Programación funcional en términos simples

La primera regla de las funciones es que deben ser pequeñas. La segunda regla de las funciones es que deben ser más pequeñas que eso.

—Robert C. Martin

Bienvenido al mundo de la programación funcional, un mundo que solo tiene funciones, que viven felices sin ninguna dependencia del mundo exterior, sin estados y sin mutaciones, para siempre. La programación funcional es una palabra de moda en estos días. Es posible que hayas escuchado sobre este término dentro de tu equipo o en una reunión de grupo local. Si ya sabes lo que significa, genial. Para aquellos que no conocen el término, no se preocupen. Este pequeño documento está diseñado para presentarte los términos *funcionales* en un lenguaje simple.

Vamos a comenzar este documento con una pregunta simple: ¿Qué es una función en matemáticas? Más adelante, vamos a crear una función en JavaScript con un ejemplo simple utilizando nuestra definición de función. El documento termina explicando los beneficios que la programación funcional brinda a los desarrolladores.

¿Qué es la programación funcional? ¿Por qué es importante?

Antes de comenzar a explorar lo que significa la programación funcional, tenemos que responder otra pregunta: ¿Qué es una función en matemáticas? Una función en matemáticas se puede escribir así:

$$f(X) = Y$$

La declaración se puede leer como "Una función *f*, que toma *X* como argumento y devuelve la salida *Y*". *X* e *Y* pueden ser cualquier número, por ejemplo. Esa es una definición muy simple. Sin embargo, hay conclusiones clave en la definición:

- Una función siempre debe tomar un argumento.
- Una función siempre debe devolver un valor.
- Una función debe actuar solo sobre los argumentos que recibe (es decir, *X*), no sobre el mundo exterior.
- Para una *X* dada, solo habrá una *Y*.

Quizás te estés preguntando por qué presentamos la definición de función en matemáticas en lugar de en JavaScript. Esa es una gran pregunta.

La respuesta es bastante simple: las técnicas de programación funcional se basan en gran medida en funciones matemáticas y sus ideas. Sin embargo, aguanta la respiración; no vamos a enseñarte programación funcional en matemáticas, sino que usaremos JavaScript. Sin embargo, a lo largo del documento, veremos las ideas de las funciones matemáticas y cómo se utilizan para ayudar a comprender la programación funcional.

Con esa definición en su lugar, veremos los ejemplos de funciones en JavaScript. Imagina que tenemos que escribir una función que realice cálculos de impuestos. ¿Cómo vas a hacer esto en JavaScript? Podemos implementar una función como se muestra en el listado 1.

Listado 1: Función Calcular Impuesto.

```
var percentValue = 5;
var calculateTax = (value) => { return value/100 * (100 +
percentValue) }
```

La función calculateTax hace exactamente lo que queremos hacer. Puedes llamar a esta función con el valor, que devolverá el valor del impuesto calculado en la consola. Se ve bien, ¿no? Hagamos una pausa por un momento y analicemos esta función con respecto a nuestra definición matemática. Uno de los puntos clave de nuestro término de función matemática es que la lógica de la función no debe depender del mundo exterior.

En nuestra función calculateTax, hemos hecho que la función dependa de la variable global percentValue. Por lo tanto, esta función que hemos creado no se puede llamar como una función real en un sentido matemático. Vamos a solucionarlo.

La solución es muy sencilla: solo tenemos que mover percentValue como nuestro argumento de función, como se muestra en el listado 2.

Listado 2: Función calculateTax reescrita.

```
var calculateTax = (value, percentValue) => { return value/100 *
(100 + percentValue) }
```

Ahora nuestra función calculateTax se puede llamar como una función real. Sin embargo, ¿qué hemos ganado? Acabamos de eliminar el acceso a la variable global dentro de nuestra función calculateTax. Eliminar el acceso a variables globales dentro de una función facilita las pruebas. (Hablaremos de los beneficios de la programación funcional más adelante en este mismo documento).

Ahora hemos demostrado la relación entre la función matemática y nuestra función de JavaScript. Con este simple ejercicio, podemos definir la programación funcional en términos técnicos simples.

La programación funcional es un paradigma en el que crearemos funciones que desarrollarán su lógica dependiendo únicamente de su entrada. Esto garantiza que una función, cuando se llama varias veces, devolverá el mismo resultado. La función tampoco cambiará ningún dato en el mundo exterior, lo que genera una base de código que se puede almacenar en caché y probar.

FUNCIONES VS. MÉTODOS EN JAVASCRIPT

Hemos hablado mucho sobre la palabra función en este documento. Antes de continuar, queremos asegurarnos de que comprendas la diferencia entre funciones y métodos en JavaScript.
En pocas palabras, una función es un fragmento de código que se puede llamar por su nombre. Se puede utilizar para pasar argumentos sobre los que puede operar y devolver valores opcionalmente. Un método es un fragmento de código que se debe llamar por su nombre y que está asociado con un objeto.
Los listados 3 y 4 brindan ejemplos rápidos de una función y un método.

Listado 3: Una función simple.

```
var simple = (a) => {return a} // A simple function
simple(5) //called by its name
```

Lista 4: Un método simple.

```
var obj = {simple : (a) => {return a} }
obj.simple(5) //called by its name along with its associated
object
```

Hay dos características más importantes de la programación funcional que faltan en la definición. Las analizamos en detalle en las próximas secciones antes de profundizar en los beneficios de la programación funcional.

Transparencia referencial

Con nuestra definición de función, hemos hecho una declaración de que todas las funciones devolverán el mismo valor para la misma entrada. Esta propiedad de una función se denomina *transparencia referencial*. En el listado 5 se muestra un ejemplo sencillo.

Listado 5: Ejemplo de transparencia referencial.

```
var identity = (i) => { return i }
```

En el listado 5, hemos definido una función sencilla denominada identidad.
Esta función devolverá lo que esté pasando como entrada; es decir, si está pasando 5, devolverá el valor 5 (es decir, la función solo actúa como un espejo o identidad). Ten en cuenta que nuestra función opera solo en el argumento entrante *i*, y no hay ninguna referencia global dentro de nuestra función (recuerda que en el listado 2, eliminamos percentValue del acceso global y lo convertimos en un argumento entrante). Esta función satisface las condiciones de una transparencia referencial. Ahora imaginemos que esta función se utiliza entre otras llamadas de función como esta:

```
sum(4,5) + identity(1)
```

Con nuestra definición de transparencia referencial, podemos convertir esa declaración en esto:

```
sum(4,5) + 1
```

Ahora bien, este proceso se denomina *modelo de sustitución*, ya que se puede sustituir directamente el resultado de la función tal como está (principalmente porque la función no depende de otras variables globales para su lógica) por su valor.

Esto conduce al código paralelo y al almacenamiento en caché. Imaginemos que con este modelo se puede ejecutar fácilmente la función dada con varios subprocesos sin siquiera la necesidad de sincronizar. ¿Por qué? La razón para sincronizar proviene del hecho de que los subprocesos no deberían actuar sobre datos globales cuando se ejecutan en paralelo.

Las funciones que obedecen a la transparencia referencial van a depender únicamente de las entradas de su argumento; por lo tanto, los subprocesos pueden ejecutarse libremente sin ningún mecanismo de bloqueo.

Debido a que la función va a devolver el mismo valor para la entrada dada, podemos, de hecho, almacenarla en caché. Por ejemplo, imaginemos que hay una función llamada factorial, que calcula el factorial del número dado.

Factorial toma la entrada como su argumento para el cual se debe calcular el factorial. Sabemos que el factorial de 5 va a ser 120. ¿Qué pasa si el usuario llama al factorial de 5 una segunda vez? Si la función factorial obedece a la transparencia referencial, sabemos que el resultado va a ser 120 como antes (y solo depende del argumento de entrada). Con esta característica en mente, podemos almacenar en caché los valores de nuestra función factorial. Por lo tanto, si se llama a factorial por segunda vez con la entrada como 5, podemos devolver el valor almacenado en caché en lugar de calcularlo una vez más.

Aquí puede ver cómo una idea simple ayuda en el código paralelo y el código almacenable en caché. Más adelante en este documento, escribiremos una función en nuestra biblioteca para almacenar en caché los resultados de la función.

LA TRANSPARENCIA REFERENCIAL ES UNA FILOSOFÍA

La transparencia referencial es un término que proviene de la filosofía analítica (https://en.wikipedia.org/wiki/Analytical_philosophy). Esta rama de la filosofía se ocupa de la semántica del lenguaje natural y sus significados.

Aquí, la palabra referencial o referente significa aquello a lo que se refiere la expresión. Un contexto en una oración es referencialmente transparente si reemplazar un término en ese contexto por otro término que se refiere a la misma entidad no altera el significado.

Así es exactamente como hemos estado definiendo la transparencia referencial aquí. Hemos reemplazado el valor de la función sin afectar el contexto.

Imperativo, declarativo, abstracción

La programación funcional también se trata de ser *declarativo* y escribir código abstracto. Necesitamos entender estos dos términos antes de continuar. Todos conocemos y hemos trabajado en un paradigma imperativo.

Tomaremos un problema y veremos cómo resolverlo de manera imperativa y declarativa. Supongamos que tienes una lista o arreglo y deseas iterar a través del arreglo e imprimirlo(a) en la consola. El código podría parecerse al listado 6.

Listado 6: Iteración sobre el enfoque imperativo del arreglo.

```
var array = [1,2,3]
for(i=0;i<array.length;i++)
    console.log(array[i]) //prints 1, 2, 3
```

Funciona bien. Sin embargo, en este enfoque para resolver nuestro problema, le estamos diciendo exactamente "cómo" debemos hacerlo. Por ejemplo, hemos escrito un ciclo for implícito con un cálculo de índice de la longitud del arreglo e imprimiendo los elementos.

Nos detendremos aquí. ¿Cuál era la tarea aquí? Imprimir los elementos del arreglo, ¿verdad? Sin embargo, parece que le estamos diciendo al compilador qué hacer. En este caso, le estamos diciendo al compilador: "Obtener la longitud del arreglo, hacer un ciclo en nuestro arreglo, obtener cada elemento del arreglo utilizando el índice, etc." Lo llamamos una

solución imperativa. La *programación imperativa* se trata de decirle al compilador cómo hacer las cosas.

Ahora pasaremos al otro lado de la moneda, la programación *declarativa*. En la programación declarativa, le diremos al compilador qué debe hacer en lugar de cómo. Las partes del "cómo" se resumen en funciones comunes (estas funciones se denominan funciones de orden superior, que no cubriremos en el curso, sólo revisamos una introducción a la programación funcional con JavaScript). Ahora podemos usar la función forEach incorporada para iterar el arreglo e imprimirlo, como se muestra en el listado 7.

Listado 7: Iteración sobre el enfoque declarativo del arreglo.

```
var array = [1,2,3]
array.forEach((element) => console.log(element))
//prints 1, 2, 3
```

El listado 7 imprime exactamente el mismo resultado que el listado 5. Sin embargo, aquí hemos eliminado las partes del "cómo", como "Obtener la longitud del arreglo, hacer un ciclo en nuestro arreglo, obtener cada elemento de un arreglo utilizando un índice, etc.". Hemos utilizado una función abstracta, que se encarga de la parte del "cómo", dejándonos a nosotros, los desarrolladores, la preocupación de nuestro problema en cuestión (la parte del "qué"). Crear estas funciones integradas es parte del aprendizaje de la programación funcional con JavaScript, cosa que no realizaremos.

La programación funcional consiste en crear funciones de forma abstracta que puedan ser reutilizadas por otras partes del código. Ahora tenemos una comprensión sólida de lo que es la programación funcional; con esto en mente, podemos explorar los beneficios de la programación funcional.

Beneficios de la programación funcional

Hemos visto la definición de programación funcional y un ejemplo muy simple de una función en JavaScript. Ahora tenemos que responder una pregunta simple: ¿Cuáles son los beneficios de la programación funcional? Esta sección te ayudará a ver los enormes beneficios que nos ofrece la programación funcional.

La mayoría de los beneficios de la programación funcional provienen de escribir funciones puras. Por lo tanto, antes de ver los beneficios de la programación funcional, debemos saber qué es una función pura.

Funciones puras

Con nuestra definición en su lugar, podemos definir qué se entiende por funciones puras. Las *funciones puras* son las funciones que devuelven la misma salida para la entrada dada. Toma el ejemplo del listado 8.

Listado 8: Una función pura simple.

```
var double = (value) => value * 2;
```

Esta función double es una función pura porque dada una entrada, siempre devolverá la misma salida. Puedes probarlo tú mismo. Llamar a la función double con la entrada 5 siempre da como resultado 10. Las funciones puras obedecen a la transparencia referencial. Por lo tanto, podemos reemplazar double(5) con 10, sin dudarlo.

Entonces, ¿cuál es el problema con las funciones puras? Proporcionan muchos beneficios, que analizamos a continuación.

Las funciones puras conducen a un código comprobable

Las funciones que no son puras tienen efectos secundarios. Tomemos nuestro ejemplo de cálculo de impuestos anterior del listado 1:

```
var percentValue = 5;
var calculateTax = (value) => { return value/100 * (100 +
percentValue) } //depends on external environment percentValue variable
```

La función calculateTax no es una función pura, principalmente porque para calcular su lógica depende del entorno externo. La función funciona, pero es muy difícil de probar. Veamos la razón de esto.

Imagina que estamos planeando ejecutar una prueba para nuestra función calculateTax tres veces para tres cálculos de impuestos diferentes. Configuramos el entorno de esta manera:

```
calculateTax(5) === 5.25
calculateTax(6) === 6.3
calculateTax(7) === 7.3500000000000005
```

Se aprobó toda la prueba. Sin embargo, debido a que nuestra función calculateTax original depende de la variable de entorno externo percentValue, las cosas pueden salir mal. Imagina que el entorno externo está cambiando la variable percentValue mientras estás ejecutando los mismos casos de prueba:

```
calculateTax(5) === 5.25
// percentValue is changed by other function to 2
calculateTax(6) === 6.3 //will the test pass?
// percentValue is changed by other function to 0
calculateTax(7) === 7.3500000000000005 //will the test pass or throw exception?
```

Como puedes ver aquí, la función es muy difícil de probar. Sin embargo, podemos solucionar el problema fácilmente eliminando la dependencia del entorno externo de nuestra función, lo que lleva el código a esto:

```
var calculateTax = (value, percentValue) => { return value/100* (100 + percentValue) }
```

Ahora puedes probar esta función sin ningún problema. Antes de cerrar esta sección, debemos mencionar una propiedad importante sobre las funciones puras: Las funciones puras tampoco deberían mutar ninguna variable de entorno externo. En otras palabras, la función pura no debería depender de ninguna variable externa (como se muestra en el ejemplo) y también debería cambiar cualquier variable externa. Ahora veremos rápidamente lo que queremos decir con cambiar cualquier variable externa. Por ejemplo, considera el código del listado 9.

Listado 9: Ejemplo de badFunction.

```
var global = "globalValue"
var badFunction = (value) => { global = "changed";
return value * 2 }
```

Cuando se llama a la función badFunction, cambia la variable global *global* al valor modificado. ¿Es algo de lo que preocuparse?

Sí. Imagina otra función que depende de la variable global para tu lógica de negocios.

Por lo tanto, llamar a badFunction afecta el comportamiento de otras funciones. Las funciones de esta naturaleza (es decir, las funciones que tienen efectos secundarios) hacen que la base de código sea difícil de probar.

Además de las pruebas, estos efectos secundarios harán que el comportamiento del sistema sea muy difícil de predecir en el caso de la depuración.

Por lo tanto, hemos visto con un ejemplo simple cómo una función pura puede ayudarnos a probar fácilmente el código. Ahora veremos otros beneficios que obtenemos de las funciones puras: código razonable.

Código razonable

Como desarrolladores, debemos ser buenos en razonar sobre el código o una función. Al crear y usar funciones puras, podemos lograrlo de manera muy simple. Para aclarar este punto, vamos a utilizar un ejemplo simple de la función double (del listado 8):

```
var double = (value) => value * 2
```

Al observar el nombre de esta función, podemos razonar fácilmente que esta función duplica el número dado y nada más. De hecho, utilizando nuestro concepto de transparencia referencial, podemos fácilmente continuar y reemplazar la llamada de la función double con el resultado correspondiente. Los desarrolladores pasan la mayor parte de su tiempo leyendo el código de otros.

Tener una función con efectos secundarios en su base de código dificulta la lectura para otros desarrolladores en su equipo. Las bases de código con funciones puras son fáciles de leer, comprender y probar.

Recuerda que una función (independientemente de si es una función pura) siempre debe tener un nombre significativo. Por ejemplo, no puedes nombrar la función double como dd dado lo que hace.

PEQUEÑO JUEGO MENTAL

Solo estamos reemplazando la función con un valor, como si supiéramos el resultado sin ver su implementación. Esa es una gran mejora en tu proceso de pensamiento sobre las funciones. Estamos sustituyendo el valor de la función como si ese fuera el resultado que devolverá.

Para darle a tu mente un ejercicio rápido, ve esta capacidad de razonamiento con nuestra función Math.max incorporada.

Dada la llamada de función:

Math.max(3,4,5,6)

¿Cuál será el resultado?

¿Viste la implementación de max para dar el resultado? No, ¿verdad? ¿Por qué? La respuesta a esa pregunta es que Math.max es una función pura. Ahora tómate una taza de café; ¡has hecho un gran trabajo!

Código paralelo

Las funciones puras nos permiten ejecutar el código en paralelo. Como una función pura no va a cambiar ninguno de sus entornos, esto significa que no tenemos que preocuparnos por la *sincronización* en absoluto. Por supuesto, JavaScript no tiene subprocesos reales para ejecutar las funciones en paralelo, pero ¿qué pasa si tu proyecto usa WebWorkers para ejecutar varias cosas en paralelo? ¿O un código del lado del servidor en un entorno de node que ejecuta la función en paralelo?

Por ejemplo, imagina que tenemos el código que se proporciona en el listado 10.

Listado 10: Funciones impuras.

```
let global = "something"
let function1 = (input) => {
        // works on input
        //changes global
       global = "somethingElse"
}
let function2 = () => {
        if(global === "something")
       {
            //business logic
        }
}
```

¿Qué pasa si necesitamos ejecutar tanto function1 como function2 en paralelo?
Imagina que el subproceso uno (T-1) elige function1 para ejecutarse y el subproceso dos (T-2) elige function2 para ejecutarse. Ahora ambos subprocesos están listos para ejecutarse y aquí viene el problema. ¿Qué pasa si T-1 se ejecuta antes que T-2? Dado que tanto function1 como function2 dependen de la variable global '*global*', ejecutar estas funciones en paralelo provoca efectos no deseados. Ahora, convierte estas funciones en una función pura como se explica en el listado 11.

Listado 11: Funciones puras.

```
let function1 = (input,global) => {
            // works on input
            //changes global
            global = "somethingElse"
}
let function2 = (global) => {
            if(global === "something")
            {
```

```
            //business logic
        }
    }
```

Aquí hemos movido la variable *global* como argumentos para ambas funciones, haciéndolas puras. Ahora podemos ejecutar ambas funciones en paralelo sin ningún problema. Debido a que las funciones no dependen de un entorno externo (variable global), no nos preocupamos por el orden de ejecución de los subprocesos como en el listado 10.
Esta sección nos muestra cómo las funciones puras ayudan a que nuestro código se ejecute en paralelo sin ningún problema.

Almacenable en caché

Debido a que la función pura siempre devolverá la misma salida para la entrada dada, podemos almacenar en caché las salidas de la función. Para que esto sea más concreto, proporcionamos un ejemplo simple. Imaginemos que tenemos una función que realiza cálculos que consumen mucho tiempo. Llamamos a esta función longRunningFunction:

```
var longRunningFunction = (ip) => { //do long running tasks and return }
```

Si la función longRunningFunction es una función pura, entonces sabemos que, para la entrada dada, devolverá la misma salida. Con ese punto en mente, ¿por qué necesitamos llamar a la función nuevamente con su entrada varias veces? ¿No podemos simplemente reemplazar la llamada a la función con el resultado anterior de la función?

(Nuevamente nota aquí cómo estamos usando el concepto de transparencia referencial, reemplazando así la función con el valor del resultado anterior y dejando el contexto sin cambios). Imaginemos que tenemos un objeto de contabilidad que mantiene todos los resultados de llamadas de función de longRunningFunction de esta manera:

```
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 . . . }
```

El longRunningFnBookKeeper es un objeto JavaScript simple, que va a mantener todas las entradas (como llaves) y salidas (como valores) en él como resultado de invocar funciones longRunningFunction.

Ahora con nuestra definición de función pura en su lugar, podemos verificar si la llave está presente en longRunningFnBookKeeper antes de invocar nuestra función original, como se muestra en el listado 12.

Listado 12: Almacenamiento en caché logrado a través de funciones puras.

```
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 }
//check if the key present in longRunningFnBookKeeper
//if get back the result else update the bookkeeping object
longRunningFnBookKeeper.hasOwnProperty(ip) ?
    longRunningFnBookKeeper[ip] :
    longRunningFnBookKeeper[ip] = longRunningFunction(ip)
```

El código en el listado 12 es relativamente sencillo. Antes de llamar a nuestra función real, estamos verificando si el resultado de esa función con la dirección *ip* correspondiente está en el objeto de contabilidad. Si es así, lo devolvemos o llamamos a nuestra función original y actualizamos el resultado en nuestro objeto de contabilidad también. ¿Viste con qué facilidad hemos logrado que las llamadas a funciones se puedan almacenar en caché usando menos código? Ese es el poder de las funciones puras.

Pipelines y Composable

Con las funciones puras, vamos a hacer solo una cosa en esa función. Ya hemos visto cómo la función pura va a actuar como una autocomprensión de lo que hace esa función al ver su nombre. Las funciones puras deben diseñarse de tal manera que solo hagan una cosa. Hacer solo una cosa y hacerla a la perfección es una filosofía de UNIX; seguiremos la misma al implementar nuestras funciones puras.

Hay muchos comandos en las plataformas UNIX y LINUX que usamos para las tareas del día a día. Por ejemplo, usamos cat para imprimir el contenido del archivo, grep para buscar los archivos, wc para contar las líneas, etc. Estos comandos resuelven un problema a la vez, pero podemos *componer* o *canalizar* (pipeline) para realizar las tareas complejas. Imaginemos que queremos encontrar un nombre específico en un archivo de texto y contar sus apariciones. ¿Cómo lo haremos en nuestro símbolo del sistema? El comando se ve así:

```
cat jsBook | grep –i "composing" | wc
```

Este comando resuelve nuestro problema mediante la composición de muchas funciones.
La composición no solo es exclusiva de las líneas de comando de UNIX/LINUX; es el corazón del paradigma de la programación funcional. En nuestro mundo, llamamos a esto composición funcional. Imaginemos que estas mismas líneas de comando se han implementado en funciones de JavaScript. Podemos usarlas con los mismos principios para resolver nuestro problema.

Ahora, pensemos en otro problema de una manera diferente. Queremos contar la cantidad de líneas de texto. ¿Cómo lo resolveremos? ¡Ajá! Tienes la respuesta.

Los comandos son, de hecho, una función pura con respecto a nuestra definición. Toma un argumento y devuelve la salida al llamador sin afectar ninguno de los entornos externos.

Son muchos los beneficios que obtenemos al seguir una definición simple. Antes de cerrar este documento introductorio, queremos mostrar la relación entre una función pura y una función matemática. Abordaremos eso a continuación.

Una función pura es una función matemática

Antes vimos este fragmento de código en el listado 12:

```
var longRunningFunction = (ip) => { //do long running tasks and return }
var longRunningFnBookKeeper = { 2 : 3, 4 : 5 }
//check if the key present in longRunningFnBookKeeper
//if get back the result else update the bookkeeping object
longRunningFnBookKeeper.hasOwnProperty(ip) ?
        longRunningFnBookKeeper[ip] :
        longRunningFnBookKeeper[ip] = longRunningFunction(ip)
```

El objetivo principal era almacenar en caché las llamadas a funciones. Lo hicimos usando el objeto de contabilidad. Imaginemos que hemos llamado a longRunningFunction muchas veces para que nuestra longRunningFnBookKeeper crezca hasta convertirse en el objeto, que se ve así:

```
longRunningFnBookKeeper = {
    1 : 32,
    2 : 4,
    3 : 5,
    5 : 6,
    8 : 9,
    9 : 10,
    10 : 23,
    11 : 44
}
```

Ahora imaginemos que la entrada longRunningFunction varía solo de 1 a 11 números enteros, por ejemplo. Debido a que ya hemos creado el objeto de contabilidad (bookkeeping)

para este rango en particular, podemos hacer referencia solo a longRunningFnBookKeeper para decir la salida longRunningFunction para la entrada dada.

Analicemos este objeto de contabilidad. Este objeto nos da una idea clara de que nuestra función longRunningFunction toma una entrada y la asigna a la salida para el rango dado (en este caso, es 1–11). El punto importante a tener en cuenta aquí es que las entradas (en este caso, las llaves) tienen, obligatoriamente, una salida correspondiente (en este caso, el resultado) en el objeto.
Además, no hay ninguna entrada en la sección de llaves que se asigne a dos salidas.

Con este análisis podemos volver a examinar la definición de función matemática, esta vez brindando una definición más concreta de Wikipedia:
(https://en.wikipedia.org/wiki/Function_(mathematics)

En matemáticas, una función es una relación entre un conjunto de entradas y un conjunto de salidas permisibles con la propiedad de que cada entrada está relacionada con exactamente una salida. La entrada a una función se llama argumento y la salida se llama valor. El conjunto de todas las entradas permitidas a una función dada se llama dominio de la función, mientras que el conjunto de salidas permisibles se llama codominio.

Esta definición es exactamente la misma que nuestras funciones puras. Echa un vistazo a nuestro objeto longRunningFnBookKeeper. ¿Puedes encontrar el dominio y el codominio de nuestra función? Con este ejemplo muy simple, puedes ver fácilmente cómo se toma prestada la idea de función matemática en el mundo del paradigma funcional (como se indica al comienzo del documento).

Lo que podríamos construir

Hemos hablado mucho sobre funciones y programación funcional en este documento introductorio. Con este conocimiento fundamental, podríamos construir una biblioteca funcional. Esta biblioteca se construiría tema por tema para un curso introductorio. Al construir la biblioteca funcional, exploraremos cómo se pueden usar las funciones de JavaScript (de manera funcional) y también cómo se puede aplicar la programación funcional en las actividades cotidianas (usando nuestra función creada para resolver el problema en nuestra base de código).

¿Es JavaScript un lenguaje de programación funcional?

Antes de cerrar este documento, tenemos que dar un paso atrás y responder una pregunta fundamental: ¿Es JavaScript un lenguaje de programación funcional?

La respuesta es sí y no. Dijimos al principio del documento que la programación funcional se trata de funciones, que deben tomar al menos un argumento y devolver un valor. Sin embargo, para ser francos, podemos crear una función en JavaScript que no pueda tomar ningún argumento y, de hecho, no devuelva nada. Por ejemplo, el siguiente código es un código válido en el motor de JavaScript:

```
var useless = () => {}
```

Este código se ejecutará sin ningún error en el mundo de JavaScript. La razón es que JavaScript no es un lenguaje funcional puro (como Haskell), sino más bien un lenguaje multiparadigma. Sin embargo, el lenguaje es muy adecuado para el paradigma de programación funcional que se analiza en este texto.

Las técnicas y los beneficios que hemos analizado hasta ahora se pueden aplicar en JavaScript puro. JavaScript es un lenguaje que admite funciones como argumentos, pasar funciones a otras funciones, etc., principalmente porque JavaScript trata a las funciones como sus ciudadanos de primera clase. Debido a las restricciones según la definición del término función, nosotros como desarrolladores debemos tenerlas en cuenta al crearlas en el mundo de JavaScript. Al hacerlo, obtendremos muchas ventajas del paradigma funcional que se analiza en esta pequeña introducción.

Resumen

En este documento hemos visto qué son las funciones en matemáticas y en el mundo de la programación. Comenzamos con una definición simple de función en matemáticas y revisamos pequeños ejemplos sólidos de funciones y el paradigma de programación funcional en JavaScript. También definimos qué son las funciones puras y analizamos en detalle sus beneficios. Al final del documento también mostramos la relación entre las funciones puras y las funciones matemáticas. También analizamos cómo se puede tratar a JavaScript como un lenguaje de programación funcional.