A Software Parser for Regular Expressions Using Python

Carl Brady

November 2019

TU Dublin Blanchardstown

Contents

Abstract

This document describes in detail, a parser for regular expressions that is created using python, a programming language. The parser analyses a combination of sentences and returns an output of the bracketed phrasal structure of the sentence. This document will cover the steps taken in creating such a program and an explanation of how each method works and the necessary files needed. To create this system, the author implemented an agile methodology to ensure that there was sufficient time to research the project and test each use case. It also meant that the system was continuously improved and could be reviewed on a regular basis.

1 Introduction

Computational Linguistics is a subsidiary of Computer Science and Linguistics. It is used in order to help computers understand human languages. Although, speaking in ones native tongue may be an action a person can do without speaking, for a computer this process can be extremely difficult [3]. We as humans, do not need to actively think when creating regular sentences or simple conversation, this act comes naturally to us. In order for a computer to act in the same way, they too must have a method of accessing this ability to understand the complex structure of sentences that we usually do not consider such as index of all words and their meaning, syntax and also the morphology of the language. Spoken language is even harder for a computer to understand, factors such as pitch, pronunciation of words or accents also must be taken into consideration when creating such a system.

Work in this field can be seen as early as the 1950's with concepts such as machine translation but it was not until the late 1980's / 1990's that real improvements were noticed [5]. The corpus based approach, which has been very successful, so successful it has even had an impact the way language is thought [2], has helped in pushing a new life into the field. In modern times the field has seen expansion in terms of voice recognition and machine learning [6]. Siri[1], Apples virtual assistant is an example of this technology used today. This report describes a program based on a simplified corpus based approach using lexicon, a rule-set and bottom-up-parsing.

2 Lexicon & Rules

This section will detail the lexicon used for the making of the software parser. A lexicon is a group of lexical categories or words that essentially describe a language [4]. A lexicon can be used to detail information about languages such as; words, morphological structures, syntax and how words are pronounced. The lexicon the author decide to create was based on words, syntax and morphological structures. The rules that were created for this program are based upon the book 'Natural Language Processing with Python' by Bird et al[4].

2.1 Lexicon

The lexicon for this program is created with a group of listings in a .txt file. Every word available from the set of example sentences is combined with a part of speech (POS) tag. The POS tag is based on the NLTK abbreviations for their POS Tagger [4]. For example, each line in the text file had a word and its associated POS tag (eg, DT the). In A.4 the full lexicon.txt file is available.

2.2 Rules

The rules used in order to create this program were made in a similar fashion, each line in the rule.txt file was made up of a left hand side argument and a right hand side argument. The left hand side of the argument is the parent to the right hand side argument. For example, each line in the text had the parent specified followed by its children (NP → DT NN). In A.5 the full rules.txt file is available

3 System Architecture

This section covers the system architecture, all of the methods used in the program and the overall code design are detailed in this section. The software program is made up of six components; 'main.py', 'parser.py', 'file_manager.py', 'lex.txt', 'rules.txt' and 'tests.txt'. The three text files are used to each store one of the following; lexicon, rules, test cases. Three python files were created, 'main.py' which acts the main file and calls upon methods from other files. 'file_manager.py' handles the opening and reading of all .txt files. 'parser.py' contains the methods for POS tagging, grammar checking, bracketed formatting and creating a tree. A class diagram used in the creation of this system can be seen in A.7.

3.1 File Manager

A file manager was used in order to read .txt files into the software program. Each .txt was stored as an array of sentences so that each element in the array could be accesed at a later stage.

```
    lex_file = open("lex.txt","r") #Open and read file that contains lexicon
    lex_arr = lex_file.readlines()
    lex_file.close()
    return lex_arr
```

As seen in the above code segment, the lexicon file is read using the open() method, which takes two variables , the name of the file and the permission needed (r = read, w = write, etc). Once the file is read, the method readLines() then stores each line of the file as an individual item in an array, which in this case we assign to the variable lex_arr. The file is then closed as it is no longer needed and the array is returned.

3.2 POS Tagging

The POS tagger was implemented by creating a method pos_tagger(x,tests,lex_arr), this method takes three arguments. 'X' is the 'test select' input from the user (further explained in section 3.6), 'tests' is the array of test cases and 'lex_arr' is the lexicon returned as an array as seen in 3.1. The selected test case was then split using the .split() method and stored in an array 'test'.

```
    for word in test:
        for lexi in lex_arr:
            lex = lex_arr[j].split()
            if test[i] == lex[1]:
                sentence += lex_arr[j]
            j+= 1
        j = 0
        i += 1
```

As can be seen above, the POS tagger checks each word in the test case sentence against every lexical entry's right hand side argument in an attempt to find a match. If a match is found, the matching lexicon entry is then appended to the string 'sentence', this continues until all words in a test sentence have been checked, the method then returns 'sentence' on completion.

3.3 Parser

A parser was implemented based on the bottom up parsing method. The method grammar_check(pos, rules) takes two arguments, the first is the sentence that was created by the POS tagger and the second is the rules that have been stored in an array from the getRules() method in file_manager.py.

```
    pos_split = pos.split()
    pos_NP = pos_split[0] + ' ' + pos_split[2] # gets the POS tag for the first two words and adds to string
    pos_VP = pos_split[4] # gets the POS tag for the third word and adds to string
    pos_NP2 = pos_split[6] + ' '+pos_split[8] + ' ' + pos_split[10] # gets the POS tag for the last three words and adds to string
```

Firstly, the method splits the POS tagged sentence. Three strings are then created based on the split and as the POS tag only appears every second word in the array only the even indexes are required. As it is known there are three phrases (NP, VP, NP) that essentially make up the correct sentence. The method first splits them accordingly, the first two tags are created as a string, etc.

```
    if pos_split[1] in 'a' and pos_split[3] == 'men': # checks for 'a men' in a sentence
        print('\'A men\' is grammatically incorrect.')
        return False
    elif pos_split[1] in 'a' and pos_split[3] == 'women': # checks for 'a women' in a sentence
        print('\'A women\' is grammatically incorrect')
        return False
```

The parser / grammar checker also checks if there is any immediate grammar mistakes that can rule this sentence out from being a coherent sentence and if so, the method returns false.

```
    while i<3: # Checking the if the childern of NP that has a parent of S are correct
        rule = rules[i].split('->')
        x = rule[1].split() # rule[1] is only split here again to fix the formatting and remove extra spacing
        y = x[0]+' '+x[1] # here we assign formatted x to y
        if pos_NP == y:
            checker += rule[0] # rule[0] is the left hand side of the matching rule
        i += 1
    i=4
```

Above we can see how a loop is created in order to ensure the children of NP are correct. As we know, all of the rules that cover the first Noun Phrase(NP) grouping in the sentence are covered in the first three lines of the rule set, it does not make sense to examine each rule in the array, therefore the bound is set as $i < 3$. The program then checks if the right hand side of the rule matches the tags from the test sentence and if they match it appends the right hand side argument to a string. This is done for all three possible parent tags (NP,VP,NP), until it is possible whether to determine if the sentence is correct or not, if the sentence is correct the method returns true.

3.4 Bracketed Phrasal Formatting

The bracketed phrasal method is only called if the grammar_check returns true. Once this happens, the bracketed phrasal takes the POS tagged sentences and uses .split. Once the sentence has been split, it is recreated with the added syntax required to create a bracketed phrasal structure.

```
     while i<3:
        br_sentence += '[' + split[i] + ' ' + split[i+1] + ']'
        i+=2
    br_sentence += ']][VP[VP'
```

As seen in the above segment, the sentence is rejoined using a loop for each phrase contained within a sentence.

3.5 Tree Structure

In order to create a tree structure like display on the command line, the program imports treelib, a package that assists creating trees with python.

```
    split = sentence.split()
    tree = Tree()
    tree.create_node("S1", "s1")
    tree.create_node("S", "s", parent="s1")
```

Once again we can see the sentence is split, from here we can add nodes to the tree with .createnode() this method takes two arguments, how the node will be displayed in the tree and its ID. As the first node in the tree does not have a parent there is none listed, for every other node in the tree a parent is listed.

3.6 Main File

The main file simply calls on all of the methods mentioned above.

```
while True:
        x = input('\n\nInput a test number, type \'help\' to see all test cases
    or \'exit\' to exit the program\n')
        if x in 'help':
            print('-----------------------------------------------------------')
            print_tests()

        elif x in 'exit':
            return
```

Once we run this program we can see the program prompts a user to input a test number or type 'help' to see the test cases or 'exit' to exit the program.

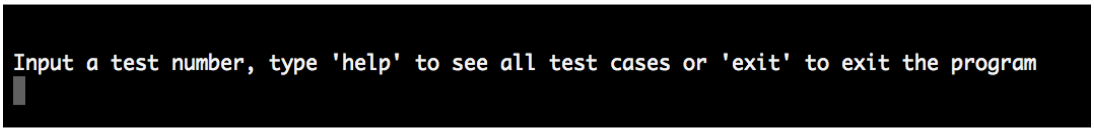

Figure 1: Initial prompt

If the user inputs help, the 32 possible test cases are shown to the user and the user is once again shown the initial prompt.

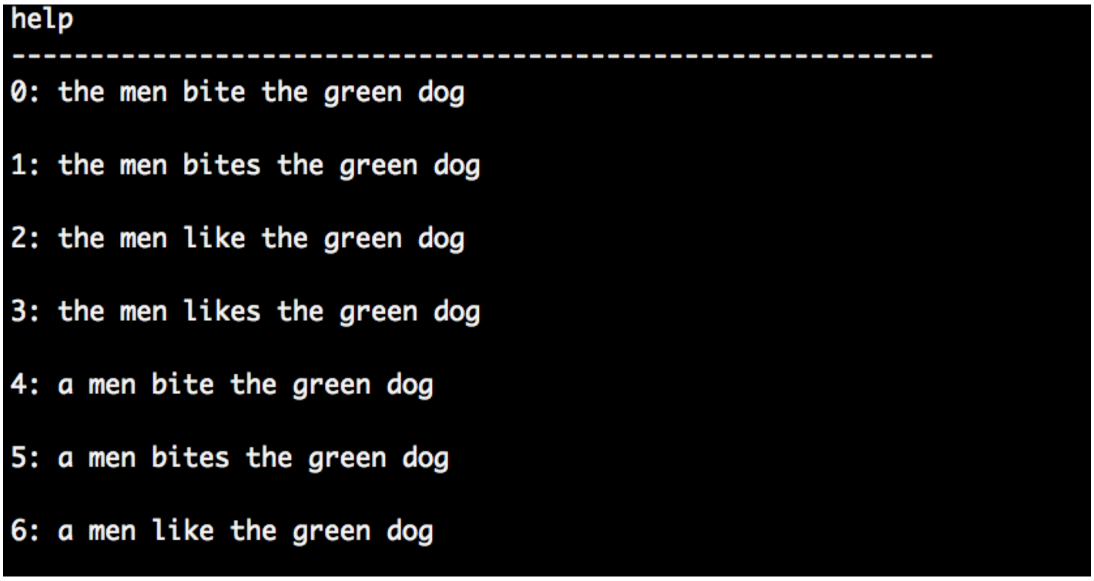

Figure 2: Help on screen information

An example of how a correct and incorrect test sentence is displayed can be seen in A.8:

4 Conclusion

The results of this software parser for regular expressions showed a 100% accuracy rate when tested against all thirty two test cases. The program and report give insight into how much detail is required to simply determine the validity of a sentence regardless of its context. The natural language toolkit for python is a great mechanism for understanding how to create corpus based systems for language processing. To conclude, the author feels that given more time, a more efficient algorithm could have been designed that takes advantage of the computational efficiency of a hash map and also its key value pairings which could be used to store left hand and right hand side arguments.

References

[1] Siri apple voice assistant.

[2] Samina Dazdarevic, Amela Lukac-Zoranic, and ahreta Fijuljanin. Corpus-based research for language teaching. *Corpus Linguistics in Language Teaching.*

[3] Michael J. Garbade. A simple introduction to natural language processing, Oct 2018.

[4] Edward Loper and Steven Bird. *NLTK: the Natural Language Toolkit.* S.n., 2009.

[5] Lenhart Schubert. Computational linguistics, Feb 2014.

[6] Jiawei Yao. Automated sentiment analysis of text data with nltk. *Journal of Physics: Conference Series*, 1187(5):052020, 2019.

A Appendix

A.1 main.py

```python
from parser import pos_tagger, grammar_check, bracketed_sentence, incorrect_sentence_parser, build_tree
from file_manager import get_tests, get_lex, get_rules, print_tests


def main():
    rules = get_rules()
    lex_arr = get_lex()
    tests = get_tests()

    while True:
        x = input('\n\nInput a test number, type \'help\' to see all test cases or \'exit\' to exit the program\n')
        if x in 'help':
            print('------------------------------------------------------------')
            print_tests()

        elif x in 'exit':
            return

        elif x.isdigit():
             print('\n\nRESULTS\n------------------------------------------------------------')
             x = int(x)
             pos = pos_tagger(x,tests,lex_arr)

             if grammar_check(pos, rules) == True:
                 print('Verdict: Sentence is grammatically correct\n')
                 bracketed_sentence(pos)
                 build_tree(pos)
             else:
                 test = tests[x].split()
                 print('Verdict: Sentence is not grammatically correct\n')
                 incorrect_sentence_parser(test)
        else:
            print('\n------------------------------------------------------------')
            print('\nPLEASE INPUT A CORRECT COMMAND!\n')
        print('------------------------------------------------------------')







if __name__== "__main__":
  main()
```

A.2 file_manager.py

```python

def get_tests():
    t=''
    tests_file = open("tests.txt","r") #Open and read file that contains tests sentnces
    tests = tests_file.readlines() #store each line as an individual test sentence
    tests_file.close()
    i = 0
    return tests

```

```
def get_lex():

    lex_file = open("lex.txt","r") #Open and read file that contains lexicon
    lex_arr = lex_file.readlines()
    lex_file.close()
    return lex_arr

def get_rules():

    rule_file = open("rules.txt","r") #Open and read file that contains rules
    rule_arr = rule_file.readlines()
    rule_file.close()
    return rule_arr

def get_grammar():
    file = open("grammar.txt","r") #Open and read file that contains rules
    file_str = file.read()
    grammar = CFG.fromstring(file_str) #CFG = Read File
    file.close()
    return grammar


def print_tests():
    t=''
    tests_file = open("tests.txt","r") #Open and read file that contains tests
        sentnces
    tests = tests_file.readlines() #store each line as an individual test
        sentence
    tests_file.close()
    i = 0
    for t in tests:
        print(str(i) +': ' + t)
        i += 1
```

A.3 parser.py

```
from treelib import Node, Tree

#pos_tagger(x,tests,lex_arr) x = test number, tests = list of test sentences,
    lex_arr = list of lexicon rules
#This method compares each word of the test sentence against the lexicon file
    and allocates the related tag
def pos_tagger(x,tests,lex_arr):
    sentence= ''
    j = 0
    i = 0
    test = tests[x].split()
    print('\nTest Sentence: ' + tests[x])
    for word in test:
        for lexi in lex_arr:
            lex = lex_arr[j].split()
            if test[i] == lex[1]:
                sentence += lex_arr[j]
            j+= 1
        j = 0
        i += 1
    return sentence


# grammar_check(pos,rules) is a boolean method that takes in the tagged pos
    sentence
# and a list of rules and compares the two and returns true if the sentence is
    correct
def grammar_check(pos, rules):
    checker = ''
    i = 1
    pos_split = pos.split()
```

```python
    pos_NP = pos_split[0] + ' ' + pos_split[2] # gets the POS tag for the first two words and adds to string
    pos_VP = pos_split[4] # gets the POS tag for the third word and adds to string
    pos_NP2 = pos_split[6] + ' '+pos_split[8] + ' ' + pos_split[10] # gets the POS tag for the last three words and adds to string


    if pos_split[1] in 'a' and pos_split[3] == 'men': # checks for 'a men' in a sentence
        print('\'A men\' is grammatically incorrect.')
        return False
    elif pos_split[1] in 'a' and pos_split[3] == 'women': # checks for 'a women' in a sentence
        print('\'A women\' is grammatically incorrect')
        return False


    while i<3: # Checking the if the childern of NP that has a parent of S are correct
        rule = rules[i].split('->')
        x = rule[1].split() # rule[1] is only split here again to fix the formatting and remove extra spacing
        y = x[0]+' '+x[1] # here we assign formatted x to y
        if pos_NP == y:
            checker += rule[0] # rule[0] is the left hand side of the matching rule
        i += 1
    i=4
    while i<6: # Checking if the childern of VP the has a parent of VP are correct
        rule = rules[i].split('->')
        x = rule[1].split()
        y = x[0]
        if pos_VP == y:
            checker += rule[0]
        i += 1

    rule = rules[6].split('-> ') #checks if the childern of NP that has a parent of VP are correct
    x = rule[1].split()
    y = x[0]+' '+x[1]+' '+x[2]
    if pos_NP2 == y:
        checker += rule[0]
    i += 1

    str_split = checker.split()
    if str_split[2] == 'NP': # all POS tags have been analysed and if the sentence is correct 'checker' should be NP VPzNP or NPp VPp NP
        if str_split[0] == 'NP' and str_split[1] == 'VPz':
            return True
        elif str_split[0] == 'NPp' and str_split[1] == 'VPp':
            return True
        else:
            return False
    else:
        return False


# bracketed_sentence(sentence) takes in a test sentence that has been proven to be correct and creates a bracketed phrasal structure
def bracketed_sentence(sentence):
    i = 0
    br_sentence = '[S1[S[NP['
    split = sentence.split()
    while i<3:
        br_sentence += '[' + split[i] + ' ' + split[i+1] + ']'
        i+=2
    br_sentence += ']][VP[VP'
```

```
    while i<5:
        br_sentence += '[' + split[i] + ' ' + split[i+1] + ']'
        i+=2
    br_sentence += '][NP['
    while i<11:
        br_sentence += '[' + split[i] + ' ' + split[i+1] + ']'
        i+=2
    br_sentence += ']]]'
    print(br_sentence)

# build_tree also takes a correct sentnce as an arg and outputs a tree to the cmd line
def build_tree(sentence):
    print('\n\nParser Tree Plot\n')
    split = sentence.split()
    tree = Tree()
    tree.create_node("S1", "s1")
    tree.create_node("S", "s", parent="s1")
    tree.create_node("NP", "np", parent="s")
    tree.create_node("VP", "vp", parent="s")
    tree.create_node("VP", "vp2", parent = "vp")
    tree.create_node("_NP", "np2", parent = "vp")
    tree.create_node(split[0],"dt" , parent="np")
    tree.create_node(split[2], "nn", parent="np")
    tree.create_node(split[4], "vb", parent="vp2")
    tree.create_node(split[6], "dt2", parent="np2")
    tree.create_node(split[8], "jj", parent="np2")
    tree.create_node(split[10], "nn1", parent="np2")
    tree.create_node(split[1],"1" , parent="dt")
    tree.create_node(split[3], "2", parent="nn")
    tree.create_node(split[5], "3", parent="vb")
    tree.create_node(split[7], "4", parent="dt2")
    tree.create_node(split[9], "5", parent="jj")
    tree.create_node(split[11], "6", parent="nn1")
    tree.show()



# incorrect_sentence_parser(sentence) takes an incorrect sentence and creates a bracketed phrasal structure for as long as the sentence is valid
def incorrect_sentence_parser(sentence):
    br_str = '[S1[S[NP['
    if repr(sentence[0]) == "'the'":
        br_str += "DT the]"

    elif repr(sentence[0]) == "'a'":
        br_str += "DT a]"

    else:
        return

    if repr(sentence[1]) in "'men'":
        if repr(sentence[0]) in ("'a'"):
            return
        else:
            x = repr(sentence[1])
            br_str += "[NNP "+ x[1:-1] +"]"

    elif repr(sentence[1]) in "'man'":
        if repr(sentence[0]) in ("'a'"):
            return
        else:
                x = repr(sentence[1])
                br_str += "[NNP "+ x[1:-1]+"]"

    else:
        return

    if repr(sentence[2]) == "'bites'":
```

```python
            if repr(sentence[1]) not in ("'man'" , "'woman'"):
                print('Sentence incorrect but correct as far as ' + repr(sentence[1]))
                br_str += "]]"
                print('\n' + br_str)
                return

        elif repr(sentence[2]) == "'bite'":
            if repr(sentence[1]) not in ("'men'" , "'women'"):
                print('Sentence incorrect but correct as far as ' + repr(sentence[1]))
                br_str += "]]]"
                print('\n' + br_str)
                return

        elif repr(sentence[2]) == "'like'":
            if repr(sentence[1]) not in ("'men'" , "'women'"):
                print('Sentence incorrect but correct as far as ' + repr(sentence[1]))
                br_str += "]]]"
                print('\n' + br_str)
                return

        elif repr(sentence[2]) == "'likes'":
            if repr(sentence[1]) not in ("'man'" , "'woman'"):
                print('Sentence incorrect but correct as far as ' + repr(sentence[1]))
                br_str += "]]]"
                print('\n' + br_str)
                return


        br_str += "]]"
        return
```

A.4 Lexicon

DT the
DT a
NN man
NN woman
NNS men
NNS women
VBZ bites
VBP bite
VBZ likes
VBP like
JJ green
NN dog

A.5 Rules

S -> NP VP
NP -> DT NN
NPp -> DT NNS
VP -> NP VP
VPp -> VBP
VPz -> VBZ

NP -> DT JJ NN

A.6 Test Cases

the men bite the green dog
the men bites the green dog
the men like the green dog
the men likes the green dog
a men bite the green dog
a men bites the green dog
a men like the green dog
a men likes the green dog
the man like the green dog
the man likes the green dog
the man bite the green dog
the man bites the green dog
a man like the green dog
a man likes the green dog
a man bite the green dog
a man bites the green dog
the women bite the green dog
the women bites the green dog
the women like the green dog
the women likes the green dog
a women bite the green dog
a women bites the green dog
a women like the green dog
a women likes the green dog
the woman bite the green dog
the woman bites the green dog
the woman like the green dog
the woman likes the green dog
a women bite the green dog
a woman bites the green dog
a woman like the green dog
a woman likes the green dog

A.7 Class Diagram

The class diagram used for designing the software is shown below

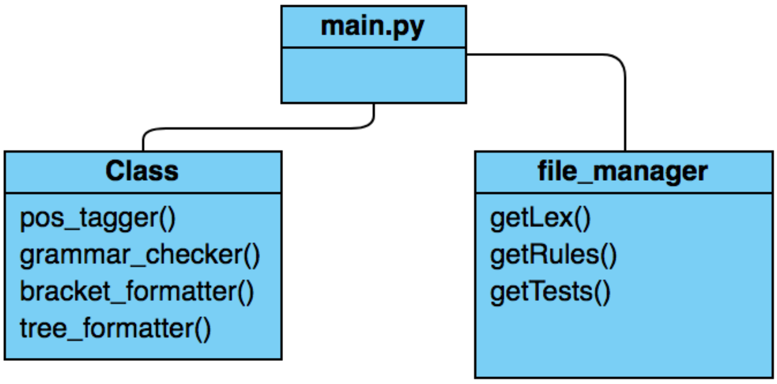

Figure 3: Class diagram

A.8 Outputs

```
RESULTS
-----------------------------------------------------------

Test Sentence: the men bite the green dog

Verdict: Sentence is grammatically correct

[S1[S[NP[[DT the][NNS men]]][VP[VP[VBP bite]][NP[[DT the][JJ green][NN dog]]]]


Parser Tree Plot

S1
└── S
    ├── NP
    |   ├── DT
    |   |   └── the
    |   └── NNS
    |       └── men
    └── VP
        ├── VP
        |   └── VBP
        |       └── bite
        └── _NP
            ├── DT
            |   └── the
            ├── JJ
            |   └── green
            └── NN
                └── dog

-----------------------------------------------------------
```

Figure 4: Output displayed for correct test case

```
RESULTS
-----------------------------------------------------------

Test Sentence: the men bites the green dog

Verdict: Sentence is not grammatically correct

Sentence incorrect but correct as far as 'men'

[S1[S[NP[DT the][NNP men]]]
-----------------------------------------------------------
```

Figure 5: Output displayed for incorrect test case