**UNIVERSIDAD**

**TECNOLÓGICA NACIONAL**

**FACULTAD REGIONAL**

**RESISTENCIA**

***1er ENTREGA TRABAJO PRÁCTICO INTEGRADOR: GRAMÁTICA***

***INTEGRANTES:***

**Puebla Ignacio**

**Ruiz Diaz Dario Nahuel**

**Sotelo Facundo**

**Manfredi Ezequiel**

**ASIGNATURA:** “Sintaxis y Semántica de los Lenguajes”

**CARRERA:** Ingeniería en Sistemas de Información

**CICLO LECTIVO:** Año 2024 - 1er Cuatrimestre

**INSTITUCIÓN:** “Universidad Técnica Nacional, Facultad Regional de Resistencia” (UTN FRRe)

**LUGAR:** Provincia del Chaco - Ciudad de Resistencia

**FECHA DE ENTREGA:** Domingo 2/06/2024

***ÍNDICE***

[***MATRIZ DE HABILIDADES*** 3](#_Toc170726114)

[***IDENTIDAD DEL GRUPO*** 3](#_Toc170726115)

[***ALIANZA*** 4](#_Toc170726116)

[***REGISTRO*** 5](#_Toc170726117)

[ **DOMINGO 14 DE ABRIL** 5](#_Toc170726118)

[ **Registros Extras del día:** 6](#_Toc170726119)

[ **SÁBADO 20 DE ABRIL** 7](#_Toc170726120)

[**-** **GRAMÁTICA:** 7](#_Toc170726121)

[**-** **Interfaz gráfica:** 11](#_Toc170726122)

[**-** **¿Cómo se hizo el Lexer?** 12](#_Toc170726123)

[**-** **Requisitos de ejecución:** 17](#_Toc170726124)

[**-** **Instalación del programa:** 17](#_Toc170726125)

[**-** **Formas de ejecutar el programa:** 17](#_Toc170726126)

[ ***Registros extra del día*** 18](#_Toc170726127)

[ **SÁBADO 29 DE JUNIO** 19](#_Toc170726128)

[**-** **Análisis sintáctico** 19](#_Toc170726129)

[**-** **¿Cómo traduce el parser?** 19](#_Toc170726130)

[**-** **Funciones auxiliares:** 20](#_Toc170726131)

[**-** **Modo de obtención del intérprete** 28](#_Toc170726132)

[**-** **Modo de ejecución del intérprete** 28](#_Toc170726133)

[**-** **Puntos fuertes y débiles del programa:** 32](#_Toc170726134)

[ ***Registros Extra del día*** 33](#_Toc170726135)

[ **Link al video presentación del Lexer-Parser:** 33](#_Toc170726136)

[**CONCLUSIONES** 34](#_Toc170726137)

[**BIBLIOGRAFÍA** 35](#_Toc170726138)

# ***MATRIZ DE HABILIDADES***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Integrantes** | **Inglés** | **Documentación** | **Programación** | **Investigación** | **Video** | **Testing** |
| **Puebla Ignacio** | ● | ★ | ● | ● |  | ● |
| **Manfredi Ezequiel Luis** | ● | ● | ★ | ● | ● | ★ |
| **Ruiz Diaz Dario Nahuel** |  | ● | ★ | ● | ★ | ★ |
| **Sotelo Facundo** | ★ | ★ | ● | ● |  | ● |

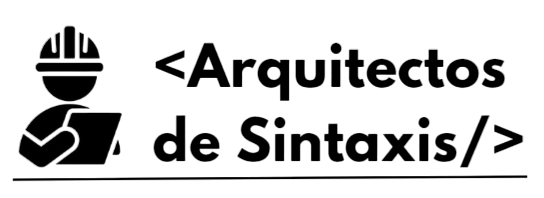
* **Espacio en blanco (“ ”): No sabe/no está interesado**
* **Estrella (★): Lo hace/es experto**
* **Punto (●) : Es competente/lo puede hacer pero no es su mayor interés**

# ***IDENTIDAD DEL GRUPO***

* **Nombre del grupo:**

“Arquitectos de Sintaxis”

* **Logo:**



# ***ALIANZA***

Los integrantes del equipo Arquitectos de Sintaxis: Manfredi Ezequiel, Ruiz Diaz Dario Nahuel, Puebla Ignacio y Sotelo Facundo, se comprometen a:

Reunirse al menos una vez por semana los días sábado o domingo entre las 17:00 y 18:00 hs vía Discord a través de una llamada en la cual poder conversar, decidir, debatir e informar con respecto al progreso del trabajo práctico integrador y las tareas de cada integrante. A su vez, se hará uso de un grupo de WhatsApp como medio de comunicación habitual entre los integrantes del equipo y para la planificación de las reuniones.

Todo ello con el objetivo de, no solo aprobar el Trabajo Práctico Integrador (y si dios quiere aprobar la materia de forma directa para no rendir final) sino también el de realizar el mejor trabajo posible en el tiempo dado.

Acordamos que la organización del equipo será la siguiente: nos dividiremos y trabajaremos en dos grupos (por lo menos durante esta primera entrega del TPI). El primer grupo de dos integrantes (Manfredi Ezequiel y Ruiz Diaz Dario Nahuel) será el encargado mayormente de todo lo referido a la programación y el diseño; mientras que el segundo grupo de dos integrantes (Puebla Ignacio y Sotelo Facundo) será el encargado de registrar y documentar, ayudar con el diseño de gramática, y brindar asistencia en lo posible. Ello incluye identificar errores, ayudar con el desarrollo del programa si fuese necesario, asistir con el diseño y la creación de la gramática, entre otras tareas.

Si bien cada grupo tiene su función, no serán grupos aislados que trabajen individualmente en todo momento, ya que se utilizarán las vías de comunicación antes mencionadas para informar, consultar, pedir ayuda u opinión de los demás integrantes del equipo; así como cualquier otra situación que surja durante la realización del Trabajo Práctico Integrador.

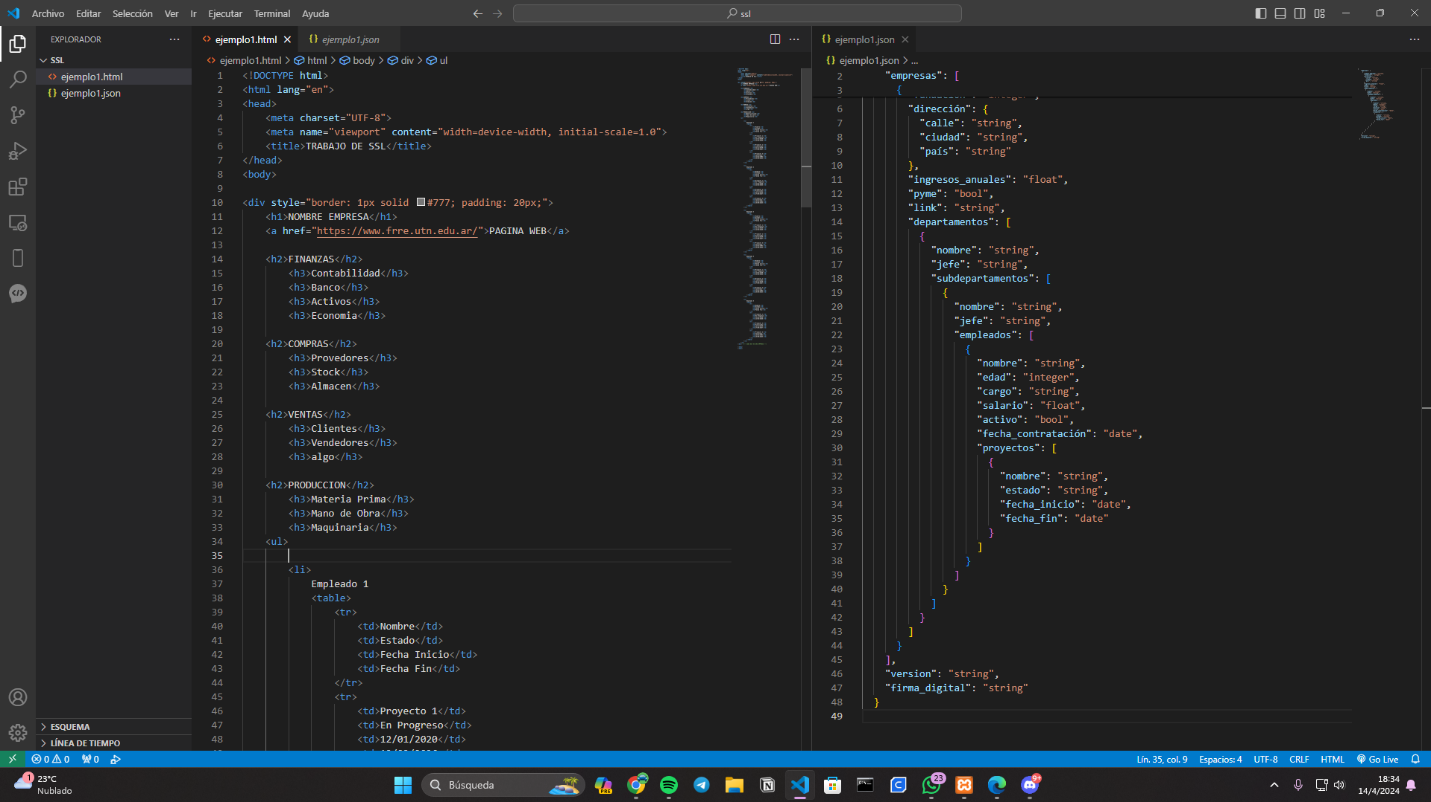
# ***REGISTRO***

## **DOMINGO 14 DE ABRIL**

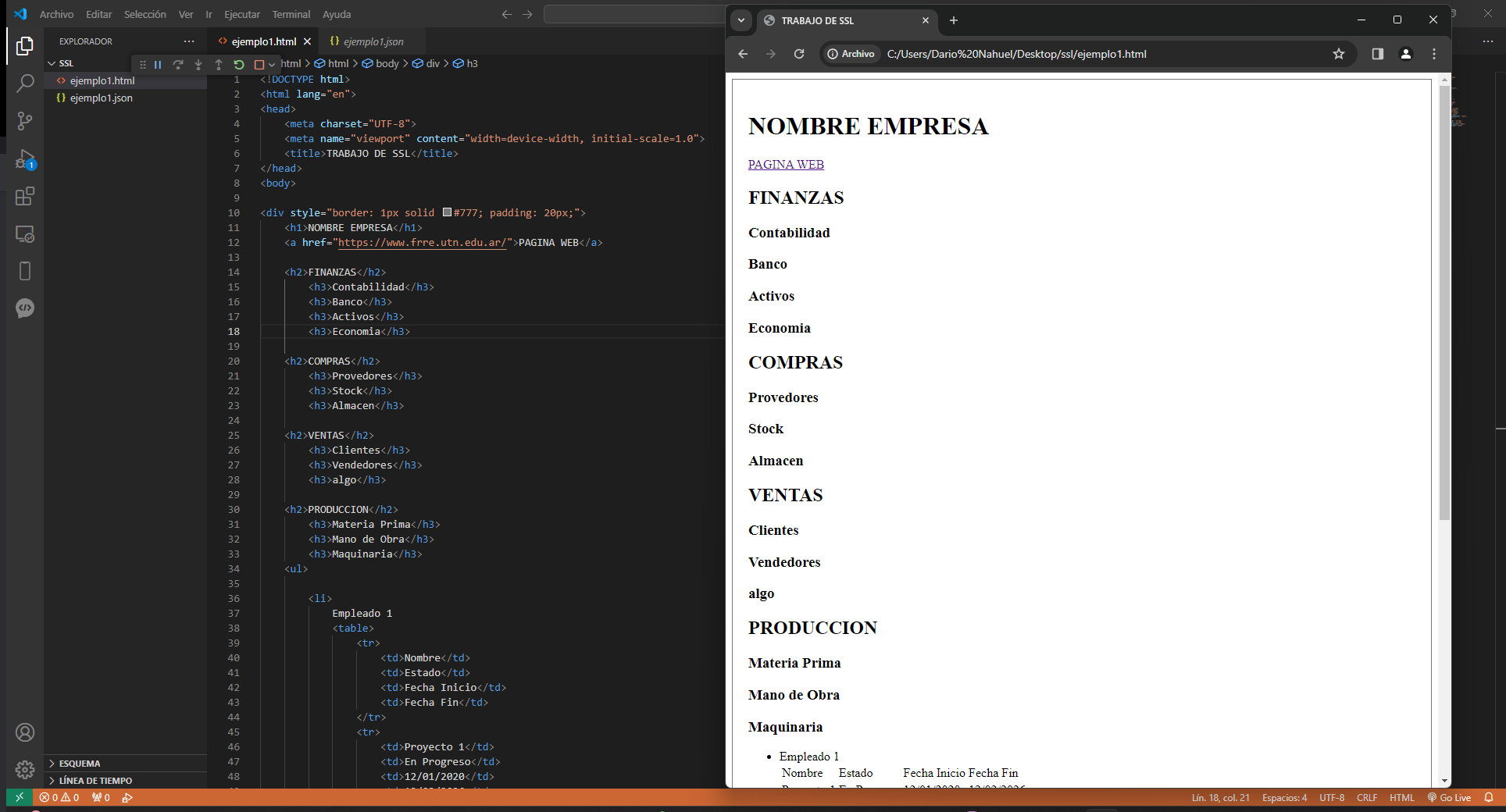
Nos reunimos vía Discord para planificar la primera parte de la entrega del TPI. Para ello, comenzamos haciendo un documento compartido de Google docs, como borrador de lo que luego sería nuestro documento Word. Lo primero que hicimos en dicho documento fué la matriz de habilidades. Nos pusimos de acuerdo para definir cada categoría y luego, individualmente, cada uno fue definiendo las capacidades y cualidades en las que se destaca. Luego de realizar la matriz de habilidades nos pusimos a planificar cómo sería la estructura de nuestro trabajo. Definimos los roles de cada integrante del equipo y nos dividimos en dos grupos: uno encargado de diseñar y programar; y el otro de llevar el registro, la documentación y asistir en lo posible.

Una vez definidos los equipos, hablando acerca del desarrollo del trabajo, uno de los integrantes (Ruiz Diaz Nahuel) propuso una interfaz gráfica simple como idea para la presentación de la información, la cual dividiese la pantalla de la siguiente forma: a la izquierda, iría el contenido del archivo de entrada; a la derecha, el contenido del archivo de salida; y debajo, habría una consola que muestre en pantalla los errores (en caso de haberlos).

El **grupo 1** encargado de la programación tomó un ejemplo de archivo en JSON para ver su estructura básica y cómo se podría traducir a HTML con el fin de usarla como base para la realización de la gramática en HTML (el lenguaje con el cual definimos que se trabajaría la gramática).



Captura del proceso de traducción de JSON a HTML del ejemplo de gramática



Captura de prueba de ejecución del código HTML

Mientras tanto, en el **grupo 2** encargado del registro y la documentación, Sotelo Facundo registraba el trabajo realizado por el grupo 1 mediante capturas de pantalla y la descripción general de lo que se iba haciendo; y Puebla Ignacio redactaba la Alianza del equipo y aportaba al registro del día.

### **Registros Extras del día:**

Al final de la reunión, se acordó crear un repositorio de GitHub para que cada integrante tuviera un mejor y más cómodo acceso al código y al documento Word. A su vez, se realizaron modificaciones al server de Discord añadiendo nuevos canales de “código” y “fotos” para poder compartir a todo el grupo las capturas de pantalla y los archivos.

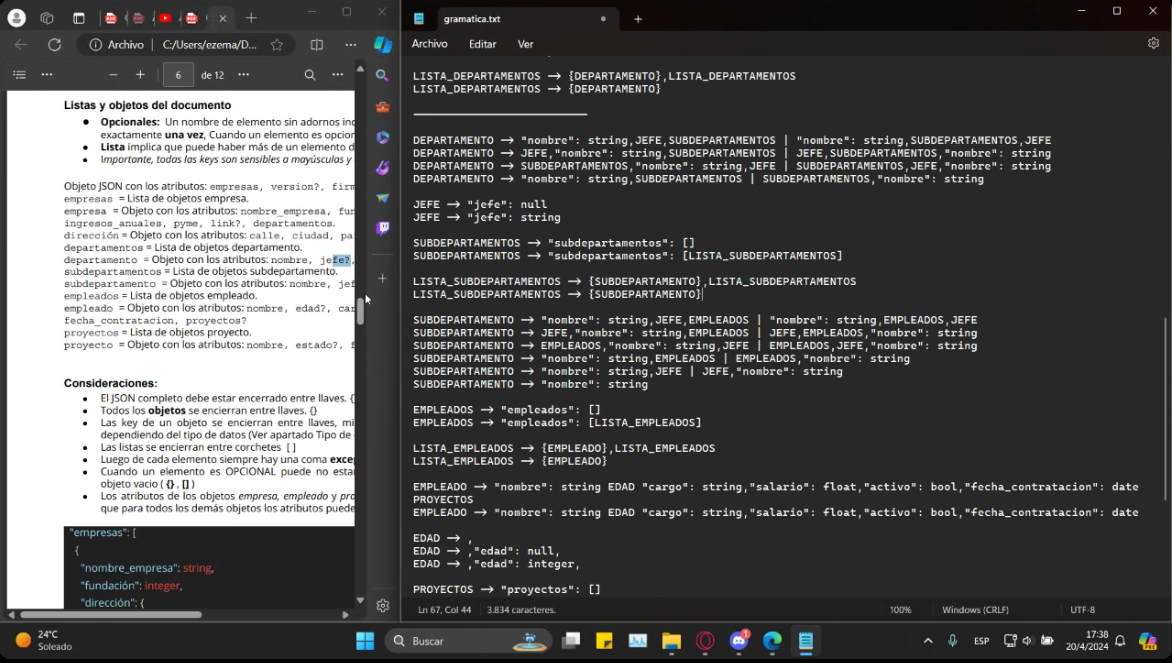
En un determinado momento de la reunión, el integrante Ruiz Diaz Nahuel, diseñó el logo del grupo. Un logo simple pero que tuviera relación con el nombre del grupo y que le diera una mayor identidad.

El integrante Manfredi Ezequiel es el autor del nombre del grupo “Arquitectos de Sintaxis”.

## **SÁBADO 20 DE ABRIL**

Nos conectamos vía llamada de Discord. El objetivo de la reunión inicialmente era reunirnos para diseñar la gramática entre todos, sin embargo, para sorpresa del resto del resto de los integrantes del equipo, resulta que Manfredi Ezequiel ya había hecho la gramática. Entonces empezamos a analizar la gramática en busca de errores o faltas y para mejorar su legibilidad.

El integrante Puebla Ignacio completó las premisas de la alianza del equipo y se corrigió algunos aspectos de su redacción para hacerla más legible. A su vez, también corrigió la redacción y añadió algo de contenido al registro. Mientras que el integrante Sotelo Facundo realizaba el registro y tomaba captura a la gramática.



Captura de la gramática sin corregir

En un punto de la reunión, luego de revisar la gramática diseñada por el alumno Manfredi Ezequiel y terminar la documentación, dedicamos las siguientes hora y media-dos horas a practicar la producción de gramáticas para el parcialito del día 25 de abril.

* **GRAMÁTICA:**

Gramatica JSON: G(N,T,P,Σ)

N: {

CONTENIDO; EMPRESAS; VERSION; FIRMA; LISTA\_EMPRESAS; EMPRESA; NOMBRE\_EMPRESA; FUNDACION;

DIRECCION; INGRESOS\_ANUALES; PYME; LINK; DEPARTAMENTOS; ATRIBUTOS\_DIRECCION;

LISTA\_DEPARTAMENTOS; CALLE; CIUDAD; PAIS; DEPARTAMENTO; NOMBRE; JEFE; SUBDEPARTAMENTOS;

LISTA\_SUBDEPARTAMENTOS; SUBDEPARTAMENTO; EMPLEADOS; LISTA\_EMPLEADOS; EMPLEADO; EDAD;

CARGO; SALARIO; ACTIVO; FECHA\_CONTRATACION; PROYECTOS; LISTA\_PROYECTOS; PROYECTO; ESTADO;

FECHA\_INICIO; FECHA\_FIN;

}

T: {

{; }; ,; [; ]; "; ; :; string; integer; bool; float; date; null; version; firma\_digital; empresas; nombre\_empresa;

fundacion; dirección; ingresos\_anuales; pyme; link; departamentos; calle; ciudad; pais; nombre; jefe;

subdepartamentos; empleados; edad; cargo; salario; activo; fecha\_contratacion; proyectos; estado; fecha\_inicio;

fecha\_fin;

}

P: {

Σ -> {CONTENIDO}

// combinatoria del orden de los campos considerando los campos opcionales VERSION y FIRMA

CONTENIDO -> EMPRESAS, VERSION, FIRMA | EMPRESAS, FIRMA, VERSION

CONTENIDO -> VERSION, FIRMA, EMPRESAS | VERSION, EMPRESAS, FIRMA

CONTENIDO -> FIRMA, EMPRESAS, VERSION | FIRMA, VERSION, EMPRESAS

CONTENIDO -> EMPRESAS, VERSION | EMPRESAS, FIRMA

CONTENIDO -> VERSION, EMPRESAS | FIRMA, EMPRESAS

CONTENIDO -> EMPRESAS

VERSION -> "version": null

VERSION -> "version": string

FIRMA -> "firma\_digital": null

FIRMA -> "firma\_digital": string

EMPRESAS -> "empresas": [LISTA\_EMPRESAS]

// recursion de empresas

LISTA\_EMPRESAS -> {EMPRESA}, LISTA\_EMPRESAS

LISTA\_EMPRESAS -> {EMPRESA}

// manteniendo el orden de los campos considerando los campos opcionales DIRECCION y LINK

EMPRESA -> NOMBRE\_EMPRESA, FUNDACION, DIRECCION, INGRESOS\_ANUALES, PYME, LINK, DEPARTAMENTOS

EMPRESA ->NOMBRE\_EMPRESA, FUNDACION, INGRESOS\_ANUALES, PYME, LINK, DEPARTAMENTOS

EMPRESA -> NOMBRE\_EMPRESA, FUNDACION, DIRECCION, INGRESOS\_ANUALES, PYME, DEPARTAMENTOS

EMPRESA -> NOMBRE\_EMPRESA, FUNDACION, INGRESOS\_ANUALES, PYME, DEPARTAMENTOS

NOMBRE\_EMPRESA -> "nombre\_empresa": string

FUNDACION -> "fundacion": integer

DIRECCION -> "direccion": null

DIRECCION -> "direccion": {}

DIRECCION -> "direccion": {ATRIBUTOS\_DIRECCION}

INGRESOS\_ANUALES -> "ingresos\_anuales": float

PYME -> "pyme": bool

LINK -> "link": null

LINK -> "link": string

DEPARTAMENTOS -> "departamentos": [LISTA\_DEPARTAMENTOS]

// combinatoria del orden de los campos

ATRIBUTOS\_DIRECCION -> CALLE, CIUDAD, PAIS | CALLE, PAIS, CIUDAD

ATRIBUTOS\_DIRECCION -> CIUDAD, CALLE, PAIS | CIUDAD, PAIS, CALLE

ATRIBUTOS\_DIRECCION -> PAIS, CALLE, CIUDAD | PAIS, CIUDAD, CALLE

CALLE -> "calle": string

CIUDAD -> "ciudad": string

PAIS -> "pais": string

// recursion de departamentos

LISTA\_DEPARTAMENTOS -> {DEPARTAMENTO}, LISTA\_DEPARTAMENTOS

LISTA\_DEPARTAMENTOS -> {DEPARTAMENTO}

// combinatoria del orden de los campos considerando el campo opcional JEFE

DEPARTAMENTO -> NOMBRE, JEFE, SUBDEPARTAMENTOS | NOMBRE, SUBDEPARTAMENTOS, JEFE

DEPARTAMENTO -> JEFE, NOMBRE, SUBDEPARTAMENTOS | JEFE, SUBDEPARTAMENTOS, NOMBRE

DEPARTAMENTO -> SUBDEPARTAMENTOS, NOMBRE, JEFE | SUBDEPARTAMENTOS, JEFE, NOMBRE

DEPARTAMENTO -> NOMBRE, SUBDEPARTAMENTOS | SUBDEPARTAMENTOS, NOMBRE

NOMBRE -> "nombre": string

JEFE -> "jefe": null

JEFE -> "jefe": string

SUBDEPARTAMENTOS -> "subdepartamentos": [LISTA\_SUBDEPARTAMENTOS]

// recursion de subdepartamentos

LISTA\_SUBDEPARTAMENTOS -> {SUBDEPARTAMENTO}, LISTA\_SUBDEPARTAMENTOS

LISTA\_SUBDEPARTAMENTOS -> {SUBDEPARTAMENTO}

// combinatoria del orden de los campos considerando los campos opcionales JEFE y EMPLEADOS

SUBDEPARTAMENTO -> NOMBRE, JEFE, EMPLEADOS | NOMBRE, EMPLEADOS, JEFE

SUBDEPARTAMENTO -> JEFE, NOMBRE, EMPLEADOS | JEFE, EMPLEADOS, NOMBRE

SUBDEPARTAMENTO -> EMPLEADOS, NOMBRE, JEFE | EMPLEADOS, JEFE, NOMBRE

SUBDEPARTAMENTO -> NOMBRE, EMPLEADOS | EMPLEADOS, NOMBRE

SUBDEPARTAMENTO -> NOMBRE, JEFE | JEFE, NOMBRE

SUBDEPARTAMENTO -> NOMBRE

EMPLEADOS -> "empleados": null

EMPLEADOS -> "empleados": []

EMPLEADOS -> "empleados": [LISTA\_EMPLEADOS]

// recursion de empleados

LISTA\_EMPLEADOS -> {EMPLEADO}, LISTA\_EMPLEADOS

LISTA\_EMPLEADOS -> {EMPLEADO}

// manteniendo el orden de los campos considerando los campos opcionales EDAD y PROYECTOS

EMPLEADO -> NOMBRE, EDAD, CARGO, SALARIO, ACTIVO, FECHA\_CONTRATACION, PROYECTOS

EMPLEADO -> NOMBRE, EDAD, CARGO, SALARIO, ACTIVO, FECHA\_CONTRATACION

EMPLEADO -> NOMBRE, CARGO, SALARIO, ACTIVO, FECHA\_CONTRATACION, PROYECTOS

EDAD -> "edad": null

EDAD -> "edad": integer

CARGO -> "cargo": string

SALARIO -> "salario": float

ACTIVO -> "activo": bool

FECHA\_CONTRATACION -> "fecha\_contratacion": date

PROYECTOS -> "proyectos": null

PROYECTOS -> "proyectos": []

PROYECTOS -> "proyectos": [LISTA\_PROYECTOS]

// recursion de proyectos

LISTA\_PROYECTOS -> {PROYECTO}, LISTA\_PROYECTOS

LISTA\_PROYECTOS -> {PROYECTO}

// manteniendo el orden de los campos considerando los campos opcionales ESTADO y FECHA\_FIN

PROYECTO -> NOMBRE, ESTADO, FECHA\_INICIO, FECHA\_FIN

PROYECTO -> NOMBRE, ESTADO, FECHA\_INICIO

PROYECTO -> NOMBRE, FECHA\_INICIO, FECHA\_FIN

ESTADO -> "estado": null

ESTADO -> "estado": string

FECHA\_INICIO -> "fecha\_inicio": date

FECHA\_FIN -> "fecha\_fin": null

FECHA\_FIN -> "fecha\_fin": date

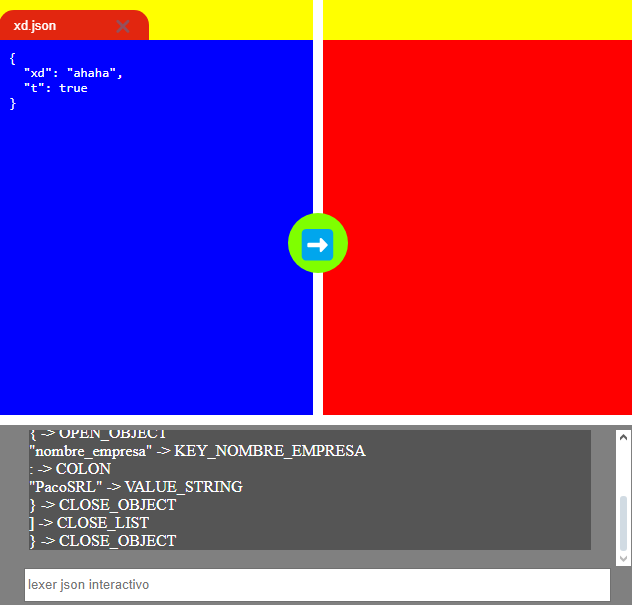
}

* **DOMINGO 26 DE MAYO**

Nos reunimos vía Discord a través de una llamada para realizar la documentación del Lexer desarrollado por el miembro del equipo Manfredi Ezequiel.

### **Interfaz gráfica:**

Teniendo en cuenta tanto el Lexer como el futuro Parser, se creó una interfaz provisional dividida en dos partes. En la parte superior se colocaron dos cuadros (izquierdo y derecho) separados por una línea que muestran el archivo de entrada y el de salida respectivamente (lo que nos servirá para el Parser); mientras que en la parte inferior se encuentra un terminal en el cual se ingresa algún input (por ejemplo una palabra), y ejecuta el Lexer devolviendo como salida los tokens identificados.



Captura de la interfaz provisional



Interfaz luego de algunas modificaciones

La estructura de la interfaz está hecha en html y la parte funcional de la interfaz (la captura de datos, la impresión de archivos, etc) se hizo en JavaScript. A su vez, el Lexer está hecho en Python, al igual que un servidor. Dicho servidor se encarga de administrar las peticiones y brindar la información necesaria tanto al Lexer como al futuro Parser.

Lo que más nos interesa de la interfaz (por estar relacionado con el Lexer) es la terminal de la parte inferior. Resumidamente la terminal funciona de la siguiente forma: en base a un input se envía la información al servidor, el cual se la suministra al Lexer, este analiza y devuelve los resultados que son recibidos en la terminal y se la procesa para mostrarla amigablemente al usuario. Además, todos los errores se mostrarán en la salida clasificados según el tipo y regla no cumplida.

### **¿Cómo se hizo el Lexer?**

Se utilizó la documentación y ejemplos brindados por la cátedra como guía para empezar a desarrollar el lexer. Primero se definieron las palabras clave; los token; y luego, para cada tipo de token, las expresiones regulares, las cuales nos sirven para detectar los token según su tipo.

La parte más complicada del desarrollo del Lexer fué el definir las expresiones regulares, en donde las más problemáticas fueron las expresiones regulares para los strings, las fechas y la url; pero tras probar varias opciones se consiguió que funcione.

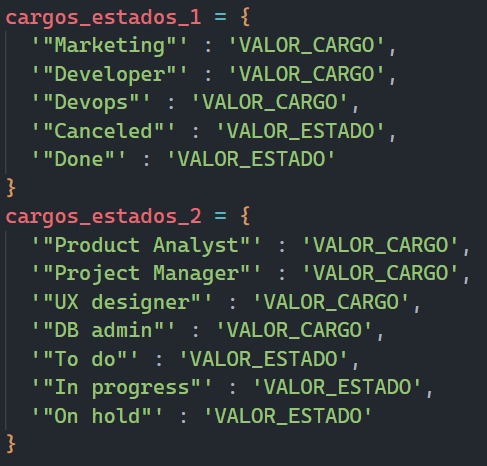
El reconocimiento y la notificación de errores funciona de la siguiente manera: A partir del input dado se analiza la totalidad del mismo acumulando todos los errores de este y los tokens debidamente reconocidos.

Los errores pueden ser de dos tipos: no\_tokens y tokens que no cumplen las reglas. Los primeros son caracteres que no hicieron match con ninguna expresión regular, mientras que los segundos son lexemas que hicieron match con alguna expresión regular. Se clasifican por tipo de token y cada uno tiene reglas que cumplir. Dichas reglas explicitan el porqué del error detectado.

#### **Reconocimientos de tokens:**

**Conjunto:** lexemas que no cambian y están agrupadas en cierto token.

Los conjuntos que definimos en el Lexer son:

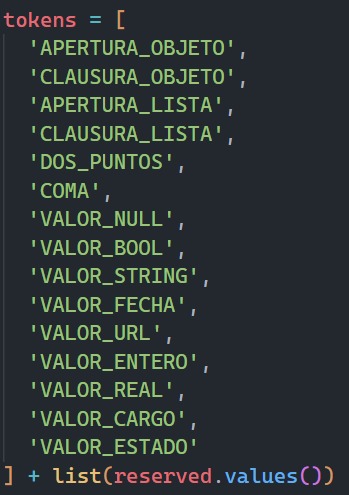


**Palabras reservadas:** lexemas que no cambian y por cada una hay un token.

Las palabras reservadas definidas en el Lexer son:



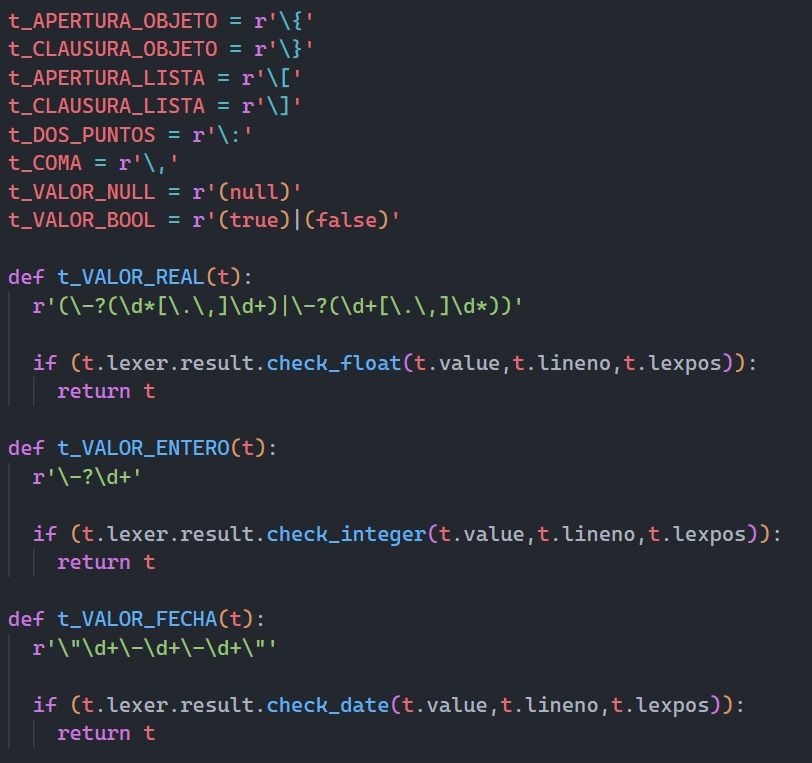
**Tokens:** Los tokens definidos son:

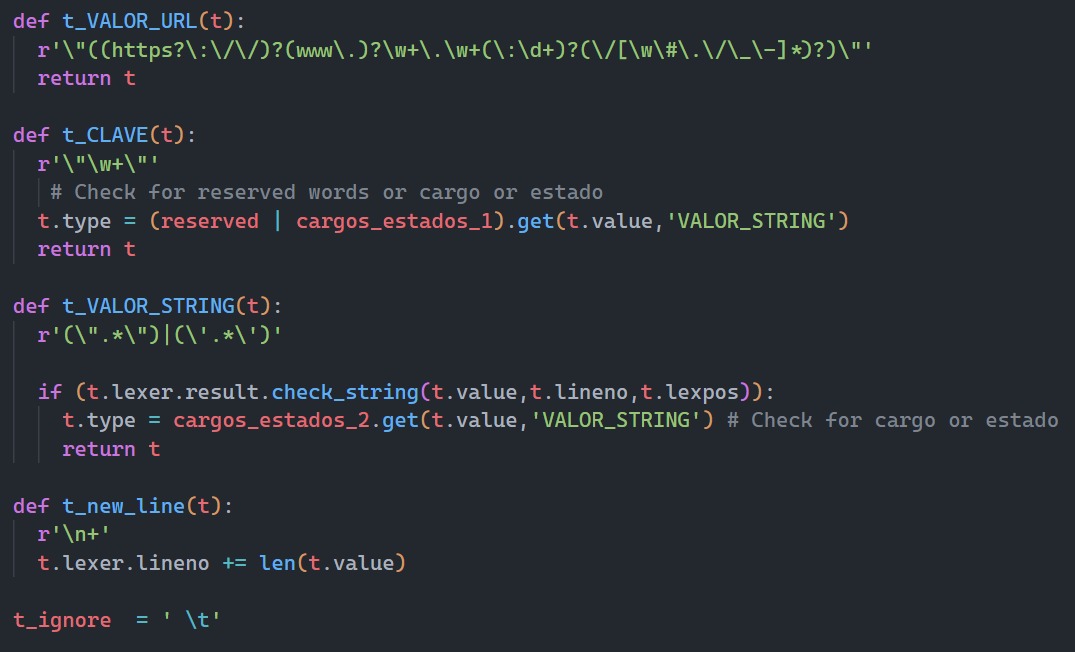


**Expresión regular**: regla generalizada que reconoce ciertos tipos de lexemas.

Ciertas expresiones regulares permiten el reconocimiento de ciertos lexemas que se considerarían errores pero que el usuario podría cometer. Estos lexemas de más sirven para detectar y clasificar el no cumplimiento de ciertas reglas para su posterior notificación.

Las expresiones regulares definidas en el Lexer son:





### **Requisitos de ejecución:**

El programa fué ejecutado en Windows 11 y los requisitos son:

* Python versión. 3.12.3
* Paquete PLY (Python Lex-Yacc) ver 3.11
* Flask versión 3.0.3

### **Instalación del programa:**

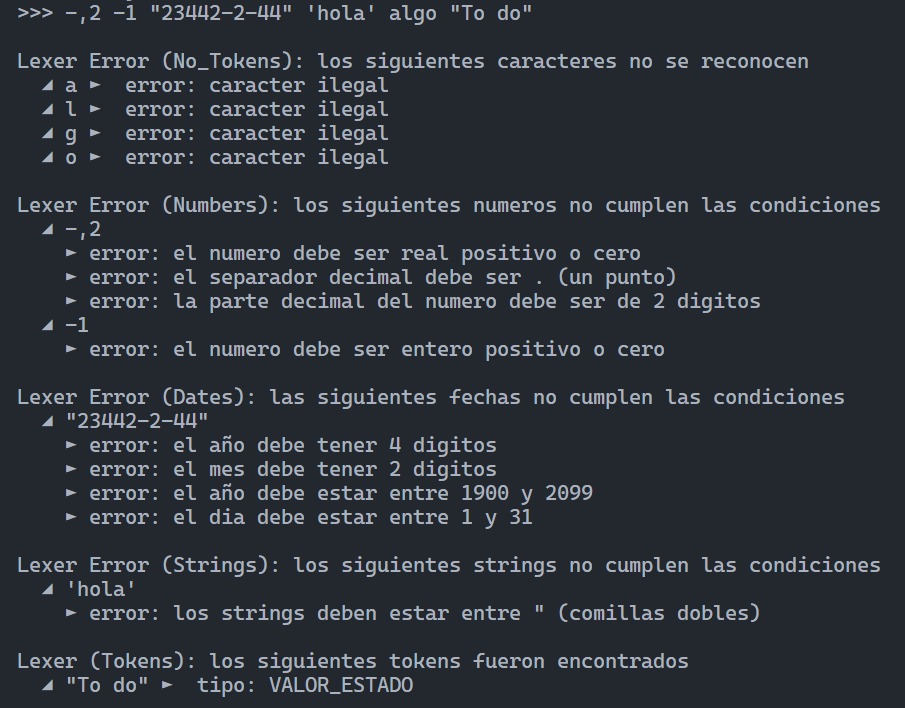
* Instalar Python
* Luego en el terminal ejecutar: pip install ply y luego pip install Flask

### **Formas de ejecutar el programa:**

Contamos con dos maneras de ejecutar el programa: la primera es por terminal y la segunda por UI (interfaz de usuario).

Por terminal:

* En el terminal en el directorio src ejecutar: python ./lex.py
* Se ejecutará el Lexer interactivo, donde se puede ingresar los inputs y se mostraran los resultados del análisis Lexico.



Por interfaz gráfica:

* En el terminal en el directorio src ejecutar: flask --app ./server.py run
* Saldrá una url en donde estará la UI ejecutándose.

Para terminar los procesos presione ctrl + C

#### **Registros extra del día:**

Para la entrega del Parser se tiene pensado añadir un botón en la interfaz para descargar el archivo de salida generado. También un botón en el medio para ejecutar el futuro Parser y mostrar el archivo de salida dado el archivo de entrada ingresado. Además, se planea mejorar el diseño de la interfaz.

La idea final que se tiene en mente para la interfaz es que el usuario pueda arrastrar un archivo de entrada, luego ejecutar el programa y que este genere un archivo de salida el cual pueda descargar pulsando un botón.

## **SÁBADO 29 DE JUNIO**

Nos reunimos vía Discord a través de una llamada para realizar la documentación final del trabajo, corregir algunos errores del Parser, y repartirnos el trabajo para hacer el video explicativo del proyecto.

### **Análisis sintáctico**

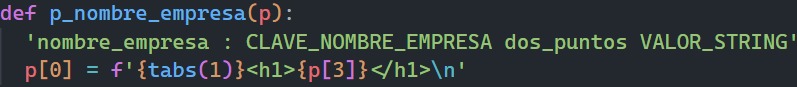
Para realizar el análisis sintáctico se tomó las producciones de la GLC previamente definidas y de esa forma obtuvimos las producciones generativas que aceptan el contenido.

### **¿Cómo traduce el parser?**

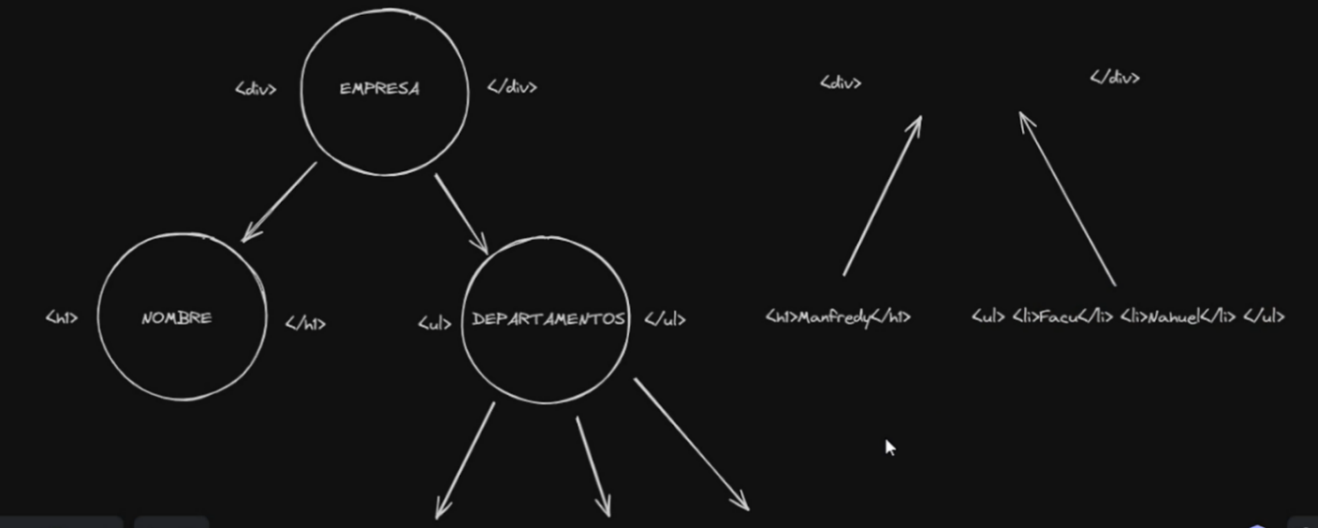
La gramática empieza desde el json (json es el equivalente al nodo raiz o sigma que aparece en la gramática) y se va transformando en un árbol de derivación. Los no terminales N son las palabras en mayúscula y los terminales T son aquellas en minúscula.

Podemos explicar el procedimiento que sigue el programa de forma análoga como si se estuviese hablando de un árbol de derivación, del cual se extienden todas las ramas y nodos hijos del nodo raíz (json); y luego se derivan sus nodos de abajo hacia arriba. Se toma el contenido del json y forma el árbol de derivación de arriba hacia abajo y, si todo está correcto, se realiza la traducción partiendo de los nodos hoja hacia el nodo raíz. Finalmente se junta todo en un solo string en el nodo raíz el cual se imprime en pantalla.





Ejemplo de cómo está definido este proceso en el código.



Esquema para ejemplificar gráficamente cómo funciona

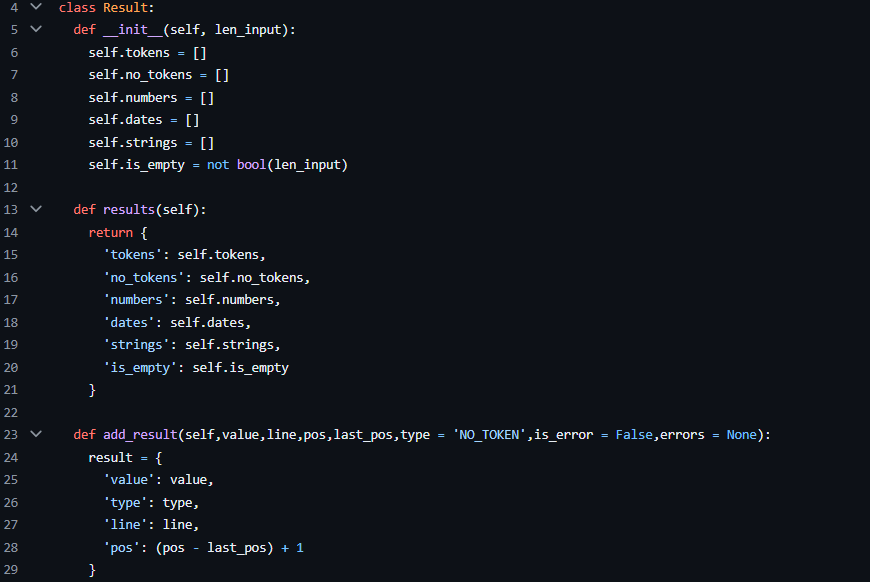
Si se encuentra un error a la hora de hacer el árbol se deja de traducir y acumula el error (sin romperse), analizando los demás nodos que no tengan error, es decir, obviamos el error hasta el final y realiza la resincronización de tokens. Esto es así ya que cuando ocurre un error en el nodo se saltean los tokens sucesores hasta la siguiente coma, clausura de objeto o lista, y a partir de dicho token se continúa el análisis hasta finalizar.

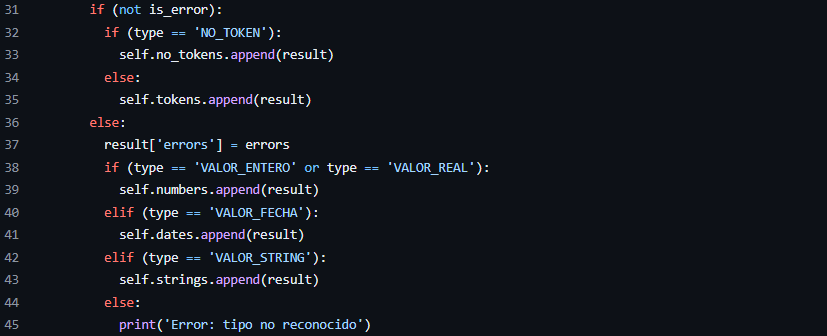
### **Funciones auxiliares:**

Las funciones auxiliares definidas en utils.py son:

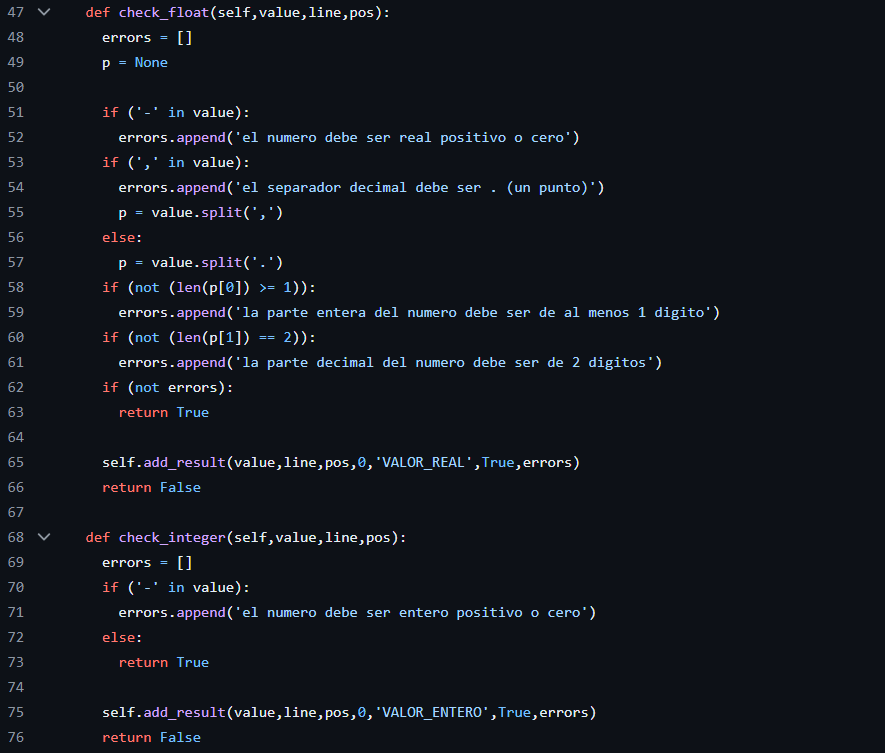
#### **Para el Lexer:**

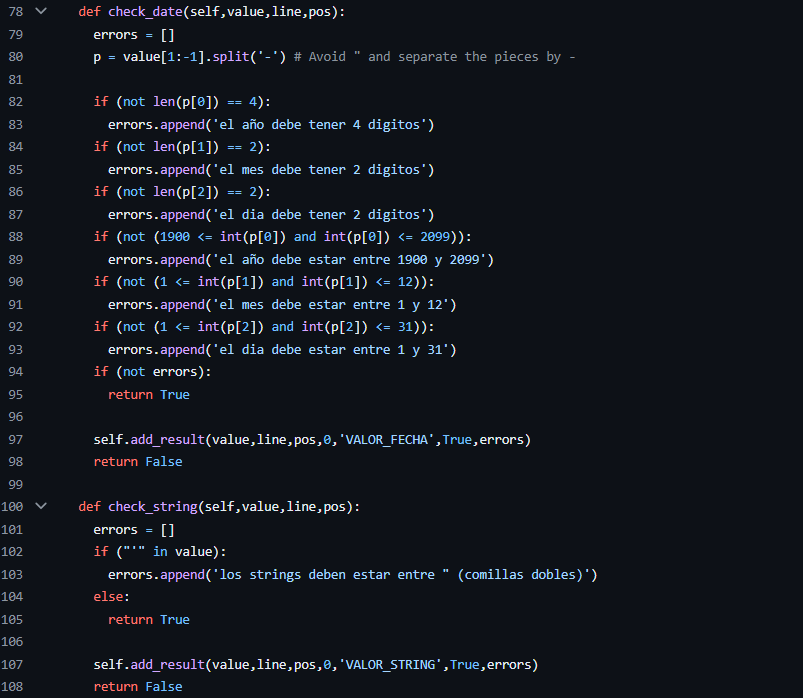
#### **Class result:** Engloba métodos o funciones sobre el análisis léxico y de los tokens. Acumula tokens, no tokens, errores numéricos, de fecha y de string (errores contemplados).



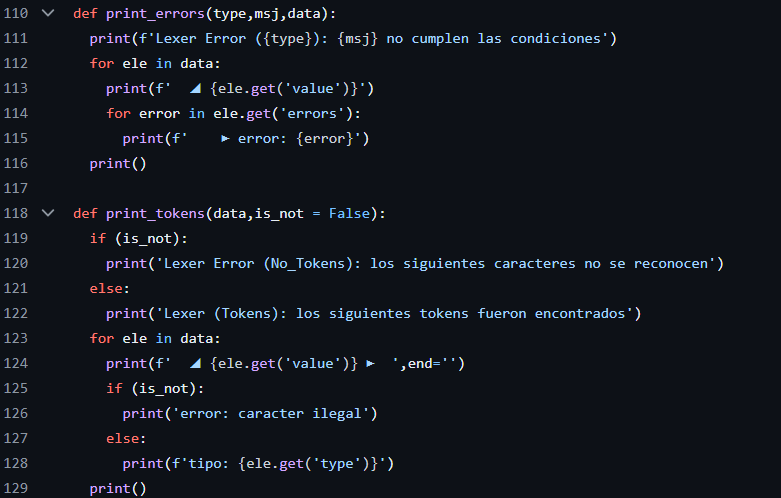


* **check\_float, check\_integer, check\_string y check\_date:** Se encargan de chequear cada resultado, del chequeo de tokens, no tokens, números, fechas y strings.



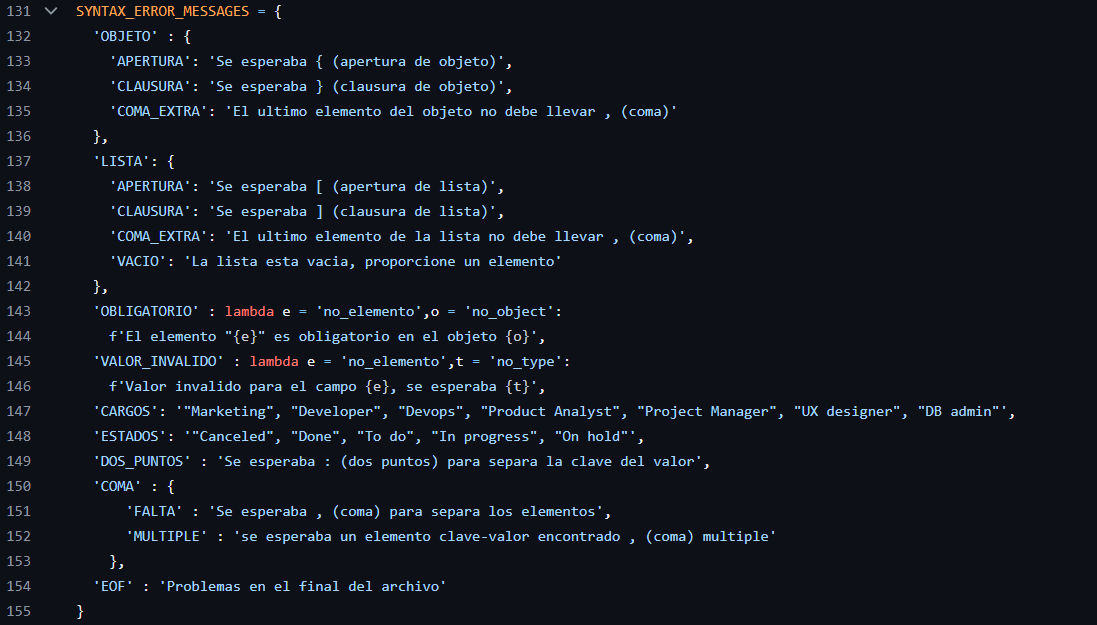


* **print\_errors y print\_tokens:** Son dos procedimientos independientes que sirven para mostrar por pantalla la lista de errores y de tokens en el terminal.



#### **Para el Parser:**

* **SYNTAX\_ERROR\_MESSAGES:** Funciona como una especie de “diccionario de errores de sintaxis”, el cual contiene todos los tipos de errores contemplados. Nos sirve para luego buscar el tipo de error y mostrar por pantalla su respectiva causa.



* **Class syntaxerror:** Se encarga de acumular los errores de sintaxis.



* **tabs**: Es una función que genera tabulaciones (espacios) para la traducción.



* **Class FilesHandler:** Se encarga de acumular los archivos de entrada para la ejecución del terminal.

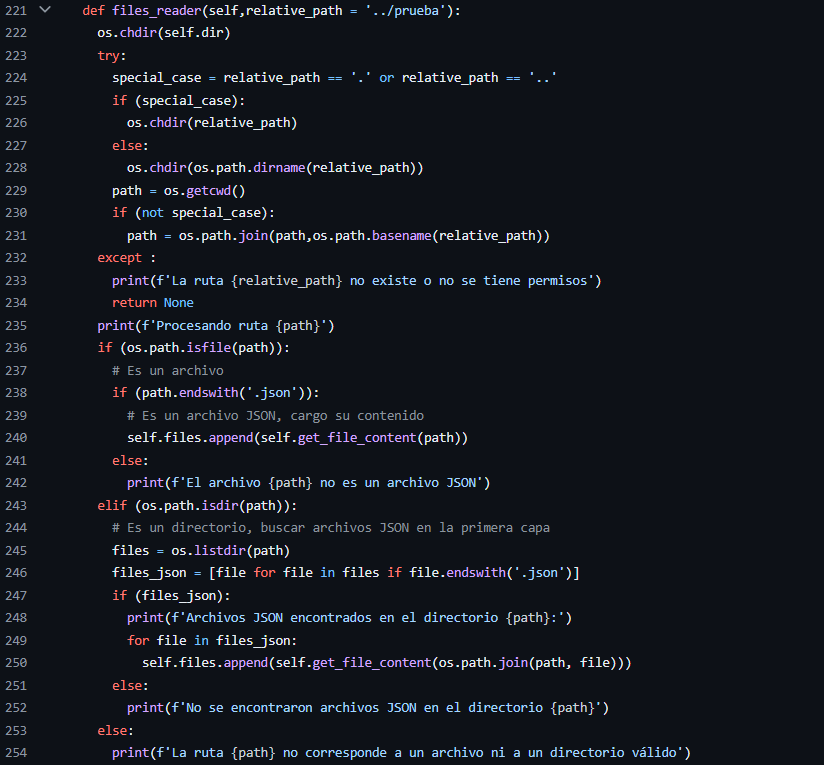


* **args\_reader:** Es un método de lectura de argumentos, a la hora de ejecutar el parser controla si se pasaron argumentos. Si hay argumentos hace una lectura de los archivos ingresados como argumentos; si no se ingresaron argumentos, setea o configura la interfaz de la terminal para cargar diferentes rutas y realiza la lectura de los archivos de la ruta una por una.

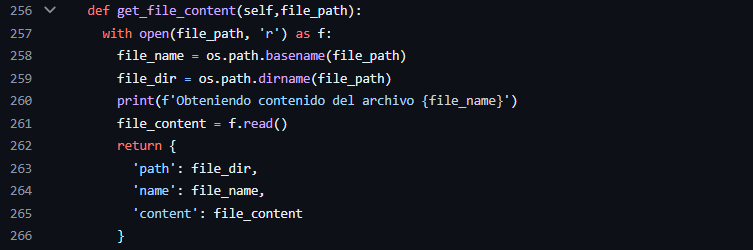


* **files\_reader:** Recibe la ruta del archivo y clasifica según tipo de ruta (de archivo o de directorio). Si es de archivo verifica si la extensión del archivo es .json; si no lo es, tira error; si lo es, lo busca y obtiene su contenido. Si es de directorio, va al directorio y busca en la primer capa de nivel todos los archivos .json; si no encuentra tira mensaje; y si lo encuentra obtiene el contenido de todos los archivos y los acumula.

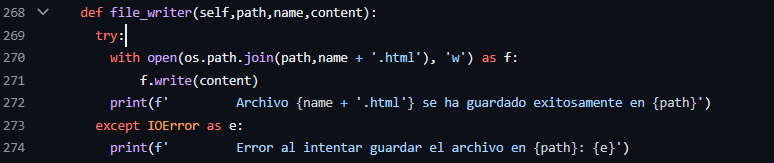
Finalmente, también contempla los problemas con el permiso de lectura de la ruta.



* **get\_file\_content:** Obtiene el contenido del archivo a través de una ruta ya depurada y arma un objeto con la ruta, el nombre y el contenido.



* **file\_writer:** Recibe la ruta, nombre y contenido del archivo a grabar. Al nombre del archivo se le adjunta la extensión **.html** y guarda el contenido en el archivo en la ruta. En caso de no guardarse correctamente, indica error al intentar guardar el archivo; sino indica que el archivo se ha guardado exitosamente.



* **RESYNC\_TOK:** Son tokens de resincronización. Tokens de fin de sentencia de la gramática.



### **Modo de obtención del intérprete**

El lexer interactivo ubicado en el directorio src, en el archivo lex.py tiende dependencias el paquete ply y funciones y objetos definidos en el archivo utils.py del mismo directorio

El parser de archivos se encuentra en el directorio src, en el archivo yacc.py tiene como dependencia el paquete ply, el lex.py y objetos y funciones del archivo utils en el mismo directorio.

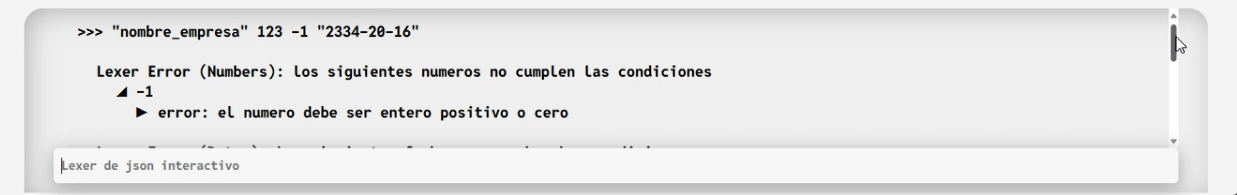
### **Modo de ejecución del intérprete**

Hay dos tipos de ejecución: una por terminal y otra por medio de la interfaz gráfica o UI hecha por nosotros.

#### **Ejecución mediante la UI:**

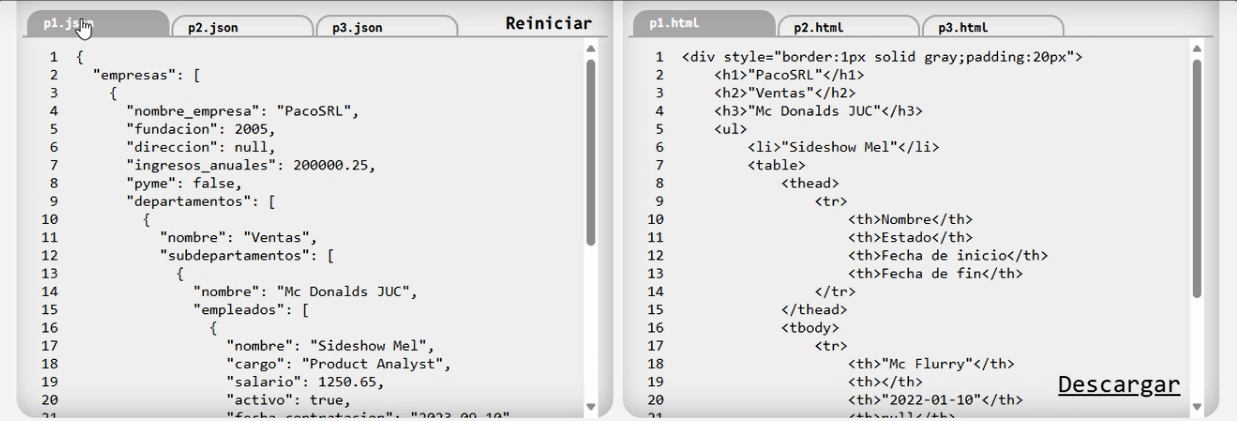
Tenemos una interfaz gráfica que integra tanto el lexer como el parser y funciona de la siguiente manera:

Ejecución del lexer por UI: La ejecución del lexer a través de la UI se realiza mediante un input al cual se le dan sentencias y detecta los tokens y los errores en dichas sentencias.

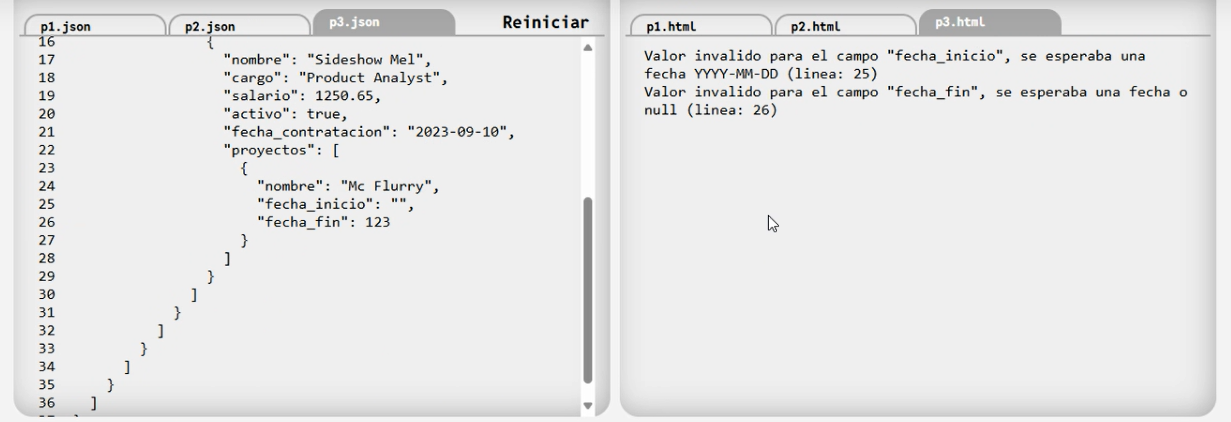


Captura del terminal de UI para el lexer

Ejecución del Parser por UI: Para la ejecución del parser se selecciona archivos .json a través de la interfaz gráfica, que luego se analiza y muestra por pantalla los errores o la traducción.



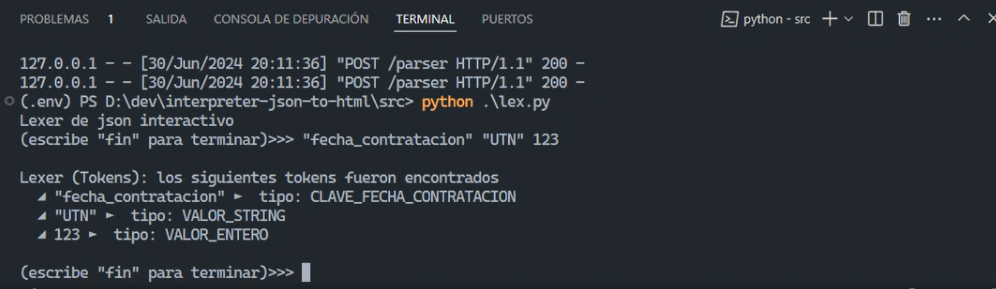


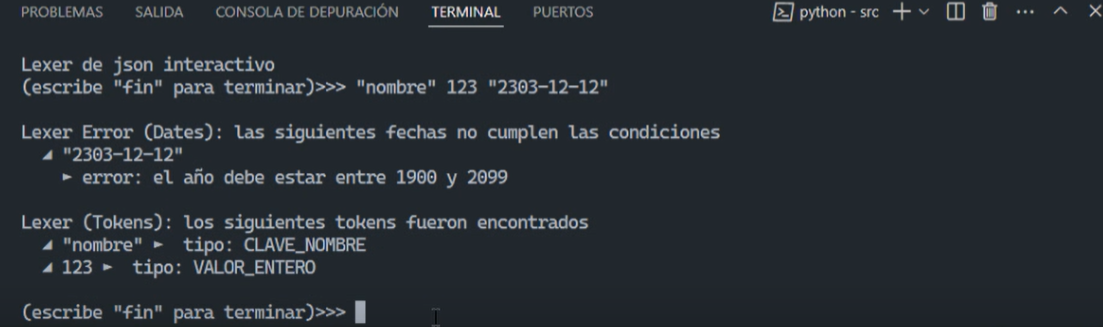


Capturas de la interfaz para el parser

#### **Ejecución mediante Terminal:**

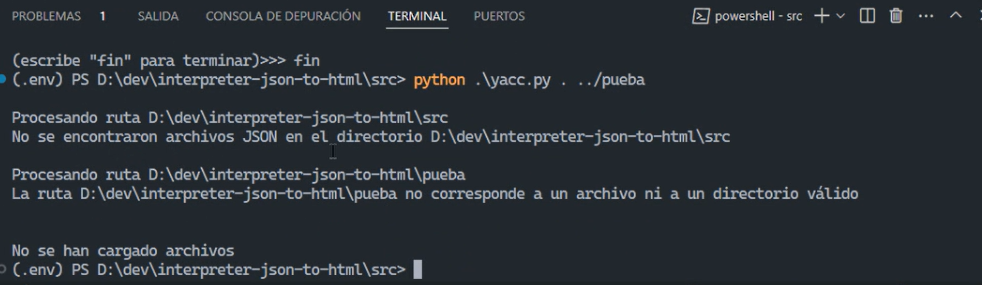
Ejecución del Lexer por terminal: La ejecución del lexer es a través de un archivo lex.py que devuelve un determinado imput o prompt en el que se pueden escribir sentencias que son analizadas y clasificadas si son tokens; sino muestra por pantalla los errores.



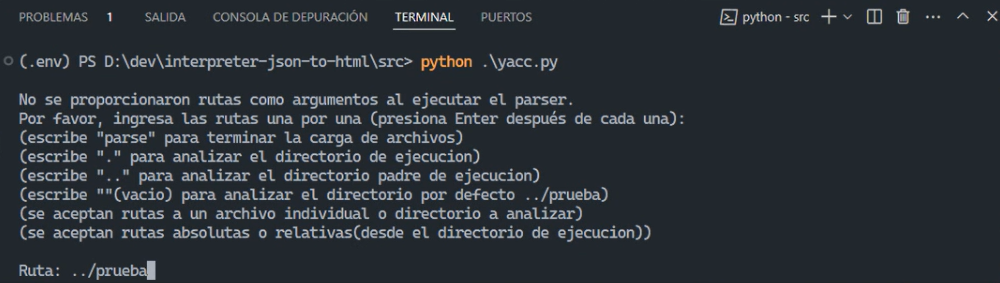


Ejecución del parser por terminal: tiene dos formas de ejecutarse:

1. La primera es a través de la ejecución del programa pasándole como argumento la ruta de los archivos o directorios a analizar;



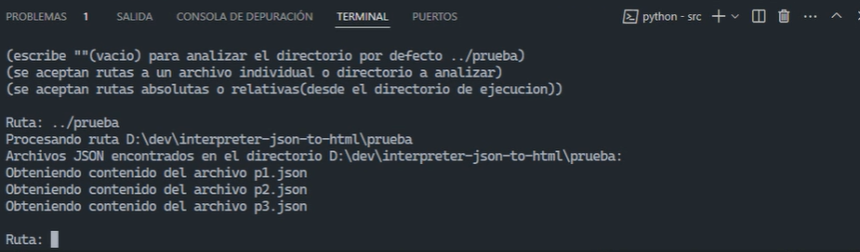
1. La segunda es, sin pasar argumentos, se despliega una interfaz a través de la cual se le puede pasar las rutas a analizar una por una.

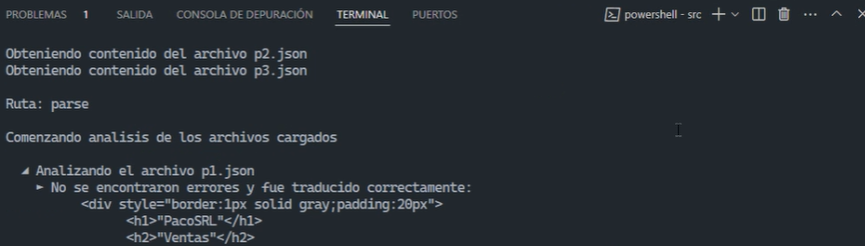


Las rutas que acepta el parser son rutas absolutas o relativas, que pueden ser a un archivo directo.

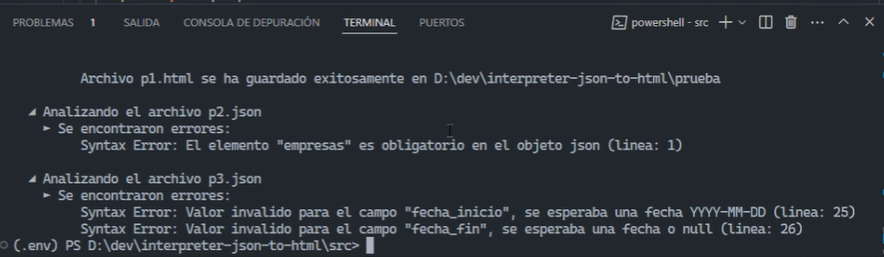
Se analiza el tipo de archivo y si es .json, se lo carga al programa.

La otra opción es que sea una ruta a un directorio, en la que se buscan todos los archivos de tipo .json en la primera capa, y luego se escribe parse para comenzar el análisis.





Se analiza archivo por archivo y si no tienen errores muestra por pantalla la traducción y guarda el archivo en el mismo dirección del archivo que se analizó. En caso de haber errores, se informa el tipo de error.



### **Puntos fuertes y débiles del programa:**

#### **Puntos fuertes:**

* Interfaz amigable para el usuario e intuitiva.
* El programa es rápido.
* Hasta donde se testeó, contempla todos los casos.

#### **Puntos débiles:**

* Ciertos errores no se contemplan y las posiciones de los errores no están del todo correctas y no siempre funcionan.

#### **Registros Extra del día:**

Finalmente, se idearon un nombre y un logo para el programa:

**Nombre:** Syntax Sculptor Lex (SSL)

**Logo:**



# **Link al video presentación del Lexer-Parser:**

Link: <https://youtu.be/5GQvwKy5Ib8>

# **CONCLUSIONES**

Como conclusión y reflexión final podemos decir que el programa **Syntax Sculptor Lex (SSL)**, en general, funciona y cumple con el objetivo del trabajo integrador, que es el de construir los analizadores léxico y sintáctico que permitan analizar, validar y transformar un documento JSON que describe y almacena información de empresas y proyectos.

Hasta donde se ha podido testear, el programa contempla todos los casos, a excepción de algunos casos de errores para los cuales a veces no se indica la posición del error detectado.

La gramática escrita en JSON también funciona correctamente y tiene en cuenta todas las consideraciones presentadas en el documento del TPI.

A mayores, se intentó cumplir con todas y cada una de los requerimientos del proyecto presentados en el documento del TPI, solucionando los diferentes problemas que se nos eran presentados a lo largo del desarrollo de cada una de las fases del trabajo (gramática, lexer y paser) por medio de la investigación, la consulta a profesores y compañeros, y del esfuerzo de cada uno de los integrantes del grupo.

Por último, para la documentación y el video promocional del proyecto se siguió la estructura presentada en el informe técnico en el apartado de **documentación del trabajo** del documento **consigna del TPI**, la cual utilizamos como guía para la realización de esta documentación y le agregamos algunas secciones extra como la de **registros extra** para cada día. La cual nos sirvió para colocar otras tareas, ideas y detalles extra que iban surgiendo durante el desarrollo del TPI pero que no eran lo suficientemente significativas como para ocupar una sección individual.

# **BIBLIOGRAFÍA**

<https://www.dabeaz.com/ply/ply.html#ply_nn29>

<https://stackoverflow.com>