UNIVERSIDAD COMPLUTENSE MADRID

APLICACIONES WEB
2018-2019
Grado en Ingeniería del Software
Facultad de Informática

JavaScript

Ejercicios

Ejercicio 1

Escribir una función `producto` que reciba dos parámetros (llamados `x` e `y`) y devuelva su producto, teniendo en cuenta que tanto la `x` como la `y` pueden ser números o vectores (representados como arrays). La función se comportará del siguiente modo:

- Si `x` e `y` son números, se calculará su producto.
- Si `x` es un número e `y` es un vector (o viceversa), se calculará el vector que resulta de multiplicar todas los componentes de `y` por `x`.
- Si `x` e `y` son vectores de la misma longitud, se calculará el producto escalar de ambos.
- En cualquier otro caso, se lanzará una excepción.

Ejercicio 2

Escribir una función `sequence` que reciba un array de funciones `[f_1, ..., f_n]` y un elemento inicial `x`. La función debe aplicar `f_1` a x, y pasar el resultado a `f_2` que a su vez le pasará el resultado a `f_3` y así sucesivamente. Se piden tres versiones diferentes de la función `sequence`:

(a) Implementar la función `sequence1` suponiendo que ninguna de las funciones del array recibido devuelve el valor undefined.

(b) Implementar la función `sequence2` para que, en el caso en que una función del array devolviera el valor undefined, la función `sequence2` devuelva directamente undefined sin seguir ejecutando las funciones restantes.

(c) Implementar la función `sequence3` para que reciba un tercer parámetro opcional (`right`), cuyo valor por defecto será `false`. Si el parámetro right tiene el valor `true`, el recorrido del elemento por las funciones será en orden inverso: desde la última función del array hasta la primera. Independientemente del recorrido, la función `sequence3` se comportará como la función `sequence2`.
(NOTA: dentro de una función se puede comprobar que el último parámetro no está presente comparándolo con `undefined`).

Ejercicio 3

(a) Escribir una función **pluck(objects, fieldName)** que devuelva el atributo de nombre **fieldName** de cada uno de los objetos contenidos en el array **objects** de entrada. Se devolverá un array con los valores correspondientes. Por ejemplo:

```
let personas = [
{nombre: "Ricardo", edad: 63},
{nombre: "Paco", edad: 55},
{nombre: "Enrique", edad: 32},
{nombre: "Adrián", edad: 34}
];
pluck(personas, "nombre") // Devuelve: ["Ricardo", "Paco", "Enrique", "Adrián"]
pluck(personas, "edad") // Devuelve: [63, 55, 32, 34]
```

(b) Implementar una función **partition(array, p)** que devuelva un array con dos arrays. El primero contendrá los elementos x de **array** tales que p(x) devuelve true. Los restantes elementos se añadirán al segundo array. Por ejemplo:

```
partition(personas, pers => pers.edad >= 60)
// Devuelve:
// [
// [ {nombre: "Ricardo", edad: 63} ],
// [ {nombre: "Paco", edad: 55}, {nombre: "Enrique", edad: 32},
// {nombre: "Adrián", edad: 34} ]
// ]
```

(c) Implementar una función **groupBy(array, f)** que reciba un **array**, una función clasificadora **f**, y reparta los elementos del **array** de entrada en distintos arrays, de modo que dos elementos pertenecerán al mismo array si la función clasificadora devuelve el mismo valor para ellos. Al final se obtendrá un objeto cuyos atributos son los distintos valores que ha devuelto la función clasificadora, cada uno de ellos asociado a su array correspondiente. Ejemplo:

```
groupBy(["Mario", "Elvira", "María", "Estela", "Fernando"],
str => str[0]) // Agrupamos por el primer carácter
// Devuelve el objeto:
// { "M" : ["Mario", "María"], "E" : ["Elvira", "Estela"], "F" : ["Fernando"] }
```

(d) Escribir una función **where(array, modelo)** que reciba un **array** de objetos y un objeto **modelo**. La función ha de devolver aquellos objetos del **array** que contengan todos los atributos contenidos en **modelo** con los mismos valores. Ejemplo:

```
where(personas, { edad: 55 })
// devuelve [ { nombre: 'Paco', edad: 55 } ]

where(personas, { nombre: "Adrián" })
// devuelve [ { nombre: 'Adrián', edad: 34 } ]

where(personas, { nombre: "Adrián", edad: 21 })
// devuelve []
```

Ejercicio 4

Reescribir las funciones del ejercicio anterior pero utilizando exclusivamente funciones de orden superior.

Ejercicio 5

Escribir una función `concatenar` que permita concatenar un número no predefinido de cadenas de caracteres utilizando un carácter separador. El primer parámetro de la función es el carácter separador. Este parámetro va seguido de las cadenas que se concatenarán. Se espera el siguiente comportamiento de la función:

```
concatenar()          // Devuelve la cadena vacía
concatenar("-")       // Devuelve la cadena vacía
concatenar("-","uno")    // Devuelve "uno-"
concatenar("-","uno","dos","tres")    // Devuelve "uno-dos-tres-"
```

Ejercicio 6

Escribe una función **mapFilter(array, f)** que se comporte como **map**, pero descartando los elementos obj de la entrada tales que f(obj) devuelve undefined. Por ejemplo:

```
mapFilter(["23", "44", "das", "555", "21"],
(str) => {
let num = Number(str);
if (!isNaN(num)) return num;
})
// Devuelve: [23, 44, 555, 21]
```

Ejercicio 7

Utilizando expresiones regulares, implementar la función **interpretarColor(str)** que, dada una cadena que representa un color en formato hexadecimal #RRVVAA devuelva un objeto con tres atributos (**rojo**, **verde** y **azu**l) con el valor (en base 10) de la componente correspondiente.
Si la cadena de entrada no es un color HTML válido, se devuelve null.
Indicación: se puede utilizar **parseInt** para realizar conversiones entre distintas bases.