

Laboratorio E Documentación

Nombre: Gabriel Alejandro Vicente Lorenzo

Carné: 20498

SLR-1

- Gramática aumentada obtenida

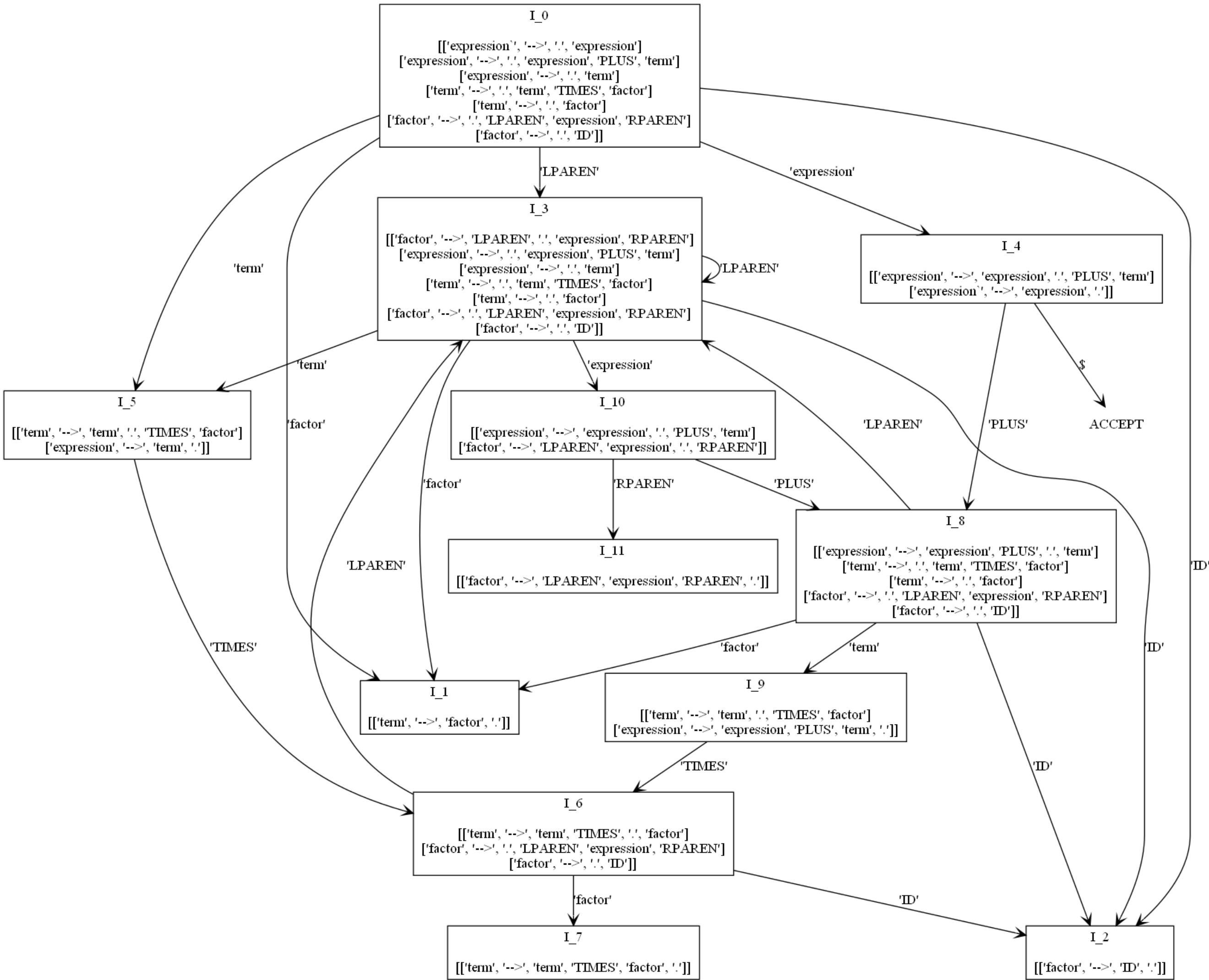
```
OUTPUT  DEBUG CONSOLE  PROBLEMS  TERMINAL

ython.exe "c:/Users/charl/Desktop/S12023/Disenio de Lenguajes/LaboratorioE_20498/src/lr.py"

----- Gramatica aumentada para LR(0) -----

expression` --> expression
expression --> expression PLUS term
expression --> term
term --> term TIMES factor
term --> factor
factor --> LPAREN expression RPAREN
factor --> ID
```

- Representación visual del LR(0)



SLR-2

- Gramática aumentada obtenida

```

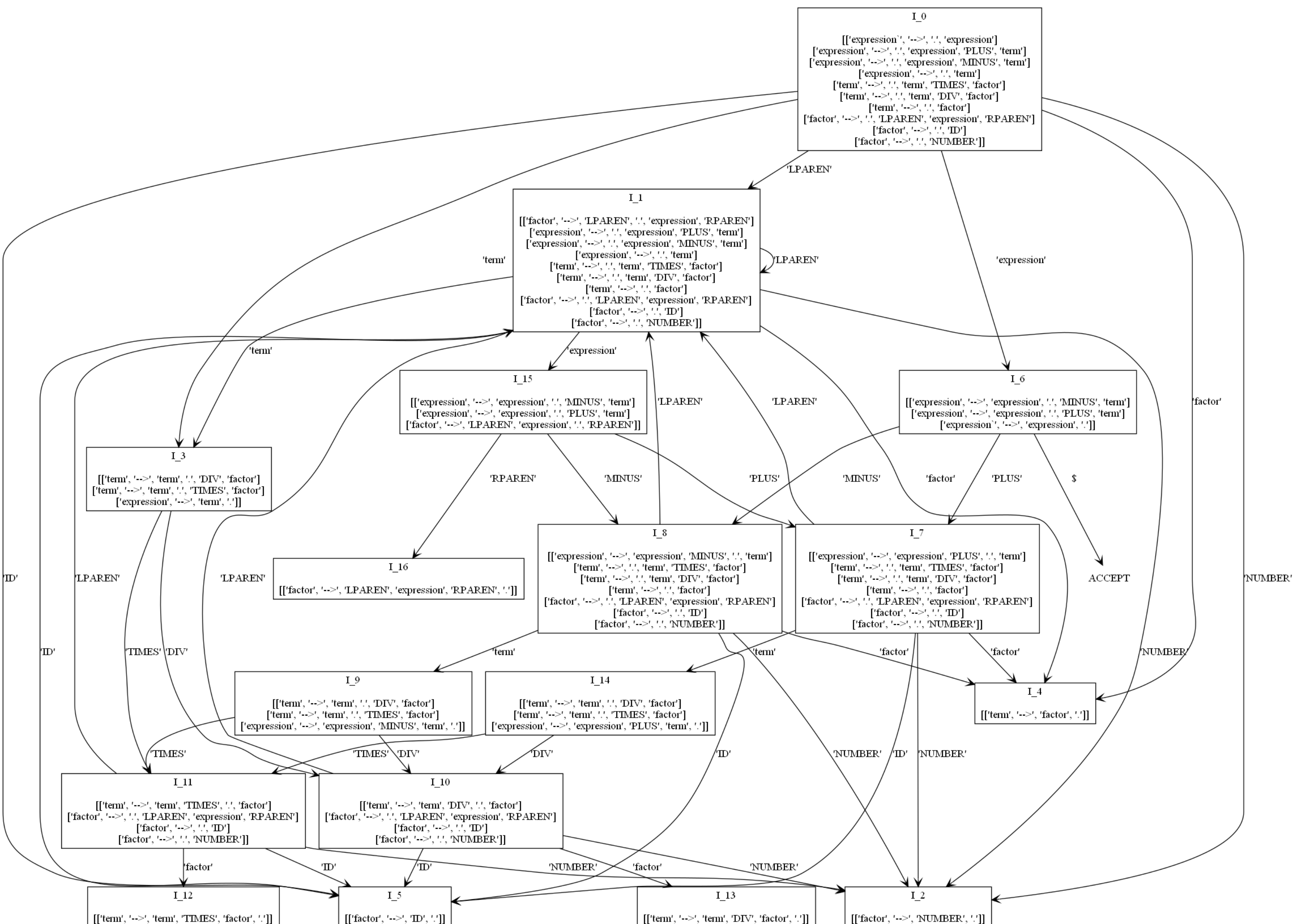
OUTPUT  DEBUG CONSOLE  PROBLEMS  TERMINAL

PS C:\Users\charl\Desktop\S12023\Disenio de Lenguajes\LaboratorioE_20498> & C:/Users/charl/AppData/Local/Programs/Python/Python38-32/python.exe "c:/Users/charl/Desktop/S12023/Disenio de Lenguajes/LaboratorioE_20498/src/lr.py"

----- Gramatica aumentada para LR(0) -----

expression` --> expression
expression --> expression PLUS term
expression --> expression MINUS term
expression --> term
term --> term TIMES factor
term --> term DIV factor
term --> factor
factor --> LPAREN expression RPAREN
factor --> ID
factor --> NUMBER
  
```

- Representación visual del LR(0)



SLR-3

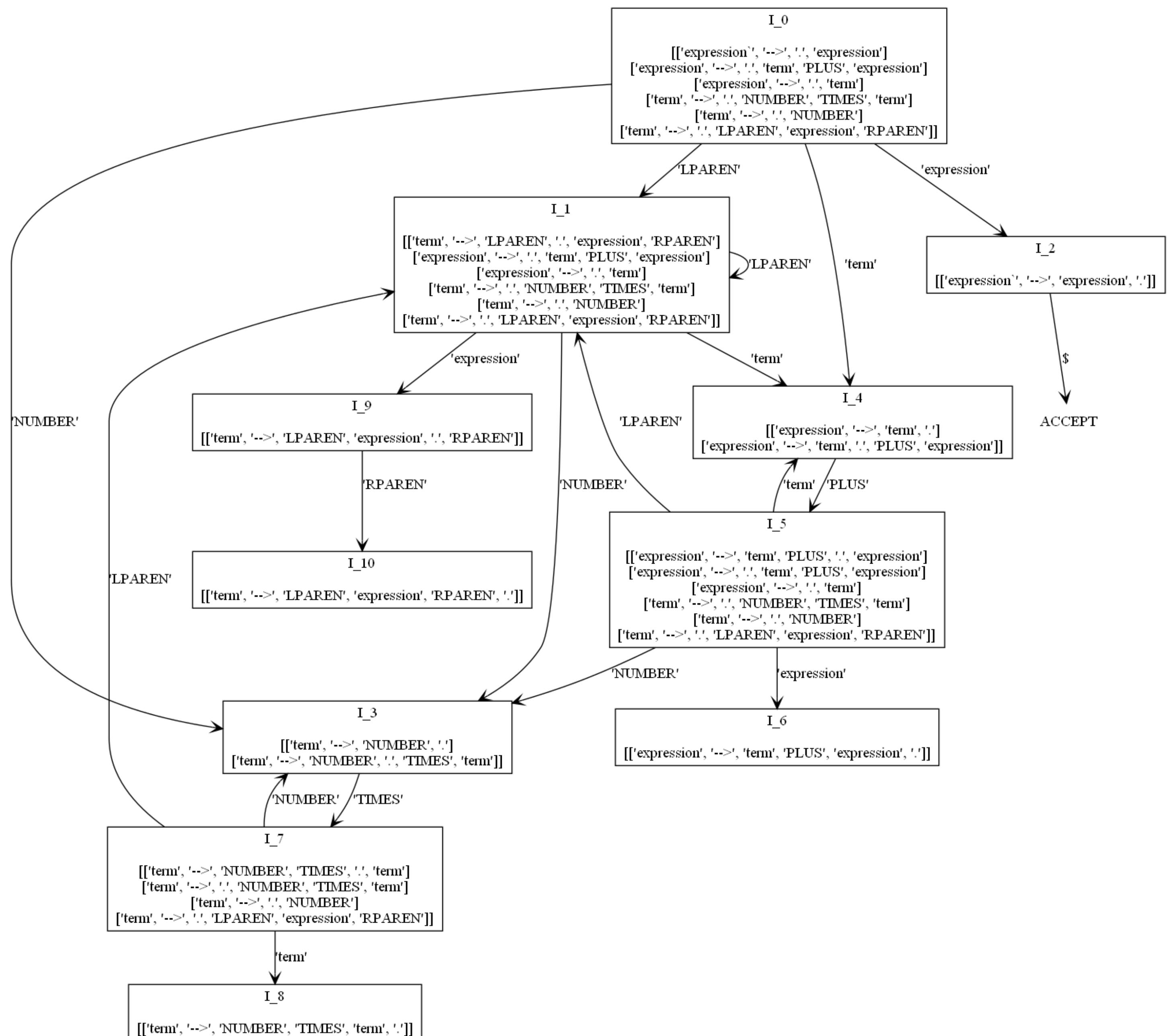
- Gramática aumentada obtenida

```
PS C:\Users\charl\Desktop\S12023\Diseño de Lenguajes\LaboratorioE_20498> & C:/Users/charl/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/charl/Desktop/S12023/Diseño de Lenguajes/LaboratorioE_20498/src/lr.py"
```

Gramatica aumentada para LR(0)

```
expression` --> expression
expression --> term PLUS expression
expression --> term
term --> NUMBER TIMES term
term --> NUMBER
term --> LPAREN expression RPAREN
```

- Representación visual del LR(0)



SLR-4

- Gramática aumentada obtenida

```
PS C:\Users\charl\Desktop\S12023\Diseño de Lenguajes\LaboratorioE_20498> & C:/Users/charl/AppData/Local/Programs/Python/Python39-32/python.exe "c:/Users/charl/Desktop/S12023/Diseño de Lenguajes/LaboratorioE_20498/src/lr.py"

----- Gramatica aumentada para LR(0) -----

p` --> p
p --> t
t --> m q
t --> m
q --> SEMICOLON m q
q --> SEMICOLON m
m --> a
a --> ID ASSIGNOP e
e --> x z
e --> x
z --> LT x
z --> EQ x
x --> r w
x --> r
w --> y w
w --> y
y --> PLUS r
y --> MINUS r
r --> f v
r --> f
v --> j v
v --> j
j --> TIMES f
j --> DIV f
f --> LPAREN e RPAREN
f --> NUMBER
f --> ID
```

- Representación visual del LR(0) (En la siguiente página)

Aclaraciones del código

Funciones CLOUSURE, GOTO, FIRST y FOLLOW

Para completar los requerimientos del laboratorio E se agregaron los siguientes elementos principales:

- Directorio LR(0)
- lr.py

Además de que se agregaron en el archivo utils.py que esta dentro de la carpeta Tools de SRC, las funciones requeridas del CLOUSURE, GOTO, FIRST y FOLLOW. A continuación, se agrega una captura de la programación de cada una.

```

def GOTO(items, simbol, dot_grammar):

    reached_u = []
    stack = []

    for element in items:
        stack.append(element)

    while len(stack) != 0:
        actual_state = stack.pop()
        if actual_state.index(".") == (len(actual_state)-1):
            pass
        else:
            if actual_state[actual_state.index(".") + 1] == simbol:
                if actual_state not in reached_u:
                    reached_u.append(MoveDot(actual_state))

    # print("Goto I")
    reached_u = CLOUSURE(reached_u, dot_grammar)
    # for x in reached_u:
    #     print(x)
    return reached_u

```

```

def CLOUSURE(items, dot_grammar):
    reached = []
    stack = []
    for production in items:
        stack.append(production)
        reached.append(production)

    while len(stack) != 0:
        actual_state = stack.pop()
        if actual_state.index(".") == (len(actual_state)-1):
            pass
        else:
            header = actual_state[actual_state.index(".") + 1]
            for production in dot_grammar:
                if header == production[0]:
                    if production not in reached:
                        reached.append(production)
                        stack.append(production)

    return reached

def MoveDot(arreglo_punto):
    otro = []
    for x in arreglo_punto:
        otro.append(x)

    for i in range(len(otro) - 1):
        if otro[i] == ".":
            otro[i], otro[i+1] = otro[i+1], otro[i]

    return otro

```

```

def FIRST(valor, grammar, terminals):

    final = []
    stack = []
    reached = []
    stack.append(valor)
    reached.append(valor)

    if valor in terminals:
        return stack
    else:
        while len(stack) != 0:
            evaluando = stack.pop(0)
            for production in grammar:
                if production[0] == evaluando:
                    if production[2] in terminals:
                        final.append(production[2])
                    else:
                        if production[2] not in stack and production[2] not in reached:
                            stack.append(production[2])
                            reached.append(production[2])

        return final

```

```

def FOLLOW(symbol, grammar, start_symbol, terminals):
    follow_set = set()
    if symbol == start_symbol:
        follow_set.add('$')
    for production in grammar:
        for i, s in enumerate(production[2:]):
            if s == symbol:
                if i == len(production[2:]) - 1:
                    if production[0] != symbol:
                        follow_set |= FOLLOW(production[0], grammar, start_symbol, terminals)
                else:
                    next_symbol = production[i+3]
                    if next_symbol in terminals:
                        follow_set.add(next_symbol)
                    else:
                        follow_set |= FIRST(next_symbol, grammar, terminals)
                        if '' in follow_set:
                            follow_set -= {''}
                            follow_set |= FOLLOW(production[0], grammar, start_symbol, terminals)

    return follow_set

```

El funcionamiento de estas funciones esta asociado directamente con la construcción del LR(0) respecto a su representación visual. En el código de lr.py estas funciones se utilizan de manera explicita, directamente en la parte de construcción. A continuación se agrega una ejecución simple de demostración de cómo se pueden llamar a estas funciones y sus resultados para comprobar que si funcionan, en este caso para la gramática y tokens relacionado al slr-1.

```
from Tools.utils import *

banner(f" Ejemplo CLOSURE con {str(['expression`, '-->', '.', 'expression'])}", False)
I = CLOSURE(['expression`, '-->', '.', 'expression'], dot_grammar)
for element in I:
    print(f"{element[0]} {' '.join(element[1:])}")

print()
banner(f" Ejemplo GOTO con 'expression' e I obtenido del CLOSURE anterior", False)
X = GOTO(I, 'expression', dot_grammar)
for element in X:
    print(f"{element[0]} {' '.join(element[1:])}")

print()
banner(" Ejemplo FIRST con 'term'", False)
Y = FIRST('term', grammar, terminals)
for element in Y:
    print(f"{element}")

print()
banner(" Ejemplo FOLLOW con 'term'", False)
Z = FOLLOW('term', grammar, start_symbol, terminals)
for element in Z:
    print(f"{element}")
```

PS C:\Users\charl\Desktop\S12023\Diseno de Lenguajes\LaboratorioE_20498> & C:/Users/charl/AppData/Local/Programs/Python/23/Diseno de Lenguajes/LaboratorioE_20498/src/prueba.py

———— Ejemplo CLOSURE con ['expression`, '-->', '.', 'expression'] ————

```
expression` --> . expression
expression --> . expression PLUS term
expression --> . term
term --> . term TIMES factor
term --> . factor
factor --> . LPAREN expression RPAREN
factor --> . ID
```

———— Ejemplo GOTO con 'expression' e I obtenido del CLOSURE anterior ————

```
expression --> expression . PLUS term
expression` --> expression .
```

———— Ejemplo FIRST con 'term' ————

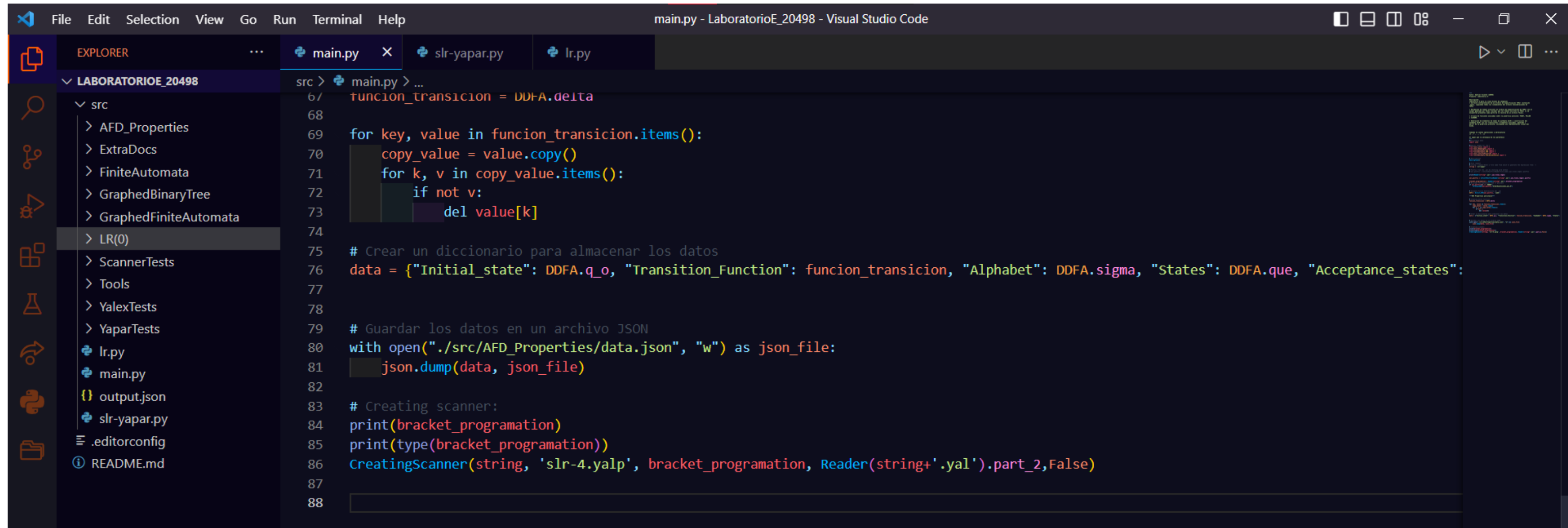
```
LPAREN
ID
```

———— Ejemplo FOLLOW con 'term' ————

```
RPAREN
TIMES
$
PLUS
```


Breve explicación del código lr.py y sus antecesores.

El proceso de ejecución inicia con main.py, al igual que en el laboratorio D, al momento en el que se ejecute main.py se creara un scanner a parir de .yal que tenga cómo parámetro la función “CreatingScanner” dentro de main.py. En este caso se creara un scanner que permite leer los .yalp de tal manera que identifique Tokens para su lectura de manera conveniente y rápida (Esto se realizó de esta manera debido a las recomendaciones que se dieron durante la clase).

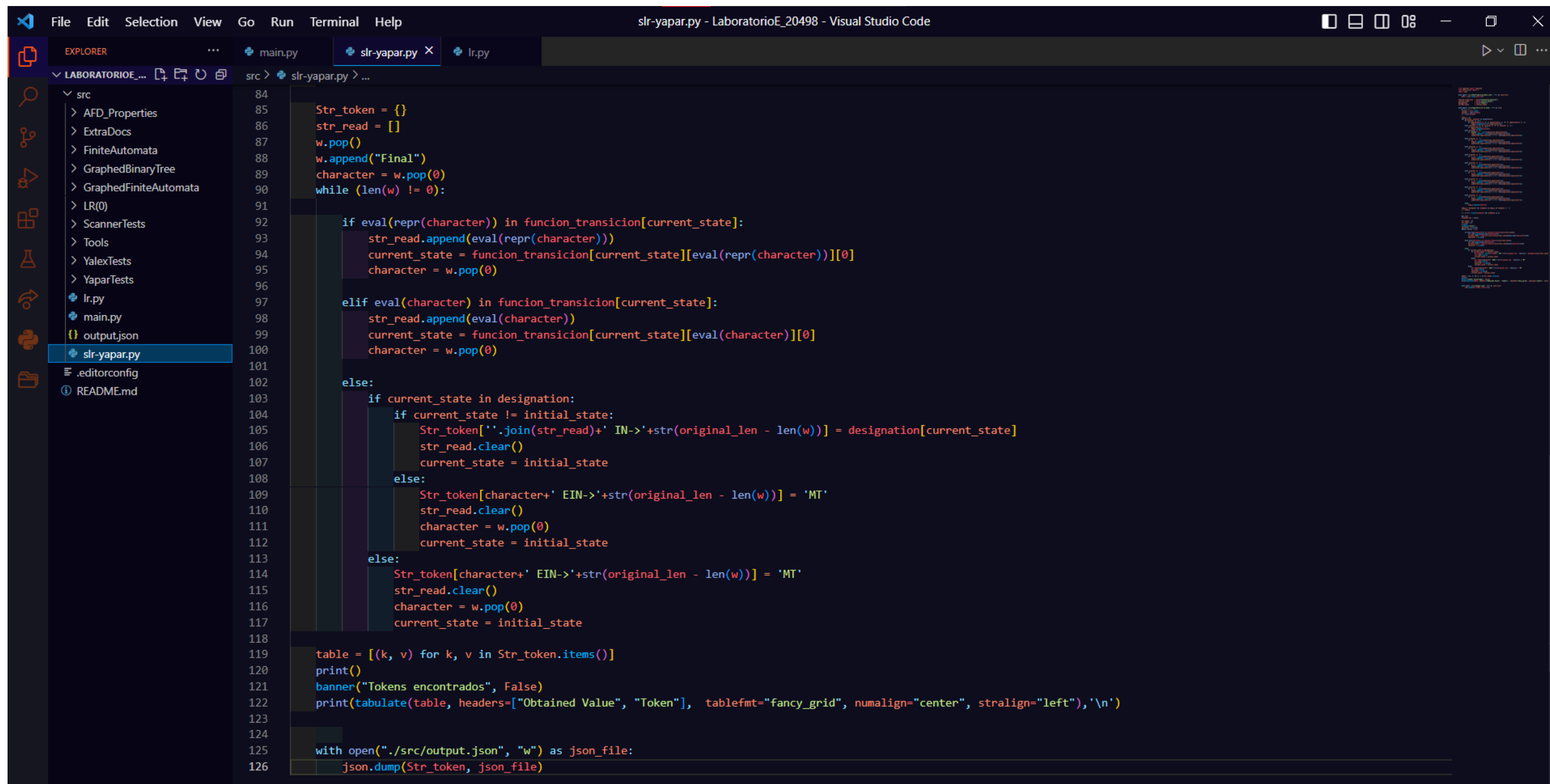


```

src > main.py > ...
67  funcion_transicion = DDFA.delta
68
69  for key, value in funcion_transicion.items():
70      copy_value = value.copy()
71      for k, v in copy_value.items():
72          if not v:
73              del value[k]
74
75  # Crear un diccionario para almacenar los datos
76  data = {"Initial_state": DDFA.q_o, "Transition_Function": funcion_transicion, "Alphabet": DDFA.sigma, "States": DDFA.que, "Acceptance_states":
77
78
79  # Guardar los datos en un archivo JSON
80  with open("../src/AFD_Properties/data.json", "w") as json_file:
81      json.dump(data, json_file)
82
83  # Creating scanner:
84  print(bracket_programation)
85  print(type(bracket_programation))
86  CreatingScanner(string, 'slr-4.yalp', bracket_programation, Reader(string+'.yal').part_2,False)
87
88

```

Luego de haber ejecutado main.py se debe de ejecutar el archivo generado que en este caso llevara el nombre de “slr-yapar.py”. Este básicamente es el scanner que leerá un archivo .yalp y apartir del yalex que se le asigno generará los TOKENS de dicho archivo y los almacenara en ouitput.json de tal manera que la tokenizacion se encuentre dentro del .json. Es importante aclarar que el archivo slr-yapar.py se actualizara dependiendo del yalex que se le pase como parámetro.



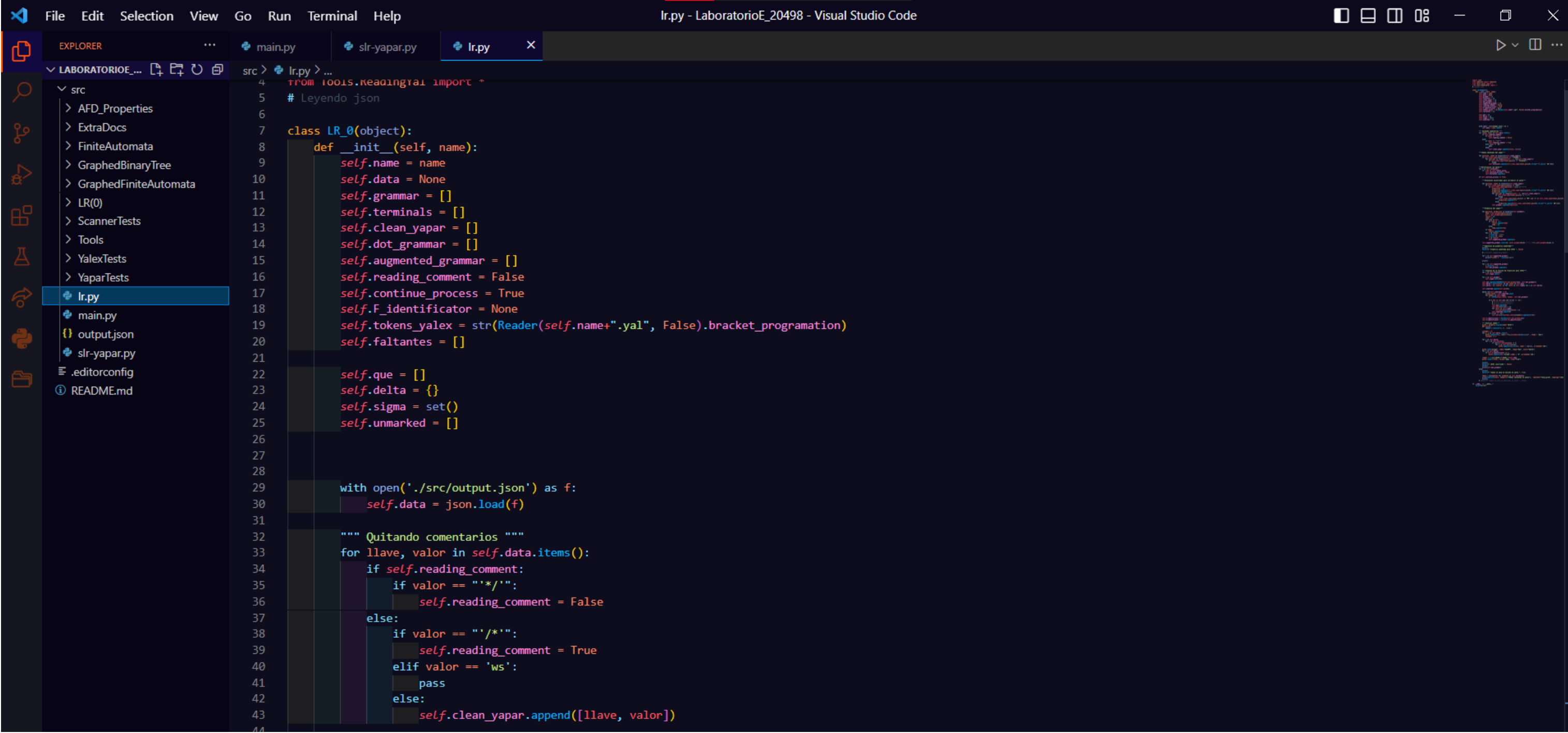
```

src > slr-yapar.py > ...
84
85  Str_token = {}
86  str_read = []
87  w.pop()
88  w.append("Final")
89  character = w.pop(0)
90  while (len(w) != 0):
91
92      if eval(repr(character)) in funcion_transicion[current_state]:
93          str_read.append(eval(repr(character)))
94          current_state = funcion_transicion[current_state][eval(repr(character))][0]
95          character = w.pop(0)
96
97      elif eval(character) in funcion_transicion[current_state]:
98          str_read.append(eval(character))
99          current_state = funcion_transicion[current_state][eval(character)][0]
100         character = w.pop(0)
101
102      else:
103          if current_state in designation:
104              if current_state != initial_state:
105                  Str_token[''.join(str_read)+' IN->'+str(original_len - len(w))] = designation[current_state]
106                  str_read.clear()
107                  current_state = initial_state
108              else:
109                  Str_token[character+' EIN->'+str(original_len - len(w))] = 'MT'
110                  str_read.clear()
111                  character = w.pop(0)
112                  current_state = initial_state
113              else:
114                  Str_token[character+' EIN->'+str(original_len - len(w))] = 'MT'
115                  str_read.clear()
116                  character = w.pop(0)
117                  current_state = initial_state
118
119  table = [(k, v) for k, v in Str_token.items()]
120  print()
121  banner("Tokens encontrados", False)
122  print(tabulate(table, headers=["Obtained Value", "Token"], tablefmt="fancy_grid", numalign="center", stralign="left"),'\n')
123
124
125  with open("../src/output.json", "w") as json_file:
126      json.dump(Str_token, json_file)

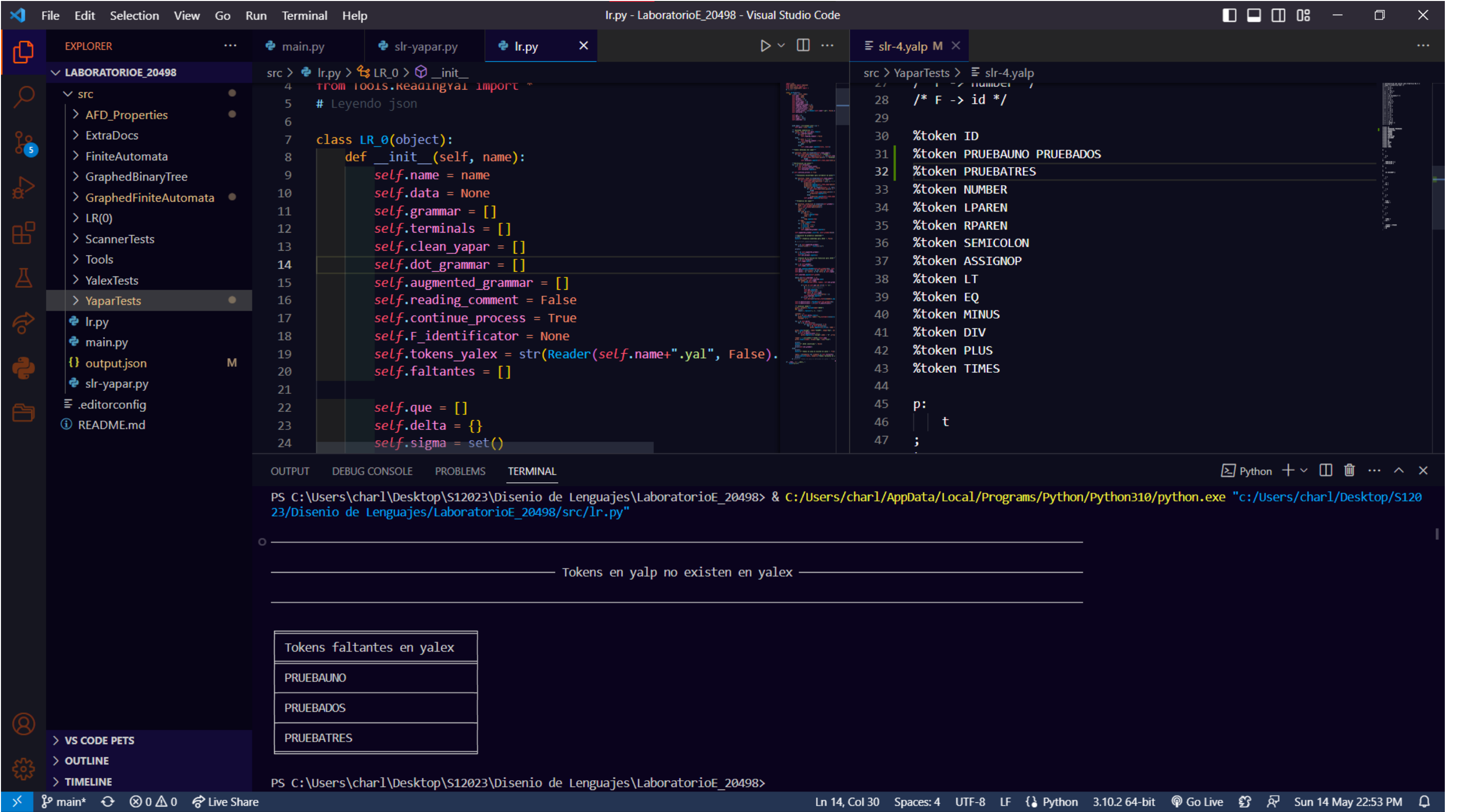
```

Una vez realizada la tokenizacion, se puede proceder a ejecutar lr.py que es el archivo principal de este laboratorio E. Ya que con la tokenizacion se puede obtener los tokens del archivo yapar, comprobar que estén dentro del yalex asociado y las producciones, que son esenciales para construir el LR(0) de cada slr.

Lr.py inicia ignorando los comentarios de la tokenizacion hasta encontrar la palabra reservada “%token”, cuando encuentre este, se agregaran todos los terminales que están después de esta palabra reservada obteniendo así todos los tokens del yapar.



Una vez obtenidos los terminales definidos con la palabra reservada “%token” dentro del yalp. Se procede a ver si estos están presentes dentro del yalex asociado. Si hay alguno que no se encuentre la construcción del LR(0) no se llevara a cabo y se termina aquí el programa. En la imagen anterior se puede ver el output de esta situación, en el que se modifíco momentáneamente los tokes asociados al yapar slr-4.



En el caso de que los tokens dentro del .yalp si estén dentro del .yal se procede con la construcción. Básicamente lo primero que se realiza es encontrar la gramática (incluyendo ors) identificando la serie de no terminal seguida de dos puntos y agregando todo lo que tenga luego de esta serie hasta llega al punto y coma que indica el inicio de otra producción.

Apartir de esta gramática se obtiene la aumentada y se quitan los ors, de tal manera que la gramática quede de una manera muchos mas legible. Cuando se ejecuta lr se puede ver cómo es que queda la gramática aumentada.

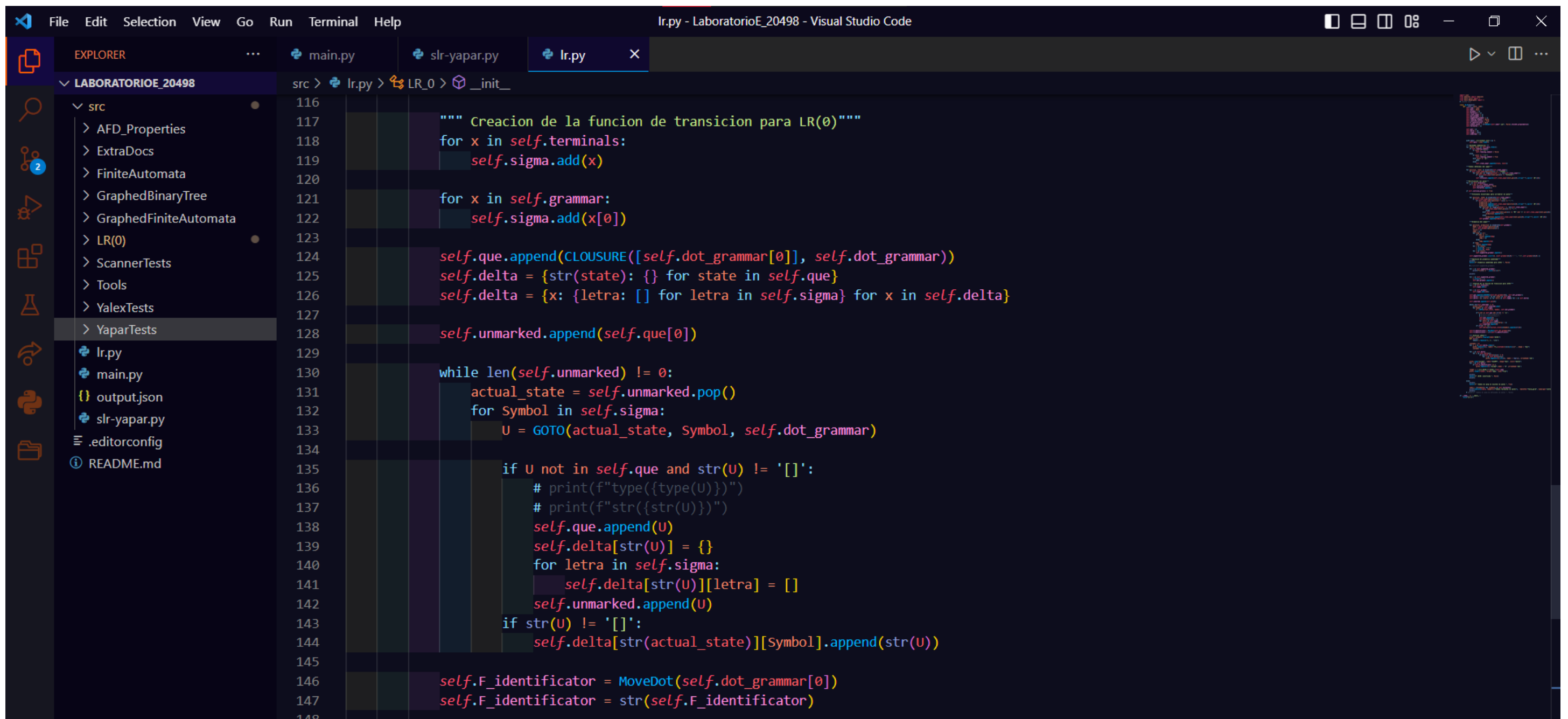
```

Gramatica aumentada para LR(0)

p` --> p
p --> t
t --> m q
t --> m
q --> SEMICOLON m q
q --> SEMICOLON m
m --> a
a --> ID ASSIGNOP e
e --> x z
e --> x
z --> LT x
z --> EQ x
x --> r w
x --> r
w --> y w
w --> y
y --> PLUS r
y --> MINUS r
r --> f v
r --> f
v --> j v
v --> j
j --> TIMES f
j --> DIV f
f --> LPAREN e RPAREN
f --> NUMBER
f --> ID

```

Luego de esto se procede a realizar la construcción del LR(0), se juntan los terminales y no terminales sin repetir para crear el alfabeto (transiciones que se pueden realizar de cada estado con el goto), se inicializa I como “estado inicial” siendo CLOSURE de la aumentada (CLOSURE(p` → p) en este caso) y luego inicia la construcción descrita en el libro del dragón. Mientras el estado a analizar sea un estado que no se ah analizado, se verifica su transición con GOTO(I_x, simbolo) siendo I_x el estado que esta siendo analziado y símbolo una letra del alfabeto por llamarlo de alguna forma, si esta nueva transición crea un nuevo estado este se agrega a los estados sin analizar y se repite el mismo proceso, de ya estar presente no se agrega a los estados a analizar y simplemente se agrega la transición.



```

116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
""" Creacion de la funcion de transicion para LR(0)"""
for x in self.terminals:
    self.sigma.add(x)

for x in self.grammar:
    self.sigma.add(x[0])

self.que.append(CLOSURE([self.dot_grammar[0]], self.dot_grammar))
self.delta = {str(state): {} for state in self.que}
self.delta = {x: {letra: [] for letra in self.sigma} for x in self.delta}

self.unmarked.append(self.que[0])

while len(self.unmarked) != 0:
    actual_state = self.unmarked.pop()
    for Symbol in self.sigma:
        U = GOTO(actual_state, Symbol, self.dot_grammar)

        if U not in self.que and str(U) != '[]':
            # print(f"type({type(U)})")
            # print(f"str({str(U)})")
            self.que.append(U)
            self.delta[str(U)] = {}
            for letra in self.sigma:
                self.delta[str(U)][letra] = []
            self.unmarked.append(U)

        if str(U) != '[]':
            self.delta[str(actual_state)][Symbol].append(str(U))

self.F_identificator = MoveDot(self.dot_grammar[0])
self.F_identificator = str(self.F_identificator)

```

Esta parte de lr.py crea el registro de las transiciones entre estados lo cual es la forma mas básica de representación del LR(0) siendo su tabla. Para poder visualizarlo de una manera más clara se itera sobre esta tabla de tal manera que se crea un nodo por cada estado y cuando se encuentra una transición esta mediante la librería graphviz se grafica de tal forma que tiene dirección de un estado a otro y tiene como titulo la transición que hace este movimiento posible.

La parte de lr.py que hace esto posible corresponde a la siguiente.



```

""" Graficar LR(0)"""
grafo = graphviz.Digraph(name="LR(0)")
def ats(s):
    return s.replace(']', [' ', '\n['])

contador = 0
for x, y in self.delta.items():
    grafo.node(ats(x), label= f"I_{contador}\n\n{ats(x)}" , shape = "box")
    contador += 1

for x in self.delta:
    for y in self.delta[x]:
        if len(self.delta[x][y]) != 0:
            for w in self.delta[x][y]:
                grafo.edge(ats(x),ats(w), label = repr(y), arrowhead='vee')

grafo.node("accept", label="ACCEPT", shape="box", color="white")
for x in self.delta:
    if self.F_identificator in x:
        grafo.edge(ats(x),"accept",label = "$" ,arrowhead='vee')

render = "./src/LR(0)/"+"LR(0)_"+self.name
grafo.render(render, format="png", view="True")

print()
banner(f" LR(0) construido ", False)
print()

```

Esto de manera breve consistiría en lr.py y en este se puede apreciar el uso de las funciones previamente mencionadas logrando la construcción del LR(0) para los 4 archivos .yalp y sus yalex asociados como se solicitaba. En las primeras cinco paginas de este documento se encuentra la evidencia y de igual forma, dentro del .zip subido en la entrega en la ruta ./src/LR(0) se encuentran las imágenes .png de cada slr.