

**Teoria de Lenguajes**  
**Primer Cuatrimestre de 2018**  
Conversor de JSON a YAML

Integrante	LU	Correo electrónico
Reigner, Ezequiel		
Zamboni, Gianfranco	219/13	<code>gianfranco375@gmail.com</code>
Pesaresi, Natalia	636/14	<code>natalia.pesaresi@gmail.com</code>

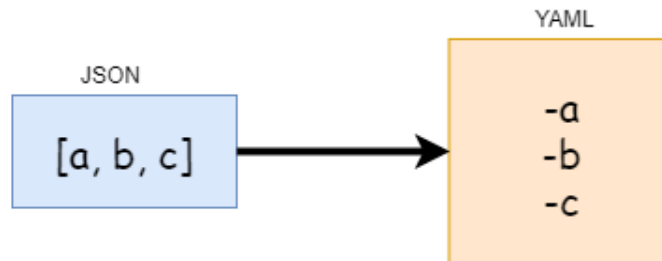
**Reservado para la ctedra**

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

# 1 Introducción

En este trabajo, se propone realizar un conversor de código JSON a código YAML. Ambos lenguajes poseen las mismas estructuras, por lo que hay equivalencias sintácticas entre ellos.

Por ejemplo en el caso de los arrays:



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

# 2 Gramática

La siguiente gramática G, genera el lenguaje aceptado por el lenguaje JSON.  
G={ {Value, Object, Array, Members, Pair, Elements}, {string, number, true, false, null, [, ], {, }, :, Value, P } }

Donde P:

```
Value -> Object | Array | string | number | true | false | null
Object -> { } | {Members}
Members -> Pair | Pair,Members
Pair -> string:Value
Array -> [ ] | [Elements]
Elements -> Value | Value,Elements
```

Una característica importante que necesitaremos conocer, a la hora de hacer el parser, será su tipo. Se quiso analizar si la gramática es LALR(1). Para ello, utilizamos una herramienta que genera para una gramática su tabla LALR. [-1-]

LR table																						
State	ACTION											GOTO										
	{	}	,	string	:	[	]	number	Object	Array	true	false	null	\$	S'	OBJECT	MEMBERS	PAIR	ARRAY	ELEMENTS	VALUE	
0				s2				s3	s4	s5	s6	s7	s8									1
1														acc								
2														r10								
3														r11								
4														r12								
5														r13								
6														r14								
7														r15								
8														r16								

Como la tabla no presenta conflictos, podemos concluir que la gramática es LALR(1),

### 3 Lexer

Para generar el Lexer, utilizamos la biblioteca de python Lex.

La estructura del lexer tiene las siguientes características:

- Definición de los tokens que posee la gramática
- Definición de las funciones t\_NUMBER y t\_STRING donde se especifica la forma de los números y cadenas de caracteres que acepta la gramática. Por ejemplo: '1.0' será un numero válido, pero '1.0ee' no.
- Definición de las funciones t\_error y t\_newline que son necesarias por la biblioteca para mostrar 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

Para generar el Parser, utilizamos la biblioteca de python Yacc. Éste tomará el árbol de tokens credo por el Lexer.

Se crearán tantas funciones como no terminales existan. Cada una de ellas

aceptará un único argumento 'p' que es un arreglo de elementos p asociados a los terminales y no terminales de la producción que tiene como parte derecha que corresponde a la función.

Por ejemplo en la función:

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

```
p = [p_0,p_1,p_2]
```

Un dato relevante es que 'p' tiene un sólo atributo 'value' que debe ser 'sintetizado'.

Durante la ejecución del parser se genera lo que llamamos 'Parsing Tree', esto es el árbol de objetos 'p' que se genera a partir de las sucesivas llamadas a función. Por ejemplo, al parsear: [a, b, c]. El 'Parsing Tree' será el árbol:  $p\_Value \rightarrow p\_Object \rightarrow [p\_Members] \rightarrow Value$ ,  $p\_Elements \rightarrow Value$ ,  $p\_Elements \rightarrow Value$

En nuestro trabajo, por las características del código YAML, deseábamos contar con un atributo TABS 'heredado', y un atributo CADENA 'sintetizado'. Por lo expresado anteriormente, esto no era posible, por lo que debimos modificar nuestra estrategia de diseño del parser.

Se procedió a crear la Clase TokenWithAttributes:

```
classTokenWithAttributes:  
    def__init__(self, tipo, yaml):  
        self.tipo = tipo  
        self.yaml = yaml
```

Por lo tanto ahora los elementos 'p', serán del tipo TokenWithAttributes. El atributo Tipo, será un string y el atributo 'Yaml' será una función que toma como parametro los tabs correspondientes a la posición del objeto actual.

La estructura final del parser será:

- Generamos el 'Parsing Tree'.  $parsedValue = p[1]$
- Utilizamos el atributo 'Yaml' del parsedValue, para imprimir la traducción.

## 5 Referencias

[-1-] Página:

<http://jsmachines.sourceforge.net/machines/lalr1.html>

.