Universidad
Carlos III de Madrid

PRÁCTICA 1: Introducción al entorno Reconocedores léxico y sintáctico

Procesadores de Lenguaje
Curso 2023-2024

Eduardo González Fernandez - NIA: 100451296
Nicolás de la Peña Serrano - NIA: 100383263

ÍNDICE

1. Introducción

En esta práctica, hemos trabajado los aspectos básicos del análisis léxico y sintáctico mediante el uso de la librería PLY. Para abordar estas cuestiones, hemos desarrollado un analizador de formato archivos AJSON. El programa recibe la ruta al archivo y, en caso de tener el formato correcto, extrae las tuplas *clave:valor*. Consta de dos partes: un analizador léxico y uno sintáctico. A continuación, profundizaremos en los aspectos de diseño e implementación.

2. Análisis léxico

Para implementar el analizador léxico, en primer lugar hemos definido la lista de tokens según los tipos de caracteres especificados en el enunciado. Esta lista se puede consultar en su totalidad en la variable 'tokens', que está definida al inicio del archivo 'ajson_lexer.py'. De entre todas las alternativas propuestas en la teoría para identificar tokens, hemos considerado que las expresiones regulares son las más apropiadas debido a su simplicidad y su gran integración dentro de Python. Es importante destacar el orden de definición de las funciones de tokenización, ya que determina qué expresiones se consultarán primero para determinar el token correspondiente a la cadena que se está evaluando. Después de realizar pruebas, decidimos declarar en primer lugar aquellos tokens que tienen una mayor combinación de posibles caracteres. A continuación, comentaremos las expresiones regulares utilizadas, omitiendo aquellas que sean triviales, como las palabras reservadas (booleanos, None, símbolos de comparación y delimitadores), las cuales buscan literalmente los caracteres especificados en el enunciado.

- SCIENTIFIC: `(\d+(\.\d*)?|\.\d+)[eE][+-]?\d+`
    - Esta regex será primera que el analizador consultará, busca números enteros, flotantes con parte entera o flotantes sin parte entera, seguido de una e (mayúscula o minúscula), un símbolo de signo de manera opcionales, seguido de un entero de al menos un dígito.
- BINARY, OCTAL, HEX: Las regex's utilizadas para este tipo de tokens son bastante similares, buscan los prefijos indicados en el enunciado, seguido de, al menos, un dígito o carácter de los permitidos para su formato.

- INTEGER Y FLOAT: Si no se encontró coincidencia con algunos de los tokens anteriores, se procederá a comprobar si es un entero o flotante “aislado” (fuera de cualquiera de los otros formatos descritos). De esta manera evitamos que, por ejemplo: *1.1E5*, sea reconocido como tokens INTEGER o FLOAT de manera errónea. Sus regex’s son triviales, se pueden consultar en sus respectivas funciones.
- STRING: `(?!TR\b|tr\b|FL\b|fl\b|NULL\b|null\b)[_a-zA-Z][_a-zA-Z0-9]*\b`
    - La regex de los strings en primer lugar verifica que no la cadena no sea literalmente alguna de las palabras reservadas. Posteriormente busca combinaciones de cadenas que comiencen por _ o alguna letra, seguido de cualquier combinación de _, letras o números.
- QUOTED_STRING: `"([^"\n]*)?"`
    - Permite cualquier cadena de caracteres que esté delimitada por comillas dobles, sin contener saltos de línea o comillas dobles en su interior, permitiendo además la cadena vacía.
- CARACTERES IGNORADOS: Por último, los saltos línea, espacios y tabulaciones, serán ignorados.

3. Análisis sintáctico

Hemos construido un analizador sintáctico con PLY Yacc, el cual utiliza reglas gramaticales para interpretar diferentes secuencias de tokens (anteriormente definidos en el análisis léxico) y construir con éstos una estructura basada en la gramática definida.
A continuación, se van a explicar las reglas sintácticas que definen cómo se construyen los elementos de un archivo AJON:

- Regla p_object: Define un objeto AJSON como una secuencia de pares clave-valor encerrados entre llaves {}. Si el objeto está vacío, se asigna el valor "vacio":
    - object -> { pairs } | {}
- Regla p_array: Define un vector AJSON como una secuencia de objetos encerrados entre corchetes []:
    - array -> [objects]
- Regla p_objects: Define una secuencia de objetos AJSON, que pueden estar separados por comas o ser un único objeto:
    - objects -> object , object | object

- Regla p_pairs: Define una secuencia de pares clave-valor, que pueden estar separados por comas o ser un único par:
    - pairs -> pair , pairs | pair
- Regla p_pair: Define un par clave-valor, donde la clave es una cadena y el valor puede ser una comparación, un arreglo o cualquier otro valor AJSON:
    - pair -> key : value | key : comparation | key : array
- Regla p_key: Define la clave de un par clave-valor como una cadena de texto con o sin comillas:
    - key -> QUOTED_STRING | STRING
- Regla p_comparation: Define una comparación entre números utilizando operadores (==, >, >=, <, <=):
    - comparation -> number comp number
- Regla p_number: Define un número, que puede ser un entero, decimal, hexadecimal, científico, octal o binario:
    - number -> INTEGER | FLOAT | HEX | SCIENTIFIC | OCTAL | BINARY
- Regla p_comp: Define los operadores de comparación utilizados en la regla p_comparation:
    - comp -> EQUAL | GRATER_EQUAL | LOWER_EQUAL | GRATER | LOWER
- Regla p_value: Define un valor AJSON, que puede ser una cadena, número, nulo, verdadero, falso o un objeto:
    - value -> QUOTED_STRING | INTEGER | FLOAT | HEX | SCIENTIFIC | OCTAL | BINARY | NULL | TRUE | FALSE | object
- Regla p_error: Define la acción a realizar en caso de un error sintáctico, imprimiendo un mensaje de error.

4. Ejecución y test

Para poner en marcha el proyecto, hemos desarrollado el archivo *main.py*, en él se procesan los argumentos pasados por línea de comandos, se intenta leer el archivo de entrada y por último se aplica el analizador sobre la cadena leída del archivo. Cabe destacar la función *print_formatted*, imprime la tuplas en el formato indicado, usa recursión dado que la profundidad del AJSON no es conocida previamente, se llama a sí misma hasta encontrar un value que no sea un iterable. Si encuentra una lista dentro de otra, significa que ha encontrado un array de objetos y debe añadir el índice el objeto a la clave.

Una vez completado el main, desarrollamos distintos casos de test, probando los casos límite e intentando representar todos los casos especiales indicados en el enunciado. Todos los casos de test pueden ejecutarse mediante el archivo *run_tests.sh* siguiendo las instrucciones indicadas en el *README.md*. A continuación haremos una breve descripción de los distintos casos propuestos:

- test1: AJSON simple, probamos los distintos tipos de clave, formatos numéricos base y comparación.
- test2: Probamos, todos los formatos númericos, objetos andidados y arrays de objetos, además de comparaciones entre todos los formatos númericos.
- test3: Caso AJSON vacío.
- test4: Caso dedicado exclusivamente a comparaciones, con el fin de aislar estos tokens.
- test5: Probamos más objetos anidados y arrays más largos.
- test6: Dedicado a todos los casos especiales dentro de los números.

5. Conclusión

Tras la realización del trabajo, se ha logrado una comprensión profunda de los procesos de análisis léxico y sintáctico, fundamentales para la interpretación y procesamiento de lenguajes. Mediante la implementación de un analizador para archivos AJSON, se ha puesto en práctica la teoría estudiada, utilizando la librería PLY para desarrollar tanto el analizador léxico como el sintáctico. El analizador léxico se ha diseñado con una serie de expresiones regulares para identificar los tokens necesarios, mientras que el analizador sintáctico se ha realizado utilizando reglas gramaticales que permiten formar la estructura de datos correspondiente a un archivo AJSON. Este enfoque ha demostrado ser efectivo para manejar la complejidad de los archivos AJSON y extraer las tuplas clave-valor de manera correcta. El trabajo ha demostrado que, con una comprensión clara de los conceptos y una buena implementación, es posible construir herramientas de análisis de lenguajes que sean robustas y eficientes. La experiencia adquirida en esta práctica proporciona una base sólida para futuras exploraciones en el campo de los compiladores y los intérpretes, entre otros.