Laboratorio de Programación Funcional 2025

1 Introducción

El objetivo de este laboratorio es implementar una biblioteca de manipulación y uso de valores JSON.

JSON (acrónimo de JavaScript Object Notation) es un formato de texto plano usado para representar datos. Es un subconjunto de la notación literal de objetos de JavaScript. Un valor JSON puede consistir en:

- Valores nulos.
- Números. Para este laboratorio consideraremos únicamente números enteros.
- Cadenas de caracteres.
- Booleanos, con sus posibles valores `true` y `false`.
- Arreglos. Listas no vacías de valores JSON. JSON es un lenguaje no tipado, en particular los arreglos pueden ser heterogéneos, conteniendo valores de distintos tipos.
- Objetos. Mapeos no vacíos clave/valor donde las claves son cadenas de caracteres, y los valores son valores JSON. Idealmente las claves son únicas, aunque las implementaciones admiten claves duplicadas.

La sintaxis de los valores JSON se define mediante la siguiente notación EBNF:

⟨*json*⟩ ::= ⟨*number*⟩ | ⟨*string*⟩ | ⟨*boolean*⟩ | '`null`' | ⟨*object*⟩ | ⟨*array*⟩

⟨*boolean*⟩ ::= '`true`' | '`false`'

⟨*object*⟩ ::= '`{`' ⟨*key*⟩ '`:`' ⟨*json*⟩ { '`,`' ⟨*key*⟩ '`:`' ⟨*json*⟩ } '`}`'

⟨*key*⟩ ::= ⟨*string*⟩

⟨*array*⟩ ::= '`[`' ⟨*json*⟩ { '`,`' ⟨*json*⟩ } '`]`'

donde $\langle string \rangle$ y $\langle number \rangle$ corresponden a cadenas de caracteres (entre comillas) y enteros, respectivamente.
El siguiente es un ejemplo de valor JSON:

```
{
  "nombre" : "Haskell SRL",
  "empleados": [
    { "nombre":"Juan",  "apellido":"Perez" },
    { "nombre":"Ana",   "apellido":"Rodriguez" },
    { "nombre":"Pedro", "apellido":"Gonzalez" },
    { "nombre":"Luisa", "apellido":"Gomez" }
  ],
  "numEmpleados" : 4,
  "enRUPE" : false,
  "expedientes_asociados" : null,
  "contactos" : [0303456,"curry@haskell.com.uy"]
}
```

La biblioteca implementa las siguientes características:

- El tipo `JSON` y funciones asociadas para manipular valores de tipo JSON.
- Una noción de tipado para objetos JSON.

2 Archivos distribuidos

La distribución consiste en cinco módulos:

- `AST.hs`

  Módulo que implementa un árbol de sintaxis abstracta para representar valores JSON en Haskell. Se define también una función `importJSON` para leer un JSON desde un archivo.

- `ParserCombinators.hs`

  Provee utilidades para realizar parsing. Se utilizan para construir la instancia de `Read` para el tipo `JSON`. **Este módulo no se entregará, por lo que no debe modificarse**.

- `JSONLibrary.hs`

  Módulo principal al que accederán los usuarios de la biblioteca. Implementa funciones para manipular valores JSON y exporta únicamente los nombres de `AST.hs` que se desean hacer visibles (el tipo `JSON`, sin sus constructores.

- `TypedJSON.hs`

  Define una noción de tipado sobre objetos JSON. Se define un tipo de datos para representar los tipos de valores JSON, así como funciones para realizar chequeo e inferencia de tipos sobre las representaciones de los valores JSON.

- `Estudiantes.hs`

  Contiene ejemplos de uso de la biblioteca desde la perspectiva de los clientes.

Los módulos públicos de la biblioteca son `JSONLibrary` y `TypedJSON`. En particular notar que `AST` es interno, por lo que no es accesible desde el módulo `Estudiantes`.

A continuación, se describe cada módulo con mayor nivel de detalle.

2.1 Módulo `AST`

En este módulo se define el siguiente tipo de datos, que se usa para representar internamente valores JSON:

```
data JSON
    = JString  String
    | JNumber  JSONNumber
    | JObject  (Object JSON)
    | JArray   Array
    | JBoolean Bool
    | JNull

type Object a = [(Key, a)]
type Array  = [JSON]
type Key    = String
type JSONNumber = Integer
```

En este módulo se define también el parser para construir valores de tipo `JSON` a partir de cadenas de caracteres con formato JSON, y otras utilidades (instancias de `Read`, `Show`, etc).

2.2 Módulo `JSONLibrary`

Este módulo define funciones para manipular valores JSON. Dado que los clientes de la biblioteca no tendrán acceso a los constructores del tipo `JSON` (por lo que el tipo es abstracto) las funciones implementadas aquí forman la interfaz pública para crear y manipular valores JSON.

Funciones principales

- `lookupField :: JSON -> Key -> Maybe JSON`

  Cuando el primer argumento es un objeto y tiene como clave el valor dado como segundo argumento, entonces se retorna el valor JSON correspondiente (bajo el constructor `Just`). De lo contrario se retorna `Nothing`. Si un objeto tiene claves repetidas, se retorna el valor de más a la derecha.

  Ejemplos:

  ```
  lookupField JNull "k" ==> Nothing
  lookupField (JObject [("k'", t)]) "k" ==> Nothing
  lookupField (JObject [("k", t)]) "k" ==> Just t
  lookupField (JObject [("k", t),("k", u)]) "k" ==> Just u
  ```

- `lookupFieldObj :: Object a -> Key -> Maybe a`

  Análoga a la anterior, pero el primer argumento es un objeto.

- `entriesOf :: Object a -> [(Key,a)]`

  Retorna todos los campos de un objeto, en el orden en que se encontraban.

- `keysOf :: Object a -> [Key]`

  Retorna la lista de claves de un objeto, manteniendo el orden en el que se encontraban.

- `valuesOf :: Object a -> [a]`

  Retorna una lista con los valores contenidos en los campos de un objeto, manteniendo el orden en el que se encontraban.

- Funciones con prefijo "`mk`".

  Funcionan como contructores. Por ejemplo `mkBoolean :: Bool -> JSON` construye un valor JSON booleano. Por ejemplo:

  ```
  mkBoolean True ==> JBoolean True .
  ```

- Funciones con prefijo "`from`".

  Funcionan como destructores. Por ejemplo, `fromJNumber :: JSON -> Maybe Integer` extrae el entero contenido en un valor JSON si este es un numeral, o devuelve `Nothing` en otro caso.

- Funciones con prefijo "`is`".

  Predicados que deciden qué tipo de valor JSON es el argumento. Por ejemplo:

  ```
  isArray JNull ==> False
  isArray (JArray []) ==> True
  ```

- `leftJoin :: Object a -> Object a -> Object a`

  Se combinan dos objetos en orden (primero los campos de la izquierda, luego los de la derecha). Se asume que los argumentos no tienen claves repetidas. El resultado tampoco debe tener claves repetidas. En caso que haya claves en común, en la unión tienen prioridad los campos del primer objeto. Por ejemplo, si:

  ```
  l = [("1", JNull),("3", JNull)]
  r = [("1", JBoolean True),("2", JNull)]
  ```

  Entonces:

  ```
  leftJoin l r ==> [("1", JNull),("3", JNull),("2", JNull)]
  ```

- `rightJoin :: Object a -> Object a -> Object a`

  Análoga a la anterior, pero en caso de claves en común, la prioridad la tienen los campos del segundo objeto.

- `filterArray :: (JSON -> Bool) -> Array -> Array`

  Dado un predicado sobre objetos JSON, y un arreglo, retorna un nuevo arreglo con los elementos que satisfacen el predicado.

- `insertKV :: (Key, a) -> Object a -> Object a`

  Se inserta un campo en un objeto. Se asume que las claves están ordenadas lexicográficamente, en caso de no cumplirse esto se inserta antes de la primera ocurrencia de una clave mayor (o al final si no hay una mayor).

- `consKV :: (Key, a) -> Object a -> Object a`

  Se inserta un campo en un objeto, al inicio.

- `sortKeys :: Object a -> Object a`

  Ordena las claves de un objeto. Es un ordenamiento estable (en caso de claves repetidas, se mantiene el orden lexicográfico relativo entre ellas).

2.3 Módulo `TypedJSON`

En este módulo se define una noción de tipado sobre valores de tipo JSON. Un valor está **bien tipado** de acuerdo a esta definición, si se cumplen las siguientes condiciones:

1. Ningún objeto tiene claves repetidas.
2. Los arreglos son **homogéneos** (los valores contenidos son todos del mismo tipo) y **no vacíos**.

Valores bien tipados La siguiente estructura de datos define los tipos JSON:

```
data JSONType
  = TyString
  | TyNum
```

```
  | TyObject (Object JSONType)
  | TyArray JSONType
  | TyBool
  | TyNull
  deriving (Show, Eq)
```

En el caso de objetos el tipo JSON representa la estructura del objeto, pero con sus claves **ordenadas**. Notar que si reordenamos las claves de un objeto bien tipado, su tipo no cambia.

A continuación se muestran valores bien tipados, con su respectiva representación del tipo.

- `JBoolean True` tiene tipo `TyBool`
- `JArray [JObject [("key 1", JNumber 1)]`

  tiene tipo

  `TyArray (TyObject [("key 1", TyNumber)])`
- `JArray [JNull, JNull]` tiene tipo `TyArray TyNull`
- `JArray [JString ""]` tiene tipo `TyArray TyString`
- ```
  JArray [JObject [("k", JArray [JBoolean True])],
          JObject [("k", JArray [JBoolean False, JBoolean True])]]
  ```

  tiene tipo

  `TyArray (TyObject [("k", TyArray TyBool)])`
- ```
  JArray [JObject [("a", JBoolean True), ("b", JArray [JNull])],
          JObject [("b", JArray [JNull]), ("a", JBoolean False)]]
  ```

  tiene tipo

  `TyArray (TyObject [("a", TyBool), ("b", TyArray TyNull))])`

Valores mal tipados Los siguientes valores están mal tipados:

1. `JArray [JNull, JObject [("key 1", JNumber 1)]`
2. `JArray [JObject [("key 1", JNumber 1)], JObject [("key 2", JNumber 1)]]`
3. `JObject [("key 1", JNumber 1), ("key 1", JString "")]`

4. `JArray [JObject [("key 1", JArray [JNumber 0])],`
   `JObject [("key 1", JBoolean True)]]`

5. `JArray [JObject [("key 1", JArray []),("key 2", JBoolean True)],`
   `JObject [("key 2", JBoolean False), ("key 1", JArray [JNull])]]`

porque:

1. Es un arreglo heterogéneo. El primer elemento es de tipo nulo, el segundo un objeto.

2. Es un arreglo heterogéneo. Contiene dos objetos con distinto tipo, pues tienen distintas listas de claves.

3. Es un objeto con claves repetidas.

4. Es un arreglo heterogéneo, porque los dos objetos tienen distinto tipo (sus campos tienen igual clave, pero sus valores son de distinto tipo).

5. Es un arreglo heterogéneo. Si bien los dos objetos tienen el mismo conjunto de claves `JArray []` está mal tipado.

Funciones principales:

- `typeOf :: JSON -> Maybe JSONType`

  Dado un valor JSON, retorna su tipo, retornando `Nothing` en caso de que el valor esté mal tipado.

- `objectWf :: Object JSONType -> Bool`

  Decide si un tipo de objeto está bien formado. Esto es, que en el mismo las claves estén ordenadas lexicográficamente y no se dupliquen. Esta función solo inspecciona las claves "externas". Si el tipo objeto contiene otros objetos, no se chequean.

- `typeWf :: JSONType -> Bool`

  Decide si un tipo está bien formado.

- `hasType :: JSON -> JSONType -> Bool`

  Dado un valor JSON y un tipo t, decide si el valor tiene el tipo t.

2.4 Módulo `Estudiantes`

En este módulo se implementan ejemplos de uso de la biblioteca desde la perspectiva de los clientes. Un sistema ya desarrollado representa la información de los estudiantes de la facultad mediante un objeto JSON con tipo `tyEstudiante`, definido como sigue:

```
tyEstudiante =
  TyObject [ ("nombre",   TyString),
             ("apellido", TyString),
             ("CI",       TyNum),
             ("cursos",   TyArray tyCurso) ]

tyCurso =
  TyObject [("nombre",   TyString),
            ("codigo",   TyNum),
            ("anio",     TyNum),
            ("semestre", TyNum),
            ("nota",     TyNum)]
```

Un valor concreto válido podría ser:

```
{ "nombre" : "Ana",
  "apellido" : "Suarez",
  "CI" : 12345678,
  "cursos" :
     [ { "nombre" : "Calculo DIV",
         "codigo" : 123,
         "anio" : 2024,
         "semestre" : 1,
         "nota" : 1 },
       { "nombre" : "Calculo DIV",
         "codigo" : 123,
         "anio" : 2019,
         "semestre" : 2,
         "nota" : 7 },
       { "nombre" : "Programacion 1",
         "codigo" : 234,
         "anio" : 2019,
         "semestre" : 2,
         "nota" : 7 } ] }
```

donde la lista de cursos va ordenada cronológicamente. Cuando dos cursos son del mismo semestre y del mismo año, se ordenan según el código.
Se deben implementar las siguientes funciones:

- `estaBienFormadoEstudiante :: JSON -> Bool`

  Decide si un valor JSON representa correctamente a un estudiante.

- Funciones con prefijo `get`.

  Dado un valor JSON que representa a un estudiante, retornan el campo correspondiente.

- `aprobados :: JSON -> Maybe JSON`

  Dado un valor JSON que representa a un estudiante, retorna el mismo valor pero con la lista de cursos aprobados (nota ≥ 3).

- `enAnio :: Integer -> JSON -> Maybe JSON`

  Dado un año y un valor JSON de estudiante, retorna el JSON con los cursos cursados ese año.

- `promedioEscolaridad :: JSON -> Maybe Float`

  Dado un valor JSON que representa a un estudiante, retorna el promedio de las notas de sus cursos.

- `addCurso :: Object JSON -> JSON -> JSON` Dado un objeto JSON que representa un curso, y un valor JSON que representa un estudiante, devuelve el estudiante con la lista de cursos actualizada agregando el nuevo curso.

3 Se pide

La tarea consiste en modificar los archivos:

- `AST.hs`
- `JSONLibrary.hs`
- `TypedJSON.hs`
- `Estudiantes.hs`

implementando las funciones solicitadas (todas aquellas que aparecen definidas como `undefined`), de manera que la biblioteca se comporte como se describe en este documento. Dentro de los archivos se pueden definir las funciones auxiliares que se consideren necesarias. No se deben modificar las signaturas de las funciones ni los demás módulos.

Dado que el equipo docente considera que el uso de IA en esta unidad curricular entorpece u obstaculiza el proceso de aprendizaje, no se permite utilizar IA generativa para resolver el laboratorio.