Universidad Carlos III

Curso Procesadores del Lenguaje 2023-24

MEMORIA PRÁCTICA 1

TITULACIÓN:	**INGENIERÍA INFORMÁTICA**	**Grupo**	**80**
Alumno/a:	**Rubén De Arriba Viejo**	**NIA:**	**100471975**
Alumno/a:	**Ana María Ortega Mateo**	**NIA:**	**100472023**

Introducción

Este documento presenta los razonamientos y pasos seguidos para implementar el analizador léxico y sintáctico requerido en el caso práctico 1, enfocado en el formato AJSON. Nuestro objetivo es desarrollar reconocedores que interpreten la estructura de un objeto AJSON y estén capacitados para reconocer los tokens específicos definidos por dicho formato. Para ello, hemos tenido que utilizar la biblioteca PLY.

1. Analizador Léxico

- **Descripción.**

Este analizador se ha diseñado para que pueda reconocer y clasificar los diferentes tokens del formato AJSON. Los tokens que nuestro analizador es capaz de detectar y por lo tanto los que han sido creados, son los siguientes:

1. Números (comprendiendo todos los formatos indicados en la práctica).
2. Cadenas de carácteres con comillas.(Cualquier cadena menos /n,/t y “”).
3. Cadenas de carácteres sin comillas.(Mayúsculas, minúsculas, _ y números)
4. Booleanos (TR y FL).
5. Valor nulo (NULL).
6. Operadores de comparación (<, >, <= , >= , ==).
7. Delimitadores ({ , }, “,” , [,] , :).

Los números pueden ser representados y reconocidos de diversas formas, permitiendo variaciones como números negativos o la inclusión de letras en mayúsculas o minúsculas, como en el caso de los números binarios (por ejemplo, 0b001 o 0B001). Los valores con comillas y sin comillas tienen diferentes formatos, que se han tenido en cuenta.

- **Implementación.**

El analizador léxico se implementa utilizando la clase LexerClass() del fichero *ajson_lexer.py*, en el que se inicializa el analizador léxico y se definen todas las expresiones regulares para cada tipo de token anteriormente definido. Tanto los tokens de números como las cadenas de caracteres sin comillas se definen en métodos, para poder pasar de cadenas de texto a números (int, float, decimal…), y así poder compararlos posteriormente. En cuanto a las cadenas sin comillas, se deben diferenciar de las palabras reservadas.Los tokens de palabras reservadas para el analizador son "TR", "FL" y "NULL", los cuales se distinguen de las cadenas de caracteres y se tratan como tokens reservados.

Los números están organizados por orden de prioridad en nuestro analizador, de modo que este considera las diferentes posibilidades de representación antes de asignar un tipo de conversión. Esto evita, por ejemplo, que el analizador detecte un número y lo asigne directamente como entero, sin tener en cuenta las diversas formas en las que podría estar representado.

En cuanto a los saltos de línea como tabulaciones y espacios son ignorados. Para el control de errores, se ha tenido en cuenta **el número de línea en el que se encuentra el error**, lo cual facilita la identificación y corrección de problemas durante el análisis.

- **Pruebas.**

Para las pruebas del analizador léxico no se ha tenido en cuenta que siga la estructura correcta de un AJSON, sino que, como lo que queremos evaluar son los tokens, simplemente se han realizado varias pruebas para ver si el analizador detecta correctamente los tokens del formato y los clasifica bien. Estas pruebas se encuentran en el directorio *lexer_test/*. Se ha creado un test de errores para comprobar que si se detecta un token no definido saltará el error. La estructura de test es la siguiente:

- test 1: Números, se prueban todos los formatos posibles de los números. El resultado es correcto, detectando todos los formatos y dando el valor correspondiente a cada uno.
- test 2: strings con comillas y strings sin comillas. El resultado es correcto, detectando los dos tipos de cadenas.
- test 3: Palabras reservadas: TR, FR, NULL, delimitadores y operadores. El resultado es correcto, detectando todas las palabras reservadas y operadores independientemente de su posición en el fichero.
- test 4: De errores, str con comillas sin saltos y sin comillas dentro y tokens no reconocidos. El resultado es el esperado, mostrando los caracteres que no son válidos e identificando de forma incorrecta debido a problemas en el formato.

2. Analizador sintáctico

- **Descripción.**

El analizador sintáctico está diseñado para procesar y comprobar la estructuración del texto según las reglas definidas en la gramática AJSON. Para lograr esto, el analizador sintáctico utiliza reglas gramaticales para descomponer el texto en elementos como claves, valores y objetos anidados, verificando que estén correctamente formados y organizados. Cualquier mal composición del fichero de entrada, el analizador sintáctico lo detectará, es decir, cualquier estructura que no cumpla con la gramática que se ha definido. En caso de que no siga la estructura, se producirá un error sintáctico.

- **Gramática definida.**

Antes de implementar el analizador sintáctico, fue necesario definir una gramática sin ambigüedades y con reglas de producción claras. Las reglas de producción se diseñaron de manera que reflejen fielmente la estructura sintáctica del formato, permitiendo un análisis preciso del texto de entrada. Las palabras en minúsculas representan los símbolos no terminales y las mayúsculas los símbolos terminales.

ajson→ LBRACKET contents RBRACKET | LBRACKET RBRACKET
contents → key DOS_PUNTOS value opcional | key DOS_PUNTOS value COMA contents
opcional → COMA | λ
key → STR_CON_COMILLAS | STR_SIN_COMILLAS
value → ajson | operation | NUMBER | other | arrayobjects
arrayobjects → LCORCHETE valores RCORCHETE| LCORCHETE RCORCHETE
valores → ajson opcional | ajson COMA valores
operation → menor | mayor | igual | mayigual | menigual
menor → NUMBER MENOR NUMBER
mayor → NUMBER MAYOR NUMBER
igual → NUMBER IGUAL NUMBER
mayigual → NUMBER MAYIGUAL NUMBER
menigual → NUMBER MENIGUAL NUMBER
other → TR | FL | NULL | STR_CON_COMILLAS

Una vez establecida la gramática, se procedió a implementar el analizador sintáctico utilizando las reglas de producción definidas.

- **Transformación a gramática reconocida por LL(1)**

Para hacer más sencillo este ejercicio, hemos decidido modificar la gramática generada para la práctica. Esto se debe a que, desarrollando la definida en la práctica, se llegaba a inconsistencias en las tablas. La gramática definida es:

ajson→ LBRACKET contents RBRACKET | LBRACKET RBRACKET
contents → key DOS_PUNTOS value | key DOS_PUNTOS value COMA contents | key DOS_PUNTOS value COMA
key → STR_CON_COMILLAS | STR_SIN_COMILLAS
value → ajson | operation | NUMBER | other | arrayobjects
arrayobjects → LCORCHETE valores RCORCHETE| LCORCHETE RCORCHETE
valores → ajson | ajson COMA valores | ajson COMA
operation → menor | mayor | igual | mayigual | menigual
menor → NUMBER MENOR NUMBER
mayor → NUMBER MAYOR NUMBER
igual → NUMBER IGUAL NUMBER
mayigual → NUMBER MAYIGUAL NUMBER
menigual → NUMBER MENIGUAL NUMBER
other → TR | FL | NULL | STR_CON_COMILLAS

La gramática definida para la práctica no contiene ninguna recursividad a izquierdas, por lo que se va a factorizar directamente, lo que genera la siguiente gramática:

ajson → LBRACKET ajson'
ajson' → contents RBRACKET | RBRACKET
contents → key DOS_PUNTOS value contents'

contents’ → COMA contents’’ | λ
contents’’ → contents | λ
key → STR_CON_COMILLAS | STR_SIN_COMILLAS
value → ajson | NUMBER operation | other | arrayobjects
arrayobjects → LCORCHETE arrayobjects’
arrayobjects’ → valores RCORCHETE| RCORCHETE
valores → ajson valores’
valores’ → COMA valores’’ | λ
valores’’ → valores | λ
operation → menor | mayor | igual | mayigual | menigual | λ
menor → MENOR NUMBER
mayor → MAYOR NUMBER
igual → IGUAL NUMBER
mayigual → MAYIGUAL NUMBER
menigual → MENIGUAL NUMBER
other → TR | FL | NULL | STR_CON_COMILLAS

Con la gramática ya factorizada, identificamos los conjuntos primero y siguiente de cada símbolo no terminal:

	PRIMERO	SIGUIENTE
ajson	LBRACKET	COMA, RBRACKET
ajson’	STR_CON_COMILLAS, STR_SIN_COMILLAS, RBRACKET	COMA, RBRACKET
contents	STR_CON_COMILLAS, STR_SIN_COMILLAS	RBRACKET
contents’	COMA	RBRACKET
contents’’	STR_CON_COMILLAS, STR_SIN_COMILLAS	RBRACKET
key	STR_CON_COMILLAS, STR_SIN_COMILLAS	DOS_PUNTOS
value	NUMBER, LBRACKET, TR, FL, NULL, STR_CON_COMILLAS, LCORCHETE	COMA, RBRACKET
arrayobjects	LCORCHETE	COMA, RBRACKET
arrayobjects’	RCORCHETE, LBRACKET	COMA, RBRACKET
valores	LBRACKET	RCORCHETE

valores’	COMA	RCORCHETE
valores’’	LBRACKET	RCORCHETE
operation	MENOR, MAYOR, IGUAL, MENIGUAL, MAYIGUAL	COMA
menor	MENOR	COMA
mayor	MAYOR	COMA
igual	IGUAL	COMA
mayigual	MAYIGUAL	COMA
menigual	MENIGUAL	COMA
other	TR, FL, NULL, STR_CON_COMILLAS	COMA, RBRACKET

Teniendo ya la tabla de conjuntos primeros y segundos, podemos generar la tabla de parsing. Esta tabla se ha decidido hacer en una hoja de cálculo por insuficiencia de espacio en un documento de texto. De esta forma, la tabla ha quedado:

	NUMBER	IGUAL	MAYIGUAL	MENIGUAL	MAYOR	MENOR	LBRACKET	RBRACKET	DOS_PUNTOS	COMA
ajson							ajson → LBRACKET ajson'			
ajson'								ajson' → RBRACKET		
contents										
contents'								contents' → λ		contents' → COMA contents"
contents"								contents" → λ		
key										
value	value → NUMBER operation						value → ajson			
arrayobjects										
arrayobjects'							arrayobjects' → valores RCORCHETE			
valores							valores → ajson valores'			
valores'										valores' → COMA valores"
valores"							valores" → valores			
operation		operation → igual	operation → mayigual	operation → menigual	operation → mayor	operation → menor				operation → λ
menor						menor → MENOR NUMBER				
mayor					mayor → MAYOR NUMBER					
igual		igual → IGUAL NUMBER								
mayigual			mayigual → MAYIGUAL NUMBER							
menigual				menigual → MENIGUAL NUMBER						
other										

STR_CON_COMILLAS	STR_SIN_COMILLAS	LCORCHETE	RCORCHETE	TR	FL	NULL	$
ajson' → contents RBRACKET	ajson' → contents RBRACKET						
contents → key DOS_PUNTOS value contents'	contents → key DOS_PUNTOS value contents'						
contents" → value	contents" → value						
key → STR_CON_COMILLAS	key → STR_SIN_COMILLAS						
value → other		value → arrayobjects		value → other	value → other	value → other	
		arrayobjects → LCORCHETE arrayobjects'					
			arrayobjects' → RCORCHETE				
			valores' → λ				
			valores" → λ				
other → STR_CON_COMILLAS				other → TR	other → FL	other → NULL	

- **Implementación.**

En la clase Parser, se inicializa y se define la estructura y las reglas gramaticales para poder interpretar el formato AJSON. El analizador sintáctico tiene conocimiento de todos los tokens definidos en LexerClass, lo que le permite reconocerlos como parte del formato que está analizando.

El analizador sintáctico se inicia con el axioma de la gramática, que en este caso es la regla p_ajson, puede estar o bien vacío o contener los distintos elementos que la estructura de la gramática permite. Por ello, se crean todos los objetos parser que componen la gramática anteriormente definida, con la misma estructura.

A medida que se analiza el archivo de entrada, se van construyendo y actualizando las estructuras de datos utilizando p[0], que permite almacenar el resultado de cada regla de producción y, gradualmente, construir la estructura de datos deseada. Esto permite representar fielmente la estructura de los datos del archivo y poder imprimirla. Se han utilizado algunas condiciones para posteriormente poder representarlas correctamente en la impresión, si por ejemplo detecta un TR p[0] será igual a True o si se cumple o incumple una condición booleana se guardará en p[0] y guardará True o False dependiendo de la condición. P[0] es una manera de ir registrando toda la estructura y poder manipularla posteriormente.

Para poder imprimirla, añadimos p[0] en una lista y se va representando el contenido según el formato pedido. Posteriormente, se accederá a esta lista mediante llamadas recursivas para ir imprimiendo el formato. Se ha hecho de esta forma ya que, si se hubiera impreso el resultado durante la ejecución del parser, se hubiesen impreso primero las últimas líneas del fichero que contiene la estructura ajson.

- **Pruebas.**

Para las pruebas del analizador sintáctico, se han utilizado distintos test para comprobar que se imprime de forma correcta y poder comprobar que la gramática implementada está bien estructurada. Estos test se encuentran en el directorio *parser_test/*. Estos test también se podrían utilizar para comprobar el analizador léxico, pero no es su verdadera finalidad.

- test 1: Comparaciones entre números, comprobar que devuelve el valor booleano correcto. El resultado de todas las comprobaciones es correcto independientemente del tipo de números que se comparen.
- test 2: Un formato AJSON genérico, con las palabras reservadas, cadenas de caracteres, clave vacía y clave anidada. El resultado es correcto, asignando de forma correcta los valores de cada clave y valor.
- test 3: Comprobar la correcta detección de claves anidadas e impresión. El resultado es correcto, imprimiendo de forma correcta cada una de las claves anidadas.
- test 4: Comprobar los arrays de objetos. El resultado es correcto, imprimiendo los índices en el orden correspondiente.

- test 5: Fichero vacío. Se imprime correctamente el mensaje correcto.
- test 6: Comprobar combinación de arrays de objetos y valores anidados. Se combinan tanto claves anidadas como arrays de objetos para comprobar su impresión, dando un resultado correcto.

3. Guía de ejecución

Para la ejecución del proyecto se han propuesto dos formas de ejecución, una para el analizador léxico y otra para el sintáctico. De esta forma se puede aislar de una mejor manera las dos partes principales del analizador y ver cual de ellas es la que provoca fallos.

Para la ejecución del analizador se debe introducir el comando:

```
python ./PL_P1_DeArriba_Ortega/main.py {InputFile} {Mode}
```

El primer parámetro de entrada es el fichero que se desea analizar. El segundo parámetro de entrada es el analizador que se desea ejecutar: 0 indica que se desea ejecutar el analizador léxico, 1 indica que se desea ejecutar el analizador sintáctico. Si se omite este segundo parámetro, se ejecutará por defecto el analizador sintáctico.