Topic: Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Cretu Dumitru and cudos to the Vasile Drumea with Irina Cojuhari

Theory

Lexers, short for lexical analyzers, are essential components in compilers and interpreters. They break down the input source code into a sequence of tokens, such as keywords, identifiers, literals, and symbols. These tokens serve as the fundamental building blocks for subsequent stages of the compilation or interpretation process. Lexers typically employ finite automata or regular expressions to efficiently recognize and classify tokens from the input stream. By identifying the syntactic structure of the source code, lexers pave the way for further analysis and processing by subsequent components in the compiler or interpreter pipeline.

Objectives:

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

Implementation description

V1

I have created the following lexer on the basis of the ELSD Project:

```
class Tokenizer:
    def __init__(self,string):
        self._string=string

    def lines_plit(self):
        self._string=self._string.split("\n")
    def tokenize(self):

        response=[]

        for line in self._string:

            line=line.replace("\n","").replace(" ","")
            lexemes=line.split(":")
            if lexemes == ['']:
                continue
            lexemes.insert(1,":")
            for token in lexemes:
                each = {}
                if token.isnumeric():
                    each["type"]= "NUM"
                    if "."in token:
                        each["type"]="FLOAT"
                elif ":" == token:
                    each['type'] = "OPERATOR"
                else:
                    each['type'] = "IDENTIFIER"
                each["value"] = token

                response.append(each)
        return response
```

The way the program works is that when a tokenizer class is instantiated a private parameter string is updated to contain the text that needs to be processed by the lexer. afterwards the text is split into each separate line, whitespace is dropped.

Our DSL looks like:

```
Age: 80
```

and so on for different attributes of a patient, we can easily notice a pattern of LITERAL("Age") OPERATOR(":") NUMBER("80") Which holds true for most of the attributes except ones with String values. Ex:

```
 Pregnancies: 5
Glucose: 130
BloodPressure: 80
SkinThickness: 25
Insulin: 100
BMI: 28.5
DiabetesPedigreeFunction: 0.55
Age: 40
```

This means that with certain exceptions our Language has 3 tokens in each line: LITERAL, OPERATOR, (NUMERICAL|STRING) I split each line based on the operator ':' then insert it into the resulted list. This transforms

```
 Age: 80
```

into

```
['Age',':','80']
```

Based on this newly created list I classify each resulted token into its type and get this:

```
{'type': 'IDENTIFIER', 'value': 'Age'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '40'}
```

Example usage of the Lexer:

```
t=Tokenizer("""
Patient:
Pregnancies: 5
Glucose: 130
BloodPressure: 80
SkinThickness: 25
Insulin: 100
BMI: 28.5
DiabetesPedigreeFunction: 0.55
Age: 40
Outcome: 1

""")
t.lines_plit()
response=t.tokenize()
for i in response:
    print(i)
```

This is the produced Output:

```
{'type': 'IDENTIFIER', 'value': 'Patient'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'IDENTIFIER', 'value': ''}
{'type': 'IDENTIFIER', 'value': 'Pregnancies'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '5'}
{'type': 'IDENTIFIER', 'value': 'Glucose'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '130'}
{'type': 'IDENTIFIER', 'value': 'BloodPressure'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '80'}
{'type': 'IDENTIFIER', 'value': 'SkinThickness'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '25'}
{'type': 'IDENTIFIER', 'value': 'Insulin'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '100'}
{'type': 'IDENTIFIER', 'value': 'BMI'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'IDENTIFIER', 'value': '28.5'}
{'type': 'IDENTIFIER', 'value': 'DiabetesPedigreeFunction'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'IDENTIFIER', 'value': '0.55'}
{'type': 'IDENTIFIER', 'value': 'Age'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '40'}
{'type': 'IDENTIFIER', 'value': 'Outcome'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUM', 'value': '1'}
```

V2

Additionally I have created a v2 of the lexer that identifies each token dynamically, not just based on the len 3, and can be easily extended for a more general purpose DSL

```
class Tokenizer:
    def __init__(self,string):
        self._string=string

    def lines_plit(self):
        self._string=self._string.split("\n")
    def tokenize(self):

        response=[]

        for line in self._string:
            for regex, token in TOKENS:
                match=re.findall(regex,line)
                if not match:
                    continue
                match=match[0] if type(match[0])==tuple else match

                longest_match = max(match, key=lambda match: len(match))
                response.append({"type":token,"value":longest_match})

                line=line.replace(longest_match,"")
        return response
```

This one works by sequentially identifying tokens in the given string. This emans that the order in which tokens are captured is very important and will influence the end result for complex scenarios. But it allows adding new types of tokens in a much easier way compared to the previous one. These are the tokens for the DSL:

```
TOKENS=[
    [r"\A\#.*$", "COMMENT"],
    [r"\b[A-Za-z]+\b","LITERAL"],
    [r":","OPERATOR"],
    [r"[+-]?((\d+(\.\d+)?)|(\.\d+))","NUMERICAL"],
]
```

Example Usage:

```
 Patient:
Pregnancies: 5
Glucose: 130
BloodPressure: 80.5
SkinThickness: 25
Insulin: 100
BMI: 28.5
DiabetesPedigreeFunction: 0.55
Age: 40
Outcome: 1
# damn this is good
""")
t.lines_plit()
response=t.tokenize()
for i in response:
    print(i)
```

Output:

```
{'type': 'LITERAL', 'value': 'Patient'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'LITERAL', 'value': 'Pregnancies'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '5'}
{'type': 'LITERAL', 'value': 'Glucose'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '130'}
{'type': 'LITERAL', 'value': 'BloodPressure'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '80.5'}
{'type': 'LITERAL', 'value': 'SkinThickness'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '25'}
{'type': 'LITERAL', 'value': 'Insulin'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '100'}
{'type': 'LITERAL', 'value': 'BMI'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '28.5'}
{'type': 'LITERAL', 'value': 'DiabetesPedigreeFunction'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '0.55'}
{'type': 'LITERAL', 'value': 'Age'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '40'}
{'type': 'LITERAL', 'value': 'Outcome'}
{'type': 'OPERATOR', 'value': ':'}
{'type': 'NUMERICAL', 'value': '1'}
{'type': 'COMMENT', 'value': '# damn this is good'}
```

Conclusions

The development of this lexer for a Domain-Specific Language (DSL) tasked with analyzing medical results represents a significant milestone in the process of creating a comprehensive system for interpreting and processing medical data. This lexer is a crucial component in the broader context of compiler design, It constitutes the base upon which the parser and AST will be later built. When using the lexer, attention must be drawn towards the order of each type of token in

TOKENS list.