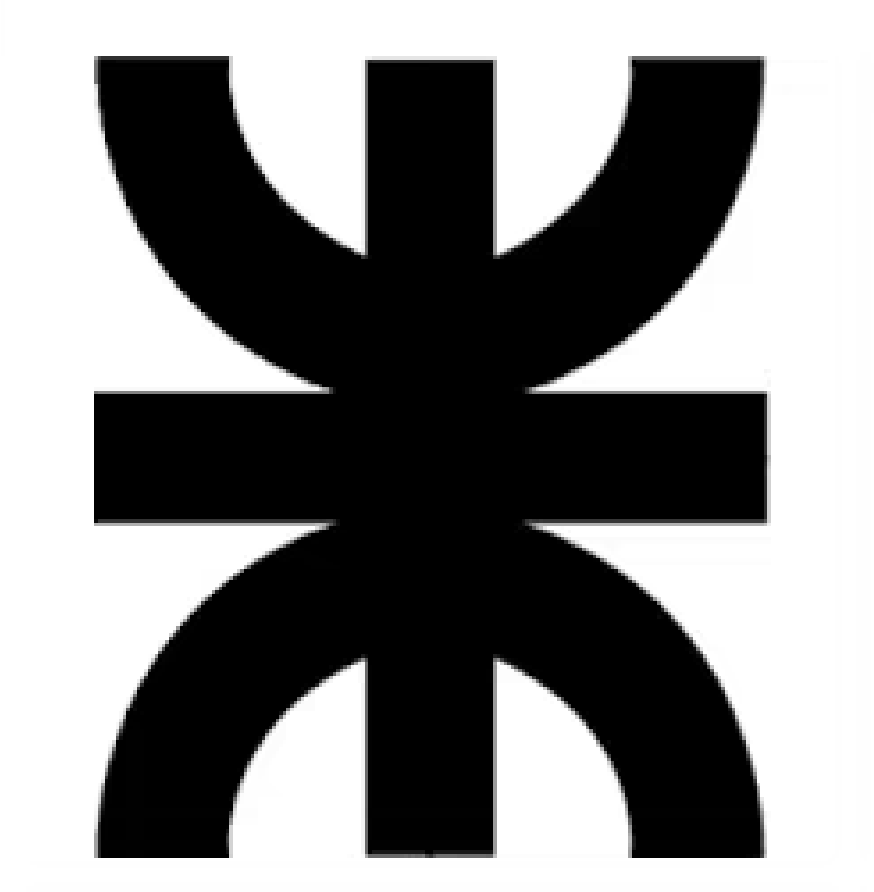
SINTAXIS Y SEMANTICA DE LOS LENGUAJES

TRABAJO PRACTICO INTEGRADOR



Grupo: 2

Comisión: 2.1

Tutora: Ing. Juliana Torre

Integrantes:

- CABRAL CASTELLA, Agustín Robertino
- JEREZ, Fabricio Ezequiel
- MARTINEZ, Agostina Denise
- TOURN, Miguel Agustín

- CICLO LECTIVO 2023 -

INDICE

INTRODUCCIÓN

- Objetivo:

En el presente Trabajo Práctico Integrador se pretende desarrollar las siguientes competencias de la asignatura, las cuales son: Capacidad para reconocer los elementos propios de la Sintaxis y Semántica de los Lenguajes de Programación, pudiendo así comprender los conceptos y procedimientos léxicos/sintácticos asociados, para así poder aplicarlos en la creación de las gramáticas necesarias para detallar los lenguajes que luego se utilizaran en las correspondientes máquinas.

- Descripción:

En esta última entrega del trabajo integrador de la materia Sintaxis y Semántica de los Lenguajes, se desarrollará el analizador léxico sintáctico (analizador lexicográfico) o PARSER de nuestro interprete. Este se alimenta del LEXER generado en la segunda entrega.

Lo que hace nuestro interprete o software, es recibir un texto, ya sea ingresado de manera interactiva por el usuario o cargado mediante un archivo .xml, una vez ya con el texto, se reconocerán todos los tags, cadenas de texto y errores (caracteres no válidos y tags mal escritos o inexistentes), esto será realizado por el LEXER, el PARSER se encargará de analizar tags que aparecen en lugares no debidos, tags faltantes, elementos con atributos incompatibles, tags no cerrados.

Lo desarrollamos en el lenguaje de programación Python con la librería PLY.

- Contenido:

El documento estará divido en las siguientes partes: Primero detallaremos la definición de Docbook, declarando cuales son los tokens o componentes léxicos que acepta (palabras reservadas como terminales) y como deben ser la estructura de estas, también se describirán todas las herramientas utilizadas para poder realizar el software. Luego describiremos las producciones de la gramática que utilizará el compilador, detallando los símbolos que utilizará y el significado de cada uno de ellos. El tipo de gramática que utilizamos es libre de contexto.

- Requerimientos de software

El programa realizado no requiere de un gran software, lo único que se debe disponer es de 15 MB, no será necesario tener ningún IDLE o COMPILADOR, lo único que se tendrá que tener en la computadora es el/los archivo/s con extensión .xml para probar el programa.

- **Herramientas**

XML (Extensible Markup Language): El XML se preocupa por estructurar la información que se pretende almacenar, es decir, define un conjunto de reglas para la codificación de documentos. Este lenguaje de marcado es un conjunto de códigos que se pueden aplicar en el análisis de datos o la lectura de textos creados por computadoras o personas. Se utiliza generalmente para definir elementos, crear un formato y generar un lenguaje personalizado. El diseño XML se centra en la simplicidad, la generalidad y la facilidad de uso y, por lo tanto, se utiliza para varios servicios web, entre otras funcionalidades.

DocBook: es un lenguaje de marcado XML (Extensible Markup Language) diseñado para la creación de documentación técnica y manuales de usuario. Proporciona una estructura estandarizada para la creación de documentos electrónicos, lo que permite que los documentos se puedan utilizar en diferentes formatos de salida, como HTML, PDF, EPUB, entre otros. Es especialmente útil para la creación de documentación técnica, ya que permite separar el contenido del formato y la presentación, lo que facilita la reutilización del contenido en diferentes contextos y dispositivos. Además, DocBook también proporciona elementos específicos para la documentación técnica, como, por ejemplo, secciones para describir las características de un producto, procedimientos de instalación, uso y mantenimiento, entre otros.

Python: de forma resumida podemos decir que, es un lenguaje de alto nivel de programación (facilidad de utilizar para el programador) interpretado, cuya filosofía hace hincapié en la legibilidad de su código, ya que es lo más parecido al lenguaje humano y se puede aprender de manera sencilla y rápida.

Se utiliza para desarrollar aplicaciones de todo tipo y se trata de un lenguaje de programación multiparadigma (resultado de integrar dos o más paradigmas en un mismo sistema). Es un lenguaje interpretado (no se traduce realmente a un formato legible por el ordenador en tiempo de ejecución), dinámico y multiplataforma (todas las herramientas necesarias están disponibles en todas las plataformas principales).

Librería RE: Este módulo proporciona operaciones de coincidencia de expresiones regulares similares a las encontradas en Perl. Esta librería fue solamente utilizada para generar las expresiones regulares que necesitábamos para los tags de DocBook, para las cadenas de texto y para las URL.

Librería PLY: es una implementación en Python de lex y yacc, herramientas populares para la construcción de compiladores.

lex.py divide el texto de entrada en una colección de tokens especificada por una colección de reglas de un lenguaje regular.

yacc.py es usado para reconocer la sintaxis del lenguaje que fue especificada en forma de una gramática libre de contexto.

Pyinstaller: lo que hace es generar un .EXE en Windows, un .DMG en MAC o el ejecutable que utilice el sistema operativo. Dentro del ejecutable se incluye el propio intérprete de Python, y por esa razón podremos utilizarlo en cualquier ordenador sin necesidad de instalar Python previamente. Esta herramienta fue implementada debido a que como se sabe *Python es un lenguaje interpretado*, el cual no genera un archivo .exe, y **la idea del ejecutable es que la persona que descargue el Lexer, pueda ejecutarlo sin tener descargado el intérprete de Python en su computadora**.

Time: usamos esta librería para poder manejar los tiempos de muestra de mensajes por pantalla, es algo más que nada estético que decidimos poner para que todos los textos mostrados al usuario no salgan de seguido, sino que sea un poco más pausado.

Anaconda: es una distribución libre y abierta de los lenguajes Python y R, utilizada en ciencia de datos, y aprendizaje automático. Esto incluye procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos. Esta herramienta fue usada por el grupo debido a que, a ninguno nos reconocía la librería ply mencionada anteriormente, esta era instalada mediante el comando **pip install ply** en la cmd (símbolo de sistema) de cada uno de los integrantes del grupo, era instalado correctamente, pero al cargar la librería en el VS CODE no la reconocía. Gracias a Anaconda pudimos reconocer a la librería ply, ya que esta distribución tiene un VS CODE ejecutable, lo único que tuvimos que hacer fue, descargar Anaconda, instalar la librería ply desde la cmd de Anaconda, abrimos el VS CODE que se encuentra dentro de Anaconda Navigator y desde ahí pudimos empezar a escribir el Lexer.

Analizador Léxico o LEXER

Definición de Token, Lexema y Patrón (con ejemplos)

Creemos conveniente para una mejor comprensión del tema, dejar una breve definición con nuestras palabras de:

Token Léxico o Token: Es una unidad indivisible con significado único dentro del lenguaje, por lo cual, es la salida del analizador léxico.

Ejemplos de Tokens:

Categoría	Ejemplos
Delimitadores	( ) , ; : [ ]
Palabras reservadas	while true do if for
Identificadores	Index f isEven
Números enteros	3 -4 55 0 7658
Números flotantes	4.5 .3 0.5 8.4e-5
Simbolos especiales	+ -* / . = < > <= >= != ++
Cadenas	"Hola mundo!"

Patrón: Es una expresión regular que permite identificar un tipo de token y referenciar al conjunto de todos los tokens que se ajustan a él.

Lexema: Es una cadena de caracteres que encaja con un patrón que describe un componente léxico, es decir, es como una instancia de un patrón.

Ejemplo de patrón y lexema:

Tipo de token	Patrón léxico	Ejemplos de lexema	Cadena propia
While	while	while	SI
Enteros	digito+	12 123 0 45	NO
División	/	/	SI
Identificador	letra (letra |digito)*	Index f isEven	NO

El analizador léxico o LEXER es la primera fase un de compilador (programa que traduce código escrito en un lenguaje de programación a otro lenguaje) y lo que hace es recibir como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de tokens (componentes léxicos) o símbolos.

Es decir, es un programa capaz de descomponer una entrada de caracteres (generalmente contenidos en un fichero) en una secuencia ordenada de tokens.

Características del AL

Por lo cual, las acciones que realiza el LEXER son:

- Procesar el lenguaje fuente como una secuencia de caracteres, es decir, convertir la secuencia de caracteres en una colección de componentes léxicos con significado único dentro del lenguaje llamados tokens.
- Agrupar la secuencia de caracteres en palabras con significado propio y después lo transforma en una secuencia de terminales.
- Buscar los componentes léxicos o palabras que componen el programa fuente, según reglas o patrones.

Procedimiento del AL

1. Se aplica un método para reconocer los posibles patrones, es decir que, en este caso las expresiones regulares son útiles para describir formalmente los patrones de los tokens.

2. Se reconocen los tokens, es decir que, los autómatas finitos reconocen los lenguajes por medio de dichas expresiones regulares.

3. Se realizan acciones preestablecidas al reconocer el token correspondiente.

Errores detectables por el AL

Algunos ejemplos de errores que detecta nuestro analizador léxico son:

o Caracteres ilegales en el programa fuente, es decir, caracteres inválidos, números malformados, entre otros.

o Errores de ortografía en palabras reservadas (tags mal escritos o tags inexistentes)

Modo de Obtención del Interprete

Luego de elegir el lenguaje de programación a utilizar (Python), recurrimos a varios videos tutoriales para así poder obtener una librería llamada PLY, la cual nos permitió realizar el analizador léxico. Después de eso, modificamos el formato y añadimos los tokens específicos de DocBook para que sean reconocidos por nuestro Lexer.

- **Construcción del LEXER**

Para la construcción del Lexer, de manera muy resumida, lo que hicimos fue primero crear una lista con las palabras reservadas que vamos a utilizar, que en su mayoría son las apertura y clausuras de los tags, luego generamos las expresiones regulares de los tags de DocBook, esto fue muy sencillo debido a que la expresión regular era tal cual la apertura o clausura del tag a trabajar.

Ejemplo de expresión regular un tag:

```
def t_CLOSE_ARTICLE (t):
    r'</article>'
    return t
```

Luego de esto, nos encargamos de hacer las expresiones regulares para las cadenas de texto, la cual acepta cualquier carácter menos los siguientes: “<” y “>” que son los que definen el cierre o clausura de un tag. También acepta la letra “ñ” y las letras con acento, esto se logró poniendo Python en la configuración **UTF-8,** este “tip” fue dado por el profesor Tortosa, ya que en la entrega del Lexer no pudimos hacer que reconozca esos caracteres mencionados, y al no saber cómo hacer, consultamos con el y nos dio su ayuda.

Ejemplo de expresión regular para texto:

```
def t_TEXTO(t) :
      r'[^<>]+'
      if t.value.upper() in reservadas:
            t.value = t.value.upper()
            t.type = t.value

      return print (f'Cadena de texto: {t.value}')
```

Siguiente a esto, se generó la expresión regular para aceptar URL, la cual fue la que más nos costo debido a que la conformación de una URL es muy compleja y se nos complicó entenderla y luego pasar a expresión regular.

Ejemplo de expresión regular para URL:

```
def t_URL(t):
    r'\"(http|https|ftp|ftps)://[^/\s:]+(:\d+)?(/[^#\s])?(\#\S)?\"'
    return t
```

Por final, fue construir el LEXER, y darle la opción al usuario de que cargue su texto de manera interactiva.

```
LEXER = lex.lex()  #armamos el lexer
print ('¡Bienvenido al Lexer!')
```

Analizador sintáctico o PARSER

Para una mejor comprensión del tema, empezaremos definiendo que es un parser.

Un analizador sintáctico o parser es un programa que normalmente es parte de un compilador. El compilador se asegura de que el código se traduce correctamente a un lenguaje ejecutable. La tarea del analizador es, en este caso, la descomposición y transformación de las entradas en un formato utilizable para su posterior procesamiento. Se analiza una cadena de instrucciones en un lenguaje de programación y luego se descompone en sus componentes individuales.

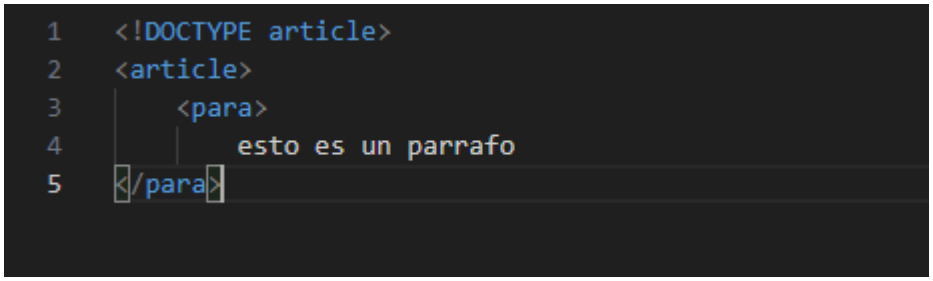
Cómo funciona un AS

Para analizar un texto, por ejemplo, los analizadores suelen utilizar un analizador léxico separado (llamado lexer), que descompone los datos de entrada en tokens (símbolos de entrada como palabras), el lexer va pasando token a token al parser y este se va fijando si la correspondencia de los mismo corresponde a la gramática definida, por eso se dice que el lexer es el “hijo” del parser.

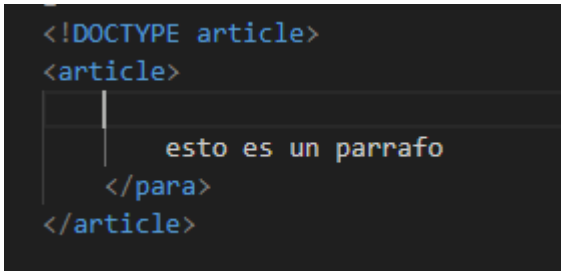
Errores detectables por el AS

Los errores que serán detectados por nuestro parser son aquellos relacionados a la sintaxis del texto a analizar, como ser:

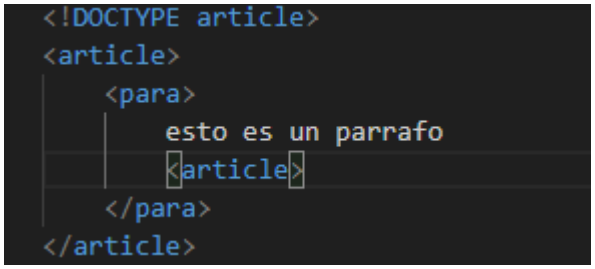
- Etiquetas sin cerrar (en el ejemplo no está cerrado </article>)



- Etiquetas sin abrir que luego se cierran (se cierra el para pero nunca se abrió)



- Etiquetas en una posición incorrecta (un artículo no puede estar dentro de un para)



Construcción del PARSER

Luego de realizar y corregir el lexer, ya que en la segunda entrega tuvo algunos errores mínimos, proseguimos con el diseño del parser utilizando la misma librería (PLY), en este caso usamos yacc lo cual nos permite hacer el analizador sintáctico. De esta forma, adaptamos y definimos las reglas de gramática en DOCBOOK. Cabe aclarar que la gramática que realizamos en la primera entrega fue cambiada casi en su mayoría debido a que había varios errores los cuales no nos permitieron realizar la codificación del parser.

A continuación, pondremos un ejemplo de la gramática que realizamos tal cual nos enseñan en la materia y como esta se “traduce” al ply.

∑ -> INICIO

INICIO -> <!DOCTYPE article> GEN_ARTICLE

GEN_ARTICLE -> <article> GEN_ALLART </article>

```
def p_inicio (p):
    """inicio : DOCTYPE gen_article"""


def p_gen_article(p):
    '''gen_article : OPEN_ARTICLE gen_allart CLOSE_ARTICLE'''
```

La regla sigma en inicio no esta definida en el parser debido a que es generada automáticamente por el PLY.

Para explicar como se realiza una producción usando PLY, realizamos lo siguiente:

Usamos una función de Python (def), luego ponemos el prefijo “p” que significa producción según la documentación de PLY, y luego escribimos el nombre del símbolo NO TERMINAL.

La producción se la escribe dentro de comillas que deben ser tres de apertura y tres de cierre, básicamente es un comentario en bloque en Python.

def p_gen_article (p):

’’’ gen_article : OPEN_ARTICLE gen_allart CLOSE_ARTICLE’’’

Esto sería equivalente a:

GEN_ARTICLE -> <article> GEN_ALLART </article>

IMPORTANTE: debemos poner NOTERMINAL (espacio):(espacio), sino nos toma como un error de sintaxis

Cabe destacar que para el PLY los terminales son en MAYUSCULAS, estos no terminales son tokens definidos previamente en el lexer, aquí el ejemplo:

```
def t_OPEN_ARTICLE  (t) :
    r'<article>'
    return t

def t_CLOSE_ARTICLE (t):
    r'</article>'
    return t
```

Los no terminales son en MINUSCULAS para el PLY, ósea se ocupa una convención contraria a lo que usamos en la materia.

Formas de carga de archivos a analizar:

En la consigna se pedía que el usuario pueda cargar su documento de manera interactiva o mediante la carga de un archivo con extensión .xml, para esto pusimos que elija entre dos opciones

```
print ('Tipo de carga:')
print ('A) Interactiva')
print ('B) Carga por archivo')
print ('Si desea usar los ejemplos del grupo, ingrese opcion B y ponga la direccion de la carpeta')
time.sleep (1)
rta = (input ('Ingrese la opcion\n>'))
time.sleep (1)
```

A) INTERACTIVA

```
if rta.upper() == "A":
        print ('Escriba su texto, recuerde que si pulsa el doble ENTER se da como finalizado el texto.')
        time.sleep (1)
        info =""
        while True:
            linea = input("")
            if linea == "":
                break
            info += linea + "\n"
```

Básicamente el usuario va escribiendo línea a línea el texto que quiere a analizar, una vez que no quiere hacerlo más, presiona doble enter y todo el texto se guarda en la variable “info”.

B) CARGA POR ARCHIVO

```
elif rta.upper ()=="B":
    print ("Por favor, ingrese por teclado la dirección de la carpeta en donde se encuentran los archivos.xml")
    directorio = input("Ingresa la dirección de archivo: ")

    # Obtener todos los archivos en el directorio
    archivos = [archivo for archivo in os.listdir(directorio) if archivo.endswith(".xml")]


    print("Archivos en la ubicación:")
    for i, archivo in enumerate(archivos, start=1):
        print(f"{i}. {archivo}")


    num_archivo = int(input("Ingresa el número del archivo que deseas seleccionar: "))

    # Verificar si el número es válido y guardar el archivo seleccionado en la variable info
    if 1 <= num_archivo <= len(archivos):
        nombre_archivo = archivos[num_archivo - 1]
        ruta_archivo = os.path.join(directorio, nombre_archivo)
        with open(ruta_archivo, "r") as archivo:
            info = archivo.read()
            print("Archivo seleccionado:", nombre_archivo)
    else:
        print("El número de archivo no es válido.")
```

Primero se le pide la dirección de donde se encuentra su archivo (path), y se obtienen todos los archivos de ese directorio que solo contengan la extensión .xml (debido a que la consigna pide que solo sean archivos con esta extension).

Se imprimen por pantalla todos los archivos que cumplan con la condición y se les asigna un número.

```
Archivos en la ubicación:
1. 3.xml
2. 1.xml
3. Texto1.xml
4. texto5.xml
5. X.xml
6. x2.xml
7. xd.xml
Ingresa el número del archivo que deseas seleccionar:
```

Se le pide al usuario que ingrese un numero para analizar el archivo, si el usuario ingresa un numero que no corresponde, se le informa que no es válido. Si el número es válido, se inicia la ejecución del interprete.

Modo de ejecución del interprete

Para la ejecución del mismo, se realiza lo siguiente

Construimos el lexer mediante lex.lex () y lo guardamos en “LEXER”, luego construimos el parser con la acción yacc.yacc () y lo guardamos en “parser”.

Y guardamos en la variable “result”, el análisis sintáctico del texto “info” utilizando el parser y lexer mencionados, los cuales fueron “construidos” anteriormente.

En el caso que result no sea “None” significara que nuestro texto es sintácticamente correcto.

```
LEXER = lex.lex()  #armamos el lexer
parser = yacc.yacc()

result = parser.parse(info, lexer = LEXER)

if result is not None :
    print ('Felicidades, su código es sintacticamente correcto')
```

Para ejecutar nuestro parser, se debe ejecutar a través del archivo ejecutable PARSER.exe.

Este nos dará la opción de una carga interactiva del documento que deseamos analizar o se puede elegir una carga mediante un archivo de extensión .xml, para la última opción mencionada se debe pegar la ruta de la carpeta de donde se encuentran los archivos que queremos analizar y nos dejará elegir entre todos los que tengan esa extensión y se podrá seleccionar uno para analizarlo. Además, como adicional, genera un archivo html, el cual tendrá el mismo nombre del archivo .xml de entrada, donde se traduce los tags DOCBOOK a HTML.

GRAMATICA

Σ -> XR //Llamado XML primero y luego a DocBook (obligatoriamente).

X-> <?xml version="1.0" encoding="UTF-8"?> //Token (terminal) XML.

INICIO -> <!DOCTYPE article > GEN_ARTICLE

GEN_ARTICLE -> <article> GEN_ALLART </article>

GEN_ALLART -> GEN_INFO GEN_TITL GEN_DATA GEN_SECCIONES | GEN_INFO GEN_DATA GEN_SECCIONES | GEN_TITL GEN_DATA GEN_SECCIONES | GEN_DATA GEN_SECCIONES | GEN_TITL GEN_DATA | GEN_DATA | GEN_INFO GEN_DATA |

GEN_DATA -> GEN_LIST | GEN_IMPORTANT | GEN_PARA| GEN_SIMP | ADR| MO| GEN_TABLE |GEN_COM| GEN_ABS | GEN_LIST GEN_DATA| GEN_IMPORTANT GEN_DATA | GEN_PARA GEN_DATA| GEN_SIMP GEN_DATA | ADR GEN_DATA | mo GEN_DATA | GEN_TABLE GEN_DATA | GEN_COM GEN_DATA | GEN_ABS GEN_DATA |

GEN_SECCIONES -> GEN_SECTION GEN_SECCIONES | GEN_SIMPSEC GEN_SECCIONES | GEN_SECTION | GEN_SIMPSEC |

GEN_SECTION -> <section> GEN_ALLART </section>

GEN_SIMPSEC -> <simplesect> SIMPSEC </simplesect>

SIMPSEC -> GEN_INFO GEN_TITL GEN_DATA | GEN_INFO GEN_DATA | GEN_DATA GEN_SECCIONES | GEN_TITL GEN_DATA | GEN_DATA |

GEN_INFO -> <info> INFO </info>

INFO -> MO | GEN_ABS | ADR | GEN_AUT | GEN_DATE | GEN_COPY | GEN_TITL | MO INFO | GEN_ABS INFO | ADR INFO | GEN_AUT INFO | GEN_DATE INFO | GEN_COPY INFO | GEN_TITL INFO

GEN_ABS -> <abstract> GEN_TITL INFOAB </abstract> | <abstract> INFOAB </abstract>

INFOAB -> GEN_PARA | GEN_SIMP | GEN_PARA INFOAB | GEN_SIMP INFOAB

ADR -> <address> GENAD </address> | <address> A </address>

GENAD -> GEN_STREET | GEN_STATE | GEN_CITY | GEN_PHONE | GEN_EMAIL | A | GEN_STREET GENAD| GEN_STATE GENAD | GEN_CITY GENAD | GEN_PHONE GENAD | GEN_EMAIL GENAD | A GENAD |

GEN_AUT -> <author> AUT </author>

AUT -> GEN_FIRSTN | GEN_SURN | GEN_FIRSTN AUT | GEN_SURN AUT

GEN_COPY <copyright> COPY </copyright>

COPY -> GEN_YEAR | GEN_YEAR GEN_HOLDER

GEN_FIRSTN -> <firstname> GENALL </firstname>

GEN_SURN -> <surname> GENALL </surname>

GEN_STREET -> <street> GENALL <street>

GEN_CITY -> <city> GENALL </city>

GEN_STATE -> <state> GENALL </state>

GEN_PHONE -> <phone> GENALL </phone>

GEN_EMAIL -> <email> GENALL </email>

GEN_DATE -> <date> GENALL </date>

GEN_YEAR-> <year> GENALL </year>

GEN_HOLDER -> <holder> GENALL </holder>

GENALL -> A| GEN_LINK | GEN_EMP | GEN_COM | A GENALL | GEN_LINK GENALL | GEN_EMP GENALL | GEN_COM GENALL

GEN_TITL -> <title> CONT_TIT </title>

CONT_TIT -> A | GEN_EMP | GEN_LINK | GEN_EMAIL | A CONT_TIT | GEN_EMP CONT_TIT | GEN_EMAIL CONT_TIT | GEN_LINK CONT_TIT

GEN_SIMP -> <simpara> CONEX </simpara>

GEN_LINK -> <link xlink:href="URL"> CONEX </link>

GEN_COM -> <comment> CONEX </comment>

CONEX -> A| GEN_EMP | GEN_LINK | GEN_EMAIL | GEN_AUT | GEN_COM | GEN_EMP CONEX | A CONEX | GEN_LINK CONEX | GEN_EMAIL CONEX | GEN_AUT CONEX | GEN_COM CONEX

GEN_PARA -> <para> OP_PARA </para>

OP_PARA -> A | GEN_LINK | GEN_EMP | GEN_EMAIL | GEN_AUT | GEN_COM | GEN_LIST | GEN_IMPORTANT | ADR | MO | GEN_TABLE | A OP_PARA | GEN_LINK OP_PARA | GEN_EMP OP_PARA | GEN_EMAIL OP_PARA | GEN_AUT OP_PARA | GEN_COM OP_PARA | GEN_LIST OP_PARA | GEN_IMPORTANT OP_PARA | ADR OP_PARA | MO OP_PARA | GEN_TABLE OP_PARA

GEN_IMPORTANT -> <important> GEN_TITL IMPORTANT </important>

IMPORTANT -> GEN_LIST | GEN_IMPORTANT | GEN_PARA | GEN_SIMP | ADR | MO | GEN_TABLE | GEN_COM | GEN_ABS | GEN_LIST IMPORTANT| GEN_IMPORTANT IMPORTANT| GEN_PARA IMPORTANT| GEN_SIMP IMPORTANT| ADR IMPORTANT | MO IMPORTANT | GEN_TABLE IMPORTANT | GEN_COM IMPORTANT| GEN_ABS IMPORTANT

GEN_LIST -> <itemizedlist mark='A'> ITEM </itemizedlist>

ITEM -> <listitem> NEWLIST </listitem>

NEWLIST -> GEN_IMPORTANT | GEN_PARA | GEN_SIMP | GEN_LIST | GEN_TABLE | MO | GEN_COM | GEN_ABS | GEN_IMPORTANT NEWLIST | GEN_PARA NEWLIST | GEN_SIMP NEWLIST | GEN_LIST NEWLIST | GEN_TABLE NEWLIST | MO NEWLIST | GEN_COM NEWLIST | GEN_ABS NEWLIST

MO -> <mediaobject> J </mediaobject>

J -> <info> INFO </info> VIDEO R | <info> INFO </info> IMAGE R |R

R -> IMAGE| VIDEO | IMAGE R | VIDEO R |

VIDEO -> <videoobject> INFO <videodata fileref= VIDEO_DATA /> </videodata> </videoobject > | <videoobject> <videodata fileref= VIDEO_DATA /> </videodata> </videoobject>

IMAGE -> <imageobject> INFO <imagedata fileref= IMAGE_DATA /> </imageobject> | <imageobject> <imagedata fileref= IMAGE_DATA /> </imageobject>

GEN_LISTITEM -> <itemizedlist mark='A'> CONEX3 </itemizedlist>

CONEX3 -> FIN | CONEX3 FIN

FIN -> GEN_IMPORTANT | GEN_PARA | GEN_SIMP | ADR | GEN_LISTITEM | MO | GEN_TABLE | GEN_COM | GEN_ABS | GEN_IMPORTANT CONEX3 | GEN_PARA CONEX3 | GEN_SIMP CONEX3 | ADR CONEX3 | GEN_LISTITEM CONEX3 | MO CONEX3 | GEN_TABLE CONEX3 | GEN_COM CONEX3 | GEN_ABS CONEX3 |

GEN_TABLE -> <table> INF_TB </table>

INF_TB ->MO|T_GROUP | MO INF_TB | T_GROUP INF_TB

T_GROUP -> <tgroup> GEN_TGROUP </tgroup>

GEN_TGROUP -> T_HEAD T_BODY T_FOOT | T_BODY T_FOOT | T_HEAD T_BODY | T_BODY

T_HEAD -> <thead> ROW </thead>

T_BODY -> <tbody> ROW </tbody>

T_FOOT -> <tfoot> ROW </tfoot>

ROW -> <row> ENTRY </row> ROW | <row> ENTRY </row> | <row> ENTRY_TB </row> | <row> ENTRY_TB </row> ROW

ENTRY_TB -> <table> T_HEAD T_BODY </table>

ENTRY -> <entry>GEN_ENTRY </entry>

GEN_ENTRY -> A| GEN_ENTRY | GEN_LISTITEM| GEN_IMPORTANT | GEN_PARA | GEN_SIMP | MO | GEN_COM | GEN_ABS | A| GEN_LISTITEM GEN_ENTRY | GEN_IMPORTANT GEN_ENTRY | GEN_PARA GEN_ENTRY | GEN_SIMP GEN_ENTRY | MO GEN_ENTRY | GEN_COM GEN_ENTRY | GEN_ABS GEN_ENTRY

ETIQUETAS

URL-> PRO SUB DOM EXT P //del URL pasa al protocolo (parte de su estructura)

PRO-> http:// | https:// | ftps:// | ftp:// //protocolo

http:// (para recursos de la web)

https:// (para recursos de la web contenidos en un servidor seguro)

ftp:// (recursos contenidos en un servidor de ficheros)

ftps:// (recursos contenidos en un servidor de ficheros seguro)

https://login.live.com/login.srf?wa=wsignin1.0&rpsnv=12&ct=1436147091&rver=6.4.6456.0&wp=MBI_SSL_SHARED&wreply=https:%2F%2Fmail.live.com%2Fdefault.aspx%3Fmkt%3Des-us%26rru%3Dinbox&lc=2058&id=64855&mkt=es-us&cbcxt=mai

P-> N | /DIR | N P | / | //puerto (opcional) "/" hace que no se ponga nada del agregado de url por ej : http://www.miweb.com/ <- la ultima "/" termina la url si es que no se necesita agregado de puerto

SUB-> A. | www. | ww1 | blog. //subdominio

DOM-> A/ | miweb //dominio

EXT-> A | .com | .ar | .org //extensión

DIR-> A | A DIR/ | LOC //ruta o directorio (opcional)

LOC-> #A | #ALOC //localizador interno (opcional) // yo sacaria LOC pq si A abarca todas las letras, ya lo estaria haciendo, siendo que es opcional. Pero si queda mas "estetico" y ordenado para la ruta de la URL le dejamos.

#texto

A-> aA | bA | cA | dA | eA | fA | gA | hA| iA | jA | qA | lA | mA | nA | ñA | oA | pA | qA | rA | sA | tA | uA | vA | wA | xA | yA | zA //tokens

A-> _A | /A | 'A | -A | +A | %A | $A | &A | (A |)A | !A | ¡A | ,A | @A | .A | áA | éA | íA | óA | úA | üA //tokens

A-> 0A | 1A | 2A | 3A | 4A | 5A | 6A | 7A | 8A | 9A //tokens

A-> a | b | c | d | e | f | g | h | i | j | k | m | n | ñ | o | p | q | r | s | t | u | v | w | x | y | z

A-> _ | / | ' | - | + | % | $ | & | (|) | ! | ¡ | , | @ | . | á | é | í | ó | ú | ü

A-> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Símbolos no terminales

GEN_ALLART : usamos este no terminal para generar todas las cosas posibles que pueden estar dentro de un GEN_ALLART

GEN_ARTICLE : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_ARTICLE

GEN_DATA : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_DATA

GEN_SECCIONES : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_SECCIONES

GEN_SECTION : usamos este no terminal para generar las posibilidades que pueden estar dentro de EN_SECTION

GEN_SIMPSEC : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_SIMPSEC

SIMPSEC : usamos este no terminal para generar las posibilidades que pueden estar dentro de SIMPSEC

GEN_INFO : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_INFO

INFO : usamos este no terminal para generar las posibilidades que pueden estar dentro de INFO

GEN_ABS : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_ABS

INFOAB : usamos este no terminal para generar las posibilidades que pueden estar dentro de INFOAB

ADR : usamos este no terminal para generar las posibilidades que pueden estar dentro de ADR

GENAD : usamos este no terminal para generar las posibilidades que pueden estar dentro de GENAD

GEN_AUT : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_AUT

AUT : usamos este no terminal para generar las posibilidades que pueden estar dentro de AUT

GEN_COPY : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_COPY

COPY : usamos este no terminal para generar las posibilidades que pueden estar dentro de COPY

GEN_FIRSTN : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_FIRSTN

GEN_SURN : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_SURN

GEN_STREET : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_STREET

GEN_CITY : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_CITY

GEN_STATE : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_STATE

GEN_PHONE : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_PHONE

GEN_EMAIL : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_EMAIL

GEN_DATE : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_DATE

GEN_YEAR : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_YEAR

GEN_HOLDER : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_HOLDER

GENALL : usamos este no terminal para generar las posibilidades que pueden estar dentro de GENALL

GEN_TITL : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_TITL

CONT_TIT : usamos este no terminal para generar las posibilidades que pueden estar dentro de CONT_TIT

GEN_SIMP : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_SIMP

GEN_EMP : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_EMP

GEN_LINK : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_LINK

GEN_COM : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_COM

CONEX : usamos este no terminal para generar las posibilidades que pueden estar dentro de CONEX

GEN_PARA : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_PARA

OP_PARA : usamos este no terminal para generar las posibilidades que pueden estar dentro de OP_PARA

GEN_IMPORTANT : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_IMPORTANT

IMPORTANT : usamos este no terminal para generar las posibilidades que pueden estar dentro de IMPORTANT

GEN_LIST : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_LIST

ITEM : usamos este no terminal para generar las posibilidades que pueden estar dentro de ITEM

NEWLIST : usamos este no terminal para generar las posibilidades que pueden estar dentro de NEWLIST

MO : usamos este no terminal para generar las posibilidades que pueden estar dentro de MO

J : usamos este no terminal para generar las posibilidades que pueden estar dentro de J

R : usamos este no terminal para generar las posibilidades que pueden estar dentro de R

VIDEO : usamos este no terminal para generar las posibilidades que pueden estar dentro de VIDEO

IMAGE : usamos este no terminal para generar las posibilidades que pueden estar dentro de IMAE

GEN_LISTITEM : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_LISTITEM

CONEX3 : usamos este no terminal para generar las posibilidades que pueden estar dentro de CONEX3

FIN : usamos este no terminal para generar las posibilidades que pueden estar dentro de FIN

GEN_TABLE : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_TABLE

INF_TB : usamos este no terminal para generar las posibilidades que pueden estar dentro de INF_TB

T_GROUP : usamos este no terminal para generar las posibilidades que pueden estar dentro de T_GROUP

GEN_TGROUP : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_TGROUP

T_HEAD : usamos este no terminal para generar las posibilidades que pueden estar dentro de T_HEAD

T_BODY : usamos este no terminal para generar las posibilidades que pueden estar dentro de T_BODY

T_FOOT : usamos este no terminal para generar las posibilidades que pueden estar dentro de T_FOOT

ROW : usamos este no terminal para generar las posibilidades que pueden estar dentro de ROW

ENTRY_TB : usamos este no terminal para generar las posibilidades que pueden estar dentro de ENTRY_TB

ENTRY : usamos este no terminal para generar las posibilidades que pueden estar dentro de ENTRY

GEN_ENTRY : usamos este no terminal para generar las posibilidades que pueden estar dentro de GEN_ENTRY

CONCLUSION

En resumen, el desarrollo de un lexer utilizando la librería PLY ha sido un desafío que nos permitió adquirir una comprensión profunda del análisis léxico en Python. En el cual, los primeros inconvenientes se nos presentaron en el entender conceptos propios del área como *tokens*, *lexemas*, *patrones*, *expresiones regulares,* etc. O la identificación de los patrones correctos y garantizar que todas las situaciones sean manejadas adecuadamente.

Luego una de las etiquetas que más problemas nos causó fue la de las url debido a su estructura específica y caracteres permitidos. También todo lo relacionado a imagen y video. O la detección de errores como caracteres ilegales o errores de ortografía en palabras reservadas.

Este proyecto y sobre todo esta última entrega nos permitió comprender con mayor detalle cómo funcionan los compiladores. Así como técnicas, conocimientos y una base sólida para futuros desarrollos en el campo de la compilación.

BIBLIOGRAFIA

Fuentes:

https://www.dabeaz.com/ply/ply.html

https://docs.hektorprofe.net/python/distribucion/pyinstaller/

https://www.anaconda.com

https://docs.python.org/es/3/library/re.html

https://www.profesionalreview.com/2018/06/30/que-es-cmd/