

Práctica 1

Introducción PDL



Procesadores del lenguaje

10 de Marzo de 2024

Grado en Ingeniería Informática

Nathalia Moniz Cordova

Celia Patricio Ferrer

Índice

Introducción	3
Reconocedor léxico	4
Palabras reservadas, tokens y reserved_map	4
Init	5
Expresiones regulares	5
Error y validación	6
Reconocedor sintáctico	6
Pruebas	8
Caso 1	8
Caso 2	8
Caso 3	9
Caso 4	10
Conclusiones	12

Introducción

Este trabajo práctico consiste en la construcción de reconocedores léxico y sintáctico para el formato de texto AJSON (*Almost a JavaScript Notation*). Este formato se asemeja al conocido JSON, pero con algunas leves diferencias. Para llevar a cabo este proyecto, se emplea el lenguaje de programación Python, haciendo uso de la librería PLY. Esta librería contiene *lex* y *yacc*, herramientas para la construcción de dichos analizadores del lenguaje. Estas herramientas permiten construir y definir la gramática y las reglas para reconocer el lenguaje.

El propósito principal de este documento es detallar el proceso de construcción del reconocedor léxico y sintáctico para el formato AJSON. Con el fin de ofrecer una documentación clara y completa de la implementación de dichos reconocedores para facilitar la comprensión del trabajo realizado. La estructura que sigue este documento es la siguiente:

- El reconocedor léxico.
- El reconocedor sintáctico.
- Las pruebas realizadas.
- Conclusiones.

Reconocedor léxico

En esta primera sección se detallan los aspectos a destacar del reconocedor léxico implementado para analizar el formato AJSON. Para realizar este analizador, se ha decidido realizarlo en una clase a la que hemos llamado *LexerClass*. A continuación, se describen los elementos que tiene esta clase: las palabras reservadas, los tokens y las expresiones regulares para identificar los diferentes tipos de elementos léxicos de este lenguaje.

Palabras reservadas, tokens y reserved_map

Tanto palabras reservadas como tokens se han realizado en tuplas por su fácil manipulación y mantenimiento, teniendo una estructura clara y ordenada sin permitir que estas se puedan modificar por error.

En primer lugar, se encuentran definidas las palabras reservadas o *reserved*. Se trata de una tupla en la que se definen ciertas palabras que tienen un significado específico dentro del lenguaje y no pueden ser utilizadas como identificadores o como nombres de variables. Aquí se encuentra: *NULL* que representa el valor nulo, *TR* que representa el valor booleano *True* y *FL* que representa el otro valor booleano *False*. Son palabras que van a tener un tratamiento especial dentro de la gramática, por lo que se han definido como tokens diferentes, para facilitar reconocerlas.

En segundo lugar, se definen los *tokens*. Se trata de otra tupla en la que se definen todas las palabras que este analizador va a reconocer durante el proceso de reconocimiento de un texto de entrada. Aquí se encuentran definidos los diferentes tipos de números, los operadores de comparación posibles, los dos tipos de cadenas de caracteres y los separadores que forman parte de este lenguaje. Además, se incluye la tupla de palabras reservadas. Los tokens son lo más importante para construir un analizador léxico sólido y robusto.

Y por último, se ha implementado la tupla denominada *reserved*. Esta tupla contiene las palabras reservadas de nuestro lenguaje y que por ende no pueden ser usadas como nombres de variables. Asimismo, también contamos con el diccionario *reserved_map* el cual utilizamos para mapear cada palabra reservada a su equivalente en minúscula y mayúscula de manera que si se escribe por ejemplo “true”, sea reconocida por el lenguaje.

Y por último, se ha implementado un diccionario denominado *reserved_map*. Este diccionario asocia las palabras reservadas con sus respectivos tokens. Esto facilita su identificación y reconocimiento durante el proceso de reconocimiento al realizar el análisis léxico.

Init

Método `__init__` constructor que inicializa el analizador léxico. Además se asigna el diccionario `reserved_map`, el cual mapea las palabras reservadas con sus respectivos tokens.

A continuación, se define qué caracteres se van a ignorar (`t_ignore`). Estos van a ser saltos de línea, tabulaciones y espacios durante el análisis, debido a que no tiene ningún significado semántico específico en este lenguaje. El analizador va a pasar por alto estos caracteres y no los va a considerar como tokens válidos para este proceso.

Expresiones regulares

Continuando con la implementación, aquí se definen las diferentes expresiones regulares que este reconocedor tiene. En primer lugar, las expresiones regulares de las comparaciones. Estas son sencillas porque para escribirlas se utilizan los símbolos que comúnmente utilizamos. Otras que también se encuentran definidas son los delimitadores de llaves (de apertura y de cierre) y la coma para separar las asignaciones. Y por último, los dos puntos para indicar la asignación de una clave con su valor correspondiente.

Continuando con la definición de las expresiones regulares, se encuentran la de todos los números que este lenguaje reconoce. En primer lugar, los números reales (`t_REAL`), es decir, un signo negativo que puede o no estar, seguido de un dígito opcional, un punto y un dígito obligatorio. La siguiente expresión definida son los números en notación científica (`t_NCIENT`). Estos están compuestos por un guión opcional para indicar si es negativo o positivo, mínimo un dígito, seguido de la letra “e” en mayúscula o minúscula, otro guión opcional y mínimo otro dígito. Estos números se convierten al número real que representan.

La siguiente expresión regular que se encuentra es la de los número binarios (`t_BIN`), representados con un “0b” o un “0B” nada más empezar y seguido de, mínimo un “0” o “1”. Estos son convertidos al número entero que corresponda. Los siguientes en ser definidos son los números en sistema de numeración octal (`t_OCT`). La definición de estos comienza con un 0, seguido de, mínimo un número entre el 0 y el 7. Y también son convertidos al número entero que corresponde.

A continuación, se definen los números en hexadecimal (`t_HEX`). Estos se representan comenzando con “0x” o “0X” y seguidos de mínimo un carácter entre el “0” al “9” y entre la “A” y la “F”. Al igual que los anteriores, se convierten al número entero equivalente. Y por último, los números enteros que pueden ser negativos o no.

Lo siguiente que se define, son las cadenas de caracteres que pueden ser con comillas dobles (*t_CADENACON*) o sin comillas (*t_CADENASIN*). Las cadenas con comillas se representan con comillas dobles al principio y al final, y continene cualquier caracter excepto las comillas dobles y los saltos de línea. Además, su valor se transforma a lo que está contenido entre estas comillas. Por el contrario, las cadenas sin comillas, se han representado que puedan empezar con cualquier letra mayúscula o minúscula e incluso, con un guión bajo (“_”), y seguido opcionalmente de cualquier letra mayúscula o minúscula, número o guión bajo.

Error y validación

Finalmente, se han incluido tres últimos métodos destinados a la depuración y a las pruebas de este reconocedor léxico. Estos métodos sirven para verificar el funcionamiento del analizador y facilitan la identificación de los posibles errores durante el proceso.

En primer lugar, el método de error (*t_error*) se encarga de manejar los errores léxicos encontrados durante el proceso de reconocimiento del texto de la entrada. Cuando el analizador encuentra un error, concretamente un caracter ilegal, este método imprime por pantalla el mensaje correspondiente y continúa con el siguiente caracter.

En segundo lugar, se encuentra el método para realizar pruebas dado un texto de entrada (*test*). Este método se encarga de inicializar el analizador con dicha entrada. Por último, itera sobre los tokens generados e imprime el tipo y el valor de cada uno.

Mientras que el último método permite realizar pruebas dado un fichero de texto como entrada (*test_with_files*). Este método lee el archivo especificado como ruta y luego ejecuta cada línea con el método anterior.

Reconocedor sintáctico

En cuanto al análisis sintáctico, hemos decidido también implementarlo en una clase para un manejo más fácil importando los tokens del analizador léxico. Para el inicializador de esta clase, además de definir el *lexer* y el *parser*, también hemos definido *contenido*, el cual usaremos para introducir la salida del analizador sintáctico y que utilizaremos para imprimir el resultado esperado. Además, en esta clase también procedimos a definir las reglas de producción de nuestra gramática que son esenciales para determinar la estructura de las sentencias que nuestro lenguaje puede aceptar. Estas reglas de producción, junto con los tokens definidos en el analizador léxico, forman la base de nuestro lenguaje de programación.

En primer lugar, contamos con la función **p_axioma** que representa las reglas de producción del axioma de la gramática. **LLAVEA contenido LLAVEC** refleja el caso

en el que el documento AJJSON se haya rellenado por lo tanto su estructura se compone de una llave de apertura, el contenido del documento y una llave de cierre y **LLAVEA LLAVEC** refleja el caso de un archivo AJJSON vacío. En esta misma función también pasamos a asignar el resultado de la regla en **p[0]** dependiendo del caso y finalmente lo guardamos en *contenido*.

Por otra parte, nos encontramos con la función **p_contenido** cuyas reglas de producción indican que el documento puede estar compuesto de: una única asignación, con coma o sin coma puesto que la coma de finalización es opcional y más de una asignación, y una asignación seguida una o más elementos. Esta última regla contiene una recursividad a derechas precisamente porque nuestro archivo AJJSON puede ser de una longitud arbitraria, de esta manera estamos teniendo en cuenta un número indefinido de asignaciones, asimismo, debido a que estamos tratando con la herramienta *yacc* que utiliza un análisis sintáctico ascendente, nos conviene hacer uso de recursividad a derechas para no caer en ciclos infinitos durante el análisis.

En nuestro analizador sintáctico, también contamos con reglas de producción para las asignaciones (**p_asignación**) que pueden estar compuestas por una cadena con comillas o sin comillas a la cual se le asigna un valor. Luego pasamos a los diferentes valores que pueden ser asignados a dichas claves (**p_valor**), estos valores pueden ser números, comparaciones, palabras reservadas (NULL, TR, FL), una cadena (delimitada por comillas dobles) y un AJJSON anidado, por esta razón para este último caso llamamos al axioma para que se pueda realizar el análisis sobre ese nuevo AJJSON que estamos asignando.

También, como hemos definido tokens para cada tipo de número y no un solo token que representa todos los números, tuvimos que crear reglas para transformar el símbolo no terminal **numero** a sus respectivos tipos (entero, real, notación científica, binario, octal y hexadecimal). Finalmente, se encuentra el conjunto de reglas **p_comparacion** que simplemente define los distintos tipos de comparaciones que acepta el lenguaje y dependiendo de qué comparación se esté usando se comparan los valores y el resultado (True o False) se almacena en **p[0]**.

Otra función a destacar en la clase Parser, es la función *imprimir* que se encarga de tomar el contenido que se fue actualizando durante el análisis y formatearlo para imprimir el archivo con el formato de salida que se pide en el enunciado. De igual manera, cabe destacar que tanto para el *lexer* como para el *parser* se han incluido funciones de error y test para el testeo con diferentes archivos de prueba e impresión errores.

Pruebas

Para probar que nuestro código funciona correctamente hemos llevado a cabo una serie de pruebas con distintos archivos AJJSON que contienen contenidos variados para cubrir la mayor cantidad de casos posibles.

Caso 1

En este primer caso hemos decidido probar la entrada de ejemplo que se nos proporciona en el enunciado y compararlo con la salida también proporcionada.

Entrada	Salida esperada	Salida del programa
<pre>{ clave_vacia: {}, clave_numero: 10, clave_cadena_caracteres: "string", clave_booleano: TR, clave_valor_nulo: NULL, clave_op_comp: 10 > 11, calve_anidada: { "clave con comillas": "", } }</pre>	<pre>FICHERO AJJSON "./test_files/f1.ajson" { clave_numero: 10 } { clave_cadena_caracteres: string } { clave_booleano: True } { clave_valor_nulo: None } { clave_op_comp: False } { calve_anidada.clave con comillas: }</pre>	<pre>FICHERO AJJSON "./test_files/f1.ajson" { clave_numero: 10 } { clave_cadena_caracteres: string } { clave_booleano: True } { clave_valor_nulo: None } { clave_op_comp: False } { calve_anidada.clave con comillas: }</pre>

Caso 2

De igual manera, debemos probar con múltiples claves anidadas y ver que la salida sigue siendo correcta.

Entrada	Salida esperada
<pre>{ clave_vacia: {}, clave_numero: 10, clave_cadena_caracteres: "string", clave_booleano: TR, clave_valor_nulo: NULL, clave_op_comp: 10 > 11, calve_anidada: { "clave_con_comillas_anidada": {</pre>	<pre>FICHERO AJSON "./test_files/f2.ajson" { clave_numero: 10 } { clave_cadena_caracteres: string } { clave_booleano: True } { clave_valor_nulo: None } { clave_op_comp: False } { calve_anidada.clave_con_comillas_anidada.cl ave_anidada_2.clave_anidada_3.clave_4: }</pre>


```

    clave_anidada_2: {
      clave_anidada_3: {
        clave_4 : "",
      },
    },
  },
}

```

Salida del programa

```

FICHERO AJSON "./test_files/f2.ajson"
{ clave_numero: 10 }
{ clave_cadena_caracteres: string }
{ clave_booleano: True }
{ clave_valor_nulo: None }
{ clave_op_comp: False }
{ calve_anidada.clave_con_comillas_anidada.clave_anidada_2.clave_anidada_3.clave_4:  }

```

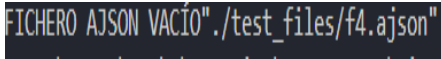
Caso 3

Hemos visto pertinente además probar que las comparaciones funcionan para los distintos tipos de números que acepta el lenguaje. Igualmente, hemos aprovechado en esta prueba insertar claves sin comillas que empiecen por barras bajas y que contengan números, los cuales deben ser también aceptados.

Entrada	Salida esperada	Salida del programa
<pre> { _comparacion_int: 1 >= 35, comparacion_hex_5: 0xFF >= 0b111011, _comparacion_cient: 1e-1 == 0.1, comp_bin: 010 < 110, comparacion_real: .12 > .09, comp: 077 < 63 } </pre>	<pre> FICHERO AJSON "./test_files/f3.ajson" { _comparacion_int: False } { comparacion_hex_5: True } { _comparacion_cient: True } { comp_bin: True } { comparacion_real: True } { comp: False } </pre>	<pre> FICHERO AJSON "./test_files/f3.ajson" { _comparacion_int: False } { comparacion_hex_5: True } { _comparacion_cient: True } { comp_bin: True } { comparacion_real: True } { comp: False } </pre>

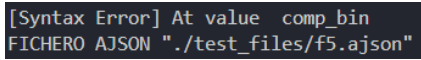
Caso 4

Otro caso importante a probar es el caso de que se nos proporcione un archivo AJSON vacío.

Entrada	Salida esperada	Salida del programa
{ }	FICHERO AJSON VACÍO"./test_files/f4.ajson"	

Caso 5

De igual manera, debemos también probar si nuestro programa nos avisa en caso de cometer errores léxicos o sintácticos. En este ejemplo hemos probado no colocando las comas de separación y revisando que nos advirtiese del error correctamente.

Entrada	Salida esperada	Salida del programa
{ _comparacion_int: 1 >= 35, comparacion_hex_5: 0xFF >= 0b111011, _comparacion_cient: 1e-1 == 0.1 comp_bin: 010 < 110, comparacion_real: .12 > .09, comp: 077 < 63 }	Debería darnos un error de sintaxis en el valor comp_bin puesto que no hemos colocado la coma de separación en el valor anterior.	

Caso 6

Por otra parte, probamos también con errores léxicos, por ejemplo, introduciendo caracteres ilegales en nuestra entrada.

Entrada	Salida esperada	Salida del programa
<pre>{ *clave_vacia: {}, clave_numero: 10, clave_cadena_caracteres: "string", clave_booleano: TR, clave_valor_nulo: NULL, clave_op_comp: 10 > 11, calve_anidada: { "clave con comillas": "", } }</pre>	Debería darnos un error de caracter ilegal en el valor <i>clave_vacia</i> , puesto que contiene un caracter no permitido según la expresión regular.	<pre>Caracter ilegal *clave_vacia: {}, clave_numero: 10, clave_cadena_caracteres: "string", clave_booleano: TR, clave_valor_nulo: NULL, clave_op_comp: 10 > 11, calve_anidada: { "clave con comillas": "", } }</pre>

Conclusiones

Durante el desarrollo de este trabajo práctico hemos desarrollado con éxito un reconocedor léxico y sintáctico para el formato de texto AJSON. Para ello, hemos utilizado el lenguaje de programación Python junto a la librería PLY, destinada a la definición de estos analizadores. Como se ha podido comprobar, se ha conseguido un programa que reconoce y devuelve los errores correspondientes de este lenguaje. Además, cabe destacar que el uso de clases era necesario para esta implementación para facilitar y aportar organización al código.

En cuanto al desarrollo de la implementación del reconocedor léxico, cabe destacar la definición de los tokens junto a la de las palabras reservadas y el uso del `reserved_map`. Esto nos ha permitido que el código fuese más fácil de manejar y de interpretar. Por lo que, esto se ha considerado como una muy buena solución al problema de las palabras reservadas.

Por otro lado, en el desarrollo de la implementación del reconocedor sintáctico, cabe mencionar la gramática con las reglas de producción definidas. Esto ha permitido explicar el proceso del reconocimiento de las sentencias del formato de texto AJSON. Además, del manejo de la recursividad derechas para la posibilidad de realizar infinitas asignaciones y la opción final del separador coma.

Sobre los principales problemas que hemos encontrado durante este proceso, han sido pocos. Hemos agradecido bastante que se hiciese en Python, ya que es un lenguaje que controlamos bastante y eso ha ayudado en esta implementación. Sin embargo, sí hemos tenido algunos problemas a la hora de realizar la impresión por pantalla del contenido del fichero de texto dado. Pero se ha podido solventar fácilmente con un método recursivo llamado por otro.

Adicionalmente, hemos llevado a cabo diversos casos de prueba para comprobar la funcionalidad y la robustez de nuestros analizadores. Estas pruebas han cubierto diversos casos y hemos obtenido resultados satisfactorios. Esto nos ha permitido ver la precisión y eficacia de nuestra implementación.