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:

```
ConjuntoDeElementos CERRADURA(I) {
    J = I;
    repeat
        for ( cada elemento A → α·Bβ en J )
            for ( cada producción B → γ de G )
                if ( B → ·γ no está en J )
                    agregar B → ·γ a J;
    until no se agreguen más elementos a J en una ronda;
    return J;
}
```

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.

```
void elementos(G') {
    C = CERRADURA({[S' → ·S]});
    repeat
        for ( cada conjunto de elementos I en C )
            for ( cada símbolo gramatical X )
                if ( ir_A(I, X) no está vacío y no está en C )
                    agregar ir_A(I, X) a C;
    until no se agreguen nuevos conjuntos de elementos a C en una iteración;
}
```

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

```
ConjuntoDeElementos CERRADURA(I) {
        repeat
            for ( cada elemento [A → α·Bβ, a] en I )
                for ( cada producción B → γ en G' )
                    for ( cada terminal b en PRIMERO(βa) )
                        agregar [B → ·γ, b] al conjunto I;
        until no se agreguen más elementos a I;
        return I;
}

ConjuntoDeElementos ir_A(I, X) {
        inicializar J para que sea el conjunto vacío;
        for ( cada elemento [A → α·Xβ, a] en I )
            agregar el elemento [A → αX·β, a] al conjunto J;
        return CERRADURA(J);
}

void elementos(G') {
        inicializar C a CERRADURA({[S' → ·S, $]});
        repeat
            for ( cada conjunto de elementos I en C )
                for ( cada símbolo gramatical X )
                    if ( ir_A(I, X) no está vacío y no está en C )
                        agregar ir_A(I, X) a C;
        until no se agreguen nuevos conjuntos de elementos a C;
}
```

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).

ENTRADA: Una gramática aumentada G'.

SALIDA: Las funciones de la tabla de análisis sintáctico LALR ACCION e ir_A para G'.

MÉTODO:

1. Construir $C = \{I_0, I_1, \ldots, I_n\}$, la colección de conjuntos de elementos LR(1).
2. Para cada corazón presente entre el conjunto de elementos LR(1), buscar todos los conjuntos que tengan ese corazón y sustituir estos conjuntos por su unión.
3. Dejar que $C' = \{J_0, J_1, \ldots, J_m\}$ sean los conjuntos resultantes de elementos LR(1). Las acciones de análisis sintáctico para el estado i se construyen a partir de J_i, de la misma forma que en el Algoritmo 4.56. Si hay un conflicto de acciones en el análisis sintáctico, el algoritmo no produce un analizador sintáctico y decimos que la gramática no es LALR(1).
4. La tabla ir_A se construye de la siguiente manera. Si J es la unión de uno o más conjuntos de elementos LR(1), es decir, $J = I_1 \cap I_2 \cap \cdots \cap I_k$, entonces los corazones de ir_A(I_1, X), ir_A(I_2, X), ..., ir_A(I_k, X) son iguales, ya que $I_1, I_2, \ldots, I_k$ tienen el mismo corazón. Dejar que K sea la unión de todos los conjuntos de elementos que tienen el mismo corazón que ir_A(I_1, X). Entonces, ir_A(J, X) = K.

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
```

```python
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

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
E &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Tabla LR(0)

```
tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/LR
File  Edit  View  Search  Terminal  Help
→  LR git:(master) ✗ python prueba.py
Introduce el nombre del archivo con la gramatica plis: gramatica.txt
Tipos de tabla
    1) LR(0)
    2) LR(1)
    3) LALR
Selecciona alguna: 1
PRODUCCIONES:
1 E->E+T
2 E->T
3 T->T*F
4 T->F
5 F->(E)
6 F->i
Tabla LR(0):
E       T       F       +       *       (       )       i       $       estado
{1}     {2}     {3}     err     err     {'d4'}  err     {'d5'}  err     0
err     err     err     {'d6'}  err     err     err     err     {'ACE'} 1
err     err     err     {'r2'}  {'d7'}  err     {'r2'}  err     {'r2'}  2
err     err     err     {'r4'}  {'r4'}  err     {'r4'}  err     {'r4'}  3
{8}     {2}     {3}     err     err     {'d4'}  err     {'d5'}  err     4
err     err     err     {'r6'}  {'r6'}  err     {'r6'}  err     {'r6'}  5
err     {9}     {3}     err     err     {'d4'}  err     {'d5'}  err     6
err     err     {10}    err     err     {'d4'}  err     {'d5'}  err     7
err     err     err     {'d6'}  err     err     {'d11'} err     err     8
err     err     err     {'r1'}  {'d7'}  err     {'r1'}  err     {'r1'}  9
err     err     err     {'r3'}  {'r3'}  err     {'r3'}  err     {'r3'}  10
err     err     err     {'r5'}  {'r5'}  err     {'r5'}  err     {'r5'}  11
→  LR git:(master) ✗
```

Tabla LR(1)

```
tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/LR
File  Edit  View  Search  Terminal  Help
→  LR git:(master) python prueba.py
Introduce el nombre del archivo con la gramatica plis: gramatica.txt
Tipos de tabla
    1) LR(0)
    2) LR(1)
    3) LALR
Selecciona alguna: 2
PRODUCCIONES:
1 E->E+T
2 F->i
3 T->T*F
4 E->T
5 T->F
6 F->(E)
```

```
Tabla LR(1):
E       T       F       +       *       (       )       i       $       estado
{1}     {2}     {3}     err     err     {'d4'}  err     {'d5'}  err     0
err     err     err     {'d6'}  err     err     err     err     {'ACC'} 1
err     err     err     {'r4'}  {'d7'}  err     err     err     {'r4'}  2
err     err     err     {'r5'}  {'r5'}  err     err     err     {'r5'}  3
{8}     {9}     {10}    err     err     {'d11'} err     {'d12'} err     4
err     err     err     {'r2'}  {'r2'}  err     err     err     {'r2'}  5
err     {13}    {3}     err     err     {'d4'}  err     {'d5'}  err     6
err     err     {14}    err     err     {'d4'}  err     {'d5'}  err     7
err     err     err     {'d15'} err     err     {'d16'} err     err     8
err     err     err     {'r4'}  {'d17'} err     {'r4'}  err     err     9
err     err     err     {'r5'}  {'r5'}  err     {'r5'}  err     err     10
{18}    {9}     {10}    err     err     {'d11'} err     {'d12'} err     11
err     err     err     {'r2'}  {'r2'}  err     {'r2'}  err     err     12
err     err     err     {'r1'}  {'d7'}  err     err     err     {'r1'}  13
err     err     err     {'r3'}  {'r3'}  err     err     err     {'r3'}  14
err     {19}    {10}    err     err     {'d11'} err     {'d12'} err     15
err     err     err     {'r6'}  {'r6'}  err     err     err     {'r6'}  16
err     err     {20}    err     err     {'d11'} err     {'d12'} err     17
err     err     err     {'d15'} err     err     {'d21'} err     err     18
err     err     err     {'r1'}  {'d17'} err     {'r1'}  err     err     19
err     err     err     {'r3'}  {'r3'}  err     {'r3'}  err     err     20
err     err     err     {'r6'}  {'r6'}  err     {'r6'}  err     err     21
➜  LR git:(master)
```

Tabla LALR

```
tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/LR
File  Edit  View  Search  Terminal  Help
➜  LR git:(master) ✗ python prueba.py
Introduce el nombre del archivo con la gramatica plis: gramatica.txt
Tipos de tabla
    1) LR(0)
    2) LR(1)
    3) LALR
Selecciona alguna: 3
PRODUCCIONES:
1 E->E+T
2 E->T
3 T->T*F
4 T->F
5 F->(E)
6 F->i
Tabla LALR:
E       T       F       +       *       (       )       i       $       estado
{1}     {4}     {7}     err     err     {'d9'}  err     {'d11'} err     0
err     err     err     {'d5'}  err     err     err     err     {'ACC'} 1
err     err     err     {'r5'}  {'r5'}  err     {'r5'}  err     {'r5'}  2
err     err     {10}    err     err     {'d9'}  err     {'d11'} err     3
err     err     err     {'r2'}  {'d3'}  err     {'r2'}  err     {'r2'}  4
err     {6}     {7}     err     err     {'d9'}  err     {'d11'} err     5
err     err     err     {'r1'}  {'d3'}  err     {'r1'}  err     {'r1'}  6
err     err     err     {'r4'}  {'r4'}  err     {'r4'}  err     {'r4'}  7
err     err     err     {'d5'}  err     err     {'d2'}  err     err     8
{8}     {4}     {7}     err     err     {'d9'}  err     {'d11'} err     9
err     err     err     {'r3'}  {'r3'}  err     {'r3'}  err     {'r3'}  10
err     err     err     {'r6'}  {'r6'}  err     {'r6'}  err     {'r6'}  11
➜  LR git:(master) ✗
```

Gramatica de prueba 2

$$
\begin{aligned}
S &\rightarrow C\ C \\
C &\rightarrow c\ C \mid d
\end{aligned}
$$

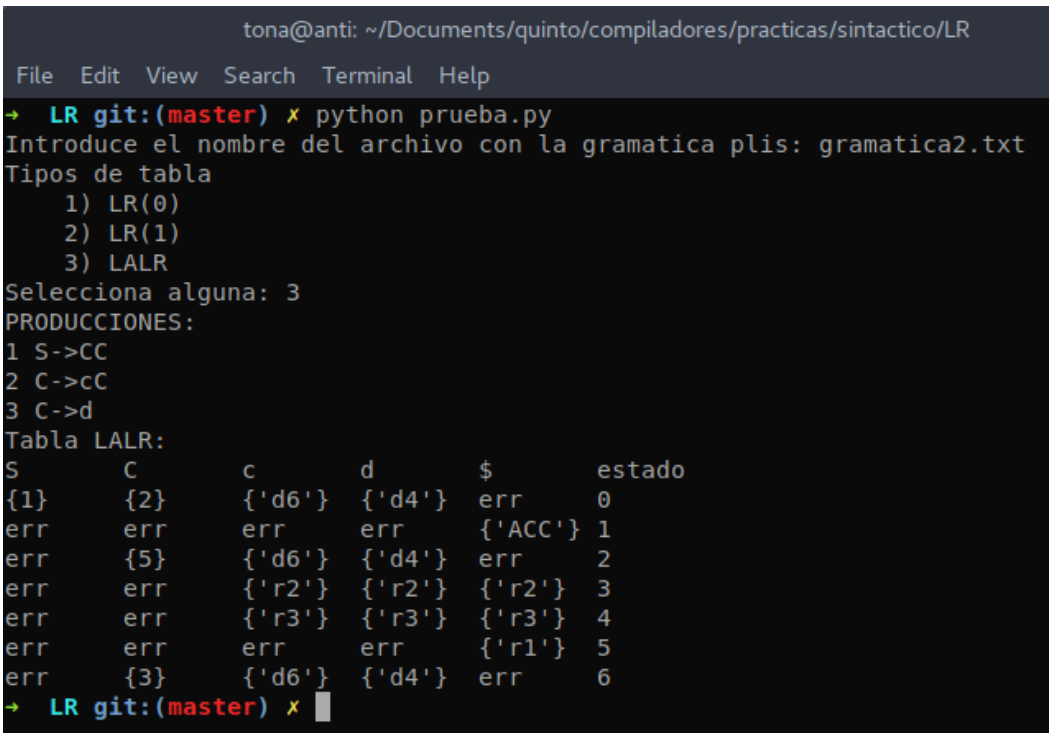
Tabla LR(0)

```
tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/LR
File  Edit  View  Search  Terminal  Help
→  LR git:(master) ✗ python prueba.py
Introduce el nombre del archivo con la gramatica plis: gramatica2.txt
Tipos de tabla
    1) LR(0)
    2) LR(1)
    3) LALR
Selecciona alguna: 1
PRODUCCIONES:
1 S->CC
2 C->cC
3 C->d
Tabla LR(0):
S       C       c       d       $       estado
{1}     {2}     {'d3'}  {'d4'}  err     0
err     err     err     err     {'ACE'} 1
err     {5}     {'d3'}  {'d4'}  err     2
err     {6}     {'d3'}  {'d4'}  err     3
err     err     {'r3'}  {'r3'}  {'r3'}  4
err     err     err     err     {'r1'}  5
err     err     {'r2'}  {'r2'}  {'r2'}  6
→  LR git:(master) ✗
```

Tabla LR(1)

```
tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/LR
File  Edit  View  Search  Terminal  Help
→  LR git:(master) ✗ python prueba.py
Introduce el nombre del archivo con la gramatica plis: gramatica2.txt
Tipos de tabla
    1) LR(0)
    2) LR(1)
    3) LALR
Selecciona alguna: 2
PRODUCCIONES:
1 S->CC
2 C->cC
3 C->d
Tabla LR(1):
S       C       c       d       $       estado
{1}     {2}     {'d3'}  {'d4'}  err     0
err     err     err     err     {'ACC'} 1
err     {5}     {'d6'}  {'d7'}  err     2
err     {8}     {'d3'}  {'d4'}  err     3
err     err     {'r3'}  {'r3'}  err     4
err     err     err     err     {'r1'}  5
err     {9}     {'d6'}  {'d7'}  err     6
err     err     err     err     {'r3'}  7
err     err     {'r2'}  {'r2'}  err     8
err     err     err     err     {'r2'}  9
→  LR git:(master) ✗
```

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.