Manual Técnico

UNIVERSIDAD
DE ANTIOQUIA
1803

Docente:

Roberto Florez Rueda y
laboratorio

Curso:

Teoría de lenguajes y
laboratorio

Por:

Laura Vanessa Tascón

Ingeniería de Sistemas

Facultad de Ingeniería

Universidad de
Antioquia
2022

Objetivos

Se ha creado dicho documento con el propósito de mostrar cómo fue diseñado el sistema para la práctica l de Teoría de lenguajes y laboratorio, cuya necesidad es la de reconocer y validar cadenas de texto en base a expresiones regulares, al mismo tiempo este documento da referencias de como interactuar con el programa en caso de ser actualizado, modificado o como dar mantenimiento en caso de fallo.

Objetivos específicos

- Guía de instalación del sistema
- Requerimientos para la ejecución
- Diagramas

Alcance

Este documento está dirigido a cualquier programador con conocimientos básicos en Python y en el manejo de expresiones regulares para la creación de Autómatas finitos determinísticos

Requerimientos técnicos

Software

- Python 3
- Algún editor de texto como Visual Studio Code, Sublime Text, Nano o cualquier otro que sirva para correr código Python
- Windows

Hardware

- Computadora, sea portátil o de escritorio (mouse, teclado, cpu, monitor)

Requerimientos mínimos de hardware

- Teclado, pantalla, cpu, mouse

Requerimientos mínimos de software

- Privilegios de administrador NO
- Sistema operativo Windows xp 7

Herramientas utilizadas para el desarrollo

- Interfaz gráfica con Tkinter para hacer más ameno el manejo de la aplicación
- Un editor de texto para Python, en este caso se uso Visual Studio Code

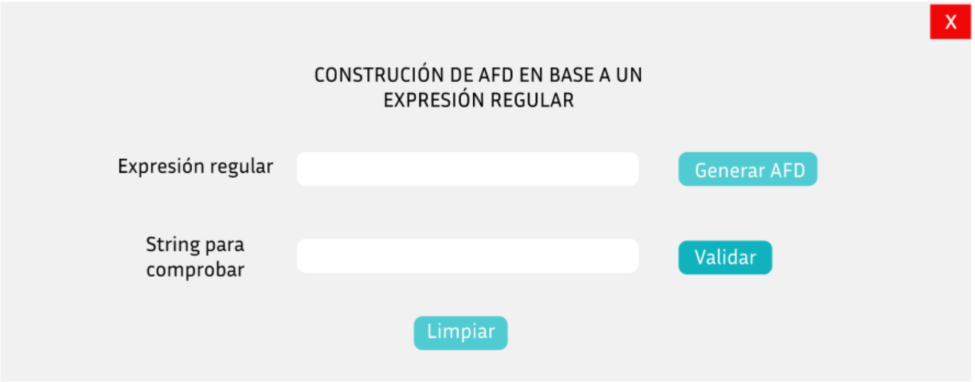
Instalación

Para poder correr el programa debe instalar Python 3 para poder compilar el código.

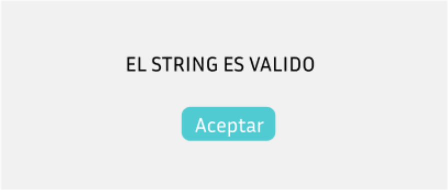
Análisis

- **Propósito**
  El propósito de este proyecto es poder validar strings pertenecientes a una expresión regular por medio de un autómata finito determinístico basado en la expresión regular.

- **Historias de usuario y mockup**
  https://docs.google.com/spreadsheets/d/1GVAYT_StjnFkaTWQaQSnclAtKP1nI3P3/edit?usp=sharing&ouid=102312595000450157281&rtpof=true&sd=true

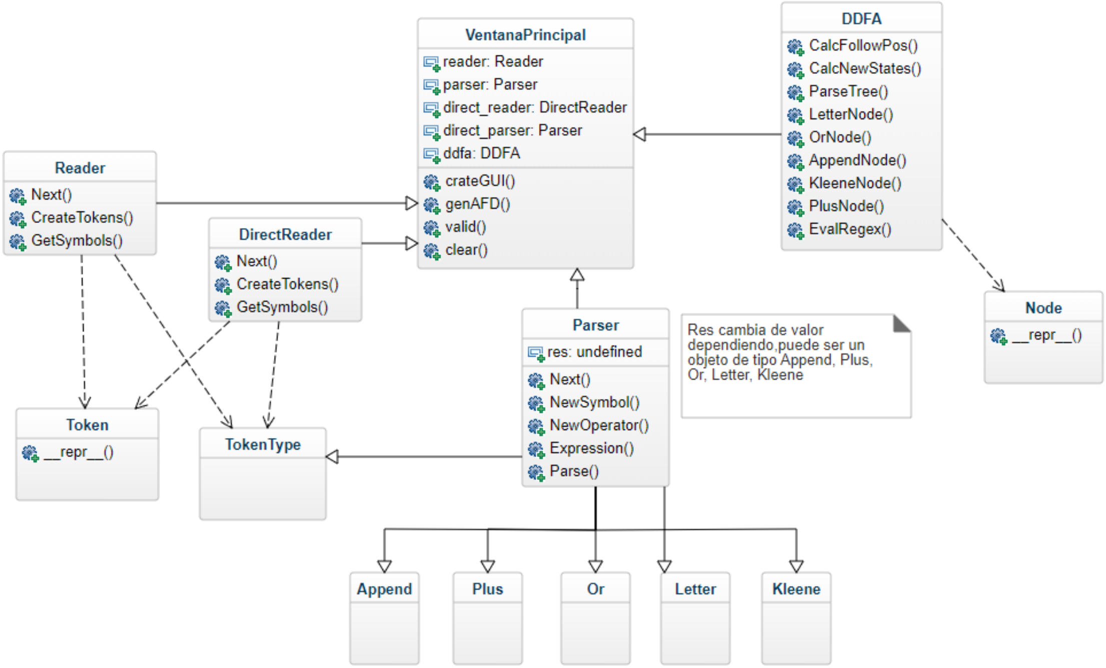


CONSTRUCIÓN DE AFD EN BASE A UN EXPRESIÓN REGULAR



EL STRING ES VALIDO

- ### Diagrama de clases

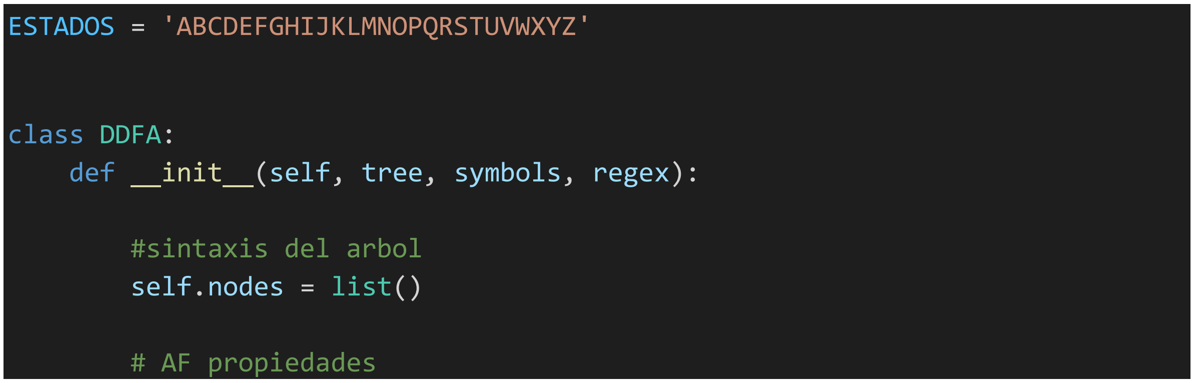


Configuración

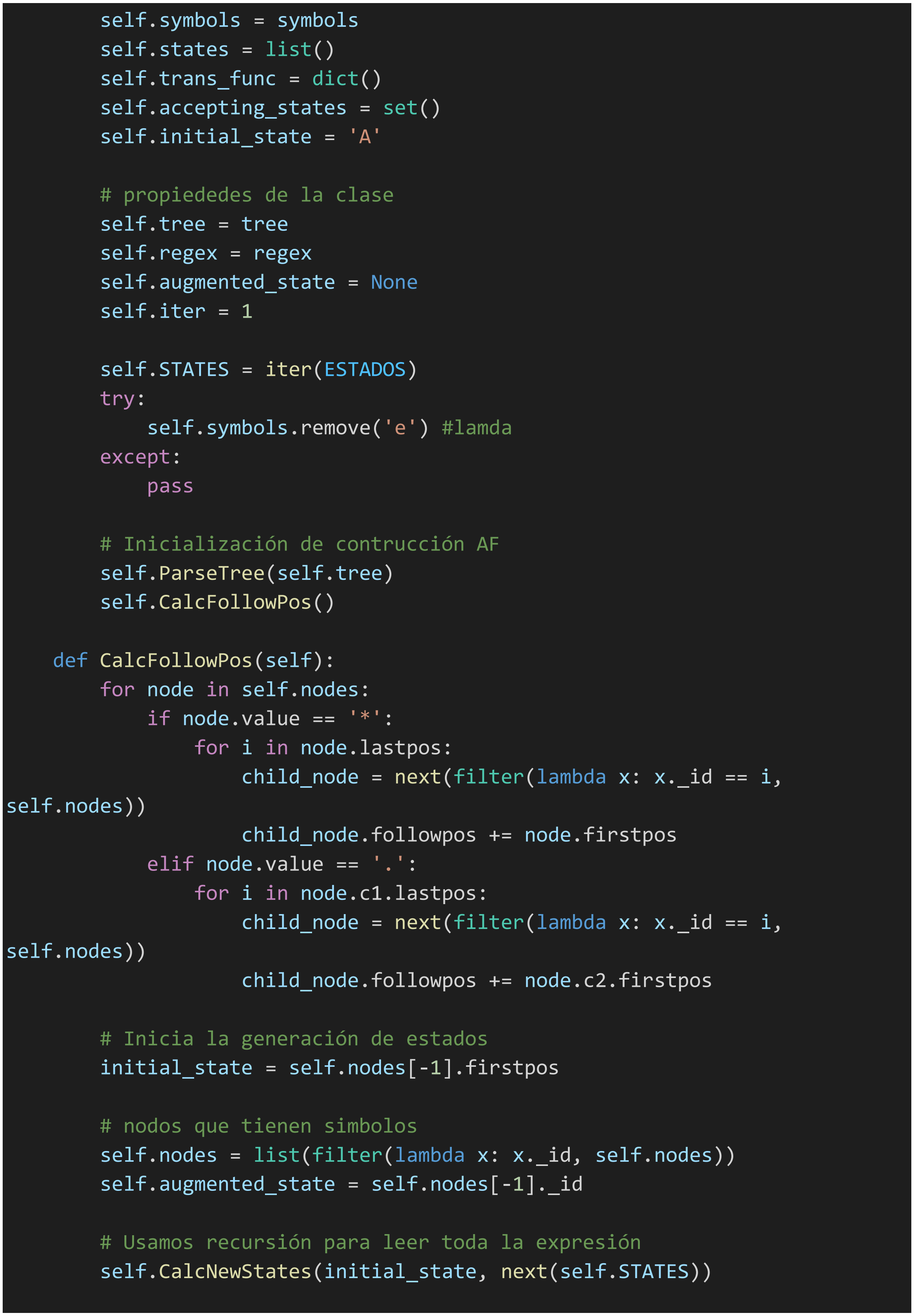
No es necesaria una configuración del sistema

Diseño de arquitectura

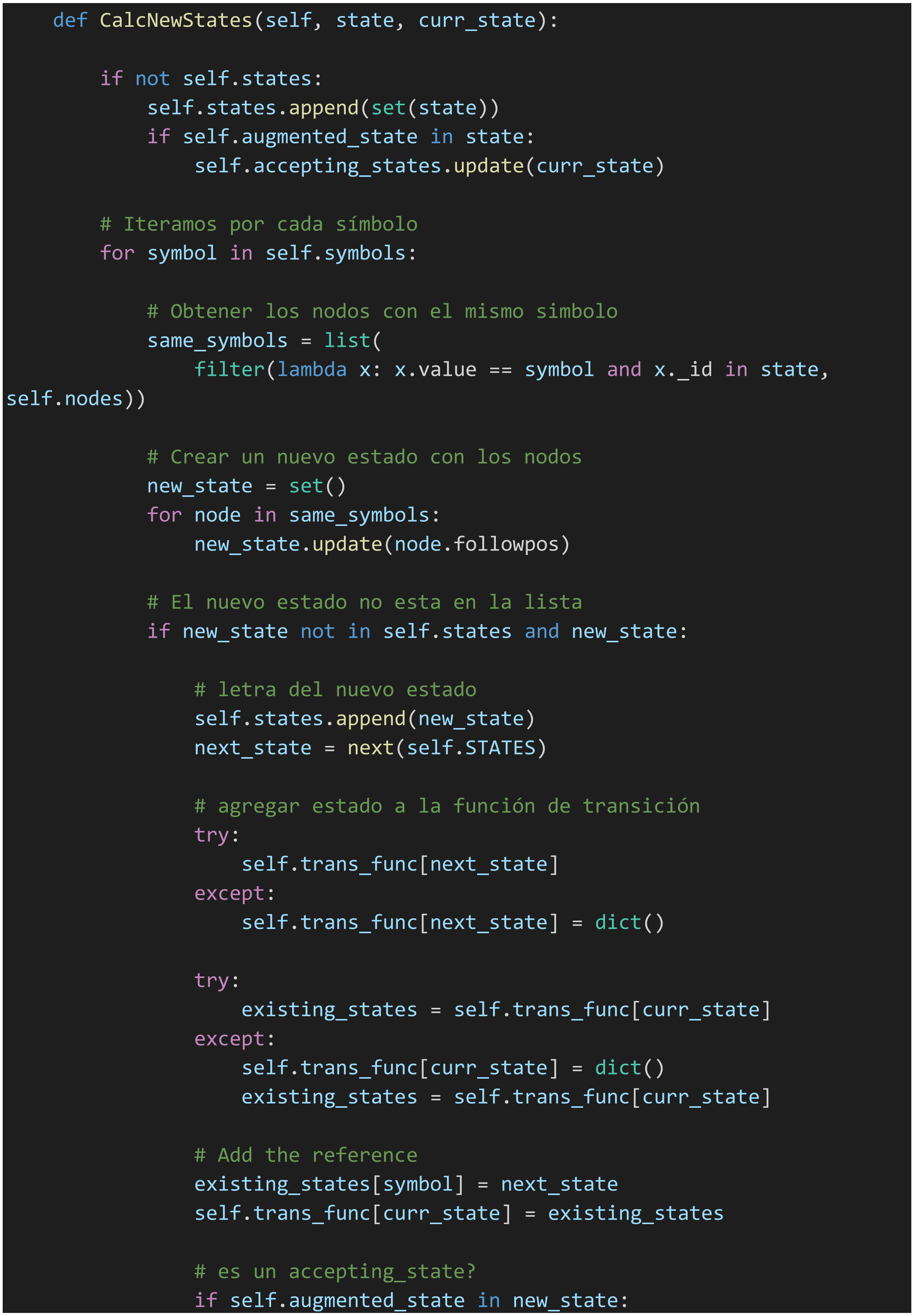
- ### Desarrollo
  ### afd



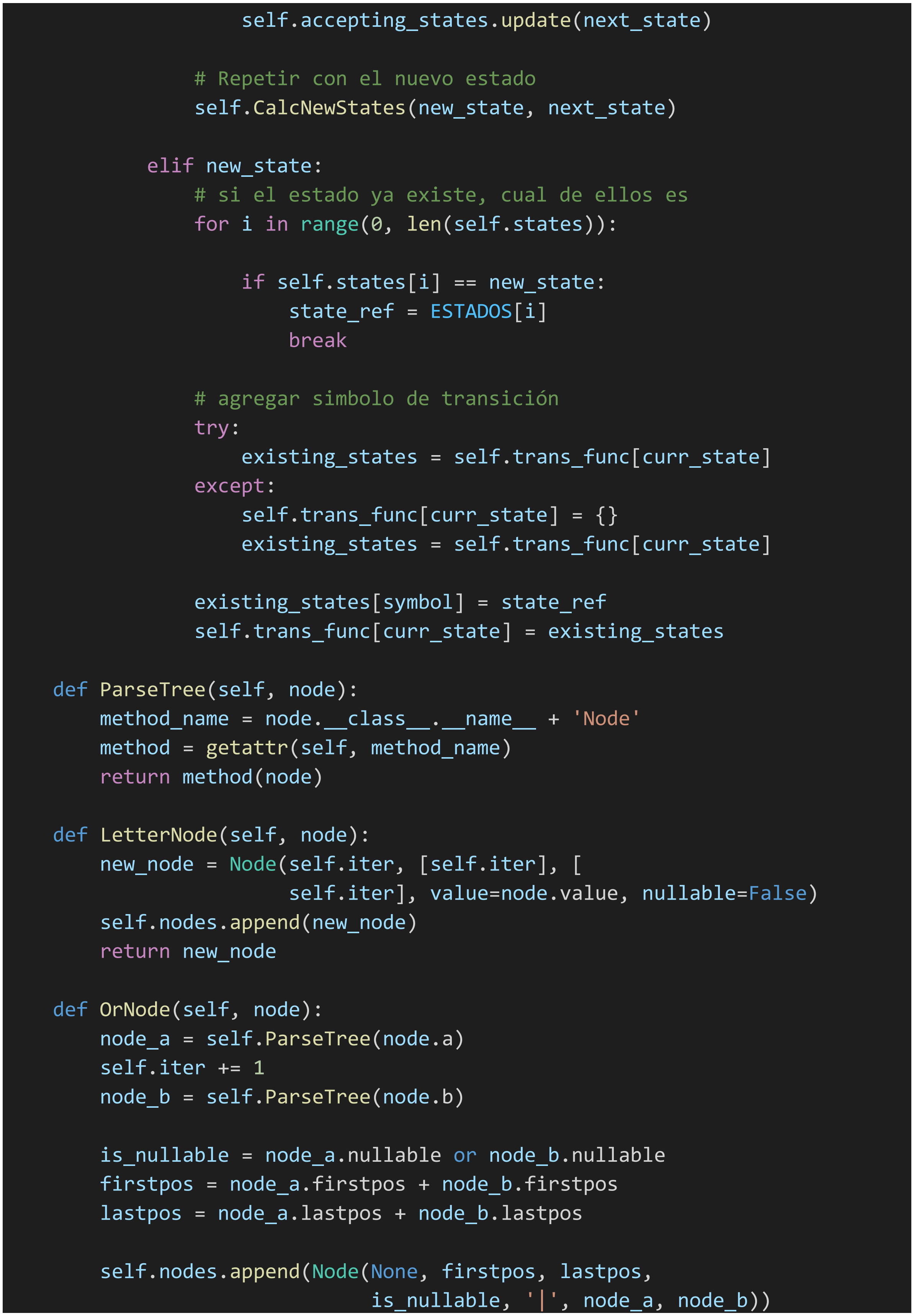
```
ESTADOS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'


class DDFA:
    def __init__(self, tree, symbols, regex):

        #sintaxis del arbol
        self.nodes = list()

        # AF propiedades
```



```
        self.symbols = symbols
        self.states = list()
        self.trans_func = dict()
        self.accepting_states = set()
        self.initial_state = 'A'

        # propiededes de la clase
        self.tree = tree
        self.regex = regex
        self.augmented_state = None
        self.iter = 1

        self.STATES = iter(ESTADOS)
        try:
            self.symbols.remove('e') #lamda
        except:
            pass

        # Inicialización de contrucción AF
        self.ParseTree(self.tree)
        self.CalcFollowPos()

    def CalcFollowPos(self):
        for node in self.nodes:
            if node.value == '*':
                for i in node.lastpos:
                    child_node = next(filter(lambda x: x._id == i,
self.nodes))
                    child_node.followpos += node.firstpos
            elif node.value == '.':
                for i in node.c1.lastpos:
                    child_node = next(filter(lambda x: x._id == i,
self.nodes))
                    child_node.followpos += node.c2.firstpos

        # Inicia la generación de estados
        initial_state = self.nodes[-1].firstpos

        # nodos que tienen simbolos
        self.nodes = list(filter(lambda x: x._id, self.nodes))
        self.augmented_state = self.nodes[-1]._id

        # Usamos recursión para leer toda la expresión
        self.CalcNewStates(initial_state, next(self.STATES))
```



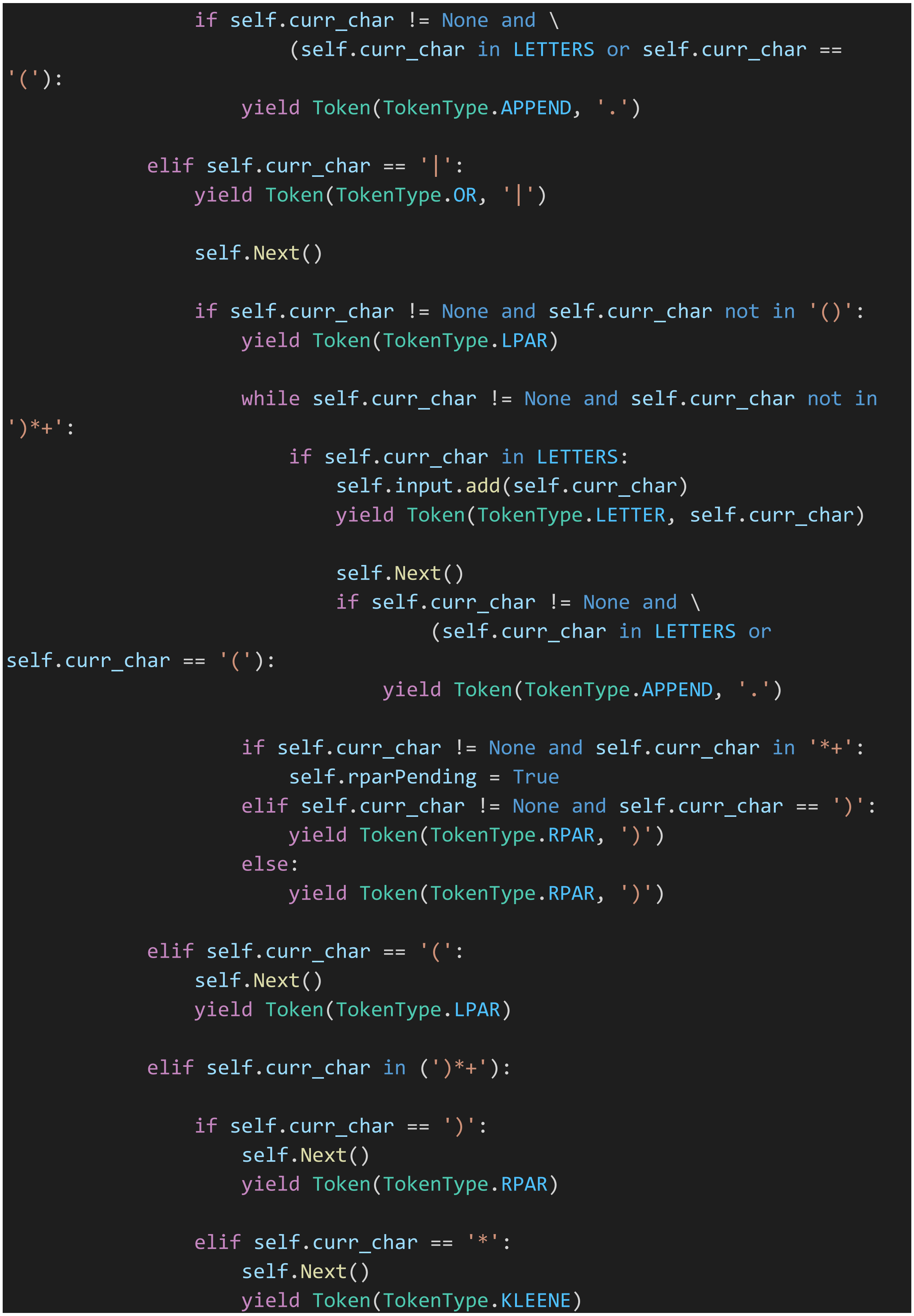
```
    def CalcNewStates(self, state, curr_state):

        if not self.states:
            self.states.append(set(state))
            if self.augmented_state in state:
                self.accepting_states.update(curr_state)

        # Iteramos por cada símbolo
        for symbol in self.symbols:

            # Obtener los nodos con el mismo simbolo
            same_symbols = list(
                filter(lambda x: x.value == symbol and x._id in state,
self.nodes))

            # Crear un nuevo estado con los nodos
            new_state = set()
            for node in same_symbols:
                new_state.update(node.followpos)

            # El nuevo estado no esta en la lista
            if new_state not in self.states and new_state:

                # letra del nuevo estado
                self.states.append(new_state)
                next_state = next(self.STATES)

                # agregar estado a la función de transición
                try:
                    self.trans_func[next_state]
                except:
                    self.trans_func[next_state] = dict()

                try:
                    existing_states = self.trans_func[curr_state]
                except:
                    self.trans_func[curr_state] = dict()
                    existing_states = self.trans_func[curr_state]

                # Add the reference
                existing_states[symbol] = next_state
                self.trans_func[curr_state] = existing_states

                # es un accepting_state?
                if self.augmented_state in new_state:
```



```
                    self.accepting_states.update(next_state)

                # Repetir con el nuevo estado
                self.CalcNewStates(new_state, next_state)

            elif new_state:
                # si el estado ya existe, cual de ellos es
                for i in range(0, len(self.states)):

                    if self.states[i] == new_state:
                        state_ref = ESTADOS[i]
                        break

                # agregar simbolo de transición
                try:
                    existing_states = self.trans_func[curr_state]
                except:
                    self.trans_func[curr_state] = {}
                    existing_states = self.trans_func[curr_state]

                existing_states[symbol] = state_ref
                self.trans_func[curr_state] = existing_states

    def ParseTree(self, node):
        method_name = node.__class__.__name__ + 'Node'
        method = getattr(self, method_name)
        return method(node)

    def LetterNode(self, node):
        new_node = Node(self.iter, [self.iter], [
                        self.iter], value=node.value, nullable=False)
        self.nodes.append(new_node)
        return new_node

    def OrNode(self, node):
        node_a = self.ParseTree(node.a)
        self.iter += 1
        node_b = self.ParseTree(node.b)

        is_nullable = node_a.nullable or node_b.nullable
        firstpos = node_a.firstpos + node_b.firstpos
        lastpos = node_a.lastpos + node_b.lastpos

        self.nodes.append(Node(None, firstpos, lastpos,
                               is_nullable, '|', node_a, node_b))
        return Node(None, firstpos, lastpos, is_nullable, '|', node_a,
node_b)

    def AppendNode(self, node):
        node_a = self.ParseTree(node.a)
        self.iter += 1
        node_b = self.ParseTree(node.b)

        is_nullable = node_a.nullable and node_b.nullable
        if node_a.nullable:
            firstpos = node_a.firstpos + node_b.firstpos
        else:
            firstpos = node_a.firstpos

        if node_b.nullable:
            lastpos = node_b.lastpos + node_a.lastpos
        else:
            lastpos = node_b.lastpos

        self.nodes.append(
            Node(None, firstpos, lastpos, is_nullable, '.', node_a, node_b))

        return Node(None, firstpos, lastpos, is_nullable, '.', node_a,
node_b)

    def KleeneNode(self, node):
        node_a = self.ParseTree(node.a)
        firstpos = node_a.firstpos
        lastpos = node_a.lastpos
        self.nodes.append(Node(None, firstpos, lastpos, True, '*', node_a))
        return Node(None, firstpos, lastpos, True, '*', node_a)

    def PlusNode(self, node):
        node_a = self.ParseTree(node.a)

        self.iter += 1

        node_b = self.KleeneNode(node)

        is_nullable = node_a.nullable and node_b.nullable
        if node_a.nullable:
            firstpos = node_a.firstpos + node_b.firstpos
        else:
            firstpos = node_a.firstpos
        if node_b.nullable:
            lastpos = node_b.lastpos + node_a.lastpos
        else:
            lastpos = node_b.lastpos

        self.nodes.append(
            Node(None, firstpos, lastpos, is_nullable, '.', node_a, node_b))

        return Node(None, firstpos, lastpos, is_nullable, '.', node_a, node_b)



    def EvalRegex(self):
        curr_state = 'A'
        for symbol in self.regex:

            if not symbol in self.symbols:
                return 'No'

            try:
                curr_state = self.trans_func[curr_state][symbol]
            except:
                if curr_state in self.accepting_states and symbol in self.trans_func['A']:
                    curr_state = self.trans_func['A'][symbol]
                else:
                    return 'No'

        return 'Yes' if curr_state in self.accepting_states else 'No'




class Node:
    def __init__(self, _id, firstpos=None, lastpos=None, nullable=False, value=None, c1=None, c2=None):
        self._id = _id
        self.firstpos = firstpos
        self.lastpos = lastpos
        self.followpos = list()
        self.nullable = nullable
        self.value = value
        self.c1 = c1
        self.c2 = c2
    def __repr__(self):
        return f'''
    id: {self._id}
    value: {self.value}
    firstpos: {self.firstpos}
    lastpos: {self.lastpos}
    followpos: {self.followpos}
    nullabe: {self.nullable}
    '''
```

direct_reader

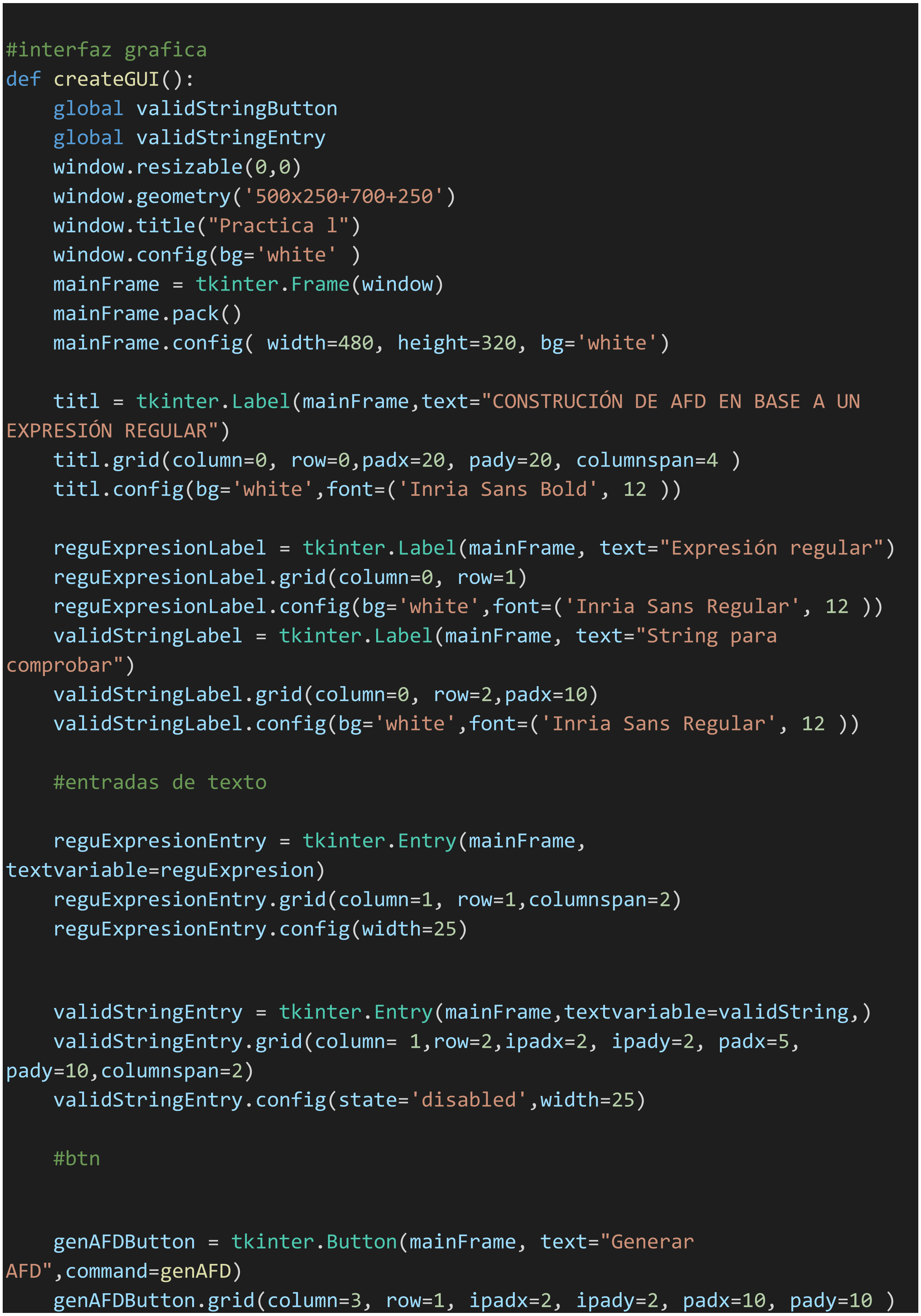
```
from tokens import Token, TokenType

LETTERS = 'abcdefghijklmnopqrstuvwxyz01234567890.'


class DirectReader:

    def __init__(self, string: str):
        self.string = iter(string.replace(' ', ''))
        self.input = set()
        self.rparPending = False
        self.Next()

    def Next(self):
        try:
            self.curr_char = next(self.string)
        except StopIteration:
            self.curr_char = None

    def CreateTokens(self):
        while self.curr_char != None:

            if self.curr_char in LETTERS:
                self.input.add(self.curr_char)
                yield Token(TokenType.LETTER, self.curr_char)

                self.Next()

                # para finalizar, se verifica si necesitamos agregar un
token append
```



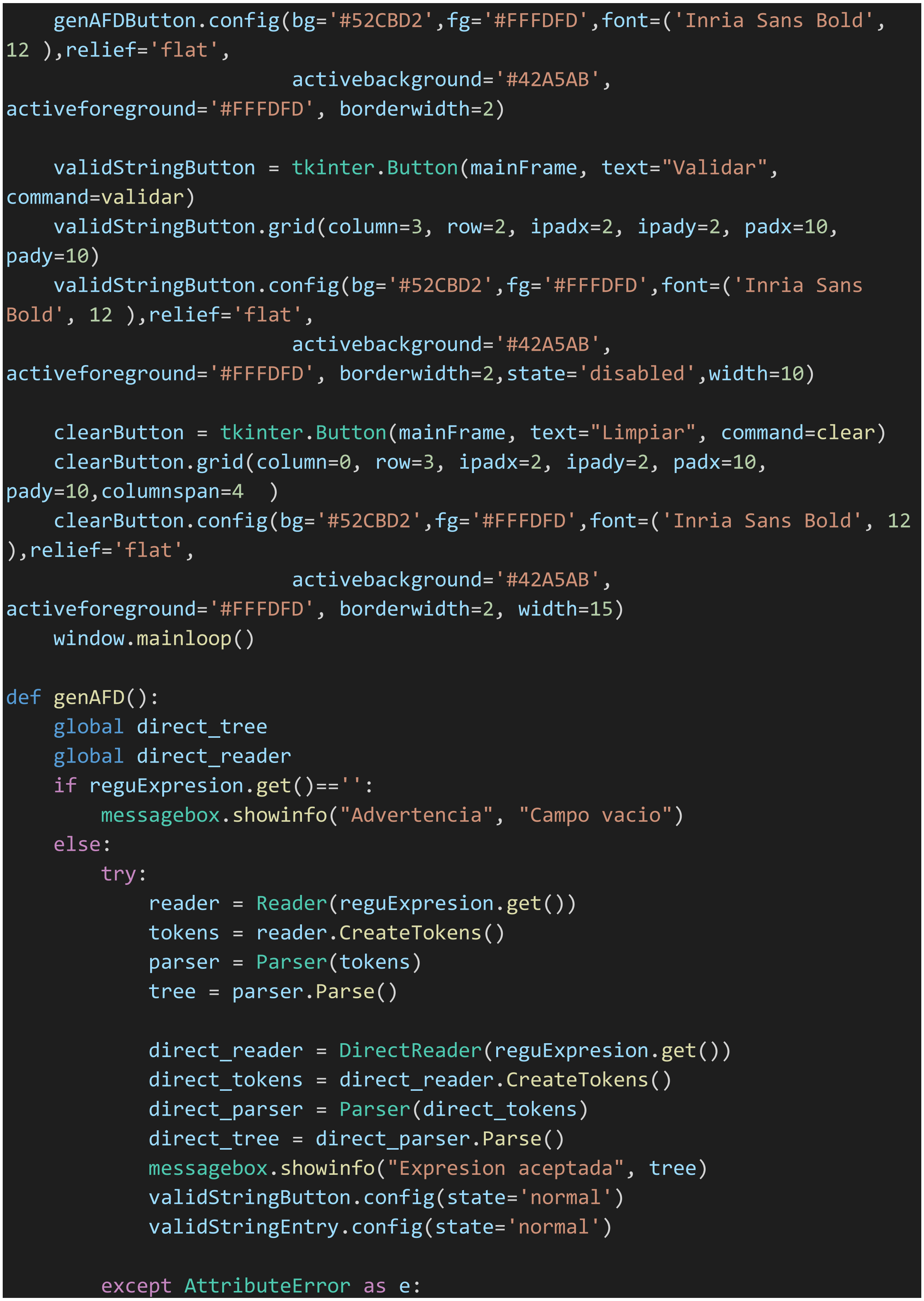
```
                if self.curr_char != None and \
                        (self.curr_char in LETTERS or self.curr_char ==
'('):
                    yield Token(TokenType.APPEND, '.')

            elif self.curr_char == '|':
                yield Token(TokenType.OR, '|')

                self.Next()

                if self.curr_char != None and self.curr_char not in '()':
                    yield Token(TokenType.LPAR)

                    while self.curr_char != None and self.curr_char not in
')*+':
                        if self.curr_char in LETTERS:
                            self.input.add(self.curr_char)
                            yield Token(TokenType.LETTER, self.curr_char)

                            self.Next()
                            if self.curr_char != None and \
                                    (self.curr_char in LETTERS or
self.curr_char == '('):
                                yield Token(TokenType.APPEND, '.')

                    if self.curr_char != None and self.curr_char in '*+':
                        self.rparPending = True
                    elif self.curr_char != None and self.curr_char == ')':
                        yield Token(TokenType.RPAR, ')')
                    else:
                        yield Token(TokenType.RPAR, ')')

            elif self.curr_char == '(':
                self.Next()
                yield Token(TokenType.LPAR)

            elif self.curr_char in (')*+'):

                if self.curr_char == ')':
                    self.Next()
                    yield Token(TokenType.RPAR)

                elif self.curr_char == '*':
                    self.Next()
                    yield Token(TokenType.KLEENE)
                elif self.curr_char == '+':
                    self.Next()
                    yield Token(TokenType.PLUS)


                if self.rparPending:
                    yield Token(TokenType.RPAR)
                    self.rparPending = False

                #para finalizar, se verifica si necesitamos agregar un token
append
                if self.curr_char != None and \
                        (self.curr_char in LETTERS or self.curr_char ==
'('):
                    yield Token(TokenType.APPEND, '.')

            else:
                raise Exception(f'Invalid entry: {self.curr_char}')

        yield Token(TokenType.APPEND, '.')
        yield Token(TokenType.LETTER, '#')

    def GetSymbols(self):
        return self.input
```

main

```
from asyncio.windows_events import NULL
from pickle import FALSE
from reader import Reader
from parsing import Parser
from afd import DDFA
from direct_reader import DirectReader
from cgitb import text
from tkinter import CENTER, messagebox, ttk as ttk
import tkinter
from tkinter import StringVar, font
from turtle import title

#var global
window = tkinter.Tk()
reguExpresion = StringVar()
validString = StringVar()
```



```
#interfaz grafica
def createGUI():
    global validStringButton
    global validStringEntry
    window.resizable(0,0)
    window.geometry('500x250+700+250')
    window.title("Practica l")
    window.config(bg='white' )
    mainFrame = tkinter.Frame(window)
    mainFrame.pack()
    mainFrame.config( width=480, height=320, bg='white')

    titl = tkinter.Label(mainFrame,text="CONSTRUCIÓN DE AFD EN BASE A UN
EXPRESIÓN REGULAR")
    titl.grid(column=0, row=0,padx=20, pady=20, columnspan=4 )
    titl.config(bg='white',font=('Inria Sans Bold', 12 ))

    reguExpresionLabel = tkinter.Label(mainFrame, text="Expresión regular")
    reguExpresionLabel.grid(column=0, row=1)
    reguExpresionLabel.config(bg='white',font=('Inria Sans Regular', 12 ))
    validStringLabel = tkinter.Label(mainFrame, text="String para
comprobar")
    validStringLabel.grid(column=0, row=2,padx=10)
    validStringLabel.config(bg='white',font=('Inria Sans Regular', 12 ))

    #entradas de texto

    reguExpresionEntry = tkinter.Entry(mainFrame,
textvariable=reguExpresion)
    reguExpresionEntry.grid(column=1, row=1,columnspan=2)
    reguExpresionEntry.config(width=25)


    validStringEntry = tkinter.Entry(mainFrame,textvariable=validString,)
    validStringEntry.grid(column= 1,row=2,ipadx=2, ipady=2, padx=5,
pady=10,columnspan=2)
    validStringEntry.config(state='disabled',width=25)

    #btn


    genAFDButton = tkinter.Button(mainFrame, text="Generar
AFD",command=genAFD)
    genAFDButton.grid(column=3, row=1, ipadx=2, ipady=2, padx=10, pady=10 )
```



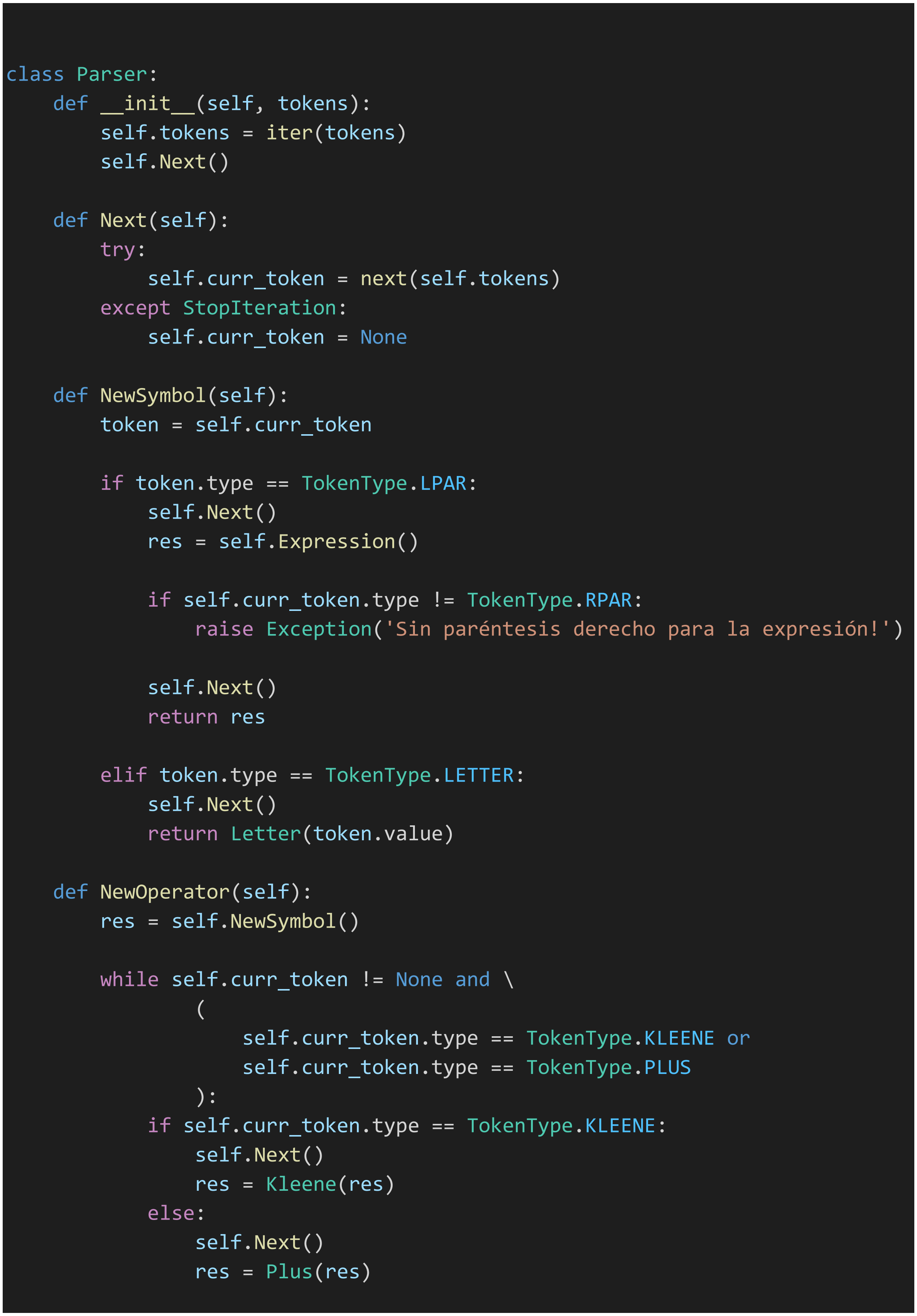
```
    genAFDButton.config(bg='#52CBD2',fg='#FFFDFD',font=('Inria Sans Bold',
12 ),relief='flat',
                        activebackground='#42A5AB',
activeforeground='#FFFDFD', borderwidth=2)

    validStringButton = tkinter.Button(mainFrame, text="Validar",
command=validar)
    validStringButton.grid(column=3, row=2, ipadx=2, ipady=2, padx=10,
pady=10)
    validStringButton.config(bg='#52CBD2',fg='#FFFDFD',font=('Inria Sans
Bold', 12 ),relief='flat',
                        activebackground='#42A5AB',
activeforeground='#FFFDFD', borderwidth=2,state='disabled',width=10)

    clearButton = tkinter.Button(mainFrame, text="Limpiar", command=clear)
    clearButton.grid(column=0, row=3, ipadx=2, ipady=2, padx=10,
pady=10,columnspan=4  )
    clearButton.config(bg='#52CBD2',fg='#FFFDFD',font=('Inria Sans Bold', 12
),relief='flat',
                        activebackground='#42A5AB',
activeforeground='#FFFDFD', borderwidth=2, width=15)
    window.mainloop()

def genAFD():
    global direct_tree
    global direct_reader
    if reguExpresion.get()=='':
        messagebox.showinfo("Advertencia", "Campo vacio")
    else:
        try:
            reader = Reader(reguExpresion.get())
            tokens = reader.CreateTokens()
            parser = Parser(tokens)
            tree = parser.Parse()

            direct_reader = DirectReader(reguExpresion.get())
            direct_tokens = direct_reader.CreateTokens()
            direct_parser = Parser(direct_tokens)
            direct_tree = direct_parser.Parse()
            messagebox.showinfo("Expresion aceptada", tree)
            validStringButton.config(state='normal')
            validStringEntry.config(state='normal')

        except AttributeError as e:
            messagebox.showinfo("ERROR:", "Expresión invalida (missing
parenthesis)")

        except Exception as e:
            messagebox.showinfo("ERRPR: ",e)
        pass

def validar():
    if reguExpresion.get()=='':
        messagebox.showinfo("Advertencia", "Campo vacio")
    else:
        ddfa = DDFA(direct_tree, direct_reader.GetSymbols(),
validString.get())
        ddfa_regex = ddfa.EvalRegex()
        messagebox.showinfo("Pertenece la cadena a la expresión
regular?",ddfa_regex)
        validString.set("")
        pass

def clear():
    reguExpresion.set("")
    validString.set("")

    pass

if __name__ == "__main__":
    createGUI()
```

nodes

```
#tipos de clases segun el simbolo
class Letter:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return f'{self.value}'


class Append():
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __repr__(self):
        return f'({self.a}.{self.b})'


class Or():
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return f'({self.a}|{self.b})'


class Kleene():
    def __init__(self, a):
        self.a = a

    def __repr__(self):
        return f'{self.a}*'


class Plus():
    def __init__(self, a):
        self.a = a

    def __repr__(self):
        return f'{self.a}+'



class Expression():
    def __init__(self, a, b=None):
        self.a = a
        self.b = b

    def __repr__(self):
        if self.b != None:
            return f'{self.a}{self.b}'
        return f'{self.a}'
```

parsing

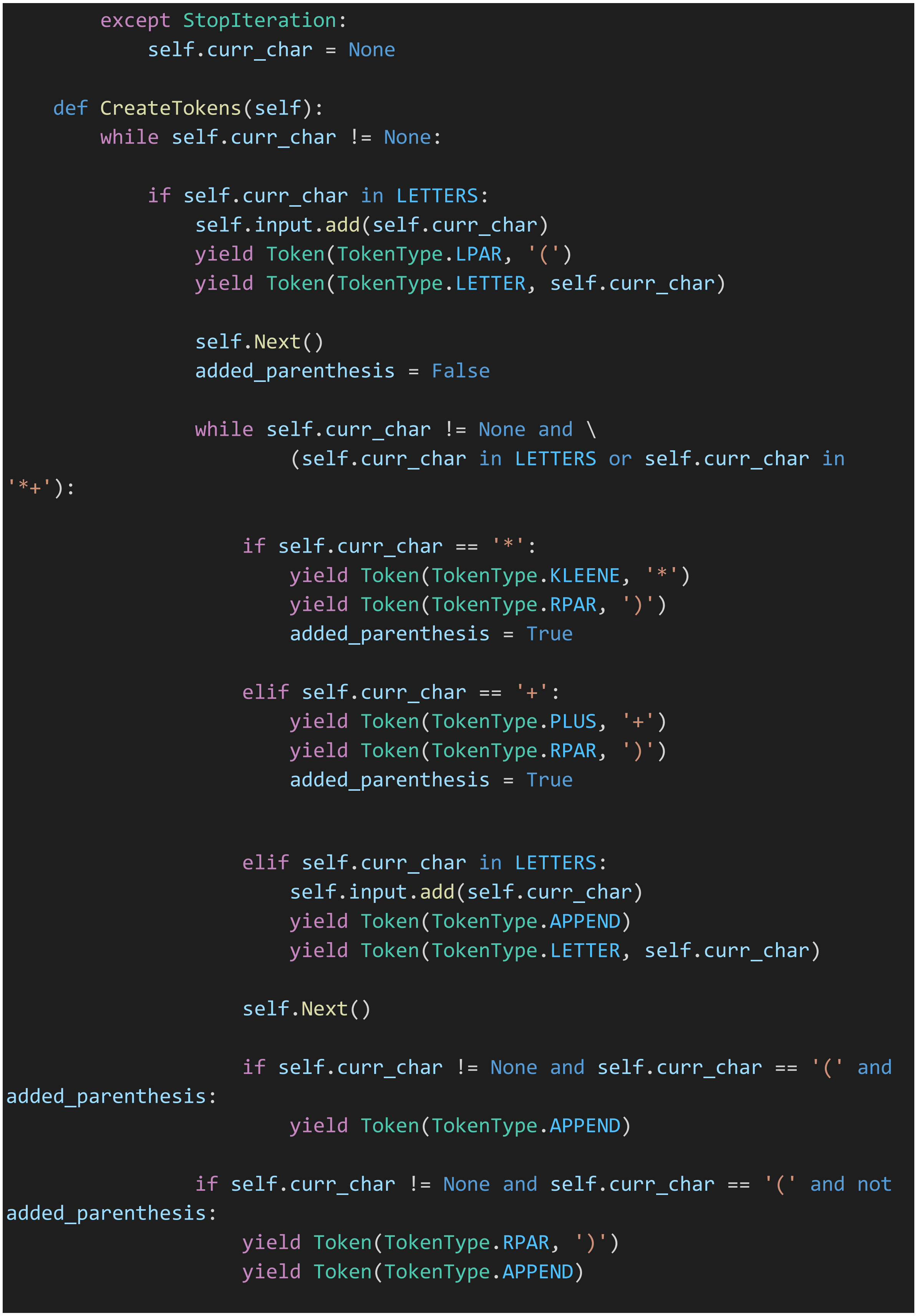
```
from tokens import TokenType
from nodes import *
```



```python
class Parser:
    def __init__(self, tokens):
        self.tokens = iter(tokens)
        self.Next()

    def Next(self):
        try:
            self.curr_token = next(self.tokens)
        except StopIteration:
            self.curr_token = None

    def NewSymbol(self):
        token = self.curr_token

        if token.type == TokenType.LPAR:
            self.Next()
            res = self.Expression()

            if self.curr_token.type != TokenType.RPAR:
                raise Exception('Sin paréntesis derecho para la expresión!')

            self.Next()
            return res

        elif token.type == TokenType.LETTER:
            self.Next()
            return Letter(token.value)

    def NewOperator(self):
        res = self.NewSymbol()

        while self.curr_token != None and \
                (
                    self.curr_token.type == TokenType.KLEENE or
                    self.curr_token.type == TokenType.PLUS
                ):
            if self.curr_token.type == TokenType.KLEENE:
                self.Next()
                res = Kleene(res)
            else:
                self.Next()
                res = Plus(res)
        return res

    def Expression(self):
        res = self.NewOperator()

        while self.curr_token != None and \
                (
                    self.curr_token.type == TokenType.APPEND or
                    self.curr_token.type == TokenType.OR
                ):
            if self.curr_token.type == TokenType.OR:
                self.Next()
                res = Or(res, self.NewOperator())

            elif self.curr_token.type == TokenType.APPEND:
                self.Next()
                res = Append(res, self.NewOperator())

        return res

    def Parse(self):
        if self.curr_token == None:
            return None

        res = self.Expression()

        return res
```

reader

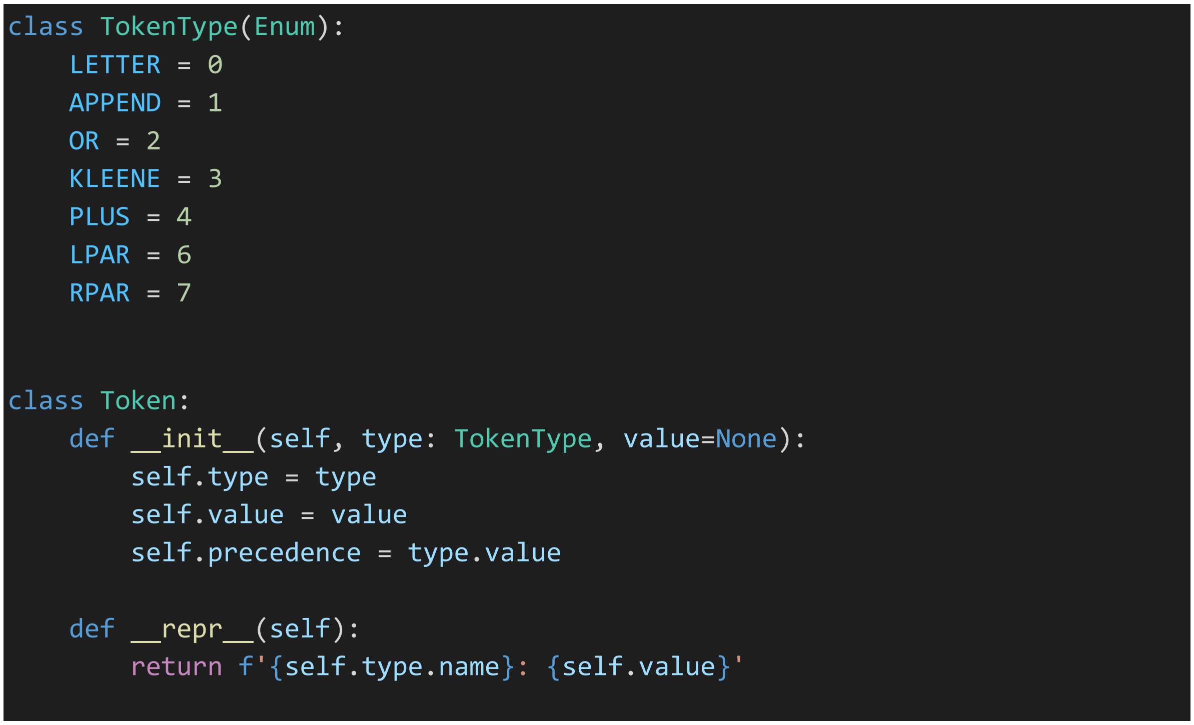
```
from tokens import Token, TokenType

LETTERS = 'abcdefghijklmnopqrstuvwxyz01234567890.'


class Reader:
    def __init__(self, string: str):
        self.string = iter(string.replace(' ', ''))
        self.input = set()
        self.Next()

    def Next(self):
        try:
            self.curr_char = next(self.string)
```



```
        except StopIteration:
            self.curr_char = None

    def CreateTokens(self):
        while self.curr_char != None:

            if self.curr_char in LETTERS:
                self.input.add(self.curr_char)
                yield Token(TokenType.LPAR, '(')
                yield Token(TokenType.LETTER, self.curr_char)

                self.Next()
                added_parenthesis = False

                while self.curr_char != None and \
                        (self.curr_char in LETTERS or self.curr_char in
'*+'):

                    if self.curr_char == '*':
                        yield Token(TokenType.KLEENE, '*')
                        yield Token(TokenType.RPAR, ')')
                        added_parenthesis = True

                    elif self.curr_char == '+':
                        yield Token(TokenType.PLUS, '+')
                        yield Token(TokenType.RPAR, ')')
                        added_parenthesis = True


                    elif self.curr_char in LETTERS:
                        self.input.add(self.curr_char)
                        yield Token(TokenType.APPEND)
                        yield Token(TokenType.LETTER, self.curr_char)

                    self.Next()

                    if self.curr_char != None and self.curr_char == '(' and
added_parenthesis:
                        yield Token(TokenType.APPEND)

                if self.curr_char != None and self.curr_char == '(' and not
added_parenthesis:
                    yield Token(TokenType.RPAR, ')')
                    yield Token(TokenType.APPEND)
                elif not added_parenthesis:
                    yield Token(TokenType.RPAR, ')')

            elif self.curr_char == '|':
                self.Next()
                yield Token(TokenType.OR, '|')

            elif self.curr_char == '(':
                self.Next()
                yield Token(TokenType.LPAR)

            elif self.curr_char in (')*+'):

                if self.curr_char == ')':
                    self.Next()
                    yield Token(TokenType.RPAR)

                elif self.curr_char == '*':
                    self.Next()
                    yield Token(TokenType.KLEENE)

                elif self.curr_char == '+':
                    self.Next()
                    yield Token(TokenType.PLUS)



                # para finalizar, se verifica si necesitamos agregar un
token append
                if self.curr_char != None and \
                        (self.curr_char in LETTERS or self.curr_char ==
'('):
                    yield Token(TokenType.APPEND, '.')

            else:
                raise Exception(f'Entrada invalida: {self.curr_char}')

    def GetSymbols(self):
        return self.input
```

tokens

```
from enum import Enum
```



```python
class TokenType(Enum):
    LETTER = 0
    APPEND = 1
    OR = 2
    KLEENE = 3
    PLUS = 4
    LPAR = 6
    RPAR = 7


class Token:
    def __init__(self, type: TokenType, value=None):
        self.type = type
        self.value = value
        self.precedence = type.value

    def __repr__(self):
        return f'{self.type.name}: {self.value}'
```