Assignment 1: String

• Write a grammar to generate string like First name and last name, name with age or telephone number etc.
• Generate Antlr lexer and parser.
• Check parse tree with Test file.

Write a Grammar

The grammar below describes the structure of the strings that need to be parsed:

```
grammar Gramar;

names: (transaction (';' transaction)*) EOF;

transaction: person operation;

person: name '('age ',' phone')';

name: firstName lastName;

operation: type ammount;

firstName: STRING;
lastName: STRING;
type: STRING;
ammount: DIGITS;
age: DIGITS;
phone: PHONE;

DIGITS: [0-9]+;
STRING: [a-zA-Z]+;
PHONE: [+0-9]+;
WS: [ \t\r\n]+ -> skip;
```

This grammar defines the structure for:

- names: A series of transactions separated by semicolons.
- transaction: A combination of person and operation.
- person: A name with personal information(age and phone).
- name: Consists of a firstName and lastName.
- operation: A type and an ammount.

Generate ANTLR Lexer and Parser

Components of ANTLR

1. **Grammar File (.g4 Files)**:
    - These file define the lexical and syntactical structure of a language.
    - They include rules for how different constructs of the language are identified and organized.
2. **Lexer**:
    - The lexer (or tokenizer) processes the input text and divides it into tokens. Each token represents a meaningful unit such as a keyword, identifier, number, etc.
3. **Parser**:
    - The parser takes the tokens produced by the lexer and arranges them according to the grammar rules, producing a parse tree (or abstract syntax tree, AST).

To generate Lexer and Parser based on my Gramar I used ANTLR package to build python components.

```
(.venv)(base) ~/Desktop/uni/aut/a1 git:[master]
antlr4 -Dlanguage=Python3 Gramar.g4
```

Where antlr4 satnds for:

```
(.venv)(base) ~/Desktop/uni/aut/a1 git:[master]
alias antlr4='java -jar /Users/keru/Desktop/uni/aut/a1/antlr-4.13.1-complete.jar'
```

Files Generated

After running the command, you will typically see files such as:

- **Gramar.interp:** This file contain information about Gramar file.

  Token Literal Names:

  This section lists the literal names of tokens as defined in the grammar. The literals correspond to the exact characters or strings recognized by the lexer. null indicates a token that does not have a fixed literal representation (e.g., identifiers, numbers).
  So here are listed all fixed tokens in gramar

  Token Symbolic Names:

  This section lists the symbolic names for tokens which are used in Gramar rules. indicates a token that does not have a specific symbolic name (e.g., operators or punctuation).
  Here are listed all “String type rules” defined in Gramar file.

Rule names:

```
rule names:
names
transaction
person
name
operation
firstName
lastName
type
ammount
age
phone
```

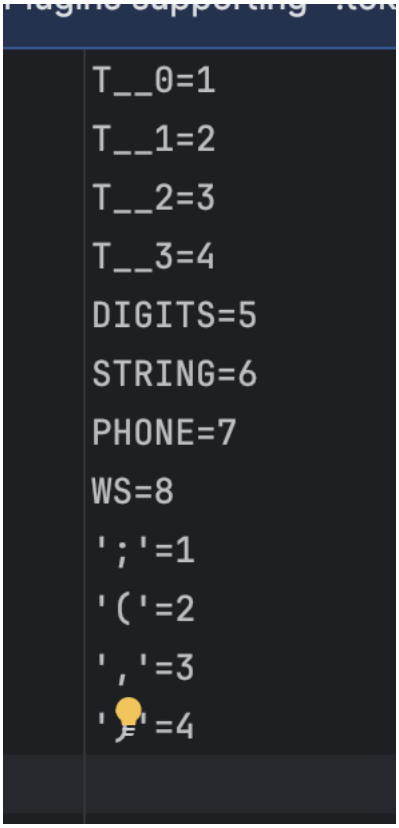
This section lists the names of the parsing rules defined in the grammar. These rules represent the structure of the language being parsed.

atn:

```
atn:
[4, 1, 8, 61, 2, 0, 7, 0, 2, 1, 7, 1, 2,
```

The *atn* section contains a serialized form of the Augmented Transition Network. The *ATN* is a state machine used by ANTLR's parser to recognize language constructs.
This serialized data is used internally by the generated parser to efficiently parse input according to the grammar rules.

- **Gramar.tokens** and **GramarLexer.tokens:** files contains the token types and their corresponding numerical values used by the lexer and parser.



Automatic Token Names (T__0, T__1, etc.):
- These are generated for literals in the grammar.
- T__0=1 maps to the first literal ';'.
- T__1=2 maps to '(', and so on.

Explicit Token Names:
- DIGITS=5, STRING=6, PHONE=7, WS=8 are explicitly defined in the grammar.

Literal Mappings:
- The literals are mapped to their token types: ';'=1, '('=2, ','=3, ')'=4.

Purpose
- Maps token names to numeric values for lexer and parser consistency.
- Helps the lexer recognize input sequences and the parser to apply rules correctly.

- **GramarLexer.interp**: file is similar to Grammar.interp, but it specifically pertains to the lexer component of the grammar.

  Also have token literal and symbolic names;

  Also contain rule names :

```
rule names:
T__0
T__1
T__2
T__3
DIGITS
STRING
PHONE
WS
```

  Metadata for Lexer:

```
channel names:
DEFAULT_TOKEN_CHANNEL
HIDDEN

mode names:
DEFAULT_MODE
```

  And atn but different from Gramar.interp.

```
atn:
[4, 0, 8, 47, 6, -1, 2, 0, 7, 0, 2, 1, 7, 1,
```

- **GramarLexer.py:** This lexer class is responsible for tokenizing the input text according to the rules defined in the grammar.

ATN:

```
1 usage
def serializedATN():
    return [
        4,0,8,47,6,-1,2,0,7,0,2,1,7,1,2,2,7,2,2,3,7,3,2,4,7,4,2,5,7
        6,7,6,2,7,7,7,1,0,1,0,1,1,1,1,1,2,1,2,1,3,1,3,1,4,4,4,27,8,
        4,12,4,28,1,5,4,5,32,8,5,11,5,12,5,33,1,6,4,6,37,8,6,11,6,1
        1,7,4,7,42,8,7,11,7,12,7,43,1,7,1,7,0,0,8,1,1,3,2,5,3,7,4,9
```

Uses the same ATN as defined in GramarLexer.interp

Lexer initialization:

```
2 usages
class GramarLexer(Lexer):

    atn = ATNDeserializer().deserialize(serializedATN())

    decisionsToDFA = [ DFA(ds, i) for i, ds in enumerate(atn.decisionToState) ]

    T__0 = 1
    T__1 = 2
    T__2 = 3
    T__3 = 4
    DIGITS = 5
    STRING = 6
    PHONE = 7
    WS = 8

    channelNames = [ u"DEFAULT_TOKEN_CHANNEL", u"HIDDEN" ]

    modeNames = [ "DEFAULT_MODE" ]

    literalNames = [ "<INVALID>",
            "'.'", "'('", "','", "')'" ]

    symbolicNames = [ "<INVALID>",
            "DIGITS", "STRING", "PHONE", "WS" ]

    ruleNames = [ "T__0", "T__1", "T__2", "T__3", "DIGITS", "STRING", "PHONE",
                  "WS" ]

    grammarFileName = "Gramar.g4"

    def __init__(self, input=None, output:TextIO = sys.stdout):
        super().__init__(input, output)
        self.checkVersion("4.13.1")
        self._interp = LexerATNSimulator(self, self.atn, self.decisionsToDFA, PredictionContextCache())
        self._actions = None
        self._predicates = None
```

- Deserializes the ATN and initializes DFA (Deterministic Finite Automata) for each decision state.
- Defines token types (e.g., T__0, DIGITS, STRING).
- Specifies token channels and lexer modes defined in GramarLexer.interp.
- Maps literal and symbolic names to token types.
- Lists the names of lexer rules.
- Initializes the lexer with input and output streams.

- **GramarParser.py:** is responsible for interpreting the tokenized input and building a parse tree according to the grammar rules.

  Serialized ATN
  - Contains the serialized form of the ATN defined in Gramar.interp used by the parser to understand the structure of the input.

  Parser Initialization
  - Defines the grammar file name and deserializes the ATN.
  - Initializes DFA (Deterministic Finite Automaton) for decision states.
  - Sets up the shared context cache for efficient parsing.
  - Defines the literal and symbolic names for tokens.
  - Lists the names of the parser rules.

  Rule Methods
  - Defines methods for each rule in the grammar.
  - Each method represents a parser rule and contains logic to match and process specific parts of the input.

  Rules and Parsing Logic
  - **names**: Matches the entire input sequence and ensures it ends with the EOF token.
  - **transaction**: Matches a person followed by an operation.
  - **person**: Matches a name, followed by age and phone within parentheses.
  - **name**: Matches a firstName followed by a lastName.
  - **operation**: Matches a type followed by an ammount.
  - **firstName, lastName, type**: Matches a string token.
  - **ammount, age**: Matches a digit token.
  - **phone**: Matches a phone token.

  The GramarParser is an implementation of the parser for defined gramar. It uses the token stream produced by the lexer to build a parse tree, matching the input text against the grammar rules. The parser ensures that the input adheres to the defined structure and handles any errors that occur during parsing.

Check Parse Tree with Test File

Text file

```
John Doe (23, +232131123) withdraw 25;
Jane Smith (30, +123456789) deposit 100;
Alice Johnson (45, +987654321) transfer 200;
Bob Brown (22, +112233445) withdraw 50
```

Test script

This script is designed to parse an input file using a lexer and parser generated by ANTLR, and then create a visual representation of the parse tree using Graphviz. Here's a step-by-step explanation of the script:

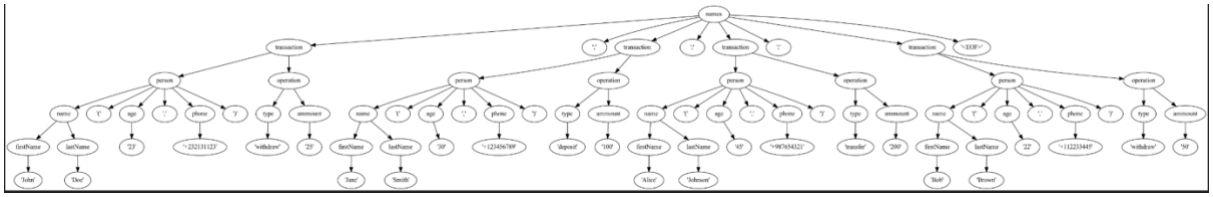
```
1 usage
def main(argv):
    input_stream = FileStream(argv[1], encoding='utf-8')
    lexer = GramarLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = GramarParser(stream)
    tree = parser.names()

    dot = to_dot(tree, parser)
    dot.render( filename: 'parse_tree', format='png')
    print("Parse tree saved as 'parse_tree.png'.")

if __name__ == '__main__':
    if len(sys.argv) != 2:
        sys.exit(1)
    main(sys.argv)
```

Test execution

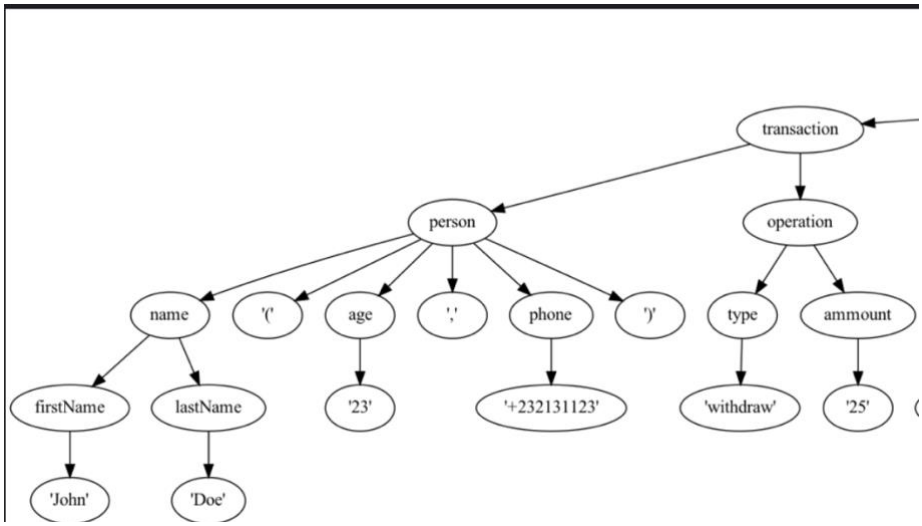
Execute this script on the test.txt with comand

```
(.venv)(base) ~/Desktop/uni/aut/a1 git:[master]
python3 test.py test.txt
Parse tree saved as 'parse_tree.png'.
```

And the result tree saved to the file looks like:



More detailed view:



Overview

During first assignment in Automata I learned syntax to write a grammar file to define the structure of strings, use ANTLR to generate lexer and parser classes, and understand their roles in tokenizing and parsing input text. Also understand on abstract level roles of different components and interactions in ANTLR proccesses.