

Instituto Politécnico Nacional

Escuela Superior de Cómputo

* **UNIDAD DE APRENDIZAJE:** COMPILADORES
* **PROFESOR:** RAFAEL NORMAN SAUCEDO DELGADO
* **EQUIPO:**

ARTEAGA LARA SAMUEL

BARRERA PÉREZ TONATIHU

FLORES TEPATL GISELLE

MENDOZA JAIMES IAN

MONROY MARTOS ELIOTH

## **Práctica 7. Diseño e implementación de la clase para obtener las tablas LR(0), LR(1) y LALR**

* **GRUPO:** 3CM6

# Introducción

# **Analizadores Sintácticos Ascendentes**

El objetivo de un análisis ascendente consiste en construir el árbol sintáctico desde abajo hacia arriba, es decir, desde los tokens (nodos hoja del árbol) hacia el axioma inicial (raíz del árbol), lo cual disminuye el número de reglas mal aplicadas con respecto al caso descendente.

Cualquier mecanismo de análisis ascendente consiste en partir de una configuración inicial e ir aplicando operaciones, cada una de las cuales permite pasar de una configuración origen a otro destino.

Las operaciones disponibles son las siguientes:

1. ACEPTAR: se acepta la cadena.

2. RECHAZAR: la cadena de entrada no es válida.

3. REDUCIR: consiste en aplicar una regla de producción hacia atrás a algunos elementos situados en el extremo derecho de “α”.

4. DESPLAZAR: consiste únicamente en quitar el terminal más a la izquierda de β y ponerlo a la derecha de α.[1]

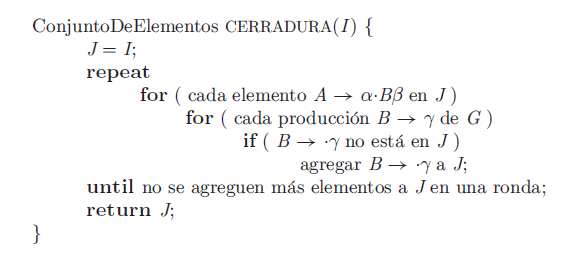
**Gramáticas LR(k)**

* Representan un conjunto más amplio de gramáticas que LL(1)
* Expresión más sencillas
* Se caracterizan por:
  + L: Procesamos la cadena de entrada de izquierda a derecha (from left to-right)
  + R: proporcionan la derivación más a la derecha de la cadena de entrada en orden inverso (rightmost derivation)
  + k: se examinan k símbolos de la entrada por anticipado, para tomar la decisión sobre la acción a realizar.

**El método LR(0):** Es el más fácil de implementar, pero el que tiene menos poder de reconocimiento. No usa la información del símbolo de preanálisis para decidir la acción a realizar.

Para construir la colección LR(0) canónica de una gramática, definimos una gramática aumentada y dos funciones, CERRADURA e ir\_A.

**Cerradura:**



**ir\_A**

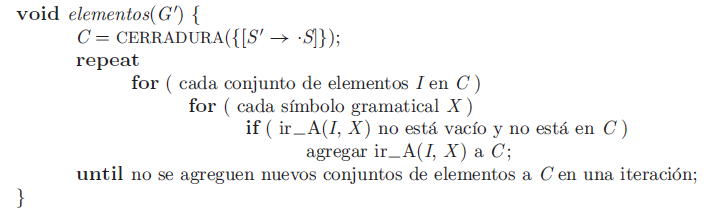
ir\_A(I, X), en donde I es un conjunto de elementos y X es un símbolo

gramatical.

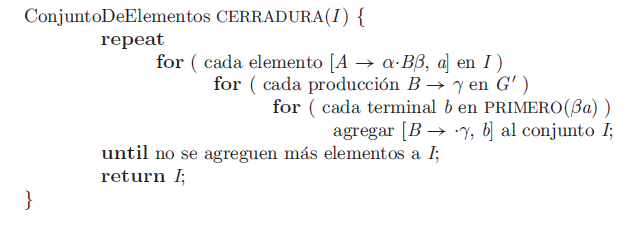
ir\_A(I, X) se define como la cerradura del conjunto de todos los elementos [A →

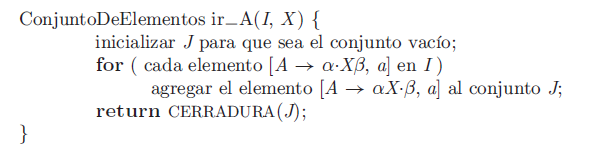
αX·β], de tal forma que [A → α·Xβ] se encuentre en I.

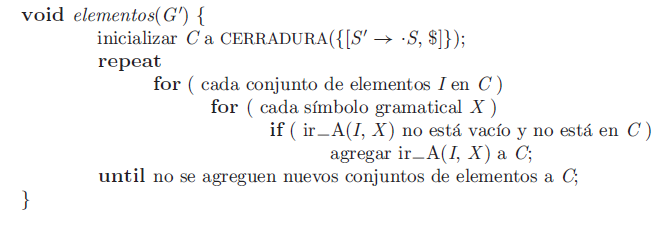
De manera intuitiva, la función ir\_A se utiliza para definir las transiciones en el autómata LR(0) para una gramática. Los estados del autómata corresponden a los conjuntos de elementos, e ir\_A(I, X) especifica la transición que proviene del estado para I, con la entrada X.



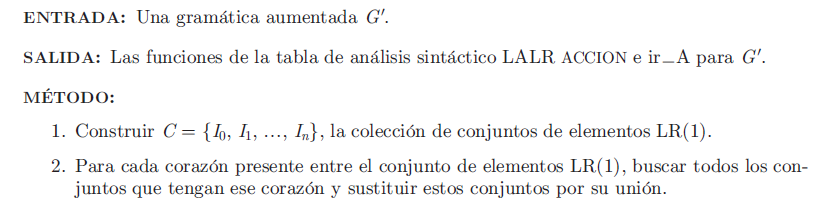
**El método LR(1):** Es el más poderoso y costoso. El tamaño del autómata pila para el reconocimiento se incrementa considerablemente.

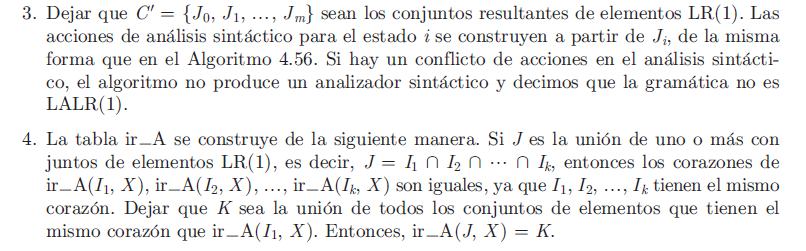






**El método LALR(1):** (del inglés Look-Ahead LR, con símbolo de anticipación). Es una versión simplificada del LR(1), que combina el coste en tamaño (eficiencia) de los métodos SLR con la potencia del LR(1).





# 

# 

# 

# 

# Desarrollo

gramatica.py

|  |
| --- |
| import re  import string  class Gramatica:  def \_\_init\_\_(self, archivo):  self.nombre\_archivo = archivo  self.no\_terminales = dict()  self.terminales = dict()  self.inicial = None  self.gramatica = dict()  self.gramatica\_id = dict()  self.extendido = None  def leer\_archivo(self):  archivo = open(self.nombre\_archivo, 'r')  primera = 0  j = 1  for linea in archivo:  if primera == 0:  self.guardar\_no\_terminales(linea)  primera = 1  continue  if primera == 1:  self.inicial = linea[0]  primera = 2  j = self.guardar\_produccion(linea, j)  self.terminales.update({"$": j})  for caracter in string.ascii\_uppercase:  if caracter not in self.no\_terminales:  self.extendido = caracter  break  self.crear\_id\_gramatica()  def crear\_id\_gramatica(self):  contador = 1  for clave, valor in self.gramatica.items():  for produccion in valor.get("producciones"):  self.gramatica\_id.update({clave + '->' + produccion: contador})  contador += 1  def imprimir\_gramatica(self):  for clave, valor in self.gramatica\_id.items():  print(str(valor) + ' ' + clave)  def guardar\_produccion(self, linea, j):  izq = linea[0]  der = linea[3:]  der = re.match("[^\n]\*", der).group()  if izq not in self.gramatica:  self.gramatica.update({izq: {  "producciones": list(),  "primero": False,  "siguiente": False  }})  self.gramatica.get(izq).get("producciones").append(der)  for c in der:  expr\_regular = re.match("[a-df-z\(\)\+\-\\*]", c)  if expr\_regular is not None and c not in self.terminales:  self.terminales.update({c: j})  j += 1  return j  def obtener\_izq(self, N):  return self.gramatica.get(N).get("producciones")  def guardar\_no\_terminales(self, linea):  i = 1  for c in linea:  expr\_regular = re.match("[A-Z]", c)  if expr\_regular is not None and c not in self.no\_terminales:  self.no\_terminales.update({c: i})  i += 1 |

Este primer archivo contiene la clase necesaria para leer la gramática de un archivo y guarda los componentes principales de las gramáticas libres de contexto además de tener métodos auxiliares para la manipulación de dichas gramáticas.

auxiliares.py

|  |
| --- |
| import pdb  import re  from sintactico.LR.gramatica import Gramatica  class Tipo:  # LR(0)  TIPO\_A = 0 # A->X.aY  TIPO\_B = 1 # A->X.  TIPO\_C = 3 # S'->S. aun no lo uso  class Auxiliares(Gramatica):  def \_\_init\_\_(self, archivo):  super(Auxiliares, self).\_\_init\_\_(archivo)  def es\_epsilon(self, A):  return A == 'e'  # Tal vez de error  def es\_terminal(self, A):  return A in self.terminales  def es\_no\_terminal(self, A):  return A in self.no\_terminales  def es\_inicial(self, S):  return self.inicial == S  def primero(self, A):  conjunto = set()  for a in A:  if a in self.gramatica:  self.gramatica.get(a)["primero"] = False  conjunto\_extra = self.P(a)  conjunto.update(conjunto\_extra)  if 'e' not in conjunto\_extra:  if 'e' in conjunto:  conjunto.remove('e')  break  return conjunto  def P(self, A):  conjunto = set()  if self.es\_terminal(A) or self.es\_epsilon(A):  conjunto.add(A)  else:  if self.gramatica.get(A).get("primero"):  return conjunto  else:  self.gramatica.get(A)["primero"] = True  producciones = self.gramatica.get(A).get("producciones")  for produccion in producciones:  i = 0  while i < len(produccion):  extra = self.P(produccion[i])  conjunto.update(extra)  if 'e' in extra:  i += 1  else:  break  return conjunto  def siguiente(self, N):  for clave, valor in self.gramatica.items():  valor["siguiente"] = False  return self.S(N)  def S(self, N):  conjunto = set()  if not self.es\_terminal(N) and self.gramatica.get(N).get("siguiente"):  return conjunto  else:  if not self.es\_terminal(N):  self.gramatica.get(N)["siguiente"] = True  if self.es\_inicial(N):  conjunto.add('$')  # A -> xN  no\_terminales = self.obtener\_izquierda(N)  if len(no\_terminales) != 0:  for n in no\_terminales:  conjunto.update(self.S(n))  # A -> xNy  no\_terminales = self.obtener\_derecha(N)  if len(no\_terminales) != 0:  for simbolo in no\_terminales:  primero = self.primero(simbolo.get("cadena"))  if 'e' in primero:  primero.remove('e')  conjunto.update(self.S(simbolo.get("clave")))  conjunto.update(primero)  if not self.es\_terminal(N):  self.gramatica.get(N)["siguiente"] = False  return conjunto  def obtener\_izquierda(self, N):  claves = list()  for clave, valor in self.gramatica.items():  for v in valor.get("producciones"):  if N == v[len(v)-1]:  claves.append(clave)  return claves  def obtener\_derecha(self, N):  simbolos = list()  for clave, valor in self.gramatica.items():  for v in valor.get("producciones"):  for m in re.finditer(N, v):  if m.start() != len(v)-1:  simbolos.append({"clave": clave, "cadena": v[m.start()+1:]})  return simbolos  def obtener\_producciones(self, N):  claves = list()  for clave, valor in self.gramatica.items():  for v in valor.get("producciones"):  if v.find(N) != -1:  claves.append(clave)  return claves |

En esta clase se encuentran definidas las funciones de PRIMERO y SIGUIENTE.

lr\_cero.py

|  |
| --- |
| from sintactico.LR.auxiliares import Auxiliares, Tipo  class Elemento(Tipo):  def \_\_init\_\_(self, izquierda, derecha, punto):  self.izq = izquierda  self.der = derecha  self.punto = punto  self.ID = self.izq + "->" + self.der + ':' + str(self.punto)  self.tipo = None  def \_\_repr\_\_(self):  return str(self)  def \_\_str\_\_(self):  return self.ID  def set\_tipo(self, terminales):  if len(self.der) == self.punto:  self.tipo = self.TIPO\_B  elif self.der[self.punto] in terminales:  self.tipo = self.TIPO\_A  class Conjunto:  def \_\_init\_\_(self, kernel):  self.kernel = kernel  self.repr\_kernel = self.obtener\_representacion()  self.conjunto = None  self.numero = None  def obtener\_representacion(self):  representacion = set()  for elemento in self.kernel:  representacion.add(elemento.ID)  return representacion  def \_\_repr\_\_(self):  return str(self)  def \_\_str\_\_(self):  return str(self.repr\_kernel) + ":" + str(self.numero)  class LR\_CERO(Auxiliares, Tipo):  def \_\_init\_\_(self, archivo):  super(LR\_CERO, self).\_\_init\_\_(archivo)  self.leer\_archivo()  self.conjuntos = None  self.num\_columnas = 0  self.num\_filas = 0  self.tabla = None  def cerradura(self, I):  J = list(I)  agregado = dict()  for A in J:  if A.punto < len(A.der) and not self.es\_terminal(A.der[A.punto]):  if A.der[A.punto] not in agregado:  agregado.update({A.der[A.punto]: False})  producciones = self.obtener\_izq(A.der[A.punto])  for pro in producciones:  if not agregado.get(A.der[A.punto]):  ele = None  if pro[0] == 'e':  ele = Elemento(A.der[A.punto], pro, 1)  else:  ele = Elemento(A.der[A.punto], pro, 0)  ele.set\_tipo(self.terminales)  J.append(ele)  agregado[A.der[A.punto]] = True  J\_set = frozenset(J)  return J\_set  def mover(self, I, X):  conjunto = set()  for i in I:  if i.punto < len(i.der) and i.der[i.punto] == X:  ele = Elemento(i.izq, i.der, i.punto+1)  ele.set\_tipo(self.terminales)  conjunto.add(ele)  return frozenset(conjunto)  def obtener\_conjuntos(self):  lista = list()  inicio = set()  ele = Elemento(self.extendido, self.inicial, 0)  ele.set\_tipo(self.terminales)  inicio.add(ele)  conjunto\_inicio = Conjunto(inicio)  conjunto\_inicio.conjunto = self.cerradura(inicio)  conjunto\_inicio.numero = 0  indice = 1  lista.append(conjunto\_inicio)  for i in lista:  for X in self.no\_terminales:  kernel = self.mover(i.conjunto, X)  if len(kernel) != 0:  if not self.ya\_existe(lista, kernel):  conjunto\_aux = Conjunto(kernel)  conjunto\_aux.conjunto = self.cerradura(kernel)  conjunto\_aux.numero = indice  lista.append(conjunto\_aux)  indice += 1  for X in self.terminales:  kernel = self.mover(i.conjunto, X)  if len(kernel) != 0:  if not self.ya\_existe(lista, kernel):  conjunto\_aux = Conjunto(kernel)  conjunto\_aux.conjunto = self.cerradura(kernel)  conjunto\_aux.numero = indice  lista.append(conjunto\_aux)  indice += 1  self.conjuntos = set(lista)  self.num\_columnas = len(self.terminales) + len(self.no\_terminales)  self.num\_filas = indice  self.tabla = [["err"] \* self.num\_columnas for i in range(self.num\_filas)]  def ya\_existe(self, lista, kernel):  aux = set()  for elemento in kernel:  aux.add(elemento.ID)  for conjunto in lista:  if conjunto.repr\_kernel == aux:  return conjunto.numero  return False  def construir\_tabla(self):  print('PRODUCCIONES:')  self.imprimir\_gramatica()  for I in self.conjuntos:  for X, valor in self.no\_terminales.items():  temp = self.mover(I.conjunto, X)  num = self.ya\_existe(self.conjuntos, temp)  if num:  self.agregar\_elemento(I.numero, valor-1, num)  for elemento in I.conjunto:  if elemento.tipo == self.TIPO\_A:  temp = self.mover(I.conjunto, elemento.der[elemento.punto])  num = self.ya\_existe(self.conjuntos, temp)  if num:  i = I.numero  j = len(self.no\_terminales) + self.terminales.get(elemento.der[elemento.punto])-1  self.agregar\_elemento(i, j, "d"+str(num))  if elemento.tipo == self.TIPO\_B:  if elemento.izq != self.extendido:  siguientes = self.siguiente(elemento.izq)  for sig in siguientes:  llave = elemento.izq + '->' + elemento.der  r = self.gramatica\_id.get(llave)  i = I.numero  j = len(self.no\_terminales) + self.terminales.get(sig)-1  self.agregar\_elemento(i, j, "r" + str(r))  else:  i = I.numero  j = len(self.no\_terminales)+self.terminales.get("$")-1  self.agregar\_elemento(i, j, "ACE")  self.imprimir\_tabla("Tabla LR(0):")  def agregar\_elemento(self, i, j, num):  if self.tabla[i][j] == "err":  self.tabla[i][j] = set()  self.tabla[i][j].add(num)  def imprimir\_tabla(self, titulo):  print(titulo)  for t in self.no\_terminales.keys():  print(t, end="\t")  for t in self.terminales.keys():  print(t, end="\t")  print("estado")  for fila, edo in zip(self.tabla, range(self.num\_filas)):  for columna in fila:  print(columna, end="\t")  print(edo) |

Estas clases son las más importante de los tres archivos que conforman la creación de tablas para análisis ascendente debido a que el resto de clases en los demás archivos hereda de las clases definidas en este archivo.

lr\_uno.py

|  |
| --- |
| from sintactico.LR.lr\_cero import Elemento as ElementoCero  from sintactico.LR.lr\_cero import Conjunto as ConjuntoCero  from sintactico.LR.lr\_cero import LR\_CERO  class Elemento(ElementoCero):  def \_\_init\_\_(self, izquierda, derecha, punto, terminal):  super(Elemento, self).\_\_init\_\_(izquierda, derecha, punto)  self.terminal = terminal  self.ID = self.ID + "," + self.terminal  class Conjunto(ConjuntoCero):  def \_\_init\_\_(self, kernel):  super(Conjunto, self).\_\_init\_\_(kernel)  class LR\_UNO(LR\_CERO):  def \_\_init\_\_(self, archivo):  super(LR\_UNO, self).\_\_init\_\_(archivo)  def cerradura(self, I):  agregado = dict()  for i in I:  agregado.update({str(i): True})  J = list(I)  for A in J:  if A.punto < len(A.der) and self.es\_no\_terminal(A.der[A.punto]):  B = A.der[A.punto]  producciones = self.obtener\_izq(B)  for pro in producciones:  sub\_cadena = A.der[A.punto+1:]  sub\_cadena += A.terminal  primeros = self.primero(sub\_cadena)  for pri in primeros:  ele = Elemento(B, pro, 0, pri)  if str(ele) not in agregado:  ele.set\_tipo(self.terminales)  J.append(ele)  agregado.update({str(ele): True})  return set(J)  def mover(self, I, X):  J = set()  for i in I:  if i.punto < len(i.der) and i.der[i.punto] == X:  ele = Elemento(i.izq, i.der, i.punto+1, i.terminal)  ele.set\_tipo(self.terminales)  J.add(ele)  return J  def obtener\_conjuntos(self):  lista = list()  inicial = Elemento(self.extendido, self.inicial, 0, "$")  inicial.set\_tipo(self.terminales)  inicio = set()  inicio.add(inicial)  conjunto\_inicio = Conjunto(inicio)  conjunto\_inicio.conjunto = self.cerradura(inicio)  conjunto\_inicio.numero = 0  indice = 1  lista.append(conjunto\_inicio)  for con in lista:  for X in self.no\_terminales:  kernel = self.mover(con.conjunto, X)  if len(kernel) != 0:  if not self.ya\_existe(lista, kernel):  conjunto\_aux = Conjunto(kernel)  conjunto\_aux.conjunto = self.cerradura(kernel)  conjunto\_aux.numero = indice  lista.append(conjunto\_aux)  indice += 1  for X in self.terminales:  kernel = self.mover(con.conjunto, X)  if len(kernel) != 0:  if not self.ya\_existe(lista, kernel):  conjunto\_aux = Conjunto(kernel)  conjunto\_aux.conjunto = self.cerradura(kernel)  conjunto\_aux.numero = indice  lista.append(conjunto\_aux)  indice += 1  self.conjuntos = set(lista)  self.num\_columnas = len(self.terminales) + len(self.no\_terminales)  self.num\_filas = indice  self.tabla = [["err"] \* self.num\_columnas for i in range(self.num\_filas)]  def construir\_tabla(self):  print('PRODUCCIONES:')  self.imprimir\_gramatica()  for I in self.conjuntos:  for A, valor in self.no\_terminales.items():  temp = self.mover(I.conjunto, A)  num = self.ya\_existe(self.conjuntos, temp)  if num:  i = I.numero  j = valor-1  self.agregar\_elemento(i, j, num)  for elemento in I.conjunto:  if elemento.tipo == self.TIPO\_A:  temp = self.mover(I.conjunto, elemento.der[elemento.punto])  num = self.ya\_existe(self.conjuntos, temp)  if num:  i = I.numero  j = len(self.no\_terminales)+self.terminales.get(elemento.der[elemento.punto])-1  self.agregar\_elemento(i, j, "d"+str(num))  if elemento.tipo == self.TIPO\_B:  if elemento.izq != self.extendido:  llave = elemento.izq + '->' + elemento.der  r = self.gramatica\_id.get(llave)  i = I.numero  j = len(self.no\_terminales)+self.terminales.get(elemento.terminal)-1  self.agregar\_elemento(i, j, "r" + str(r))  else:  i = I.numero  j = len(self.no\_terminales)+self.terminales.get('$')-1  self.agregar\_elemento(i, j, 'ACC')  self.imprimir\_tabla("Tabla LR(1):") |

Muchos métodos de la clase LR\_CERO son sobrescritos en esta clase debido a que la construcción de los conjuntos al igual que las operaciones de mover y cerradura son diferentes por los elementos canónicos de la tabla.

lalr.py

|  |
| --- |
| from sintactico.LR.lr\_uno import Conjunto as ConjuntoUno  from sintactico.LR.lr\_cero import Elemento as ElementoCero  from sintactico.LR.lr\_uno import LR\_UNO  class Elemento(ElementoCero):  def \_\_init\_\_(self, izquierda, derecha, punto, terminal):  super(Elemento, self).\_\_init\_\_(izquierda, derecha, punto)  self.terminal = terminal  self.ID\_CERO = self.ID  self.ID = self.ID + "," + self.terminal  class Conjunto(ConjuntoUno):  def \_\_init\_\_(self, kernel):  super(Conjunto, self).\_\_init\_\_(kernel)  self.conjunto\_cero = self.obtener\_conjunto\_cero()  def obtener\_conjunto\_cero(self):  representacion = set()  for elemento in self.kernel:  representacion.add(elemento.ID\_CERO)  return representacion  class LALR(LR\_UNO):  def \_\_init\_\_(self, archivo):  super(LALR, self).\_\_init\_\_(archivo)  def obtener\_conjuntos(self):  lista = list()  inicial = Elemento(self.extendido, self.inicial, 0, "$")  inicial.set\_tipo(self.terminales)  inicio = set()  inicio.add(inicial)  conjunto\_inicio = Conjunto(inicio)  conjunto\_inicio.conjunto = self.cerradura(inicio)  conjunto\_inicio.numero = 0  indice = 1  lista.append(conjunto\_inicio)  for con in lista:  for X in self.no\_terminales:  kernel = self.mover(con.conjunto, X)  if len(kernel) != 0:  if not super(LALR, self).ya\_existe(lista, kernel):  conjunto\_aux = Conjunto(kernel)  conjunto\_aux.conjunto = self.cerradura(kernel)  conjunto\_aux.numero = indice  lista.append(conjunto\_aux)  indice += 1  for X in self.terminales:  kernel = self.mover(con.conjunto, X)  if len(kernel) != 0:  if not super(LALR, self).ya\_existe(lista, kernel):  conjunto\_aux = Conjunto(kernel)  conjunto\_aux.conjunto = self.cerradura(kernel)  conjunto\_aux.numero = indice  lista.append(conjunto\_aux)  indice += 1  self.conjuntos = set(lista)  self.num\_columnas = len(self.terminales) + len(self.no\_terminales)  self.num\_filas = indice  def unir\_conjuntos(self):  lista = list(self.conjuntos)  i = 0  nuevos\_conjuntos = set()  while i < len(lista):  temp = lista[i]  j = i + 1  agregar = False  kernel = set()  otros\_conjuntos = set()  while j < len(lista):  if temp.conjunto\_cero == lista[j].conjunto\_cero:  kernel = kernel.union(lista[j].kernel)  otros\_conjuntos = otros\_conjuntos.union(lista[j].conjunto)  lista.pop(j)  agregar = True  j += 1  if agregar:  kernel = kernel.union(temp.kernel)  aux\_conjunto = Conjunto(kernel)  otros\_conjuntos = otros\_conjuntos.union(temp.conjunto)  aux\_conjunto.conjunto = otros\_conjuntos  aux\_conjunto.numero = None  nuevos\_conjuntos.add(aux\_conjunto)  lista.pop(i)  i = 0  else:  i += 1  numero\_conjuntos = len(lista) + len(nuevos\_conjuntos)  indices = [True] \* numero\_conjuntos  self.conjuntos = set()  for ele in lista:  if ele.numero < numero\_conjuntos:  self.conjuntos.add(ele)  indices[ele.numero] = False  else:  conta = 0  while not indices[conta]:  conta += 1  ele.numero = conta  indices[ele.numero] = False  self.conjuntos.add(ele)  j = 0  lista\_conjuntos = list(nuevos\_conjuntos)  while j < len(lista\_conjuntos):  conta = 0  while not indices[conta]:  conta += 1  lista\_conjuntos[j].numero = conta  indices[conta] = False  self.conjuntos.add(lista\_conjuntos[j])  j += 1  self.num\_filas = len(self.conjuntos)  self.tabla = [["err"] \* self.num\_columnas for i in range(self.num\_filas)]  def construir\_tabla(self):  print('PRODUCCIONES:')  self.imprimir\_gramatica()  for I in self.conjuntos:  for A, valor in self.no\_terminales.items():  temp = self.mover(I.conjunto, A)  num = self.ya\_existe(self.conjuntos, temp)  if num:  i = I.numero  j = valor-1  self.agregar\_elemento(i, j, num)  for elemento in I.conjunto:  if elemento.tipo == self.TIPO\_A:  temp = self.mover(I.conjunto, elemento.der[elemento.punto])  num = self.ya\_existe(self.conjuntos, temp)  if num:  i = I.numero  j = len(self.no\_terminales)+self.terminales.get(elemento.der[elemento.punto])-1  self.agregar\_elemento(i, j, "d"+str(num))  if elemento.tipo == self.TIPO\_B:  if elemento.izq != self.extendido:  llave = elemento.izq + '->' + elemento.der  r = self.gramatica\_id.get(llave)  i = I.numero  j = len(self.no\_terminales)+self.terminales.get(elemento.terminal)-1  self.agregar\_elemento(i, j, "r" + str(r))  else:  i = I.numero  j = len(self.no\_terminales)+self.terminales.get('$')-1  self.agregar\_elemento(i, j, 'ACC')  self.imprimir\_tabla("Tabla LALR:")  def ya\_existe(self, lista, kernel):  aux = set()  for elemento in kernel:  aux.add(elemento.ID\_CERO)  for conjunto in lista:  if conjunto.conjunto\_cero == aux:  return conjunto.numero  return False |

El archivo anterior utiliza la mayoría de métodos de la clase LR\_UNO pero el método para construir conjuntos cambia porque los elementos que forman dichos conjuntos son diferentes a los usados en LR\_UNO, también se incluye el método para unir conjuntos de estados y se modifica el método de construir tabla debido a que cambia la forma de buscar elementos que ya existen necesario para las transiciones y los desplazamientos.

prueba.py

|  |
| --- |
| from sintactico.LR.lalr import LALR  from sintactico.LR.lr\_cero import LR\_CERO  from sintactico.LR.lr\_uno import LR\_UNO  archivo = input('Introduce el nombre del archivo con la gramatica plis: ')  tipo = input("""Tipos de tabla  1) LR(0)  2) LR(1)  3) LALR  Selecciona alguna: """)  if int(tipo) == 1:  analizador = LR\_CERO(archivo)  analizador.obtener\_conjuntos()  elif int(tipo) == 2:  analizador = LR\_UNO(archivo)  analizador.obtener\_conjuntos()  else:  analizador = LALR(archivo)  analizador.obtener\_conjuntos()  analizador.unir\_conjuntos()  analizador.construir\_tabla() |

El archivo anterior es usado con fines de prueba para poder ingresar alguna gramática y seleccionar la tabla a construir

# Resultados

Se realizaron dos pruebas sobre cada tabla. Por lo que se tienen dos archivos de texto cada uno con una gramática de prueba, la estructura del archivo además de contar con las producciones cuenta con la declaración de los símbolos no terminales en la primera línea del archivo.

Es importante señalar que en la tabla las casillas con la palabra “err” indican que ocurrió un error sintáctico, el resto de elementos que se encuentran entre llaves son el resto de posibles operaciones que se pueden realizar.

**Gramatica de prueba 1**

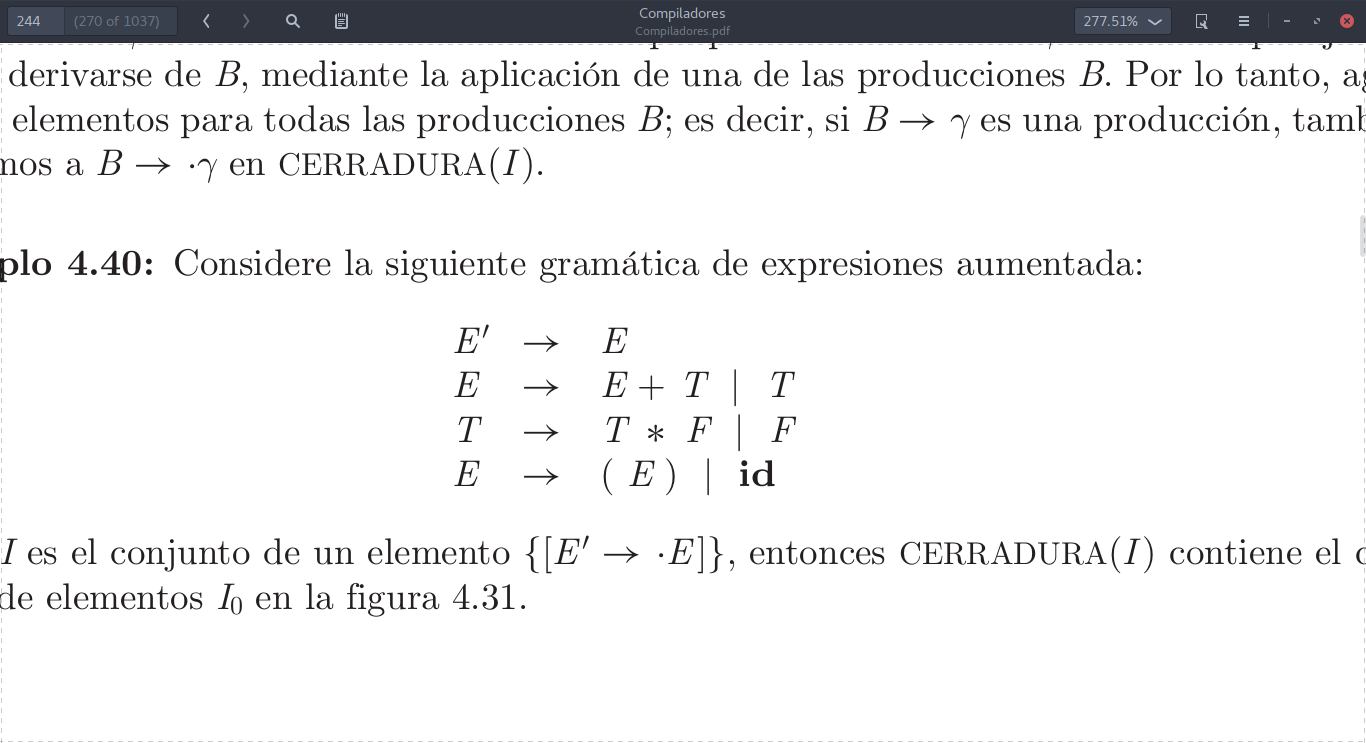


Tabla LR(0)

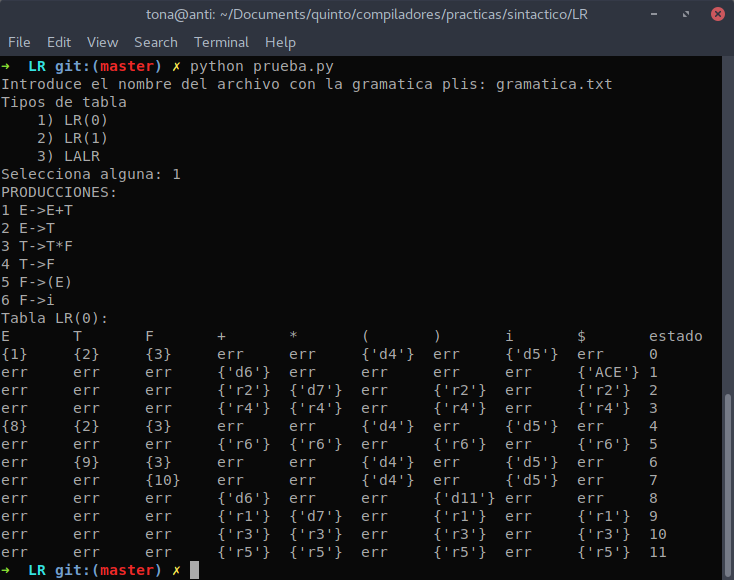
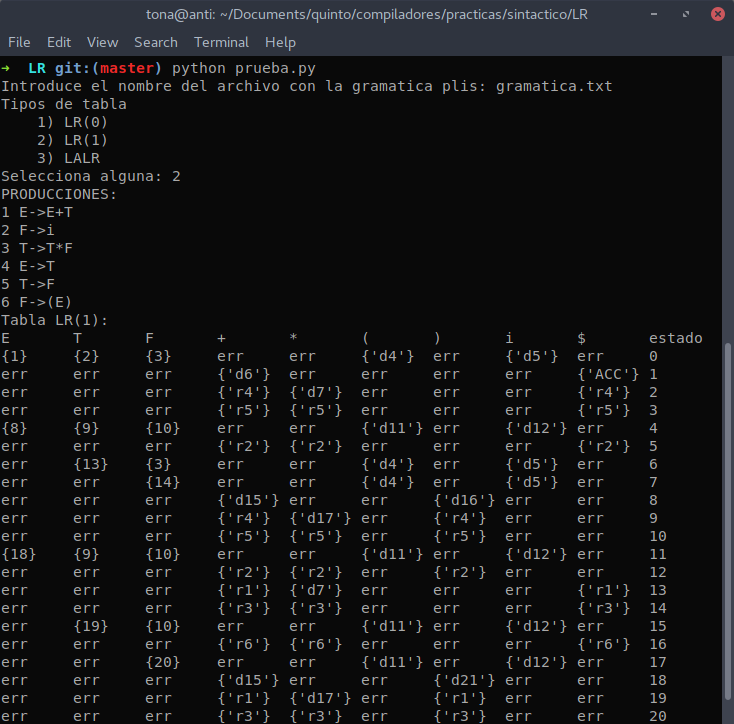


Tabla LR(1)



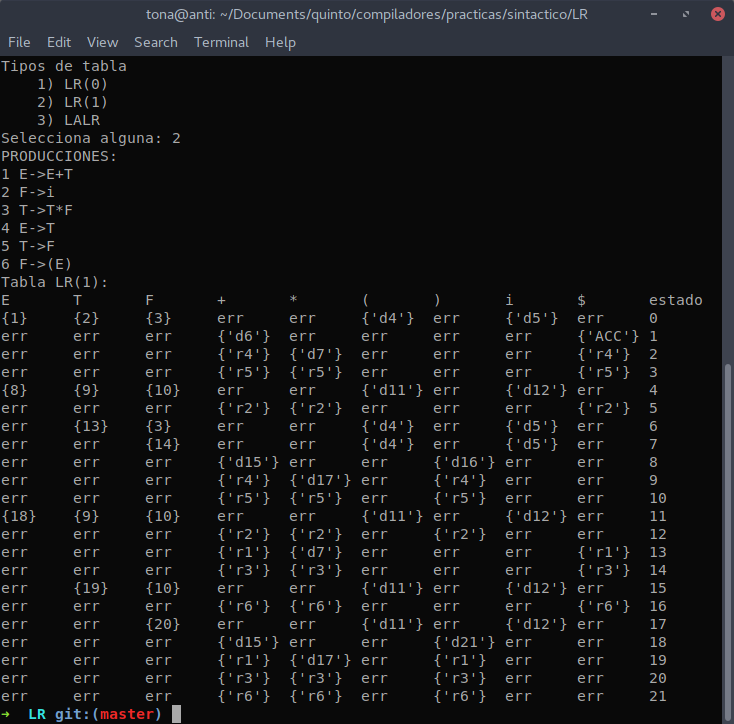
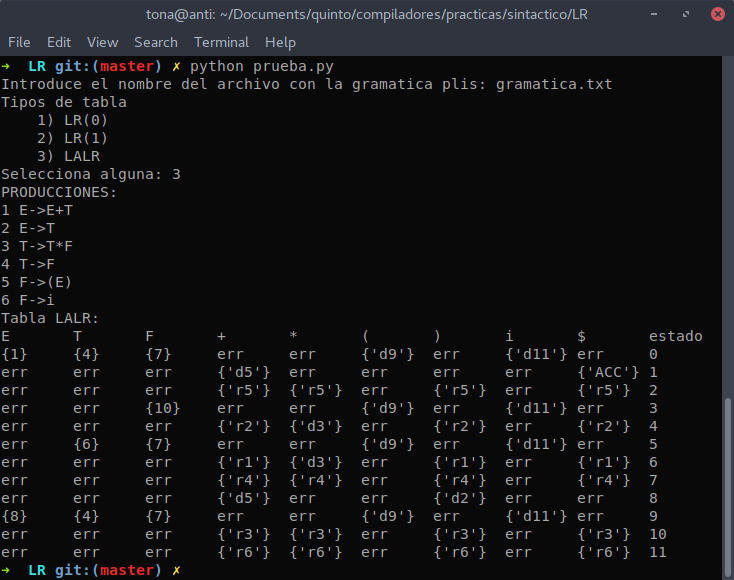


Tabla LALR

****

**Gramatica de prueba 2**

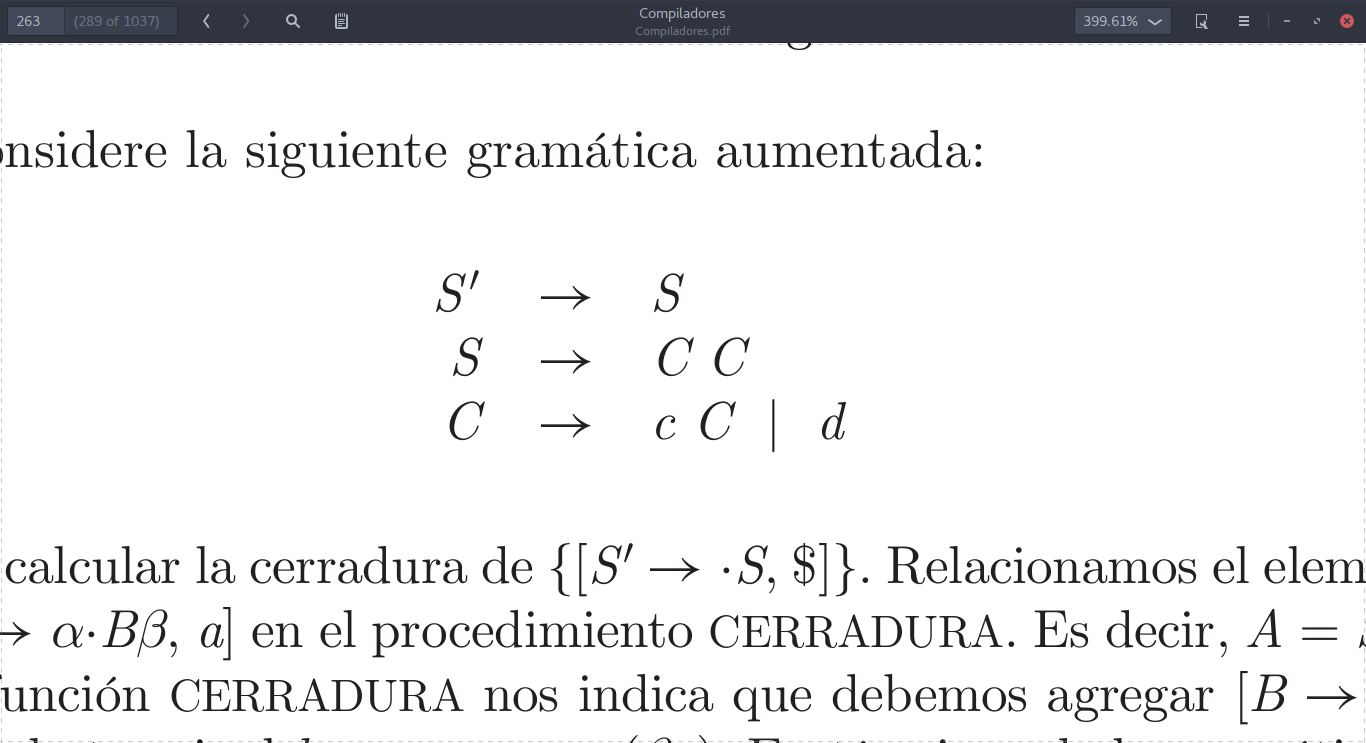
****

Tabla LR(0)

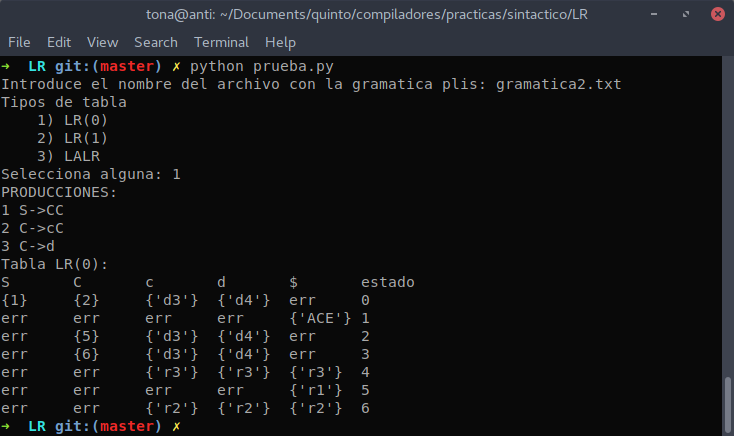
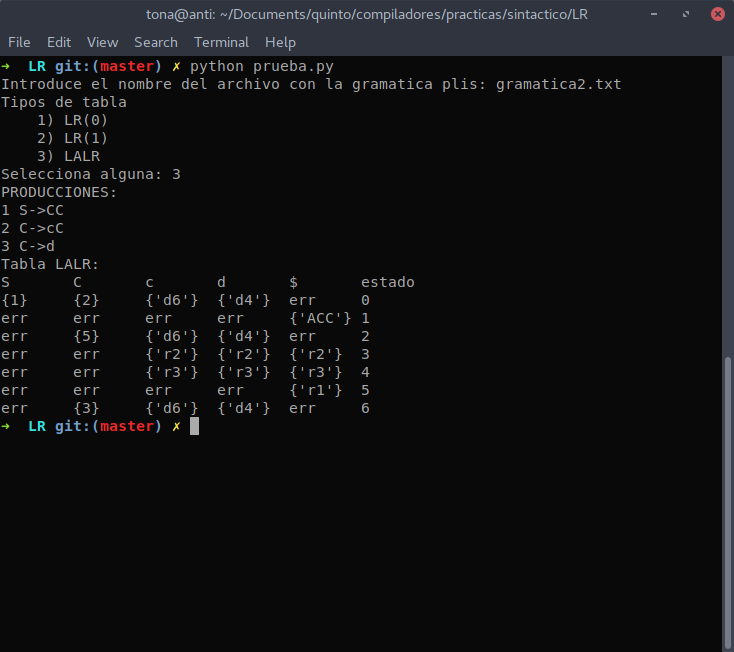


Tabla LR(1)



Tabla LALR



# Conclusiones

## Arteaga Lara Samuel

## Con la realización de esta práctica pudimos reafirmar los conocimientos de los analizadores LR, pudiendo hacer una comparación de estos con los LL, viendo las ventajas que ofrece un analizador ascendente sobre uno descendente, haciendo notar el por qué es que son más utilizados en la práctica, siendo que los LR construyen un autómata de pila que reconocen mediante transiciones una derivación por la derecha. Los analizadores vistos en esta ocasión (LR(0), LR(1) y LALR) nos permitieron de igual forma ver cuales son las ventajas y desventajas que presentan incluso sobre ellos mismos.

## Barrera Pérez Carlos Tonatihu

Esta práctica deja en claro que utilizar analizadores sintácticos LR es la mejor opción que se tiene a la hora de realizar un compilador debido a la versatilidad que presentan respecto a los analizadores LL o por descenso recursivo. Es por esto que la mayoría, si no es que todas, de las herramientas para generación de analizadores utilizan este tipo de analizador. Respecto al desarrollo de la práctica sin duda el usar herencia en las clases que se diseñaron redujo el trabajo ya que muchos atributos y métodos se repiten entre los tres tipos de analizadores LR que se trabajaron en esta práctica.

## Flores Tepatl Giselle

En esta práctica pudimos conocer otro tipo de analizadores sintácticos; los analizadores ascendentes, como LR(0), LR(1) y LALR; los cuales, a diferencia de los descendentes, construyen el árbol de análisis sintáctico partiendo de los nodos hoja hacia la raíz del árbol.

Este tipo de analizadores ascendentes presentan ciertas ventajas sobre los descendentes; como por ejemplo, resultan ser métodos potentes que permiten reconocer la mayoría de las construcciones de los lenguajes de programación.

## Mendoza Jaimes Ian

Las gramáticas LR son más versátiles que las LL, por lo que los analizadores del tipo LR son sin duda los más utilizados. Tenemos tres diferentes opciones, que difieren en velocidad y espacio utilizado, por lo que dependerá de nosotros decidir cual es la mas conveniente para nuestro compilador. En esta práctica pudimos ver las tres técnicas y compararlas asi como conocer como es que se llevan a cabo.

## Monroy Matos Elioth

Después de haber trabajado con los analizadores sintácticos descendentes (LL) en esta práctica trabajamos con los analizadores sintácticos ascendentes los cuales son más poderosos que los primeros mencionados, y en la práctica resultan ser los más usados ya que imponen menos restricciones sobre las gramáticas que podemos usar (en el caso de los LL la gramática no podía tener reducción izquierda, LR si lo permite). En esta práctica los implementamos mediante en tres formas: LR(0), LR(1) y LALR, donde cada uno tiene sus propias ventajas y desventajas y depende del diseñador del compilador escoger cual resulta ser el más adecuado para el trabajo que se desarrolla.

# Referencias

* [1] EcuRed. (2017). “Analizador Sintáctico AScendente”. [Online]. Disponible: https://www.ecured.cu/Analizador\_sint%C3%A1ctico\_ascendente
* Aho, A. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Boston: Pearson/Addison Wesley.