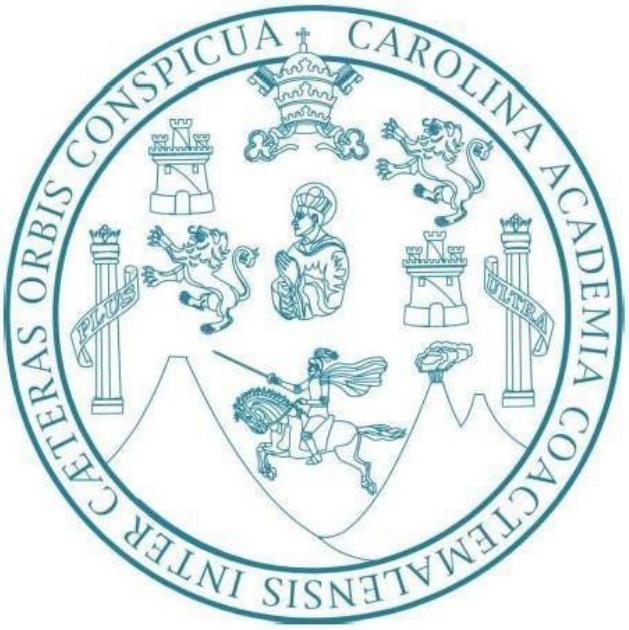
Universidad de San Carlos de Guatemala

División de Ciencias de la Ingeniería
Lenguajes Formales y de Programación



DOCUMENTACIÓN PRÁCTICA 1

PARSER-PY

Carné: 201831260

Nombre: Michael Kristopher Marín Reyes

Sección: A

PARSER-PY

El siguiente documento muestra información y descripción sobre los algoritmos implementados y funcionamiento de los métodos y funciones que permiten el funcionamiento del analizador léxico y sintáctico para el lenguaje de programación python, este analizador es un escáner el cual reconoce y clasifica componentes léxicos de un programa, siendo alguno de estos; identificadores, operadores, palabras clave, entre otros. Posteriormente estos son enviados al analizador sintáctico para que pueda detectar bloques de código que correspondan a asignaciones, declaraciones, ciclos, funciones y métodos.

Entorno de desarrollo

Para el desarrollo de esta aplicación se utilizó lo siguiente:

- JDK 17.
- Java Swing para entorno gráfico de la aplicación.
- Maven para la gestión y construcción del proyecto.
- Graphviz para creación de gráfico de tokens.
- IDE Apache NetBeans.
- Sistema operativo Ubuntu 22.04 LTS.
- PlantUML para la realización de diagrama de clases.

Paquetes

backend

Este paquete contiene cuatro clase y y un paquete que se encargan de la lógica para el funcionamiento para leer archivos de entrada y la clase main.

Clase LeerArchivoTexto

Esta clase únicamente cuenta con un solo método y un solo atributo, este se encarga de cargar un archivo que se mostrará en el editor de texto.

abrirArchivo(String ruta)

Este método recibe una cadena, esta cadena representa el path de donde se encuentra el archivo tipo .txt, en este método se lee cada línea del archivo y lo devuelve para ser mostrado posteriormente en el editor de texto que se encuentra en la parte del frontend y que el usuario pueda ver el contenido del archivo de manera visual dentro de la aplicación.

Clase Main

Esta clase es la encargada de iniciar el funcionamiento de la aplicación, únicamente contiene el método main.

main(String[] args)

Este método es el método principal que se encarga de iniciar la funcionalidad del programa.

backend/lexico

En este paquete se encuentran las clases que conforma el analizador léxico para el funcionamiento de Parser-Py.

Clase AnalizadorLexico

Esta clase se encarga de leer el texto que es recibido como parámetro, leer carácter por carácter y determinar los diferentes tipos de tokens, tiene los siguientes métodos:

analizar(String cadena)

Este método se encarga de leer carácter por carácter la cadena recibida como parámetro, clasifica palabras encontradas y las guarda en un arrayList llamada listaToken, la cadena recibida como parámetro es separada carácter por carácter y así cuando vaya detectando caracteres anide caracteres y detectar si existe alguna palabra reservada, cadena, entre otros a través de una sentencia de selección switch y también if else, al determinar el token.

crearToken(String token, int linea, int columna)

Este método se encarga recibir un token o cadena ya determinada por el método anterior, seguidamente realiza cierto tipo de validaciones por medio de sentencias if else, este token identificado se agrega al array que es de tipo Token, se crea un nuevo token y en el constructor se envía el tipo de token, lexema, fila y columna. Esta lista creada sera utilizada para ver los

tokens obtenidos y por verlos en la tabla de reportes designada para ello, ademas de generar un grafico que desglosa el token carácter por carácter.

esNumero(String lexema)

Este método se encarga de recibir un lexema y convertir la cadena en un entero o en un decimal, en caso que ocurra una excepción de tipo NumberFortmatException entonces el lexema recibido no es un lexema que pueda ser convertido en entero o decimal, este método es un booleano.

esCadena(String lexema)

Al igual que el método anterior este se encarga de verificar que verificar que si el lexema está dentro comillas dobles (“ ”) o comillas simples (‘ ’) entonces determinará que el lexema recibido es un token de tipo cadena.

esID(Strin lexema)

Se encarga de definir si el lexema recibido es de tipo identificador, es decir, que en el lenguaje python sea una variable.

iniciarDiccionarios()

Se encarga únicamente de crear un diccionario clave valor utilizando HashMap en la que según un String, este devolverá el valor de que represente el String, en este caso devolvera un tipo de Token.

Clase Token

Esta clase se encarga de crear un token, contiene las características que identifican a un token, además de colocar un patrón que determina al token. Los atributos de un token son: lexema, linea, columna, token, patron.

Métodos getter

Cada atributo de esta clase contiene métodos getter, estos sirven para obtener el valor contenido dun atributo en específico.

Métodos setter

Cada atributo de esta clase contiene el método setter, estos sirven para modificar los datos de un atributo de un token en específico.

backend/lexico/identificadores

Este paquete contiene clases tipo Enum, estas clases determinan los tipos de identificadores que tendrá el lenguaje python, tanto su clasificación y el tipo al que corresponde.

Enum AritmeticosEnum

Esta clase de tipo enum contiene valores que identifican el tipo de token que será utilizado, los valores son: suma (+), resta (-), exponente (**), división (/, //), modulo (%), multiplicación (*).

Enum ComparacionEnum

Esta clase se encarga de contener los valores que se utilizarán como token, los valores son: igual (==), asignacion (=), diferente (!=), mayor que (>), menor que (<), mayor igual que (>=), menor igual que (<=).

Enum LogicoEnum

Esta clase contiene los valores que servirán como operadores lógicos, and, or y not.

Enum OtroEnum

Esta clase contiene otros tipos de tokens, estos son: entero, decimal, cadena, id, comentario, parentesis ziquierdo, parentesis derecho, comentario, llave izquierda, llave derecha, corchete izquierdo, corchete derecho, coma, punto, punto y coma, dos puntos.

Enum PalabraClaveEnum

Esta clase contiene los tokens que en el lenguaje python son utilizados como palabras reservadas.

Enum TipoToken

Esta clase contiene la clasificación del token al que pertenece, es decir, que cada token generado pertenece a una clasificación como lógico, aritmético, comparación, otros.

backend/sintactico

En este paquete se almacenan todas las clases que se encargarán de generara la lógica para el funcionamiento del analizador sintáctico para Parser-Py.

Clase AnalizadorSintactico

Esta clase se encarga de manejar los tokens generados por el analizador léxico y utiliza para determinar bloque de código que correspondan a una asignación, declaración, sentencias, ciclos, funciones o métodos.

Clase BloqueCodigo

Esta clase se encarga de crear un objeto con atributos que permite almacenarse en un ArrayList que posteriormente servirá para verlos en una tabla de reportes.

Clase DetectarBloqueCodigo

Esta clase se encarga de detectar bloques de código de los tokens generados por el analizador léxico, al encontrar bloques de código se realizará el análisis sintáctico.

backend/sintactico/asignaciondeclaracion

Este paquete contiene clases que cumplirán con la función de autómatas, cada una identificará el bloque de código enviado por la clase AnalizadorSintactico.

Clase Asignacion

Esta clase se encarga de ser un autómata que verificará si el bloque recibido es un bloque de asignación, de lo contrario no será un bloque válido para una asignación.

Método esAceptado

Este método es un getter sobre el atributo esAceptado, sirve para determinar el estado final del autómata.

Método asignacion

Este método se encarga de recibir el primer token en un primer estado y si es correcto, llamará al siguiente estado.

Método operadorIdentificador

Este método se encarga de verificar que despues de un identificador exista un operador asignación.

Método valorAsignado

Este método se encarga de verificar que el token que se encuentre en ese estado sea válido, de ser así seguiría al siguiente estado.

Método valorAsignado

Este método se encarga de verificar que el valor que se asigne a la variable sea válido, al verificarlo continua al siguiente estado.

Método esComparacion

Este método es el estado final, determina que el valor sea válido, este es un método booleano y por tanto al obtener el valor esperado retorna true.

Clase Declaracion

Esta clase se encarga de ser un autómata que verificará si el bloque de código recibido corresponde a una declaración.

Método esAceptado

Este método es un getter sobre el atributo esAceptado, sirve para determinar el estado final del autómata.

Método declaracion

Este método se encarga de recibir el primer token y analizar si es válido para dirigirse al segundo estado.

Método operadorAsignacion

Este método se encarga de verificar que el segundo token es un token de tipo asignación.

Método valorAsignado

Este método se encarga de verificar que el último token sea un estado de aceptación y de ser así el valor booleano cambia su valor de false a true.

Clase OperadorTernario

Esta clase se encarga de ser un autómata que verifica si los tokens enviados corresponden a un operador ternario.

Método ternario

Se encarga de verificar que el primer valor corresponda a un token válido.

Método condicionTrue

Verifica que la siguiente condición sea algún identificador o un valor que sea una opción válida para la expresión regular.

Método siguienteCondicional

En este caso se verificará que exista 2 palabras reservadas que exige la expresión regular y de ser así procede al siguiente estado.

Método condicionElse

Verifica que el token siguiente contenga la palabra reservada else, para poder seguir al siguiente estado.

Método utlimoToken

Se encarga de verificar que el último token recibido sea un token válido para que el estado de aceptación cambie y por tanto el atributo de la clase llamado aceptado cambie de false a true.

backend/sintactico/ciclos

Este paquete contiene las clases que servirán como autómatas que detectarán bloques de código correspondientes a ciclos.

Clase CicloFor

Esta clase es tiene la función de encontrar si el bloque de código corresponde a un ciclo for.

Clase CicloWhile

Esta clase detectará si el bloque de código corresponde a un ciclo while.

backend/sintactico/condicionalesfuncionesmetodos

Este paquete contiene las clases que detectaran condicionales, funciones y métodos.

Clase CondicionalIfElse

Esta clase detecta si un bloque de código corresponde a una sentencia if-else, if-elif-else.

Clase FuncionMetodo

Esta clase detecta si un bloque de código corresponde a una función o un método.

frontend

Este paquete contiene las clases y subpaquetes que conformarán al entorno gráfico que verá el ususario.

Clase EditorPanel

Esta clase hereda de JPanel y se encarga de mostrar al usuario dos áreas de texto, en la primera es donde puede escribir texto o donde se ubicará el texto de un archivo que haya cargado, contiene botones los cuales uno sirve para limpiar tanto el área de consola como el area del editor de texto, el otro botón comienza el analisis del texto ingresado para identificar los tokens.

initComponents()

Este método se encarga de inicializar todos aquellos componentes que se utilizan en este panel.

ejecutarBotonActionPerformed(ActionEvent evt)

Este método se ejecuta cuando se presiona el botón analizar, dentro de este método realiza una validación para verificar si existe texto en el area de editor de texto, si el area de texto está vacía mostrará un mensaje indicando al usuario que no ha escrito nada, de lo contrario este comenzará el analisis léxico y mostrará el resultado de tokens encontrados en el area de consola.

limpiarBotonActionPerformed(ActionEvent evt

Este método se encarga de limpiar el texto en el área de editor de texto, consola y el arrayList que contiene los tokens, el arraList también se limpia debido a que para comenzar un nuevo analisis este no confunda datos con resultados de analisis previos.

mostrarColumna()

Este método se encarga de mostrar la columna en la que se encuentra el puntero, mostrando el valor en un Jlabel.

setAreaEditor(String textoLeido)

Este método se encarga de mostrar en el area de editor de texto el contenido de un archivo que el usuario haya cargado.

Clase NumeroLinea

Es una clase que extiende `JPanel`. Esta clase se utiliza para crear un panel que muestra el número de línea correspondiente en un área de texto, similar a un editor de texto. El panel de números de línea se coloca junto al componente de texto y muestra el número de línea actual resaltado en color.

`NumeroLinea(JTextComponent component, int minimumDisplayDigits)`

Permite especificar el número mínimo de dígitos para la numeración de líneas.

`getUpdateFont()`

Devuelve si la actualización de la fuente está habilitada.

`setUpdateFont(boolean updateFont)`

Se encarga de habilitar o deshabilitar la actualización de la fuente.

`getBorderGap()`

Devuelve el espacio entre el borde del panel y los números de línea.

`setBorderGap(int borderGap)`

Establece el espacio entre el borde del panel y los números de línea, recalculando el ancho preferido del panel.

`getCurrentLineForeground()`

Devuelve el color de primer plano para la línea actual.

`setCurrentLineForeground(Color currentLineForeground)`

Establece el color de primer plano para la línea actual.

`getDigitAlignment()`

Devuelve la alineación de los números de línea (izquierda, centro o derecha).

`setDigitAlignment(float digitAlignment)`

Establece la alineación de los números de línea.

`getMinimumDisplayDigits()`

Devuelve el número mínimo de dígitos para la numeración de líneas.

`setMinimumDisplayDigits(int minimumDisplayDigits)`

Establece el número mínimo de dígitos para la numeración de líneas.

`paintComponent(Graphics g)`

Método para dibujar los números de línea en el panel. Itera a través de las líneas visibles y pinta los números correspondientes.

`isCurrentLine(int rowStartOffset)`

Comprueba si la línea dada es la línea actual, en función de la posición del cursor.

`getTextLineNumber(int rowStartOffset)`

Obtiene el número de línea para una posición de inicio de línea dada.

`getOffsetX(int availableWidth, int stringWidth)`

Calcula el desplazamiento X necesario para alinear correctamente el número de línea.

`getOffsetY(int rowStartOffset, FontMetrics fontMetrics)`

Calcula el desplazamiento Y necesario para alinear correctamente el número de línea verticalmente.

`caretUpdate(CaretEvent e)`

Maneja los eventos de cambio de posición del cursor, actualizando el resaltado del número de línea actual.

`changedUpdate(DocumentEvent e)`

Manejan eventos de cambio en el documento de texto, actualizando los números de línea en respuesta a las modificaciones del contenido.

`insertUpdate(DocumentEvent e)`

Manejan eventos de cambio en el documento de texto, actualizando los números de línea en respuesta a las modificaciones del contenido.

`removeUpdate(DocumentEvent e)`

Manejan eventos de cambio en el documento de texto, actualizando los números de línea en respuesta a las modificaciones del contenido.

`documentChanged()`

Realiza cambios en la numeración de líneas en respuesta a los cambios en el documento.

`propertyChange(PropertyChangeEvent evt)`

Maneja los cambios en las propiedades del componente, como la fuente, para actualizar la apariencia de los números de línea.

Clase RenderizarTabla

Se encarga de renderizar los botones que aparecen en la última columna de la tabla de reportes para que el usuario pueda hacer uso de ellas y ver otra ventana con el gráfico generado por graphviz.

getTableRendererComponent(JTable table, Object objetoRenderizable, boolean, isSelect, boolean, hasFocus, int row, int column)

Este método de la clase DefaultTableCellRenderer sirve para que los botones en la última columna de la tabla tengan un tamaño adecuado.

Clase VentanaPrincipal

Esta clase hereda de JFrame se encarga de mostrar todo el funcionamiento de la aplicación, esta contiene un panel en el cual se mostrarán todos los paneles existentes mencionados en este documento, cuenta con un barra botones que permite navegar entre las diferentes opciones de la aplicación.

abrirBotonActionPerformed(ActionEvent evt)

Este método se encarga de abrir un File chooser el cual le permitirá al usuario navegar entre los archivos de su computadora y cargar el archivo de texto en el que el usuario desea ejecutar un analisis léxico, este método hace uso de la clase LeerArchivoTexto, al recibir el texto leido se muestra en el area de editor de texto.

editorTextMouserClicked(MouseEvent evt)

Este método se encarga de cambiar el panel por el panel que contiene el panel que contiene el editor de texto.

reportesBotonMouseClicked(MouseEvent evt)

Se encarga de cambiar la vista actual y mostrar el panel que contiene el tabla de reportes.

ayudaMouseClicked(MouseEvent evt)

Abre un mensaje el cual muestra tips para que el usuario pueda saber que hacer.

acercaDeBotonMouseClicled(MouseEvent evt)

Se encarga de mostrar un mensaje emergente que muestra datos del desarrollador.

pintarPanel(Component panel)

Este método se encarga de recibir un componente como parámetro, quitar todo lo que en el panel contenedor tiene y posteriormente agregar el nuevo componente.

frontend/graphviz

En este paquete se encuentra la funcionalidad que permite graficar tokens, las clases son las siguientes:

Clase GraphvizUtil

Esta clase es la encargada de generar un archivo que contendrá el gráfico.

generarGrafico(String dotCode, String outFileName)

Este método genera una imagen la cual contiene un token separado por nodos, este lo guarda y posteriormete es utilizado cuando se reliza una acción desde la tabla de reportes.

Clase VisualizarGrafico

Esta clase se encarga únicamente de mostrar al usuario la gráfica del token, la clase hereda de JFrame y contiene un Jlabel que es la que muestra la imagen generada por graphviz.

frontend/reporteslexico

Clase ReportesLexicoPanel

Esta clase hereda de JPanel e implementa la interfaz MouseListener, se encarga de mostrár únicamente una tabla en la cual se muestran los resultados de los tokens encontrados en el analisis, además en la última columna muestra un botón el cual al presionarlo se abre una ventana emergente que muestra un gráfico.

initComponents()

Este método se encarga de iniciar el comportamiento de los componentes que existen en este Panel.

actualizarTabla()

Este método se encarga de crear un Modelo de tabla que luego se asignará a la tabla exitencia en este panel. Seguidamente se recorre un arrayList y se cargan los datos en cada casilla correspondiente en la tabla que es mostrada al usuario.

mouseClicked(MouseEvent me)

Este método es sobreescrito y se encarga de obtener el evento al presionar alguno de los botones de la última columna para que abra la ventana con el gráfico que se desea.

frontend/reportessintactico

Clase ReportesSintacticoPanel

Esta clase hereda de JPanel e implementa la interfaz MouseListener, se encarga de mostrar únicamente una tabla en la cual se muestran los resultados de los tokens encontrados en el análisis, además en la última columna muestra un botón el cual al presionarlo se abre una ventana emergente que muestra un gráfico.

initComponents()

Este método se encarga de iniciar el comportamiento de los componentes que existen en este Panel.

actualizarTabla()

Este método se encarga de crear un Modelo de tabla que luego se asignará a la tabla existente en este panel. Seguidamente se recorre un arrayList y se cargan los datos en cada casilla correspondiente en la tabla que es mostrada al usuario.

mouseClicked(MouseEvent me)

Este método es sobreescrito y se encarga de obtener el evento al presionar alguno de los botones de la última columna para que abra la ventana con el gráfico que se desea.

Gramáticas

Las gramáticas para el funcionamiento del analizador sintáctico sirve para definir las reglas del lenguaje, permite definir la estructura del lenguaje python en este caso.

Gramática para letra, dígitos, números, cadenas:

letra ::= [a-z]

| [A-Z]

digito ::= [0-9]

<numero> ::= digito+

<cadena> ::= "" caracter* ""

Gramática para declaración de variables:

<declaracion> ::= <identificador> <operador_asignacion> <expresion>

<identificador> ::= letra [letra | digito]*

<operador_asignacion> ::= "="

<expresion> ::= <cadena>
| <numero>
| <lista>
| <diccionario>

<lista> ::= "[" <elemento_lista>* "]"

<elemento_lista> ::= <expresion> ","

<diccionario> ::= "{" <elemento_diccionario>* "}"

<elemento_diccionario> ::= "" <identificador> "':' <expresion> ","

caracter ::= cualquiera que no sea ""

Gramática para condicionales if if-else if-elif-else

<sentencia> ::= <declaracion>
| <condicional>

<declaracion> ::= <identificador> <operador_asignacion> <expresion>

<condicional> ::= "if" <expresion> ":" <bloque_codigo> <bloque_condicional>*

<bloque_codigo> ::= INDENT <sentencia>* DEDENT

<bloque_condicional> ::= "elif" <expresion> ":" <bloque_codigo>
| "else" ":" <bloque_codigo>

<expresion> ::= <booleano>
| <comparacion>

<booleano> ::= "True"
| "False"

<comparacion> ::= <expresion_aritmetica> <operador_comparacion> <expresion_aritmetica>

<expresion_aritmetica> ::= <numero>
| <identificador>
| "(" <expresion_aritmetica> ")"
| <expresion_aritmetica> <operador_aritmetico> <expresion_aritmetica>

<identificador> ::= letra [letra | digito]*

<operador_asignacion> ::= "="

<operador_aritmetico> ::= "+"
| "-"
| "*"

| "**"

| "/"

| "//"

<operador_comparacion> ::= "=="

| "!="

| "<"

| ">"

| "<="

| ">="

Ciclos:

Ciclo For:

<declaracion> ::= "for" <identificador> "in" <expresion> ":" <bloque>

<bloque> ::= <sentencia>

| <sentencia> <bloque>

<expresion> ::= <rango>

| <llamada_funcion>

<rango> ::= "range" "(" <numero> "," <numero> ")"

<llamada_funcion> ::= <identificador> "(" <argumentos> ")"

<argumentos> ::= <expresion>

| <expresion> "," <argumentos>

<sentencia> ::= <asignacion>

| <llamada_funcion>

| <estructura_condicional>

<asignacion> ::= <identificador> "=" <expresion>

<estructura_condicional> ::= "if" <expresion> ":" <bloque> "else" ":" <bloque>

<identificador> ::= letra [letra | digito]*

Ciclo While

<declaracion> ::= "while" <expresion> ":" <bloque>

<bloque> ::= <sentencia>
| <sentencia> <bloque>

<expresion> ::= <comparacion>

<comparacion> ::= <expresion> <operador_comparacion> <expresion>

<operador_comparacion> ::= "<"
| "<="
| "=="
| "!="
| ">="
| ">"

<sentencia> ::= <asignacion>
| <llamada_funcion>
| <estructura_condicional>

<asignacion> ::= <identificador> "=" <expresion>

<llamada_funcion> ::= <identificador> "(" <argumentos> ")"

<argumentos> ::= <expresion>
| <expresion> "," <argumentos>

<estructura_condicional> ::= "if" <expresion> ":" <bloque> "else" ":" <bloque>

<identificador> ::= letra [letra | digito]*

Funciones / Métodos

<declaracion> ::= "def" <identificador> "(" <lista_parametros> ")" ":" <bloque>

<lista_parametros> ::= <parametro>
| <parametro> "," <lista_parametros>

<parametro> ::= <identificador>

<bloque> ::= <sentencia>
| <sentencia> <bloque>

<sentencia> ::= <asignacion>
| <llamada_funcion>
| <estructura_condicional>
| <retorno>

<asignacion> ::= <identificador> "=" <expresion>

<llamada_funcion> ::= <identificador> "(" <argumentos> ")"

<argumentos> ::= <expresion>

| <expresion> "," <argumentos>

<estructura_condicional> ::= "if" <expresion> ":" <bloque> "else" ":" <bloque>

<retorno> ::= "return" <expresion>

<identificador> ::= letra [letra | digito]*

Gramática para Asignaciones

<asignacion> ::= <identificador> <operador_asignacion> <expresion>

| <lista_identificadores> <operador_asignacion> <lista_expresiones>

<lista_identificadores> ::= <identificador> ("," <identificador>)*

<lista_expresiones> ::= <expresion> ("," <expresion>)*

<identificador> ::= letra [letra | digito]*

<operador_asignacion> ::= "="

| "+="

| "-="

| "*="

| "/="

| "%="

<expresion> ::= <cadena>

| <numero>

<cadena> ::= "" caracter* ""

<numero> ::= digito+

caracter ::= cualquiera que no sea ""

Gramática para condicionales if if-else if-elif-else

<sentencia> ::= <declaracion>
| <condicional>

<declaracion> ::= <identificador> <operador_asignacion> <expresion>

<condicional> ::= "if" <expresion> ":" <bloque_codigo> <bloque_condicional>*

<bloque_codigo> ::= INDENT <sentencia>* DEDENT

<bloque_condicional> ::= "elif" <expresion> ":" <bloque_codigo>
| "else" ":" <bloque_codigo>

<expresion> ::= <booleano>
| <comparacion>

<booleano> ::= "True"
| "False"

<comparacion> ::= <expresion_aritmetica> <operador_comparacion> <expresion_aritmetica>

<expresion_aritmetica> ::= <numero>
| <identificador>
| "(" <expresion_aritmetica> ")"
| <expresion_aritmetica> <operador_aritmetico> <expresion_aritmetica>

<identificador> ::= letra [letra | digito]*

<operador_asignacion> ::= "="

<operador_aritmetico> ::= "+"
| "-"
| "*"
| "**"
| "/"
| "//"

<operador_comparacion> ::= "=="
| "!="
| "<"
| ">"
| "<="
| ">="

Ciclos:

Ciclo For:

<declaracion> ::= "for" <identificador> "in" <expresion> ":" <bloque>

<bloque> ::= <sentencia>
| <sentencia> <bloque>

<expresion> ::= <rango>
| <llamada_funcion>

<rango> ::= "range" "(" <numero> "," <numero> ")"

<llamada_funcion> ::= <identificador> "(" <argumentos> ")"

<argumentos> ::= <expresion>
| <expresion> "," <argumentos>

<sentencia> ::= <asignacion>
| <llamada_funcion>
| <estructura_condicional>

<asignacion> ::= <identificador> "=" <expresion>

<estructura_condicional> ::= "if" <expresion> ":" <bloque> "else" ":" <bloque>

<identificador> ::= letra [letra | digito]*

Ciclo While

<declaracion> ::= "while" <expresion> ":" <bloque>
| <parametro> "," <lista_parametros>

<parametro> ::= <identificador>

<bloque> ::= <sentencia>
| <sentencia> <bloque>

<sentencia> ::= <asignacion>
| <llamada_funcion>
| <estructura_condicional>
| <retorno>

<asignacion> ::= <identificador> "=" <expresion>

<llamada_funcion> ::= <identificador> "(" <argumentos> ")"

<argumentos> ::= <expresion>
| <expresion> "," <argumentos>

<estructura_condicional> ::= "if" <expresion> ":" <bloque> "else" ":" <bloque>

<retorno> ::= "return" <expresion>

<identificador> ::= letra [letra | digito]*

<bloque> ::= <sentencia>
| <sentencia> <bloque>

<expresion> ::= <comparacion>

<comparacion> ::= <expresion> <operador_comparacion> <expresion>

<operador_comparacion> ::= "<"
| "<="
| "=="
| "!="
| ">="
| ">"

<sentencia> ::= <asignacion>
| <llamada_funcion>
| <estructura_condicional>

<asignacion> ::= <identificador> "=" <expresion>

<llamada_funcion> ::= <identificador> "(" <argumentos> ")"

<argumentos> ::= <expresion>
| <expresion> "," <argumentos>

<estructura_condicional> ::= "if" <expresion> ":" <bloque> "else" ":" <bloque>

<identificador> ::= letra [letra | digito]*

Funciones / Métodos

<declaracion> ::= "def" <identificador> "(" <lista_parametros> ")" ":" <bloque>

<lista_parametros> ::= <parametro>
| <parametro> "," <lista_parametros>

<parametro> ::= <identificador>

<bloque> ::= <sentencia>
| <sentencia> <bloque>

<sentencia> ::= <asignacion>
| <llamada_funcion>
| <estructura_condicional>
| <retorno>

<asignacion> ::= <identificador> "=" <expresion>

<llamada_funcion> ::= <identificador> "(" <argumentos> ")"

<argumentos> ::= <expresion>

| <expresion> "," <argumentos>

<estructura_condicional> ::= "if" <expresion> ":" <bloque> "else" ":" <bloque>

<retorno> ::= "return" <expresion>

<identificador> ::= letra [letra | digito]*

No Terminales

Gramática para letra, dígitos, números, cadenas:

- letra
- digito
- numero
- cadena
- carácter
-

Gramática para declaración de variables:

- declaracion
- identificador
- operador_asignacion
- expresion
- lista
- elemento_lista
- diccionario
- elemento_diccionario

Gramática para condicionales if if-else if-elif-else:

- sentencia
- declaracion
- condicional
- bloque_codigo
- bloque_condicional
- expresion
- booleano
- comparacion
- expresion_aritmetica
- identificador
- operador_asignacion
- operador_aritmetico
- operador_comparacion

Ciclos:

- Ciclo For:
    - declaracion
    - bloque
    - rango
    - llamada_funcion
    - argumentos
    - asignacion
    - estructura_condicional
    - identificador
- Ciclo While:
    - declaracion
    - parametro
    - bloque
    - comparacion

- operador_comparacion
- asignacion
- llamada_funcion
- retorno
- identificador

Funciones / Métodos:

- declaracion
- lista_parametros
- parametro
- bloque
- asignacion
- llamada_funcion
- argumentos
- estructura_condicional
- retorno
- identificador

Terminales

Gramática para letra, dígitos, números, cadenas:

- [a-z]
- [A-Z]
- [0-9]
- "
- cualquiera que no sea ""

Gramática para declaración de variables:

- =
- [

-]
- {
- }
- :

Gramática para condicionales if if-else if-elif-else:

- if
- elif
- else
- True
- False

Ciclos:

- Ciclo For:
    - for
    - in
- Ciclo While:
    - while

Funciones / Métodos:

- def
- (
-)
- ,
- return

Gramática para Asignaciones:

- =
- +=
- -=

- *=
- /=
- %=

Jerarquía de Operaciones:

1. **Exponente** (`**`)
2. **Negación** (unario `-` para números negativos)
3. **Multiplicación, División, División entera** (`*`, `/`, `//`, `%`)
4. **Suma y Resta** (`+`, `-`)
5. **Comparaciones** (`==`, `!=`, `<`, `>`, `<=`, `>=`)
6. **Operadores lógicos** (`and`, `or`, `not`)

Estas son las reglas de precedencia estándar en Parser-Py. Si alguna operación se encuentra entre paréntesis `()`, se evaluará primero antes de las operaciones fuera de los paréntesis.