DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: "Conversor de JSON a YAML"

Informe y análisis de resultados.

Teoría de Lenguajes

Grupo 6

Integrante	LU	Correo electrónico
Enrique, Natalia	459/12	natu.enrique@gmail.com
Mascitti, Augusto	954/11	mascittija@gmail.com
Len, Nicolás	819/11	nicolaslen@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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) 4576-3359
http://www.fcen.uba.ar

Índice

Introducción

Se desea realizar un conversor de JSON a YAML.

Se recibirá como entrada una cadena de texto, se verificará si cumple la sintaxis y restricciones de tipado del lenguaje JSON, y finalmente se convertirá el código con la 'indentación' y reglas adecuadas del lenguaje YAML. En caso de detectarse algún error, se deberá informar claramente cuáles son las características del mismo.

Durante este trabajo utilizaremos la herramienta PLY, que nos permitirá definir las reglas para el lexer, las reglas para el parser, y luego generar el árbol sintáctico. Una vez generado el mismo, podremos utilizarlo para el análisis semántico de la expresión dada.

1. Implemenatación

Para este trabajo decidimos utilizar una gramática LALR (Look-Ahead, Left to Right, RightMost derivation), es decir, definimos una gramática que al ser procesada por un parser LALR no presenta conflictos de ningún tipo.

1.1. Lexer

Antes de definir la gramática, procederemos a mostrar las reglas utilizadas para generar el lexer. La herramienta PLY nos permite definir las reglas mediante la utilización de expresiones regulares. A continuación detallamos las reglas generadas:

Token	Expresión Regular	Símbolo Terminal
BEGIN_ARRAY	\[	[
BEGIN_OBJECT	\{	{
END_ARRAY	\]	]
END_OBJECT	\}	}
NAME_SEPARATOR	\:	:
VALUE_SEPARATOR	\,	,
QUOTATION_MARK	\”	”
FALSE	false	false
TRUE	true	true
NULL	null	null
DECIMAL_POINT	\ .	.
DIGITS	$[0-9]+$	Números del 0 al 9
E	$[eE]$	Exponencial
MINUS	\-	-
PLUS	\+	+
STRING	[\\,\ /,\b,\f,\n,\r,\t,\x20-\x21, \x23-\x5B,\x5D-\xFFFF]+	Cadenas de texto y nombres de claves
ignore	’ \t \n \r’	Espacios, tabs y saltos de línea

A la hora de definir las reglas del lexer, nos encontramos con dos conjuntos de producciones que vale la pena distinguir. En el primero se encuentran aquellas producciones donde se leen objetos, arrays, números entre otros (reglas simples). En el otro, se encuentran aquellas producciones donde se leen strings delimitado por comillas dobles. Además, fue necesario crear tokens específicos para aquellos casos en los que recibimos números, ya que podríamos obtener enteros, exponenciales o de punto flotante.

1.2. Gramática

Una vez definidas las reglas del lexer y utilizando los tokens generados, definimos la siguiente gramática...

S → elements
elements → value
elements → value VALUE_SEPARATOR elements
value → string
value → number
value → object
value → array
value → TRUE
value → FALSE
value → NULL
object → BEGIN_OBJECT END_OBJECT
object → BEGIN_OBJECT members END_OBJECT
array → BEGIN_ARRAY END_ARRAY
array → BEGIN_ARRAY elements END_ARRAY
members → pair

members → pair VALUE_SEPARATOR members
pair → STRING NAME_SEPARATOR value
number → integer
number → integer DECIMAL_POINT DIGITS
number → integer E DIGITS
number → integer E PLUS DIGITS
number → integer E MINUS DIGITS
number → integer DECIMAL_POINT DIGITS E DIGITS
number → integer DECIMAL_POINT DIGITS E PLUS DIGITS
number → integer DECIMAL_POINT DIGITS E MINUS DIGITS
integer → DIGITS
integer → MINUS DIGITS
string → QUOTATION_MARK STRING QUOTATION_MARK
string → QUOTATION_MARK QUOTATION_MARK

1.3. Parser

En esta parte del trabajo realizamos la traducción de la expresión recibida que se separó de dos etapas:

- Chequeo de valores.
- Traducción de cada expresión.

Una vez realizado el análisis sintáctico, si la cadena de entrada no fue rechazada, no solo sabemos que es una cadena válida del lenguaje, sino que también disponemos del Abstract Syntax Tree. Luego, la idea consiste básicamente en diseñar e implementar clases y/o estructuras de datos que nos permitan obtener datos de cada nodo del árbol, a partir de información obtenida de las clases de los nodos hijos. Separar las reglas de cada producción, nos permitirá realizar operaciones más complejas y de un modo más simple e intuitivo sobre el árbol.

Para eso creamos una superclase Expressions, para poder implementar las reglas semánticas con clases que extiendan a Expressions. En esta gramática, únicamente se desea crear la cadena YAML convertida a partir de una cadena JSON. Es por eso que todas las subclases implementarán el método value, que será el método que nos permitirá a obtener el valor convertido de cada expresión.

1.3.1. Chequeo de Valores

Para poder realizar el chequeo de valores, cada nodo del árbol (representando a una producción), tiene que poder determinar cuál es su propio valor, y a excepción de las hojas del árbol, dependen dependen de sus subexpresiones. La idea, entonces, es que cada subexpresión pueda retornar su propio valor. De esta forma, se podría ir determinando el valor de toda la expresión y validando si es correcto.

1.3.2. Traducción de cada expresión

Para cada expresión, luego de corroborar su validez con respecto al lenguaje, se realizarán determinadas operaciones para conseguir la traducción deseada. Como YAML trabaja las jerarquías de sus expresiones con indentaciones, en las expresiones que se generen objetos y listas con JSON, se guardará en una lista la indentación que se deberá aplicar en sus elementos, representados por las hojas de los subárboles de los nodos que inician un objeto o una lista. Cuando se llega a las expresiones de esas hojas, se usa la función Indent (implementada en la superclase Expression para ser reutilizada), que a partir de esa lista, genera la indentación correspondiente.
Por otro lado, para verificar que un objeto no cuente con claves repetidas, creamos un set "keys"donde se almacena las claves de cada elemento del objeto, en caso de que todavía no pertenezcan a "keys". En caso de pertenecer, se generará una excepción, finalizando la ejecución del programa.

2. Casos de Prueba

Por último testeamos el parser con algunos casos.

- Test 1:
  JSON:

  ```
  [1, [2, 3], [1, [2, 3]]]
  ```

  YAML:

  ```
  - 1
  -
    - 2
    - 3
  -
    - 1
    -
      - 2
      - 3
  ```

- Test 2:
  JSON:

  ```
  [1, [3, 4], {"h":"o","l":"a"}]
  ```

  YAML:

  ```
  - 1
  -
    - 3
    - 4
  -
    h: o
    l: a
  ```

- Test 3:
  JSON:

  ```
  "hola\n\b\t\tnico\u0033"
  ```

  YAML:

  ```
  "hola\\n\\b\\t\\tnico\\u0033"
  ```

- Test 4:
  JSON:

  ```
  [1, [3, 4], {"h":"o","l":"a"}
  ```

  Notemos que para el primer corchete, falta el cierre del mismo, por lo que no será una expresión válida.
  YAML:

  ```
  Hubo un error durante el parseo.
  La expresión no puede ser parseada por la producción (..)
  ```

- Test 5: JSON:

  ```
  [ "Cadena con salto\nde línea", [null, 35.6e9, {}], -1,true ]
  ```

  YAML:

  ```
  - "Cadena con salto\nde línea"
  -
    -
    - 35.6e9
    - {}
  - -1
  - true
  ```

3. Código

El programa requiere tener instalado python 2.7 y el plugin pip para instalar sus librerías.
Al ingresar en la carpeta del código desde la consola, se deberá ejecutar el siguiente comando:

```
make install
```

que a través de pip, instalará el PLY y todas sus dependencias.
Una vez instalado, para utilizar el programa se deberá ejecutar el siguiente comando:

```
python main.py
```

De modo interactivo, se podrán ingresar expresiones por stdin. Si se ingresa la cadena exit, se finaliza la ejecución.

3.1. main.py

```
"""Archivo principal de json2yaml."""
from json2yaml import parse
import sys
from string import join

def execute(expression):
        try:
                parsed = parse(expression)
                print "YAML> {0}\n".format(parsed.value([]))
        except Exception as e:
                sys.stderr.write(str(e) + "\n")
                exit(1)


def main():
        if len(sys.argv) < 2:
                while True:
                        expression = raw_input('JSON> ')
                        if expression == "exit":
                                break
                        else:
                                execute(expression)
        else:
                execute(join(sys.argv[1:]))



if __name__=='__main__':
        main()
```

3.2. lexer.py

```
#! coding: utf-8
import ply.lex as lex


""" Lista de tokens """
tokens = [
  # Initial state tokens
  'BEGIN_ARRAY',
  'BEGIN_OBJECT',
```

```
    'END_ARRAY',
    'END_OBJECT',
    'NAME_SEPARATOR',
    'VALUE_SEPARATOR',
    'QUOTATION_MARK',
    'FALSE',
    'TRUE',
    'NULL',
    'DECIMAL_POINT',
    'DIGITS',
    'E',
    'MINUS',
    'PLUS',
    'STRING'
]

    # 2 estados del lexer:
    #
    #   default:
    #     Por defecto, lee objetos, arrays, numeros, etc.
    #   string:
    #     Delimitado por comillas dobles.

states = (
    ('string', 'exclusive'),
)

# Default state tokens
t_BEGIN_ARRAY           = r'\['
t_BEGIN_OBJECT          = r'\{'
t_END_ARRAY             = r'\]'
t_END_OBJECT            = r'\}'
t_NAME_SEPARATOR        = r'\:'
t_VALUE_SEPARATOR       = r'\,'
t_FALSE                 = r'false'
t_TRUE                  = r'true'
t_NULL                  = r'null'
t_DECIMAL_POINT         = r'\.'
t_DIGITS                = r'[0-9]+'
t_E                     = r'[eE]'
t_MINUS                 = r'\-'
t_PLUS                  = r'\+'


t_ignore = '␣\t\n\r'

# No ignorar ningun caracter dentro del estado string
t_string_ignore = ''


# Ingresa el estado string en una comilla de apertura
def t_QUOTATION_MARK(t):
    r'\"'
    t.lexer.push_state('string')
    return t

def t_string_STRING(t):
    r'[\\,\/,\b,\f,\n,\r,\t,\x20-\x21,\x23-\x5B,\x5D-\xFFFF]+'
    t.value = unicode(t.value, encoding='utf8')
```

```
  return t

# Sale del estado string en una comilla de cierre
def t_string_QUOTATION_MARK(t):
  r'\x22'  # '"'
  t.lexer.pop_state()
  return t

def t_error(t):
    raise Exception("Expresion invalida '%s'" %t.value)

def t_string_error(t):
    raise Exception("Expresion invalida '%s'" %t.value)

# Build the lexer
lexer = lex.lex()

def apply_lexer(string):
    """Aplica el lexer al string dado."""
    print string
    lexer.input(string)

    return list(lexer)
```

3.3. parser.py

```
#! coding: utf-8
"""Parser JSON to YAML."""
import ply.yacc as yacc
from lexer import tokens
from expressions import *

# E -> ...
def p_expression_elements_value(subexpr):
  'elements : value'
  subexpr[0] = LastElementArrayExpression(subexpr[1])

def p_expression_elements_list(subexpr):
  'elements : value VALUE_SEPARATOR elements'
  subexpr[0] = ElementArrayExpression(subexpr[1], subexpr[3])

# O -> ...
def p_expression_object(subexpr):
  'object : BEGIN_OBJECT members END_OBJECT'
  subexpr[0] = ObjectExpression(subexpr[2])

def p_expression_object_empty(subexpr):
  'object : BEGIN_OBJECT END_OBJECT'
  subexpr[0] = ObjectEmptyExpression()

# M -> ...
def p_expression_members(subexpr):
  'members : pair'
  subexpr[0] = LastElementObjectExpression(subexpr[1])

def p_expression_members_list(subexpr):
  'members : pair VALUE_SEPARATOR members'
  subexpr[0] = ElementObjectExpression(subexpr[1], subexpr[3])
```

```
# P -> ...
def p_expression_pair(subexpr):
  'pair : string NAMESEPARATOR value'
  subexpr[0] = PairExpression(subexpr[1], subexpr[3])

# A -> ...
def p_expression_array_empty(subexpr):
  'array : BEGIN_ARRAY END_ARRAY'
  subexpr[0] = ArrayEmptyExpression()

def p_expression_array_list(subexpr):
  'array : BEGIN_ARRAY elements END_ARRAY'
  subexpr[0] = ArrayExpression(subexpr[2])

# V -> ...
def p_expression_value_string(subexpr):
  'value : string'
  subexpr[0] = ValueExpression(subexpr[1])

def p_expression_value_false(subexpr):
  'value : FALSE'
  subexpr[0] = FalseExpression(subexpr[1])

def p_expression_value_true(subexpr):
  'value : TRUE'
  subexpr[0] = TrueExpression(subexpr[1])

def p_expression_value_null(subexpr):
  'value : NULL'
  subexpr[0] = EmptyExpression()

def p_expression_value_number(subexpr):
  'value : number'
  subexpr[0] = ValueExpression(subexpr[1])

def p_expression_value_object(subexpr):
  'value : object'
  subexpr[0] = ValuePopExpression(subexpr[1])

def p_expression_value_array(subexpr):
  'value : array'
  subexpr[0] = ValuePopExpression(subexpr[1])

# int -> ...
def p_expression_integer(subexpr):
  'integer : DIGITS'
  subexpr[0] = NumberExpression(subexpr[1])

def p_expression_integer_negative(subexpr):
  'integer : MINUS DIGITS'
  subexpr[0] = NegativeNumberExpression(subexpr[2])

# N -> ...
def p_expression_number(subexpr):
  'number : integer'
  subexpr[0] = ValueExpression(subexpr[1])

def p_expression_frac(subexpr):
  'number : integer DECIMAL_POINT DIGITS'
```

```
  subexpr[0] = FracExpression(subexpr[1], subexpr[3])

def p_expression_exp(subexpr):
  'number : integer E DIGITS'
  subexpr[0] = ExpExpression(subexpr[1], subexpr[3])

def p_expression_exp_positive(subexpr):
  'number : integer E PLUS DIGITS'
  subexpr[0] = ExpExpression(subexpr[1], subexpr[4])

def p_expression_exp_negative(subexpr):
  'number : integer E MINUS DIGITS'
  subexpr[0] = ExpNegativeExpression(subexpr[1], subexpr[4])

def p_expression_frac_exp(subexpr):
  'number : integer DECIMAL_POINT DIGITS E DIGITS'
  subexpr[0] = FracExpExpression(subexpr[1], subexpr[3], subexpr[5])

def p_expression_frac_exp_positive(subexpr):
  'number : integer DECIMAL_POINT DIGITS E PLUS DIGITS'
  subexpr[0] = FracExpExpression(subexpr[1], subexpr[3], subexpr[6])

def p_expression_frac_exp_negative(subexpr):
  'number : integer DECIMAL_POINT DIGITS E MINUS DIGITS'
  subexpr[0] = FracExpNegativeExpression(subexpr[1], subexpr[3], subexpr[6])

# S -> ...
def p_expression_string(subexpr):
  'string : QUOTATION_MARK STRING QUOTATION_MARK'
  subexpr[0] = StringExpression(subexpr[2])

def p_expression_string_empty(subexpr):
  'string : QUOTATION_MARK QUOTATION_MARK'
  subexpr[0] = EmptyExpression()

def p_error(p):
  message = "Hubo un error durante el parseo.\n"
  if p is not None:
    message += "Expresion '{0}' incorrecta en la posicion {1}."
    .format(p.value, str(p.lexpos))
  else:
    message += "La expresion no puede ser parseada por la produccion {0} de {1}:{2}."
    .format(parser.symstack, __file__.split("/")[-1], parser.state)

  raise Exception(message)


# Build the parser
parser = yacc.yacc(debug=True)

def apply_parser(str):
    return parser.parse(str)
```

3.4. expression.py

```
#! coding: utf-8

class Expression(object):
```

```
  def value(self, prefixs):
    raise NotImplementedError("Este metodo lo deben implementar las clases que hereden

  def indent(self, prefixs, exp):
    spaces = ""
    prefix = ""

    if prefixs:
      spaces = ("  " * (len(prefixs) - 1))
      prefix = prefixs[len(prefixs) - 1]

    return "{0}{1}{2}".format(spaces, prefix, exp)

# V -> S
# V -> N
# N -> int
class ValueExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs):
    return self.expression.value(prefixs)

# V -> O
# V -> A
class ValuePopExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs):
    exp = self.expression.value(prefixs)
    prefixs.pop()
    return exp

# O -> { M }
class ObjectExpression(Expression):
  def __init__(self, members):
    self.members = members

  def value(self, prefixs):
    prefixs.append("")
    return "\n{0}".format(self.members.value(prefixs, set()))

# O -> { }
class ObjectEmptyExpression(Expression):
  def value(self, prefixs):
    prefixs.append("")
    return "{}"

# A -> [ ]
class ArrayEmptyExpression(Expression):
  def value(self, prefixs):
    prefixs.append("- ")
    return "[]"

# A -> [ E ]
class ArrayExpression(Expression):
  def __init__(self, expression):
    self.expression = expression
```

```
  def value(self, prefixs):
    prefixs.append("- ")
    return "\n{0}".format(self.expression.value(prefixs))

# E -> V , E
class ElementArrayExpression(Expression):
  def __init__(self, valueExpression, elementsExpression):
    self.valueExpression = valueExpression
    self.elementsExpression = elementsExpression

  def value(self, prefixs):
    exp = "{0}\n{1}".format(self.valueExpression.value(prefixs), self.elementsExpressio
    return Expression.indent(self, prefixs, exp)

# M -> P , M
class ElementObjectExpression(Expression):
  def __init__(self, pairExpression, elementsExpression):
    self.pairExpression = pairExpression
    self.elementsExpression = elementsExpression

  def value(self, prefixs, keys):
    exp = "{0}\n{1}".format(self.pairExpression.value(prefixs, keys), self.elementsExp
    return Expression.indent(self, prefixs, exp)

# E -> V
class LastElementArrayExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs):
    return Expression.indent(self, prefixs, self.expression.value(prefixs))

# M -> P
class LastElementObjectExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs, keys):
    return Expression.indent(self, prefixs, self.expression.value(prefixs, keys))

# P -> S : V
class PairExpression(Expression):
  def __init__(self, stringExpression, valueExpression):
    self.stringExpression = stringExpression
    self.valueExpression = valueExpression

  def value(self, prefixs, keys):
    key = self.stringExpression.value(prefixs)

    if (key in keys):
      raise Exception("Clave repetida")
    else:
      keys.add(key)

    return "{0}: {1}".format(key, self.valueExpression.value(prefixs))

# S -> " string "
class StringExpression(Expression):
```

```
  def __init__(self, expression):
    self.expression = expression.encode('utf-8')

  def value(self, prefixs):
    if ((self.expression[:1] == '-') or ("\\" in self.expression) or ("\/" in self.expr
      exp = self.expression.replace("\\","\\\\").replace("\/","\\/").replace("\b","\\b"
      return "\"{0}\"".format(exp)
    else:
      return self.expression

# int -> DIGITS
class NumberExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs):
    return str(self.expression)

# int -> MINUS DIGITS
class NegativeNumberExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs):
    return "-{0}".format(str(self.expression))

# N -> int DECIMAL_POINT DIGITS
class FracExpression(Expression):
  def __init__(self, integerExpression, fracExpresssion):
    self.integerExpression = integerExpression
    self.fracExpresssion = fracExpresssion

  def value(self, prefixs):
    return "{0}.{1}".format(self.integerExpression.value(prefixs), str(self.fracExpress

# N -> int E DIGITS
# N -> int E PLUS DIGITS
class ExpExpression(Expression):
  def __init__(self, integerExpression, expExpresssion):
    self.integerExpression = integerExpression
    self.expExpresssion = expExpresssion

  def value(self, prefixs):
    return "{0}e{1}".format(self.integerExpression.value(prefixs), str(self.expExpresss

# N -> int DECIMAL_POINT DIGITS E DIGITS
class FracExpExpression(Expression):
  def __init__(self, integerExpression, fracExpresssion, expExpresssion):
    self.integerExpression = integerExpression
    self.expExpresssion = expExpresssion
    self.fracExpresssion = fracExpresssion

  def value(self, prefixs):
    return "{0}.{1}e{2}".format(self.integerExpression.value(prefixs), str(self.fracExp

# N -> int DECIMAL_POINT DIGITS E MINUS DIGITS
class FracExpNegativeExpression(Expression):
  def __init__(self, integerExpression, fracExpresssion, expExpresssion):
    self.integerExpression = integerExpression
```

```
    self.expExpresssion = expExpresssion
    self.fracExpresssion = fracExpresssion

  def value(self, prefixs):
    return "{0}.{1}e-{2}".format(self.integerExpression.value(prefixs), str(self.fracEx

# N -> int E MINUS DIGITS
class ExpNegativeExpression(Expression):
  def __init__(self, integerExpression, expExpresssion):
    self.integerExpression = integerExpression
    self.expExpresssion = expExpresssion

  def value(self, prefixs):
    return "{0}e-{1}".format(self.integerExpression.value(prefixs), "e-", str(self.expl

# V -> true
class TrueExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs):
    return "true"

# V -> false
class FalseExpression(Expression):
  def __init__(self, expression):
    self.expression = expression

  def value(self, prefixs):
    return "false"

# V -> null
# S -> " "
class EmptyExpression(Expression):
  def value(self, prefixs):
    return ""
```