MANUAL TÉCNICO

Little Code

Elías Abraham Vasquez Soto

201900131

PROYECTO 2 – LENGUAJES FORMALES Y DE PROGRAMACIÓN B-

ÍNDICE

OBJETIVOS 3

Objetivo General 3
Objetivos Específicos 3

INTRODUCCIÓN 4

DESCRIPCIÓN DE LA SOLUCIÓN 5

FUNCIONAMIENTO DE LA APLICACIÓN 6

Archivos y carpetas que componen la aplicación 6
Arboles de derivacion (Carpeta) *6*
Documentación (Carpeta) *6*
images (Carpeta) *6*
Reportes HTML (Carpeta) *6*
clases.py *7*
main.py *7*
Lógica del sistema 8
Tabla de Tokens *8*
Clases (clases.py) *8*
Main (main.py) *11*
Método del árbol – AFD 14
Expresión regular *14*
Árbol binario *14*
Tabla Follow Pos: *15*
Tabla de transiciones: *16*
Autómata finito determinista: *16*
Gramática Libre de Contexto 17
Tokens y lexemas en los que se basa la gramática *17*
Gramática *18*
Librerías utilizadas 19
Tkinter: *19*
Os: *19*
Traceback: *19*
IDE utilizado 20
Lenguaje de programación utilizado 21
Sistema Operativo utilizado 22
Requerimientos de sistema 22
Hardware: *22*
Software: *22*

Objetivos

Objetivo General

Brindar una guía que contenga la información del manejo de métodos, el flujo del programa y la manera en que se encuentra desarrollado el analizador léxico y sintáctico de la aplicación *Little Code*, para facilitar futuras actualizaciones que sean requeridas, así como el mantenimiento correcto de la misma.

Objetivos Específicos

- Mostrar una descripción completa del IDE, lenguaje de programación, sistema operativo y demás herramientas utilizadas para el desarrollo de la aplicación.
- Proporcionar una explicación amplia y clara de la metodología utilizada para desarrollar el autómata finito determinista utilizado para el análisis léxico de los comandos que pueden ser ingresados al programa, así como de la metodología para realizar la gramática, que define al lenguaje programado.
- Proporcionar una explicación técnica y formal de los métodos y atributos que conforman la aplicación, así como las clases y módulos implementados.

Introducción

El objetivo de este manual técnico es el de dar a conocer al lector que pueda requerir hacer modificaciones futuras al software, o bien, comprender la manera en que esta ha sido desarrollada, el detalle del desarrollo de la aplicación *Little Code* (la cual posee un lenguaje definido, en donde puede procesar distintas instrucciones ingresadas por el usuario), indicando el IDE utilizado para su creación, las librerías utilizadas, y demás.

La aplicación, la cual cuenta con una interfaz gráfica, tiene como finalidad leer los comandos que el usuario ingresa en el editor de texto integrado, y procesarlos por medio de un analizador léxico y sintáctico. Con estos datos, se guardan datos, y se pueden realizar distintas acciones con ellos, según el usuario lo requiera. Además, es posible generar reportes de Tokens leídos en el analizador léxico, árbol de derivación creado en el analizador sintáctico, así como de Errores (tanto léxicos como sintácticos) ocurridos en el mencionado proceso, todos en formato HTML, a excepción del árbol de derivación, que se genera en formato .PNG.

Descripción de la solución

Para poder desarrollar este proyecto, se desarrolló una interfaz gráfica con Tkinter, para que la interacción del usuario con la aplicación sea lo más intuitiva y agradable posible. Se analizó el código que el usuario ingresa, carácter por carácter, por medio de un autómata finito determinista, con el objetivo de separar correctamente los distintos tokens que componen a las instrucciones que el usuario puede ejecutar en la aplicación. Dichos tokens, se analizaron por medio de una gramática, que identifica el orden en que deben venir, para así, realizar las acciones que el usuario haya solicitado. En ambos procesos, pueden ocurrir diversos errores, los cuales son reportados al usuario de distintas formas, según el tipo de error detectado.

Entre las consideraciones más importantes que se tomaron en cuenta para el desarrollo de la aplicación, se encuentran:

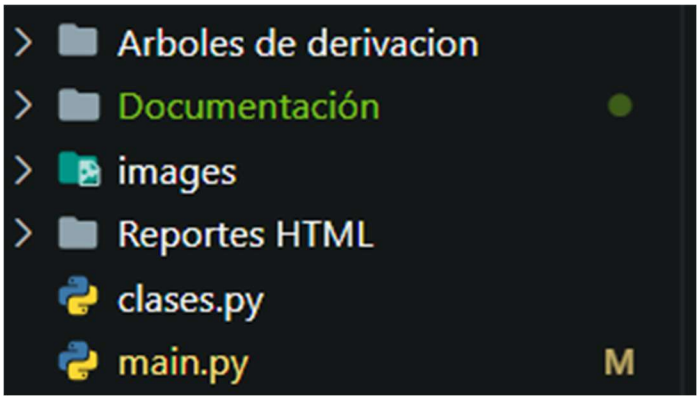
- El código que el usuario ingresa al programa, cuenta con una sintaxis específica, por lo que la definición correcta de la gramática que define al lenguaje del programa desarrollado es vital.

- El usuario puede ingresar las instrucciones que desee, en el orden que desee, por lo que, debe existir una anticipación a los posibles errores de ejecución que puedan generarse, así como a los posibles errores léxicos y sintácticos que puedan ser detectados. En algunos casos, la recuperación de dichos errores debe ser posible.

Funcionamiento de la aplicación

Archivos y carpetas que componen la aplicación

La aplicación cuenta con 2 archivos en formato PY, y 4 carpetas, siendo todas parte vital para la correcta ejecución de esta.



Arboles de derivacion (Carpeta)

En esta carpeta, se almacenan los árboles de derivación, cada vez que el usuario lo solicite. Este se guarda tanto en formato .dot, donde puede ver el código del árbol en graphviz, así como la imagen del árbol en formato .png.

Documentación (Carpeta)

En esta carpeta, se almacenan el reporte de Tokens y el de Errores, cada vez que el usuario lo solicite.

images (Carpeta)

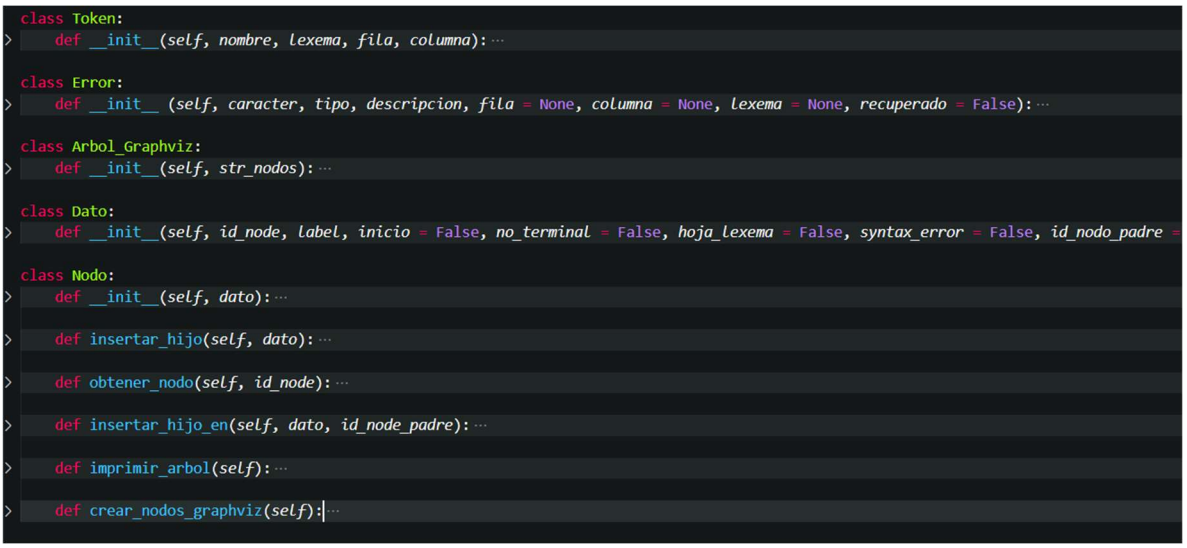
En esta carpeta, se almacenan todas las imágenes que se utilizan en la interfaz gráfica.

Reportes HTML (Carpeta)

En esta carpeta, se almacenan el reporte de Tokens y el de Errores, cada vez que el usuario lo solicite.

clases.py

Archivo en donde se almacenan todas las clases utilizadas en la aplicación.



Ver **Lógica del sistema** para más detalle de estas.

main.py

Archivo en donde se encuentra la lógica principal de la aplicación, así como el desarrollo de la interfaz gráfica. Ver **Lógica del sistema** para más detalle de esta.

Lógica del sistema

Tabla de Tokens

En esta tabla, se detallan todos los tokens que el analizador léxico es capaz de reconocer, así como la expresión regular que los define, y un lexema de ejemplo.

- Alfabeto a usar en la expresión regular:

N = {0,1,2,3,4,5,6,7,8,9}

L = {a,b,c,...,z,A,B,C,...,Z}

S = { '=','[',']',',',':','{','}', '(', ')' }

Token (Nombre)	Expresión Regular	Lexema Ejemplo
Cadena	"(^")*"	"Campo 1"
Número	(-)? N+ ('.' N+)?	25.00
Id	(L | '_') (L | N | '_')*	Proyecto
Símbolo	S	[
Comentario Línea**	'#' (^\n)*	# Esto es un comentario
Comentario Multilínea**	''' [^(''')]* '''	''' Comentario '''

**Los comentarios no son tokens reconocidos, pero sus expresiones regulares se toman en cuenta en el autómata finito determinista para poder ser ignorados.

Clases (clases.py)

Para el desarrollo de la aplicación, se utilizó el paradigma de programación orientado a objetos. Para esto, se crearon distintas clases en el archivo **clases.py**.

- class Token:

  Define los atributos de los distintos objetos "Token" a reconocer en el analizador léxico.

  - Atributos:
    - Nombre: str

- Lexema: str | int | float
- Fila: int
- Columna: int
- Id_token: int*

*Para la gramática se utilizaron ids de tokens, para identificar todas las posibles instrucciones que el usuario puede ingresar (Ver **Gramática Libre del Contexto**).

- class Dato:

  Clase utilizada para la generación del árbol de derivación, define los atributos distintos objetos "Dato", que representan el tipo de nodo que posee el árbol.

  Atributos:

  - Id_node: int
  - Label: str
  - Inicio: boolean
  - No_terminal: boolean
  - Hoja_lexema: boolean
  - Syntax_error: boolean
  - Id_nodo_padre: int | None

- class Nodo:

  Clase utilizada para la generación del árbol de derivación, define los atributos distintos objetos "Nodo", que representan los nodos que posee el **árbol n-ario** de derivación.

  - Atributos:
    - Dato: Dato()
    - Nodos_hijo: list
    - Str_nodos_graphviz: str

- Métodos:
    - Insertar_hijo(dato): Método que se encarga de insertar el Dato que recibe como parte de los nodos hijo, convirtiéndolo antes un objeto Nodo.
    - Obtener_nodo(id_node): Función que recibe como parámetro el id de un nodo, y se encarga de recorrer todo el árbol n-ario, partiendo de la raíz, hasta retornar el nodo que coincida con el id.
    - Insertar_hijo_en(dato, id_node_padre): Función que recibe como parámetro el dato que se desea insertar, y el id del nodo padre. Esta función, por medio de las dos descritas anteriormente, ingresa un nodo en una posición en específico del árbol.
    - Crear_nodos_graphviz(): Función que recorre todo el árbol, partiendo de la raíz, concatenando para cada nodo, un string con el código de graphviz necesario para graficar el nodo. Al finalizar, retorna el string.

- class Arbol_Graphviz:

  Clase utilizada para la generación del árbol de derivación, define el atributo del árbol que se genera en graphviz.

  - Atributos:
    - Str_graphviz: str

ÁRBOL N-ARIO: Es una estructura de datos donde cada nodo posee un número indeterminado de hijo, siendo esta una estructura recursiva.

Un árbol n-ario puede tomarse como un árbol de n elementos asociados a cada uno de sus componentes. Los conceptos de este árbol usados para el proyecto, específicamente para el árbol de derivación sintáctico, son:

*Nodo

*Raíz

*Hoja

Main (main.py)

En el archivo **main.py** es donde se desarrolló toda la lógica correspondiente al analizador léxico, sintáctico, interfaz gráfica, generación de reportes, etc. A continuación, se detalla la estructura de este archivo, así como la relación entre métodos y variables que en él se encuentra.

- Variables globales

```
texto_lfp = ""
codigo = ""
palabras_reservadas = ["Claves", "Registros", "imprimir", "imprimirln", "conteo", "promedio",
                       "contarsi", "datos", "sumar", "max", "min", "exportarReporte"]
index = 0
tokens_leidos = []
errores_encontrados = []
encabezados = []
registros = []
registro_aux = []
errores_registros = 0
errores_claves = 0
flag_final = False
nodo_raiz = None
id_node_padre = None
id_node_padre_aux_1 = None
id_node = 0
err_syntax = False
```

  - Texto_lfp, código y palabras reservadas:
    - Variables utilizadas para definir las palabras reservadas que el lenguaje puede reconocer, así como para guardar el código que el usuario ingrese a la aplicación.
  - Tokens_leidos: list
    - Arreglo de objetos de tipo Token, en donde se almacenarán los tokens reconocidos por el analizador léxico.
  - Errores_encontrados: list
    - Arreglo de objetos de tipo Error, en donde se almacenan los errores léxicos y sintácticos encontrados.
  - Encabezados, registros:
    - Arreglos en donde se almacenan las claves y los registros que el usuario ingrese al programa.

- Index, registro_aux, errores_registros, errores_claves, flag_final:
  - Variables de control utilizadas en el análisis sintáctico.
- Nodo_raiz, id_node_padre, id_node_padre_aux1, id_node, err_syntax:
  - Variables utilizadas para almacenar y controlar la creación del árbol de derivación, durante el análisis sintáctico.

- class Interfaz:

Esta clase se encuentra declarada dentro del archivo main.py, y contiene todos los métodos y funciones que hacen posible tanto la generación de la interfaz gráfica, como del analizador léxico, el analizador sintáctico y de reportes. Los métodos que en esta clase se encuentran son:

- __init__(ventana):

  Recibe como parámetro una ventana, la cual es una instancia de la clase Tk(), para insertar en ella los distintos widgets, según se requieran.

- abrirArchivo():

  Abre un cuadro de diálogo, en donde el usuario podrá seleccionar el archivo LFP que desee cargar al programa.

- Obtener_codigo():

  Se manda a llamar cuando el usuario decide analizar el código, obtiene el código ingresado, y llama tanto al analizador léxico, como al sintáctico.

- is_symbo(caracter), is_number(caracter), is_letter(caracter):

  Recibe como parámetro un carácter, y devuelve un booleano, indicando si es un símbolo, un número, o bien, una letra, esto para reconocer los tokens que correspondan.

- Analizador_lexico():

  Método que corresponde al **analizador léxico**, por lo que es aquí donde se encuentran implementados los autómatas finitos deterministas generados con el método del árbol (ver Método del árbol).

- Analizador_sintactico():

Método que corresponde al **analizador sintáctico**, por lo que es aquí donde se encuentra la llamada a la gramática implementada, luego de haber realizado el análisis léxico, y de haber reconocido todos los tokens.

- Inicio(), cadenas(), valor(), valores(), mult_registros(), otra_ins():

Métodos que corresponden a los No terminales de la gramática que define al lenguaje. Son llamados durante el análisis sintáctico.

- Panic_mode():

Método que es llamado cuando se encuentra un error sintáctico, ignorando los tokens hasta que se encuentre un token de sincronización, y el análisis pueda reanudarse.

- Reporte_ejecucion():

Método que genera el HTML correspondiente al reporte de Claves y Registros, cuando el usuario lo solicite.

- Reporte():

Método que detecta el reporte solicitado por el usuario, y en base a eso, realiza la llamada a los métodos respectivos.

- Reporte_tokens(), reporte_errores(), arbol_derivacion():

Método que genera el HTML de reporte de tokens y errores, o bien, el del árbol de derivación, según el usuario lo haya solicitado.

- Main

Declara una ventana, instancia de la clase Tk(), para crear la aplicación, instancia de la clase Interfaz, y ejecutar el programa.

Método del árbol - AFD

Los pasos para la expresión regular planteada fueron:

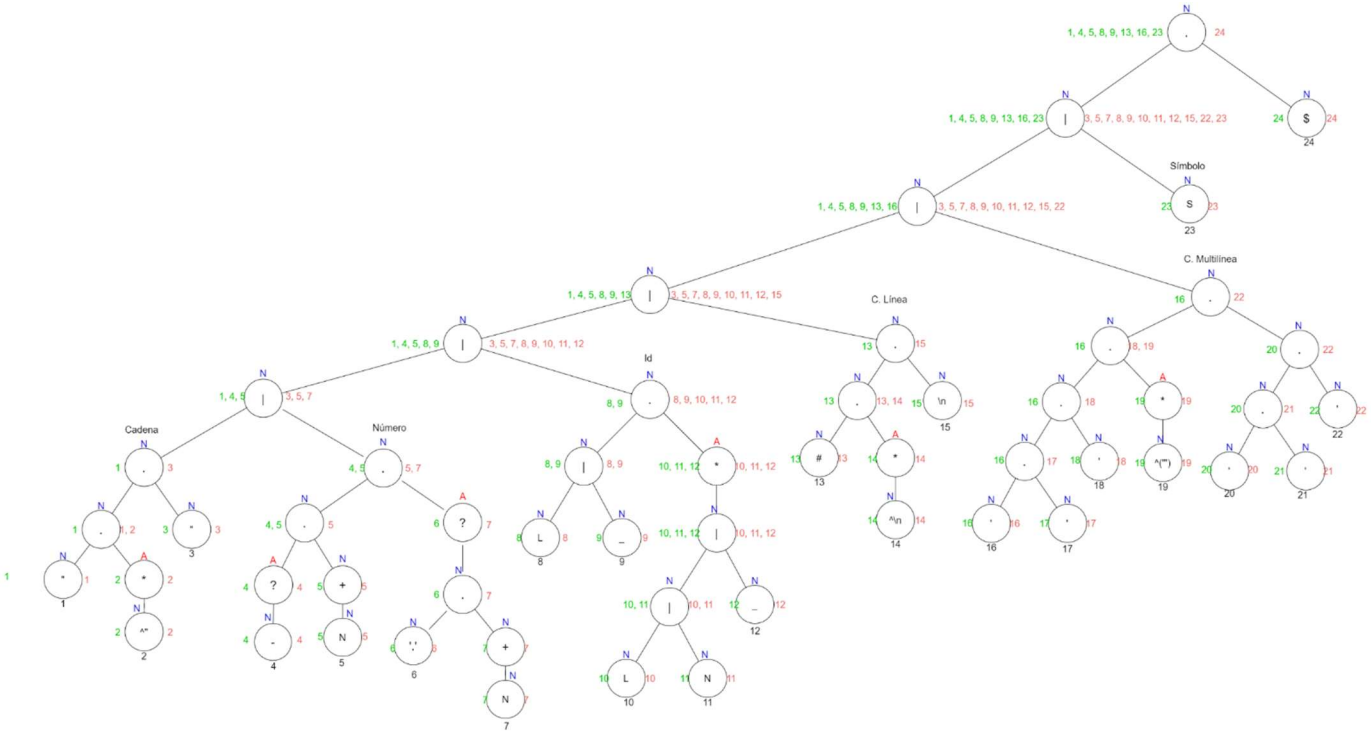
- Concantenar a la expresión regular un símbolo final
- Generar el árbol binario
- Realizar la tabla Follow Pos y la tabla de transiciones
- Generar el autómata a implementar en el código

Expresión regular

["(^") * " | (−)? N + ('. ^' N+)? |((L ⊣ |'_') (L | N |'_') *)| '#' (^\n) * \n | '''[^(´´)] * ''' | S] $

*Expresión regular formada a partir de las expresiones regulares planteadas en la tabla de tokens (ver **Tabla de Tokens**).

Árbol binario



*En la carpeta **Documentación** del proyecto, se encuentra la imagen del árbol binario, por si se desea obtener una mejor visión de este.

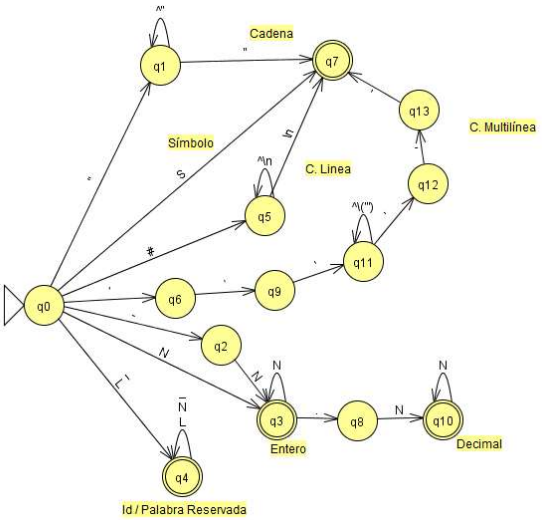
Tabla Follow Pos:

HOJA	VALOR	SIGUIENTE
1	“	2, 3
2	^”	2, 3
3	“	24
4	-	5
5	N	5, 6, 24
6	.	7
7	N	7, 24
8	L	10, 11, 12, 24
9	_	10, 11, 12, 24
10	L	10, 11, 12, 24
11	N	10, 11, 12, 24
12	_	10, 11, 12, 24
13	#	14, 15
14	^\n	14, 15
15	\n	24
16	‘	17
17	‘	18
18	‘	19, 20
19	^(’’)	19, 20
20	‘	21
21	‘	22
22	‘	24
23	S	24
24	$	-

Tabla de transiciones:

ESTADO	“	^”	-	N	.	L	_	#	^\n	\n	‘	^\(”)	S	Aceptación
q0={1, 4, 5, 8, 9, 13, 16, 23}	q1	-	q2	q3	-	q4	q4	q5	-	-	q6	-	q7	
q1={2, 3}	q7	q1	-	-	-	-	-	-	-	-	-	-	-	
q2={5}	-	-	-	q3	-	-	-	-	-	-	-	-	-	
q3={5, 6, 24}	-	-	-	q3	q8	-	-	-	-	-	-	-	-	Aceptación
q4={10, 11, 12, 24}	-	-	-	q4	-	q4	q4	-	-	-	-	-	-	Aceptación
q5={14, 15}	-	-	-	-	-	-	-	-	q5	q7	-	-	-	
q6={17}	-	-	-	-	-	-	-	-	-	-	q9	-	-	
q7={24}	-	-	-	-	-	-	-	-	-	-	-	-	-	
q8={7}	-	-	-	q10	-	-	-	-	-	-	-	-	-	
q9={18}	-	-	-	-	-	-	-	-	-	-	q11	-	-	
q10={7, 24}	-	-	-	q10	-	-	-	-	-	-	-	-	-	Aceptación
q11={19, 20}	-	-	-	-	-	-	-	-	-	-	q12	q11	-	
q12={21}	-	-	-	-	-	-	-	-	-	-	q13	-	-	
q13={22}	-	-	-	-	-	-	-	-	-	-	q7	-	-	

Autómata finito determinista:



*En la carpeta **Documentación** del proyecto, se encuentra la imagen del autómata finito determinista, por si se desea obtener una mejor visión de este.

Gramática Libre de Contexto

Tokens y lexemas en los que se basa la gramática

Token	ID
Cadena	1
Entero	2
Decimal	3
Palabra reservada: Claves	4
Palabra reservada: Registros	5
Palabra reservada: imprimir	6
Palabra reservada: imprimirln	7
Palabra reservada: conteo	8
Palabra reservada: promedio	9
Palabra reservada: contarsi	10
Palabra reservada: datos	11
Palabra reservada: sumar	12
Palabra reservada: max	13
Palabra reservada: min	14
Palabra reservada: exportarReporte	15
Id***	16***
Símbolo: Igual	17
Símbolo: Abrir Corchete	18
Símbolo: Cerrar Corchete	19
Símbolo: Coma	20
Símbolo: Punto y coma	21
Símbolo: Abrir Llave	22
Símbolo: Cerrar Llave	23
Símbolo: Abrir Paréntesis	24
Símbolo: Cerrar Paréntesis	25

***Token Id se utilizó para reconocerlo como error sintáctico, mas no se utiliza en la gramática.

*La gramática, en el código fuente, se trabajó utilizando los ID's de tokens.

Gramática

Terminales = {Cadena, Entero, Decimal, PR_Claves, PR_Registros, PR_imprimir, PR_imprimirln, PR_conteo, PR_promedio, PR_contarsi, PR_datos, PR_sumar, PR_max, PR_min, PR_exportarReporte, S_Igual, S_AC_[, S_CC_], S_Coma, S_PtoComa, S_AL_{, S_CL_}, S_AP_(, S_CP_)}

No Terminales = {INICIO, OTRA_INS, CADENAS, VALOR, MULT_REGISTROS, VALORES, VALOR_F}

Inicio = <INICIO>

Producciones:

<INICIO> ::= PR_Claves S_Igual S_AC_[Cadena <CADENAS> S_CC_] <OTRA_INS>

| PR_Registros S_Igual S_AC_[S_AL_{ <VALOR> S_CL_} <MULT_REGISTROS> S_CC_] <OTRA_INS>

| PR_imprimir S_AP_(Cadena S_CP_) S_PtoComa <OTRA_INS>

| PR_imprimirln S_AP_(Cadena S_CP_) S_PtoComa <OTRA_INS>

| PR_conteo S_AP_(S_CP_) S_PtoComa <OTRA_INS>

| PR_promedio S_AP_(Cadena S_CP_) S_PtoComa <OTRA_INS>

| PR_contarsi S_AP_(Cadena S_Coma <VALOR_F> S_CP_) S_PtoComa <OTRA_INS>

| PR_datos S_AP_(S_CP_) S_PtoComa <OTRA_INS>

| PR_sumar S_AP_(Cadena S_CP_) S_PtoComa <OTRA_INS>

| PR_max S_AP_(Cadena S_CP_) S_PtoComa <OTRA_INS>

| PR_min S_AP_(Cadena S_CP_) S_PtoComa <OTRA_INS>

| PR_exportarReporte S_AP_(Cadena S_CP_) S_PtoComa <OTRA_INS>

<CADENAS> ::= S_Coma Cadena <CADENAS> | ε

<VALOR> ::= Cadena <VALORES> | Entero <VALORES> | Decimal <VALORES>

<VALORES> ::= S_Coma <VALOR> | ε

<MULT_REGISTROS> ::= S_AL_{ <VALOR> S_CL_} <MULT_REGISTROS> | ε

<VALOR_F> ::= Cadena | Entero | Decimal

<OTRA_INS> ::= <INICIO> | ε

Librerías utilizadas

```
from tkinter import *
from tkinter import filedialog, messagebox
from clases import *
import traceback, os
```

Tkinter:

Tkinter es el paquete más utilizado para crear interfaces gráficas en Python, por lo que fue el módulo escogido para la elaboración del presente proyecto. Es orientada a objetos basada en Tc y Tk.

Os:

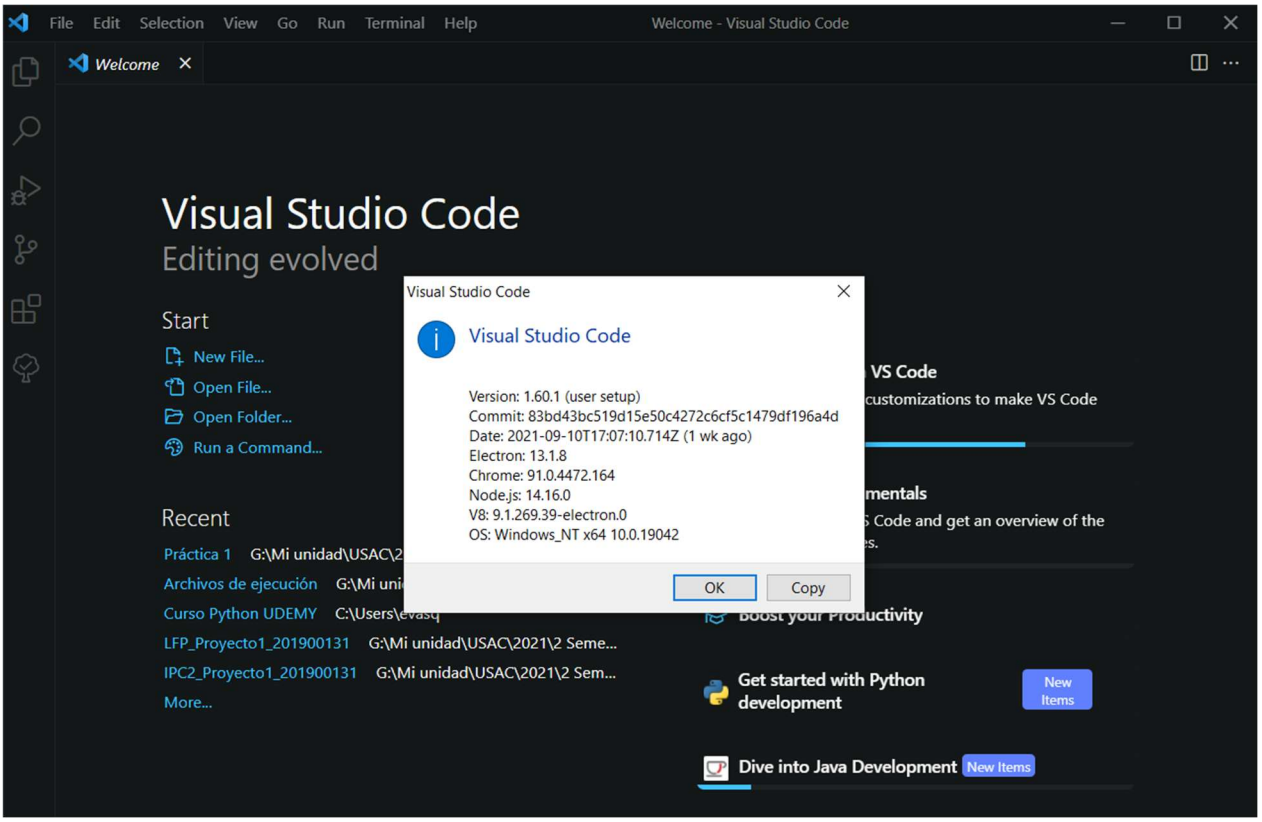
El módulo os de Python permite realizar operaciones dependientes del Sistema Operativo como crear una carpeta, listar contenidos de una carpeta, conocer acerca de un proceso, finalizar un proceso, etc. Para el proyecto, se utilizó para la obtención de rutas absolutas de ciertos archivos, así como para navegar entre directorios.

Traceback:

Este módulo brinda una interfaz estándar para extraer, formatear y mostrar trazas de pilas de programas de Python. Dicho módulo copia el comportamiento del intérprete de Python cuando muestra una traza de pila. Para el proyecto, se utilizó para reportar en consola las excepciones posibles que se vayan generando durante la ejecución de la aplicación.

IDE utilizado

El IDE con el que se desarrolló el proyecto fue Visual Studio Code versión 1.60.1, debido a que el desarrollo de la parte lógica resulta más sencillo y por su apoyo al programador gracias a su asistente que detecta errores semánticos y sintácticos del código, por lo cual ayudan y hacen que la duración de la fase de programación sea más corta. Además, la posibilidad de personalización del IDE hace que sea el favorito del desarrollador de esta aplicación.



Requisitos para Visual Studio Code:

a) Hardware:

Visual Studio Code es una pequeña descarga (<200 MB) y ocupa un espacio en disco de <500 MB. VS Code es liviano y debería ejecutarse fácilmente en el hardware actual. La recomendación:

- Procesador de 1,6 GHz o más rápido
- 1 GB de RAM

b) Plataformas

VS Code se ha probado en las siguientes plataformas:

- OS X El Capitan (10.11+)
- Windows 7 (con .NET Framework 4.5.2), 8.0, 8.1 y 10 (32 y 64 bits)
- Linux (Debian): Ubuntu Desktop 16.04, Debian 9
- Linux (Red Hat): Red Hat Enterprise Linux 7, CentOS 8, Fedora 24

c) Requisitos adicionales para Windows

Se requiere Microsoft .NET Framework 4.5.2 para VS Code. Si está utilizando Windows 7, asegúrese de que .NET Framework 4.5.2 esté instalado.

d) Requisitos adicionales para Linux

GLIBCXX versión 3.4.21 o posterior

GLIBC versión 2.15 o posterior

Lenguaje de programación utilizado

Python, versión 3.9.7, la más reciente a la fecha en que este manual es redactado. Python es un lenguaje de programación interpretado de alto nivel. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa, etc., facilitando en gran manera el desarrollo de esta aplicación.

Para más información del lenguaje utilizado, instalación, etc.: https://www.python.org

Sistema Operativo utilizado

El sistema operativo en el que se realizó el desarrollo de la aplicación fue Windows 10 Home de 64 bits.

Requisitos para Windows 10:

- Último sistema operativo: Asegúrate de que estás ejecutando la versión más reciente: Windows 7 SP1 o Windows 8.1 Update.
- Procesador: de 1 gigahercio (GHz), o procesador o SoC más rápido
- RAM: 1 gigabyte (GB) para 32 bits o 2 GB para 64 bits
- Espacio en disco duro:16 GB para el sistema operativo de 32 bits o 20 GB para el sistema operativo de 64 bits
- Tarjeta gráfica: DirectX 9 o posterior con controlador WDDM 1.0
- Pantalla: 800 x 600

Requerimientos de sistema

Hardware:

- Procesador: 1 gigahercio (GHz) o más rápido
- RAM: 2 GB (mínimo)
- Espacio en disco duro: 100MB libre (mínimo)

Software:

- Sistema operativo:
  - Windows 7 (con .NET Framework 4.5.2), 8.0, 8.1 y 10 (32 y 64 bits)
  - Linux (Debian): Ubuntu Desktop 16.04, Debian 9
  - Linux (Red Hat): Red Hat Enterprise Linux 7, CentOS 8, Fedora 24
- Python 3
- Librerías instaladas (ver sección Librerías utilizadas)