25/04/22

LFP

Manual Técnico

Universidad de San Carlos de Guatemala

Lenguajes Formales de Programación

Carlos Eduardo Soto Marroquín
201902502

Primer Semestre *abril 2022*

Contents

Introducción

Dicho proyecto procesa información ingresada desde una caja de texto, procesa la información por medio de un análisis léxico, si todo se encuentra bien, se dirige a analizar sintácticamente, para retornar valores por consola o generar reportes, según sea el comando ingresado en la caja de texto. Esto se realiza por medio de una aplicación de escritorio.

Especificación Técnica

Los requerimientos mínimos para el posterior uso del programa son:

- Lenguajes implementados: Python, HTML, CSS, JavaScript
- IDE usado: Ninguno
- Editor de código: visual studio code
- Sistema operativo: Windows 10 (64 bits)
- Interfaz gráfica: Tkinter, HTML, CSS, JS
- Librerías implementadas: Tkinter

Objetivos:

- Aplicación de conceptos alfabeto, símbolos, cadenas y reglas
- Desarrollo de un analizador léxico para procesamiento de información
- Implementación de interfaz gráfica en Python
- Aplicación de conocimientos de desarrollo WEB

Alcances del proyecto:

La intención del proyecto es que el encargado del desarrollo de esta aplicación aprenda a revisar un orden correcto de tokens, por medio de las producciones del lenguaje libre de contexto del lenguaje, y posteriormente de su análisis, generar reportes de los equipos de la temporada, de los resultados de un equipo en una cierta temporada. Todo esto por medio de la programación, implementando el lenguaje de Python para el análisis sintáctico, y su salida de reportes estadísticos aplicando teoría del desarrollo web. Con esto se busca que el desarrollador entienda como trabajar con analizadores sintácticos, aplicando los conceptos de lenguajes formales.

Lógica del programa

Antes de comenzar con la explicación de cada clase, es necesario explicar como se maneja la información de forma general, ya que se utilizó la arquitectura MVC(Modelo-Vista-Controlador)

MVC:

Es un patrón de diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control. Enfatiza una separación entre la lógica de negocios y su visualización. Esta “separación de preocupaciones”, proporciona una mejor división del trabajo y una mejora de mantenimiento.

Las tres partes del patrón de diseño de software MVC se pueden describir de la siguiente manera:

Modelo:

Representa los datos. Un modelo se ocupa de obtener los datos o escribir datos en un almacenamiento, como una base de datos o un archivo. El modelo también puede contener la lógica para validar los datos para garantizar la integridad de los datos.

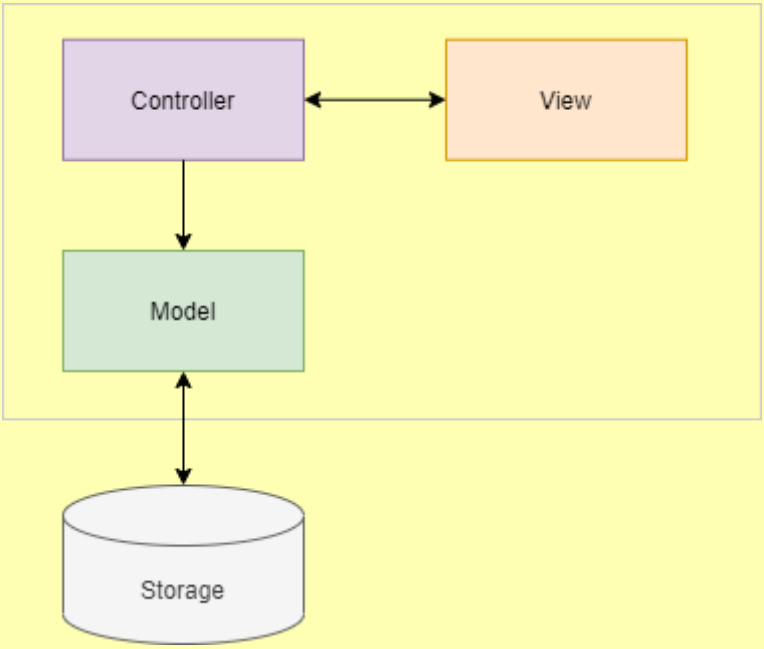
El modelo **no** debe depender de la vista y el controlador.

Vista:
Una vista es la interfaz de usuario que representa los datos en el modelo. La vista no se comunica directamente con el modelo, Idealmente, una vista debería tener muy poca lógica para mostrar datos.

La vista se comunica con el controlador directamente.

Controlador:
Un controlador actúa como intermediario entre las vistas y los modelos. El controlador enruta (redirige los datos a otro lugar) los datos entre las vistas y los modelos.

A continuación, una imagen, para que se comprenda mejor este diseño de software:



Elaboración: pythontutorial.net, enlace

Implementando este diseño de software, fue de manera útil más sencillo manejar los datos de la aplicación.

Clases utilizadas:

Se presentarán cada una de las clases, métodos, funciones, procedimientos que se implementaron para toda la aplicación. Primero hablemos de todas las clases que se utilizaron:

- Main
- Vista
- Controlador
- Modelo
- AnalizadorLéxico
- AnalizadorSintáctico
- Token
- Error
- Equipo
- Clasificación
- LectorArchivo

Main:

Esta clase es la encargada de iniciar la aplicación, se muestra a continuación:

```
if __name__ == "__main__":
    app = App()
    app.geometry("1200x600")
    app.config(bg="#5e64ad")
    app.resizable(width=False, height=False)
    app.mainloop()
```

Token:

La encargada de almacenar los tokens válidos del lenguaje también devuelve la información por separado, se muestra a continuación:

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

    def getTipo(self):
        return self.tipo

    def getLexema(self):
        return self.lexema

    def getLinea(self):
        return self.linea

    def getColumna(self):
        return self.columna
```

Error:

La clase error es la encargada de almacenar los caracteres que no pertenecen al lenguaje, y los errores sintácticos que se encuentren en el análisis sintáctico, almacenará y mostrará la información de cada error que se encuentre, se muestra a continuación:

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

    def getTipo(self):
        return self.tipo

    def getLexema(self):
        return self.lexema

    def getLinea(self):
        return self.linea

    def getColumna(self):
        return self.columna
```

Analizador Léxico:

Es el encargado de procesar la información del cuadro de texto donde se presenta la información para su posible modificación y posterior análisis, se presenta parte de la clase:

```
from Token import Token
from Error import Error
from FuncionIS import *
import webbrowser

class AnalizadorLexico():

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

    def analizarEntrada(self, entrada):
        self.listaTokens = []
        self.listaErrores = []
        ......
```

Métodos de la clase analizador Léxico:

La clase Analizador Léxico contiene 4 métodos de la clase, más la función constructora, a continuación, se presenta cada una de ellas.

Constructor:

Este método perteneciente a la clase de analizador Léxico es la encargada de inicializar los atributos y métodos que tendrá

la instancia del objeto se presenta a continuación:

```
class AnalizadorLexico():

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

analizarEntrada:

Dicho método nos permite analizar la información por medio de una entrada, la información proviene de un cuadro de texto en la parte gráfica de la aplicación, se analiza la información y se va almacenando los tokens y errores en cada una de las listas respectivamente.

```
def analizarEntrada(self, entrada):
        self.listaTokens = []
        self.listaErrores = []


        buffer = ""
        centinela = "#"
        entrada += centinela
        linea = 1
        columna = 1


        estado = 0
        index = 0


#* Vamos a recorrer todo el archivo de entrada
        while index < len(entrada):
            c = entrada[index]
            if index == 1 and c == centinela:
                return False
            if estado == 0:
                ....
```

getListaTokens:

Este método es el encargado de retornar la lista de tokens.

```
def getListaTokens(self):
        return self.listaTokens
```

getListaErrores:

Este método es el encargado de retornar la lista de errores.

```
def getListaErrores(self):
        return self.listaErrores
```

Sub-Métodos clase analizadorLéxico:

Son métodos que complementaron al analizador a la hora de reconocer un token válido del lenguaje.

- Is_letter: Verifica si un carácter es minúscula o mayúscula del abecedario
- is_number: Verifica si el carácter es un digito entre 0 y 9.
- is_identifier: Verifica si el carácter es un identificador.

Métodos de la clase analizador Sintáctico:

La clase Analizador Sintáctico contiene métodos de la clase propia, más la función constructora, a continuación, se presenta cada una de ellas.

getLtsDatos:

Retorna la lista de datos, osea es la lista que contendrá los datos de obtenidos de cada comando.

```
ef getLtsDatos(self):
        return self.lts_datos
```

getLtsErrores:

Retorna la lista de errores sintácticos, osea es la lista que contendrá los datos de obtenidos de cada comando.

```
def getLtsErrores(self):
        return self.listaErrores
```

sacarToken:

Saca el primer token y lo elimina de la lista.

```
def sacarToken(self):
        #* Saca el primer token y lo quita de la lista
        try:
            return self.tokens.pop(0)
        except:
            return None
```

---

verToken:

Obtiene el primer token, más no lo elimina de la lista.

```
def verToken(self):

#* Saca el primer token mas no lo quita de la lista, solo se muestra
        try:
            return self.tokens[0]
        except:
            return None
```

Métodos utilizados para la gramática libre de contexto:

S:

Es el no terminal del lado izquierdo de la gramática, es con el primer símbolo que empieza toda la gramática.

INICIO:

Es un no terminal con el cual decidiremos a que parte de la gramática ir.

```
def INICIO(self):
       temporal = self.verToken()
       if temporal is None:
           #Da error por que viene una cadena vacia
           error = Error("error sintactico", "<<EOF>>", "", "")
           self.listaErrores.append(error)
       elif temporal.tipo == "resultado":
           self.RESULTADO()
       elif temporal.tipo == "jornada":
           self.JORNADA()
       elif temporal.tipo == "goles":
           self.GOLES()
       elif temporal.tipo == "tabla":
           self.TABLA()
       elif temporal.tipo == "partidos":
           self.PARTIDOS()
       elif temporal.tipo == "top":
           self.TOP()
       elif temporal.tipo == "adios":
           self.ADIOS()
       else:
           error = Error("error sintactico"
, temporal.lexema, temporal.linea, temporal.columna)
           self.listaErrores.append(error)
           self.lts_datos = ["Error-r"]
```

RESULTADO:

Es un no terminal, y el encargado de reconocer el primer comando.

JORNADA:

Es un no terminal, y el encargado de reconocer el segundo comando.

GOLES:

Es un no terminal, y el encargado de reconocer el tercer comando.

TABLA:

Es un no terminal, y el encargado de reconocer el cuarto comando.

PARTIDOS:

Es un no terminal, y el encargado de reconocer el quinto comando.

TOP:

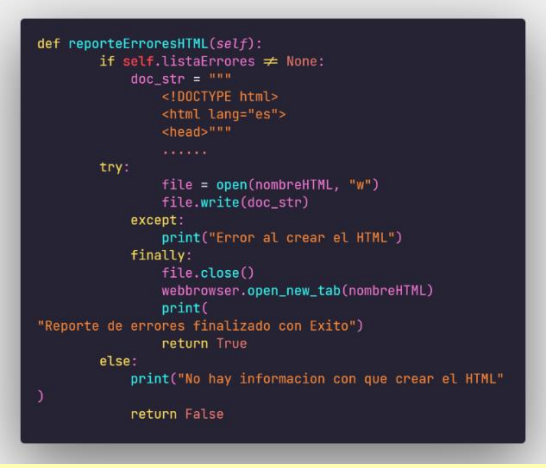
Es un no terminal, y el encargado de reconocer el sexto comando.

ADIOS:

Es un no terminal, y el encargado de reconocer el séptimo comando.

Reporte de errores HTML:

Este método es el encargado de mostrar los errores léxicos y sintácticos que se encontraron al analizar el comando que contiene la información, se muestra a continuación:



```
def reporteErroresHTML(self):
        if self.listaErrores != None:
            doc_str = """
                <!DOCTYPE html>
                <html lang="es">
                <head>"""
                ......
        try:
                file = open(nombreHTML, "w")
                file.write(doc_str)
            except:
                print("Error al crear el HTML")
            finally:
                file.close()
                webbrowser.open_new_tab(nombreHTML)
                print(
"Reporte de errores finalizado con Exito")
                return True
        else:
            print("No hay informacion con que crear el HTML"
)
            return False
```

Reporte tokens HTML:

Este método se encarga de procesar los tokens válidos y presentarlos en una tabla por medio de un reporte creado en HTML, se muestra a continuación:

```
def reporteTokensHTML(self):
        if self.listaTokens != None:
            doc_str = """
                <!DOCTYPE html>
                <html lang="es">
                <head>"""
                ....
        except:
                print("Error al crear el HTML")
            finally:
                file.close()
                webbrowser.open_new_tab(nombreHTML)
                print("Reporte de tokens finalizado con Exito")
                return True
        else:
            print("No hay informacion con que crear el HTML")
            return False
```

Vista:

Como tal, es la interfaz con la que el usuario interactúa.

```
class Vista(ttk.Frame):

    def __init__(self, parent):
        super().__init__(parent)
        self.parent = parent
        #*Se crean los widgets
        self.create_widgets()
```

Controlador:

Es el encargado de intercambiar los datos entre la vista y el controlador.

```
class Controlador():
    def __init__(self, modelo, vista):
        self.modelo = modelo
        self.vista = vista
```

Modelo:

Es que manejará la lógica de negocio, en este caso actuará como base de datos, y como la lógica principal de las respuestas a los comandos.

```
class Modelo():
#*Tiene toda la logica del negocio osea el backend
    def __init__(self, name):
        self.name = name
        self.partidos : list = self.leerArchivo()
        self.lts_erroresSG = []
        self.lts_tokensG = []
        self.lts_erroresG = []
        self.lts_tokens_tmp = None
        self.lts_datos = None
```

Partido:

Es la clase que nos ayudará a guardar cada partido que viene en el archivo csv.

```
class Partido():
    def __init__(self, fecha, temporada, jornada, equipoLocal, equipoVisitante, golesLocal, golesVisitante):
        self.fecha = fecha
        self.temporada = temporada
        self.jornada = jornada
        self.equipoLocal = equipoLocal
        self.equipoVisitante = equipoVisitante
        self.golesLocal = golesLocal
        self.golesVisitante = golesVisitante
```

Equipo:

Es la clase que almacenará un equipo con sus datos respectivos.

```
class Equipo():
    def __init__(self, nombre):
        self.nombre = nombre
        self.pts_local = 0
        self.pts_visitante = 0
        self.total_pts = 0
        self.datos_local = DatosEstadisticos()
        self.datos_visitante = DatosEstadisticos()
```

DatosEstadisticos:

Almacenará los datos repetitivos de los equipos, como los partidos jugados, partidos ganados, partidos empatados, etc.

```python
class DatosEstadisticos():
    def __init__(self):
        self.PJ = 0
        self.PG = 0
        self.PE = 0
        self.PP = 0
        self.GF = 0
        self.GC = 0
```

Clasificación:

Este objeto almacenara la temporada, y la lista de equipos que jugaron en dicha temporada.

```python
class Clasificacion():
    def __init__(self, temporada):
        self.temporada = temporada
        self.lista_equipos = []
```

Expresiones Regulares y AFD de los tokens:

Se colocará cada uno de los tokens válidos, pero antes se realizó una lista de las palabras reservadas, signos, banderas y otros símbolos que acepta el lenguaje a procesar, junto con su patrón escrito, y su Lexema.

Palabras Reservadas

TIPO	PATRON	LEXEMA
RESULTADO	palabra reservada RESULTADO	RESULTADO
VS	palabra reservada VS	VS
TEMPORADA	palabra reservada TEMPORADA	TEMPORADA
JORNADA	palabra reservada JORNADA	JORNADA
GOLES	palabra reservada GOLES	GOLES
LOCAL	palabra reservada LOCAL	LOCAL
VISITANTE	palabra reservada VISITANTE	VISITANTE
TOTAL	palabra reservada TOTAL	TOTAL
TABLA	palabra reservada TABLA	TABLA
TOP	palabra reservada TOP	TOP
SUPERIOR	palabra reservada SUPERIOR	SUPERIOR
INFERIOR	palabra reservada INFERIOR	INFERIOR
ADIOS	palabra reservada ADIOS	ADIOS
PARTIDOS	palabra reservada PARTIDOS	PARTIDOS

Banderas

TIPO	PATRON	LEXEMA
-f	palabra reservada -f	-f
-ji	palabra reservada -ji	-ji
-jf	palabra reservada -jf	-jf
-n	palabra reservada -n	-n

Signos

TIPO	PATRON	LEXEMA
menorque	caracter <	<
mayorque	caracter >	>
guion	caracter -	-

Otros

TIPO	PATRON	LEXEMA
cadena	Secuencia de caracteres que inicia con comillas dobles, seguido de cualquier caracter, hasta que encuentre otra comilla doble.	"Villareal"

Se mostrará cada expresión regular con su AFD de cada uno de los tokens válidos para el lenguaje, primero se expresará el lexema, luego el tipo de Token, después la expresión regular y por último el AFD. Pero antes de seguir, vamos a tratar de manera diferente la expresión regular y el AFD, ya que se tienen palabras reservadas, identificadores, banderas, signos, y otros símbolos en cada lenguaje, es por ello, por lo que con el afán de no hacer tan grande el problema a la hora de resolver el problema se optó por resolverlo de manera distinta con lo que se aprendió.

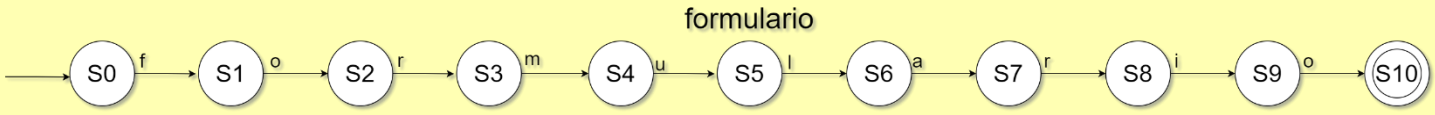
Para dicho proyecto se tiene como lenguaje aceptable, palabras reservas, identificadores, signos, cadena y banderas, pero se tiene un pequeño problema, ya que las palabras reservadas pueden ser reconocidas como un identificador. Existen dos posibles soluciones para dicho, problema.

1. Instalar las palabras reservadas en la tabla de símbolos desde el inicio
2. Crear diagramas de transición de estados separados para cada palabra clave

Solo para dar un ejemplo en concreto de que se tomará la opción dos, quedaría así:

formulario como palabra reservada de un lenguaje L(G)

Y su AFD quedaría de la siguiente manera:



Para el proyecto se tomó la primera opción, ya que se trabajará con el método del árbol, y por qué el otro método nos dejaría muchos estados con los cuales trabajar. En el caso de los identificadores, su expresión regular es la siguiente:

$$L(L|D|\_)^*$$

Donde L es cualquier letra, ya sea minúscula o mayúscula, D sería cualquier digito entre 0 y 9, y guion bajo. La expresión regular se lee: Una letra seguida de cero o muchas veces une letra o un digito o un guion bajo. Bien aun así se tiene el problema de que está expresión regular puede reconocer palabras reservadas

como identificadores, y en esta parte es donde nos beneficia decir que el proyecto es case sensitive, en otras palabras, es que no es lo mismo decir:

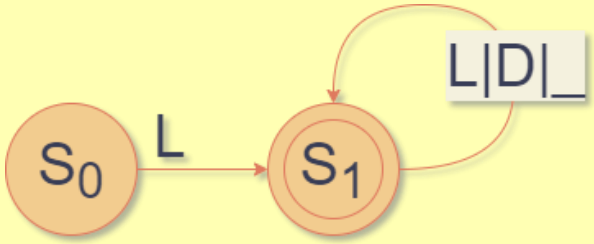
formulario != FORMULARIO

Esto lo que nos quiere decir, es que como todas las palabras reservadas tendrán que venir en mayúsculas para su reconocimiento. Entonces en la parte de la programación, se validará esto. Cualquier otra palabra se tomará como un identificador, y así solucionando el problema de ambigüedad.

De manera general se procede a mostrar el patrón general para las palabras reservadas:

$$\mathrm{L}(\mathrm{L}|D|\_)*$$

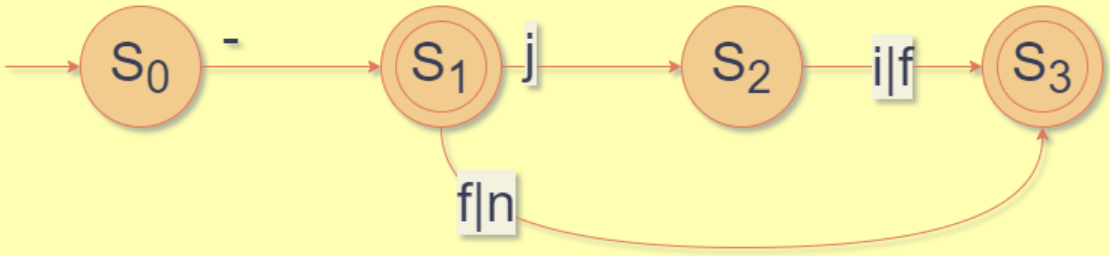
Su autómata sería:



Las banderas que son parte del lenguaje se trabajaron de la misma manera, su expresión regular es:

-(f|n|j(i|f))

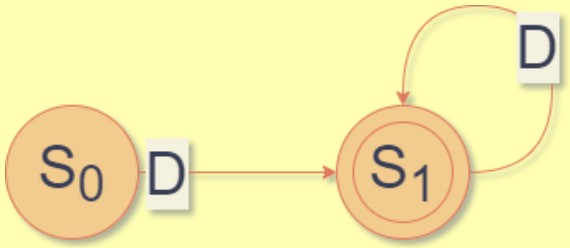
Su autómata sería:



En el caso de los enteros, su expresión regular es:

$$DD*$$

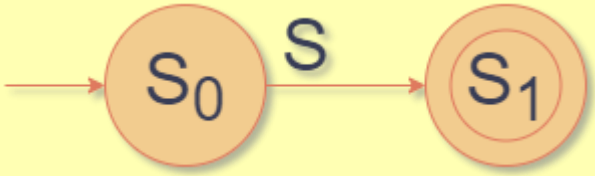
Su autómata sería:



Los signos también son parte de este lenguaje, su expresión regular es:

$$S$$

Su autómata sería:



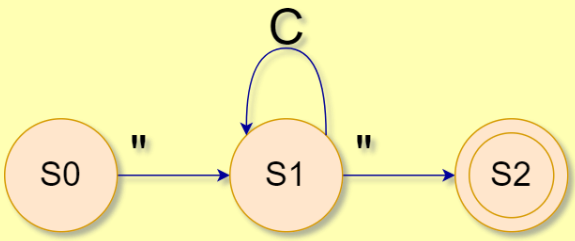
Donde S, es el conjunto de signos aceptados por el lenguaje.

Por último, se tiene otros símbolos, pero en realidad son solo las cadenas que se pueden admitir, en este caso el patrón sería que empiece con una comilla doble seguida de cualquier carácter seguida 0 o muchas veces, seguido de una comilla doble, osea:

$$"C*"$$

Donde C es cualquier carácter excepto comilla doble.

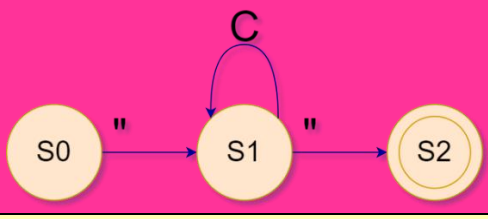
Su AFD sería el siguiente:



Donde C es cualquier carácter excepto comilla doble.

En caso de que vengan caracteres no conocidos del lenguaje, se reportarán como un error léxico. Se pasa a mostrar la tabla de tokens válidos para el lenguaje.

Tabla de Tipos

Lexema	Tipo Token	Expresión Regular	AFD
RESULTADO VS TEMPORADA JORNADA GOLES LOCAL VISITANTE TOTAL TABLA PARTIDOS TOP SUPERIOR INFERIOR ADIOS	PALABRAS RESERVADAS	L(L|D|_)*	
-f -ji -jf -n	Bandera	-(f|ji|jf|n)	
menorque mayorque guion	SIGNOS	S	
reporteGlobal1 jornada1Reporte ReporteEspañol	IDENTIFICADOR	L(L|D|_)*	
2 15 1999	Entero	DD*	
“Valencia” “Zaragoza” “Real Madrid”	Cadena	“C*”	

Método del árbol:

Se podrá dar cuenta, ya se tienen los cinco por separado, pero ahora bien como armamos nuestro autómata completo, puesto que solo hemos reconocido cada palabra reservada, signo y otros(cadenas), por separado, para esto, se utilizó el método del árbol, como se mencionó antes, están por separado y cada una de

ella globaliza cierta cantidad de palabras reservadas, signos y otros(cadenas). Así se presenta, paso a paso la creación del árbol para poder obtener un AFD óptimo.

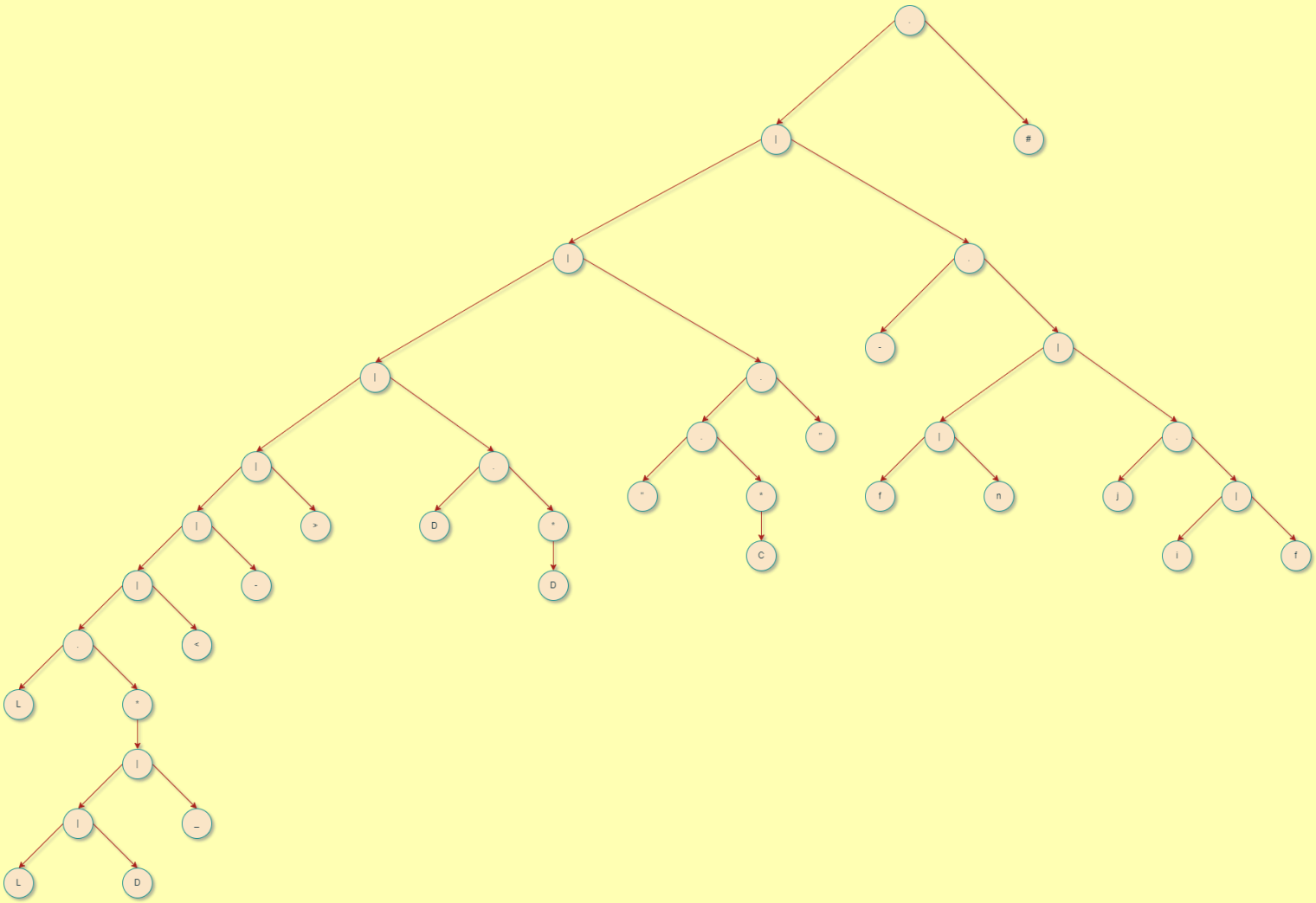
Pasos método del árbol

Patrón: L(L|D|_)*|<|-|>|DD*|"C*"|-(f|n|j(i|f))Paso 1:

Al patrón se le debe agregar el símbolo de aceptación, en este caso será “#”.

(L(L|D|_)*|<|-|>|DD*|"C*"|-(f|n|j(i|f)))#

Paso 2: Formar el árbol de la expresión regular:

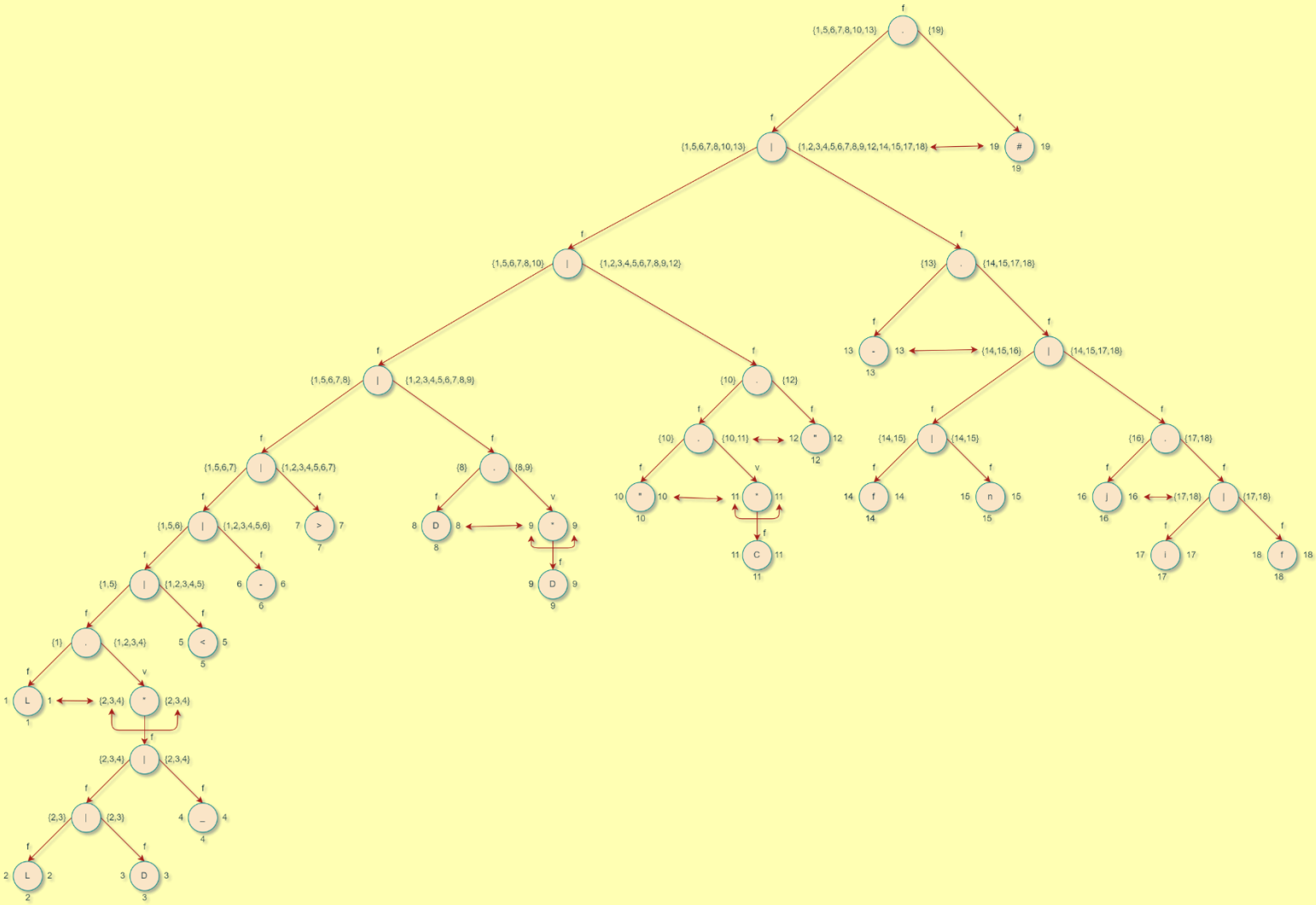


Los pasos 3,4,5 se pueden aplicar al mismo tiempo.

Paso 3: Numerar las hojas del árbol

Paso 4: Aplicar la función anulable

Paso 5: Aplicar funciones de PrimeraPos y ultimaPos



Paso 6: Calcular la función de siguiente

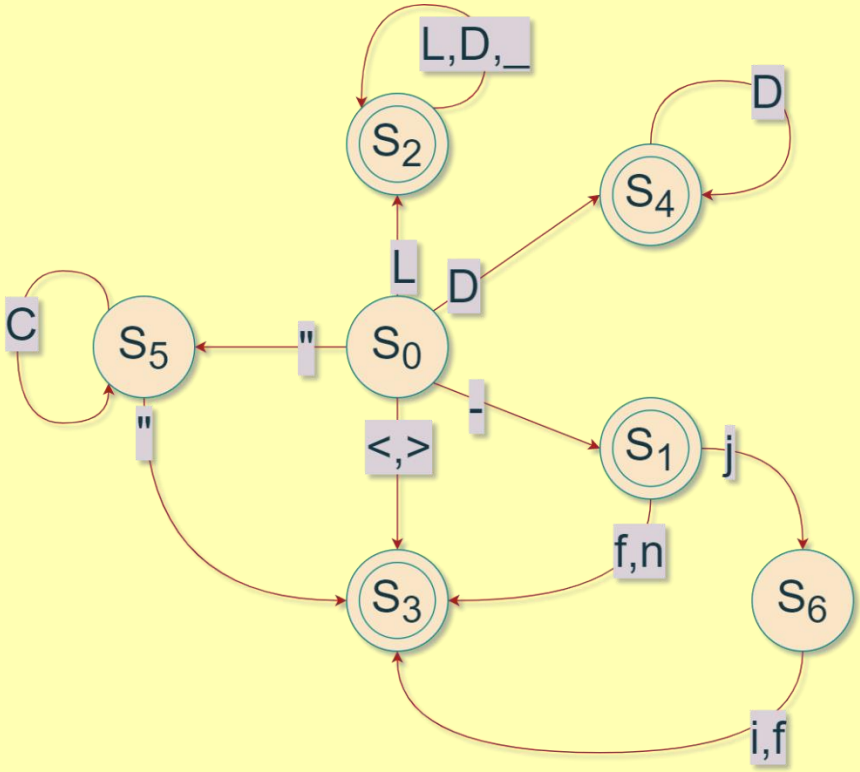
Terminal	ID	Siguientes(i)
L	1	2,3,4,19
L	2	2,3,4,19
D	3	2,3,4,19
_	4	2,3,4,19
<	5	19
-	6	19
>	7	19
D	8	9,19
D	9	9,19
"	10	11,12
C	11	11,12
"	12	19
-	13	14,15,16
f	14	19
n	15	19
j	16	17,18
i	17	19
f	18	19
#	19	

Paso 7: Calcular la tabla de transiciones

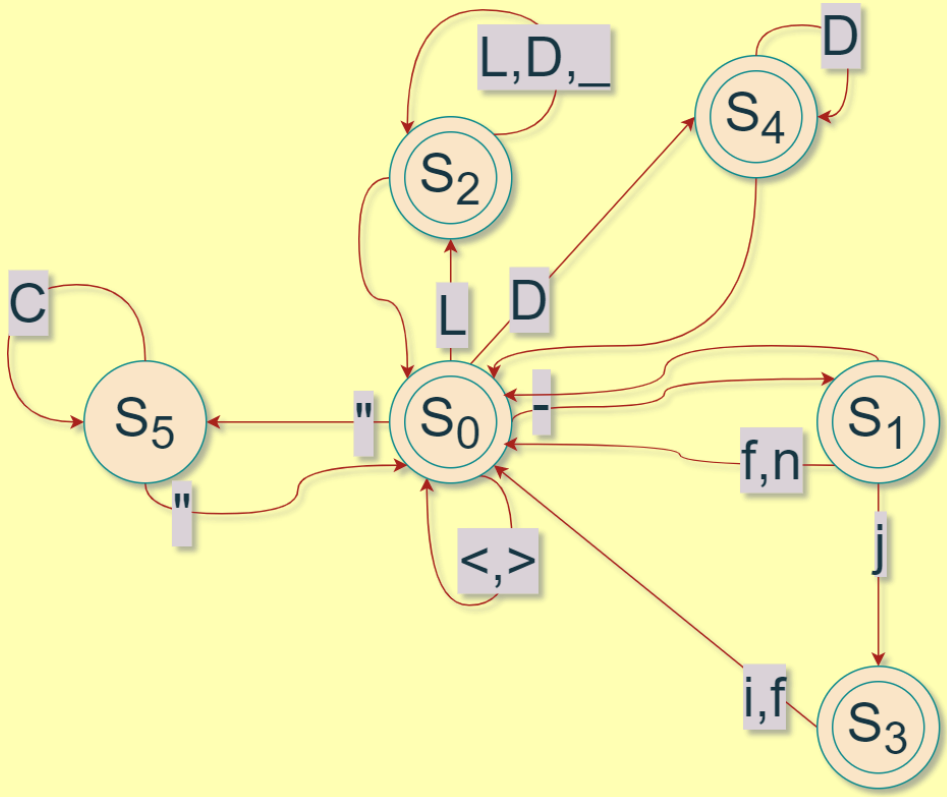
Tabla de transiciones

E/T	L	D	<	-	>	_	"	C	f	n	j	i
S0={1,5,6,7,8,10,13}	S2	S4	S3	S1	S3	----------	S5	----------	----------	----------	----------	----------
S1= {14,15,16,19}	----------	----------	----------	----------	----------	----------	----------	----------	S3	S3	S6	----------
S2={2,3,4,19}	S2	S2	----------	----------	----------	S2	----------	----------	----------	----------	----------	----------
S3={19}	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------
S4={9,19}	----------	S4	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------
S5={11,12}	----------	----------	----------	----------	----------	----------	S3	S5	----------	----------	----------	----------
S6={17,18}	----------	----------	----------	----------	----------	----------	----------	----------	S3	----------	----------	S3

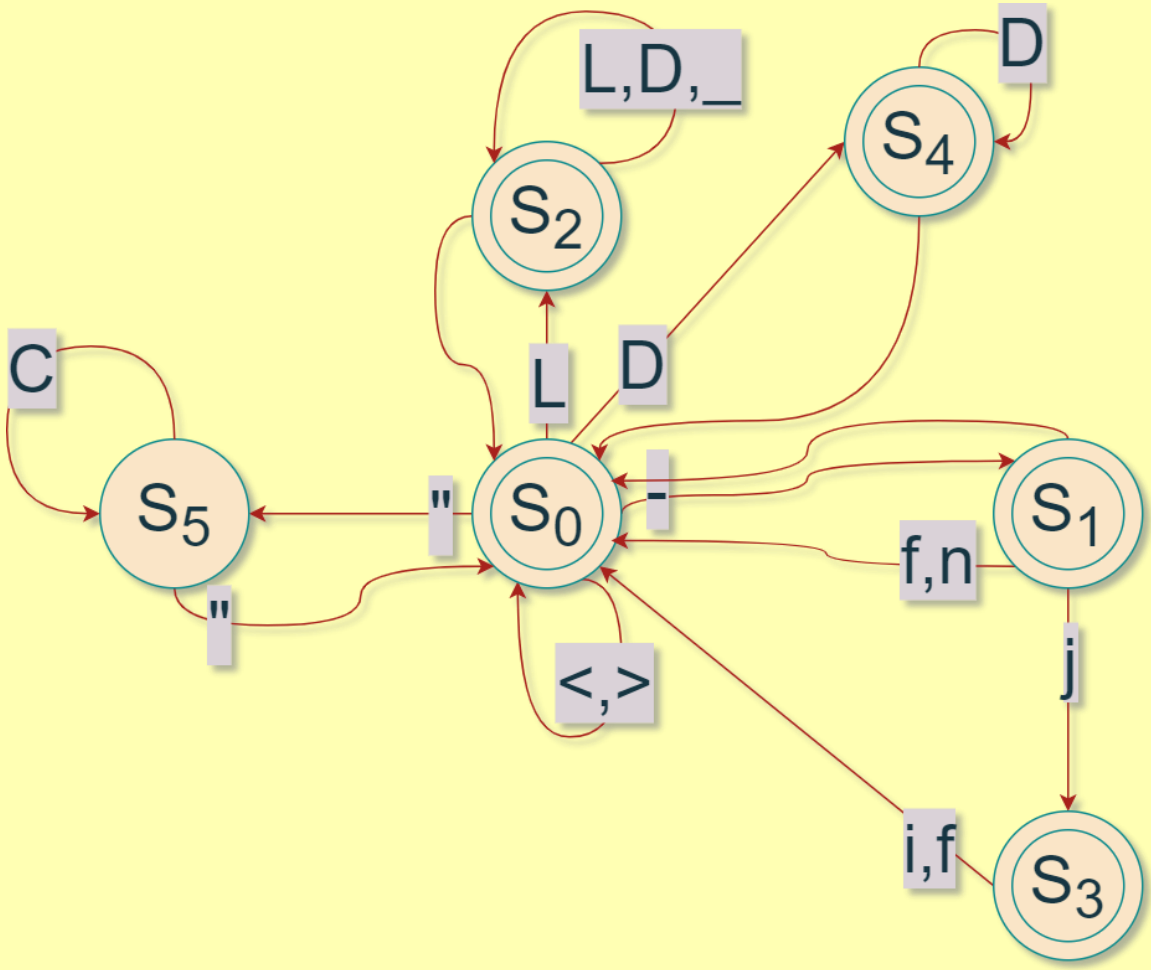
Por último, se arma el AFD con todos los estados obtenidos de la tabla de transiciones.



Llegados este punto se tiene una transición que no es de interés y se puede quitar, lo cual se muestra, el AFD óptimo:



Autómata AFD



Logrando así obtener dicho autómata óptimo, se creó el algoritmo para el análisis de los comandos entrantes de la caja de texto.

Gramática libre de contexto:

En la jerarquía de Chomsky, este viene siendo la gramática de segundo nivel, la cual indica las siguientes reglas:

1. En la parte izquierda un no terminal
2. En la parte derecha pude ser una cadena de terminales y no terminales.

De una manera más específica:

- N: No terminales, también llamadas variables sintácticas(estados).
- T: También conocidos como tokens, son elementos definidos por nuestro lenguaje.
- P: Son el conjunto de reglas de producción
- S: Símbolo inicial de la gramática.

En resumen, los tokens serán escritos con letras minúsculas, y los estados son escritos en mayúsculas.

En el proyecto se creará una GLC para verificar sintácticamente que este en orden cada token, a partir de las producciones del lenguaje. Por recomendación se deja un estado de inicio.

V como el conjunto de no terminales, osea variables, y T como el conjunto de terminales, osea las constantes.

V = {S, RESULTADO, JORNADA, GOLES, TABLA, PARTIDOS, TOP, ADIOS, BANDERAEXPORTAR, CONDICIONG, BANDERA_IJ, CONDICIONT, BANDERATOP}

T = {resultado, cadena, vs, temporada, menorque, entero, guion, mayorque, goles, local, visitante, total, tabla, top, superior, inferior, adios, partidos, banderaexportar, banderatop, banderainicial, banderafinal, identificador}

```
S ::= INICIO,
INICIO ::= RESULTADO eof
          | JORNADA eof
          | GOLES eof
          | TABLA eof
          | PARTIDOS eof
          | TOP eof
          | ADIOS eof
```

```
BANDERATOP ::= banderatop entero

CONDICIONT ::= superior
             | inferior

CONDICIONG ::= local
             | visitante
             | total

BANDERA_IJ ::= banderainicial entero banderafinal entero

BANDERAEXPORTAR ::= banderaexportar identificador
                  | BANDERA_IJ

RESULTADO ::= resultado cadena vs cadena temporada menorque entero guion entero mayorque

JORNADA ::= jornada entero temporada menorque entero guion entero guion BANDERAEXPORTAR

GOLES ::= goles CONDICIONG cadena temporada menorque entero guion entero mayorque

TABLA ::= tabla temporada menorque entero guion entero mayorque BANDERAEXPORTAR

PARTIDOS ::= partidos cadena temporada menorque entero guion entero mayorque BANDERAEXPORTAR

TOP ::= top CONDICIONT temporada menorque entero guion entero mayorque BANDERATOP

ADIOS ::= adios
```

Así completando la GLC para el proyecto.