Practica 4

Alumno: *Ángel López Manríquez*
Tema: *Gramaticas libres de contexto*
Fecha: *17 de junio de 2019*
Grupo: *2CV1*

M. en C. Luz María Sánchez García

Instituto Politecnico Nacional
Escuela Superior de Cómputo

Índice

Gramaticas libres de contexto

Lopez Manriquez Angel 2CV1

Mayo 2018

1. Introducción

En esta práctica se implementará un programa que determina si una palabra $w \in L(G)$, es decir, la pertenencia de una palabra al lenguaje generado por una gramatica en `python`, para lo cual se hizo una clase *Grammar* la cual usa un grafo dirigido para poder remover producciones unarias.

1.1. Gramáticas

La gramática es el estudio de las reglas y principios que determinan el uso de un lenguaje y cómo se organizan las palabras dentro de una oración. Nos sirven para especificar, de manera finita, el conjunto de cadenas de símbolos que constituyen un lenguaje.

Formalmente, una gramática G es la cuádrupla $G = (V_T, V_N, S, P)$, donde:

- V_T es un conjunto finito de símbolos terminales.
- V_N es un conjunto finito de símbolos no terminales.
- S es el símbolo no terminal inicial, $S \in V_N$.
- P es un conjunto finito de producciones o reglas de derivación.

Todas las cadenas que podamos generar a través de G están formadas por símbolos de V_T. El conjunto V_N contiene símbolos auxiliares para la definición de la gramática, y que no aparecen en las cadenas generadas. Es decir, se cumple que $V_N \cap V_T = \phi$. También definiremos al vocabulario como $V = V_N \cup V_T$.

Los elementos de P son las reglas que se aplican desde el símbolo inicial para obtener las cadenas del lenguaje.

La descripción de cada uno de los elementos de la gramática es:

- **Símbolos terminales:** son los elementos del alfabeto que no se pueden transformar, denotados comúnmente por letras minúsculas.
- **Variables o símbolos no terminales:** elementos auxiliares que imponen restricciones sintácticas al lenguaje. Se pueden transformar en otra cadena de variables y/o símbolos terminales. Se denotan con letras mayúsculas o de la forma `<regla>`.
- **Reglas:** permiten reemplazar variables para generar cadenas válidas. Puede haber más de una regla para una misma variable. Por ejemplo, $\alpha \to \beta \mid \gamma \mid \delta$ significa que tenemos tres reglas: $\alpha \to \beta$, $\alpha \to \gamma$, $\alpha \to \delta$; donde $\alpha, \beta, \gamma, \delta$ son cadenas de terminales y/o no terminales.

  Podemos ver estas reglas como *reglas de reemplazo*; es decir, si le aplicamos la regla $\alpha \to \beta$ a la cadena $a\alpha b$, obtenemos $a\beta b$.
- **Símbolo inicial:** es el símbolo a partir del cual se generan todas las cadenas válidas.

Como de costumbre, denotaremos por $L(G)$ al lenguaje generado por G, es decir, el conjunto de todas las cadenas que pueda generar G. Formalmente, $L(G) = \{\eta \in V_T^* \mid S \to \eta\}$.

Una cadena w pertenece a $L(G)$ ($w \in L(G)$) si está compuesta de símbolos terminales y puede derivarse a partir de S aplicando las reglas de producción de la gramática.

Diremos que la cadena w_1 deriva en w_2 en un paso ($w_1 \Rightarrow_G w_2$) si y solo si existen cadenas $x, y \in V^*$ tales que $w_1 = x\alpha y$, $w_2 = x\beta y$ y existe una regla $\alpha \to \beta \in P$. De esta forma, diremos que una cadena $w \in V^*$ es derivable a partir de G si y solo si existe una secuencia de derivación iniciando en S y terminando en la cadena w, es decir, $S = w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w$. Por sencillez, escribiremos $\alpha \Rightarrow \beta$ si α deriva a β en 0 o más pasos.

1.2. Gramática libre de contexto (GLC)

Ahora veamos las características de una gramática libre de contexto:

- Describen y generan los llamados *lenguajes libres de contexto*, que pueden ser reconocidos por un autómata de pila determinístico o no determinístico.
- Son útiles para describir bloques anidados en lenguajes de programación, ya que describen su sintaxis.
- Son libres de contexto porque, en cualquier regla, el elemento no terminal del lado derecho sin importar en qué contexto esté. Por lo tanto, todas las reglas son de la forma: $A \to \alpha$, donde $A \in V_N$ y $\alpha \in V^+$.

1.3. La forma normal de Chomsky

Una gramatica libre de contexto esta en la forma normal de Chomsky si todas las producciones son de la forma:

$$A \to a|BC$$

donde $A, B, C \in V$ y $a \in T$.

1.4. Existencia de equivalencia de la forma normal de Chomsky

Cualquier GLC $G = (V, T, S, P)$ con $\epsilon \notin L(G)$ tiene una gramatica equivalente $\hat{G}$ en la forma normal de Chomsky.

1.5. algoritmo CYK

Dada una gramatica libre de contexto en la forma normal de Chomsky G y una palabra $w = a_1a_2...a_n$ definimos $w_{ij} = a_i...a_j$ y unos subconjuntos de V

$$V_{ij} = \{A \in V : A^* w_{ij}\}$$

Claramente $w \in L(G) \iff S \in V_{1n}$.

Para obtener V_{ij} hacemos

$$V_{ij} = \bigcup_{k \in \{i, i+1, ..., j-1\}} \{A : A \to BC, \text{con } B \in V_{ik}, C \in V_{k+1,j}\}$$

2. Diseño de la solución

Para saber si la palabra es miembro de G procedemos a codificar la clase Grammar en la cual por medio de un *dict* y pasamos a la forma normal de Chomsky retiquetando las variables no terminales no sin antes haber removido las producciones unarias mediante un grafo, una vez hecho lo anterior pedimos construir la gramatica y una palabra para determinar su pertencia.

DirectedGraph.py

```
1 class DirectedGraph:
2     ''' G = (E, V) '''
3     def __init__(self):
4         self.adj = dict()
5 
6     '''
```

```
    def exists_node(node):
        def decorator(fun):
            def wrapper(*args, **kwargs):
                assert node in self.adj
                result = fun(*args, **kwargs)
                return result
            return wrapper
        return decorator
    '''

    def edges(self):
        conjunto = set()
        for a in self.adj:
            for b in self.adj[a]:
                conjunto.add((a, b))
        return conjunto

    def vertices(self):
        return self.adj.keys()

    def remove_edge(self, u, v):
        assert u in self.adj
        if v in self.adj[u]:
            self.adj[u].remove(v)

    def add_edge(self, u, v = None):
        if u not in self.adj:
            self.adj[u] = list()
        if v != None:
            if not v in self.adj:
                self.adj[v] = list()
            self.adj[u].append(v)

    def has_edge(self, u, v):
        assert u in self.adj
        return v in self.adj[u]

    def out_edges(self, u):
        assert u in self.adj
        return self.adj[u]

    def in_edges(self, u):
        edges = list()
        for key in self.adj:
            if u in self.adj[key]:
                edges.append(key)
        return edges

    def has_next(self, u):
        assert u in self.adj
```

```
57          return len(self.adj[u]) > 0
58
59      def loop(self, u):
60          assert u in self.adj
61          return u in self.adj[u]
62
63      def vertices_forward(self, u):
64          vertices = set()
65          visited = { x: False for x in self.adj }
66
67          def helper(v):
68              for value in self.adj[v]:
69                  vertices.add(value)
70                  if self.has_next(value) and not visited[value]:
71                      visited[value] = True
72                      helper(value)
73
74          assert u in self.adj
75          helper(u)
76          return vertices - set(u)
```

Grammar.py

```
1  from DirectedGraph import *
2  from collections import deque
3  import pdb
4
5  class Grammar:
6      """ G = (N, T, S, P) """
7
8      def __init__(self, nonterminals, terminals, start, productions):
9          """ nonterminals: set[string]
10             terminals: set[string]
11             start: string
12             productions: dict[string: set[string]] """
13         self.nonterminals = nonterminals
14         self.terminals = terminals
15         self.start = start
16         self.productions = productions
17
18         self.lang_prod = lambda a, b: { x  + y for x in a for y in b }
19         self.fcg_to_cnf()
20
21     def remove_unit_productions(self):
22         ''' Removemos producciones unarias mediante un grafo '''
23         g = DirectedGraph()
24         productions = self.productions.copy()
25         for key in productions: # construimos el grafo
26             for rule in productions[key]:
```

```
27                  if len(rule) == 1 and rule in self.nonterminals:
28                      g.add_edge(key, rule)
29          followings = { v: g.vertices_forward(v) for v in g.vertices() }
30          not_unit = dict()
31          not_unit_fun = lambda s: len(s) != 1 or s in self.terminals
32          for vertex in g.vertices():
33              not_unit[vertex] = set(filter(not_unit_fun, productions[vertex]))
34          for e0, e1 in g.edges():
35              productions[e0].remove(e1)
36              for node in followings[e0]:
37                  productions[e0] = productions[e0] | not_unit[node]
38          return productions
39
40      def add_lists(a, b = None):
41          ''' Retorna una lista concatenada, parecida al operador ++ de Scala o Haskell
            ↪  '''
42          vector = list(); vector.append(a); vector.append(b)
43          return tuple(vector)
44
45      def __reduce(self, collection):
46          ''' Funcion de ayuda para fcg_to_cnf. Aqui, nos aseguramos de que las tuplas
47              tengan una longitud menor o igual a 2. '''
48          def helper(queue, string):
49              n = len(string)
50              if n > 2:
51                  queue.append(tuple(string[0]) + index_fun())
52                  helper(queue, string[1 : n])
53              else:
54                  queue.append(string)
55
56          def index_fun():
57              ''' Simula un subindice a la letra B:  B_x -> x '''
58              indexed_char = 'D_{%d}' % self.k;
59              self.k += 1
60              return indexed_char,
61
62          # abcdef => aD1, bD2, cD3, dD4, ef
63          # ab => ab
64          # a => a
65          new_rules = dict() # aqui se guardaran las reglas D_{n} -> R
66          original_rules_modified = set() # modificaremos las reglas originales
67          for str_enum in collection:
68              if len(str_enum) <= 2: # ya cumple con la condicion
69                  original_rules_modified.add(str_enum)
70              else:
71                  queue = deque()
72                  helper(queue, str_enum)
73                  original_rules_modified.add(queue[0])
74                  while len(queue) > 1: # para no ir mas alla de la etiquetacion
75                      element = queue.popleft()
```

```
76                      new_rules[element[1]] = set()
77                      new_rules[element[1]].add(queue[0])
78          return original_rules_modified, new_rules
79
80      def __print_dict(self, cnf):
81          """ Esta funcion no es necesaria para el algoritmo, se uso para depurar el
            ↪  codigo """
82          for key in cnf:
83              print('{}: '.format(key))
84              for value in cnf[key]:
85                  print(value)
86              print()
87
88      def __replace_set_value(self, conjunto, old, new):
89          conjunto.remove(old)
90          conjunto.add(new)
91
92      def fcg_to_cnf(self):
93          self.cnf = self.remove_unit_productions()
94          cnf = self.cnf # para no escribir self demasiado
95          new_rules = dict() # diccionario de apoyo para guardar nuevas reglas
96          self.k = 1 # variable usada en __reduce
97          for key in cnf: # "enumeramos"
98              cnf[key] = set(map(tuple, cnf[key]))
99          for key in cnf: # hacemos que cada produccion w sea tal que |w| <= 2
100             cnf[key], temp_dict = self.__reduce(cnf[key])
101             new_rules = { **new_rules, **temp_dict } # mezclamos diccionarios (py >=
                ↪  3.5)
102         cnf = { **cnf, **new_rules }
103         for key in cnf: # creamos nuevas reglas tales que B_x -> x, x en T
104             for enum_str in cnf[key]:
105                 if len(enum_str) == 2:
106                     new_tuple = 2 * [None] # lista de longitud 2
107                     for i in range(2):
108                         char = enum_str[i]
109                         if char in self.terminals:
110                             new_rules['B_{}'.format(char)] = set()
111                             new_rules['B_{}'.format(char)].add(tuple(char))
112                             new_tuple[i] = 'B_{}'.format(char)
113                         else:
114                             new_tuple[i] = char
115                     self.__replace_set_value(cnf[key], enum_str, tuple(new_tuple))
116         self.cnf = { **cnf, **new_rules }
117         #self.__print_dict(cnf)
118         for key in self.cnf: # Removemos la enumeracion
119             for value in self.cnf[key]:
120                 self.__replace_set_value(self.cnf[key], value, ''.join(value))
121
122     def remove_useless_productions(self):
123         pass
```

```
124
125     def validate(self, w):
126         """ validate(w: string) -> bool
127             Determina si una palabra pertenece a la gramatica mediante el algoritmo
    ↪  CYK.
128             Funciona ssi la gramatica dada esta en la forma normal de Chomsky. """
129
130         def cyk(i, j):
131             ''' Crea una tabla de manera recursiva donde almacena las producciones que
                ↪
132                 generen las subpalabras de la cadena dada, se basa en el principio
133                 divide y venceras. '''
134             if j == 1: # Aqui verificamos las variables no terminales que derivan a T
135                 v[i, j] = set()
136                 for key in self.cnf:
137                     if w[i] in self.cnf[key]:
138                         v[i, j].add(key)
139             else: # Hallamos las variables no terminales que derivan a las subcadenas
140                 substring = w[i : i + j]
141                 sets = set()
142                 for x in range(1, j):
143                     left, right = divide(substring, x)
144                     l = len(left)
145                     r = len(right)
146                     if not ((i, l) in v): # Si no existen valores en la tabla
147                         cyk(i, l)
148                     if not ((i + x, r) in v):
149                         cyk(i + x, r)
150                     # Obtenemos las variables no terminales validas
151                     conjunto = rules(lang_prod(v[i, l], v[i + x, r]))
152                     if len(conjunto) > 0: # si es no vacio
153                         sets = sets.union(conjunto)
154                 v[i, j] = sets # Asignamos la union de todas las variables no
                    ↪  terminales
155                 # validas
156
157         def rules(cartprod):
158             ''' rules(cartprod: set[string]) -> set
159                 Dado un producto de lenguajes, retornamos las reglas que derivan al
160                 mismo. '''
161             nonterminals = set()
162             for value in cartprod:
163                 for key in self.cnf:
164                     if value in self.cnf[key]:
165                         nonterminals.add(key)
166             return nonterminals
167
168         def divide(w, i):
169             ''' Retorna una bina de la palabra w dividida en dos.
170                 La variable i debe estar en range(1, |w|) para efectos del algoritmo.
```

```
171                 >>> divide('pongame diez', 3)
172                 (pon, game diez) '''
173             return w[:i], w[i: len(w)]
174 
175         v = dict() # Variable que guarda los valores de la tabla
176         n = len(w)
177         #pdb.set_trace() # breakpoint para el depurador de pdb
178         cyk(0, n) # Importante obtener v[(0, n)]
179         self.__print_dict(v)
180         return self.start in v[0, n]
181 
182 def lang_prod(x, y):
183     """ lang_prod(x: set[string], y: set[string]) -> set[string]
184         Retorna el producto de dos lenguajes.  """
185     return { a + b for a in x for b in y }
```

main.py

```
1  from Grammar import *
2 
3  __author__ = "Angel Lopez Manriquez"
4 
5  def ask_productions():
6      """ Obtiene una gramatica por teclado. """
7      print("\n Programa que determina si una palabra pertenece o no a una GLC.\n")
8      nonterminals = set(input("\nIngrese las variables no terminales, separadas por ,:
       ↪  ").
9          replace(" ", "").split(','))
10     terminals = set(input("Ingrese las variables terminales, separadas por ,: ").
11         replace(" ", "").split(','))
12     start = input("Ingrese la variable inicial: ")
13     productions = dict()
14     print("Ingrese las productiones separadas por | : ")
15     for value in nonterminals:
16         productions[value] = set(input("{} --> ".format(value)).replace(" ",
           ↪  "").split('|'))
17     g = Grammar (nonterminals, terminals, start, productions)
18     want_to_continue = 'y'
19     while want_to_continue == 'y':
20         print()
21         word = input("Ingrese una palabra ")
22         if g.validate(word):
23             print("La palabra %s pertenece a L(G) :D " % word)
24         else:
25             print("La palabra %s no pertenece a L(G) D: " % word)
26         want_to_continue = input("Desea continuar? (y/n): ")
27     print("\nHasta luego. o-o// ")
28 
29 # G = (V, T, S, P)
```

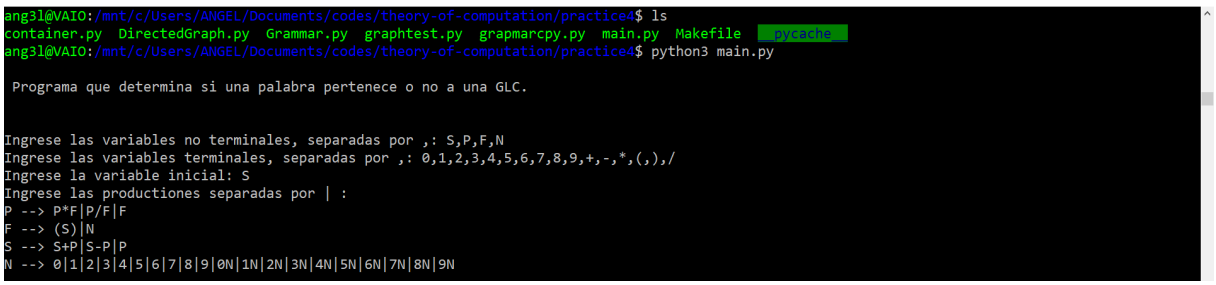
```
30 def test1():
31     g = Grammar (
32         { 'S', 'A', 'B' }, # V
33         { 'a', 'b' }, # T
34         'S', # S
35         { 'S': { 'AB' }, # P
36         'A': { 'BB', 'a' },
37         'B': { 'AB', 'b' } }
38     )
39     #w = 'aabbb'
40     w = 'aab'
41     print(g.validate(w))
42 
43 def test2():
44     g = Grammar (
45         { 'S', 'A', 'B', 'C' }, # V
46         { 'a', 'b' }, # T
47         'S', # S
48         { 'S': { 'AB', 'BC' }, # P
49         'A': { 'BA', 'a' },
50         'B': { 'CC', 'b' },
51         'C': { 'AB', 'a' } }
52     )
53     w = 'baaba'
54     print(g.validate(w))
55     g.fcgtocnf()
56 
57 def tocnftest():
58     g = Grammar (
59         { 'S', 'A', 'B' }, # V
60         { 'a', 'b', 'c' }, # T
61         'S', # S
62         { # P
63             'S': { 'ABa' },
64             'A': { 'aab' },
65             'B': { 'Ac' }
66         }
67     )
68     g.fcg_to_cnf()
69     #print(g.validate('abc'))
70 
71 def delunittest():
72     g = Grammar (
73         { 'S', 'A', 'B' }, # V
74         { 'a', 'b', 'c' }, # T
75         'S', # S
76         { # P
77             'S': { 'Aa', 'B' },
78             'A': { 'a', 'bc', 'B' },
79             'B': { 'A', 'bb' }
```

```
80          }
81      )
82
83  def finaltest():
84      g = Grammar (
85          { 'S', 'P', 'F', 'N' }, # V
86          { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '-', '*', '(', ')',
            ↪  '/' }, # T
87          'S', # S
88          { # P
89              'S': { 'S+P', 'S-P', 'P' },
90              'P': { 'P*F', 'P/F', 'F' },
91              'F': { '(S)', 'N' },
92              'N': { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0N', '1N', '2N',
                ↪  '3N', '4N', '5N', '6N', '7N', '8N', '9N' }
93          }
94      )
95      print(g.validate('1+(2*3-4)'))
96      print(g.validate('(12-7/(4+1))*8-7+(5-21)'))
97      print(g.validate('5*(4+8'))
98      print(g.validate('10+8-'))
99
100 ask_productions()
```

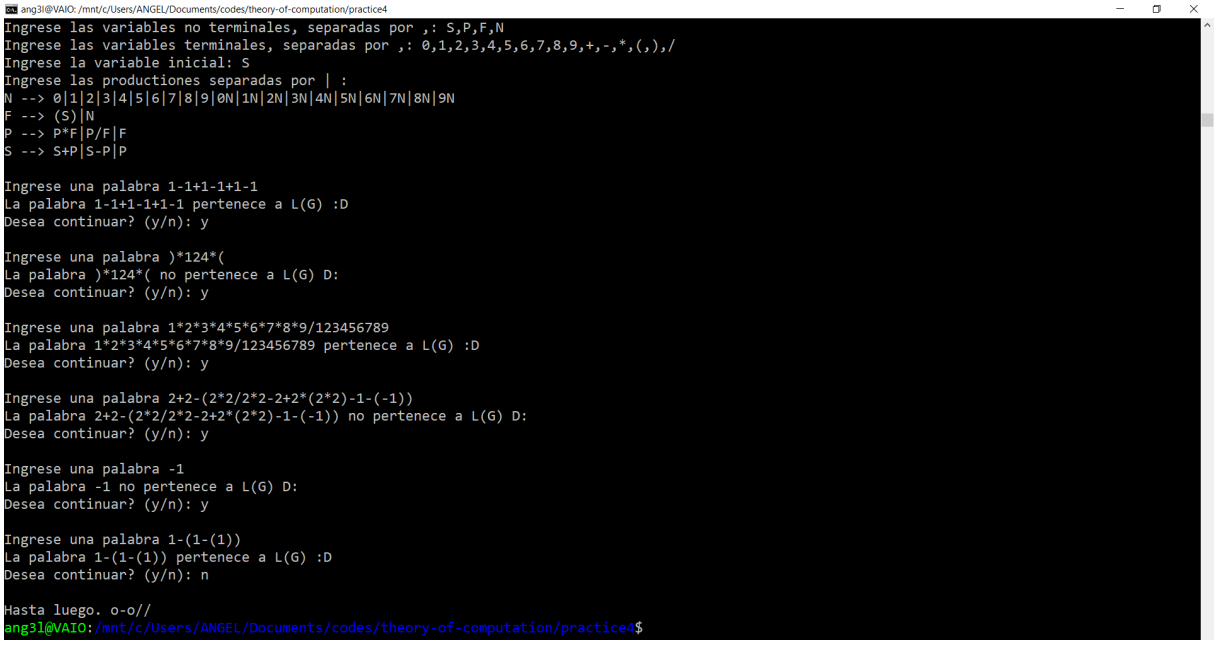
tambien se incluyen algunos test para que simplemente los corra en este archivo.

3. Funcionamiento

Procedamos a meter una GLC



vemos que al mostrar V_{ij} nos llena la pantalla con solo una palabra de longitud 3, nos limitaremos a obviar la impresion de V comentando la linea 118 de Grammar.py y asi hacer varios test, ud. puede la impresion V en cualquier momento descomentando este metodo privado por su cuenta si asi lo desea asi como cualquier GLC.



4. Conclusiones

En esta practica aprecie bastante las funciones de manejos de cadenas que nos provee python, asi como las variables de instancia de la clase list para poder enumerar las producciones de las cadenas, asi como una mejor comprension del uso del concepto de recursividad para poder resolver problemas de computo como el uso de gramaticas nos pueden ayudar para determinar estructuras especificas de una cadena, con mucho mas trabajo que con la biblioteca re.