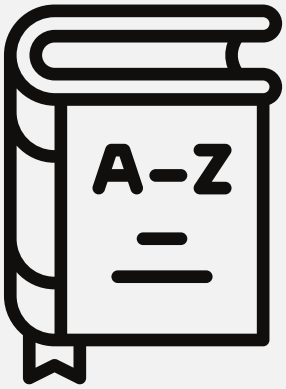


PROYECTO 2

MANUAL TECNICO

CARLOS COX BAUTISTA

202000690
LFP B+

PROYECTO 2

El manual técnico contiene la información sobre los recursos empleados para realizar la practica. Tiene como objetivo explicar el funcionamiento interno del proyecto y como realiza cada una de sus funciones durante la creación del sistema

En cuanto a este manual se ha considerado incluir todos los aspectos técnicos necesarios para el manejo y control de la aplicación para el personal que utilizara este software

OBJETIVOS Y ALCANCES

El objetivo principal de implementar este software es entregar a al usuario final una herramienta para que pueda optimizar el tiempo empleado en la búsqueda de resultados en sus diferentes tipos de la liga española de futbol.

OBJETIVOS ESPECIFICOS

- Dar a conocer como funciona la aplicación
- Conocer el alcance de toda la información por medio de una explicación detallada e ilustrada de una de las paginas que la conforman.
- Informar como se gestiona la aplicación para su correcto funcionamiento.

ESPECIFICACION TECNICA

Hardware

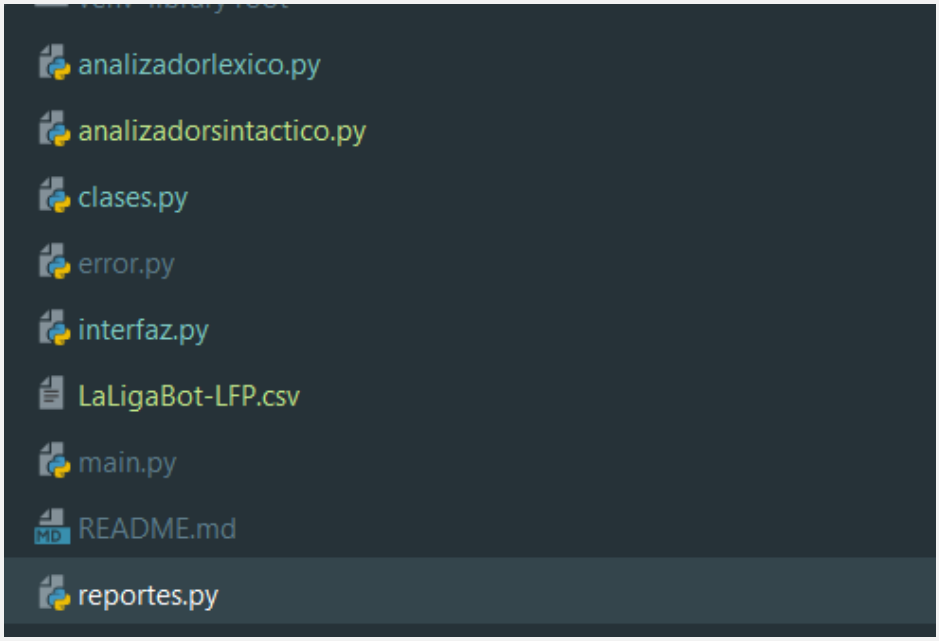
- Procesador: de 1 gigahercio (GHz), o
- procesador
- RAM: 1 gigabyte (GB) para 32 bits o 2 GB
- para 64 bits
- Espacio en disco duro:16 GB para el
- sistema operativo
- Tarjeta gráfica: DirectX 9 o posterior
- Pantalla: 800 x 600 o superior

Software

- OS: Windows o superior, o Linux
- Python: Python 3.X
- Extras:Tkinter

Clases Utilizadas

Para la realización de este programa se utilizaron clases donde se encuentran los archivos que se utilizaron para crear y dar funcionamiento al programa



Clase Main

La clase main, es el principio de nuestro programa el cual inicia nuestra interfaz grafica de la cual se incorporan las otras clases

```
def iniciar():
    interfaz.Interface()


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

Clase Interfaz

Se creo la clase interfaz con el propósito de crear los elementos gráficos utilizando Tkinter para esto,
creando así todos los elementos graficos de nuestro programa

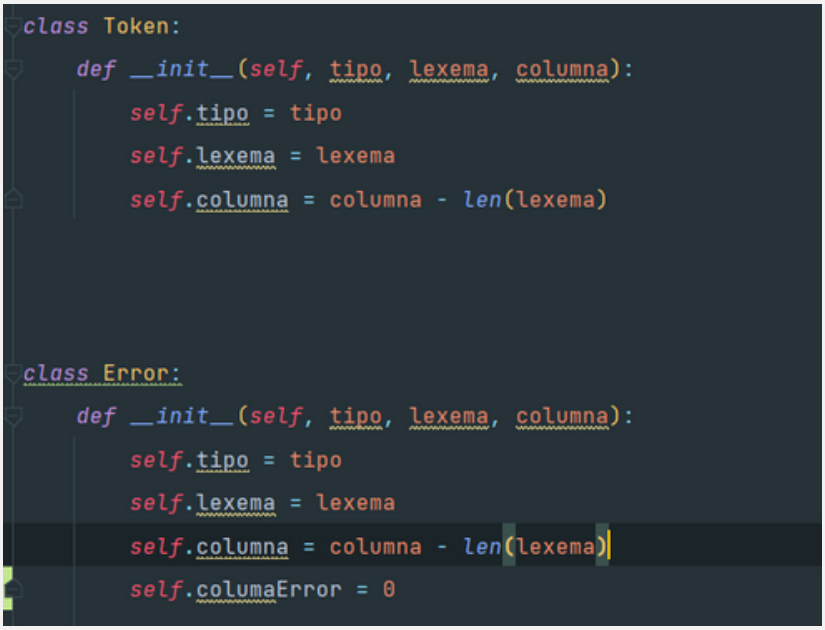
```
def __init__(self):
    self.listaErroresGlobal = []
    self.listaTokensGlobal = []
    self.root = Tk()
    self.root.geometry('1000x725')
    self.root.title('Proyecto 2')
    self.root.resizable(0, 0)
    self.Frame = Frame(self.root, width='1000', height='725')
    self.Frame.pack()
    self.Frame.config(bg='#F0F7D4')
    self.analizador = Analizadorexico()
    self.sintactico = analizadorsintactico()

    self.partidos = []
    self.reportes = reportes()

    self.text = Text(self.root, width=65, height=27, bg=BG_COLOR, fg=TEXT_COLOR,
                     font=FONT, padx=5, pady=5)
```

Clase Token y Error

Son clases muy parecidas que comparten muchas de sus características, pero se hace uso de 2 clases, para poder mantener un mejor control de los Errores y Tokens Validos



```
class Token:
    def __init__(self, tipo, lexema, columna):
        self.tipo = tipo
        self.lexema = lexema
        self.columna = columna - len(lexema)



class Error:
    def __init__(self, tipo, lexema, columna):
        self.tipo = tipo
        self.lexema = lexema
        self.columna = columna - len(lexema)
        self.columaError = 0
```

Clase Analizador Lexico

Esta clase funciona a partir de la creación de un autómata finito determinista . El cual iba leyendo carácter por carácter en base al archivo cargado a La aplicación. Reconociendo el lexema dado y verificando si pertenecía al lenguaje aceptado por medio del autómata definido. Una vez establecido su valor como lexema valido se le daba su valor respecto al tipo de token que pertenece

```
class Analizadorexico:
    def __init__(self):
        self.listaTokens = []
        self.listaErrores = []

    def limpiar(self, tipo):
        if tipo == 'errores':
            self.listaErrores = []
        elif tipo == 'tokens':
            self.listaTokens = []

    def palabrasRES(self, buffer):
        if buffer == 'TOTAL':
            return True
        elif buffer == 'PARTIDOS':
            return True
        elif buffer == 'RESULTADO':
            return True
        elif buffer == 'JORNADA':
            return True
        elif buffer == 'VS':
            return True
        elif buffer == 'TOP':
```

Clase Analizador Sintáctico

Esta clase funciona a partir de la creación de un autómata finito determinista . El cual iba leyendo carácter por carácter en base al archivo cargado a La aplicación. Reconociendo el lexema dado y verificando si pertenecía al lenguaje aceptado por medio del autómata definido. Una vez establecido su valor como lexema valido se le daba su valor respecto al tipo de token que pertenece

```
class analizadorsintactico:
    def __init__(self):
        self.instruccionDetectada = ''
        self.resultadoFinal = ''
        self.listaTokens = []
        self.listaErrores = []
        self.i = 0
        self.partidos = []
        self.instruccionencontrada = False

    def reutnError(self):
        return self.listaErrores

    def limpiar(self):
        self.listaErrores = []
        self.listaTokens = []

    def run(self):
        token = self.listaTokens[self.i]
        self.listaInstrucciones()
        token = self.listaTokens[self.i]
```

Creacion Abol Binario

Se creo una tabla donde se definían los distintos tipos de tokens que nuestro AFD iba poder ser capaz de leer y así poder crear una expresión regular para poder empezar a crear nuestro arbol binario

Tokens		
Token	**Lexema**	**Regex**
PalabraR | IDENTIFICADOR	"RESULTADO" "VS" "TEMPORADA" "JORNADA" "-r" "GOLES" "LOCAL" "VISITANTE" "TOTAL" "PARTIDOS" "-jl" "-jj" "TOP" "SUPERIOR" "INFERIOR" "-n" "ADIOS" "TABLA"	L+
Cadena	"VALENCIA"	"(L)*"
SIGNOS	< >	S+
ENTERO	12,1554	N+

L = {A-Z, a-z}

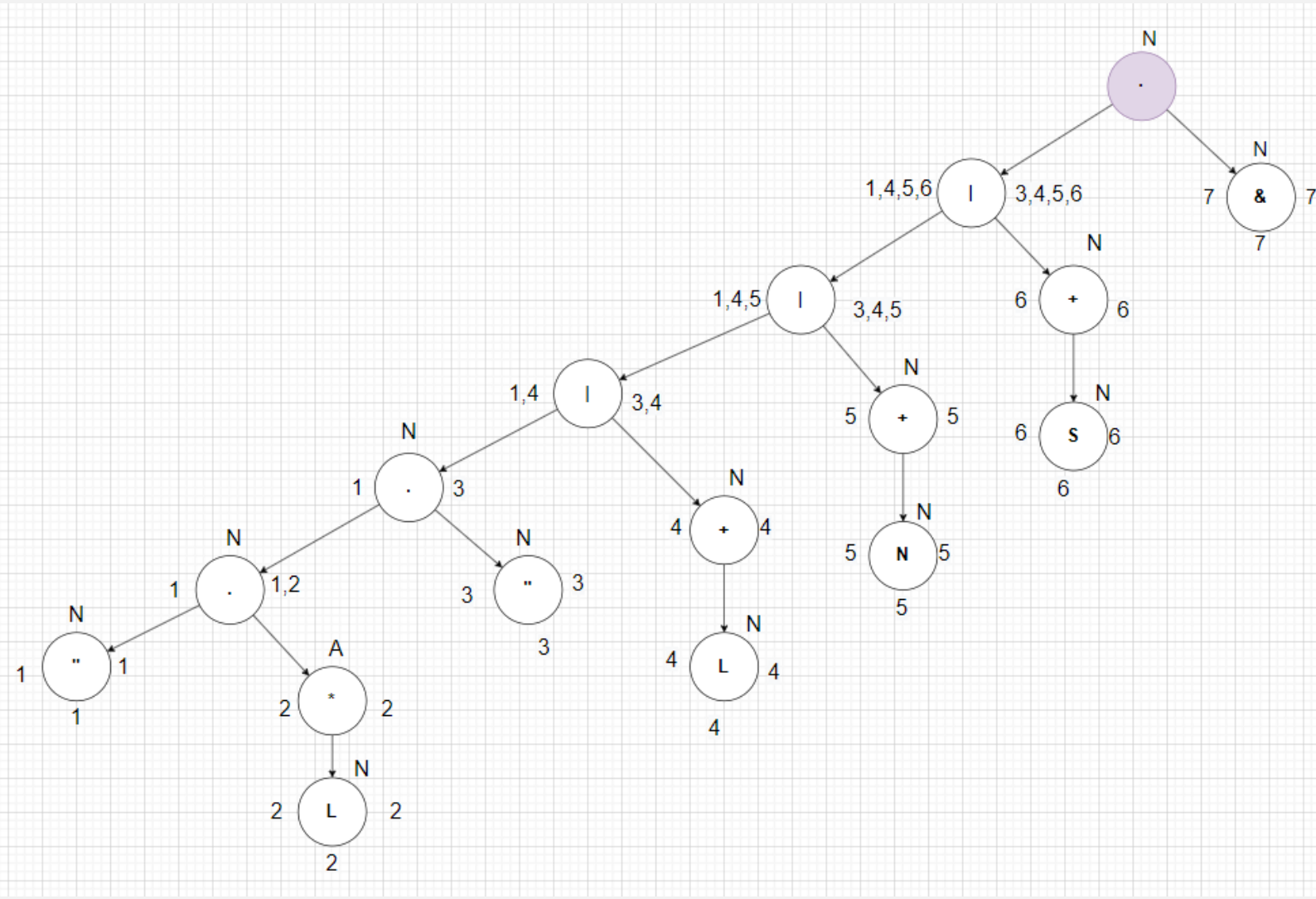
S={ < > }

N = {0-9}

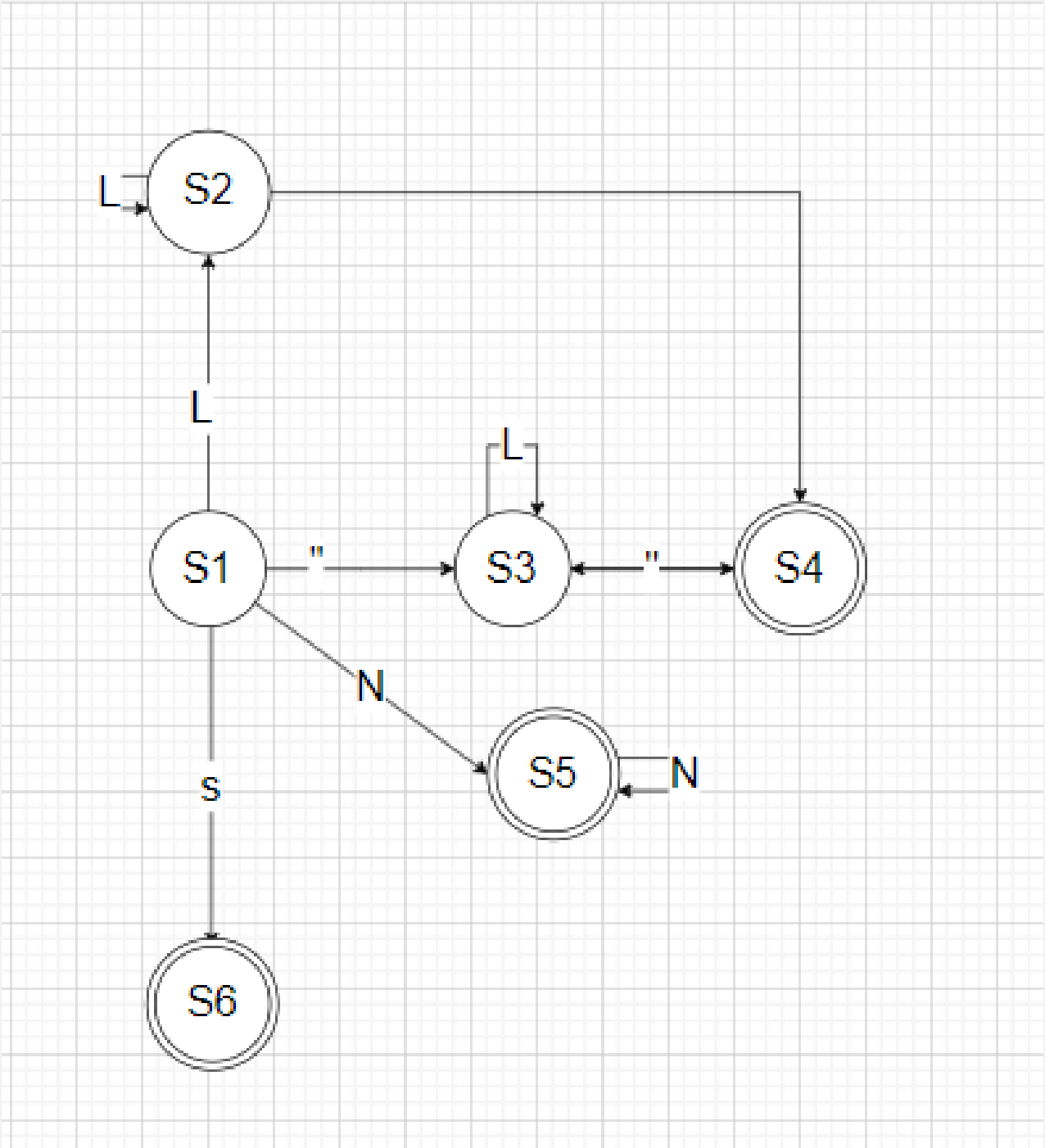
Expresion Regular

["(L)*"| L+|N+|S+]$

Con la expresión regular ya definida se empieza a construir el árbol binario colocando las hojas, concatenaciones, estableciendo anulables y no anulables y los siguientes de cada uno, etc.



Automata Finito Determinista



Gramatica de Libre Contexto

Muchas construcciones de los lenguajes de programación tienen una estructura inherentemente recursiva que se puede definir mediante gramáticas independientes del contexto.

Estas gramáticas capturan la noción de constituyente sintáctico y la noción de orden

$$G = (N, T, P, S)$$

T= Símbolos Terminales

```
T = {
    adios       resultado
    vs          temporada
    jornada     -f
    goles       local
    visitante   total
    partidos    -ji
    -jf         top
    superior    inferior
    temporada   goles
    cadena      entero
    mayorque    menorque
    guion       identificador
}
```

N= Símbolos NO Terminales

```
N= {S, RESULTADO,
    JORNADA, GOLES,
    TABLA,PARTIDOS,
    TOP,ADIOS,-F,-
    N,-FI-,-FJ,
    CONDICION, VIENETEMPORADA}
```

P = Reglas de producción

S= Símbolo inicial de la gramática

```
TOKENS = minusculas
ESTADOS = MAYUSCULAS
```

```
CONDICION::= local | visitante | total |superior | inferior

VIENETEMPORADA::=  temporada menorque entero guion entero mayorque

-F ::= (-f identificador) | epsilon
-N ::= (-n entero) | epsilon
-FI ::= (-fi en) | epsilon
-Fj ::= (-fj entero) | epsilon
```

```
S ::= RESULTADO
 RESULTADO ::= resultado cadena vs cadena VIENETEMPORADA

S ::= JORNADA
 JORNADA ::= jornada entero VIENETEMPORADA -F

S ::= GOLES
 GOLES ::= goles CONDICION cadena VIENETEMPORADA

S ::= TABLA
 TABLA ::= tabla VIENETEMPORADA -F

S ::= PARTIDOS
 PARTIDOS ::= partidos cadena VIENETEMPORADA -F -FI -FJ

S ::= TOP
 TOP ::= top CONDICION VIENETEMPORADA -N

S ::= ADIOS
 ADIOS ::= adios
```