Topic: Regular expressions

Course: Formal Languages & Finite Automata

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

Objectives:

1. Write and cover what regular expressions are, what they are used for;
2. Below you will find 3 complex regular expressions per each variant. Take a variant depending on your number in the list of students and do the following:

   a. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown).

   b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

   c. **Bonus point**: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

Theory:

The journey of developing a regex word creator has been an enlightening experience, offering deep insights into the complexities and nuances of regular expressions (regex). Regular expressions are a powerful tool used across various programming languages, including R, Python, C, C++, Perl, Java, and JavaScript, for processing and mining unstructured text data. They are particularly useful in applications such as search engines, lexical analysis, spam filtering, and text editors

Implementation description

Variant 1:

Variant 4:

$$(S|T)(U|V)w^*y^+24$$

$$L(M|N)O^3p^*Q(2|3)$$

$$R^*S(T|U|V)w(x|y|z)^2$$

Due to the big similarity between this and the first laboratory I decided to implement a translator between the given regex and the grammar of the first laboratory, everything else is highly similar.

```
 import src.lab1.grammarHelper as grammarHelper

import src.lab1.language as language
from collections import defaultdict
variant="""
(a|b)(c|d)E+G?
P(Q|R|S)T(UV|W|X)*Z+
1(0|1)*2(3|4)^(5)36
"""
grammars=[]
REGEX = [x for x in variant.split("\n") if x]
terminals="Sabcdefghijklmnoprtqwxyz"
nonterminals=terminals.upper()
print(REGEX)
mapping={}
```

```
mapping={}
for index,expr in enumerate(REGEX):
    print(expr)
    mapping[index]=defaultdict(dict)
    mapping[index]={x:x.lower() for x in expr if x.isalpha()}
    print(mapping)
    expr=expr.lower()
    final="\n"

    starting=0

    expr=expr.replace("("," ").replace(")"," ").replace("*"," * ").replace("^"," ^ ").replace("+"," + ").replace("?"," ? ")
    expr=[x for x in expr.split(" ") if x]
    VT=[]
    for i,node in enumerate(expr):
        node=str(node)
        if "|" in node:
            choices=node.split("|")
            for choice in choices:
                final+=f'{nonterminals[starting]} -> {choice}{nonterminals[starting+1]}\n'
                if choice not in VT:
                    VT.append(choice)
        elif node == "+":
            final+=f'{nonterminals[starting]} -> {nonterminals[starting-1]}\n'
            starting-=1

        elif node == "*":
            final+=f'{nonterminals[starting-1]} -> {nonterminals[starting]}\n'
            final+=f'{nonterminals[starting]} -> {nonterminals[starting-1]}\n'
            starting-=1

        elif node == "?":
            final+=f'{nonterminals[starting-1]} -> {nonterminals[starting]}\n'
            starting-=1
        elif node == "^":
            times = int(expr[i+1])

            for j in range(times-1):

                previous=[x for x in final.split("\n") if x]
                previous = [x for x in previous if (x[0]==nonterminals[starting-1+j])]
                s=""

                for x in previous:

                    s+=nonterminals[starting+j]
                    s+=x[1:-1]
                    s+=nonterminals[starting+j+1]
                    s+="\n"

                final+=s
            starting+= times-2


        elif node.isalnum():
            if expr[i-1] == "^":
                continue
            final+=f'{nonterminals[starting]} -> {node}{nonterminals[starting+1]}\n'
            VT.append(node)
        starting+=1


    final+=f'{nonterminals[starting]} ->  \n'


    lb="{"
```

```
    rb="}"
    grammar=f"Varianta 20:\n VN= {lb}{', '.join(list(nonterminals[:starting+1]))}{rb},\n VT= {lb}{', '.join(VT)}{rb},\n P={lb}{final
    print(grammar)
    grammars.append(grammar)




for index,grammar in enumerate(grammars):
    rules, VN, VT, F = grammarHelper.grammar_to_language(grammar)




    lang=language.Language(rules,VN,VT,F)

    lang.generate_word(5,'S')
    for word in lang.words:
        for key,value in mapping[index].items():
            if value!=key:
                word = word.replace(value,key)
        print("Generated words: ",word)
```

This is the entire code, now let us proceed to analyse it in more detail by parts.

```
 variant="""
(a|b)(c|d)E+G?
P(Q|R|S)T(UV|W|X)*Z+
1(0|1)*2(3|4)^(5)36
"""
```

This the code representation of the given regex variant

```
 REGEX = [x for x in variant.split("\n") if x]
for index,expr in enumerate(REGEX):
    mapping[index]=defaultdict(dict)
    mapping[index]={x:x.lower() for x in expr if x.isalpha()}
    expr=expr.lower()
    final="\n"

    starting=0

    expr=expr.replace("("," ").replace(")"," ").replace("*"," * ").replace("^"," ^ ").replace("+"," + ").replace("?"," ? ")
    expr=[x for x in expr.split(" ") if x]
```

in the code above i split the variant by lines, having a separation in each regex item. then i identify 'operations' - creation of nodes '(&)' |*+? etc, I created some space around them and split each line by space. this effectively, separates the given regex item into nodes ex:

Given reges:

```
 (a|b)(c|d)E+G?
```

becomes:

```
 ['a|b', 'c|d', 'e', '+', 'g', '?']
```

THis splits the regex into nodes, both by keeping paranthesis for complex nodes, and keeping in mind operations, what follow is parsing this structure for each specific type of operation and implement its behavior in a grammar:

```
for i,node in enumerate(expr):
        node=str(node)
        if "|" in node:
            choices=node.split("|")
            for choice in choices:
                final+=f'{nonterminals[starting]} → {choice}{nonterminals[starting+1]}\n'
                if choice not in VT:
                    VT.append(choice)
        elif node == "+":
            final+=f'{nonterminals[starting]} → {nonterminals[starting-1]}\n'
            starting-=1

        elif node == "*":
            final+=f'{nonterminals[starting-1]} → {nonterminals[starting]}\n'
            final+=f'{nonterminals[starting]} → {nonterminals[starting-1]}\n'
            starting-=1

        elif node == "?":
            final+=f'{nonterminals[starting-1]} → {nonterminals[starting]}\n'
            starting-=1
        elif node == "^":
            times = int(expr[i+1])

            for j in range(times-1):

                previous=[x for x in final.split("\n") if x]
                previous = [x for x in previous if (x[0]==nonterminals[starting-1+j])]
                s=""

                for x in previous:

                    s+=nonterminals[starting+j]
                    s+=x[1:-1]
                    s+=nonterminals[starting+j+1]
                    s+="\n"

                final+=s
            starting+= times-2


        elif node.isalnum():
            if expr[i-1] == "^":
                continue
            final+=f'{nonterminals[starting]} → {node}{nonterminals[starting+1]}\n'
            VT.append(node)
        starting+=1


    final+=f'{nonterminals[starting]} →  \n'
```

Here I handled all of the operators, transforming them into grammar in a variable called final

```
 lb="{"
rb="}"
grammar=f"Varianta 20:\n VN= {lb}{', '.join(list(nonterminals[:starting+1]))}{rb},\n VT= {lb}{', '.join(VT)}{rb},\n P={lb}{final}{rb
```

Here I transform the resulted transitions in the format required by the grammar in lab 1 this is the result:

```
 Varianta 20:
  VN= {S, A, B, C, D},
  VT= {a, b, c, d, e, g},
  P={
 S → aA
 S → bA
 A → cB
 A → dB
 B → eC
 C → B
 C → gD
 C → D
 D →
 }
```

Yes, there are epsilon productions, mostly for specifing the end of the regex.

```



for index,grammar in enumerate(grammars):
    rules, VN, VT, F = grammarHelper.grammar_to_language(grammar)




    lang=language.Language(rules,VN,VT,F)

    lang.generate_word(5,'S')
    for word in lang.words:
        for key,value in mapping[index].items():
            if value!=key:
                word = word.replace(value,key)
        print("Generated words: ",word)
```

Then i iterate through all of the resulted grammars and call the word generator based on the lab 1 code. I also map the uppercase letters present in the regex back into uppercase at the end, since they conflict with Nonterminals in the grammar, they were made lowercase in the beginning.

This is how the output of the program looks like:

```
 Generated words:  bdEEEEG
Generated words:  bcEEG
Generated words:  adEG
Generated words:  acEG
Generated words:  bdEG
```

```
 Generated words:  PSTWWXXZ
Generated words:  PQTXWZ
Generated words:  PRTZ
Generated words:  PQTWZ
Generated words:  PRTXZ
Generated words:  PRTUVZZZ
```

```
 Generated words:  123344336
Generated words:  1023444436
Generated words:  1023344336
Generated words:  1024344336
Generated words:  1023333336
Generated words:  1123433436
```

Clearly, for each variant the respective words were generated, here is the steps taken to do so:

```

S -> bA
 -> bdB
 -> bdeC
```

```
 -> bdegD
 -> bdeg

S -> aA
 -> adB
 -> adeC
 -> adegD
 -> adeg

S -> bA
 -> bdB
 -> bdeC
 -> bdeD
 -> bde

S -> aA
 -> adB
 -> adeC
 -> adeB
 -> adeeC
 -> adeeB
 -> adeeeC
 -> adeeegD
 -> adeeeg

S -> aA
 -> adB
 -> adeC
 -> adeB
 -> adeeC
 -> adeegD
 -> adeeg

S -> bA
 -> bcB
 -> bceC
 -> bceB
 -> bceeC
 -> bceegD
 -> bceeg
Generated words:  bdEG
Generated words:  adEG
Generated words:  bdE
Generated words:  adEEEG
Generated words:  adEEG
Generated words:  bcEEG

S -> pA
 -> prB
 -> prtC
 -> prtxD
 -> prtxzE
 -> prtxzD
 -> prtxzzE
 -> prtxzz

S -> pA
 -> psB
 -> pstC
 -> pstwD
 -> pstwzE
 -> pstwz

S -> pA
 -> prB
```

```
 -> prtC
 -> prtwD
 -> prtwzE
 -> prtwzD
 -> prtwzC
 -> prtwzwD
 -> prtwzwzE
 -> prtwzwz

S -> pA
 -> psB
 -> pstC
 -> pstwD
 -> pstwC
 -> pstwD
 -> pstwC
 -> pstwxD
 -> pstwxC
 -> pstwxwD
 -> pstwxwC
 -> pstwxwD
 -> pstwxwzE
 -> pstwxwzD
 -> pstwxwzC
 -> pstwxwzuvD
 -> pstwxwzuvC
 -> pstwxwzuvuvD
 -> pstwxwzuvuvC
 -> pstwxwzuvuvuvD
 -> pstwxwzuvuvuvC
 -> pstwxwzuvuvuvxD
 -> pstwxwzuvuvuvxC
 -> pstwxwzuvuvuvxwD
 -> pstwxwzuvuvuvxwzE
 -> pstwxwzuvuvuvxwzD
 -> pstwxwzuvuvuvxwzC
 -> pstwxwzuvuvuvxwzuvD
 -> pstwxwzuvuvuvxwzuvzE
 -> pstwxwzuvuvuvxwzuvz

S -> pA
 -> psB
 -> pstC
 -> pstuvD
 -> pstuvC
 -> pstuvuvD
 -> pstuvuvC
 -> pstuvuvxD
 -> pstuvuvxzE
 -> pstuvuvxz

S -> pA
 -> pqB
 -> pqtC
 -> pqtuvD
 -> pqtuvzE
 -> pqtuvzD
 -> pqtuvzC
 -> pqtuvzwD
 -> pqtuvzwC
 -> pqtuvzwuvD
 -> pqtuvzwuvzE
 -> pqtuvzwuvz
Generated words:  PRTXZZ
Generated words:  PSTWZ
Generated words:  PRTWZWZ
```

```
Generated words:  PSTWXWZUVUVUVXWZUVZ
Generated words:  PSTUVUVXZ
Generated words:  PQTUVZWUVZ

S -> 1A
 -> 11B
 -> 112C
 -> 1123D
 -> 11233E
 -> 112334F
 -> 1123344G
 -> 11233443H
 -> 1123344336I
 -> 1123344336

S -> 1A
 -> 1B
 -> 1A
 -> 1B
 -> 12C
 -> 124D
 -> 1243E
 -> 12433F
 -> 124333G
 -> 1243333H
 -> 124333336I
 -> 124333336

S -> 1A
 -> 1B
 -> 12C
 -> 124D
 -> 1244E
 -> 12444F
 -> 124444G
 -> 1244444H
 -> 124444436I
 -> 124444436

S -> 1A
 -> 11B
 -> 112C
 -> 1123D
 -> 11234E
 -> 112343F
 -> 1123434G
 -> 11234343H
 -> 1123434336I
 -> 1123434336

S -> 1A
 -> 1B
 -> 1A
 -> 11B
 -> 11A
 -> 111B
 -> 1112C
 -> 11124D
 -> 111243E
 -> 1112434F
 -> 11124343G
 -> 111243433H
 -> 11124343336I
 -> 11124343336
```

```
S -> 1A
 -> 1B
 -> 12C
 -> 124D
 -> 1243E
 -> 12434F
 -> 124343G
 -> 1243434H
 -> 124343436I
 -> 124343436
Generated words:  1123344336
Generated words:  124333336
Generated words:  124444436
Generated words:  1123434336
Generated words:  11124343336
Generated words:  124343436
```

Conclusions

The development of this regex word creator, has helped in understanding the intricacies behind simple regex instructions, and the way differnt functions work. The process of creating a regex word creator has highlighted the importance of starting with simple patterns and gradually adding complexity. This approach, combined with testing incrementally, ensures that the regex patterns are accurate and efficient.