uc3m | Universidad Carlos III de Madrid

P1 - Lex & Yacc

Procesadores del Lenguaje

Fecha 10/03/2024

Curso III

Cuatrimestre 2º

Año 2023-2024

Titulación

Grado en Ingeniería Informática

Profesor

Victor Granados Harinero – vigranad@inf.uc3m.es

Integrantes

Santiago Kiril Cenkov Stoyanov

100472051 – **100472051@alumnos.uc3m.es**

Adrián Ruiz Albertos

100363617 – **100363617@alumnos.uc3m.es**

Repositorio

https://github.com/SanKiril/acad-Lang_Proc

Contenidos

Descripción de la práctica

La primera práctica consistió en diseñar e implementar un analizador léxico y sintáctico de AJSON (Almost a JavaScript Object Notation), un formato de fichero personalizado basado en JSON. El analizador utiliza los módulos `lex` y `yacc` de la librería Python **PLY (Python Lex-Yacc)**.

Analizador léxico

El analizador léxico está implementado en el fichero `ajson_lexer.py` como la clase `AJSONLexer` y emplea el módulo `lex` para el análisis léxico. Este debe reconocer:

Números

Se utiliza un *token* para cada uno de los siguientes tipos de números:

Enteros

Nombre: **`INTEGER`**

Características:

1. Positivos implicitamente o negativos explicitamente: `r'\-?'`
2. Sin ceros al principio: `r'([1-9]\d*|0)'`
3. Valor: entero, base 10.

Expresión regular:

```
r'\-?([1-9]\d*|0)'
```

Reales

Nombre: **`REAL`**

Características:

1. Positivos implicitamente o negativos explicitamente: `r'\-?'`
2. Con o sin parte entera: `r'([1-9]\d*|0)?'`
3. Sin ceros al principio de la parte entera: `r'([1-9]\d*|0)'`
4. Con separador decimal: `r'\.'`
5. Con parte decimal: `r'\d+'`
4. Valor: coma flotante de simple precisión.

Expresión regular:

```
r'\-?([1-9]\d*|0)?\.\d+'
```

Notación científica

Nombre: **`SCIENTIFIC`**

Características:

1. Positivos implicitamente o negativos explicitamente: `r'\-?'`
2. Con o sin parte entera: `r'([1-9]\d*|0)?'`
3. Sin ceros al principio de la parte entera: `r'([1-9]\d*|0)'`
4. Con o sin separador decimal y parte decimal: `r'(\.\d+)?'`
5. Con separador exponencial: `r'[eE]'`
6. Con parte exponencial: `r'\-?([1-9]\d*|0)'`
7. Sin ceros al principio de la parte exponencial: `r'([1-9]\d*|0)'`
8. Parte exponencial positiva implicitamente o negativa explicitamente: `r'\-?'`
9. Valor: aritmética de precisión arbitraria (empleando el objeto python `Decimal`).

Expresión regular:

```
r'\-?([1-9]\d*|0)?(\.\d+)?[eE]\-?([1-9]\d*|0)'
```

Binarios

Nombre: **`BINARY`**

Características:

1. Con identificador binario: `r'0[bB]'`
2. Con parte binaria: `r'[01]+'`
3. Valor: binario, base 2.

Expresión regular:

```
r'0[bB][01]+'
```

Octales

Nombre: **`OCTAL`**

Características:

1. Con identificador octal: `r'0'`
2. Con parte octal: `r'[0-7]+'`
3. Valor: octal, base 8.

Expresión regular:

```
r'0[0-7]+'
```

Hexadecimales

Nombre: **`HEXADECIMAL`**

Características:

1. Con identificador hexadecimal: `r'0[xX]'`
2. Con parte hexadecimal: `r'[0-9a-fA-F]+'`
3. Valor: hexadecimal, base 16.

Expresión regular:

```
r'0[xX][0-9a-fA-F]+'
```

Cadenas de caracteres

Se utiliza un *token* para cada uno de los siguientes tipos de cadenas de caracteres:

Explícitas

Nombre: **`STRING_EXPLICIT`**

Características:

1. Comienzan y terminan con un identificador de cadena: `r'\"[^\"\n\r]*\"'`
2. Con o sin parte de texto: `r'[^\"\n\r]*'`
3. No contienen identificadores de cadena o caracteres especiales de fichero en la parte de texto: `r'[^\"\n\r]'`

Expresión regular:

```
r'\"[^\"\n\r]*\"'
```

Implícitas

Nombre: **`STRING_IMPLICIT`**

Características:

1. Comienzan con una letra o carácter especial de cadena: `r'[a-zA-Z_]'`
2. Contienen o no caracteres alfanuméricos después del primer carácter: `r'\w*'`

Expresión regular:

```
r'[a-zA-Z_]\w*'
```

Booleanos

Se han implementado como *tokens* reservados **`'TR'`** (valor: *True*) y **`'FL'`** (valor: *False*) los cuales se reconocen tanto en minúsculas como en mayúsculas.

Valor nulo

Se ha implementado como un *token* reservado **`'NULL'`** (valor: *None*) el cual se reconoce tanto en minúsculas como en mayúsculas.

Operadores de comparación

Nombre: **`COMPARATOR`**

Expresión regular:

```
r'\=\=|\>\=|\>|\<\=|\<'
```

El orden de cada comparador dentro de la expresión regular es importante (por ejemplo si se cambiara a: `r'...\>|\>\=...'`, no se reconocería `'>='` porque primero se reconoce `'>'`).

Delimitadores

Se han implementado como literales (**`'{'`**, **`'}'`**, **`'['`**, **`']'`**, **`':'`** y **`','`**) por

simplicidad.

Analizador sintáctico

El analizador sintáctico está implementado en el fichero `ajson_parser.py` como la clase `AJSONParser` y emplea el módulo `yacc` para el análisis sintáctico. Este debe reconocer:

1. Un fichero AJSON vacío o con un único objeto AJSON.
2. Objetos AJSON delimitados por `'{'` y `'}'`; vacíos (`{}` = None) o no.
3. Objetos AJSON no vacíos formados por pares clave-valor delimitados entre sí por `':'` y delimitados entre ellos por `','`, opcional para el último par.
4. Claves formadas por una cadena de caracteres.
5. Valores formados por una cadena de caracteres explícita, un número, un booleano, un valor nulo, una comparación, un objeto AJSON o un *array*.
6. Arrays delimitados por `'['` y `']'`; vacíos o no; formados por objetos AJSON y delimitados entre ellos por `','`, opcional para el último par.
7. Comparaciones formadas por un número, un operador de comparación y un número; serán evaluadas a *True* o *False*.

Gramática

Todo analizador sintáctico (analizador ascendente LALR en el caso de `yacc`) opera mediante una gramática la cual debe cumplir:

- Gramática independiente del contexto. Tipo 2 según la jerarquía de Chomsky.
- Sin ambiguedades.

La gramática empleada en el analizador sintáctico es:

(1): **`file ::=`**
`object | λ`
(2): **`object ::=`**
`'{' object_content '}'`
(2, 3): **`object_content ::=`**
`object_entry ',' object_content | object_entry | λ`
(3): **`object_entry ::=`**

```
     key ':' value
(4): key ::=
     STRING_EXPLICIT | STRING_IMPLICIT
(5): value ::=
     array | object | comparison | number | TR | FL | NULL |
     STRING_EXPLICIT
(6): array ::=
     '[' array_content ']'
(6): array_content ::=
     object ',' array_content | object | λ
(7): comparison ::=
     number COMPARATOR number
number ::=
     SCIENTIFIC | REAL | HEXADECIMAL | OCTAL | BINARY | INTEGER
```

Ejecución

Entrada

El analizador léxico y sintáctico son invocados por el programa principal implementado en el fichero `main.py`. La llamada al programa a través de la línea de comandos debe ser:

```
python3 ./main.py <path>.ajson -<mode>
```

- `<path>`: *ruta a un fichero AJSON*
- `<mode>`: *lex = analizador léxico || par = analizador léxico, sintáctico y semántico*

Salida

Analizador léxico y sintáctico

Empleando estructuras de datos de python durante la ejecución de las reglas gramaticales y los métodos `self.parse` y `self.__output` de la clase `AJSONParser` se consigue dar el siguiente formato a la salida:

- Para un fichero vacío u objeto AJSON que lo compone vacío: "`>>> EMPTY AJSON FILE <path>.ajson`"
- Para un fichero no vacío u objeto AJSON que lo compone no vacío: "`>>> AJSON`

`FILE <path>.ajson"`

- Para entradas de un objeto AJSON no anidado: `"{ clave: valor }"`
- Para entradas de un objeto AJSON anidado: `"{ clave_1.clave_2. ... .clave_n: valor }"`
- Para entradas de un array: `"{ clave_1.indice.clave_2. ... .clave_n: valor }"`

Analizador léxico

Empleando el método `self.tokenize` de la clase `AJSONLexer` se consigue dar el siguiente formato a la salida:

- El tipo de *token* a la izquierda.
- El valor del *token* a la derecha.

Pruebas

Descripción	Entrada	Salida
test_ok_tokens.ajson: tokens correctos son reconocidos e ignorados los espacios, tabulaciones y saltos de línea. *<mode> = -lex*	`-12` `-0` `0` `12` `-12.34` `-.34` `0.0` `.34` `12.34` `-12.34e56` `-12E56` `-.34e56` `0E0` `.34e-56` `12E-56` `12.34e-56` `0b0` `0B110` `00` `061` `0x0` `0Xa` `0xF1f` `"string explicit"` `string_implicit` `{}` `[]` `:` `,` `TR FL NULL tr fl null Tr Fl NuLl`	`INTEGER -12` `INTEGER 0` `INTEGER 0` `INTEGER 12` `REAL -12.34` `REAL -0.34` `REAL 0.0` `REAL 0.34` `REAL 12.34` `SCIENTIFIC -1.234E+57` `SCIENTIFIC -1.2E+57` `SCIENTIFIC -3.4E+55` `SCIENTIFIC 0` `SCIENTIFIC 3.4E-57` `SCIENTIFIC 1.2E-55` `SCIENTIFIC 1.234E-55` `BINARY 0` `BINARY 6` `OCTAL 0` `OCTAL 49` `HEXADECIMAL 0` `HEXADECIMAL 10` `HEXADECIMAL 3871` `STRING_EXPLICIT string explicit` `STRING_IMPLICIT string_implicit` `{ {` `} }` `[ [` `] ]` `: :` `, ,` `TR True` `FL False` `NULL None` `TR True` `FL False`

		NULL None TR True FL False NULL None
test_nok_tokens.ajson: tokens incorrectos no son reconocidos. *<mode> = -lex*	"incorrect string explicit" incorrect#string#implicit === => =<	ValueError: [ERROR][LEXER]: Illegal character: # PROVIDED: *<first_illegal_character>*
test_ok_empty_file.ajson: fichero vacío es reconocido. *<mode> = -par*		>>> EMPTY AJSON FILE ./tests/test_ok_empty_file.ajson
test_ok_empty_file_content.ajson: fichero con contenido vacío es reconocido. *<mode> = -par*	{}	>>> EMPTY AJSON FILE ./tests/test_ok_empty_file_content.ajson
test_ok_comparison.ajson: comparación entre números es evaluada. *<mode> = -par*	{ "comparison": 0xFF > 0b11101 }	>>> AJSON FILE ./tests/test_ok_comparison.ajson { comparison: True }
test_ok_empty_object.ajson: objeto vacío es reconocido y evaluado a None. *<mode> = -par*	{ empty_object: {} }	>>> AJSON FILE ./tests/test_ok_empty_object.ajson { empty_object: None }
test_ok_strings.ajson: cadenas de caracteres explícitas se muestran sin comillas. *<mode> = -par*	{ "string explicit": "hasn't quotes" }	>>> AJSON FILE ./tests/test_ok_strings.ajson { string explicit: hasn't quotes }
test_ok_reserved.ajson: palabras reservadas se convierte su valor. *<mode> = -par*	{ "TR": tr, "FL": fl, "NULL": null }	>>> AJSON FILE ./tests/test_ok_reserved.ajson { TR: True } { FL: False } { NULL: None }
test_ok_numbers.ajson:	{	>>> AJSON FILE

todos los números son reconocidos. *<mode> = -par*	integer: -12, real: -12.34 scientific: -12.34e56, binary: 0b110, octal: 061, hexadecimal: 0xF1F }	./tests/test_ok_numbers.ajson { integer: -12 } { real: -12.34 } { scientific: -1.234E+57 } { binary: 6 } { octal: 49 } { hexadecimal: 3871 }
test_ok_objects.ajson: objetos anidados y control de su ejecución. *<mode> = -par*	{ k1: { k2: "v_k2", k3: "v_k3", k4: { k5: "v_k5" } } }	>>> AJSON FILE ./tests/test_ok_objets.ajson { k1.k2: v_k2 } { k1.k3: v_k3 } { k1.k4.k5: v_k5 }
test_ok_arrays.ajson: arrays, control de su ejecución y comas al final de objetos o arrays. *<mode> = -par*	{ k1: [{k2: "v_k2"}, {k3: "v_k3"}, {k4: [{k5: "v_k5"},],},], }	>>> AJSON FILE ./tests/test_ok_arrays.ajson { k1.0.k2: v_k2 } { k1.1.k3: v_k3 } { k1.2.k4.0.k5: v_k5 }
test_nok_object.ajson: objeto sin cierre de llaves. *<mode> = -par*	{	ValueError: [ERROR][PARSER]: Not matching production rule: # PROVIDED: None
test_nok_array.ajson: array sin cierre de corchete. *<mode> = -par*	{ incorrect_array: [}	ValueError: [ERROR][PARSER]: Not matching production rule: # PROVIDED: }
test_nok_entry.ajson: entrada de objeto sin delimitador de dos puntos. *<mode> = -par*	{ entry "ko" }	ValueError: [ERROR][PARSER]: Not matching production rule: # PROVIDED: ko
test_nok_entries.ajson: entradas de objeto sin delimitador de coma. *<mode> = -par*	{ first_entry: "ok", second_entry: "ko" third_entry: "ok" }	ValueError: [ERROR][PARSER]: Not matching production rule: # PROVIDED: third_entry
test_nok_string_key.ajson: cadenas de caracteres implícitas no pueden ser	{ "string explicit": "ok", string_implicit:	ValueError: [ERROR][PARSER]: Not matching production rule:

valores. *<mode> = -par*	"ok", ko: string_implicit }	# PROVIDED: string_implicit