished the given task, specifically:

- Implement your own Lexer.

Also, I managed to understand better the concept of Lexers, Tokenization, Tokens and Lexemes and their main difference. At the same time, I understood how any Compiler "reads" the code and separates each Lexeme it finds when it analyzes the written code.

Bibliography

[1] "Lexer and Parser Definition | IntelliJ Platform Plugin SDK." n.d. IntelliJ Platform Plugin SDK Help. Accessed March 16, 2024. https://plugins.jetbrains.com/docs/intellij/lexer-and-parser-definition.html.

[2] "Kaleidoscope: Kaleidoscope Introduction and the Lexer — LLVM 19.0.0git Documentation." n.d. Llvm.org. Accessed March 16, 2024. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html.

[3] "How to Create Your Own Programming Language - Episode 2: The Lexer." n.d. Www.youtube.com. Accessed March 17, 2024. https://www.youtube.com/watch?v=N103OVKmDR4.

[4] "LFA-Laboratory-Works/Laboratory-Work-3-Lexer/ExamplePrograms/Tokenized_example_1.Json at Main · Ghenntoggy1/LFA-Laboratory-Works." n.d. GitHub. Accessed March 17, 2024. https://github.com/Ghenntoggy1/LFA-Laboratory-Works/blob/main/Laboratory-Work-3-Lexer/ExamplePrograms/Tokenized_example_1.json.