DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

JSON to YAML

Teoría de lenguajes

Integrante	LU	Correo electrónico
Daniel Nicolás Kundro	111/15	Dkundro@gmail.com
Rodrigo Oscar Kapobel	695/12	rokapobel35@gmail.com
Alicia Amalia Alvarez Mon	224/15	aliciaysuerte@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) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

La teoría de lenguajes de programación es una rama muy importante de las ciencias de la computación. Dentro de los campos de estudio que ésta cubre, uno de los más importantes es la teoría de los compiladores que es la base formal sobre la escritura de compiladores (o más generalmente traductores).

Estos son programas que traducen un programa escrito en un lenguaje a otro. Las acciones de un compilador se dividen generalmente en análisis sintáctico (escanear y analizar), análisis semántico (determinando que es lo que debería de hacer un programa), optimización (mejorando el rendimiento indicado por cierta medida, típicamente la velocidad de ejecución) y generación de código (generando la salida de un programa equivalente en el lenguaje deseado. A menudo el conjunto de instrucciones de una CPU). Un compilador se compone de:

- Un analizador léxico que crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico.
- El analizador sintáctico (parser) que convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada.

Para este trabajo práctico nos centraremos en el uso y testeo de una herramienta de parsing para traducir JSON a YAML. La herramienta elegida es $antlr$.

$antlr$ utiliza la técnica ELL(k) tanto para el análisis léxico como para el sintáctico, generando parsers recursivos-iterativos.

2. Lenguajes

2.1. JSON

JavaScript Object Notation es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

JSON está constituído por dos estructuras:

- Una colección de pares de nombre y valor. En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
- lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o sequencias.

2.2. YAML

Es un formato de serialización de datos legible por humanos inspirado en lenguajes como XML, C, Python, Perl, así como el formato para correos electrónicos especificado en RFC 2822. Mapeos y secuencias se pueden representar usando la misma notacion que en JSON, en ese caso se dice que usan el estilo $flow$. Pero también puede utilizarse el estilo $block$ que utiliza una linea por elemento, sin marcadores de comienzo y fin, donde el nivel de indentación de la linea marca a que estructura pertenece el elemento. Para este trabajo práctico utilizaremos el último estilo.

Veamos un ejemplo de cada lenguaje para mayor comprensión:

JSON:

```
[ { clave1: valor1, clave2: [ elem1, elem2 ], clave3: valor3 }, elem2, [ elem1, elem2, elem3 ] ]
```

YAML con estilo $block$ traducido del JSON anterior.

```
-
 - clave1: valor1
 - clave2:
     - elem1
     - elem2
 - clave3: valor3
- elem2
 -
  - elem1
  - elem2
  - elem3
```

3. Gramática y modificaciones

El primer paso para poder convertir de JSON a YAML es contar con la gramática de JSON para poder ingresarla en ANTLR y así generar el parser que reconocerá las cadenas JSON. Esta gramática puede ser encontrada en el sitio web oficial de JSON (https://www.json.org/json-es.html) y tiene las siguientes producciones:

- object → {} | { members }
- members → pair | pair, members
- pair → string : value
- array → [] | [elements]
- elements → value | value, elements
- value → string | number | object | array | true | false | null
- string → ” ” | ” chars ”
- chars → char | char chars
- char → any Unicode character except ” or \ or control character | \” | \\ | \/ | \b | \f | \n | \r | \t | \u four-hex-digits
- number → int | int frac | int exp | int frac exp
- int → digit | digit1-9 digits | - digit | - digit1-9 digits
- frac → . digits
- exp → e digits
- digits → digit | digit digits
- e → e | e+ | e- | E | E+ | E-

Dado que ANTLR acepta gramáticas ELL, se pueden utilizar expresiones regulares en las producciones. A su vez, cabe destacar que la gramática previamente enunciada no es LL(1) ya que posee conflictos (por ejemplo, la intersección entre Prim(array → []) y Prim (array → [elements]) no es vacía). Luego, puede ser reescrita aprovechando el poder expresivo de las expresiones regulares y evitando los conflictos pre-existentes (para que sea ELL) de la siguiente manera:

- object → { (} | members })
- members → pair (, members)?
- pair → string : value
- array → [( ] | elements])
- elements → value (, elements)?
- value → string | number | object | array | true | false | null
- string → ” char* ”
- char → any Unicode character except ” or \ or control character | \” | \\ | \/ | \b | \f | \n | \r | \t | \u four-hex-digits
- number → int (exp | frac (exp)?)?
- int → digit | digit1-9 digits | - (digit | digit1-9 digits)

- frac → . digits
- exp → e digits
- digits → (digit)+
- e → e (+ | -)? | E (+ | -)?

Como se puede observar, mediante la reescritura se reduce la cantidad de producciones. Particularmente, algunas de las producciones eliminadas (por ejemplo: chars → char | char chars) introducían conflictos que hacían que la gramática no sea LL(1).

Más allá de los cambios por expresiones regulares, el método más utilizado en la reescritura fue la factorización, por ejemplo en la siguiente producción:

- elements → value | value, elements

Pasó a ser:

- elements → value (, elements)?

En otros casos, se eliminó la recursión explotando los beneficios de las expresiones regulares, por ejemplo:

- digits → digit | digit digits

Pasó a ser:

- digits → (digit)+

Por último, para que ANTLR pueda aceptar la gramática, es necesario escribirla en el formato que ANTLR requiere y, en tal proceso, elegir cuales serán los tokens. A grandes rasgos, el proceso de reescritura consiste en escribir con minúsculas el nombre de los no terminales, con mayúsculas los nombres de los tokens y entre comillas simples los símbolos terminales. Luego, la gramática anterior puede ser reescrita de la siguiente manera (para el presente informe, se decidió dejar → en las producciones en lugar de ':' por facilidad de lectura):

- object → '{' ('}' | members '}');
- members → pair (',' members)?;
- pair → STRING ':' value;
- array → '[' (']' | elements ']');
- elements → value (',' elements)?;
- value → STRING | NUMBER | object | array | TRUE | FALSE | NULL;
- STRING →' " ' CHAR* ' " ';
- NUMBER → INTEGER (EXP | FRAC (EXP)?)?;
- fragment INTEGER → DIGIT | DIGITSNZ DIGITS | '-' (DIGIT | DIGITSNZ DIGITS);
- fragment FRAC → '.' DIGITS;
- fragment EXP → E DIGITS;
- fragment DIGITS → (DIGIT)+;
- fragment E → 'e' ('+' | '-')? | 'E' ('+' | '-')?;
- fragment CHAR → '\u0020'..'\u0021' | '\u0023'..'\u002F' | '\u003A'..'\u005B' | '\u005D'..'\uFFFE'| '\u0030'..'\u0039' | CTRL;

- fragment CTRL → ' \" ' | ' \\ ' | ' \/ ' | ' \b ' | ' \f ' | ' \n ' | ' \r ' | ' \t ' | ' \u ' FOURHEXDIGITS
- fragment DIGIT → '0'..'9';
- fragment DIGITNZ → '1'..'9';
- fragment FOURHEXDIGITS → '0' HEXDIGIT HEXDIGIT HEXDIGIT HEXDIGIT;
- fragment HEXDIGIT → DIGIT | 'a'..'f' | 'A'..'F';
- TRUE → 'true';
- FALSE → 'false';
- NULL → 'null';

Como se puede observar, para las producciones de la gramática se eligieron aquellas que definen la estructura de JSON mientras que para los tokens se eligieron aquellas que definen los valores específicos (como strings, números, booleanos, etc). Adicionalmente, fué necesario agregar algunas producciones para definir los dígitos, las constantes booleanas, los números hexadecimales de cuatro dígitos y los caracteres de control.

En algunas producciones se agregó el modificador "fragment". Los fragment sirven para construir las producciones de los tokens en base a ellos, sin embargo, no son considerados tokens. Por ejemplo, el token STRING está compuesto de caracteres (CHAR), que a su vez pueden ser caracteres de control (CTRL).

La gramática previamente enunciada es la versión final lista para ser utilizada en ANTLR. Esta gramática reconocerá satisfactoriamente cadenas de JSON, sin embargo, para poder llevar a cabo nuestro objetivo no alcanza únicamente con reconocerlas. Por lo tanto, en la siguiente sección se explicará cómo se puede llevar a cabo el proceso de traducción de JSON a YAML a partir de esta gramática.

4. Implementación

Para la implementación de la traducción de JSON a YAML se ha agregado traducción dirigida por sintáxis. Es decir, código asociado a cada producción para poder traducir el formato JSON a YAML introduciendo los espacios necesarios y además validar que no haya claves duplicadas en los diccionarios de JSON.

Para lograr el espacio necesario en la traducción dentro de las estructuras anidadas se utiliza un atributo heredado llamado $numSpaces$ del tipo int.

Este atributo se pasa como parámetro en todos los no terminales existentes. El inicializador de este atributo es un no terminal nuevo incluido en la grámatica llamado jsonToYaml que será la raiz de la grámatica en el parser. El valor de inicialización es cero.

```
jsonToYaml returns [int numSpaces]
    init {
        numSpaces = 0;
    }
    : value[numSpaces] EOF;
```

El no terminal object puede ser $\{\}$ o puede ser $\{members\}$. En el segundo caso se pasa a members por parámetro numSpaces + 1, ya que que las estructuras anidadas en members contendran un espacio.

EOF se agrega para el correcto funcionamiento de $antlr$.

```
object[int numSpaces]
    locals [
        Set<String> usedKeys = new HashSet<String>()
    ] :
    "{" ("}" { print("{}"); }
    | members[numSpaces + 1] "}");

members[int numSpaces]
    : pair[numSpaces] (',' members[numSpaces])?;
```

El Set $usedKeys$ se utiliza en el no terminal pair para controlar que no existan claves repetidas, dado que member es el no terminal que produce listas $clave : valor$.

```
pair[int numSpaces]
    init{
        int tab = numSpaces;
    } :
    { print("\n"); }
    STRING {
        String key;
        while(tab > 1){
            print(" ");
            tab--;
        }
        if ((STRING.text).charAt(1)=='-'||
        (STRING.text).contains("\\n")) {
            key = STRING.text;
        } else {
            key = (STRING.text).substring(1,
            (STRING.text).length() - 1);
        }
        print(key);
        if (object::usedKeys.contains(key)) {
            throw new Error(
                "Duplicated keys are not allowed: " + key
            );
        } else {
            object::usedKeys.add(key);
        }
    }
    ':' {print(": ");}
    value[numSpaces];
```

En el no terminal pair primero se imprime un fin de linea y luego se utiliza una variable tab inicializada con el valor de $numSpaces$ para imprimir tab cantidad de espacios para la clave. Luego pasa al no terminal value $numSpaces$ para seguir anidando los espacios.

Vale recalcar que el hecho de que la cantidad de espacios en cada nivel sea más fácil de calcular utilizando un atributo heredado fué determinante a la hora de elegir $antlr$ por su cualidad de utilizar algorítmos recursivos-iterativos. En otras herramientas como ply esta tarea sería más complicada, ya que al ser parsers bottom-up basados en lenguajes LALR, primero, habría que calcular la cantidad de espacios de cada token, y recién cuando el árbol haya sido calculado, printear toda la cadena con su correspondiente indentación.

Para el manejo de claves duplicadas, como se mencionó, se hace uso de $usedKeys$. Guardamos el string (valor text de STRING sin comillas) a imprimir en Formato YAML en una variable key y luego si la misma no está en el set, se agrega y si no se arroja un error por salida de errores standard.

Los arreglos pueden ser vacíos, o pueden contener elementos, en cuyo caso, el no terminal array le pasa a elements $numSpaces + 1$. Además, elements realiza el mismo procedimiento que pair, imprime un fin de linea y luego la cantidad de espacios indicada en $numSpaces$. Al final como el procedimiento es recursivo se pasa $numSpaces$ de nuevo a elements.

```
array[int numSpaces] :
    '[' (']' { print("[]"); } |
    elements[numSpaces + 1] ']');

elements[int numSpaces]
    init{
        int tab = numSpaces;
    } : {
        print("\n");
        while(tab > 1){
            print(" ");
            tab--;
        }
        print("- ");
    }
    value[numSpaces] (',' elements[numSpaces])?;
```

Por último, el no terminal value, puede ser STRING, NUMBER, otro object u otro array reenviando $numSpaces$, TRUE, FALSE o NULL. En el caso de STRING, se procesa el atributo text al igual que en pair, y se lo imprime.

```
value[int numSpaces] :
    STRING {
        if ((STRING.text).charAt(1) == '-' ||
        (STRING.text).contains("\\n")) {
            print(STRING.text);
        } else {
            print((STRING.text).substring(1,
            (STRING.text).length()-1));
        }
    }
    | NUMBER {print(NUMBER.text);}
    | object[numSpaces]
    | array[numSpaces]
    | TRUE { print("true"); }
    | FALSE { print("false"); }
    | NULL { print(""); };
```

5. Entrada para ANTLR

A continuación se presenta la entrada completa que será ingresada a ANTLR para generar automáticamente el código necesario para llevar a cabo la traducción:

```
grammar jsonGrammar;

@header {
    import java.util.Set;
        import java.util.HashSet;
}

jsonToYaml returns [int numSpaces]
    @init {
        $numSpaces = 0;
    } :
    value[$numSpaces] EOF;

object[int numSpaces]
    locals [
        Set<String> usedKeys = new HashSet<String>()
    ] :
    '{' ('}' { System.out.print("{}"); } | members[$numSpaces + 1] '}');

members[int numSpaces] :
    pair[$numSpaces] (',' members[$numSpaces])?;

pair[int numSpaces]
    @init {
        int tab = $numSpaces;
    } :
    { System.out.print("\n"); }
    STRING {
        String key;
        while(tab > 1){
            System.out.print(" ");
            tab--;
        }
        if (($STRING.text).charAt(1)=='-' || ($STRING.text).contains("\\n")) {
            key = $STRING.text;
        } else {
            key = ($STRING.text).substring(1, ($STRING.text).length() - 1);
        }
        System.out.print(key);
        if ($object::usedKeys.contains(key)) {
            throw new Error("Duplicated keys are not allowed: " + key);
        } else {
            $object::usedKeys.add(key);
        }
    }
    ':' { System.out.print(": "); }
    value[$numSpaces];

array[int numSpaces] :
    '[' (']' {System.out.print("[]");}|elements[$numSpaces + 1] ']');

elements[int numSpaces]
    @init {
        int tab = $numSpaces;
    } : {
        System.out.print("\n");
        while(tab > 1){
            System.out.print(" ");
            tab--;
        }
        System.out.print("- ");
    }
    value[$numSpaces] (',' elements[$numSpaces])?;
```

```
value[int numSpaces] :
    STRING {
        if (($STRING.text).charAt(1)=='-'||($STRING.text).contains("\\n")) {
            System.out.print($STRING.text);
        } else {
            System.out.print(($STRING.text).substring(1, ($STRING.text).length() - 1));
        }
    }
        | NUMBER {System.out.print($NUMBER.text);}
        | object[$numSpaces]
        | array[$numSpaces]
        | TRUE { System.out.print("true"); }
        | FALSE { System.out.print("false"); }
        | NULL { System.out.print(""); }
        ;

/*
 * Lexer Rules
 */

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines

STRING          : '"' (CHAR)* '"';
NUMBER          : INTEGER (EXP | FRAC (EXP)?)?;
TRUE: 'true';
FALSE: 'false';
NULL: 'null';

fragment INTEGER: DIGIT | DIGITNZ  DIGITS | '-' (DIGIT | DIGITNZ  DIGITS);
fragment FRAC   : '.' DIGITS;
fragment EXP    : E DIGITS;
fragment DIGITS : DIGIT+;
fragment E      : 'e' ('+' |'-')? | 'E' ('+' | '-')? ;
fragment DIGIT  : '0'..'9';
fragment DIGITNZ: '1'..'9';
fragment CHAR : '\u0020'..'\u0021' | '\u0023'..'\u002F' | '\u003A'..'\u005B' |
                '\u005D'..'\uFFFE' | '\u0030'..'\u0039' | CTRL;
fragment CTRL
  :  '\\"'
  |  '\\\\'
  |  '\\/'
  |  '\\b'
  |  '\\f'
  |  '\\n'
  |  '\\r'
  |  '\\t'
  |  '\\u'  HEXDIGIT HEXDIGIT HEXDIGIT HEXDIGIT
  ;
fragment HEXDIGIT: (DIGIT | 'a'..'f' | 'A'..'F');
```

6. Guía de uso

6.1. Contenido adjunto

El presente informe está acompañado por los siguientes ítems:

- Programa (.jar): Este .jar es el programa final que traducirá Json a Yaml (recibiendo como parámetro una única cadena Json). Es el resultado de compilar el código fuente provisto.
- Entrada para ANTLR (.g4): Este .g4 es el archivo que contiene la gramática de Json y las instrucciones necesarias para realizar la traducción. ANTLR toma este archivo y genera automáticamente el código.
- Código fuente
  - Código generado por ANTLR.
  - Código completo (el utilizado para manejar las funciones generadas por ANTLR junto con el código generado por ANTLR).
  - Proyecto para importar en Eclipse (este ítem es equivalente al anterior en cuanto a contenido de código).
- Biblioteca ANTLR: Versión 4.7.1 de ANTLR. Este .jar puede ser útil en caso de querer compilar el código fuente provisto.

6.2. Instrucciones (Linux)

6.2.1. Para utilizar el programa

Para utilizar el programa basta con escribir la siguiente instrucción en la terminal:
java -jar 'ruta al jar/jsonToYaml.jar' "cadena json a traducir"
Por ejemplo, si queremos traducir la cadena "[2424, 341231]", se puede utilizar el siguiente comando:
java -jar 'ruta al jar/jsonToYaml.jar' "[2424, 341231]"

Al ser un .jar autocontenido, no es necesario realizar ninguna otra acción, puede ser utilizado directamente.

6.2.2. Para generar el código utilizando ANTLR

Basta con ejecutar el siguiente comando en la terminal (estando ubicado en el directorio que contiene el archivo .g4):
antlr4 jsonGrammar.g4

Aclaración: Se asume que Antlr está correctamente instalado y configurado. En caso contrario, puede encontrarse un instructivo en la siguiente página web:
https://github.com/antlr/antlr4/blob/master/doc/getting-started.md .

6.2.3. Para compilar el código

Si se quiere compilar nuevamente el .jar provisto junto con este informe, hay dos opciones posibles:

- Utilizando Eclipse:
  - Importar el archivo comprimido correspondiente al proyecto.
  - Añadir la biblioteca de ANTLR al proyecto.
  - Compilar y exportar el .jar.
- Compilando el código manualmente desde la terminal.

7. Casos de prueba

Debemos ser capaces de reconocer las sentencias correctas y traducirlas efectivamente al formato yaml. De la misma manera, debemos asegurarnos que las impresiones inválidas sean rechazadas.

Para poder comprobar que cumplimos con estas características, y analizar los métodos que debimos utilizar y las dificultades con las cuales nos encontramos, es que mostraremos los siguientes ejemplos.

7.1. Expresiones válidas

7.1.1. Entrada objeto con ”: ” en la clave

Los chars en JSON incluyen todos los unicode que no son de control, comillas o contrabarra. También incluye el caracter ”:” que es la que nos diferencia la parte de claves y de valores de pair. Nuestra gramática debe diferenciar cuando este character se encuentra dentro de la clave o está actuando de separador.

Sea la entrada: {”ejempl:o1:” : 1, ”:”:”:”} la salida de nuestro programa es:

```
ejempl:o1:: 1
:: :
```

Las leyes léxicas de los Strings los diferencian por paridad de comillas en JSON, por lo que se puede distinguir cual ”:” es parte de la clave/valor y cual es el separador. Al traducirse no se las pone porque la representación de string en YAML no las incluye. Los números también son caracteres, pero serán strings o números dependiendo de si están entre comillas o no respectivamente.

Cadenas como { ”hola”1: 5}, {”hola:1” } o incluso {hola:1} contienen sintaxis inválidas y no serán reconocidas por el parser. { ’hola’:1} si es una cadena válida de una clave con string ’hola’ y valor uno como número.

7.1.2. Entrada objeto con caracteres especiales de escape en la clave

Continuando con nuestra revisión de que las reglas del análizador léxico de los strings funcionan, ahora veremos que pasa cuando los strings contienen caracteres de escape o las especiales ”\” y ” ” ”. En JSON, cuando una de estos caracteres pertenece a un string este se diferencia con un ”\” adelante.

Siendo nuestra entrada { ” ejemplo\”” : 2, ”saltolinea\n” : ”\t”} la salida de nuestro programa será:

```
ejemplo\": 2
"saltolinea\n": \t
```

Como podemos ver, nuestro parser traduce bien estas cadenas, diferenciando los caracteres pertenecientes al string de los especiales mediante el \. La segunda clave de nuestro ejemplo está traducida entre comillas, esto es una identificación para no confundirla con código a seguir en YAML.

En la definición de nuestra gramática, debemos asegurar que antlr entienda que nos referímos a que debe interpretar los caracteres y no la acción que representan como caracteres (por ejemplo, si queremos usar \como parte del string). Esto se logra en antlr con dos \antes de un caractér ”x”. De esta manera, nos referimos a que imprima \x en lugar de realizar la acción \x. Por eso, al definir los fragmentos de control de la siguiente manera:

```
fragment CTRL
  :   '\\"'
  |   '\\\\'
  |   '\\/'
  |   '\\b'
  |   '\\f'
  |   '\\n'
  |   '\\r'
  |   '\\t'
  |   '\\u'  HEXDIGIT HEXDIGIT HEXDIGIT HEXDIGIT;
```

Nos asegura la correcta interpretación de los caracteres de control.

7.1.3. Ejemplo del enunciado

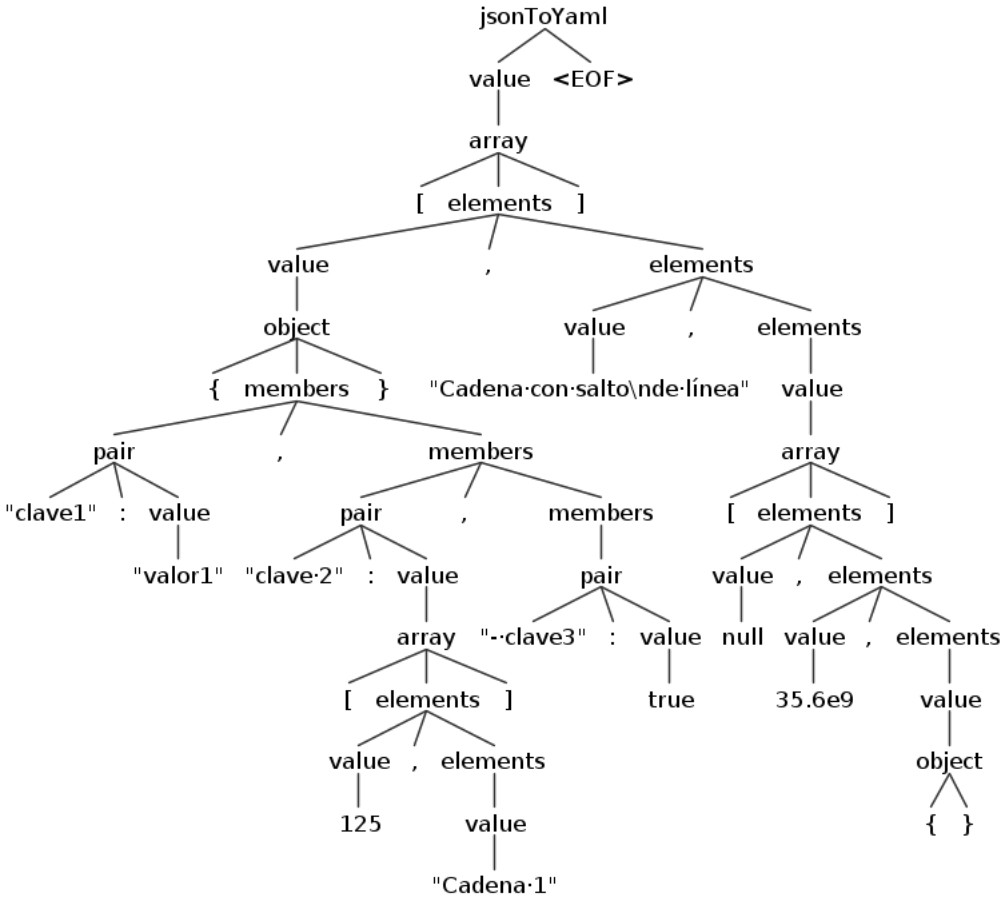
Aquí vemos que el ejemplo del enunciado funciona correctamente.
Siendo la entrada:

[{”clave1”: ”valor1”, ”clave 2”: [125, ”Cadena 1”], ”- clave3”: true},”Cadena con salto\nde linea”, [null, 35.6e9, {}]]

la salida de nuestro programa es :

```
-
 clave1: valor1
 clave 2:
  - 125
  - Cadena 1
 "- clave3": true
- "Cadena con salto\nde l nea"
-
 -
 - 35.6e9
 - {}
```

Como podemos ver la traducción del array funciona correctamente, incluyendo la indentación de las anidaciones. Siguiendo YAML, las ” en -clave3 y Cadenaconsalto\ndelinea son que se diferencien de la indentación. La siguiente imágen es el árbol de parsing generado para el input anterior.



7.1.4. Entradas con arreglos u objetos anidados

Como en el ejemplo anterior muestra, como nuestra gramática puede tener objetos anidados en objetos y viceversa, y el estilo *block* de YAML muestra estas anidaciones mediante la identación del objeto. Elegimos usar un atributo heredado para poder determinar la indentación con la que cada elemento debe mostrarse para cumplir este formato.

Para comprobar que funciona correctamente utilizaremos un ejemplo más extendido que el anterior, un objeto con anidación de hasta tres niveles, y su traducción.

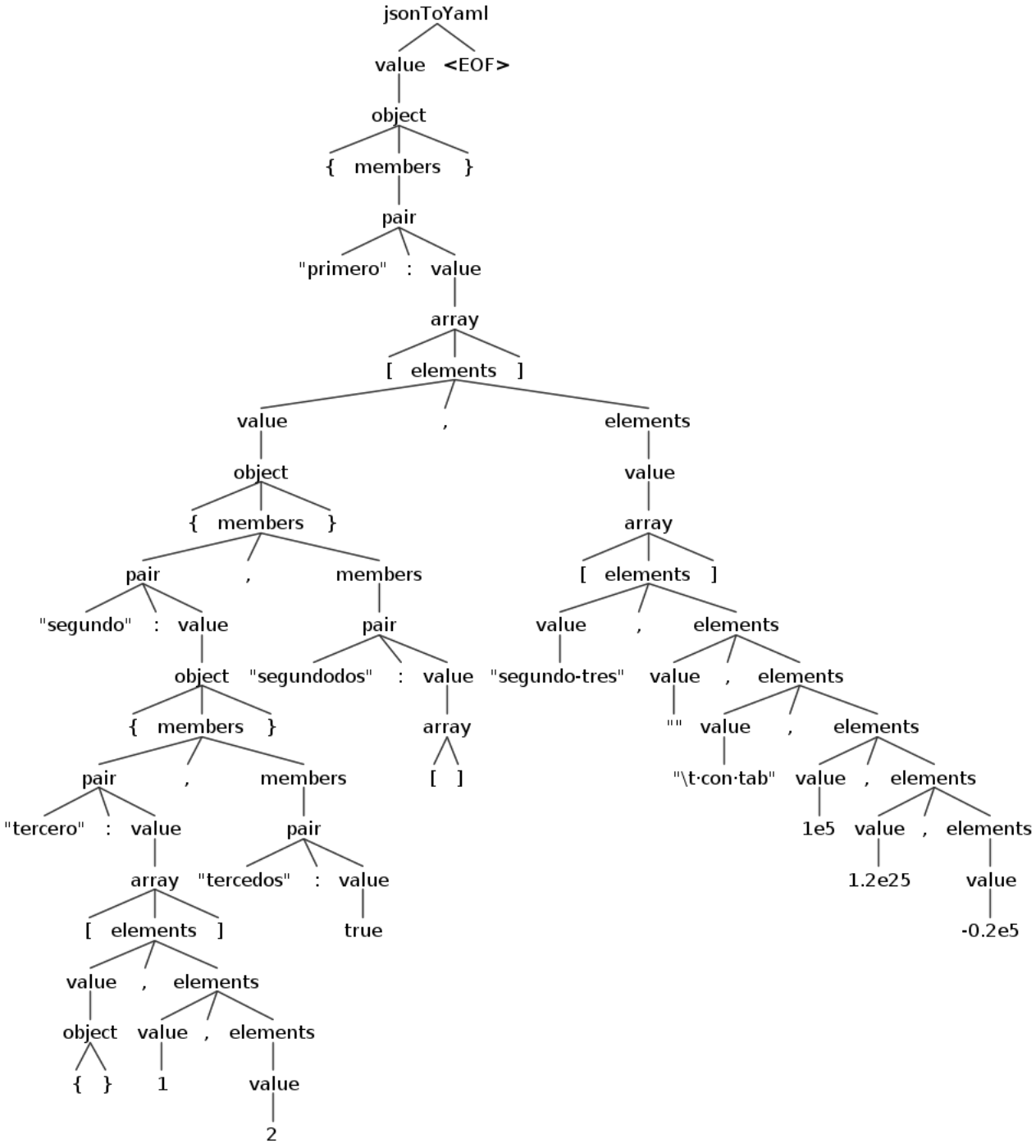
Sea esta la siguiente entrada:

{”primero”: [{”segundo” : { ”tercero”: [{}, 1, 2], ”tercedos”: true }, ”segundodos”: []}, [”segundotres”, ””, ”\t con tab”, 1e5, 1.2e25, -0.2e5]]}

se obtiene la siguiente salida:

```
primero:
 -
  segundo:
   tercero:
    - {}
    - 1
    - 2
   tercedos: true
  segundodos: []
 -
  - segundo-tres
  -
  - \t con tab
  - 1e5
  - 1.2e25
  - -0.2e5
```

Como podemos ver la indentacion es correcta. La diferencia en segundotres con respecto a segundodos y segundouno se debe a que la representación de los elementos de un array incluyen un "-", mientras que los de objet no lo incluyen, así que en la traducción están en el mismo nivel de indentación, con el guión identificando el tipo de valor. La anidación sigue las reglas de indentación sean arreglos anidados en objects o al revés. EL nivel de indentación está definido por un valor heredado *numSpaces*, que se pasa como parámetro y aumenta cada vez que se entra a un array o un object no vacío, evento en el cuál este nuevo valor estaría anidado en el anterior, por los que sus miembros requieren ser impresos con más indentación. El árbol de parsing generado es el siguiente:



7.2. Expresiones inválidas

Hay dos clases de errores que pueden aparecer. Errores sintácticos, es decir, entradas mal formadas, o que no cumplen alguna condición semántica de la gramática, en este caso, jsons con claves repetidas. Veremos ejemplos de ambas situaciones.

7.2.1. Expresiones con errores sintácticos

La correctitud de esta instancia depende de la correctitud de nuestra gramática y nuestra implementación de la misma. Si la implementación es correcta, el programa debería rechazar cadenas con errores sintácticos.

7.2.2. Ejemplo del enunciado

Siendo la entrada:

[{çlave1”: ”valor1”, çlave2”: [125, Çadena 1”}

la salida traducida será:

```
-
 clave1: valor1
 clave2:
  - 125
  - Cadena 1line 2:0 mismatched input '<EOF>' expecting ']'
```

El error se debe a que no hay corchete terminal para el arreglo, por lo que el arreglo no termina correctamente, y así nos lo informa el mensaje de error.

7.2.3. Ejemplos léxicos de strings

Anteriormente mencionamos unas cuantas cadenas de strings que no serían reconocidas por estar escritas incorrectamente, comprobemos que ese es el caso.

Sea la entrada: { ”hola”1: 5}, la salida es:

```
holaline 1:8 extraneous input '1' expecting ':'
: 5
```

Sea la entrada: {”hola:1” }, la salida es:

```
hola:1line 1:9 mismatched input '}' expecting ':'
```

Sea la entrada {hola:1} , la salida es:

```
line 1:1 token recognition error at: 'h'
line 1:2 token recognition error at: 'o'
line 1:3 token recognition error at: 'l'
line 1:4 token recognition error at: 'a'
line 1:5 no viable alternative at input ':'
```

En los tres casos da error como debería, y nos especifica que se produjo.

En el primero, terminamos las comillas antes del string, por lo que el parser interpreta que la clave se acabó, pero no es así, por lo que el error sale cuando trata de parsear el 1, ya que luego de las comillas de cierre se espera un ”:”.

El segundo es debido a que pasamos solo un string en un objeto, que espera un pair, conformado por un string, ”:” y un value. Al no detectar el ”:” da error.

Por último, el tercero es debiso a que no le pasamos un string, así que al estar en un objeto que busca la primera parte de un pair (la clave), se traba porque no encuentra las ” que representan el inicio de un string en json.

7.2.4. Expresiones con claves repetidas

Por una restricción de enunciado se considera un error de semántica, es decir, no se cumple con las restricciones del traductor si hay claves duplicadas. Por lo que se debe rechazar entradas que contengan la misma clave en más de un miembro de una cadena JSON.

Para ello, en la implementación, el no terminal tiene un set local llamado $usedKeys$ de strings donde llevamos la cuenta de las claves entradas en un mismo diccionario y que actualizamos en los pairs del mismo.

Que sea local significa que al llamarlo el parser sabrá que nos referimos a la declaración más cercana en la cadena de producciones.

7.2.5. Ejemplo del enunciado

Si la entrada es la siguiente:

{”clave1”: ”valor1”, ”clave 2”: [125, ”Cadena 1”], ”clave 2”: true}

la salida es:

```
clave1: valor1
clave 2:
 - 125
 - Cadena 1
clave 2Exception in thread "main" java.lang.reflect.InvocationTargetException
...
Caused by: java.lang.Error:
Duplicated keys are not allowed: clave 2
...
```

Se arroja una excepción existente en el código para que descarte este tipo de cadenas. Por supuesto esto sucede si las dos claves se refieren al mismo diccionario. Si la entrada es, por ejemplo, de la forma:

[{ ”hola”: 1}, {”hola”: 2}]

la salida será:

```
-
 hola: 1
-
 hola: 2
```

No hay ningún problema ya que las claves se repiten en distintos diccionarios.

En el caso de que haya más de una repetición, por ejemplo, con la entrada {”hola”:1, ”hola”:2, ”hola”:3}, la salida también dará error.

Como podemos ver, la excepción ocurre cuando se detecta la primer clave repetida, lo que indica que nuestra implementación no ignora errores, si no que detiene la ejecución justo en el punto donde se detecta el problema. No acumula errores, pero nos permite tener más certeza en donde se produjo el el error.

8. Conclusiones

1. Separar los analizadores léxicos de los sintácticos nos permite mucha flexibilidad, y facilita la verificación de la correctitud. Además nos permite poder mejorar el lenguaje (o a traducción) eficientemente, ya que sólo con modificar lo deseado en la gramática y recompilar, se actualiza. En la versión realizada todo está en un sólo archivo de código por simplicidad, pero el analizador léxico y el sintáctico pueden estar separados, lo cual brindaría aún más eficacia y extensibilidad a las gramáticas.

2. El uso de atributos nos permite guardar información, que luego nos habilita a realizar procedimientos semánticos dentro de una gramática como por ejemplo la traducción realizada en este trabajo práctico.

3. $antlr$ puede ser difícil de entender por la poca documentación existente. Sobre todo a la hora de agregar acciones semánticas en la gramática. Pero es una herramienta potente que permite reconocer gramáticas ELL(k) con acciones semánticas de manera sencilla una vez que entendemos como funciona.

4. El conocimiento de los algoritmos de parsing y los distintos tipos de gramáticas para realizar herramientas de traducción son esenciales para este tipo de problemas. Eliminando la necesidad de tener que programar los algoritmos por nosotros mismos y sólo tener que pensar en cuál es la traducción deseada si además ya poseemos las reglas sintácticas de la gramática.