DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Conversor de JSON a YAML

Teoria de Lenguajes
Primer Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Regnier, Ezequiel	836/13	eze_regnier@hotmail.com
Zamboni, Gianfranco	219/13	gianfranco376@gmail.com
Pesaresi, Natalia	636/14	natalia.pesaresi@gmail.com

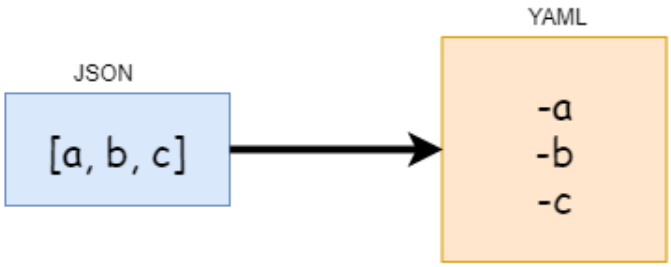
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

En este trabajo, se propone realizar un conversor de código JSON a código YAML. Como ambos lenguajes poseen las mismas estructuras, la conversión solo implica formatear correctamente la cadena de entrada para que su sintaxis sea acorde a la cadena de salida.

Por ejemplo en el caso de los arrays:



Para realizar esta conversión, se necesitará un Lexer que reconozca las cadenas que pertenecen al lenguaje JSON. Y un parser que traduzca sintácticamente la escritura a YAML.

2. Gramática

Para implementar el traductor, se decidio usar la librería ply, una herramienta de python que utilizas grámaticas LR para generarlo. Por lo que la grámatica diseñada, además de generar el lenguaje descripto en la especificación de JSON, debe cumplir con este requisito.

Se propone:

$$G =< \{Value, Object, Array, Members, Pair, Elements\},$$
$$\{\texttt{string}, \texttt{number}, \texttt{true}, \texttt{false}, \texttt{null}, \texttt{[}, \texttt{]}, \{, \}, \texttt{:}\}, Value, P >$$

Donde P es:

(0) $S' \rightarrow Value$

(1) $Value \rightarrow Object$

(2) $Value \rightarrow Array$

(3) $Value \rightarrow \texttt{string}$

(4) $Value \rightarrow \texttt{number}$

(5) $Value \rightarrow \texttt{true}$

(6) $Value \rightarrow \texttt{false}$

(7) $Value \rightarrow \texttt{null}$

(8) $Object \rightarrow \{\ \}$

(9) $Object \rightarrow \{Members\}$

(10) $Members \rightarrow Pair$

(11) $Members \rightarrow Pair\texttt{,}Members$

(12) $Pair \rightarrow \texttt{string :}Value$

(13) $Array \rightarrow \texttt{[ ]}$

(14) $Array \rightarrow \texttt{[}Elements\texttt{]}$

(15) $Elements \rightarrow Value$

(16) $Elements \rightarrow Value\texttt{,}Elements$

Para verificar que efectivamente esta es una grámatica LR, se usó la herramienta *LALR(1) Parser Generator* [-1-] para generar las tablas LR de acción y goto y verificar, de esta forma, que no haya conflictos.

Figura 1: LR Table

State	ACTION												GOTO						
	`string`	`number`	`true`	`false`	`null`	{	}	,	:	[	]	$	S'	$Value$	$Object$	$Members$	$Pair$	$Array$	$Elements$
0	S4	S5	S6	S7	S8	S9				S10				1	2			3	
1												Acc							
2							R1	R1			R1	R1							
3							R2	R2			R2	R2							
4							R3	R3			R3	R3							
5							R4	R4			R4	R4							
6							R5	R5			R5	R5							
7							R6	R6			R6	R6							
8							R7	R7			R7	R7							
9	S14						S11									12	13		
10	S4	S5	S6	S7	S8	S9				S10	S15			17	2			3	16
11							R8	R8			R8	R8							
12							S18												
13							R10	S19											
14									S20										
15							R13	R13			R13	R13							
16											S21								
17								S22			R15								
18							R9	R9			R9	R9							
19	S14															23	13		
20	S4	S5	S6	S7	S8	S9				S10				24	2			3	
21							R14	R14			R14	R14							
22	S4	S5	S6	S7	S8	S9				S10				17	2			3	25
23							R11												
24							R12	R12											
25											R16								

En la tabla, los números que acompañan a los reduce indican cual producción debe usarse para reducir cuando se llega a ese estado. Se puede ver que ninguna celda de la tabla tiene conflictos.

3. Lexer

Una vez pensada la gramática, se comenzó la implementación del lexer que consta de la difinición de cada token que posee la gámatrica junto con su forma.

- Los tokens `true`, `false`, `null`, `[`, `]`, {, } y `:` se definen simplemente con el caracter o string que le corresponde.
- `string` y `number` son expresiones regulares un poco más comlejas que aceptan las cadenas que, según la especificación de JSON, pueden ser consideradas como strings o números, respectivamente.
- Además se definen las funciones t_error y t_newline que son necesarias que la herramienta muestre un mensaje de error cuando sea requerido e identificar un salto de línea respectivamente.

Por último, se define la instanciación del lexer de la siguiente manera: lexer = lex.lex().

4. Parser

El lexer permite tokenizar la cadena de entrada, resta conseguir la traducción a YAML de la misma usando la cadena tokenizada. Para esto, la herramienta pide que se describan las producciones de la grámatica y se definan los atributos necesarios para poder traducir de la manera correcta. Se decidió agregar los siguientes atributos:

- **claves:** Un array de strings sintetizado que indica, en los objetos, as claves que contiene. Esto nos permite chequear que no haya claves repetidas.
- **tabs:** Un atributo heredado de tipo int que indica la cantidad de tabs precedentes con la que debe ser impresa la traducción YAML de ese símbolo. Esta cantidad depende del nivel de nesting en el que se encuentre dentro del JSON.
- **yaml:** Un atributo sintetizado de tipo string que indica la traducción YAML de ese símbolo.

(0) $S' \rightarrow Value$ `{ Value.tab = -1; S'.yaml = "---\n" ++ Value.yaml}`

(1) $Value \rightarrow Object$ `{ Object.tabs = Value.tabs + 1; Value.yaml = Object.yaml; }`

(2) $Value \rightarrow Array$ `{ Array.tabs = Value.tabs + 1; Value.yaml = Array.yaml}`

(3) $Value \rightarrow$ `string { Value.yaml =  string.value;}`

(4) $Value \rightarrow$ `number { Value.yaml = str(number);}`

(5) $Value \rightarrow$ `true { Value.yaml = "true";}`

(6) $Value \rightarrow$ `false { Value.yaml = "false";}`

(7) $Value \rightarrow$ `null { Value.yaml = "";}`

(8) $Object \rightarrow$ `{ } { Object.yaml = {};}`

(9) $Object \rightarrow \{Members\}$

```
{ Member.tabs = Object.tabs;
  Members.claves = [],
  Object.yaml = "\n" ++ Members.yaml;}
```

(10) $Members \rightarrow Pair$

```
{ Pair.tabs = Members.tabs;
  Pair.claves = Members.claves;
  Members.yaml = Pair.yaml;}
```

(11) $Members \rightarrow Pair\,,Members_1$

```
{ Pair.tabs = Members.tabs;
  Members_1.tabs = Members.tabs;
  if( Pair.claves[0] not in Members_1.claves ){
     Members.yaml = Pair.yaml ++ "\n" ++ Members_1.yaml;
     Members.claves = Pair.claves ++ Members_1.claves;
  } else {
     ERROR;
  }
}
```

(12) $Pair \rightarrow$ `string` $:Value$

```
{ Pair.claves = [ string.value ];
  Value.tabs = Pair.tabs;
  Pair.yaml = string.value ++ ":" ++ Value.yaml;}
```

(13) $Array \rightarrow$ `[ ] { Array.yaml = "[]"}`

(14) $Array \rightarrow$ `[`$Elements$`]`

```
{ Elements.tabs = Array.tabs;
  Array.yaml = "\n" ++ Elements.yaml;}
```

(15) $Elements \rightarrow Value$

```
{ Value.tabs = Elements.tabs;
  Elements.yaml = ("  "*Value.tabs) ++ "- " ++ Value.yaml;}
```

(16) $Elements \rightarrow Value\,,Elements_1$

```
{ Value.tabs = Elements.tabs;
  Elements_1.tabs = Elements.tabs;
  Elements.yaml = ("  "*Value.tabs) ++ "- " ++ Value.yaml ++ "\n" ++ Elements_1.yaml ;}
```

Por cada símbolo de la grámatica se implementaron una o más funciones que corresponden a las producciones de ese símbolo. Cada una de ellas acepta un único argumento 'p' que es un arreglo de elementos asociados a los terminales y no teminales de la producción.

En este array el primer objeto (p[0]) es el símbolo de la producción, luego, le siguen, en orden de aparición, cada símbolo de la producción.

Por elemplo en la función p_object:

```
def p_object(p):
'''object : LLAVEIZQ LLAVEDER'''
...
```

Aquí $p = [p_0, p_1, p_2]$ con p_0 el objeto asociado al simbolo *Object* y p_1 y p_2 los objetos asociodos a los tokens { y } respectivamente.

Ply, la herramienta que se está usando, solo permite el uso de atributos sintetizados. Por esta razón, al momento de implementar lo atributos para cada símbolo, se decició hacer la traducción recorriendo dos veces el parsing tree y colapsar el atributo **tab** en el atributo **yaml**.

La primer pasada corresponde a la creación del árbol. Aquí se crearán los nodos del mismo cada uno con su atributo **yaml** que es una función lambda que toma como parámetros la cantidad de tabs con la que debe imprimirse un token y devuelve el string generado a partir de constantes y los strings devueltos por los hijos en la derivación de ese símbolo.

Quedando los elementos del array p de tipo `TokenWithAttributes` que tiene la siguiente forma:

```
class TokenWithAttributes:
        def __init__(self, yaml, claves):
                self.yaml = yaml;  #La función lambda
                self.claves = [];  # Atributo solo usado por los objetos.
```

En la segunda pasada, solo se invoca el atributo **yaml** del primer símbolo (S') con el parametro -1, lo que provocará una cadena de llamadas recursivas sobre todos los símbolos del parsing tree y terminará devolviendo la cadena deseada.

5. Desiciones de diseño

Para realizar la conversión de código JSON a YAML se precisó utilizar el estilo 'double-quoted', que conciste en explicitar el tipo string de una cadena agregándole comillas, ya que había 2 conflictos:

- **Cadenas de un tipo que podían confundirse con otro.** Por ejemplo, si se tiene la cadena $-$Cadena3 como clave de un objeto, a simple vista pareciera ser un item de un arreglo, cuando no lo es.
- **Otro conflicto se presenta cuando tenemos un catacter como '\n',** ya que se necesita una representación para el salto de línea. Decidimos utilizar el estilo para la representación de todas las direcivas '\x' ya que parece adecuado para la traducción.

6. Pruebas

Se testeó el parser con los archivos `jasonObjet.txt`,`jasonObjet1.txt`,`jasonObjet2.txt` y `jasonObjet3.txt` consiguiendo los resultados deseados.

- `jasonObjet.txt` y `jasonObjet1.txt` contienen una cadena JSON válida cada uno
- `jasonObjet2.txt` contiene una cadena JSON incompleta, por lo que su sintaxis es érronea
- y `jasonObjet3.txt` contiene una cadena JSON con un objeto cuyas claves están repetidas.

7. Requerimientos de Software

Como se mencionó se usaron las siguientes herramientas:

- Python, Version: 2.6
- PLY, Version: 3.5

7.1. Comando de ejecución

Para ejecutar el trabajo es necesario escribir por entrada standard un objeto JSON, por ejemplo:

```
python3 tptleng.py < jasonObjet.txt
```

8. Referencias

[-1-] Página: LR Parser Table Generator.