```
    1: """
    2: Clase que asiste la implementacion de las reglas semanticas
    3: en la traduccion dirigida por la sintaxis. Cuya finalidad es obtener un NFA
    4: """
    5: class NFA_output:
    6:     #atributo de clase que mantiene el conteo de estados
    7:     iestado = 0
    8:     lista_arcos_NFA = []
    9:
   10:
   11:     """metodo que retorna una tupla que representa un arco"""
   12:     def gen_arco(self, terminal):
   13:         inicio = self.inc_iestado()
   14:         fin = self.inc_iestado()
   15:
   16:         return  [(inicio, fin, terminal)]
   17:
   18:     """metodo que retorna una nueva lista de arcos,segun la definicion
   19:           de la cerradura de klenee
   20:     """
   21:     def gen_kleene(self, lista_arcos):
   22:         inicio = self.inc_iestado()
   23:         arcoinicial = (inicio , self.get_eini(lista_arcos), "Ep")
   24:         fin = self.inc_iestado()
   25:         arcoalvacio = (inicio, fin, "Ep")
   26:
   27:         arcociclo = (self.get_efin(lista_arcos), self.get_eini(lista_arcos), "Ep")
   28:
   29:         arcofinal = (self.get_efin(lista_arcos), fin, "Ep")
   30:
   31:         listanueva = [arcoinicial] + [arcoalvacio] + lista_arcos + [arcociclo] + [arcofinal]
   32:
   33:         return listanueva
   34:
   35:     """metodo que retorna una nueva lista de arcos,segun la definicion
   36:         de la cerradura positiva de klenee
   37:     """
   38:     def gen_kleene_positivo(self, lista_arcos):
   39:         inicio = self.inc_iestado()
   40:         arcoinicial = (inicio , self.get_eini(lista_arcos), "Ep")
   41:         fin = self.inc_iestado()
   42:         arcociclo = (self.get_efin(lista_arcos), self.get_eini(lista_arcos), "Ep")
   43:         arcofinal = (self.get_efin(lista_arcos), fin, "Ep")
   44:         listanueva = [arcoinicial] + lista_arcos + [arcociclo]  +  [arcofinal]
   45:         return listanueva
   46:
   47:     """metodo que retorna una nueva lista de arcos segun la definicion
   48:        de la concatenacion
   49:     """
```

```
50:     def gen_concat(self,lista_arcos1, lista_arcos2):
51:         arconuevo = (self.get_efin(lista_arcos1),self.get_eini(lista_arcos2), "Ep")
52:         listanueva =lista_arcos1 + [arconuevo] + lista_arcos2
53:         return listanueva
54:
55:     """metodo que retorna una nueva lista de arcos segun la definicion de la
56:        decision
57:     """
58:     def gen_decision(self, lista_arcos1, lista_arcos2):
59:         inicio = self.inc_iestado()
60:         fin = self.inc_iestado()
61:         arco_inicio_sup = ( inicio , self.get_eini(lista_arcos1) , "Ep" )
62:         arco_inicio_inf = ( inicio , self.get_eini(lista_arcos2) , "Ep" )
63:         arco_fin_sup = (self.get_efin(lista_arcos1), fin , "Ep")
64:         arco_fin_inf = (self.get_efin(lista_arcos2), fin , "Ep")
65:         return [arco_inicio_sup] + [arco_inicio_inf] + lista_arcos1 + lista_arcos2 + [arco_fin_sup] + [arco_fin_inf]
66:
67:     """metodo que retorna una nueva lista de arcos segun la definicion de
68:         cero o una repeticion
69:     """
70:     def gen_cero_o_uno(self,lista_arcos):
71:         inicio = self.inc_iestado()
72:         fin = self.inc_iestado()
73:         arcoinicial = (inicio, self.get_eini(lista_arcos), "Ep")
74:         arcofinal = (fin, self.get_efin(lista_arcos), "Ep")
75:         arco_cero = (inicio, fin, "Ep")
76:         return [arcoinicial] + [arco_cero] + lista_arcos + [arcofinal]
77:
78:     """retorna el estado inicial de una lista de arcos"""
79:     def get_eini(self, lista_arcos):
80:         if len(lista_arcos) > 0:
81:             return lista_arcos[0][0]
82:
83:     """retorna el estado final de una lista de arcos"""
84:     def get_efin(self, lista_arcos):
85:         longitud = len(lista_arcos)
86:         if longitud > 0:
87:             return lista_arcos[longitud - 1][1]
88:
89:     """retorna el nuevo numero de estado que debe crearse """
90:     def inc_iestado(self):
91:         self.iestado = self.iestado + 1
92:         return self.iestado
```

```
from config import DIR_SALIDA_CODIGO,PREFIJO_ARCHIVO_GEN
import os.path
class Codigo_output():
    def __init__(self,d_dfa):
        self.estado_inicial = d_dfa["estado_inicial"]
        self.conjunto_aceptacion = d_dfa["estados_aceptacion"]
        self.__lsarcos = d_dfa['lista_arcos']
        self.__identacion = 4
        __nombre_archivo = ""

    def codigo_out(self,d_estados):
        return ""

    def gen_inicio(self):
        out = "global cadena" + "\n"
        out += "global aceptado" + "\n"
        out += "global pertenece " + "\n"
        out += "cadena = raw_input('evaluar>')" + "\n"
        out += "lector = avanzar_entrada()" + "\n"
        out += "c_entrada = lector.next()" + "\n"
        out += "estado=" + self.__encerrar(str(self.estado_inicial)) + "\n"
        out += "ent = False\n"
        out += "error = False\n"


        #if self.estado_inicial in self.conjunto_aceptacion:
        #    out = out + "aceptado = True" + "\n"
        #else:
        #    out = out + "aceptado = False " + "\n"

        out += "while not estado in " + str(self.conjunto_aceptacion) + " or c_entrada != None"  + ":\n"
        return out

    def gen_func_avanzar(self):
        out = "def avanzar_entrada():" + "\n" \
           "    lmax = len(cadena)" + "\n" \
           "    cont = -1" + "\n" \
           "    while cont <= lmax:" + "\n" \
           "        cont = cont + 1" + "\n"\
           "        if cont == lmax:" + "\n"\
           "            yield None" + "\n"\
           "        yield cadena[cont]" +  "\n"
        return out

    def gen_bloque(self,arco):
        nivel = self.__identacion * ' '
        out = nivel + "if estado == " + self.__encerrar(str(arco[0])) + ":\n"
        out += nivel + "    if c_entrada==" + self.__encerrar(str(arco[2])) + ":\n"
        out += nivel + "        estado = " + self.__encerrar(str(arco[1])) + "\n"
```

```
50:         out += nivel + "        c_entrada = lector.next()" + "\n"
51:         out += nivel + "        ent = True\n"
52:         out += nivel + "        continue\n"
53:         return out
54:
55:     def __gen_evaluacion_final(self):
56:         out = "if estado in " + str(self.conjunto_aceptacion) + " and not error :"
57:         out += "\n    print 'correcto!'"
58:         out += "\nelse:"
59:         out += "\n    print 'Error!'"
60:         out += "\nraw_input('')"
61:         return out
62:
63:     def gen_main(self):
64:         self.__crear_archivo()
65:         self.__persistir(self.gen_func_avanzar())
66:         self.__persistir(self.gen_inicio())
67:         for arco in self.__lsarcos:
68:             self.__persistir(self.gen_bloque(arco))
69:
70:         self.__persistir(self.__final_while())
71:
72:         self.__persistir(self.__gen_evaluacion_final())
73:
74:     def __persistir(self,cadena):
75:         archivo = open(self.__nombre_archivo,'a')
76:         archivo.write(cadena)
77:         archivo.close()
78:
79:     """crea un archivo en un directorio especificado , pero sin sobre escribir los
80:     generados anteriormente
81:     """
82:     def __crear_archivo(self):
83:         if not os.path.exists(DIR_SALIDA_CODIGO):
84:             os.mkdir(DIR_SALIDA_CODIGO)
85:
86:         self.__nombre_archivo = DIR_SALIDA_CODIGO + os.sep + PREFIJO_ARCHIVO_GEN
87:         archivo = open(self.__nombre_archivo ,'w')
88:         archivo.close()
89:
90:     def __encerrar(self,cadena):
91:         return "'" + cadena + "'"
92:
93:     def __final_while(self):
94:         out = "    if not ent:\n        error = True\n        break\n"
95:         out += "    ent = False\n"
96:         return out
```

```
    1: #la idea es obtener un solo html con todo el resultado de la salida
    2: import os
    3: import webbrowser
    4: from config import DIR_SALIDA_REPORTE , NAM_ARCHIVO_REPORTE
    5: from emit_grafo import dibujar
    6: #from emit_grafo import dibujar
    7:
    8: class ReporteHTML:
    9:     titulo=""
   10:     subtitulo=""
   11:     file = None
   12:     dicc_nfa = None
   13:     dicc_dfa = None
   14:     dicc_nfamin = None
   15:     dir_salida = DIR_SALIDA_REPORTE
   16:     nombre_archivo= NAM_ARCHIVO_REPORTE
   17:
   18:
   19:     def __init__(self, titulo, subtitulo, dicc_nfa, dicc_dfa, dicc_nfamin ):
   20:         self.titulo = titulo
   21:         self.subtitulo = subtitulo
   22:         self.dicc_nfa = dicc_nfa
   23:         self.dicc_dfa = dicc_dfa
   24:         self.dicc_nfamin = dicc_nfamin
   25:
   26:     def abrir_archivo(self):
   27:         if not os.path.exists(self.dir_salida):
   28:             os.mkdir(self.dir_salida)
   29:         self.file = open(self.dir_salida + os.path.sep + self.nombre_archivo, "w")
   30:
   31:     def cerrar_archivo(self):
   32:         self.file.close()
   33:
   34:     def ht_html(self, cerrar=True):
   35:         if cerrar:
   36:             return "</html>"
   37:         else:
   38:             return "<html>"
   39:
   40:     def ht_hn(self, valor,  texto):
   41:         return "<h"+str(valor) + ">" + texto + "<h"+str(valor) + "/>"
   42:
   43:
   44:     def secc_documento(self):
   45:         self.abrir_archivo()
   46:         #inicio
   47:         self.file.write(self.ht_html())
   48:         self.file.write("<head>")
   49:
```

```
50:         self.file.write(self.css_segmento())
51:
52:         self.file.write("</head>")
53:
54:         self.file.write("<body>")
55:
56:         #titulos
57:         self.file.write(self.ht_hn(1,self.titulo))
58:         self.file.write(self.ht_hn(1,self.subtitulo))
59:
60:         #secciones
61:         self.file.write(self.secc_paso("NFA",self.dicc_nfa ))
62:         self.file.write(self.secc_paso("DFA",self.dicc_dfa))
63:         self.file.write(self.secc_paso("DFA Min",self.dicc_nfamin))
64:         self.file.write(self.__img_dibujografo())
65:         #fin
66:         self.file.write("</body>")
67:         self.file.write(self.ht_html(True))
68:
69:
70:         self.cerrar_archivo()
71:
72:     def ht_tabla(self, ls_arcos):
73:         lista_estados=[]
74:         tabla = {}
75:         for arco in ls_arcos:
76:             einicial = arco[0]
77:             efinal = arco[1]
78:             terminal = arco[2]
79:             #
80:             if not terminal in tabla :
81:                 tabla[terminal]={}
82:             #
83:             if einicial not in tabla[terminal]:
84:                 tabla[terminal][einicial]=[]
85:             #
86:             if efinal not in tabla[terminal]:
87:                 tabla[terminal][efinal]=[]
88:             #
89:             tabla[terminal][einicial].append(efinal)
90:
91:             if not einicial in lista_estados:
92:                 lista_estados.append(einicial)
93:
94:             if not efinal in lista_estados:
95:                 lista_estados.append(efinal)
96:
97:         terminal = ''
98:         fila_cab = '<th></th>'
```

```
 99:         fila_est = ''
100:         #generalos la columnas
101:         for terminal in tabla :
102:             fila_cab += '<th>'
103:             fila_cab += terminal
104:             fila_cab += '</th>'
105:
106:         for iestado in lista_estados:
107:             fila_est += "<tr>\n"
108:             fila_est += "\t" + '<td>' + str(iestado) + '</td>' + "\n"
109:             for terminal in tabla :
110:                 if iestado in tabla[terminal] and len(tabla[terminal][iestado]) > 0:
111:                     fila_est += "\t" + '<td>' +  str(tabla[terminal][iestado]) + '</td>' + "\n"
112:                 else:
113:                     fila_est += "\t" + '<td>' +  " - " + '</td>' + "\n"
114:
115:             fila_est += "</tr>\n"
116:         salida = "<table>\n" + "<caption>Tabla de transiciones</caption>" + \
117:         "<tr>" + fila_cab + "</tr>"  + fila_est  + "</table>"
118:         return salida
119:
120:     def secc_paso(self,tituloseccion, dicc_paso):
121:
122:         salida = self.ht_hn(2, tituloseccion)
123:         salida += self.ht_hn(3, "Estado de inicial:" + str(dicc_paso['estado_inicial']))
124:         salida += self.ht_hn(3, "Estados de aceptacion:" + str(dicc_paso['estados_aceptacion']))
125:         if 'renombrados' in dicc_paso:
126:             for equiv in dicc_paso['renombrados']:
127:                 salida += "<p>" + dicc_paso['renombrados'][equiv] + "=" + str(equiv) + "</p>"
128:
129:         salida += self.ht_tabla(dicc_paso['lista_arcos'])
130:         salida += "\n<hr>\n"
131:         return salida
132:
133:     def css_segmento(self):
134:         return """
135:         <style>
136:         body {font-family:"Lucida Console";margin-left:5%; color : rgb(40,40,40); }
137:          table, th, td {border: 1px solid grey;  padding:3px;}
138:          </style>
139:          """
140:     def abrir_reporte(self):
141:         webbrowser.open_new(self.dir_salida + os.path.sep + self.nombre_archivo)
142:
143:     def __img_dibujografo(self):
144:         dibujar(self.dicc_nfamin['lista_arcos'],self.dicc_nfamin['estado_inicial'],
145:                 self.dicc_nfamin['estados_aceptacion'],
146:                  self.dir_salida + os.path.sep + "grafo")
147:         return "\n" + '<p>'+ self.subtitulo +'<img src="grafo.png"/> </p>' +"\n"
```

```
1:
2: DIR_SALIDA_CODIGO = 'codigo_gen'
3: PREFIJO_ARCHIVO_GEN='gen_cod.py'
4: #reporte
5: DIR_SALIDA_REPORTE = 'reporte'
6: NAM_ARCHIVO_REPORTE = 'reporte.html'
7:
8:
```

```
"""
Clase que reconoce algun lenguaje generado por la sigte gramatica
expr ::= concat '|' concat
         concat
concat ::= rep '.' rep
        | rep
rep ::= atom '*' | atom '+' | atom '?'
        | atom
atom ::= '(' expr ')'
       | char
char ::= a..z

"""

from anlex import AnLex, Lexconst

class AnSintactico:
    pre_analisis = (None, None)
    #Referencia a una instancia del analizador lexico
    anlex = None
    #referencia a una instancia del traductor NFA
    emisor_nfa = None
    #lista de arcos obtenido al final del analisis
    lista_arcos_nfa = None

    """La intanciacion del analizador sintactico requiere la instancia de:
    un analizador lexico
    un emisorNFA
    """
    def __init__(self, anlex, nfa_instance):
        self.anlex = anlex
        self.emisor_nfa = nfa_instance

    """metodo que avanza un caracter si el lexema es el esperado"""
    def parea(self, lexema):
        if self.pre_analisis["lexema"] == lexema:
            self.pre_analisis = self.anlex.next_token()
            if self.pre_analisis["tipo"] != Lexconst.EOF:
                print (("token consumido: '" + self.pre_analisis["lexema"] + "'"))
        else:
            raise Exception("Error sintactico, se esperaba: '" + lexema)

    """metodo principal que inicia el analisis sintactico"""
    def analizar(self):
        self.pre_analisis = self.anlex.next_token()
        #while self.pre_analisis['tipo'] != Lexconst.EOF:
        lista_arcos = self.expr()

        self.lista_arcos_nfa = lista_arcos
```

```
50:         if self.pre_analisis['tipo'] != Lexconst.EOF:
51:             raise Exception("Hubo algun Error,no se ha consumido toda la cadena")
52:
53:     """Valida la sintaxis de un no terminal atomico y construye
54:     su lista de arcos de arcos
55:     """
56:     def atomico(self):
57:         print (("atomico(),preanalisis='" + self.pre_analisis["lexema"] + "'"))
58:         if self.pre_analisis["tipo"] == Lexconst.SIMB_DEF:
59:             lexemaant = self.pre_analisis["lexema"]
60:             self.parea(self.pre_analisis["lexema"])
61:             #retorna el par de estados (arco) que representa al lexema
62:             return self.emisor_nfa.gen_arco(lexemaant)
63:
64:         if self.pre_analisis["lexema"] == '(':
65:             self.parea('(')
66:             #obtiene la lista de arcos construida por los metodos llamados en jerarquia
67:             lista_nodos = self.expr()
68:             self.parea(')')
69:             #retorna la lista de nodos que le toco construir
70:             return lista_nodos
71:         raise Exception("Error al formar atomico")
72:
73:     """Valida la sintaxis de un no terminal repeticion y construye
74:     su lista de arcos de arcos
75:     """
76:     def repeticion(self):
77:         print (("repeticion(),preanalisis='" + self.pre_analisis["lexema"] + "'"))
78:         nodo_hijo = self.atomico()
79:         if self.pre_analisis["lexema"] == '*':
80:             self.parea('*')
81:             #genera la lista de arcos para la cerradura
82:             nodos_res = self.emisor_nfa.gen_kleene(nodo_hijo)
83:             return nodos_res
84:
85:         elif self.pre_analisis["lexema"] == '+':
86:             self.parea('+')
87:             #genera la lista de arcos para la cerradura positiva
88:             nodos_res = self.emisor_nfa.gen_kleene_positivo(nodo_hijo)
89:             return nodos_res
90:         elif self.pre_analisis["lexema"] == '?':
91:             self.parea('?')
92:             #genera la lista de arcos para la regla uno o cero
93:             nodos_res = self.emisor_nfa.gen_cero_o_uno(nodo_hijo)
94:             return nodos_res
95:         return nodo_hijo
96:
97:     def concat(self):
98:         print (("concat(),preanalisis='" + self.pre_analisis["lexema"] + "'"))
```

```
   99:         listanueva = self.repeticion()
  100:         while True :
  101:             if self.pre_analisis["lexema"] == '.':
  102:                 self.parea('.')
  103:                 nuevo_nodo = self.repeticion()
  104:                 #concatena sucesivamente los resultados de generar listas de arcos
  105:                     #por medio de llamadas a repeticion()
  106:                 listanueva = self.emisor_nfa.gen_concat(listanueva, nuevo_nodo)
  107:                 continue
  108:             else:
  109:                 return listanueva
  110:
  111:     def expr(self):
  112:         print (("expr(),preanalisis='" + self.pre_analisis["lexema"] + "'"))
  113:         listanueva = self.concat()
  114:
  115:         while True:
  116:             if self.pre_analisis["lexema"] == '|':
  117:                 self.parea('|')
  118:                 lista_nodoshijos = self.concat()
  119:
  120:                 listanueva = self.emisor_nfa.gen_decision(listanueva, lista_nodoshijos)
  121:                 continue
  122:             else:
  123:                 return listanueva
```

```
 1: class Modelogui:
 2:     lista_temp_simbolos =[]
 3:     lista_alfabetos_creados={}
 4:
 5:     def nuevo_simbolo(self, simb):
 6:         self.lista_temp_simbolos.append(str(simb))
 7:
 8:     def nuevo_alfabeto(self, nombre):
 9:         self.lista_alfabetos_creados[nombre] = self.lista_temp_simbolos
10:         self.lista_temp_simbolos=[]
11:
12:     def get_ultimo_agregado(self):
13:         tam= len(self.lista_alfabetos_creados)
14:         return str  + str(self.lista_alfabetos_creados[tam-1])
```

```
    1: import pygraphviz as pgv
    2:
    3: def dibujar(lista_arcos,einicial,eaceptacion,archivo):
    4:     A=pgv.AGraph(strict=False,directed=True)
    5:     A.node_attr['shape']='circle'
    6:     A.node_attr['fixedsize']='true'
    7:     A.node_attr['height']=0.5
    8:     A.node_attr['width']=0.5
    9:     for arco in lista_arcos:
   10:         A.add_edge(arco[0],arco[1])
   11:
   12:         n = A.get_edge(arco[0],arco[1])
   13:         n.attr['label']=arco[2]
   14:         node = A.get_node(arco[1])
   15:
   16:         if arco[0] == einicial:
   17:             node.attr['color'] = "#565050"
   18:             node.attr['style']='setlinewidth(2)'
   19:
   20:         if arco[1] in  eaceptacion:
   21:             node.attr['color'] = "#514e86"
   22:             node.attr['style']='setlinewidth(3)'
   23:
   24:     A.write(archivo + '.dot')
   25:     B=pgv.AGraph(archivo + '.dot')
   26:     B.layout()
   27:     B.draw(archivo + ".png")
```

```
 1: from anlex import AnLex
 2: from emit_NFA import NFA_output
 3: from emit_DFA import DFA_output
 4: from gen_codigo import Codigo_output
 5: from reporte import ReporteHTML
 6: from ansintactico import AnSintactico
 7:
 8: def principal(adm_simbolos, def_regular):
 9:
10:     expresion_reg = def_regular
11:     # se instancia un administrador de simbolos
12:     simbolos_admin = adm_simbolos
13:
14:     analizador_lex = AnLex(simbolos_admin)
15:     # instancia el flujo a analizar
16:     analizador_lex.set_flujo(expresion_reg)
17:
18:     # instancia un traductor NFA requerido por el analizador sintactico
19:     traductor_nfa = NFA_output()
20:
21:     analizador_sintac = AnSintactico(analizador_lex , traductor_nfa)
22:     analizador_sintac.analizar()
23:
24:     # instancia un traductor DFA con la lista de arcos generado anteriormente
25:     traductor_DFA = DFA_output(analizador_sintac.lista_arcos_nfa)
26:     #
27:     traductor_DFA.afn_to_afd()
28:
29:     # renombra estados del conjuntoDFA
30:     traductor_DFA.renombrar_nfa()
31:
32:     # minimiza estados del NFA
33:     traductor_DFA.minimizar(traductor_DFA.lista_DFA_renam)
34:
35:     generadorCodigo = Codigo_output(traductor_DFA.get_dfa_minimo())
36:     generadorCodigo.gen_main()
37:     reporteHTML = ReporteHTML("Resultados",
38:                               def_regular,
39:                               traductor_DFA.get_nfa_inst(),
40:                               traductor_DFA.get_dfa_renombrado(),
41:                               traductor_DFA.get_dfa_minimo())
42:     reporteHTML.secc_documento()
43:     reporteHTML.abrir_reporte()
44:
```

```
"""Clase que contiene las estructuras de datos y metodos
   necesarios para convertir un NFA a un DFA

"""
class DFA_output:
    lista_nfa = []
    lista_DFA =[]
    lista_DFA_renam = []
    lista_DFA_min=[]
    #estado inicial del nfa
    est_nfa_inicial= 0
    #estado de aceptacion del nfa
    est_nfa_aceptacion =  0
    #conjunto de estados de aceptacion
        #y estado inicial , posterior al renombramiento del DFA
    ls_renam_aceptacion = []
    est_renam_inicial = 0
    #para guardar las equivalencias del renombramiento
    mp_erenombrados = {}
    #conjunto de estados de aceptacion y estado inicial pos minimizacion
    ls_min_aceptacion=[]
    est_min_inicial = 0

    """El constructor recibe como parametro una lista de tuplas que representa al NFA"""
    def __init__(self,lista_arcos_nfa):
        self.lista_nfa = lista_arcos_nfa
        #obtenemos estado final
        self.est_nfa_inicial = self.lista_nfa[0][0]
        #obtenemos  el estado de aceptacion
        longitud = len(self.lista_nfa)
        if longitud > 0:
            self.est_nfa_aceptacion = self.lista_nfa[longitud - 1][1]

    """Metodo que retorna una lista los estados alcanzables
    desde un estado  y un caracter recibidos como parametros
    """
    def estados_alcanz(self, estado, tr ,ls_arcos = None):
        estados_alcanzables=[]
        if ls_arcos is None:
            ls_arcos = self.lista_nfa
        for arco in ls_arcos:
            if arco[0] == estado and arco[2] == tr:
                estados_alcanzables.append(arco[1])
        return estados_alcanzables

    """Metodo que retorna una lista los estados alcanzanbles directamente,
    por transiciones epsilon desde un un cojunto de estados
    """
    def trans_directa_E(self,conjunto):
```

```
50:         ealcanzables = []
51:         for estado in conjunto:
52:             estados = self.estados_alcanz(estado, "Ep")
53:             ealcanzables = ealcanzables + estados
54:             #ealcanzables = ealcanzables  +  estados + [estado]
55:         return ealcanzables
56:
57:     """Metodo que retorna una lista de estados, que representa al conjunto cerraduraEpsilon
58:     de un un conjunto de estados recibido como parametro
59:     """
60:     def cerradura_E(self,conjunto):
61:         cerradura = conjunto
62:         resultante = self.trans_directa_E(conjunto)
63:         while ( len(resultante) > 0  ):
64:             cerradura = cerradura + resultante
65:             resultante = self.trans_directa_E(resultante)
66:         return cerradura
67:
68:     """Metodo que retorna una lista de tuplas , donde cada tupla  representa a un arco,
69:     y cada arco representa a una transicion de una lista de estados a otra solo por medio
70:     de un terminal osea distinto a Epsilon.
71:     """
72:     def mover_terminal_x(self, conjunto):
73:         pre_resul = []
74:         ls_movx = []
75:         for estado in conjunto:
76:             for arco in self.lista_nfa:
77:                 if arco[2] != "Ep" and  arco[0] == estado:
78:                     caracter = arco[2]
79:                     edestino = arco[1]
80:                     pre_resul.append((caracter, edestino))
81:         if len(pre_resul) == 0:
82:             return []
83:         if len(pre_resul) == 1:
84:             return [([pre_resul[0][1]], pre_resul[0][0])]
85:         #ordena por la terminal
86:         pre_resul.sort()
87:
88:         caracter_actual = pre_resul[0][0]
89:         ls_aux = []
90:         for par in pre_resul:
91:             caracter = par[0]
92:             edestino = par[1]
93:             if caracter_actual == caracter:
94:                 ls_aux.append(edestino)
95:             else:
96:                 ls_movx.append((ls_aux, caracter_actual))
97:                 ls_aux = []
98:                 caracter_actual = caracter
```

```
 99:                 ls_aux.append(edestino)
100:         ls_movx.append((ls_aux, caracter_actual))
101:         return ls_movx
102:
103:     """Retorna una lista de tuplas que represeta a un DFA,
104:     utiliza la lista de tuplas NFA , que es un atributo de la clase ;
105:     seteado en el momento de la construccion del objeto.
106:     Esto es a fines de evitar que los metodos requieran constantemente
107:     recibirla como parametro
108:     """
109:     def afn_to_afd(self):
110:         grupo_e = []
111:
112:         einicial = self.lista_nfa[0][0]
113:         grupo_e.append( self.cerradura_E([einicial]))
114:         #retorna una lista de tuplas  [([1,2],a)([a,b], ) ]
115:         for estados_e in grupo_e:
116:             conj_a = self.mover_terminal_x(estados_e)
117:             for ele_term in conj_a:
118:                 conj_e = self.cerradura_E(ele_term[0])
119:                 self.lista_DFA.append((estados_e, conj_e, ele_term[1]))
120:                 if estados_e != conj_e:
121:                     if  not conj_e in grupo_e:
122:                         grupo_e.append(conj_e)
123:
124:     """Metodo que renombra la lista de arcos, para facilitar su legibilidad
125:     el procedimiento setea el nuevo estado de inicio y los de aceptacion
126:     """
127:     def renombrar_nfa(self):
128:         mapa_cadenas = {}
129:         nn_ini = ''
130:         nn_fin = ''
131:         char_actual = 0
132:         prefijo = "st"
133:         for tupla in self.lista_DFA:
134:
135:             repr_ini = ''.join(str(tupla[0]))
136:             repr_fin = ''.join(str(tupla[1]))
137:             if repr_ini in mapa_cadenas:
138:                 nn_ini = mapa_cadenas[repr_ini]
139:             else:
140:                 char_actual = char_actual + 1
141:                 nn_ini = prefijo + str(char_actual)
142:                 mapa_cadenas[repr_ini] = nn_ini
143:
144:             if repr_fin in mapa_cadenas:
145:                 nn_fin = mapa_cadenas[repr_fin]
146:             else:
147:                 char_actual = char_actual + 1
```

```
148:                 nn_fin = prefijo + str(char_actual)
149:                 mapa_cadenas[repr_fin] = nn_fin
150:
151:             #verificamos si el estado es un estado inicial
152:             if self.est_nfa_inicial in tupla[0]:
153:                 self.est_renam_inicial = nn_ini
154:             #verificamos si el estado inicial es un estado de aceptacion y aun no esta en la lista
155:             if self.est_nfa_aceptacion in tupla[0] and \
156:                 not nn_ini in self.ls_renam_aceptacion :
157:                     self.ls_renam_aceptacion.append(nn_ini)
158:             #verificamos si el estado es un estado de aceptacion y aun no esta en la lista
159:             if self.est_nfa_aceptacion in tupla[1] and \
160:                 not nn_fin in self.ls_renam_aceptacion :
161:                     self.ls_renam_aceptacion.append(nn_fin)
162:
163:             self.lista_DFA_renam.append((nn_ini, nn_fin, tupla[2]))
164:         #carga apunta al atributo que guarda el resultado del renombramiento
165:         self.mp_erenombrados = mapa_cadenas
166:
167:     """metodo que obtiene la solo lista de terminales presentes , en la lista de arcos"""
168:     def __solo_terminales(self,lista_arcos):
169:         lista_terminales = []
170:         for arco in lista_arcos:
171:             if not arco[2] in lista_terminales:
172:                 lista_terminales.append(arco[2])
173:         return lista_terminales
174:
175:     """metodo que retorna lista la lista estados de no aceptacion """
176:     def __conj_no_aceptacion(self,lista_arcos):
177:         lista_noacep = []
178:         for arco in lista_arcos:
179:             if not(arco[0] in self.ls_renam_aceptacion):
180:                 if not(arco[0] in lista_noacep) :
181:                     lista_noacep.append(arco[0])
182:         return lista_noacep
183:
184:     """Metodo que implemente el algoritmo de minimizacion de estados
185:     sobre la lista de arcos renombrada """
186:     def minimizar(self, ls_arcos):
187:         ls_terminales = self.__solo_terminales(ls_arcos)
188:         ls_ls_no_acep = self.__conj_no_aceptacion(ls_arcos)
189:         ls_ls_aceptacion = self.ls_renam_aceptacion[:]
190:         ls_ls_estados = [ls_ls_no_acep] + [ls_ls_aceptacion]
191:         for terminal in ls_terminales:
192:             for conj_estados in ls_ls_estados:
193:                 #si tiene un solo elemento se ignora
194:                 if len(conj_estados) <= 1:
195:                     continue
196:                 for estado in conj_estados:
```

```
197:                     #verifica si el estado puede alcanzar algun estado con esa terminal
198:                     alcanzable = self.estados_alcanz(estado, terminal, self.lista_DFA_renam)
199:                     #detemina si el estado alcanzable no esta en el propio conjunto
200:                     #si NO esta en el conjunto, debemos separar el estado en un nuevo conjunto
201:                     if len(alcanzable) == 0:
202:                         ls_ls_estados.append([estado])
203:                         conj_estados.remove(estado)
204:                     #al ser dfa , por medio de una terminal ya solo puede ir a un estado
205:                     elif not alcanzable[0][0] in conj_estados:
206:                         ls_ls_estados.append([estado])
207:                         conj_estados.remove(estado)
208:
209:         #eliminar conjuntos repeticos
210:         #sustituye cada elemento por su representante correspondiente
211:         self.lista_DFA_min = []
212:         for arco in self.lista_DFA_renam:
213:             ini = arco[0]
214:             fin = arco[1]
215:             for conj_estados in ls_ls_estados:
216:                 if len(conj_estados)<=1:
217:                     continue
218:                 representa = conj_estados[0]
219:                 if arco[0] in conj_estados:
220:                     ini = representa
221:                 if arco[1] in conj_estados:
222:                     fin = representa
223:             if not (ini,fin,arco[2]) in self.lista_DFA_min:
224:                 self.lista_DFA_min.append((ini,fin,arco[2]))
225:
226:         #encuentra los nuevos estados de aceptacion, e inicial
227:         for conj_estados in ls_ls_estados:
228:             if len(conj_estados) == 0 :
229:                 continue
230:             if self.est_renam_inicial in conj_estados and self.est_min_inicial == 0 :
231:                 self.est_min_inicial =  conj_estados[0]
232:             for eaceptacion in self.ls_renam_aceptacion:
233:                 if  eaceptacion in conj_estados and not conj_estados[0]  in self.ls_min_aceptacion:
234:                     self.ls_min_aceptacion.append(conj_estados[0])
235:
236:     """Metodos que retornan  los resultados de las operaciones en una diccionario
237:     con claves uniforme, para ser usadas por otros programas como el de reporte
238:     """
239:
240:     """Resumen posterior a minimizar"""
241:     def get_dfa_minimo(self):
242:         return {"estado_inicial":self.est_min_inicial,
243:                 "estados_aceptacion":self.ls_min_aceptacion,
244:                 "lista_arcos":self.lista_DFA_min}
245:
```

```
246:     """Resumen posterior a renombrar """
247:     def get_dfa_renombrado(self):
248:         return {"estado_inicial":self.est_renam_inicial,
249:                 "estados_aceptacion":self.ls_renam_aceptacion,
250:                 "lista_arcos":self.lista_DFA_renam,
251:                 "renombrados":self.mp_erenombrados
252:                 }
253:
254:     """Resumen de nfa recibido como parametro
255:     es cuestionable que este metodo deberia estar en la clase que convierte a NFA
256:     como la existencia de esta clase solo tiene sentido con la salida de los metodos de NFA
257:     se considero aceptable implementarlo aqui
258:     """
259:     def get_nfa_inst(self):
260:         return {"estado_inicial":self.est_nfa_inicial,
261:                 "estados_aceptacion":self.est_nfa_aceptacion,
262:                 "lista_arcos":self.lista_nfa}
```

```
#lista de caracteteres a ignorar

#simbolos_reservados = frozenset([".", "+", "="])
#alfabetos = {'LETRAS': [chr(l) for l in range(97, 122)],
#    'DIGITOS': ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']}

"""Clase que administra los alfabetos y simbolos reservados"""
class SimbolosAdmin:
    def __init__(self):
        self.simbolos_reservados = frozenset([".", "+", "="])
        self.alfabetos = {'LETRAS': [chr(l) for l in range(97, 122)],
    'DIGITOS': ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']}

    def agregar_alfabeto(self, nombre, listasimbolos):
        if nombre in self.alfabetos:
            return False
        self.alfabetos[nombre]= listasimbolos
        return True



"""clase estatica que encapsula las constantes"""
class Lexconst:
    EOF=-1
    SIMB_DEF = 1
    PARENT_DER = 2
    PARENT_IZQ = 3
    KLEENE = 4
    KLEENE_PLUS = 5
    UNION = 6
    UNOCERO = 7
    CONCAT = 8
    comp_lex = {"(": PARENT_IZQ, ")": PARENT_DER, "*": KLEENE, "+": KLEENE_PLUS,
"|": UNION, "?": UNOCERO, ".": CONCAT}
    ignore_list = frozenset([' '])


"""Tipos de componentes lexicos:
    tokens reservados son aquellos que no pueden formar parte de los alfabetos
    definidos.
    Luego estan los simbolos de los alfabetos definidos por el usuario
    """
class Token_attr:
    AT_LEXEMA = "lexema"
    AT_TIPO = "tipo"

class AnLex:
    """Analizador Lexico"""
    pos = -1
```

```
    num_linea = 0
    alfabetos = {}
    simbolos_reservados=[]
    def __init__(self, obSimbolAdmin):
        self.alfabetos = obSimbolAdmin.alfabetos
        self.simbolos_reservados = obSimbolAdmin.simbolos_reservados

    def set_flujo(self, flujo):
        self.fuente = flujo
        #analogo al eof
        self.maxpos = len(flujo)-1
        AnLex.pos = -1
    """Retorna el siguiente caracter no vacio o bien None cuando ya no
    quedancaracteres a consumir"""
    def get_char(self):
        if AnLex.pos < self.maxpos:
            AnLex.pos += 1
            while self.fuente[AnLex.pos] in Lexconst.ignore_list:
                AnLex.pos += 1
            if AnLex.pos <= self.maxpos:
                return self.fuente[AnLex.pos]
        else:
            return None

    #Devuelve el caracter
    def unget_char(self):
        AnLex.pos -= 1
        return self.fuente[self.pos]

    def es_simbolo(self, s):
        for alfabeto in self.alfabetos:
            for simbolos in self.alfabetos[alfabeto]:
                if s in simbolos:
                    return True
        return False

    def next_token(self):
        while True:
            ch = self.get_char()
            if ch is None:
                return self.set_tokeninf('', Lexconst.EOF)
            if ch in Lexconst.ignore_list:
                True
            elif ch == "\n":
                AnLex.num_linea += 1
            elif ch in Lexconst.comp_lex:
                #coincide con algun caracter reservado
                return self.set_tokeninf(ch, Lexconst.comp_lex[ch])
            elif self.es_simbolo(ch):
```

```
   99:                 #debe ser el componente lexico definido por los alfabetos
  100:                 return self.set_tokeninf(ch, Lexconst.SIMB_DEF)
  101:
  102:     def set_tokeninf(self, lexema, componente):
  103:         return {"lexema": lexema, "tipo": componente}
```

```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file '.\gui_entrada.ui'
#
# Created: Tue Oct 22 12:56:08 2013
#      by: PyQt4 UI code generator 4.10.3
#
# WARNING! All changes made in this file will be lost!

from PyQt4 import QtCore, QtGui


try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s

try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)

"""Clase que tratara con la intefaz grafica a fin modificar lo menos posible el codigo generado
por la herramienta de qt
"""


"""
Codigo generado por la herramienta de pyqt4
"""


class Ui_Form(object):

    def setupUi(self, Form):
        Form.setObjectName(_fromUtf8("Form"))
        Form.resize(620, 375)

        self.listWidget_Alfabetos = QtGui.QListWidget(Form)
        self.listWidget_Alfabetos.setGeometry(QtCore.QRect(80, 120, 421, 71))
        self.listWidget_Alfabetos.setObjectName(_fromUtf8("listWidget_Alfabetos"))

        self.lineEdit_AlfabetoNombre = QtGui.QLineEdit(Form)
        self.lineEdit_AlfabetoNombre.setGeometry(QtCore.QRect(170, 70, 113, 20))
        self.lineEdit_AlfabetoNombre.setObjectName(_fromUtf8("lineEdit_AlfabetoNombre"))
```

```
50:
51:         self.pushButton_Agregar = QtGui.QPushButton(Form)
52:         self.pushButton_Agregar.setGeometry(QtCore.QRect(290, 70, 75, 23))
53:
54:
55:
56:         self.pushButton_Agregar.setObjectName(_fromUtf8("pushButton_Agregar"))
57:         self.label_2 = QtGui.QLabel(Form)
58:         self.label_2.setGeometry(QtCore.QRect(70, 70, 91, 16))
59:         self.label_2.setObjectName(_fromUtf8("label_2"))
60:
61:         self.lineEdit_simbolo = QtGui.QLineEdit(Form)
62:         self.lineEdit_simbolo.setGeometry(QtCore.QRect(170, 90, 40, 20))
63:         self.lineEdit_simbolo.setObjectName(_fromUtf8("lineEdit_simbolo"))
64:
65:
66:         #lista de simbolos
67:         self.edt_lssimb = QtGui.QLineEdit(Form)
68:         self.edt_lssimb.setGeometry(QtCore.QRect(220, 90, 120, 20))
69:         self.edt_lssimb.setObjectName(_fromUtf8("edt_lssimb"))
70:
71:
72:
73:         self.label = QtGui.QLabel(Form)
74:         self.label.setGeometry(QtCore.QRect(110, 90, 46, 13))
75:         self.label.setObjectName(_fromUtf8("label"))
76:
77:
78:
79:         self.label_defregular = QtGui.QLabel(Form)
80:         self.label_defregular.setGeometry(QtCore.QRect(30, 50, 121, 17))
81:         font = QtGui.QFont()
82:         font.setPointSize(9)
83:         font.setBold(True)
84:         font.setWeight(75)
85:         self.label_defregular.setFont(font)
86:         self.label_defregular.setObjectName(_fromUtf8("label_defregular"))
87:         self.label_alfabetos = QtGui.QLabel(Form)
88:         self.label_alfabetos.setGeometry(QtCore.QRect(30, 220, 115, 17))
89:         font = QtGui.QFont()
90:         font.setPointSize(9)
91:         font.setBold(True)
92:         font.setWeight(75)
93:         self.label_alfabetos.setFont(font)
94:         self.label_alfabetos.setObjectName(_fromUtf8("label_alfabetos"))
95:         self.lineEdit = QtGui.QLineEdit(Form)
96:         self.lineEdit.setGeometry(QtCore.QRect(80, 240, 420, 21))
97:         self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
98:         self.btn_procesar = QtGui.QPushButton(Form)
```

```
   99:         self.btn_procesar.setGeometry(QtCore.QRect(80, 300, 75, 23))
  100:         font = QtGui.QFont()
  101:         font.setBold(True)
  102:         font.setWeight(75)
  103:         self.btn_procesar.setFont(font)
  104:         self.btn_procesar.setStyleSheet(_fromUtf8("color:rgb(27, 27, 81)\n"
  105: ""))
  106:         self.btn_procesar.setObjectName(_fromUtf8("btn_procesar"))
  107:         self.label_3 = QtGui.QLabel(Form)
  108:         self.label_3.setGeometry(QtCore.QRect(30, 10, 403, 30))
  109:         self.label_3.setObjectName(_fromUtf8("label_3"))
  110:         self.btn_acercade = QtGui.QPushButton(Form)
  111:         self.btn_acercade.setGeometry(QtCore.QRect(580, 0, 40, 23))
  112:         self.btn_acercade.setObjectName(_fromUtf8("btn_acercade"))
  113:
  114:         self.retranslateUi(Form)
  115:         QtCore.QMetaObject.connectSlotsByName(Form)
  116:
  117:     def retranslateUi(self, Form):
  118:         Form.setWindowTitle(_translate("Form", "Reg-Tool", None))
  119:         self.pushButton_Agregar.setText(_translate("Form", "Agregar", None))
  120:         self.label_2.setText(_translate("Form", "Nombre Alfabeto:", None))
  121:         self.label.setText(_translate("Form", "Simbolo:", None))
  122:         self.label_defregular.setText(_translate("Form", "Alfabetos", None))
  123:         self.label_alfabetos.setText(_translate("Form", "DefiniciÃ³n regular", None))
  124:         self.btn_procesar.setText(_translate("Form", "Procesar", None))
  125:         self.label_3.setText(_translate("Form", "1-Defina nuevos alfabetos  2-seleccione los que utilizara  \n3-Ingrese
 la expresion regular  4-Procesar", None))
  126:         self.btn_acercade.setText(_translate("Form", "...?", None))
  127:         self.edt_lssimb.setText(_translate("Form", "", None))
  128:
  129:         #############
  130:         self.btn_ejecutar = QtGui.QPushButton(Form)
  131:         self.btn_ejecutar.setGeometry(QtCore.QRect(430, 295, 90, 26))
  132:         self.btn_ejecutar.setText("Ejecutar codigo")
  133:         font = QtGui.QFont()
  134:         font.setPointSize(8)
  135:         font.setBold(True)
  136:         self.btn_ejecutar.setFont(font)
  137:         #############
  138:
  139:
  140:         #self.listWidget_Alfabetos.addItems( self.lineEdit_simbolo.text())
  141:
  142:     def buttonClicked(self):
  143:         nombre = self.lineEdit_AlfabetoNombre.text()
  144:         #modelo_gui.nuevo_alfabeto(nombre )
  145:         item = QtGui.QListWidgetItem()
  146:         item.setText(nombre)
```

```
147:         self.listWidget_Alfabetos.addItem(item)
148:
149: ########################################################################
150: class MyLineEdit(QtGui.QLineEdit):
151:     def __init__(self, *args):
152:         QtGui.QLineEdit.__init__(self, *args)
153:
154:     def event(self, event):
155:         if (event.type()==QtCore.QEvent) and (event.key()==QtCore.Qt.Key_Enter):
156:             self.emit(QtCore.SIGNAL("enterPressed"))
157:             return True
158:         return QtGui.QLineEdit.event(self, event)
159:
160: ########################################################################
161:
162:
```

```
def avanzar_entrada():
    lmax = len(cadena)
    cont = -1
    while cont <= lmax:
        cont = cont + 1
        if cont == lmax:
            yield None
        yield cadena[cont]
global cadena
global aceptado
global pertenece
cadena = raw_input('evaluar>')
lector = avanzar_entrada()
c_entrada = lector.next()
estado='st1'
ent = False
error = False
while not estado in ['st4', 'st1', 'st3', 'st2'] or c_entrada != None:
    if estado == 'st1':
        if c_entrada=='a':
            estado = 'st2'
            c_entrada = lector.next()
            ent = True
            continue
    if estado == 'st1':
        if c_entrada=='c':
            estado = 'st3'
            c_entrada = lector.next()
            ent = True
            continue
    if estado == 'st1':
        if c_entrada=='d':
            estado = 'st4'
            c_entrada = lector.next()
            ent = True
            continue
    if estado == 'st2':
        if c_entrada=='a':
            estado = 'st2'
            c_entrada = lector.next()
            ent = True
            continue
    if estado == 'st3':
        if c_entrada=='c':
            estado = 'st3'
            c_entrada = lector.next()
            ent = True
            continue
    if estado == 'st4':
```

```
50:         if c_entrada=='d':
51:             estado = 'st4'
52:             c_entrada = lector.next()
53:             ent = True
54:             continue
55:     if not ent:
56:         error = True
57:         break
58:     ent = False
59: if estado in ['st4', 'st1', 'st3', 'st2'] and not error :
60:     print 'correcto!'
61: else:
62:     print 'Error!'
63: raw_input('evaluar>')
```

```
def avanzar_entrada():
    lmax = len(cadena)
    cont = -1
    while cont <= lmax:
        cont = cont + 1
        if cont == lmax:
            yield None
        yield cadena[cont]
global cadena
global aceptado
global pertenece
cadena = raw_input('evaluar>')
lector = avanzar_entrada()
c_entrada = lector.next()
estado='st1'
ent = False
error = False
while not estado in ['st2', 'st1'] or c_entrada != None:
    if estado == 'st1':
        if c_entrada=='b':
            estado = 'st2'
            c_entrada = lector.next()
            ent = True
            continue
    if estado == 'st2':
        if c_entrada=='b':
            estado = 'st2'
            c_entrada = lector.next()
            ent = True
            continue
    if not ent:
        error = True
        break
    ent = False
if estado in ['st2', 'st1'] and not error :
    print 'correcto!'
else:
    print 'Error!'
```