Python Implementation of The Prefix Grammar to Right-Linear Grammar Transformation Algorithm.

Keshav Raminedi

August 10, 2020

Abstract

In this project I implement a algorithm proposed in section 2.2 of [1] to convert the prefix grammar to right linear grammar, the algorithm was implemented in polynomial time as estimated in [1]. The algorithm was implemented in python3 language.
Keywords: prefix grammar, right-linear grammar, regular-grammar.
Note:The code and the instructions to use the program are in the Appendix, I suggest to try and test the code before reading the report.

1 Introduction

Regular expression are another type of language-defining notation, they can also be defined as a programming language in which we express some important applications such as text search applications or compiler component.These expressions are only capable of defining regular languages.For every regular language there exists a regular grammar that defines it.
Prefix grammars are an alternative characterization of Regular grammars as mentioned in [1]. Prefix grammars are very intriguing because they defer from other characterizations of regular grammar as a result a polynomial sized bi-directional relationship has not been formed yet in [1].

2 Definition and Problem Statement

A prefix grammar G in is defined by specifying a finite subset of strings L_G and productions that rewrite the prefixes of strings in L_G. In section 2.2 of [1] an algorithm was proposed where a prefix grammar was converted into a Right-Linear grammar.The goal is to successfully implement this algorithm in python .

3 Algorithm

An algorithm was proposed in section 2.2 of [1],where the input is a prefix grammar G_P has productions $a_1 -> b_1 \dots a_m -> b_m$ and base strings $l_1 \dots l_n$, the output is a Right linear grammar G_R for language of G_P.

1. Initially add the productions:
   $S -> l_1 | \dots | l_n | b_1 V_{a_1} | \dots | b_m V_{a_m}$
2. Add productions of the form
   $V_{a_i} -> w V_{a_j}$
   Where
   $S -> \phi_1 V_{a_{i_1}}$ $V_{a_{i_1}} -> \phi_2 V_{a_{i_2}} \dots, V_{a_{i_{k-1}}} -> \phi_k w V_{a_{i_k}}$
   are production already in $\phi_1, \dots, \phi_K$ and w are strings of terminals of G_R, $\phi_1 \phi_2 \dots \phi_K = a_1$, V_{a_i} and $V_{a_{i_1}}, \dots, V_{a_{i_k}}$ are non-terminals of G_R and V_{a_j} is a non-terminal or is a ϵ
3. Go back to step 2 until no such productions can be added.

4 Idea behind the program

The choice of data structures for productions in prefix or right linear grammar is a dictionary where a key and value system is present, the keys are strings, note that the prefix grammar have no non-terminals so a list of strings as values of dictionary will do but for a right linear grammar we need a list of tuples as there are terminals and non-terminals. for example, {'S': [('', 'aa'), ('a', 'ab'), ('aa', 'c')], 'a': [('a', 'b'), ('', 'a')], 'aa': [('', '')]} is a right linear grammar and {'a': ['ab'], 'aa': ['c']} is prefix grammar.
As converting a prefix grammar to right linear grammar goes,the first step of adding the basic rules can be done in polynomial time, the challenging part is the addition of the rules of type $V_{a_i} -> w V_{a_j}$, because we have to parse through the existing Right linear grammar continuously till no such rules can be added as mentioned in the step 2 of the algorithm.
After the initial productions G_{init} is generated as there are finite number of non terminals excluding 'S', for each of the non-terminals, search if this non terminal is a prefix of any of the right hand side terminals, if P is the non-terminal search all the rules to find a combination which starts from 'S' where p is a prefix

$S -> \phi_1 A$, $A -> \phi_2 B \dots, Q -> \phi_k w R$

of $\phi_1 \phi_2 \dots \phi_K w$ then a rule of type $P -> wR$ is added note that w is the suffix and $p = \phi_1 \phi_2 \dots \phi_K$. A recursive function is good enough as it will keep track of the previous calls and return back to pursue other rules of the existing right linear grammar
Imagine a big tree with many branches and connectors where one starts with 'S' and parses through the tree to find if any such paths result in a prefix of the

non-terminals, there is a possibility if any of these conditions satisfy:
if in search of 'P' a non terminal

1. a rule $A->B$ i.e there is non-terminal string on the right hand side, there is a possibility as rules starting with B can result in rule where P is a prefix. so make a recursive call that searches all the rules starting with B.

2. a rule $A->xB$ where 'P' is a prefix of 'x' add the rule $P->suffix(p,x)B$.

3. a rule $A->xB$ where 'x' is a prefix of 'P', there is a possibility if suffix(x,p) is a prefix of any of rules starting from B, so make a recursive call with starting point as B and search if suffix(x,p) is a prefix.

The same conditional arguments are in the function **main**.
There are a few nomenclatures in the Program which would be easier for further explanation:
Note: P is non-terminal we are searching and A-¿xB is the popped rule from the stack of rules of 'A' .

1. start: a string ,parameter of main function, start is a non-terminal on the left hand side of the rules, the rules that are parsed through in the present function call .Example: start is A here, because we are parsing through the rules of A(rules with A as left hand side of the non-terminal).

2. target: a string, parameter of main function, the non-terminal or part of a non-terminal for which we search a rule where the non-terminal or part of a non-terminal is the prefix of the terminals on the right hand side. Example: 'P' and suffix(x,P) can be the target based on the 3 conditions explained above.

3. finaltarget: a string, parameter of main function, the idea is to keep track of the original non-terminal for which we are searching a rule for.Example :P is always the final target because a recursive call function is set in motion in the function pgtorlg in search of a rule with P as nonterminal on the left hand side terminal. path: a list of tuples,parameter of main function, which has tuples of type (start,target,finaltarget).

The infinite-loop condition: for example if$A->B$ and $B->A$ are part of the existing right linear grammar, while parsing through the existing rules in search of new rules we will come across these and can be stuck in an infinite loop, to avoid such a condition just after main function starts there is a list named path to which we add the tuples of the type (start,target,finaltarget) , this list of tuple is chosen because while parsing if a start(left hand side non terminal) is seen again with the same combination of target and final target then that means it is in a infinite- loop because in between these tuples there have been other tuples (other recursive calls) and there has been no change in target and the final target.

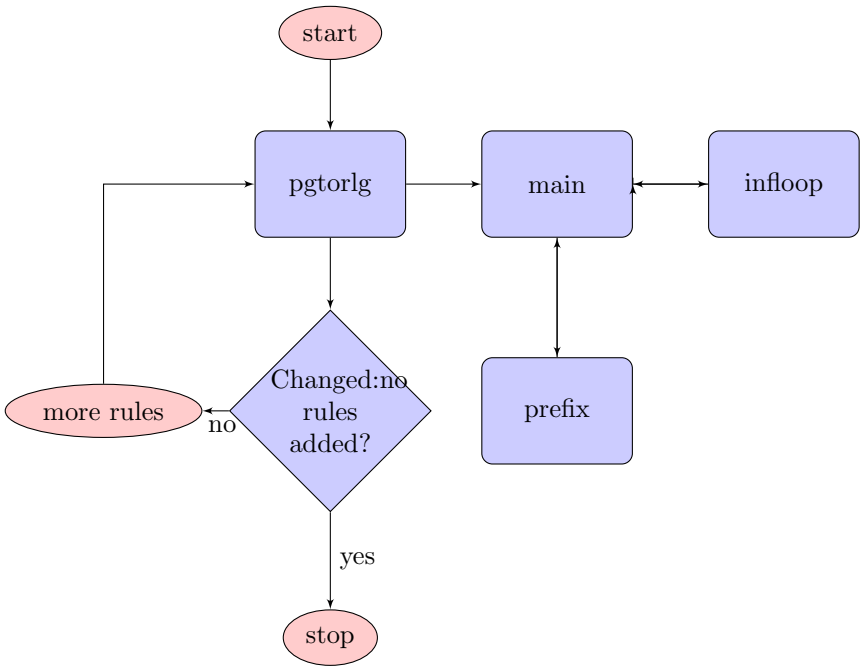

Figure 1: Flow of the whole program

5 Implementation

The whole flow of the code is represented in a flowchart as shown in the figure 1, all the violet coloured figures are functions which include(all the helper functions unnecessary in the process of adding new rules are not shown in flow chart eg : printer,cstack) :

1. **pgtorlg** as in prefix grammar to right linear grammar
2. **main** adds the rules of type $V_{a_i} -> wV_{a_j}$
3. **prefix** checks if a string is prefix of another string and returns suffix
4. **infloop** checks if the recursive calls have got into a infinite-loop
5. **changed** checks if no new production is added to right linear grammar
6. **cstack** helper function which gives a copy of a list, to keep intact the lists in the dictionary(contains right linear grammar)
7. **cleaning** helper function removes rules of type $A -> A$
8. **printer** helps print the right linear grammar rather than just displaying the dictionary

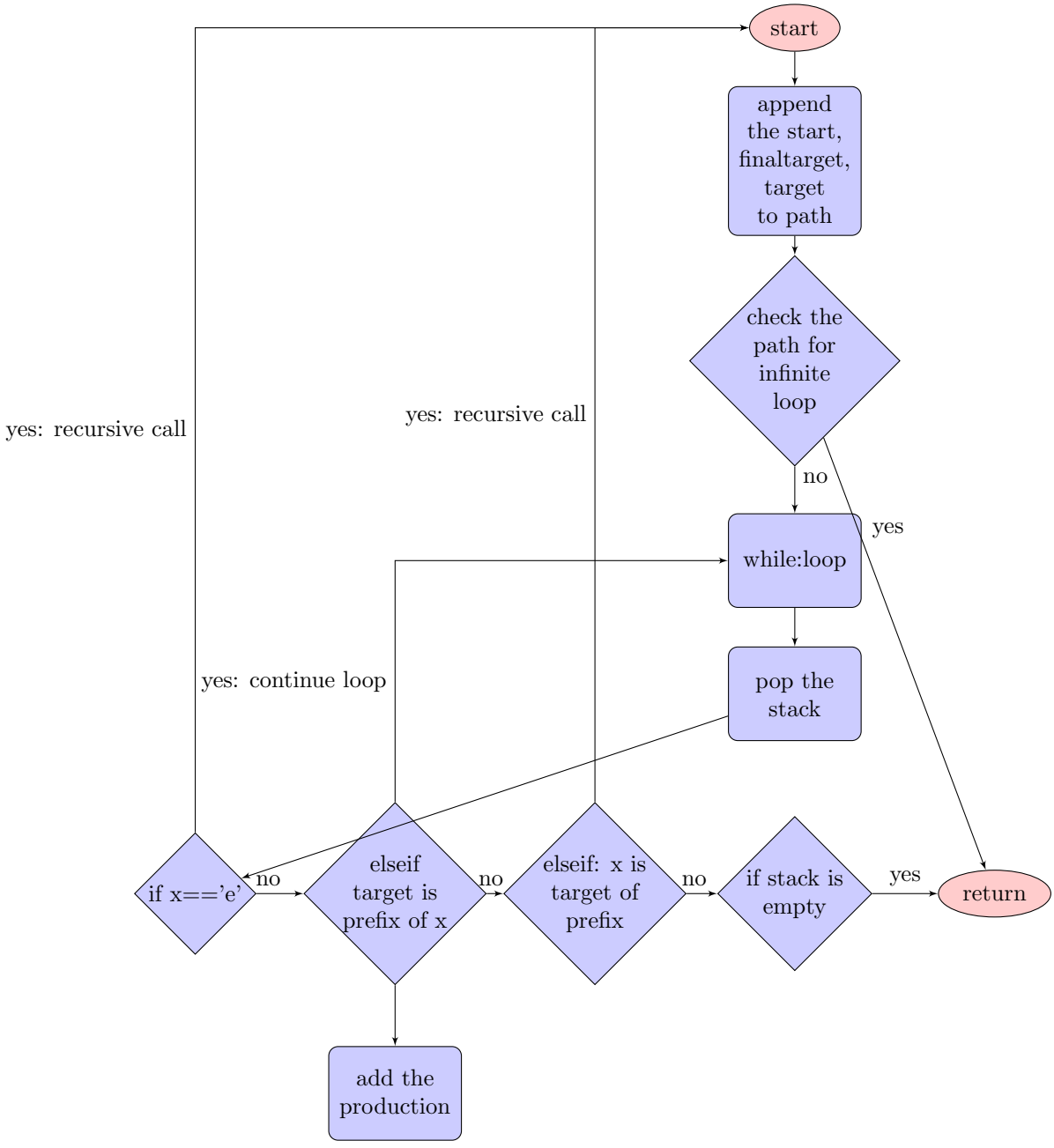

Figure 2: How rule of type $V_{a_i} -> wV_{a_j}$ are added

The pgtorlg function adds he basic rules which include $S->l_1|\ldots|l_n|b_1V_{a_1}|\ldots|b_mV_{a_m}$, and later gets into a while loop where it continuously calls the main function and doesn't break until no new rule is added to the right linear grammar.The calls to main function are made in this while loop where the remaining rules of type $V_{a_i}->wV_{a_j}$ are added. there is flowchart for the main, prefix,infloop functionality separately in figure 2. The initial call of main from the pgtorlg is made with start='S', target = non-terminal for which we searching a new rule, finaltarget here is the same as target but remember that we included finaltarget as parameter of main to keep track of the original non-terminal for adding the rules.
In figure 2 the flow starts with start and the appending of (start,target,finaltarget) is added to path to keep track of the recursive function calls for a final target and make sure it doesn't get into a infinite loop.
Then there is conditional check to make sure there are no infinite loop for which a function named infloop was written,if a infinite-loop is detected it ends that recursive call as in returns to the previous recursive call resuming its parsing.
Then we enter the while loop where where first the stack of the rules of start are popped with each iteration and checked if any of the three conditions are met as mentioned in the previous section.if any of those conditions are met either a rule is added or a recursive call is made, if no conditions are met the next rule in the stack is popped and the loop continues.
The loop is broken when the stack is empty as in all the rules are parsed through.
The function main returns dictionary representing the right linear grammar, this return is no where utilised in the recursive calls but is used to compare the previous dictionary of right linear grammar in the pgtorlg function to make a choice of ending the program if no new rule is added .

Note:If you got the basic idea of the implementation you can skip this section, else please go through the description and detailed explanation of function main and pgtorlg.

1. Function **pgtorlg**:The parameter of the function is the Prefix-grammar, represented by: a tuple of symbols, list of base strings, productions are represented using a dictionary where the keys are strings of left-hand side of the productions and the values are list of strings of all the rights-hand side of the the productions.
   The function returns a dictionary which is a representation of the Right linear grammar, where the keys are the left hand side of the rules in the right-linear grammar and the value are list of tuples where each tuple has two strings, the

   The first step in the algorithm as mentioned in section 3 was to create the initial productions, which include adding the base strings to the productions and productions of the form
   $S->l_1|\ldots|l_n|b_1V_{a_1}|\ldots|b_mV_{a_m}$.
   The base strings are added to the Right-linear grammar in the first 'FOR' loop, all the base strings are iterated over ,there is an if else to check if an

'e' is present in the base string, in that case an empty string is added to the Right linear grammar.
The productions of the type
$S->l_1|\ldots|l_n|b_1V_{a_1}|\ldots|b_mV_{a_m}$.
are added by iterating over the productions of the prefix grammar in second 'For' loop which is nested.
The second step of the Algorithm starts here where there is a loop that doesn't end until there is no new production added to the Right linear grammar.

There are two function calls here

- **main**: For adding the remaining productions of the type
  $V_{a_i}->wV_{a_j}$ .The main function call is further enclosed in a for loop iterating over the non-terminals excluding 'S',i.e before checking if a new rule is added, a search for new rules for each non-terminal(as left hand side) excluding 'S' is made.

- **changed**: To check if a new rule is added to the Right linear grammar, else terminate calls to main function i.e the whole algorithm.

2. Function **main**:The parameters of this function are

   - start: string,the non-terminal's rule that we are presently parsing through:example the start is 'S' for all the starting calls
   - target:string, the target might be a prefix of a terminal string on the right hand side of the rule .
   - finaltarget:string, the non-terminal which is solely a paramter to keep track of the original target while adding the rule
   - stack: Has the list of rules of the start, which are popped and are made to go through the three conditions mentioned below.
   - path: The list of tuple (start,target,finaltarget), keeps track of the recursive and helps in maintaining a infinite-loop free environment. Remember path is different for each call from pgtorlg.
   - dic:dictionary of right linear grammar to keep track of it add all the new rules to it.

   The function returns a dictionary of the existing Right-linear grammar. This is a recursive function, and in each of these recursive calls first there is a list 'path' to which that track the path of the recursive calls, an 'if' condition to check if the function went into a infinite loop.
   Then a while loop starts and a stack is popped, the loop ends when the stack is empty, in this loop are the three conditions below.
   The remaining productions of type $V_{a_i}->wV_{a_j}$
   can be added if:

- The present rule has w=='e' as in empty string, there is still a possibility of finding a prefix for a_i in rules starting with V_{a_j} : make a recursive call with start as a_j and target as same a_i as we din't parse through anything and final target as a_i
- The present a_i is a prefix of any terminals on the right hand side of the productions in the existing Right linear grammar: just add the rule and continue the loop

- If the terminal string on the right hand side is a prefix of any of the a_i , we have to parse through the remaining rules with the suffix. i.e make a recursive call with target as suffix but final target as a_i and start as a_j.

Then there is the return statement to return the dictionary after all the recursive calls to the pgtorlg where a comparison is made with the previous one, if there is no difference then the whole program ends.

Algorithm 1 pgtorlg : creates G_{init} and calls main for the remaining productions

```
1: function PGTORLG (Symbols:Tuple,Strings:Array,Productions:Dictionary)
2:     G_init ← emptydictonary
3:     G_init['S'] ← []
4:     for i in Strings do                          ▷ Adding Strings to G_init
5:         if i ==' e' then
6:             G_init['S'] ← G_init['S'] + [('','')]
7:         else
8:             G_init['S'] ← G_init['S'] + [('', i)]
9:         end if
10:    end for
11:    for i in Production.keys() do                ▷ Adding Productions to G_init
12:        for j in Productions[i] do:
13:            if j ==' e' then
14:                G_init['S'] ← G_init['S'] + [(i,'')]
15:            else
16:                G_init['S'] ← G_init['S'] + [(i, j)]
17:            end if
18:        end for
19:    end for
20:                                  ▷ All the Initial Productions have been added
21:    Change ← False
22:    while not Change do                  ▷ V_ai -> wV_aj are added in main
23:        q ← G_init
24:        for i in Productions do          ▷ for each non-terminal excluding 'S'
25:            G_init ← main(CStack(G_init['S']),' S', G_init, i, i, []) ▷ function call
   to main
26:        end for
27:        if not Changed(q,G_init) then            ▷ end if no new rule added
28:            Change ← False
29:        end if
30:    end while
31:    return G_init
32: end function
```

Algorithm 2 main : adds the remaining productions

```
1: function MAIN(Stack:Array,Start:Str,dic:dictionary,Target:str,Finaltarget:str,path:Array)
2:     done ← False
3:     Stack1 ← CStack(Stack)
4:     path ← path + [(Start, Target, Finaltarget)]      ▷ Helps detect Infinite loops
5:     if infloop(path) then   ▷ End this recursive call if infinite loop present
6:         done ← True
7:     end if
8:     while not done do
9:         t ← stack.pop()                                  ▷ pop the first rule of start
10:        if t[1] ==″ then                ▷ if Right side terminal string is empty
11:            if  t[0] in dic then
12:                main(CStack(dic[t[0]]),t[0],dic,Target,Finaltarget,path)      ▷ recursive call with new start
13:            end if
14:        else if Prefix(Target,t[1])[0] then   ▷ if target is prefix of a teminal string on the right hand side
15:            if Finaltarget not in dic then
16:                dic[Finaltarget] ← []
17:            end if
18:            if (tuple1[0],prefix(Target,t[1])[1]) not in dic[Finaltarget] then
19:                dic[Finaltarget] ← dic[Finaltarget] + (t[0], prefix(Target, t[1])[1])      ▷ adding new rule
20:            end if
21:        else if Prefix(t[1],Target)[0] then   ▷ terminal string on right hand side is prefix of target
22:            if t[0] in dic then
23:                temp ← prefix(t[1], Target)[1]
24:                main(cstack(dic[t[0]]),t[0],dic,temp,Finaltarget,path)      ▷ Recursive call with suffix as target
25:            end if
26:        end if
27:        if Stack1 is not empty then
28:            done ← True
29:        end if
30:    end while
31: end function
```

Algorithm 3 Prefix : Checks if s is a prefix of s1

```
1: function PREFIX(s:Str s1:Str)
2:     Prefixes ← [s1]
3:     for i 1 . . . to n do                          ▷ Adds all the prefixes of s1
4:         prefixes ← Prefixes + s1[0 : i]
5:     end for
6:     prefixes.remove('')                            ▷ removes the empty string
7:     if s in Prefixes then                          ▷ Checks if s is a prefix of s1
8:         return (True,s1[len(s):])
9:     end if
10:    return (False,")
11: end function
```

Algorithm 4 infloop : Checks if s is a prefix of s1

```
1: function INFLOOP(L:List)
2:     dic ← emptydictionary
3:     for i in L do
4:         if i[0] not in dic then
5:             dic[i[0]] ← []
6:         end if
7:         if (i[1],i[2]) not in dic[i[0]] then: ▷ checking if we came through the
   same call before
8:             dic[i[0]] ← dic[i[0]] + ((i[1], i[2]))
9:         else
10:            return True                            ▷ infinite loop so return true
11:        end if
12:    end for
13:    return False
14: end function
```

6 Complexity of Python Implementation

As mentioned in [1] the complexity of the algorithm is going to be exponential.let the number of base strings be n and number of productions be m, therefore the number of non-terminals of G_R are m+1(including S). let γ be length of the longest b_i which are right-hand side terminal of Productions of G_P.let α be the longest base strings in G_P.let β be the longest a_i which are left-hand side terminal of Productions of G_P
The complexity of step 1 in algorithm is
O((m+n+1)*($\gamma + \alpha$)), because:
Parsing through each of the base strings and productions takes O(m+n+1), but for each of these a prefix check is necessary before adding them to the productions of G_R, as each prefix check can either take a time of O(γ) or $O(\alpha)$ so

we can say that O((m+n+1)*($\gamma + \alpha$))is the upper bound (not a strict upper bound).
The second and third step's complexity, the most costing in the algorithm :
$O(m * (m+1) * (n|\alpha| + m|\gamma| + 1) * (\gamma + \alpha) * m * \beta * (m+n+1)^2)$, because:

β is the longest a_i and each character of a_i can be parsed in $(m+n+1)^2$ recursive-function calls in function "main" of Algorithm 2, because recursive-function calls more than $(m+n+1)^2$ and not parsing through a character means there is an infinite loop which is the function of function "infloop" in algorithm 4, that path is stopped at $(m+n)^2$. Therefore to check if a new rule involving a_i can be added can be done in $O(3 * \beta * (m+n+1)^2)$(3 because of 3 if conditions in the main function, but can be dropped as it is a constant).
There are m such a_i and not to forget the prefix check cost which is O($\gamma + \alpha$) there for the cost to add a single production to G_R is :
$O((\gamma + \alpha) * m * \beta * (m+n+1)^2)$
From claim 4 of [1] we know that maximum of $m * (m+1) * (n|\alpha| + m|\gamma| + 1)$ productions can be added to G_R, note that this is a worst case scenario if a new rule is not generated in an iteration the changed function in Algorithm 5 will terminate the whole Algorithm.
When you add the complexity of step 1 to step 2 and 3, it can be removed as it is lower polynomial compared to the step 2 and 3.
Therefore the complexity is:
$O(m * (m+1) * (n|\alpha| + m|\gamma| + 1) * (\gamma + \alpha) * m * \beta * (m+n+1)^2)$

7 Conclusion

Implementation of the algorithm was complete, is it the most efficient one? that is up for debate may be a iterative rather than recursive function can be written but it may not curb the function call or parsing it will take to transform a prefix grammar to right-linear grammar, but has a potential of curbing the space complexity as it will reduce the parameters in the recursive function 'main'.This might help in the next step i.e of finding a way to establish a polynomial size relationship (bidirectional) between prefix grammars and any representations of regular grammars as mentioned in [1].

8 References

[1] Michael Frazier, C. David Page Jr. Prefix grammars: an alternative characterization of the regular languages. Information Processing Letters 51(2): 67–71, July 26, 1994.

9 Appendix

9.1 Instructions

compiling the file shows you the results of a few inputs, you can change/edit the commands in __name__ == '__main__':

command:pgtorlg(tuple of symbols,list of base strings,dictionary of productions)

please do not use 'e' in the productions as it has been assumed to be as empty string in this program.
Note:dictionary of productions: where the keys are left hand side strings of the productions and values are list of strings on the right hand side.
output:prints the right-linear grammar example: lets run G2 from [1]: pgtorlg(('a','b'),['ab'],{'a':['aab'],'aababb':['ab','abb']}) Note:e denotes epsilon uppercase letters enclosed in ¡¿ are non terminals lowercase letters are terminals
$< S > -- > ab$
$< S > -- > aab < A >$
$< S > -- > ab < AABABB >$
$< S > -- > abb < AABABB >$
$< A > -- > b$
$< A > -- > ab < A >$
$< A > -- > b < AABABB >$
$< A > -- > bb < AABABB >$
$< AABABB > -- > e$
$< AABABB > -- > b < AABABB >$

9.2 Examples(generated by program)

- The prefix grammar is ('a', 'b') ['ab'] {'a': ['aab'], 'ab': ['abb']} The Right-linear grammar is:
  $< S > -- > ab$
  $< S > -- > aab < A >$
  $< S > -- > abb < AB >$
  $< A > -- > bb < AB >$
  $< A > -- > ab < A >$
  $< A > -- > b$
  $< AB > -- > b < AB >$
  $< AB > -- > e$

- The prefix grammar is ('a', 'b') ['ab'] {'a': ['aab'], 'aababb': ['ab', 'abb']} The Right-linear grammar is:
  $< S > -- > ab$
  $< S > -- > aab < A >$
  $< S > -- > ab < AABABB >$

$< S > -- > abb < AABABB >$
$< A > -- > bb < AABABB >$
$< A > -- > b < AABABB >$
$< A > -- > ab < A >$
$< A > -- > b$
$< AABABB > -- > e$
$< AABABB > -- > b < AABABB >$

- The prefix grammar is ('a', 'b', 'c') ['aa'] {'a': ['ab'], 'aa': ['c']} The Right-linear grammar is:
  $< S > -- > aa$
  $< S > -- > ab < A >$
  $< S > -- > c < AA >$
  $< A > -- > b < A >$
  $< A > -- > a$
  $< AA > -- > e$

- The prefix grammar is ('a', 'b') ['ab'] {'a': ['aab'], 'aababb': ['ab', 'abb'], 'ab': ['b']} The Right-linear grammar is:
  $< S > -- > ab$
  $< S > -- > aab < A >$
  $< S > -- > ab < AABABB >$
  $< S > -- > abb < AABABB >$
  $< S > -- > b < AB >$
  $< A > -- > bb < AABABB >$
  $< A > -- > b < AABABB >$
  $< A > -- > ab < A >$
  $< A > -- > b$
  $< AABABB > -- > e$
  $< AABABB > -- > b < AABABB >$
  $< AB > -- > b < AABABB >$
  $< AB > -- >< AABABB >$
  $< AB > -- > e$

- The prefix grammar is ('a', 'b', 'c', 'd') ['abcd'] {'a': ['cdd'], 'cd': ['abb']} The Right-linear grammar is:
  $< S > -- > abcd$
  $< S > -- > cdd < A >$
  $< S > -- > abb < CD >$
  $< A > -- > bb < CD >$
  $< A > -- > bcd$
  $< CD > -- > d < A >$

- The prefix grammar is ('a', 'b', 'c', 'd') ['ab', 'cd'] {'a': ['b'], 'b': ['a'], 'ab': ['bba'], 'c': ['d'], 'd': ['c']} The Right-linear grammar is:

$< S > -- > ab$
$< S > -- > cd$
$< S > -- > b < A >$
$< S > -- > a < B >$
$< S > -- > bba < AB >$
$< S > -- > d < C >$
$< S > -- > c < D >$
$< A > -- >< B >$
$< A > -- > b$
$< B > -- > ba < AB >$
$< B > -- >< A >$
$< AB > -- > e$
$< AB > -- > a < AB >$
$< C > -- >< D >$
$< C > -- > d$
$< D > -- >< C >$

- The prefix grammar is ('a', 'b') ['ba'] {'b': ['bab'], 'ba': ['a'], 'a': ['bb']} The Right-linear grammar is:
$< S > -- > ba$
$< S > -- > bab < B >$
$< S > -- > a < BA >$
$< S > -- > bb < A >$
$< B > -- > b < A >$
$< B > -- > ab < B >$
$< B > -- > a$
$< BA > -- > b < B >$
$< BA > -- > e$
$< A > -- >< BA >$

- The prefix grammar is ('a', 'b') ['ab'] {'a': ['e'], 'b': ['ab'], 'ab': ['e ']} The Right-linear grammar is:
$< S > -- > ab$
$< S > -- >< A >$
$< S > -- > ab < B >$
$< S > -- > e < AB >$
$< A > -- > b < B >$
$< A > -- > b$
$< B > -- > e$
$< AB > -- >< B >$
$< AB > -- > e$

- The prefix grammar is ('a', 'b', 'c') ['abc'] {'a': ['e'], 'bc': ['ab'], 'b': ['c'], 'c': ['a']} The Right-linear grammar is:
$< S > -- > abc$

$< S > -- >< A >$
$< S > -- > ab < BC >$
$< S > -- > c < B >$
$< S > -- > a < C >$
$< A > -- >< C >$
$< A > -- > b < BC >$
$< A > -- > bc$
$< BC > -- > e$
$< B > -- > c$
$< B > -- >< BC >$
$< C > -- >< B >$
$< C > -- > e$

- The prefix grammar is ('a', 'b') ['ab'] {'a': ['aab'], 'aa': ['aabba'], 'aabba': ['e']} The Right-linear grammar is:
  $< S > -- > ab$
  $< S > -- > aab < A >$
  $< S > -- > aabba < AA >$
  $< S > -- >< AABBA >$
  $< A > -- > abba < AA >$
  $< A > -- > ab < A >$
  $< A > -- > b$
  $< AA > -- > bba < AA >$
  $< AA > -- > b < A >$
  $< AABBA > -- >< AA >$

- The prefix grammar is ('a', 'b', 'c') ['bcb'] {'b': ['a', 'e'], 'ac': ['e'], 'c': ['e']} The Right-linear grammar is:
  $< S > -- > bcb$
  $< S > -- > a < B >$
  $< S > -- >< B >$
  $< S > -- >< AC >$
  $< S > -- >< C >$
  $< B > -- > cb$
  $< B > -- > e$
  $< AC > -- > b$
  $< C > -- > b$

- The prefix grammar is ('a', 'b', 'c') ['c'] {'c': ['ab'], 'a': ['e'], 'b': ['c']} The Right-linear grammar is:
  $< S > -- > c$
  $< S > -- > ab < C >$
  $< S > -- >< A >$
  $< S > -- > c < B >$

$< C > -- >< B >$
$< C > -- > e$
$< A > -- > b < C >$
$< B > -- >< C >$

- The prefix grammar is ('a', 'b') ['a'] {'a': ['bab'], 'ba': ['a'], 'ab': ['e'], 'bab': ['b']} $TheRight - linear grammaris$:
$< S > -- > a$
$< S > -- > bab < A >$
$< S > -- > a < BA >$
$< S > -- >< AB >$
$< S > -- > b < BAB >$
$< A > -- >< BA >$
$< A > -- > e$
$< BA > -- > b < A >$
$< AB > -- >< A >$
$< BAB > -- >< A >$ ———————————————————

- The prefix grammar is ('a', 'b') ['abab'] {'aba': ['e'], 'b': ['a'], 'ab': ['b']} The Right-linear grammar is:
$< S > -- > abab$
$< S > -- >< ABA >$
$< S > -- > a < B >$
$< S > -- > b < AB >$
$< ABA > -- > b$
$< B > -- >< AB >$
$< B > -- > e$
$< AB > -- > ab$ ———————————————————

- The prefix grammar is ('a', 'b') ['ababa'] {'a': ['e', 'bb'], 'bab': ['e'], 'b': ['aba']} The Right-linear grammar is:
$< S > -- > ababa$
$< S > -- >< A >$
$< S > -- > bb < A >$
$< S > -- >< BAB >$
$< S > -- > aba < B >$
$< A > -- > ba < B >$
$< A > -- > baba$
$< A > -- > e$
$< BAB > -- > a$
$< BAB > -- >< A >$
$< B > -- > b < A >$
$< B > -- > aba$
$< B > -- > a < B >$
———————————————————

- The prefix grammar is ('c', 'b') ['bc'] {'b': ['bc', 'c', 'e'], 'ccc': ['bc'], 'c': ['e']} The Right-linear grammar is:
  $< S > -- > bc$
  $< S > -- > bc < B >$
  $< S > -- > c < B >$
  $< S > -- >< B >$
  $< S > -- > bc < CCC >$
  $< S > -- >< C >$
  $< B > -- > c < CCC >$
  $< B > -- > c < B >$
  $< B > -- > c$
  $< CCC > -- > e$
  $< CCC > -- >< B >$
  $< C > -- > e$
  $< C > -- >< B >$
  $< C > -- >< CCC >$

- The prefix grammar is ('a', 'd', 'c', 'b') ['abcd'] {'b': ['acd', 'e'], 'a': ['bcd', 'e'], 'c': ['e', 'abd'], 'd': ['abc', 'e']} The Right-linear grammar is:
  $< S > -- > abcd$
  $< S > -- > acd < B >$
  $< S > -- >< B >$
  $< S > -- > bcd < A >$
  $< S > -- >< A >$
  $< S > -- >< C >$
  $< S > -- > abd < C >$
  $< S > -- > abc < D >$
  $< S > -- >< D >$
  $< B > -- > cd < A >$
  $< B > -- > cd$
  $< B > -- > d < C >$
  $< B > -- > c < D >$
  $< A > -- > bc < D >$
  $< A > -- > bd < C >$
  $< A > -- > cd < B >$
  $< A > -- > bcd$
  $< C > -- > d < B >$
  $< C > -- > d < A >$
  $< C > -- >< D >$
  $< C > -- > d$
  $< D > -- >< A >$
  $< D > -- >< B >$
  $< D > -- >< C >$
  $< D > -- > e$

- The prefix grammar is ('a', 'g') ['aga'] {'ag': ['a'], 'a': ['ag'], 'aga': ['e'], 'aa': ['e']} The Right-linear grammar is:
  $< S > -- > aga$
  $< S > -- > a < AG >$
  $< S > -- > ag < A >$
  $< S > -- >< AGA >$
  $< S > -- >< AA >$
  $< AG > -- >< A >$
  $< AG > -- > a$
  $< A > -- > g < A >$
  $< A > -- >< AG >$
  $< A > -- > ga$
  $< AGA > -- > e$
  $< AA > -- > e$

- The prefix grammar is ('c', 'b') ['bccb'] {'b': ['bc'], 'bc': ['bccc'], 'bccc': ['b'], 'bb': ['e']} The Right-linear grammar is:
  $< S > -- > bccb$
  $< S > -- > bc < B >$
  $< S > -- > bccc < BC >$
  $< S > -- > b < BCCC >$
  $< S > -- >< BB >$
  $< B > -- >< BCCC >$
  $< B > -- > ccc < BC >$
  $< B > -- > c < B >$
  $< B > -- > ccb$
  $< BC > -- > cc < BC >$
  $< BC > -- >< B >$
  $< BC > -- > cb$
  $< BC > -- > c < BC >$
  $< BC > -- > b$
  $< BCCC > -- >< BC >$
  $< BCCC > -- > b$
  $< BCCC > -- > cb$
  $< BCCC > -- >< B >$
  $< BCCC > -- > cc < BC >$
  $< BCCC > -- > c < BC >$
  $< BB > -- > e$

9.3 Code in python3

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 14 21:58:09 2020
```

```
@author: Keshav Raminedi
"""

def parser(productions):
    '''prints the Right linear grammar for legibility
    input:dictonary
    output: returns nothing
    '''
    for i in productions:
        for j in productions[i]:
         if (j[0]!='' and j[0]!='e') and (j[1]=='' or j[1]=='e'):
             #print(1)
             print('<'+i.upper()+'>'+' --> '+'<'+j[0].upper()+'>'),
         elif (j[0]==''or j[0]=='e') and (j[1]!='' and j[1]!='e'):
            #print(2)
            print('<'+i.upper()+'>'+' --> '+j[1])
         elif (j[0]==''or j[0]=='e') and (j[1]=='' or j[1]=='e'):
            #print(3)
            print('<'+i.upper()+'>'+' --> '+'e')
         else:
           #print(4)
           print('<'+i.upper()+'>'+' --> '+j[1]+'<'+j[0].upper()+'>'),
##########################################################################################################
def cleaning(dic):
  '''cleans the grammar
  dic:dictonary
  returns:dictonary'''
  for i in dic:
      for j in dic[i]:
       if j[1]=='':
           if j[0]==i:
               dic[i].remove(j)
  return dic
##########################################################################################################
def cstack(l):
 '''returns the copy of a list
 l: list
 returns:list'''
 p=[]
 for i in l:
  p.append(i)
 return p
def prefix(s,s1):
 '''checks if s is a prefix of s1
 s:string
 s1:string
```

```
 return:tuple(boolean,string)'''
 prefixes=[s1]
 for i in range(0,len(s1)):
   prefixes.append(s1[0:i])
 prefixes.remove('')
 if s in prefixes:
       return (True,s1[len(s):])# returning true when it is a prefix with the suffix
 return (False,'')#return false when not a prefix with an empty string

##########################################################################################
def inf_loop(l):
    '''checks if the main function is paring through a infinite loop that is going nowhere
    l:list
    return:boolean'''
    dic={}
    for i in l:
        if i[0] not in dic:
            dic[i[0]]=[]
        if (i[1],i[2]) not in dic[i[0]]: # if in dic then it is an infinite loop
         dic[i[0]].append((i[1],i[2]))
        else:
           return True
    return False
##########################################################################################
def main(stack,start,dic,target,finaltarget,path):
 '''produces rules other than Ginit
 Ginit:initial rules after
 stack:list
 dic:dictonary
 target:string
 finaltarget:string
 path=list
 path1=list
 returns:dictonary'''
 done=False
 stack1=cstack(stack)
 #start1=start
 target1=target
 finaltarget1=finaltarget
 path.append((start,target,finaltarget))
#inifite loop check
 if inf_loop(path):
     done=True
 '''non-terminal start: P...for each rule in stack of form A->xB.......'''
 while(not done):
     tuple1=stack1.pop()
```

```
     if tuple1[1]=='':#if x=='e'
         if tuple1[0] in dic:
          main(cstack(dic[tuple1[0]]),tuple1[0],dic,target1,finaltarget1,path)#recursive ca
     elif prefix(target1,tuple1[1])[0]:# if P is prefix of x
         if finaltarget1 not in dic:
           dic[finaltarget1]=[]
         if (tuple1[0],prefix(target1,tuple1[1])[1]) not in  dic[finaltarget1]:
          dic[finaltarget1].append((tuple1[0],prefix(target1,tuple1[1])[1]))
     elif prefix(tuple1[1],target1)[0]:# if x is prefix of P
        if tuple1[0] in dic:
         target2=prefix(tuple1[1],target1)[1]
         main(cstack(dic[tuple1[0]]),tuple1[0],dic,target2,finaltarget1,path)#recursive cal
     if not stack1:# when stack is empty end the loop
      done=True
 return dic

 ##########################################################################################
def change(p,q):
    '''returns false if there is no difference between two dictonaries
    p=dictonary
    q=dictonary
    return:boolean'''
    if p.keys()!=q.keys():
        return True
    for i in p.keys():
        if p[i]!=q[i]:
            return True #return true no change
    return False
##########################################################################################
def pgtorlg(symbols,strings,productions):
    ''' produces Ginit and calls the main function which produces the remainig rules
    symbols:tuple of symbols in prefix grammar
    strings:list of strings
    productions:dictonary
    returns:dictonary (right linear grammar )'''
    print('The prefix grammar is ')
    print(symbols,strings,productions)
    productions_rlg_initial={'S':[]}
    '''.....
    for i in strings: #adds the base strings to right linear grammar
     if i=='e':
      productions_rlg_initial['S'].append(('',''))
     else:
      productions_rlg_initial['S'].append(('',i))
    '''.....
    for i in productions.keys():#adds the initial productions of type S->xA
```

```
      for j in productions[i]:

       if j=='e':

        productions_rlg_initial['S'].append((i,''))
       else:
        productions_rlg_initial['S'].append((i,j))
    '''.....................................................................................'
    p=productions_rlg_initial
    changed=False
    while(not changed):#continously checking for new rules for each nonterminal except 'S'
        q=p.copy()
        for i in productions.copy():
            p=main(cstack(p['S']),'S',p,i,i,[]) #call to main function for rules of type A-
        if not change(p,q):# if  no new rule can be added
            changed=True #end the while loop
    #print(p)
    print('The Right-linear grammar is:')
    parser(cleaning(p))
##########################################################################################################
def space():
  print('-------------------------------------------------')
##########################################################################################################
if  __name__ == '__main__':

    pgtorlg(('a','b'),['ab'],{'a':['aab'],'ab':['abb']})
    space()

    pgtorlg(('a','b'),['ab'],{'a':['aab'],'aababb':['ab','abb']})
    space()

    pgtorlg(('a','b','c'),['aa'],{'a':['ab'],'aa':['c']})
    space()
    pgtorlg(('a','b'),['ab'],{'a':['aab'],'aababb':['ab','abb'],'ab':['b']})
    space()
    pgtorlg(('a','b','c','d'),['abcd'],{'a':['cdd'],'cd':['abb']})
    space()
    pgtorlg(('a','b','c','d'),['ab','cd'],{'a':['b'],'b':['a'],'ab':['bba'],'c':['d'],'d':['
    space()
    pgtorlg(('a','b'),['ba'],{'b':['bab'],'ba':['a'],'a':['bb']})
    space()
    pgtorlg(('a','b'),['ab'],{'a':['e'],'b':['ab'],'ab':['e ']})
    space()
    pgtorlg(('a','b','c'),['abc'],{'a':['e'],'bc':['ab'],'b':['c'],'c':['a']})
    space()
    pgtorlg(('a','b'),['ab'],{'a':['aab'],'aa':['aabba'],'aabba':['e']})
```

```
space()
pgtorlg(('a','b','c'),['bcb'],{'b':['a','e'],'ac':['e'],'c':['e']})
space()
pgtorlg(('a','b','c'),['c'],{'c':['ab'],'a':['e'],'b':['c']})
space()
pgtorlg(('a','b','c'),['bc'],{'b':['e'],'c':['abab'],'aba':['b'],'bb':['e']})
space()
pgtorlg(('a','b'),['ab'],{'b':['e'],'a':['b'],'bb':['b']})
space()
pgtorlg(('a','b'),['a'],{'a':['bab'],'ba':['a'],'ab':['e'],'bab':['b']})
space()
pgtorlg(('a','b'),['abab'],{'aba':['e'],'b':['a'],'ab':['b']})
space()
pgtorlg(('a','b'),['ababa'],{'a':['e','bb'],'bab':['e'],'b':['aba']})
space()
pgtorlg(('c','b'),['bc'],{'b':['bc','c','e'],'ccc':['bc'],'c':['e']})
space()
pgtorlg(('a','d','c','b'),['abcd'],{'b':['acd','e'],'a':['bcd','e'],'c':['e','abd'],'d'
space()
pgtorlg(('a','g'),['aga'],{'ag':['a'],'a':['ag'],'aga':['e'],'aa':['e']})
space()
pgtorlg(('c','b'),['bccb'],{'b':['bc'],'bc':['bccc'],'bccc':['b'],'bb':['e']})
```