

Teoría de Lenguajes

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico

Integrante	LU	Correo electrónico
Cioppettini, Enzo	405/15	tenstrings5050@gmail.com
Jaratz, Tomas	59/15	tomyjaral@gmail.com
Pastore, Tomas	266/15	pastoretomas96@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción	3
2. Gramática	4
3. Implementación	7
4. Ejemplos	12
5. Manual de uso	13
6. Conclusiones	14

1. Introducción

El objetivo de este trabajo práctico es desarrollar un conversor de JSON a YAML. Como bien es sabido estos dos son formatos de serialización de datos muy utilizados en el desarrollo de software. La idea es utilizar las herramientas vistas en la materia para implemetar un *parser* que sea capaz de reconocer cadenas pertenecientes al formato JSON y luego generar el output correspondiente en YAML.

Decidimos utilizar Python y ANTLR para generar el lexer y el parser asociados a la gramática de JSON, la cual la obtuvimos de la página oficial, sin embargo tuvimos que realizar ciertas modificaciones en la gramática para que la implementación del parser sea exitosa.

2. Gramática

Las producciones de la gramática a las que llegamos son las siguientes y se encuentran en el archivo **json.g4** :

```
collection → obj | arr  
  
obj → { } | { pair (, pair)* }  
  
arr → [] | [ valueArray (, valueArray)* ]  
  
pair → pairFirst | value  
  
pairFirst → string  
  
valueArray → value  
  
value → collection | string | num | tr | fs | nil  
  
string → STRING  
  
num → NUM  
  
tr → true  
  
fs → false  
  
nil → null
```

Las producciones *pairFirst*, *valueArray* y *collection* no son necesarias en sí para generar la gramática, por ejemplo *pairFirst* podría ser sustituida sin problemas por *string*, mientras que *valueArray* por *value*. Más adelante en el informe vamos a ver cual es el sentido de estas producciones, por ahora basta con decir que son un detalle particular de la implementación, que si bien podría haber sido omitido, nos simplificó las cosas a la hora de escribir el código, por otro lado la gramática aceptada sigue siendo la misma de JSON.

El siguiente es el contenido exacto del archivo json.g4 que toma ANTLR para generar el parser:

```
grammar json;

//PARSER RULES

json : value EOF;

collection :
    obj | arr ;

obj :
    '{' |
    '{' pair (',' pair)* '}';

pair :
    pairFirst ':' value;

pairFirst :
    string;

arr :
    '[' | '[' valueArray (',' valueArray)* ']';

valueArray :
    value;

value :
    collection | string | num | tr | fs | nil;

string :
    STRING;

num :
    NUMBER;

tr :
    'true';

fs :
    'false';

nil :
    'null';

//LEXER RULES

fragment DIGIT : [0-9];

WHITESPACE : [ \n\t\r]+ -> skip;

NUMBER :
    INT FRAC? EXP?;

INT :
    '-'? DIGIT |
```

```

    '-'? [1-9][0-9]*;

FRAC :
    '.' DIGIT*;

EXP :
    [eE][+|-]? DIGIT*;

fragment INVALID :
    ~ ["\u0000-\u001f\\"];

fragment ESCAPED :
    '\\ ' ["\\/\bfnrt];

fragment HEX :
    [0-9a-fA-F];

fragment STRINGUNICODE:
    '\\u' HEX HEX HEX HEX;

STRING :
    '"' ( INVALID | ESCAPED | STRINGUNICODE )* '"';

```

3. Implementación

En **json.g4** como vimos antes, es donde implementamos la gramática y además las reglas del lexer.

En **toYaml.py** definimos las reglas de conversión de JSON a YAML. El parser generado por ANTLR en base a la gramática define una serie de métodos que se ejecutan al 'entrar' y 'salir' de cada una de las reglas de producción que sirven para generar la cadena de salida, en nuestro caso en formato YAML. Por ende la única tarea difícil fue entender que acciones tomar en cada producción tanto antes de entrar como al salir de estas.

En la clase *toYaml* definimos 4 atributos necesarios para esto:

- **counter**: es un entero utilizado como una referencia global que nos permite mantener la indentación, al entrar en un objeto lo aumentamos y así los pares dentro del objeto saben cuantos espacios dejar para la indentación, los mismo para los arrays. En los métodos **enterCollection**, **exitCollection**, **enterPair** y **enterValueArray** se puede observar su uso. Al salir de una producción *collection* se decrementa el valor de counter para mantener la consistencia.
- **first**: es un booleano que sirve para saber si hace falta realizar un salto de línea al entrar en una producción del tipo *collection*, lo cual sucede en todos los casos salvo en la primera producción.
- **firstPair**: la misma idea pero para los pares
- **firstArrayValue**: idem

El código es el siguiente:

```
from antlr4 import *
if __name__ is not None and "." in __name__:
    from .jsonParser import jsonParser
else:
    from jsonParser import jsonParser

from jsonListener import jsonListener

class toYaml(jsonListener):

    def __init__(self, output):
        self.output = output
        self.counter = 0
        self.first = True
        self.firstPair = True
        self.firstArrayValue = True

    # Enter a parse tree produced by jsonParser#json.
    def enterJson(self, ctx:jsonParser.JsonContext):
        pass

    # Exit a parse tree produced by jsonParser#json.
    def exitJson(self, ctx:jsonParser.JsonContext):
        self.output.write("\n")

    # Enter a parse tree produced by jsonParser#obj.
    def enterObj(self, ctx:jsonParser.ObjContext):
```

```

keys = [p.pairFirst().getText() for p in ctx.pair()]

if len(keys) != len(set(keys)):
    raise Exception("Clave repetida")
else:
    self.firstPair = True

# Exit a parse tree produced by jsonParser#obj.
def exitObj(self, ctx:jsonParser.ObjContext):
    pass

def enterCollection(self, ctx:jsonParser.CollectionContext):
    empty = None
    collectionValues = None
    if ctx.obj():
        collectionValues = [p for p in ctx.obj().pair()]
        empty = '{}'
    elif ctx.arr():
        collectionValues = [p for p in ctx.arr().valueArray()]
        empty = '[]'

    if collectionValues == []:
        self.output.write(empty)
    else:
        if not self.first:
            self.output.write('\n')
            self.counter = self.counter + 2
        else:
            self.first = False

def exitCollection(self, ctx:jsonParser.CollectionContext):
    self.counter = self.counter - 2

# Enter a parse tree produced by jsonParser#pair.
def enterPair(self, ctx:jsonParser.PairContext):
    if self.firstPair:
        self.firstPair = False
    else:
        self.output.write('\n')

    self.output.write(' ' * self.counter)

def enterValueArray(self, ctx:jsonParser.ValueArrayContext):
    if self.firstArrayValue:
        self.firstArrayValue = False
    else:
        self.output.write('\n')

    self.output.write(' ' * self.counter + '-')

def exitValueArray(self, ctx:jsonParser.ValueArrayContext):
    pass

# Exit a parse tree produced by jsonParser#pair.
def exitPair(self, ctx:jsonParser.PairContext):
    pass

```



```

# Exit a parse tree produced by jsonParser#pair.
def exitPairFirst(self, ctx:jsonParser.PairContext):
    self.output.write(': ')

# Enter a parse tree produced by jsonParser#arr.
def enterArr(self, ctx:jsonParser.ArrContext):
    self.firstArrayValue = True

# Exit a parse tree produced by jsonParser#arr.
def exitArr(self, ctx:jsonParser.ArrContext):
    pass

# Enter a parse tree produced by jsonParser#value.
def enterValue(self, ctx:jsonParser.ValueContext):
    pass

# Exit a parse tree produced by jsonParser#value.
def exitValue(self, ctx:jsonParser.ValueContext):
    #self.output.write("\n")
    pass

# Enter a parse tree produced by jsonParser#string.
def enterString(self, ctx:jsonParser.StringContext):
    str = ctx.STRING().getText()
    if not ('-' in str or '\\n' in str):
        str = str.replace('"', '')
    self.output.write(str)

# Exit a parse tree produced by jsonParser#string.
def exitString(self, ctx:jsonParser.StringContext):
    pass

# Enter a parse tree produced by jsonParser#num.
def enterNum(self, ctx:jsonParser.NumContext):
    self.output.write(ctx.NUMBER().getText())

# Exit a parse tree produced by jsonParser#num.
def exitNum(self, ctx:jsonParser.NumContext):
    pass

# Enter a parse tree produced by jsonParser#tr.
def enterTr(self, ctx:jsonParser.TrContext):
    self.output.write(ctx.getText())

# Exit a parse tree produced by jsonParser#tr.
def exitTr(self, ctx:jsonParser.TrContext):
    pass

# Enter a parse tree produced by jsonParser#fs.
def enterFs(self, ctx:jsonParser.FsContext):
    self.output.write(ctx.getText())

```

```

# Exit a parse tree produced by jsonParser#fs.
def exitFs(self, ctx:jsonParser.FsContext):
    pass

# Enter a parse tree produced by jsonParser#nil.
def enterNil(self, ctx:jsonParser.NilContext):
    self.output.write('')

# Exit a parse tree produced by jsonParser#nil.
def exitNil(self, ctx:jsonParser.NilContext):
    pass

```

Como introducimos en la gramática anteriormente, se puede observar que en la regla **exitPair-First** imprimimos un ":", de haber dejado solo un *string* no hubiéramos podido hacerlo, pero ANTLR nos permite jugar con la gramática para salvar estos casos. Lo mismo sucede en **enter-ValueArray**, donde, para no hacer dos saltos de línea al parsear el primer elemento, utilizamos el atributo *firstArrayValue*.

Por último en **main.py** es donde armamos todas las estructuras e importamos los módulos necesarios para correr el parser, el código es el siguiente:

```

#!/usr/bin/env python3
import sys
from antlr4 import *
from jsonLexer import jsonLexer
from jsonParser import jsonParser
from toYaml import toYaml
from jsonListener import jsonListener
import codecs

class StdinStream(InputStream):

    def __init__(self, encoding='utf-8', errors='strict'):
        bytes = sys.stdin.read()
        super().__init__(bytes)

def main(filename = None):
    if filename:
        input = FileStream(filename)
    else:
        input = StdinStream()
    lexer = jsonLexer(input)
    stream = CommonTokenStream(lexer)
    parser = jsonParser(stream)
    tree = parser.json()

    output = sys.stdout

    converter = toYaml(output)
    listener = jsonListener()
    walker = ParseTreeWalker()
    walker.walk(converter, tree)

```

```
if __name__ == '__main__':  
    if len(sys.argv) == 2:  
        main(sys.argv[1])  
    else:  
        main()
```

No hay mucho que explicar, simplemente seguimos un tutorial de ANTLR y lo que se ve es la configuración básica para hacerlo funcionar.

4. Ejemplos

A continuación incluimos un ejemplo del correcto funcionamiento del parser, la entrada en formato JSON y la salida en YAML:

Entrada

```
1 {
2   "materia" : "tleng" ,
3   "divertida" : true ,
4   "informacion" :
5     {
6       "anio" : 2018 ,
7       "cuatrimestre" : "primero" ,
8       "integrantes" : ["enzo", "tomi", "tomi"] ,
9       "grupo" : "el dragon" ,
10      "rendimiento" :
11        {
12          "primer parcial" : 43 ,
13          "segundo parcial" : 50
14        },
15      "lista de objetos" : [{
16        "a" : 1,
17        "b" : "dos",
18        "c" : true
19      }, {
20        "a" : 2,
21        "b" : "uno",
22        "c" : false
23      }]
24    } ,
25
26  "caracter especial" : "ti\nerra",
27  "evaluacion final del tp" : "Aprobado summa cum laude"
28 }
29 }
```

Salida

```
1 materia: tleng
2 divertida: true
3 informacion:
4   anio: 2018
5   cuatrimestre: primero
6   integrantes:
7     -enzo
8     -tomi
9     -tomi
10  grupo: el dragon
11  rendimiento:
12    primer parcial: 43
13    segundo parcial: 50
14  lista de objetos:
15    -
16      a: 1
17      b: dos
18      c: true
19    -
20      a: 2
21      b: uno
22      c: false
23  caracter especial: "ti\nerra"
24  luacion final del tp: Aprobado summa cum laude
```

5. Manual de uso

En esta sección veremos los requerimientos para correr el programa y cómo utilizarlo. Los requerimientos son:

- Python 3
- Java 1.6 o superior
- Instalar ANTLR4 (el runtime): `pip install antlr4-python3-runtime`

Para correr el programa simplemente ejecutar `main.py` pasándole por la entrada estandar el json. Otra forma es pasarle el nombre del archivo con el json al programa, por ejemplo: `python3 main.py examples/ex1.json`.

6. Conclusiones

- *ANTLR* demostró ser una herramienta muy poderosa que nos permitió llevar, casi sin problemas, la implementación del parser a partir de la gramática de JSON, sin embargo tuvimos que realizar ciertas modificaciones a esta para que el código sea mas sencillo.
- Podemos destacar que los temas vistos durante la cursada como gramáticas, parsers, resolución de conflictos etc. fueron útiles para poder entender y resolver los problemas presentados. El trabajo práctico fue una buena aplicación de los conceptos vistos, los cuales de otra forma hubiesen quedado solo en el marco teórico.