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 6:
Parser & Abstract Syntax Tree.

Course: Formal Languages & Finite Automata

Student: Gusev Roman

Group: FAF-222

Professor: Cojuhari Irina

University Assistant: Crețu Dumitru

Chișinău, 2024

Content

Theory

- **Definitions:**
  - **Parser** - The parser is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as Syntax Analyzer [1];
  - **Abstract Syntax Tree** - Abstract Syntax Tree is a kind of tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code [2].

Objectives

- Get familiar with parsing, what it is and how it can be programmed;
- Get familiar with the concept of AST;
- In addition to what has been done in the 3rd lab work do the following:
  - In case you didn't have a type that denotes the possible types of tokens you need to:
    - Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens;
    - Please use regular expressions to identify the type of the token;
  - Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work;
  - Implement a simple parser program that could extract the syntactic information from the input text.

Implementation Description

- For the start, I decided to implement the Lexer that was developed in Laboratory Work 3 - Lexer
- For the current Laboratory Work, I decided to implement Parser for the same 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 Parser 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 modified was the Token class themselves. I have several Tokens that I decided to hold in Enum classes, and that have been used in the Lexer to classify them and, therefore, by Parser. I added several new Tokens:

```
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 modified the Token class, that will describe what type of Lexeme I have in input. I added 3 methods for representation in the Abstract Syntax Tree, for checking if the Token is of a specific kind and if the Token has a specific string value in it.

```
class Token:
    ...
    def ast_repr(self, _):
```

```
        return f"{self.kind}        {self.string}"

    def has(self, *strings):
        return self.string in strings

    def of(self, *kinds):
        return self.kind in kinds
    ...
```

- Next Class developed for the Parser implementation was "Node", that will hold the structure of the Abstract Syntax Tree. This class is an Abstract class, that will be inherited by other classes that will represent the Nodes of the AST. The class has several methods that will be used in the subclasses, such as:
  - **line**: that will return the line of the Node,
  - **nodes**: that will return the Nodes of the current Node,
  - **ast_repr**: that will represent the Node in the AST,
  - **mark**: that will mark the Nodes in the AST,
  - **construct**: that will construct the Node from the Parser.

```
from abc import ABC, abstractmethod
class Node(ABC):
    @property
    def line(self):
        return self.nodes()[0].line

    @abstractmethod
    def nodes(self):
        pass

    def __repr__(self):
        return self.ast_repr()

    def ast_repr(self, prefix="\t"):
        ast_string = "("
```

```
        ast_string += type(self).__name__ + "("
        string = type(self).__name__
        nodes = self.nodes()

        for i, node in enumerate(nodes):
            at_last = (i == len(nodes) - 1)
            symbol = "           " if at_last else "           "
            prefix_symbol = "" if at_last else "   "

            node_string = node.ast_repr(f"{prefix}{prefix_symbol}\t
               ")
            string += f"\n{prefix}{symbol} {node_string}"

        ast_string += ")"
        return string

    def mark(self):
        for node in self.nodes():
            node.mark()

    @classmethod
    @abstractmethod
    def construct(cls, parser):
        pass
```

- Next class developed was the class for the Primary Nodes, that will hold the Tokens that are not Binary, i.e., that are represented by a single derivation. This class will hold the Token itself and will have the methods to return the Nodes of the current Node and to represent the Node in the AST.
- Next class - BinaryNode, that will hold the Tokens that are Binary, i.e., that are represented by 3 derivations. This class will hold the Left Node, Operator Node and Right Node. Also, it will have the methods to return the Nodes. Besides that it will have a method to construct the Binary Node from the Parser.

```
from abc import ABC, abstractmethod
```

```
...
class PrimaryNode(Node, ABC):
    def __init__(self, token):
        self.token = token

    def nodes(self):
        return [self.token]

    def ast_repr(self, prefix="\t"):
        return f"{self.token.kind}        {self.token.string}"
...
class BinaryNode(Node, ABC):
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right

    def nodes(self):
        return [self.left, self.op, self.right]

    @classmethod
    def construct_binary(cls, parser, make, part, ops):
        node = part.construct(parser)

        while parser.next().has(*ops):
            op = parser.take()
            right = part.construct(parser)
            node = make(node, op, right)

        return node
```

- After that, I developed the class for Parser that will hold the tokens from the Lexer and will construct the AST from them. This class holds the Tokens and the Index of the current Token. It has several methods that will be used in the Parser:

  - **expression**: that will return the Expression Node,
  - **formula**$_{content}$: *that will return the Additive Expression Node*, **next** : *that will return the next Token*,
  - **take**: that will take the next Token,
- **expecting**$_{has}$: *that will check if the next Token has a specific string value*,

- Also, this class has 2 properties that will return the Expression and Formula Content Nodes, that will be used further in expressions to avoid the circular imports in classes.

```
...
class Parser:
    @property
    def expression(self):
        return Expression

    @property
    def formula_content(self):
        return Additive_Expression

    def __init__(self):
        self.tokens = None
        self.index = 0

    def next(self):
        return self.tokens[self.index]

    def take(self):
        token = self.next()
        self.index += 1
        return token

    def expecting_has(self, *strings):
        if self.next().has(*strings):
            return self.take()

        raise ParserError(self.next(), f"Expected one of: {strings}
```

```
            ")

    def expecting_of(self, *kinds):
        if self.next().of(*kinds):
            return self.take()

        raise ParserError(self.next(), f"Expected one of the type:
            {kinds}")

    def build_ast(self, tokens):
        self.tokens = tokens
        node = Expression.construct(self)
        if self.next().of("EOF"):
            return node

        raise ParserError(self.next(), f"Unexpected token: {self.
            next()}")
```

- Then, I started to define the classes for each expression in the Language that I developed. I started with the Program_Expression class, that will hold the Commands of the Program. This class will have the methods to return the Nodes of the current Node and to represent the Node in the AST. This class will add new Command Expressions to the list of Commands until the End of File.

```
...
class Program_Expression(Node):
    def __init__(self, commands):
        self.commands = commands

    def nodes(self):
        return self.commands

    @classmethod
    def construct(cls, parser):
        commands = []
```

```
        while True:
            if parser.next().of("EOF"):
                break
            command = Command_Expression.construct(parser)
            commands.append(command)

        return Program_Expression(commands)
    ...
```

- Next Expression developed was the Command_Expression class, that will hold the Commands of the Program. This class has the methods to return the Nodes of the current Node and to represent the Node in the AST. Also, this class may derive into 3 different classes - Variable_Expression, Comment_Expression and Command_Expression.

```
...
class Command_Expression(Node):
    def __init__(self, command, right):
        self.command = command
        self.right = right

    def nodes(self):
        return [self.command, self.right]

    @classmethod
    def construct(cls, parser):
        if parser.next().of("COMMENT_LINE", "COMMENT_BLOCK"):
            return Comment_Expression(parser.take())

        command = Variable_Expression.construct(parser)

        right = parser.expecting_has(";")

        return Command_Expression(command, right)
```

- Here I provide 3 classes that represent Commenting methods - Block and Line comments. They are taking the token that are of Type COMMENT_LINE and COMMENT_BLOCK and construct their derivations in the AST.

```
...
class Comment_Expression(Node):
    def __init__(self, comment):
        self.comment = comment

    def nodes(self):
        return [self.comment]

    @classmethod
    def construct(cls, parser):
        if parser.next().of("COMMENT_LINE"):
            return Comment_Line_Expression(parser)
        else:
            return Comment_Block_Expression(parser)

class Comment_Block_Expression(Node):
    def __init__(self, comment_line):
        self.comment_line = comment_line

    def nodes(self):
        return [self.comment_line]

    @classmethod
    def construct(cls, parser):
        return Comment_Block_Expression(parser.expecting_of("
           COMMENT_BLOCK"))

class Comment_Line_Expression(Node):
    def __init__(self, comment_line):
        self.comment_line = comment_line
```

```
    def nodes(self):
        return [self.comment_line]

    @classmethod
    def construct(cls, parser):
        return Comment_Line_Expression(parser.expecting_of("
           COMMENT_LINE"))
```

- The next class developed was the Variable_Expression class, that will hold the Variables of the Program. This class derives into 2 different ways - Data declaration and Formula declaration. This class has the methods to return the Nodes of the current Node and to represent the Node in the AST.
- If the keyword is "Formula", then the Expression will be Additive_Expression, otherwise, it will be Read_From_Path_Expression.

```
...
class Variable_Expression(Node):
    def __init__(self, keyword, identifier, op, expression):
        self.keyword = keyword
        self.identifier = identifier
        self.op = op
        self.expression = expression

    def nodes(self):
        return [self.keyword, self.identifier, self.op, self.
           expression]

    @classmethod
    def construct(cls, parser):
        # Expect 'Formula' or 'Data' keyword
        keyword = parser.expecting_has("Formula", "Data")

        # Expect Identifier (ID)
        identifier = Identifier.construct(parser)
```

```
        # Expect '=' symbol
        op = parser.expecting_has("=")

        # Parse Expression
        if keyword.has("Formula"):
            expression = Additive_Expression.construct(parser)
        else:
            expression = Read_From_Path_Expression.construct(parser
               )

        return Variable_Expression(keyword, identifier, op,
           expression)
```

- The next class is Read From Path expression that will hold the Path of the Data that will be read from. It is designed in the same manner as previous classes, that will have the methods to return the Nodes of the current Node and keeping track of the Tokens and their types. Also, this class includes another class - Path_Expression, that will hold the specific path of the Data that will be read from.

```
...
class Read_From_Path_Expression(Node):
    def __init__(self, keyword, left, path, right):
        self.keyword = keyword
        self.left = left
        self.path = path
        self.right = right

    def nodes(self):
        return [self.keyword, self.left, self.path, self.right]

    @classmethod
    def construct(cls, parser):
        keyword = parser.expecting_of("READ_FROM")
        left = parser.expecting_of("LPAREN")
```

```
        path = Path_Expression.construct(parser)
        right = parser.expecting_of("RPAREN")

        return Read_From_Path_Expression(keyword, left, path, right
            )
...
class Path_Expression(Node):
    def __init__(self, expression):
        self.expression = expression

    def nodes(self):
        return [self.expression]

    @classmethod
    def construct(cls, parser):
        expression = parser.expecting_of("PATH")
        return Path_Expression(expression)
```

- For the Formula declaration, I implemented parse methods for the Additive Expression, that will hold the mathematical addition or Subtraction of the Variables. This class has the methods to return the Nodes of the current Node and to represent the Node in the AST. Then, if the Operator is "+" or "-", then the right Node will be the Additive Expression, otherwise, it will be the Multiplicative Expression. Then, if the Operator is "*" or "/", then the right Node will be the Multiplicative Expression, otherwise, it will be the Exponential Expression. The same structure is persisted across all the classes that represent the mathematical expressions, such as Unary Expressions, Primary Expressions that involves parenthesis and so on.

```
...
class Additive_Expression(BinaryNode):
    @classmethod
    def construct(cls, parser):
        return cls.construct_binary(parser, cls,
            Multiplicative_Expression, ["+", "-"])
...
```

```
7
8  class Multiplicative_Expression(BinaryNode):
9      @classmethod
10     def construct(cls, parser):
11         return cls.construct_binary(parser, cls,
               Exponential_Expression, ["*", "/"])
12 ...
13 class Exponential_Expression(BinaryNode):
14     @classmethod
15     def construct(cls, parser):
16         node = Unary_Expression.construct(parser)
17
18         if not parser.next().has("**"):
19             return node
20
21         op = parser.take()
22         right = Exponential_Expression.construct(parser)
23         return Exponential_Expression(node, op, right)
24 ...
25 class Unary_Expression(Node):
26     def __init__(self, op, expression):
27         self.op = op
28         self.expression = expression
29
30     def nodes(self):
31         return [self.op, self.expression]
32
33     @classmethod
34     def construct(cls, parser):
35         if parser.next().has("+", "-"):
36             op = parser.take()
37             expression = Unary_Expression.construct(parser)
38             return Unary_Expression(op, expression)
```

```
        return Primary_Expression.construct(parser)
```

- For the Primary Expressions class, I implemented the classes for the Number, Identifier and IdentifierOrNumber, that will be required to be in the Primary Expressions. The Number class will hold the Number Token, the Identifier class will hold the Identifier Token, and last class will be a general if the input is not met the previous 2 classes. This class has the methods to return the Nodes of the current Node.

```
...
class Primary_Expression(Node):
    def __init__(self, left, expression, right):
        self.left = left
        self.expression = expression
        self.right = right

    def nodes(self):
        return [self.left, self.expression, self.right]

    @classmethod
    def construct(cls, parser):
        if not parser.next().has("("):
            if parser.next().of("ID"):
                return Identifier.construct(parser)
            elif parser.next().of("NUMBER"):
                return Number.construct(parser)
            else:
                return IdentifierOrNumber.construct(parser)

        left = parser.take()
        expression = parser.formula_content.construct(parser)
        right = parser.expecting_has(")" if left.has("(") else left
           .string)

```

```
        return Primary_Expression(left, expression, right)
...
class Number(PrimaryNode):
    @classmethod
    def construct(cls, parser):
        return Number(parser.expecting_of("NUMBER"))


...
class Identifier(PrimaryNode):
    @classmethod
    def construct(cls, parser):
        return Identifier(parser.expecting_of("ID"))
```

- In the Main class, I decided to modify how User choose between 3 methods of Input and will display the AST of the input:

```
parser = Parser()
tree = parser.build_ast(tokens)
print("TOKENS:")
for token in tokens:
    print(token)

print("AST:")
print(tree.ast_repr())
```

- Also, by the use of Tkinter libray, I decided to leave the User to choose manually a .txt File from the Explorer in order to tokenize its insides as it was in the previous Lexer Laboratory Work:

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

    file_path = filedialog.askopenfilename(initialdir="./Laboratory
```

```
        -Work-6-Parser-AST/ExamplePrograms",
                                              title="Select a file")

    return file_path
...
```

Conclusions / Results

I present here one output for the Laboratory Work nr.6.

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

```
Laboratory Work 6 - Parser and AST
Student: Gusev Roman
Group: FAF-222
```

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

```
MY LEXER AND PARSER:
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: ')'
SEMICOLON: ';'
EOF: 'EOF'
```

```
AST:
Program_Expression
  └── Command_Expression
      ├── Variable_Expression
      │   ├── VARIABLE_TYPE ── Data
      │   ├── ID ── data
      │   ├── ASSIGN_OPERATOR ── =
      │   └── Read_From_Path_Expression
      │       ├── READ_FROM ── ReadFrom
      │       ├── LPAREN ── (
      │       ├── Path_Expression
      │       │   └── PATH ── "/path1/file1"
      │       └── RPAREN ── )
      └── SEMICOLON ── ;
```

- Manual all line at once:

```
Enter lines of code. Type 'exit' to finish.
> Formula x = (-5+6);
> Data y = ReadFrom("/path1");
> exit
TOKENS:
VARIABLE_TYPE: 'Formula'
ID: 'x'
ASSIGN_OPERATOR: '='
LPAREN: '('
OPERATOR: '-'
NUMBER: '5'
OPERATOR: '+'
NUMBER: '6'
RPAREN: ')'
SEMICOLON: ';'
VARIABLE_TYPE: 'Data'
ID: 'y'
```

```
ASSIGN_OPERATOR: '='
READ_FROM: 'ReadFrom'
LPAREN: '('
PATH: '"/path1"'
RPAREN: ')'
SEMICOLON: ';'
EOF: 'EOF'
AST:
Program_Expression
   ├── Command_Expression
   │   ├── Variable_Expression
   │   │   ├── VARIABLE_TYPE ── Formula
   │   │   ├── ID ── x
   │   │   ├── ASSIGN_OPERATOR ── =
   │   │   └── Primary_Expression
   │   │       ├── LPAREN ── (
   │   │       ├── Additive_Expression
   │   │       │   ├── Unary_Expression
   │   │       │   │   ├── OPERATOR ── -
   │   │       │   │   └── NUMBER ── 5
   │   │       │   ├── OPERATOR ── +
   │   │       │   └── NUMBER ── 6
   │   │       └── RPAREN ── )
   │   └── SEMICOLON ── ;
   └── Command_Expression
       ├── Variable_Expression
       │   ├── VARIABLE_TYPE ── Data
       │   ├── ID ── y
       │   ├── ASSIGN_OPERATOR ── =
       │   └── Read_From_Path_Expression
       │       ├── READ_FROM ── ReadFrom
       │       ├── LPAREN ── (
       │       ├── Path_Expression
       │       │   └── PATH ── "/path1"
```

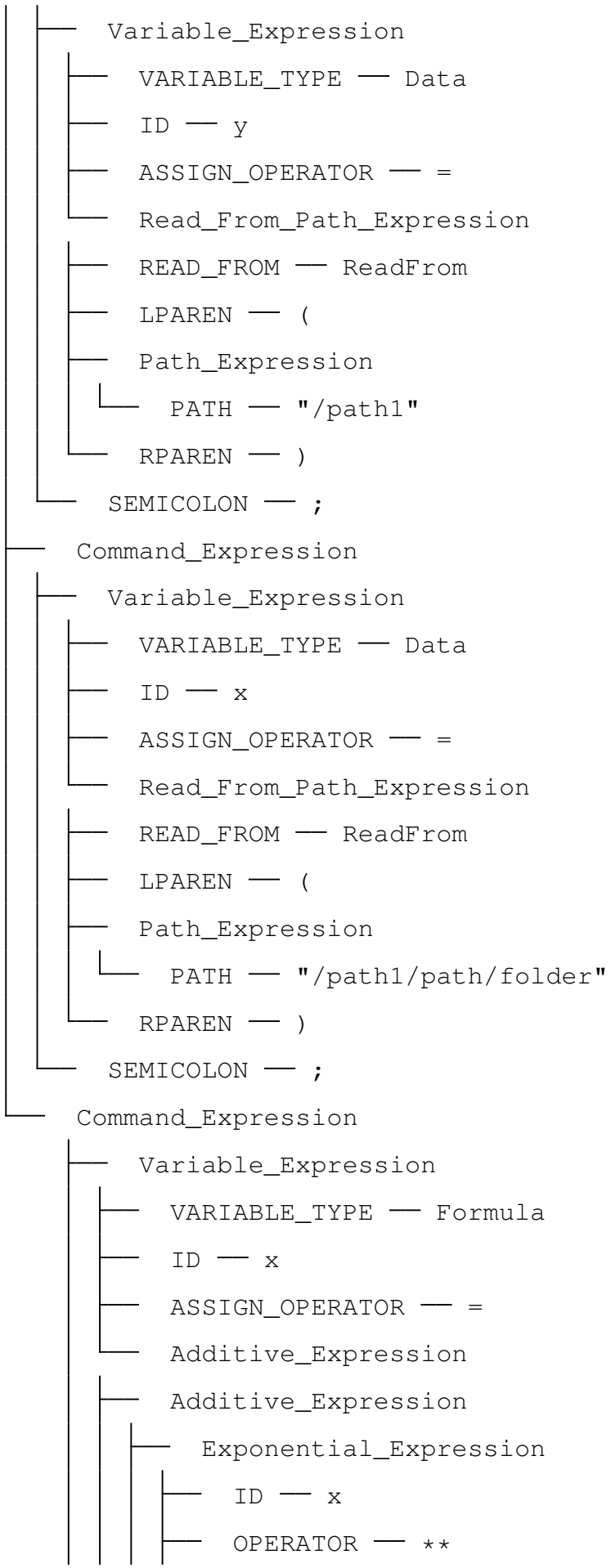
```
    │   └── RPAREN ── )
    └── SEMICOLON ── ;
```

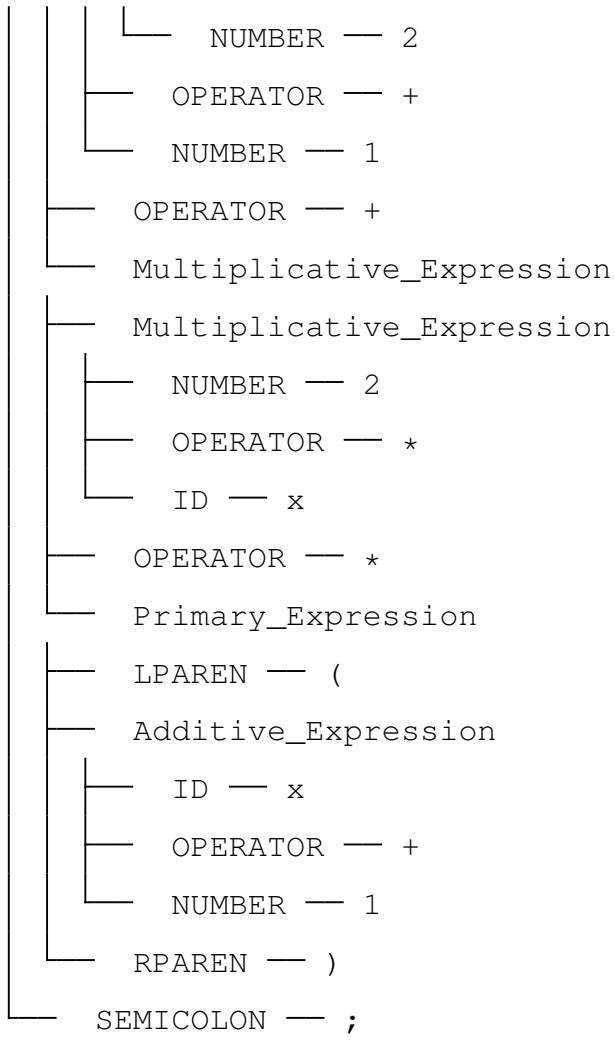
- File input:
  - Here are the contents of the file:

```
Formula x = (-5+6);
# This is Comment Line :)
Data y = ReadFrom("/path1");
Data x = ReadFrom("/path1/path/folder");
Formula x = x**2+1+2*x*(x+1);
```

  - Console Output:

```
...
AST:
Program_Expression
    ├── Command_Expression
    │   ├── Variable_Expression
    │   │   ├── VARIABLE_TYPE ── Formula
    │   │   ├── ID ── x
    │   │   ├── ASSIGN_OPERATOR ── =
    │   │   └── Primary_Expression
    │   │   ├── LPAREN ── (
    │   │   ├── Additive_Expression
    │   │   │   ├── Unary_Expression
    │   │   │   │   ├── OPERATOR ── -
    │   │   │   │   └── NUMBER ── 5
    │   │   │   ├── OPERATOR ── +
    │   │   │   └── NUMBER ── 6
    │   │   └── RPAREN ── )
    │   └── SEMICOLON ── ;
    ├── Comment_Expression
    │   └── COMMENT_LINE ── # This is Comment Line :)
    ├── Command_Expression
```


Variable_Expression
VARIABLE_TYPE — Data
ID — y
ASSIGN_OPERATOR — =
Read_From_Path_Expression
READ_FROM — ReadFrom
LPAREN — (
Path_Expression
PATH — "/path1"
RPAREN —)
SEMICOLON — ;
Command_Expression
Variable_Expression
VARIABLE_TYPE — Data
ID — x
ASSIGN_OPERATOR — =
Read_From_Path_Expression
READ_FROM — ReadFrom
LPAREN — (
Path_Expression
PATH — "/path1/path/folder"
RPAREN —)
SEMICOLON — ;
Command_Expression
Variable_Expression
VARIABLE_TYPE — Formula
ID — x
ASSIGN_OPERATOR — =
Additive_Expression
Additive_Expression
Exponential_Expression
ID — x
OPERATOR — **



As a conclusion to this Laboratory Work nr.6, I can say that I accomplished the given task, specifically:

- Implement your own Lexer.
- Build the Abstract Syntax Tree.

Also, I managed to understand better the concept of Parsers, Abstract Syntax Trees, Tokens and Lexemes and their main usages. 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] "Types of Parsers in Compiler Design," GeeksforGeeks, Jul. 26, 2019. Available: https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/. [Accessed: Apr. 26, 2024].

[2] "Abstract Syntax Tree (AST) in Java," GeeksforGeeks, Aug. 11, 2021. Available: https://www.geeksforgeeks.org/abstract-syntax-tree-ast-in-java/. [Accessed: Apr. 26, 2024].

[3] Introspective Thinker, How to Create Your Own Programming Language - Episode 3: The Parser, (Jan. 28, 2023). Available: https://www.youtube.com/watch?v=yEBUod_e8Yk. [Accessed: Apr. 27, 2024]