Universidad del Valle

Facultad de Ingeniería Escuela de Ingeniería de Sistemas y Computación

Fundamentos de programación

Definición de función sin nombre usando flecha

En javascript se pueden definir funciones de diferentes maneras. Ya hemos visto casi todas ellas, pero vamos a recapitular antes de seguir adelante con este capítulo. La definición estándar de función es la siguiente:

```
function f(a, b) {
   return a + b;
}
```

Pero también hemos visto que podemos definir una función sin nombre y asignarla a una variable local. Como en el siguiente ejemplo:

```
const f = function(a, b) {
   return a + b;
}
```

Sin embargo, alguna veces estamos interesados en definir una función que hace lo mismo que f, pero que solo usaremos una vez en todo nuestro programa. En ese caso, podemos definir una función sin nombre, usando la definición flecha. La función que suma dos números a y b, se escribe en flecha de la siguiente manera:

```
const suma=(a, b) => a + b;
((a, b) => a + b);


(a, b) => {const x = 0;
           return a + b + x};
```

Las 2 formas anteriores son equivalentes, pero la primera es mucho más corta. En los casos donde la función solo consta de una expresión, podemos omitir las llaves y el return.

1. Abstracción de diseño: definiciones similares

Muchas definiciones de datos y de funciones vistas se parecen. Por ejemplo, la definición para una lista de símbolos difiere de la de una lista de números únicamente en dos aspectos: el nombre de los tipos de datos y en las palabras “símbolos” y “números”. De forma parecida, una función que busca un símbolo específico en una lista de símbolos es indistinguible de otra que busca un número específico en una lista de números.

Las repeticiones son la fuente de muchas equivocaciones de programación. De ahí que los buenos programadores buscan evitarlas tanto como sea posible. Al desarrollar un conjunto de funciones, especialmente funciones derivadas del mismo formato, se aprende a identificar repeticiones. Momento en el que conviene revisar las funciones a fin de eliminar las repeticiones tanto como sea posible.

La eliminación de repeticiones es el paso más importante en el proceso de edición (de programas). En esta sección, se analiza la similaridad en definiciones de funciones y datos. Los medios para evitar la similaridad de funciones son específicos de cada lenguaje. JavaScript es un lenguaje débilmente tipado, donde a diferencia de algunos lenguajes de programación los operadores no son funciones.

1.1 Similaridades en funciones

El empleo de recetas de diseño determina completamente el formato de una función a partir de la definición de datos de la entrada. De hecho, el formato es un método alternativo de expresar qué sabemos de los datos de entrada. De ahí que funciones que consumen la misma clase de datos se parezcan. Al analizar las dos funciones de la siguiente figura, las cuales consumen listas de strings(nombres de juguetes) y buscan juguetes específicos. La función a la izquierda busca **'muñeca''**, y la de la derecha **‘carro’** en una lista de strings(**ulds**). Las dos funciones son prácticamente casi indistinguibles. Cada una consume listas de símbolos; cada cuerpo de función consiste de una **expresión condicional if** con dos cláusulas. Cada una produce false si la entrada es empty; cada una emplea una segunda **expresión condicional** anidada para determinar si el primer ítem es el deseado. La única diferencia es el la cadena que se emplea en la comparación de la expresión condicional anidada: **contieneMuneca** emplea **’muñeca’** y **contieneCarro** emplea **‘carro’**. Para enfatizar las diferencias, los dos símbolos se marcan.

```
function contieneMuneca( ulds) {
    if (isEmpty(ulds))
        return false;
    else {
        if (first(ulds) == 'muñeca') {
            return true;
        } else {
            return contieneMuneca(rest(ulds));
        }
    }
}
```

```
function contieneCarro( ulds) {
    if (isEmpty(ulds))
        return false;
    else {
        if (first(ulds) == 'carro') {
            return true;
        } else {
            return contieneCarro(rest(ulds));
        }
    }
}
```

Los buenos programadores definen una sola función que busca una muñeca o un carro en una lista de juguetes. Esta función más general se define así:

```
// contieneItem : lista, cadena -> booleano
// determinar si ulds contiene la cadena s
function contieneItem(ulds, s) {
    if (isEmpty(ulds))
        return false;
    else {
        if (first(ulds) == s) {
            return true;
        } else {
            return contieneItem(rest(ulds), s);
        }
    }
}
```

Se puede buscar ahora **'muñeca'** mediante aplicar **contieneItem** a **'muñeca'** y **ulds**. Sin embargo, **contieneItem** opera para cualquier otro símbolo también.

El proceso de combinar dos funciones relacionadas en una sola definición se denomina **Abstracción Funcional**. Definir versiones abstractas de funciones es benéfico. El primer beneficio es que una sola función puede realizar muchas tareas diferentes. En el primer ejemplo, **contieneItem** puede buscar diferentes símbolos en lugar de uno solo.

La computación toma prestado el “término abstracto” de las matemáticas. Un matemático se refiere al “6” como un número abstracto debido a que sólo representa diferentes maneras de nombrar seis cosas. En contraste, “6 cm” o “6 huevos” son instancias concretas de “6” debido a que expresa una medida y un conteo.

En el caso de **contieneMuneca** y **contieneCarro**, la abstracción es poco interesante. Sin embargo, existen casos más interesantes: En la siguiente figura, la función de arriba

consume una lista de números y un límite, y produce una lista de todos aquellos números que están por debajo del límite; la de abajo produce aquéllos por encima del límí

```
//inferior: lista, numero -> lista
// construir una lista de números
// en uldn inferiores a t
function inferior(uldn, t) {
    if (isEmpty(uldn)) {
        return [];
    } else if (first(uldn) < t){
        return cons(first(uldn), inferior(rest(uldn), t));
    } else {
        return inferior(rest(uldn), t);
    }
}

//superior: (listade número), número -> lista
// construir una lista de números
// en uldn superiores a t
function superior(uldn, t) {
    if (isEmpty(uldn)) {
        return [];
    } else if (first(uldn) > t){
        return cons(first(uldn), superior(rest(uldn), t));
    } else {
        return superior(rest(uldn), t);
    }
}
```

Dos funciones similares adicionales. La diferencia entre las dos funciones es el operador de comparación. La de la izquierda emplea <, la derecha >. Siguiendo el primer ejemplo, se abstrae de las dos funciones con un parámetro adicional que representa el operador relacional concreto en **inferior** y **superior**, así:

```
//filtro1: (listade X), numero -> lista
// construir una lista de números
// en uldn inferiores o superiores
// a t según la función dada
function filtro1(uldn, t, f) {
```

```
    if (isEmpty(uldn)) {
        return [];
    } else if (f(first(uldn), t)){
        return cons(first(uldn), filtro1(rest(uldn), t, f));
    } else {
        return filtro1(rest(uldn), t, f);
    }
}
```

Para aplicar esta nueva función, se proporcionan tres argumentos: un operador relacional una lista uldn de números, un número t, y una función de orden **f**. La función extrae entonces todos aquellos ítems i en L para los cuales **f(first(uldn), t)** es verdadero. Puesto que no se sabe como escribir contratos para funciones como **filtro1**, se omite el contrato por ahora. Se analizará el problema de los contratos más adelante.

Considérese cómo opera **filtro1** con un ejemplo. Es claro, que si la lista de entrada es empty, el resultado es empty también, sin importar los demás argumentos:

```
filtro([], 5, (a, b) => a < b)// => []
```

Si se analiza un caso ligeramente más complicado:

```
filtro([4], 5, (a, b) => a < b)// => [4]
```

El resultado debe ser (cons 4 empty) debido a que el único elemento de la lista es 4 y (< 4 5) es verdadero.

Adicionalmente, una vez que se ha definido una función abstracta como filtro1, se le puede dar también usos adicionales. Tres de ellos son:

1. **filtro1(uldn, t, `(a, b) => a == b`)**: esta expresión extrae todos los números en uldn que son iguales a t.
2. **filtro1(uldn, t, `(a, b) => a <= b`)**: produce la lista de números en uldn que son menores o igual a t.
3. **filtro1(uldn, t, `(a, b) => a >= b`)**: calcula la lista de números que son mayores o iguales al límite.

En general, el tercer argumento de **filtro1** puede ser cualquier función de JavaScript que opera sobre 2 valores y retorna un booleano. Considérese el siguiente ejemplo:

```
//cuadradoGT : numero, numero-> booleano
function cuadradoGT(x, c) {
```

```
    return  x * x >  c;
}
```

La función produce **true** siempre que el área de un cuadrado con lado **x** sea mayor que un límite **c**, o sea la función prueba si es válida la afirmación $\mathbf{x^2 > c}$. Ahora si se aplica **filtro1** a esta función teniendo en cuenta una lista de números:

```
    filtro1([1, 2, 3, 4, 5], 10, cuadradoGT);
```

Esta aplicación en particular extrae aquellos números en `[1, 2, 3, 4, 5]` cuyo cuadrado sea mayor que 10. Para ello utilizamos: filtro1(`[1, 2, 3, 4, 5]`, 10, cuadradoGT) y obtenemos [4 5]

Hasta ahora se ha visto que definiciones de función abstractas son más flexibles. Una segunda ventaja, e igualmente importante, de las definiciones abstractas es que se puede cambiar una sola definición para corregir. Considérense dos variantes de **filtro1** en la siguiente figura. La primera variante utiliza la expresión cond anidada, algo no deseable para un programador experimentado. La segunda variante emplea una expresión local que facilita la lectura de la expresión cond anidada.

```
function filtro1(uldn, t, f) {
    if (isEmpty(uldn)) {
        return [];
    } else if (f(first(uldn), t)){
        return cons(first(uldn), filtro1(rest(uldn), t, f));
    } else {
        return filtro1(rest(uldn), t, f);
    }
}
```

```
function filtro1(uldn, t, f) {
    if (isEmpty(uldn)) {
        return [];
    } else {
          if (f(first(uldn), t)){
              return cons(first(uldn), filtro1(rest(uldn), t, f));
          } else {
               return filtro1(rest(uldn), t, f);
```

```
        }
    }
}
```

```
function filtro1(uldn, t, f) {
   if (isEmpty(uldn)) {
      return [];
   }
   const firstItem = first(uldn);
   const restoFiltrado = filtro1(rest(uldn), t, f);
   if (f(firstItem, t)){
       return cons(firstItem, restoFiltrado);
   }
   return restoFiltrado;
}
```

Dos modificaciones de filtro1

Aunque ambos cambios son triviales, la clave es que todas las que emplean filtro1, incluyendo las empleadas para definir las funciones **inferior1** y **superior1**, se benefician de los cambios. De forma parecida, si la modificación corrige un error lógico, todos los usos de la función resultan mejorados. Finalmente, es posible agregar nuevas tareas a funciones abstractas, por ejemplo, un mecanismo para contar cuántos elementos se filtran. En este caso todos los empleos de la función resultarán beneficiados desde la nueva funcionalidad.

Ejercicio 49: Abstraer las siguientes dos funciones en una:

```
// ldnnv, una lista de números no vacía
// min1 : lista -> número
// determinar el número menor en uldn
function min1(uldn) {
   if (length(uldn) == 1) {
       return first(uldn);
   } else {
       if ( first(uldn) < min1 (rest(uldn))) {
           return first(uldn);
       } else {
```

```
            return min1 (rest(uldn));
        }
    }
}
```

```
// max1 : lista -> número
// determinar el número mayor en uldn
function max1(uldn) {
    if (length(uldn) == 1) {
        return first(uldn);
    } else {
        if ( first(uldn) > max1 (rest(uldn))) {
            return first(uldn);
        } else {
            return max1 (rest(uldn));
        }
    }
}
```

Las dos funciones consumen listas no vacías de números y generan un sólo número. La primera produce el número menor en la lista, la segunda el mayor.

Definir min1 y max1 en términos de una función abstracta. Probarla en las siguientes listas:
1. [3, 7, 6, 2, 9, 8]
2. [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 11]
3. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

¿por qué es tan lenta la evaluación de las listas más grandes?

Mejorar la función abstracta, introduciendo una constante local para el resultado de la recursión natural.

1.2. Similaridad en definiciones de datos

Ahora veamos la siguientes definiciones de datos, ambas definen una clase de listas, en la arriba una lista de números, en la abajo registros de inventario representados en una estructuras:

Una lista-de-números es o Una lista-de-Objetos es o empty([]), cons(n, l) donde n es un número o un Objeto y l es una lista-de-números o l es una lista-de-Objetos.

Dada la similaridad entre las definiciones de datos, las funciones que consumen elemento de estas clases son similares también. Véase el siguiente ejemplo. La función de abajo es inferiorRi, la cual extrae aquellos registros de inventario cuyos precios son inferiores a un cierto límite.

```
// inferior: (listade número), número -> lista
// construir una lista de números
// en uldn inferiores a t
function inferior(uldn, t) {
    if (isEmpty(uldn)) {
        return [];
    } else if (first(uldn) < t){
        return cons(first(uldn), inferior(rest(uldn), t));
    } else {
        return inferior(rest(uldn), t);
    }
}
```

```
// inferiorRi : (listade Objeto), número -> lista
// construir una lista de registros en uldri
// que contengan un precios inferior a t
function inferiorRi(uldn, t) {
   if (isEmpty(uldn)) {
       return [];
   } else if (first(uldn).precio < t){
       return cons(first(uldn), inferiorRi(rest(uldn), t));
   } else {
       return inferiorRi(rest(uldn), t);
   }
}
```

Señalar las diferencias en funciones similares

Si se abstraen las dos funciones, se obtiene **filtro1**. A su vez, se puede definir inferiorRi en términos de **filtro1**:

```
function inferiorRi(uldn, t) {
   return filtro1(uldn, t, (a, b) => a.precio < b);
```

```
}
```

No debe sorprender descubrir otro empleo para **filtro1** (después de todo, ya se argumentó que la abstracción promueve el reuso de funciones para propósitos diferentes). Aquí se constata que **filtro1** no filtra únicamente listas de números, sino listas de cosas arbitrarias (en la medida se pueda definir una función que compare dichas cosas arbitrarias con números).

De hecho, todo lo que se requiere es una función que compare ítems en la lista con ítems que se pasen a **filtro1** como segundo argumento. A continuación se presenta una función que extrae todos los ítems con la misma etiqueta a partir de una lista de registros de inventarios:

```
// encuentra : lista -> booleano ;;
// determinar si ldRI contiene un registro para t
function encuentra(uldri, t) {
   return isList(filtro1(uldri, t, eqRi));
}

// eqRi: object, string -> booleano
// comparar el nombre de ri y p
function eqRi (ri, p) {
   return ri.nombre == p;
}
```

Este nuevo operador relacional compara el nombre en un registro de inventario con algún otro símbolo.

Ejercicio 64: Determinar los valores de

```
1. inferiorRi([{nombre: 'muñeca', precio: 8}, {nombre:'robot', precio: 12}], 10);

2. encuentra([{nombre: 'muñeca', precio: 8}, {nombre:'robot', precio: 12}, {nombre:
'muñeca', precio: 12}], 'muñeca')
```

Es decir que **filtro1** opera uniformemente en muchas formas de datos de entrada. La palabra “uniformemente” significa que **filtro1** se aplica a una lista de X, su resultado es también una lista de X no importando X que clase de datos sea. Dichas definiciones se denominan polimórficas o funciones **genéricas**.

Por supuesto que **filtro1** no es la únicamente función que puede procesar listas arbitrarias. Existen otras muchas funciones que procesan listas independientemente de qué contienen.

A continuación se presentan dos funciones que determinan la longitud de listas de números y listas de objetos(Ri):

```
function longitudRi(uldn) {
   if (isEmpty(uldn))
       return 0;
   else {
       return 1 + longitudRi(rest(uldn));
   }
}
```

```
function longitudNum(uldn) {
   if (isEmpty(uldn))
       return 0;
   else {
       return 1 + longitudNum(rest(uldn));
   }
}
```

Las dos funciones difieren únicamente en sus nombres. Si se hubiera seleccionado el mismo nombre, por ejemplo, longitud, las dos definiciones serían idénticas.

Para escribir contratos precisos para funciones como longitud, se requieren definiciones de datos con parámetros. Éstas se denominan Definiciones de Datos Paramétricas y convienen en no especificar nada acerca de una clase de datos. En cambio, emplean variables para indicar que cualquier forma de datos Scheme pueden emplearse en cierto lugar. Se puede decir que una definición de datos paramétrica abstrae su referencia de una colección particular de datos de la misma forma que una función abstrae a partir de un valor particular.

Una definición paramétrica de listas de ÍTEM

Una lista de ITEM es cualquiera de:
1. empty o
2. [s l]

donde
1. s es un ÍTEM y
2. l es una lista de ÍTEM.

El término ÍTEM es un Tipo Variable que representa cualquier colección arbitraria de datos Scheme: símbolos, números, booleanos, RIs, etc. Mediante reemplazar ÍTEM con algunos de estos nombres, se obtiene una instancia concreta de dicha definición de datos abstracta

para listas de símbolos, números, booleanos, RIs, etc. A fin de hacer el lenguaje de contratos más conciso, se introduce una abreviación adicional:

(listade ITEM)

Se emplea (**listade** ITEM) como nombre de definiciones de datos abstractas como la anterior. Pudiéndose emplear (**listade** string) para la clase de todas las listas de símbolos, (**listade** número) para la clase de todas las listas de números, (**listade** (listade número)) para la clase de todas las listas de listas de números, etc.

En contratos se emplea (**listade** X) para expresar que una función opera en todo tipo de listas:

```
// longitud : (listade X) -> número
// calcula la longitud de una lista
function longitud(uldn) {
   if (isEmpty(uldn))
       return 0;
   else {
       return 1 + longitud(rest(uldn));
   }
}
```

La **X** es exactamente una variable, un nombre que representa alguna clase de datos. Si ahora se aplica **longitud** a un elemento de, por ejemplo, (**listade** string) o (**listade** RI), se obtiene un número.

La función **longitud** es un ejemplo de polimorfismo simple. Opera en todas las clases de listas. Si bien existen otros ejemplos útiles de funciones polimórficas simples, los casos más comunes requieren que se definan funciones como **filtro1**, la cual consume una forma paramétrica de datos y funciones que operan en dichos datos. Esta combinación es extremadamente poderosa y facilita significativamente la construcción y mantenimiento de los sistemas de software.

Problem 1

1. Implemente una función que eleva al cuadrado cada uno de los elementos de una lista
2. Implemente una función que convierte a negativo cada uno de los elementos de una lista
3. Implemente una función retornal la longuitud de cada uno de los elementos de una lista

cuadrado([1, 2, 3]) // => [1, 4, 9]
negativo([1, 2, -3]);// => [-1, -2, 3]
strLng([‘aa’, “bbc”, ‘nnnnn’]); // => [2, 3, 5]

Problema 2

4. Implemente una función que retorna el máximo de una lista
5. Implemente una función que retorna el mínimo de una lista
6. Implemente una función retornal la suma de los elementos de la lista
7. Implemente una función retornal la productoria de los elementos de la lista