Proyecto del curso de Lenguajes Formales 2019

JSONQL

El proyecto final del curso de Lenguajes Formales de 2019 se trata de implementar el lenguaje *JSONQL*. Éste es un lenguaje inventado por la cátedra a los efectos del curso. Se trata de un lenguaje simple de consultas sobre un documento JSON. Se trata de un lenguaje conciso y algorítmico a usar por desarrolladores.

La cátedra de la asignatura se reserva el derecho de modificar y ajustar esta definición cuando y como considere conveniente.

Sintaxis

Las consultas siempre toman un documento JSON como referencia. Se tratan de expresiones que siempre tiene un valor JSON válido como resultado. Las operaciones se ordenan usando paréntesis curvos, al igual que en la mayoría de los lenguajes.

Operaciones compartidas con Javascript

Los siguientes ejemplos muestran el uso de operaciones aritméticas y lógicas que no usan el documento de referencia. Por esta razón se pone en `null`. JSONQL incluye mucha de la sintaxis de expresiones de Javascript. Los operadores de Javascript que se pueden usar en JSONQL

Referencia	Consulta	Resultado	Consulta	Resultado
`null`	`1`	`1`	`-1.2`	`-1.2`
`null`	`+3.4e-5`	`0.00034`	`0xC4f3`	`50419`
`null`	`1 + 2 * 3`	`7`	`1 / 2 - 3`	`-2.5`
`null`	`1 == 1.0`	`true`	`1 < 2`	`true`
`null`	`1 >= 2`	`false`	`1 > 2 && true`	`false`
`null`	`!true`	`false`	`false || 1 <= 2`	`true`
`null`	`"x"`	`"x"`	`"x" + "x"`	`"xx"`
`null`	`"x" != "x" + "x"`	`true`	`+"12"`	`12`

JSONQL también toma de Javascript la sintaxis para literales de *expresión regular*, y agrega para actuar con éstas los operadores `~=` (encaja), `!~` (no encaja) y `~` (encaje).

Referencia	Consulta	Resultado
`null`	`/\d+/`	`/\d+/`
`null`	`/|*/`	*SyntaxError: invalid regular expression!*
`null`	`"12" ~= /\d+/`	`true`
`null`	`"ab" ~= /\d+/`	`false`
`null`	`"jj" !~ /\*+/`	`true`
`null`	`"++" !~ /\++/`	`false`
`null`	`1.2 ~ /\d+\.\d+/`	`["1.2"]`
`null`	`1.2 ~ /(\d+)\.(\d+)/`	`["1.2", "1", "2"]`

Además de lo anterior, JSONQL define las siguientes construcciones:

- `(valueTrue) if (cond) else (valueFalse)`: Expresión condicional, similar al operador ternario `(?:)` de Java y Javascript. La sintaxis está tomada de Python.
- `let var = (value) in (exp)`: Operador *let-in* inspirado en la programación funcional. Retorna la

evaluación de la expresión de la derecha, evaluada sobre un contexto que tiene la variable `var` asociada al valor dado.

- `func(arg1,arg2)`: Llamada a función. Las funciones disponibles las define el usuario del lenguaje a la hora de procesar la consulta.
- `(value) <- (array)`: Operador de pertenencia. Verifica si el valor está en el array.
- `(key) <- (object)`: Operador de pertenencia. Verifica si la clave está en el objeto.
- `(key) : (value) <- (object)`: Operador de pertenencia. Verifica si el par *clave:valor* está en el objeto.
- `(value) /\ (value)`: Operador de intersección. Puede operar sobre *arrays*, objetos o *strings*.
- `(value) \/ (value)`: Operador de unión. Puede operar sobre *arrays*, objetos o *strings*.

Indización

La expresión `$` retorna el documento completo de referencia de la consulta. Si el valor es de tipo `object`, `array` o `string` se pueden indizar sus miembros o elementos. La notación de la indización es similar a la de Javascript, poniendo el índice entre paréntesis rectos a la derecha del valor indizado. Al contrario que con Javascript, una indización con un índice no válido genera un error de ejecución. *JSONQL* no permite expresiones con valor no definido (o `undefined`).

Referencia	Consulta	Resultado
`{"x":1,"y":2}`	`$`	`{"x":1,"y":2}`
`{"x":1,"y":2}`	`$["x"]`	`1`
`{"x":1,"y":{"a":true,"b":false}`	`$["y"]`	`{"a":true,"b":false}`
`{"x":{"a":true,"b":false},"y":2}`	`$["x"]["a"]`	`true`
`{"x":{"a":true,"b":false},"y":2}`	`$["y"]["a"]`	*Error: Invalid index "a"!*
`{"x":{"a":true,"b":false},"y":2}`	`$["x"]["y"]`	*Error: Invalid index "y"!*
`[1,2,3]`	`$[1]`	`2`
`[1,2,3]`	`$[3]`	*Error: Invalid index "3"!*
`null`	`$`	`null`
`null`	`$[0]`	*Error: Invalid index "0"!*
`42`	`$`	`42`
`42`	`$["length"]`	*Error: Invalid index "length"!*
`["abc"]`	`$[0][1]`	`"b"`

Cuando se manejan *strings* como índices, se puede utilizar una sintaxis abreviada, utilizando el operador `..`

Referencia	Consulta	Resultado
`{"x":1,"y":2}`	`$.x`	`1`
`{"x":1,"y":{"a":true,"b":false}`	`$.y`	`{"a":true,"b":false}`
`{"x":{"a":true,"b":false},"y":2}`	`$.x.a`	`true`
`{"x":{"a":true,"b":false},"y":2}`	`$.y.a`	*Error: Invalid index "a"!*
`{"x":{"a":true,"b":false},"y":2}`	`$.x.y`	*Error: Invalid index "y"!*

Las indizaciones pueden ser múltiples, en cuyo caso se genera un *array* (o *string*) con los valores obtenidos en el orden dado. Esta sintaxis también permite definir rangos de índices numéricos, y usar índices negativos, de forma similar a *Python*.

Referencia	Consulta	Resultado	Consulta	Resultado
`{"x":1,"y":2}`	`$["x","y"]`	`[1,2]`	`$["y","x"]`	`[2,1]`

Referencia	Consulta	Resultado	Consulta	Resultado
`{"x":1,"y":2}`	`$["x","x"]`	`[1,1]`	`$["x","y","x"]`	`[1,2,1]`
`[1,2,3]`	`$[1,0]`	`[2,1]`	`$[0,0,0]`	`[1,1,1]`
`[1,2,3]`	`$[$[0]]`	`[2]`	`$[$[0],$[0]-1]`	`[2,1]`
`[1,2,3,4,5]`	`$[2:]`	`[3,4,5]`	`$[:2]`	`[1,2]`
`[1,2,3,4,5]`	`$[2:4]`	`[3,4]`	`$[-1,-3]`	`[5,3]`
`"Hello\tworld!"`	`$[0:5,-1]`	`"Hello!"`	`$[-1,6,-1]`	`"!\t!"`

Las indizaciones sobre objetos pueden usar expresiones regulares, combinadas con las notaciones anteriores.

Referencia	Consulta	Resultado
`{"x":1,"y":2,"a1":3,"a2":4,"?":null}`	`$[/\w\d/]`	`[3,4]`
`{"x":1,"y":2,"a1":3,"a2":4,"?":null}`	`$[/x|y/, /a./]`	`[1,2,3,4]`
`{"x":1,"y":2,"a1":3,"a2":4,"?":null}`	`$["?", /a./]`	`[null,3,4]`

Definiciones por comprensión

JSONQL permite definir objetos y arrays por comprensión. Los arrays por comprensión se definen entre paréntesis rectos, mientras que los objetos por comprensión entre llaves. La comprensión se separa en dos partes por el símbolo `\`: la *proyección* a la izquierda y la *fuente* a la derecha.

La proyección varía si se define un objeto o un array por comprensión. Para objetos se compone de uno o más pares de expresiones: primero *clave* y luego *valor*, separados por `:`. Para arrays se compone de una o más expresiones. Todos los valores que define la proyección forman parte de la definición al mismo nivel. Es decir que si la proyección tiene más de una componente, ésta no define una sublista, sino varios miembros de la misma definición.

La fuente es la misma en ambos tipos de definición por comprensión. Se compone de uno o más de dos tipos de construcciones: *orígenes* de datos y *restricciones*. Los orígenes de datos definen la iteración sobre un array u objeto, usando el operador `<-` de manera distinta. Las restricciones son condiciones (expresiones *booleanas*) que los datos deben cumplir para formar parte de la definición.

Referencia	Consulta	Resultado
`[1,2,3,4,5]`	`[x*x \ x <- $]`	`[1,4,9,16,25]`
`[1,2,3,4,5]`	`[x*2, x*3 \ x <- $]`	`[2,3,4,6,6,9,8,12,10,15]`
`[1,2,3]`	`[x+y \ x <- $, y <- $]`	`[2,3,4,3,4,5,4,5,6]`
`[1,2,3]`	`[x+y \ x <- $, x < 3, y <- $]`	`[2,3,4,3,4,5]`
`[1,2,3]`	`{"x"+ x: x \ x <- $}`	`{"x1":1,"x2":2,"x3":3}`
`[1,2]`	`{x: x, "2*"+ x: 2*x \ x <- $}`	`{"1":1,"2*1":2,"2":2,"2*2":4}`
`[1,2,3]`	`{"x": x \ x <- $}`	*Error: repeated key "x"!*
`{"x":1,"y":2}`	`[v \ k:v <- $]`	`[1,2]`
`{"x":1,"y":2,"z":3}`	`{v:k \ k:v <- $}`	`{"1":"x","2":"y","3":"z"}`
`{"x":1,"y":2,"z":3}`	`{v:k \ k:v <- $, v < 3}`	`{"1":"x","2":"y"}`
`{"x":1,"y":2,"z":3}`	`{v:k \ k:v <- $, v < 3, k > "x"}`	`{"2":"y"}`
`[1,2,3]`	`[v \ v <- $, false]`	`[]`
`{"x":1,"y":2,"z":3}`	`{v:k \ k:v <- $, false}`	`{}`
`null`	`[0 \ v <- []]`	`[]`
`null`	`{"a":"a" \ true}`	`{"a":"a"}`

Implementación

La implementación se realizará en *Java* usando CUP para construir el analizador sintáctico (o *parser*), y JFlex para el analizador léxico (o *lexer*). Para procesar JSON se utilizará el paquete `org.json` (jar, git).

Representación de JSON

Los valores JSON se representarán con los siguientes tipos de datos de Java:

JSON:	boolean	number	string	array	object
Java:	`Boolean`	`Double`	`String`	`List<Object>`	`Map<String,Object>`

A estos efectos se define la interfaz `JSONHandler`, con dos métodos sobrecargados:

- `parse` recibe un texto en JSON como `String`, `Reader` o `InputStream`; y retorna un `Object` con el valor que representa.
- `stringify` recibe un `Object` con un valor JSON y lo convierte a un `String` en JSON.

La primera realización de esta interfaz será usando la implementación de `org.json.simple`.

Interfaces

La clase principal de la implementación deberá realizar la interfaz `JSONQLBuilder`, la cual tendrá los métodos:

- `parse` toma un texto JSONQL (un `String`, `Reader` o `InputStream`) y retorna un valor JSON que representa el AST del código de la consulta.
- `build` para construir consultas en base a . La consulta que se construye deberá realizar la interfaz `JSONQLQuery`.

La interfaz `JSONQLQuery` tiene los métodos:

- `run` que toma una valor JSON (de tipo `Object`) de referencia, y retorna otro valor JSON (también de tipo `Object`) o arroja una excepción del tipo `JSONQLRuntimeException`.
- `getHandler` y `setHandler` son el selector y modificador de la propiedad `handler` de tipo `JSONHandler`.
- `parseAndRun` toma un texto JSON (como `String`, `Reader` o `InputStream`), obtiene el valor JSON que representa y lo usa para ejecutar la consulta con éste como referencia. El resultado es un valor JSON (de tipo `Object`) o una excepción de tipo `JSONQLRuntimeException`.

Trabajo y entrega

El trabajo se realizará desde el 20 al 4 de julio. La entrega será el 4 de julio a las 21 horas. Se tomará una defensa breve vía Webasignatura ese mismo día inmediatamente luego de la entrega.

La cátedra armará un repositorio en GitHub para el proyecto. Los estudiantes trabajarán sobre un *fork* de este proyecto, hasta el momento de la entrega. La entrega se realizará mediante un *pull request*.

Durante el transcurso del trabajo, los docentes irán agregando código al proyecto, en lo que respecta a la representación del código JSONQL y su ejecución. Los estudiantes deberán actualizar su repositorio con las actualizaciones de los docentes. Si los estudiantes desean colaborar con el trabajo de los docentes, esto deberá hacerse en otros *pull requests*.

Extras

Las siguientes extensiones se pueden implementar como extras al proyecto de curso.

- Construir un parser de JSON especifico para la implementación de JSONQL, haciendo innecesaria la dependencia con `org.json`.
- Construir un parser de JSON extendido, que pueda procesar los valores numéricos `Infinity` o `+Infinity`, `-Infinity` y `NaN`. Agregar a JSONQL la posibilidad de manejar estos valores.
