

*Universidad Tecnológica Nacional
Facultad Regional Resistencia*

Sintaxis y Semántica de los Lenguajes

INGENIERÍA EN SISTEMAS DE INFORMACIÓN

TRABAJO PRACTICO INTEGRADOR

- Primer cuatrimestre
- COMISIÓN: ISI A/B.
- AÑO LECTIVO: 2024
- GRUPO: SpaceX.

ALUMNOS:

- BADER SORAIRO, Cain A.
- BLANCO, Facundo.
- SULCA GOMEZ, Oscar Martin.
- SAN CRISTOBAL, Juan Manuel.

DOCENTES:

- Tomaselli Gabriela.
- Torre Juliana.
- Tortosa Nico
- Vigil Rodrigo

Introducción	2
Conformación del grupo	5
Matriz de habilidades	5
Alianza de Equipo - SpaceX	6
Gramática	7
Símbolos de la gramática	9
Producciones	10
Analizador léxico	14
Ejecución del lexer	15
Control de Errores	15
Analizador Sintáctico	17
Traducción a HTML	18
Conclusiones	20
Gramática	20
Analizador Léxico	21
Analizador Sintactico	21
Traducción a HTML	21
Bibliografía	15

Introducción

El presente documento busca relatar el desarrollo del TPI de la materia Sintaxis y Semántica de los Lenguajes, que se dicta en la Universidad Tecnológica Nacional, Facultad Regional Resistencia.

El trabajo consiste en el análisis, validación y traducción de un documento JSON con un formato particular para el almacenamiento de información sobre empresas. En la primera parte del trabajo, el objetivo fue la creación de la gramática generadora del lenguaje. Definimos todas las producciones necesarias para la obtención del lenguaje, que luego será necesaria para los analizadores léxicos y sintácticos. En la segunda parte construimos el analizador léxico a partir de la gramática. El objetivo de esta parte es leer un string con el formato de un archivo json y detectar cada elemento o token que habíamos definido como elemento terminal en la gramática. En la tercera parte el enfoque fue el más complejo ya que se trabajó utilizando como base de los anteriores trabajos, es decir que se usaron gran parte del trabajo hecho en la primera y segunda parte para poder construir un analizador sintáctico ya que es crucial en la interpretación, su función principal es analizar una secuencia de tokens y determinar su estructura gramatical de acuerdo con las reglas de una gramática formal. Luego de ello se realiza la traducción a HTML el cual es un lenguaje de marcado con el cual se organiza la información interpretada la cual estaba en un archivo formato json.

Para llegar a la solución se trabajó de manera ardua y utilizando mucho la creatividad, al comenzar con la primera parte la cual fue la generación de la gramática se trabajó el diseño con una aplicación web denominada draw.io la cual permitió realizar los diagramas necesario para que el equipo tenga una visión general de cómo a ser nuestra gramática. Una vez generada realizada nuestra gramática el equipo procedió a realizar el analizador léxico, el cual no teníamos una visión muy clara de que expresiones regulares deberíamos escoger para la realización de nuestros token, para ello fue crucial la asistencia a clase y el preguntar en cada oportunidad a los profesores cuál era la mejor manera de como elegir las expresiones regulares para la generación de los token. Una vez que con ayuda de los profesores supimos la longitud y de donde hasta adonde iban a ser nuestros token y nuestras palabras reservadas pasamos a la última etapa. Una vez realizado la etapa del analizador léxico procedimos a escribir toda la gramática en



python para que esto funcione, sin embargo nos encontramos con varios problemas al realizar el compilador ya que el objetivo principal era que se pueda detectar todos los errores e informar, sin embargo después de varios intentos el grupo optó por tomar como ejemplo los compiladores de los lenguajes antiguos como fortran o cobol los cuales al encontrar el primer error el compilador se detenía. Una vez solucionada esta etapa el grupo decidió que para realizar la traducción a html, y para este caso se utilizó la lista de token creadas por el lexer, lo que se hizo fue analizar si el analizador sintáctico funcionaba, en ese caso recorrimos la lista y fuimos traduciendo todo esto al lenguaje HTML y así es una breve introducción de como afrontamos algunos de los desafíos que nos encontramos durante la elaboración del trabajo práctico integrador.

Información y requerimientos de software para ejecutar y compilar

1) Verificación e instalación de python.

```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19045.4529]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Josue Centurion>python --version
Python 3.12.3

C:\Users\Josue Centurion>
```

en el caso de no tener instalado se procede a acceder a la pagina de python para <https://www.python.org/>

2) Descarga de Visual Studio Code y luego lo instalamos

Visita el sitio web oficial de Visual Studio Code en code.visualstudio.com.

Haz clic en el botón de descarga para Windows.

3) Instalamos la biblioteca PLY

```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19045.4529]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Josue Centurion>python --version
Python 3.12.3

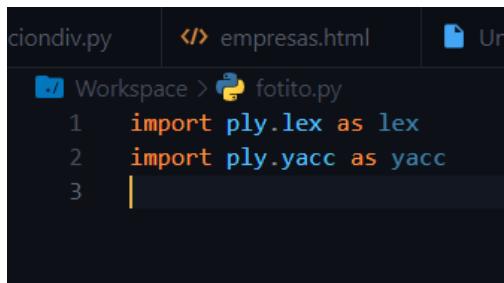
C:\Users\Josue Centurion>pip install ply
Requirement already satisfied: ply in c:\users\josue centurion\appdata\local\programs\python\python312\lib\site-packages
(3.11)

[notice] A new release of pip is available: 24.0 -> 24.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Josue Centurion>
```

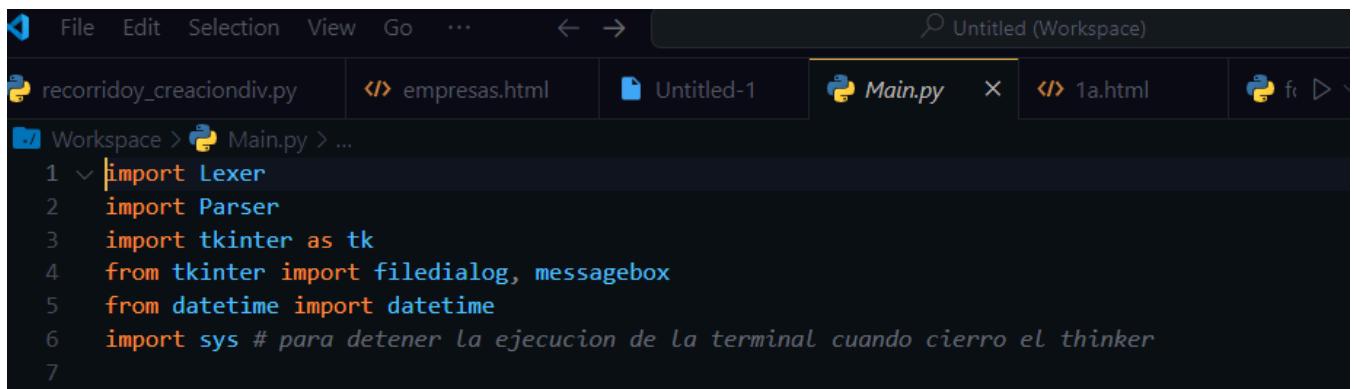


Y luego lo utilizamos



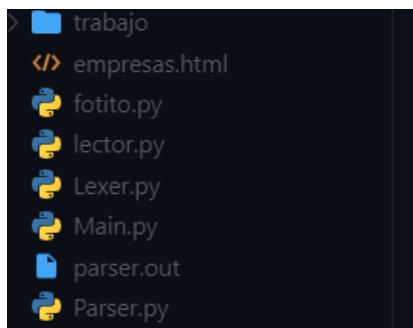
```
ciondiv.py  </> empresas.html  Ur
Workspace > fotito.py
1 import ply.lex as lex
2 import ply.yacc as yacc
3 |
```

Y algunas necesitamos algunas bibliotecas



```
File Edit Selection View Go ... ← → Untitled (Workspace)
recorridoy_creadiondiv.py  </> empresas.html  Untitled-1  Main.py  1a.html  f...
Workspace > Main.py > ...
1 ✓ import Lexer
2 import Parser
3 import tkinter as tk
4 from tkinter import filedialog, messagebox
5 from datetime import datetime
6 import sys # para detener la ejecucion de la terminal cuando cierro el thinker
7
```

4) Establecemos un entorno/directorio de trabajo



```
> trabajo
  </> empresas.html
    fotito.py
    lector.py
    Lexer.py
    Main.py
    parser.out
    Parser.py
```

Luego de esto se va a disponer de todas las cosas necesarios y/o esenciales para poder comenzar y ver el código funcionar.



Conformación del grupo

Matriz de habilidades

A continuación se puede observar una matriz la cual está compuesta por una columna las cuales describen las habilidades y las filas los nombres de los integrantes.

Significados de los símbolos

- ★ (La persona es experta)
- (La persona no es experta pero podría desenvolverse en dicha competencia)
- blank (La persona desconoce por completo)

		Habilidades					
		Programación	Inglés	Investigación	Documentación	Grabación	Edición de fotos y videos
Miembros del equipo	Cain	•	★	•	•	•	
	Martín	•	•	•	•	•	★
	Facundo	★	★	•	★	★	•
	Juan	•	★	★	•		

		Habilidades			
		Resolución de problemas	Trabajo en equipo	Habilidades de comunicación	Aprendizaje continuo y adaptabilidad
Miembros del equipo	Cain	★	★	★	•
	Martín	★	★	•	•
	Facundo	★	★	•	-
	Juan	•	★	★	•



Alianza de Equipo - SpaceX

Como equipo, nuestro objetivo principal es la realización de un trabajo que no solo funcione correctamente, si no que se haga con paciencia y constancia. Buscamos llevar un avance semanal medible que nos permite ir detectando y corrigiendo errores en el camino. A su vez, sabiendo que la segunda y tercera entrega están muy cercanas una con la otra, creemos que es indispensable aprovechar las primeras semanas para avanzar lo más posible.

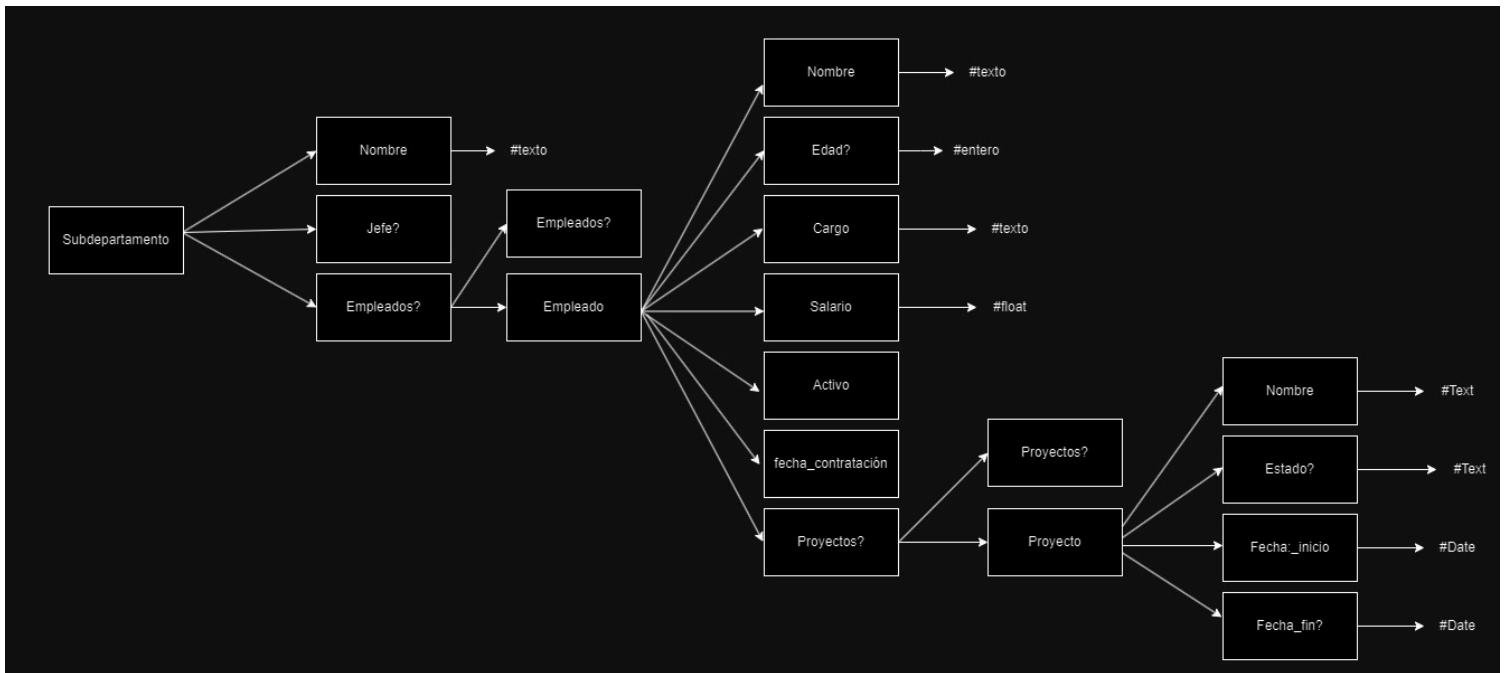
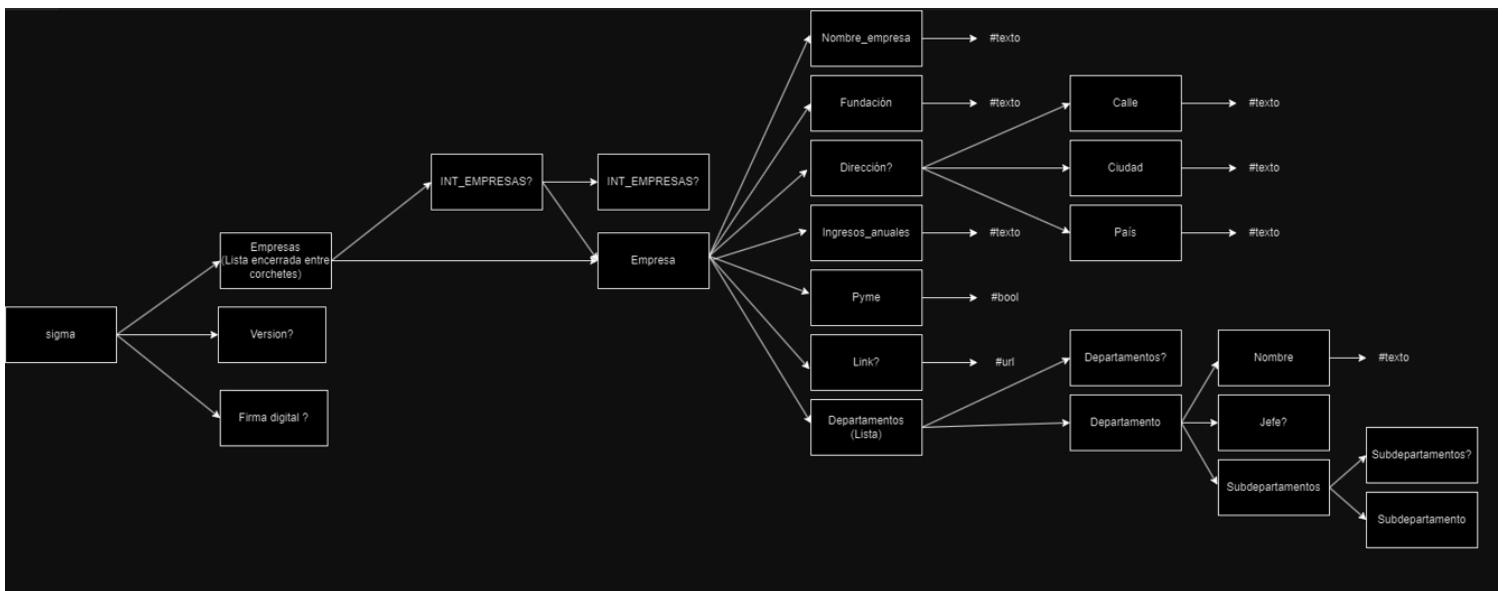
Por lo expuesto anteriormente, nos disponemos a realizar, como mínimo, una videollamada grupal por semana con la participación de los 4 integrantes, para trabajar en conjunto con el tema principal de esa semana. Estas videollamadas se llevarán a cabo, en principio, los días martes por la mañana, a través de Google Meet, con la posibilidad de mover ese día en caso de que sea necesario. Allí, además de avanzar con el TP, definiremos qué cuestiones podemos dividir entre cada integrante para realizar durante la semana de forma individual, y definiremos el tema a trabajar en la próxima reunión.

Por fuera de estas reuniones, los 4 integrantes nos comprometemos a mantener una constante comunicación sobre avances y problemas, y a mantener actualizado el contenido a medida que se vaya generando o editando. Para esto, utilizaremos tres herramientas: WhatsApp, para un contacto mas rápido e informal en caso de dudas puntuales o cuestiones triviales. Google Drive, para mantener todos los archivos pertinentes actualizados y accesibles en todo momento. Trello para llevar constancia tanto de los avances realizados, los pendientes, y las cuestiones a hablar en la próxima reunión o a consultar con el profesor; Y GitHub para realizar actualizaciones de código acerca del Lexer y el Parser, a fin de que todos puedan ver los cambios hechos, en qué momento y para una correcta centralización de la información.



Gramática

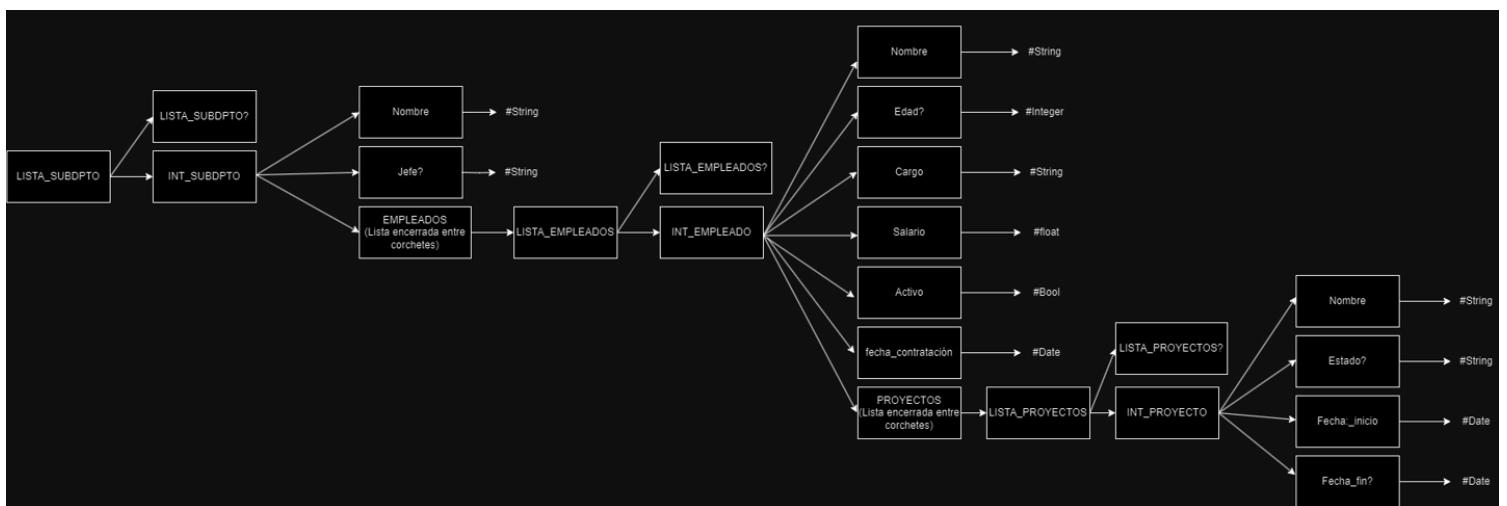
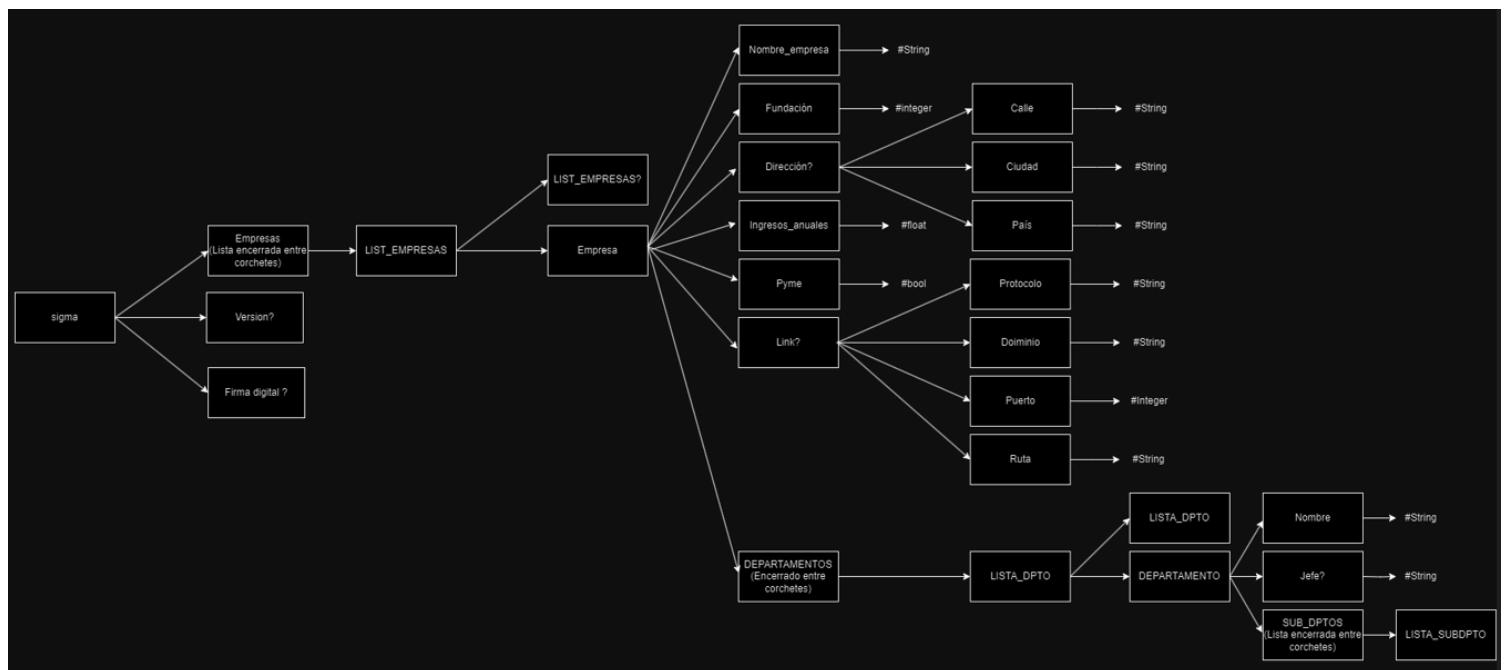
Para comenzar la creación de la gramática, decidimos usar la herramienta draw.io, a partir de la cual se pueden crear diagramas y mapas conceptuales colaborativamente. La idea detrás de esto era comprender las derivaciones y reglas gramaticales de una manera más visual, previo a la definición literal de cada una. El primer diagrama que realizamos fue el siguiente:



donde, para el manejo de las listas, hicimos uso de producciones recursivas. Las mismas nos permiten definir tantos objetos dentro de las listas como sea necesario.



Después de analizar minuciosamente los terminales obligatorios de JSON (corchetes, llaves, comillas, etcétera), detectamos que la forma en la que definimos la recursividad en las listas no era la adecuada porque llevaba a la creación de terminales de más. Por esto, luego de algunas modificaciones llegamos al diagrama final, que soluciona estos inconvenientes de recursión.



La solución implica añadir no terminales auxiliares que hacen de “intermediarios”, y asociando los no terminales del lenguaje a una sola producción no recursiva. Entonces, a partir de este diagrama, construimos todas las producciones, que se muestran a continuación:



Símbolos de la gramática

SIMBOLOS NO TERMINALES	SIMBOLOS TERMINALES
Σ	“empresas”: ” version”: ” firma digital”:
DIRECCION VERSION FIRMA DIGITAL	“nombre_empresa”: ”fundacion”:
LIST_EMPRESA EMPRESA	“ingresos_anuales”: ”pyme”: ”ink”:
NOMBRE_EMPRESA FUNDACION	“direccion”:
INGRESOS_ANUALES PYME	“calle”: “ciudad”: “pais”:
LINK	“departamentos”: ”subdepartamentos”:
DIRECCION INT_DIRECCION	“nombre”: ” jefe?”: “empleados”:
CALLE CIUDAD PAIS	“nombre”: ”edad”: ”cargo”:
DEPARTAMENTO LISTA_DPTO	”salario”: ”activo”: ”fecha_contratacion”:
JEFE SUB_DPTO LISTA_SUB SUB_DPTOS	“proyectos”: ”estado”:
NOMBRE JEFE EMPLEADOS	“fecha_inicio”:
LISTA_EMPLEADOS	“fecha_fin”:
INT_EMPLEADOS	integer date string float bool null
EDAD CARGO SALARIO	http:// https:// 80 link_string
ACTIVO FEC_CONT	
PROYECTOS LISTA_PROYECTOS	
INT_PROYECTO ESTADO	
FEC_INICIO FEC_FIN	



PROTOCOLO PUERTO RUTA	DOMINIO	
----------------------------	---------	--

Producciones

$\Sigma \rightarrow \{ \langle \text{EMPRESAS} \rangle, \langle \text{VERSION} \rangle, \langle \text{FIRMA_DIGITAL} \rangle \mid$
 $\{ \langle \text{EMPRESAS} \rangle, \langle \text{FIRMA_DIGITAL} \rangle, \langle \text{VERSION} \rangle \mid$
 $\{ \langle \text{VERSION} \rangle, \langle \text{FIRMA_DIGITAL} \rangle, \langle \text{EMPRESAS} \rangle \mid$
 $\{ \langle \text{FIRMA_DIGITAL} \rangle, \langle \text{VERSION} \rangle, \langle \text{EMPRESAS} \rangle \mid$
 $\{ \langle \text{FIRMA_DIGITAL} \rangle, \langle \text{EMPRESAS} \rangle, \langle \text{VERSION} \rangle \mid$
 $\{ \langle \text{EMPRESAS} \rangle, \langle \text{VERSION} \rangle \mid$
 $\{ \langle \text{EMPRESAS} \rangle, \langle \text{FIRMA_DIGITAL} \rangle \mid$
 $\{ \langle \text{VERSION} \rangle, \langle \text{EMPRESAS} \rangle \mid$
 $\{ \langle \text{FIRMA_DIGITAL} \rangle, \langle \text{EMPRESAS} \rangle \mid$
 $\{ \langle \text{EMPRESAS} \rangle \}$

$\langle \text{VERSION} \rangle \rightarrow \text{"version": string} \mid \text{"version": null}$
 $\langle \text{FIRMA_DIGITAL} \rangle \rightarrow \text{"firma_digital": string} \mid \text{"firma_digital": null}$

$\langle \text{EMPRESAS} \rangle \rightarrow \text{"empresas": [} \langle \text{LIST_EMPRESAS} \rangle \text{]}$

$\langle \text{LIST_EMPRESAS} \rangle \rightarrow \{ \langle \text{EMPRESA} \rangle \mid$
 $\{ \langle \text{EMPRESA} \rangle \}, \langle \text{LIST_EMPRESAS} \rangle$

$\langle \text{EMPRESA} \rangle \rightarrow \langle \text{NOMBRE_EMPRESA} \rangle, \langle \text{FUNDACIÓN} \rangle,$
 $\langle \text{DIRECCION} \rangle,$
 $\langle \text{INGRESOS_ANUALES} \rangle, \langle \text{PYME} \rangle, \langle \text{LINK} \rangle,$
 $\langle \text{DEPARTAMENTOS} \rangle \mid$
 $\langle \text{NOMBRE_EMPRESA} \rangle, \langle \text{FUNDACION} \rangle,$
 $\langle \text{INGRESOS_ANUALES} \rangle, \langle \text{PYME} \rangle, \langle \text{LINK} \rangle,$
 $\langle \text{DEPARTAMENTOS} \rangle \mid$
 $\langle \text{NOMBRE_EMPRESA} \rangle, \langle \text{FUNDACIÓN} \rangle, \langle \text{DIRECCION} \rangle,$
 $\langle \text{INGRESOS_ANUALES} \rangle, \langle \text{PYME} \rangle, \langle \text{DEPARTAMENTOS} \rangle$
 \mid
 $\langle \text{NOMBRE_EMPRESA} \rangle, \langle \text{FUNDACION} \rangle,$
 $\langle \text{INGRESOS_ANUALES} \rangle, \langle \text{PYME} \rangle, \langle \text{DEPARTAMENTOS} \rangle$



```

<NOMBRE_EMPRESA> → "nombre_empresa": string
<FUNDACION> → "fundacion": integer
<INGRESOS_ANUALES> → "ingresos_anuales": float
<PYME> → "pyme": bool

<LINK> → "link": <PROTOCOLO><DOMINIO>:<PUERTO><RUTA> |
           "link": null
<PROTOCOLO> → http:// | https://
<DOMINIO> → link_string
<PUERTO> → 80 | integer
<RUTA> → link_string

<DIRECCION> → "direccion": { <INT_DIRECCION> } | "direccion": {}
| "direccion": null

<INT_DIRECCION> → <CALLE> , <CIUDAD> , <PAIS> |
                     <CALLE> , <PAIS> , <CIUDAD> |
                     <CIUDAD> , <PAIS> , <CALLE> |
                     <CIUDAD> , <CALLE> , <PAIS> |
                     <PAIS> , <CIUDAD> , <CALLE> |
                     <PAIS> , <CALLE> , <CIUDAD>

<CALLE> → "calle": string
<CIUDAD> → "ciudad": string
<PAIS> → "pais": string

<DEPARTAMENTOS> → "departamentos": [ <LISTA_DPTO> ]

<LISTA_DPTO> → { <DEPARTAMENTO> }
<LISTA_DPTO> → { <DEPARTAMENTO> } , <LISTA_DPTO>

<DEPARTAMENTO> → <NOMBRE> , <JEFE> , <SUB_DPTOS> |

```



```

<NOMBRE> , <SUB_DPTOS> , <JEFE> |
<JEFE> , <SUB_DPTOS> , <NOMBRE> |
<JEFE> , <NOMBRE> , <SUB_DPTOS> |
<SUB_DPTOS> , <NOMBRE> , <JEFE> |
<SUB_DPTOS> , <JEFE> , <NOMBRE> |
<NOMBRE> , <SUB_DPTOS> |
<SUB_DPTOS> , <NOMBRE>

```

<SUB_DPTOS> → “subdepartamentos”: [<LISTA_SUB>]

<LISTA_SUB> → { <SUB_DPTO> }

<LISTA_SUB> → { <SUB_DPTO> } , <LISTA_SUB>

<SUB_DPTO> → <NOMBRE> , <JEFE> , <EMPLEADOS> |
<NOMBRE> , <JEFE> |
<NOMBRE> , <EMPLEADOS> |
<NOMBRE> |
<JEFE> , <NOMBRE> , <EMPLEADOS> |
<JEFE> , <EMPLEADOS> , <NOMBRE> |
<EMPLEADOS> , <JEFE> , <NOMBRE> |
<EMPLEADOS> , <NOMBRE> , <JEFE> |
<JEFE> , <NOMBRE> |
<EMPLEADOS> , <NOMBRE>

<JEFE> → “jefe”: string | “jefe”: null

<EMPLEADOS> → “empleados”: [LISTA_EMPLEADOS] | “empleados”: [] |
“empleados”: null

<LISTA_EMPLEADOS> → { <INT_EMPLEADO> } |
{ <INT_EMPLEADO> } , <LISTA_EMPLEADOS>

<INT_EMPLEADO> → <NOMBRE> , <EDAD> , <CARGO> , <SALARIO>,
<ACTIVO> , <FEC_CONT> , <PROYECTOS> |
<NOMBRE> , <EDAD> , <CARGO> , <SALARIO> ,
<ACTIVO> , <FEC_CONT> |
<NOMBRE> , <CARGO> , <SALARIO> , <ACTIVO> ,
<FEC_CONT> , <PROYECTOS> |
<NOMBRE> , <CARGO> , <SALARIO> , ACTIVO ,



<p><FEC_CONT></p> <p><NOMBRE> → “nombre”: string</p> <p><EDAD> → “edad”: integer “edad”: null</p> <p><CARGO> → “cargo”: string</p> <p><SALARIO> → “salario”: float</p> <p><ACTIVO> → “activo”: bool</p> <p><FEC_CONT> → “fecha_contratacion”: date</p> <p>PROYECTOS → “proyectos”: [LISTA_PROYECTOS] “proyectos”: null</p> <p><LISTA_PROYECTOS> → { <INT_PROYECTO> } { <INT_PROYECTO> } , <LISTA_PROYECTOS></p> <p><INT_PROYECTO> → <NOMBRE> , <ESTADO> , <FEC_INICIO> , <FEC_FIN> <NOMBRE> , <FEC_INICIO> , <FEC_FIN> <NOMBRE> , <ESTADO> , <FEC_INICIO> <NOMBRE> , <FEC_INICIO></p> <p><ESTADO> → “estado”: string “estado”: null</p> <p><FEC_INICIO> → “fecha_inicio”: date</p> <p><FEC_FIN> → “fecha_fin”: date “fecha_fin”: null</p>



Analizador léxico

Un **analizador léxico** (lexer) es la primera fase de un compilador, consistente en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de *tokens* (componentes léxicos) o símbolos. Estos *tokens* sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico (parser). La especificación de un lenguaje de programación a menudo incluye un conjunto de reglas que definen el léxico. Estas reglas consisten comúnmente en expresiones regulares que indican el conjunto de posibles secuencias de caracteres que definen un token. Un **token** es una cadena con un significado asignado y por lo tanto identificado. Está estructurado como un par que consta de un «nombre de token» y un «valor de token».

El lexer fue construido en Python (**versión 3.12**), utilizando la librería OS (**No hace falta descargar ya viene en las librerías de python**) y la librería PLY (**Python Lex-Yacc, versión 3.9**). Esta librería otorga múltiples funcionalidades para la construcción de compiladores, es la más famosa y popular entre la variedad que teníamos, tal como, LEXER o ANTLR (aunque no está escrito en python tiene soporte para escribir un analizador léxico o sintáctico en python). Su funcionalidad es evaluar un código de formato **.json** identificando los símbolos terminales (lista de tokens) que definimos anteriormente en la gramática. Ésta lista la dividimos en dos, una corresponde a las palabras "reservadas" de nuestro lenguaje (departamentos, dirección, pyme, etc), y otra a los tipos de datos que cada palabra reservada va a tener, por ejemplo:

- Un token puede estar compuesto por uno o más tokens.
- El token de dirección está compuesto por *calle*, *ciudad* y *país*. Estos 3 últimos tienen tipos de datos asignados que son tipo string (que en nuestra gramática está definido como texto).

Establecimos la expresión regular para cada uno de los tokens. Para el caso específico fecha y texto, se definió una función que permite hacer un análisis más específico.

- Fecha: Además de comprobar que el formato sea el correcto (yyyy/mm/dd), se verifica que tanto el año, mes y día estén dentro de los rangos aceptables (para años consideramos el rango de 1900 a 2099). No se tuvo en cuenta que febrero puede tener 28 o 29 días como límite.
- Texto: Diferencia entre las palabras reservadas, opciones de strings que tienen que venir únicamente para los tokens: cargo de empleado, estado de proyectos; y cadenas de texto genéricas. Las palabras reservadas están acompañadas de un ':' a su derecha, usamos esto para diferenciar ambos tipos de cadenas.



cargo (empleado) = Tipo de dato string, una de las siguientes opciones:

- Product Analyst | Project Manager | UX designer | Marketing | Developer | Devops | DB admin

estado (proyecto) = Tipo de dato string, una de las siguientes opciones:

- To do | In progress | Canceled | Done | On hold

Ejecución del lexer

Para poder ejecutar el programa, en el zip descargado (SpaceX) abrir la carpeta “dist” ahí encontrara el ejecutable, hacer doble click e ingresar el texto que desea analizar, para finalizar hacer un salto de linea (enter) e ingresar por teclado las teclas “Control + D”, nuevamente hacer un “enter” y el código será analizado y mostrado por pantalla con sus respectivos controles de errores.

Si desea ver el código y hacerlo funcionar dentro del programa deberá contar con un **compilador** de su agrado, el lenguaje **Python** y librería **PLY** (versiones mencionadas anteriormente). Si ya posee todo, en el zip descargado (SpaceX) hacer click derecho en el archivo “Lexer” con extensión “.py” luego en “abrir con” y selecciona el compilador con el cual desea abrir el archivo.

Control de Errores

Sabiendo que hay algunas etiquetas que obligatoriamente deben estar en el archivo JSON de entrada, se buscó implementar una funcionalidad que permita informar al usuario cuando falta alguna etiqueta al final de análisis. Ejemplo.

```
Ingrasa texto. Presiona Control + z (Ctrl + z) para finalizar.
```

```
"empresas":
```

```
^Z
```

```
Finalizando la lectura (Control + z detectado).
```

```
Nº Linea: 1 | Token: EMPRESAS | Valor: "empresas"
```

```
Las etiquetas faltantes obligatorias que no aparecieron son:
```

```
DEPARTAMENTOS, FUNDACION, INGRESOS ANUALES, NOMBRE, NOMBRE EMPRESA, PYME, SUBDEPARTAMENTOS
```

Para implementar esta funcionalidad lo que se hizo fue leer todos los tokens que el usuario estaba utilizando. Una vez leído todos los tokens lo que hicimos fue hacer una lista con el tipo de token y analizar estos token utilizados con una lista de **ETIQUETAS_OBLIGATORIAS** que debían venir de manera obligatoria. Una vez comparado con esta lista de Etiquetas obligatorias, en el caso de que hayan venido todas las etiquetas obligatorias no se muestra nada. Pero en el caso de que no haya venido alguna etiqueta obligatoria. Se mostrará por pantalla que falta esa etiqueta.



- **ETIQUETAS MAL ESCRITAS**

Para analizar esta parte, a partir expresión regular:

```
r'\"([^\n]|(\.\.))*?\\"[\:]*[\:]?'
```

Partimos de que si viene un carácter dos puntos, sabemos que se trata de una palabra reservada, y no un string cualquiera. Entonces, buscamos en la lista de tokens reservados, y si no se encuentra una coincidencia, lo detectamos como un error. Para eso creamos un token error que avisa al usuario mostrando que la etiqueta está mal escrita y que ahora su `t.type` del token va a ser **ERROR**. En este ejemplo detectamos que el usuario escribió “empre”: y al tener el mismo formato que las palabras reservadas entonces identificamos que el usuario quiso escribir una palabra reservada. Analizamos su contenido con la lista de palabras reservadas y como son diferentes advertimos de la etiqueta mal escrita.

```
Nº Linea: 1 | Token: ERROR | Valor: "empre"
```



Analizador Sintáctico

Para realizar el analizador sintáctico el profesor en cada una de las clases nos fue explicando que cada vez que realizamos el Parser este crea un árbol y que existen dos tipos de recorridos del árbol, el ascendente y el descendente y que lo tomemos en cuenta.

Con esto en mente comenzamos a investigar y también el profesor nos aclaro que al utilizar python el tipo de recorrido es Parser-Ascendente

Parser Ascendente (Bottom-Up Parser)

En un parser ascendente, el proceso comienza desde los símbolos terminales de la gramática y se mueve hacia arriba para construir las producciones que derivan en el símbolo inicial. Esto fue necesario saber a la hora de ir haciendo las producciones o en otras las reglas de la gramática ya que al recorrer desde las hojas hasta el nodo inicial, o nodo raíz tuvimos que poner en cierto orden. Sin embargo no fue un impedimento porque en el intérprete que estamos generando no hizo gran diferencia.

Cuando estuvimos realizando como equipo en analizador sintáctico nuestra idea principal era que en nuestro compilador cada vez que detectaba algún error en el orden es decir un error sintáctico, siga leyendo el código y así ver todos los errores y después mostrar por pantalla cada uno de los errores. Sin embargo, esto nos puso muchísimos problemas a la hora de analizar. Optamos por hacer algo más simple y recordamos que a los inicios de la programación los lenguajes tales como cobol y fortran cuando al realizar el proceso de compilación encontraban un solo error dejaban de ejecutar el compilador y directamente informaba el error así el usuario programador podía continuar desde ese punto

Básicamente para la creación del Parser se utilizó como guía la gramática creada en la primer entrega, en base a esto se fue desarrollando las funciones que nos ayudaron a hacer el parser

Algunos problemas que nos encontramos es que estaba invertido los caracteres y apertura lista y de apertura bloque

La función objeto empresa tenía nuevamente apertura lista y cierre lista, pero ya estaba en su no terminal anterior “lista empresas”. Para continuar la dirección tenía una apertura llave y no una apertura bloque y otros problemas menores. Todo esto dificultó muchísimo en la creación del parser ya que nos tiraba muchísimos errores que había que corregir.

nos encontramos que al recorrer toda la lista de tokens al momento de traducir los tokens encontrados en el archivo a HTML, estamos iterando sobre una lista incorrecta que estaba incompleta dado que usábamos tipos_tokens (o algo así) que usaba un set() lo cual elimina todos los tokens duplicados que teníamos, dejando solo uno de cada uno, de esta forma nunca podíamos recorrer todos los tokens encontrados por el



lexer sino solo 1 de cada uno. PAra solucionar esto creamos un diccionario que tiene como valor el tipo de token (token.type) y como llave su valor (token.value), de esta forma en el main.py obtenemos el diccionario creado en el lexer con todos los tokens y sus valores, iteramos sobre este y los tokens de interés para la traducción los traducimos, dejando de lado el resto

Traducción a HTML

Esta fue una de las partes más desafiantes para el equipo ya que lo que hicimos fue trabajar en conjunto con el lexer y el parser. Nuestro equipo decidió que cuando se iba realizando el análisis léxico fuera creando una lista la cual poseía el t.type : t.value, es decir el token type y su token valor, esto con cada uno de los token. Luego de que el Parser nos hubiera analizado la sintaxis del json y nos diera valido, es decir que si al analizar la sintaxis no arrojaba ningún error entonces procedemos a trabajar con la lista generada por el lexer y comenzar a recorrerla, con una función recorremos la lista y cada vez que encontrábamos los t.type token que necesitábamos entonces buscábamos al siguiente t.type que sea de TEXTO y cuando este era de texto agarramos su valor. Este es el caso más trabajado por el equipo sin embargo vimos muchos inconvenientes a la hora de recorrer la lista ya que muchas token como apertura de lista y otros estaban muchas veces repetidas en el archivo, lo cual producía un algoritmo muy laborioso para recorrer y el cual nos tiraba muchos errores.

Un camino intermedio que se tomó fue ir escribiendo desde el parser a cada uno de los token y sus valores necesarios, lo cual nos ayudo muchisimo ya que eliminamos muchas de las cosas que no eran necesarias, muchos token que sólo estaban de relleno y de esta manera fue mucho más fácil recorrer la lista.

Otro gran problema es que al comienzo trabajamos con un diccionario en el cual estamos almacenando todos los token encontrados en el LEXER con su respectivo tipo y valor pero no funcionaba para la traducción a HTML. porque cada token tenia un identificador único, entonces se encontraba uno repetido lo sobrescribirá anterior, por este motivo para solucionar esto se decidió almacenar las cosas en una tupla que tenga el valor del token y su tipo

Funciones auxiliares

```
import tkinter as tk  
from tkinter import filedialog, messagebox
```

```
def Buscar_Archivo():
```



```
# Crear una ventana Tkinter oculta
global texto_ingresado # Hacemos global a esta variable, para poder
ocuparla en el analizador de texto

root = tk.Tk()
root.withdraw()

# Abrir un cuadro de diálogo para seleccionar el archivo
file_path = filedialog.askopenfilename(
    title="Seleccionar archivo",
    filetypes=(("JSON files", ".json"), ("All files", ".*")) # oculta
archivos con extensiones que no sea .json
)
# Eliminamos todo el contenido previo en caso de que quiera analizar
otra vez.
Ingreso_de_Texto.delete(1.0, tk.END)

# Acá copia toda la info del archivo en la pantalla
texto_ingresado = leer_archivo(file_path)
Ingreso_de_Texto.insert(tk.END, texto_ingresado)

messagebox.showinfo("BUSCAR ARCHIVO", "¡Se ha cargado el archivo con
exito!")

def leer_archivo(file_path):
    with open(file_path, 'r') as file:
        return file.read()
```

Modo de ejecución del intérprete

Interactiva: Al realizar la entrega del Lexer se nos pedía que el usuario tenga la posibilidad de poder ingresar por pantalla la información del archivo json. Con los cual hicimos de esa manera, le permitimos al usuario ingresar el archivo json por pantalla y para finalizar tenia que apretar la combinacion de teclas CRTL + Z y asi finalizar el programa, sin embargo la consigna pedía CRTL + D pero esto no era posible en Windows solo en ordenadores que corrian sistema operativo de MACos. Sin embargo este pequeño ajuste fue permitido

A partir de un archivo: Ya para esta última entrega se nos pidió que el usuario pueda tener una interfaz y la cual usamos la biblioteca tkinter para poder crear esa interfaz y que el usuario pueda buscar en su escritorio el archivo json para su correcto funcionamiento.



Conclusiones

Gramática

Al momento de realizar las producciones recursivas para EMPLEADOS, EMPRESAS, DEPARTAMENTOS, etc, nos dimos cuenta que existía una mala recursión, por lo que nos los creaba los objetos como listas dentro de la lista general, lo cual no era lo correcto; Los eliminamos ya que solo los necesitamos para una LISTA_EMPLEADOS por cada no terminal EMPLEADOS creado, nombramos de una forma más distintiva el símbolo EMPLEADO a fin de poder diferenciar mejor al no terminal de empleado.

Problema

<EMPLEADOS> → “empleados”: [EMPLEADOS]
→ <EMPLEADO>

Solución

<EMPLEADOS> → “empleados”: [LISTA_EMPLEADOS]
<LISTA_EMPLEADOS> → { <INT_EMPLEADO> } |
{ <INT_EMPLEADO> } , <LISTA_EMPLEADOS>



Analizador Léxico

Al comenzar con el lexer comprendimos que las producciones gramaticales producidas en la anterior entrega iban a tener cambios, los objetos de las listas de empresas, empleados, departamentos y subdepartamentos ya no iban a ser necesarias, dado que al generar un nuevo objeto de alguna lista simplemente se abría un corchete. El mayor reto que nos presentó el lexer fue, definir la expresión regular “texto” dado que para identificarlo debe estar entre comillas, como todos los demás tokens; La solución que encontramos fue que al matchear con token texto primero verifique que no es una palabra reservada que tienen el carácter “:” al final. Con eso solucionado nuestro próximo problema fue decidir de qué manera tratamos al string URL, dado que si creamos tokens de cada parte de la url (protocolo, dominio, puerto, ruta), a la hora de hacer el parser tendríamos que hacer mayor control en este. Para simplificarlo buscamos la expresión regular en la cual tendría que venir una URL.

Para terminar con el trabajo, buscamos la forma de finalizar la ejecución con las teclas Control + D. Investigando en diferentes documentaciones encontramos que las teclas Control + D son teclas utilizadas en arquitectura Mac mientras que en arquitectura Windows la tecla de escape es Control + Z, la cual fue la que implementamos al contar únicamente con esta arquitectura.

Analizador Sintáctico

Cuando estuvimos realizando como equipo en analizador sintáctico nuestra idea principal era que en nuestro compilador cada vez que detectaba algún error en el orden es decir un error sintáctico, siga leyendo el código y así ver todos los errores y después mostrar por pantalla cada uno de los errores. Sin embargo, esto nos puso muchísimos problemas a la hora de analizar. Optamos por hacer algo más simple y recordamos que a los inicios de la programación los lenguajes tales como cobol y fortran cuando al realizar el proceso de compilación encontraban un solo error dejaban de ejecutar el compilador y directamente informaba el error así el usuario programador podía continuar desde ese punto.

Traducción a HTML

Este fue una de las partes más desafiantes para el equipo ya que como se explicó en la parte más arriba, nos llevó muchísimo tiempo decidir cual camino tomar, si trabajar desde el lexer o trabajar desde el parser, al inicio le dedicamos mucho tiempo y trabajamos desde el lexer pero después al recorrer la lista generada por el lexer para realizar la traducción vimos que había muchos token que solo ocupaban espacio y hacen que el algoritmo sea más complejo para su recorrido e identificación de token útiles. Por lo tanto creamos la lista de token en el mismo momento que se ejecutaba el parser. Con esto el recorrido fue mucho mas sencillo y logramos resolver



PUNTOS FUERTES

- Interfaz gráfica sencilla
- interfaz gráfica clara
- Gran precisión en el análisis léxico
- Gran precisión en la trata de errores del analizador sintáctico
- Código Prolíjo y entendible
- El usuario no tiene que cargar el archivo json desde el terminal, sino que busca directamente la ubicación del archivo

PUNTOS DÉBILES

- Al encontrar un error en el análisis sintáctico detiene completamente su funcionamiento
- En la última entrega el usuario no tiene la posibilidad de agregar el texto en el terminal

Bibliografía

- GitHub Oficial de PLY: <https://github.com/dabeaz/ply>
- Documentación oficial de python sobre expresiones regulares: <https://docs.python.org/es/3/library/re.html>
- <https://stackoverflow.com> para la consulta de dudas concretas sobre código.
- <https://www.dabeaz.com/ply/ply.html>
- <https://www.ericknavarro.io/2020/02/10/24-Mi-primer-proyecto-utilizando-PLY/>
- <https://www.aprendeaprogramar.com/mod/forum/discuss.php?d=2545>
- <https://docs.python.org/es/3/library/tkinter.html>

Fotos de algunas reuniones del meeting



TPI - Sintaxis y Semántica de los Lenguajes

