Cuadernos de Lingüística Hispánica n.° 34
ISSN 0121-053X • ISSN en línea 2346-1829
Julio - Diciembre 2019, pp. 109-130

Programación de buscadores en JavaScript para diccionarios digitales*

Haakon S. Krohn**

Recepción: 12 de enero de 2019
Aprobación: 03 de junio de 2019

Forma de citar este artículo: Krohn, H. (2019). Programación de buscadores en JavaScript para diccionarios digitales. *Cuadernos de Lingüística Hispánica*, (34), pp 109-130.

https://doi.org/10.19053/0121053X.n34.2019.9410

* Artículo de investigación. Este artículo forma parte del proyecto de investigación 745-B8-131 "Diccionario digital bilingüe bribri (fase 1)", inscrito en el Instituto de Investigaciones Lingüísticas de la Universidad de Costa Rica, con presupuesto de la Vicerrectoría de Investigación de dicha institución.

** Magíster en Lingüística, Universidad de Costa Rica. Profesor, Escuela de Filología, Lingüística y Literatura, Universidad de Costa Rica. Correo electrónico hkrohn@gmail.com. https://orcid.org/0000-0003-1400-9615

Resumen

En este artículo, dirigido a lexicógrafos sin conocimientos previos sobre la programación, se explica, paso a paso, cómo programar motores de búsqueda con distintas funcionalidades en JavaScript para diccionarios digitales. De esta manera se espera contribuir a que los lexicógrafos logren participar más activamente en el desarrollo de esta herramienta tan esencial para un diccionario. Los ejemplos se basan en las características de los buscadores de varios diccionarios disponibles en la web y se enfocan principalmente en tres tipos de correspondencias entre el término de búsqueda y las palabras clave: completa, parcial y con comodín. Además, se muestra cómo crear una función adicional para permitir aún mayor flexibilidad por medio de la sustitución de caracteres.

Palabras clave: lexicografía, lingüística informática, programación informática.

Search Engine Programming in JavaScript for Digital Dictionaries

Abstract

In this paper, aimed at lexicographers without previous knowledge about programming, we explain, step by step, how to program search engines with different functionalities in JavaScript for digital dictionaries. In this way, we expect to contribute to lexicographers to participate more actively in the development of this tool so essential for a dictionary. The examples are based on characteristics of the search engines of several dictionaries available on the web and focus mainly on three types of correspondences between the search term and the keywords: complete, partial and with wildcard. Also shown is how to create an additional function to allow even more flexibility by means of character substitution.

Keywords: lexicography, computational linguistics, computational programmi

Programmation de moteurs de recherche dans JavaScript pour des dictionnaires digitaux

Résumé

Dans cet article, dirigé aux lexicographes sans connaissances préalables en programmation, on explique, étape par étape, comment programmer des moteurs de recherche avec de différentes fonctionnalités en JavaScript pour des dictionnaires digitaux. De cette manière, on espère contribuer à faire que les lexicographes puissent participer plus activement dans le développement de cet outil si essentiel pour un dictionnaire. Les exemples se basent sur les caractéristiques des moteurs de recherche de plusieurs dictionnaires disponibles dans le web et se focalisent principalement sur trois types de correspondance entre le terme de recherche et les mots clés : complète, partielle et avec des remplacements. En plus, on montre comment créer une fonction additionnelle pour permettre encore plus de flexibilité par le biais de la substitution de caractères.

Mots clés: lexicographie, linguistique informatique, programmation informatique.

Programação de Java8script search para dicionários digitais

Resumo

Este artigo, destinado a lexicógrafos sem conhecimento prévio sobre programação, explica, passo a passo, como programar mecanismos de pesquisa com diferentes funcionalidades em Java Script para dicionários digitais. Dessa forma, espera-se que os lexicógrafos possam participar mais ativamente do desenvolvimento dessa ferramenta essencial para um dicionário. Os exemplos são baseados nos recursos de mecanismo de pesquisa de vários dicionários disponíveis na Web e focam principalmente em três tipos de correspondência entre o termo de pesquisa e as palavras-chave: completo, parcial e curinga. Além disso, mostra como criar uma função adicional para permitir ainda mais flexibilidade, substituindo caracteres.

Palavras-chave: lexicografia, linguística de computadores, programação de computadores.

Introducción

Ya que nos hallamos inmersos en la era digital desde hace varios años, es cada vez más común que los diccionarios se elaboren con el fin de publicarse en formato digital. Esto no solo ahorra espacio físico y simplifica la distribución de la obra, sino que también, por lo general, reduce el tiempo que los usuarios tienen que invertir en encontrar el artículo que buscan: en vez de la tradicional búsqueda manual de un lema entre miles de otros en páginas impresas, los diccionarios digitales suelen ofrecer buscadores que ayudan a localizar el artículo casi inmeditamente. Sin embargo, si el motor de búsqueda no está satisfactoriamente implementado, el tiempo de búsqueda podría, incluso, llegar a sobrepasar el de los diccionarios impresos. Por consiguiente, se trata de un componente esencial, y posiblemente minusvalorado, de los diccionarios digitales.

El motor de búsqueda es, normalmente, programado por los mismos informáticos que desarrollan el resto de la plataforma del diccionario. Empero, nos parece esencial que también los autores del diccionario participen en la elaboración de este componente, puesto que el funcionamiento debe adecuarse a varios aspectos directamente relacionados con su área de especialidad, de los que se pueden destacar los siguientes: (1) la estructura morfológica de la lengua descrita, (2) el sistema de escritura de la lengua, (3) la estructura del diccionario y (4) los usuarios meta. Por esta razón, consideramos oportuno que los lexicógrafos adquieran una comprensión básica de la programación de buscadores. Este objetivo es el que se pretende cumplir en el presente escrito, donde se desarrollan varios ejemplos de motores de búsqueda acompañados por explicaciones dirigidas a lectores que no posean conocimientos previos de la programación. La limitación del espacio no nos permite entrar en detalles acerca de los principios más básicos de la programación, pero las soluciones expuestas son más transparentes que las que normalmente se utilizarían, con el fin de que el proceso sea comprensible para todos los lectores. Esto significa que los algoritmos de búsqueda presentados son muy básicos y no los más eficientes, pero perfectamente funcionales para la mayoría de los diccionarios.

Los ejemplos se elaboran en JavaScript, principalmente por tratarse de un lenguaje de programación que constituye una de las piedras angulares del desarrollo web, canal por el que se publica gran parte de los diccionarios digitales. Además, se trata, posiblemente, del lenguaje más utilizado en el mundo de la informática en general (Subramaniam, 2018, p. xi). También para la elaboración de diccionarios publicados por otro medio, por ejemplo, en una aplicación para celulares, la comprensión de las estructuras en JavaScript ayudará considerablemente para entender la sintaxis y el diseño de otros lenguajes de programación.

El artículo está organizado como se señala a continuación. En primer lugar, en el apartado 1, se explican los fundamentos de las funciones JavaScript que se desarrollan a lo largo del escrito. Seguidamente, en la sección 2, se describe cómo crear un código que reconozca distintos tipos de correspondencias entre el término de búsqueda y las palabras clave del diccionario. Por último, en el apartado 3, se desarrolla una función adicional para permitir mayor variación ortográfica en el término de búsqueda.

Cabe subrayar que el acicate para el presente texto fue la elaboración de un motor de búsqueda para un diccionario digital bribri–español, español–bribri[1], el cual, al finalizarse, se publicará en la web. Especialmente la gran cantidad de diacríticos utilizados para escribir esta lengua fue un factor que influyó directamente en la implementación del buscador, como se explica al final del apartado[1] .

1. Fundamentos teóricos

JavaScript es un lenguaje de *script*, creado por Brendan Eich para el navegador Netscape en 1995 con el fin de posibilitar páginas web más dinámicas. Hoy en día es uno de los lenguajes de programación más importantes en el mundo de la informática, ya que se utiliza en la gran mayoría de los sitios web modernos (Flanagan, 2011, p. 1). Una ventaja de JavaScript es que se incrusta fácilmente dentro de HTML, el lenguaje de marcado empleado para codificar páginas web. Además, como afirma Subramanian (2018, p. vi), es uno de los lenguajes de programación más poderosos, extendidos y flexibles que existen actualmente.

A lo largo del artículo se desarrollarán distintas variantes de una función JavaScript que llamaremos `buscar()`, la cual sería invocada, directa o indirectamente, cuando un usuario del diccionario realice una búsqueda, típicamente por medio de un clic en un botón virtual o por pulsar la tecla Intro. La estructura básica de esta función es la que se visualiza en el ejemplo 1. Dentro de las llaves, sustituyendo los puntos suspensivos, se definirán las instrucciones de la función, en las que se hará énfasis en el siguiente apartado.

```
(1) function buscar(búsqueda) {
          ...

          }
```

1 El bribri es una lengua chibcense hablada en Costa Rica y Panamá.

Esta función en particular toma un argumento obligatorio: el término buscado por el usuario, el cual es asignado a una variable llamada `búsqueda`. El término de búsqueda, al igual que todos los otros objetos de texto en JavaScript, tiene el formato de una cadena de caracteres (ing.: "string"), un tipo de dato elemental que siempre se representa entre comillas. Típicamente, en un paso anterior, este se extraería de un campo de texto utilizado para las búsquedas, pero ese procedimiento es irrelevante para la función de búsqueda en sí, por lo que no se explicará aquí.

El propósito de la función `buscar()` es comparar el término de búsqueda con los miembros de un conjunto de cadenas de texto que funcionan como palabras clave y representan las distintas entradas en el diccionario. Seguidamente, debe retornar el resultado de esta comparación. En un sistema simple, las palabras clave podrían ser simplemente los lemas registrados en el diccionario, pero también se podrían incluir varias palabras clave distintas para cada entrada. Por ejemplo, en el *Diccionario de la lengua española* (Real Academia Española, 2018) y muchos otros diccionarios en línea, las formas conjugadas de los lexemas también funcionan como palabras clave.

Vamos a presuponer que, al principio de la función, todas las palabras clave son incrustadas en una matriz (ing.: "array") asignada a una variable, la cual denominaremos pc (abreviatura de *palabras clave*). No especificaremos cómo llevar a cabo la población de esta matriz, ya que eso depende del formato en el que esté guardada la información lexicográfica.

La estructura de la matriz de palabras clave, por su lado, depende de si todas estas son distintas entre sí. Algunos diccionarios, tal como el *Clave: diccionario de uso del español actual* (Ediciones SM, 2014), siguen este principio, de manera que todos los homógrafos están registrados dentro de la misma entrada, además de que solamente los lemas (y no, por ejemplo, las formas conjugadas) funcionan como palabras clave para el motor de búsqueda. En este caso, las palabras clave pueden incrustarse en una matriz unidimensional, cuya longitud equivale al número de entradas en el diccionario. Un fragmento de esta matriz podría, por lo tanto, ser el siguiente (los puntos suspensivos representan la omisión de las demás palabras clave):[2]

(2) `[..., "habanero", "habano", "habeas corpus", "haber", "habichuela", "hábil", "habilidad", "habilidoso", "habitación", "habilitado", ...]`

2 Las palabras clave de este ejemplo corresponden a lemas encontrados en el Clave: diccionario de uso del español actual (Ediciones SM, 2014).

Sin embargo, en muchos diccionarios, los homógrafos se registran como entradas independientes, lo que también permite la inclusión de formas conjugadas como palabras clave. Si se opta por esta alternativa, las palabras tienen que registrarse en una estructura de datos más compleja, pues si no, surgirían ambigüedades a la hora de visualizar los resultados de la búsqueda. En esencia, cada entrada debe tener un identificador único que la diferencie de todas las demás. Los lenguajes de programación proveen muchas posibles soluciones para esta última opción, pero la que tal vez sea la más intuitiva es la incrustación de las palabras clave en una matriz bidimensional. Suponiendo que los identificadores únicos son números asociados con cada entrada, la matriz pc podría tener la estructura esbozada en el ejemplo 3. Se aprecia que todas las formas conjugadas del verbo ***hablar*** estarían asociadas al mismo valor númerico (en este ejemplo, 245), el cual representaría la entrada de dicho verbo. Asimismo, las formas singular y plural del sustantivo homónimo ***habla*** estarían registradas con otro identificador (244). Igualmente, todas las formas de un adjetivo como *nuevo* también se registrarían con referencia a un mismo identificador (402).

(3)
```
[...
["habla", 244],
["hablas", 244],
...
["hablar", 245],
["hablo", 245],
["hablas", 245],
["habla", 245],
...
["hable", 245],
["hables", 245],
...
["nuevo", 402],
["nueva", 402],
["nuevos", 402],
 ["nuevas", 402],
 ...]
```

Por añadidura, es necesario definir qué acción realizará la función después de completar la búsqueda. Se pueden resaltar al menos dos posibilidades: retornar el resultado de la búsqueda, o invocar otra función con el resultado de la búsqueda enviado como argumento. Por motivos de simplicidad, se va a utilizar la primera opción en los ejemplos de este artículo, lo cual significa que los resultados se guardarán en una matriz asignada a una variable denominada `resultados`, que será retornada por la función. Inmediatamente después, la función `buscar()` será terminada. Esto implica que la función, en primera instancia, debe ser invocada por otra función, la cual haga uso de los resultados de manera adecuada para visualizárselos al usuario. La abstracción obtenida por medio del empleo de funciones en JavaScript nos permite ignorar las instrucciones de esta otra función, la cual es de menor relevancia para el lexicógrafo y que, por esa razón, queda fuera del alcance del presente escrito.

2. Tipos de correspondencias

Uno de los primeros pasos en la elaboración del buscador consiste en decidir qué parte de una palabra clave tiene que coincidir con qué parte del término de búsqueda para que se considere una correspondencia y se añada a la lista de resultados. En el presente apartado se discuten distintos tipos de correspondencias, con base en un análisis de las listas de resultados proporcionadas por los motores de búsqueda de algunos de los diccionarios más importantes en la web para las lenguas español e inglés. A este respecto, adelantamos que la mayoría de los diccionarios examinados emplea un sistema de correspondencia con un comodín, con prioridad a las correspondencias completas; no obstante, también se observan otras funcionalidades.

2.1 Correspondencia completa

El tipo de correspondencia más fácil de programar es la completa, la cual consiste en que una palabra clave sea exactamente igual al término de búsqueda. Algunos buscadores rudimentarios consisten únicamente en una sentencia condicional que determina si existe una correspondencia de este tipo. En caso positivo, se visualiza el artículo representado por la palabra clave en cuestión, y si no, se muestra un mensaje que informa que el término no se encontró en el diccionario. Esta búsqueda es tan simple que podría definirse en una sola línea de programación en nuestra función (aparte de las que pueblan la matriz de todas las palabras clave y la asigna a la variable pc):

```
(4)  return pc.includes(búsqueda);
```

Este fragmento de código retorna el resultado de un método que determina si la cadena de caracteres guardada en la variable `búsqueda` (que es igual al término

de búsqueda ingresado por el usuario) se encuentra en la matriz pc. El resultado de la prueba es de tipo booleano: *true* (*verdadero*) o *false* (*falso*). Con base en este valor, otra función es responsable de mostrar el artículo correspondiente o un aviso de que no se encontró el término buscado.

Sin embargo, lo que parece ser más común en los diccionarios en la web es que una prueba de este tipo (pero más compleja) se incluya al inicio de la función de búsqueda, seguida de pruebas de otros tipos de correspondencias. De esta manera, una correspondencia completa se halla antes que las correspondencias menos exactas, y se le puede dar prioridad a la hora de visualizar los resultados. Verbigracia, el artículo relacionado con la correspondencia completa puede mostrarse de inmediato, sin que siquiera se realicen las otras pruebas, las cuales se llevarían a cabo solo si no se encuentra ninguna correspondencia completa. Este funcionamiento se identifica, por ejemplo, en el *Diccionario de la lengua española* (Real Academia Española, 2014), los *Oxford Dictionaries* (Oxford University Press, 2018), el *Cambridge Dictionary* (Cambridge University Press, 2018) y el *Merriam-Webster Online Dictionary* (Merriam-Webster, 2018). En todas estas obras digitales, los homógrafos se visualizan en una misma página, y también se utilizan formas conjugadas como palabras clave.

Para una función que también busque otras clases de correspondencias, no solo las completas, la línea presentada en el ejemplo 4 no es suficiente, desde luego. En primera instancia, supondremos que todas las palabras clave del diccionario son diferentes y que los otros tipos de correspondencias deben buscarse solamente si no se encuentra una completa. En este caso, la función podría empezar con las líneas apreciadas en el ejemplo 5. Aquí se realiza una prueba de si el término de búsqueda se encuentra entre las palabras clave y, en caso de que sí, se retorna el mismo término (para que se visualice el artículo correspondiente) y se finaliza la función. En caso contrario, no se realiza ninguna acción y la función continúa con las líneas subsiguientes (detalladas en los apartados 3.2 y 3.3).

(5)
```
if (pc.includes(búsqueda)) {
 return búsqueda;
 }
```

En cambio, si se desea incluir los otros tipos de correspondencias dentro de los resultados también en caso de que se encuentre una correspondencia completa, la solución más simple consiste en inicializar una nueva variable, que llamaremos `resul-`

tados, al principio de la función. A esta se le asigna primeramente una matriz unidimensional vacía, a la que se insertarán todas las correspondencias identificadas. En el ejemplo 6, donde también se presupone que todas las palabras clave son distintas, la inicialización (especificada mediante la palabra `let`) de esta variable se lleva a cabo en la primera línea. La siguiente línea define una sentencia condicional (ing.: "if statement") que comprueba si el término de búsqueda se encuentra registrado dentro de la matriz `pc`. En caso afirmativo, se realizan varias acciones. Primero, el número que corresponde a la posición de la palabra clave dentro de la matriz es asignado a una variable i. Seguidamente, esta palabra clave se agrega al final de la matriz `resultados` (por medio del método `push` en la línea 4). Asimismo, la palabra clave se elimina de la matriz `pc` para evitar que aparezca más de una vez en la lista de resultados (método `splice` en la línea 5), debido a que la misma palabra clave también presentaría correspondencias parciales con el término de búsqueda. De esta manera, las correspondencias completas se colocan primero en la lista de resultados, pero también se pueden incluir resultados menos atinados.

```
(6) let resultados = [];
    if (pc.includes(búsqueda)) {
    let i = pc.indexOf(búsqueda);
    resultados.push(pc[i]);
    pc.splice(i, 1);
    }
```

Si, por el contrario, las palabras clave se pueden repetir, es necesario modificar la función, puesto que, en tal caso, la matriz `pc` tendría una estructura distinta. El ejemplo 7 constituye una de muchas posibles soluciones, con una matriz bidimensional (siguiendo la estructura propuesta en el ejemplo 3) asociada a la variable `pc`. Aquí se utiliza un bucle *for* (ing.: "for loop") (líneas 2–7), que es responsable de iterar sobre todas las palabras clave. La variable i alude a la posición de la palabra clave en la matriz, valor que va aumentando en cada vuelta del bucle hasta llegar al final de la matriz. Cada palabra clave se compara con el término de búsqueda (línea 3) y, en caso de hallarse una correspondencia completa, el identificador numérico de la palabra clave se añade a la matriz de resultados (línea 4), antes de que se elimine la palabra de la matriz `pc` (línea

5). Luego, el bucle *for* sigue iterando sobre el resto de las palabras clave[3], realizando la misma prueba de equivalencia.

(7)
```
let resultados = [];
for (let i = 0; i < pc.length; i++) {
if (búsqueda === pc[i][0]) {
resultados.push(pc[i][1]);
pc.splice(i, 1);
i--;
}
}
```

2.2 Correspondencia parcial

Con "correspondencia parcial" nos referimos a los casos en los que el término de búsqueda es una subcadena de alguna de las palabras clave. Esto quiere decir que el término de búsqueda se encuentra, en su totalidad, dentro de una palabra clave, pero con caracteres adicionales a la izquierda, a la derecha o por ambos lados.

Los buscadores de ciertos diccionarios en la web usan este tipo de correspondencias, de alguna u otra manera. El *Clave: diccionario de uso del español*, cuyo buscador utiliza únicamente los lemas como palabras clave, ofrece tres botones de radio (con las etiquetas "Comienza", "Contiene" y "Acaba") por medio de los que el usuario selecciona qué parte del lema debe corresponder al término de búsqueda para aparecer entre los resultados. El *Larousse Diccionario Avanzado de Español* (Ediciones Larousse, 2017), por su lado, prefiere las correspondencias completas, pero si no encuentra ninguna, muestra como primeros resultados, alfabéticamente, los lemas que comienzan con la cadena de caracteres ingresada en el buscador.

Para programar una función de este tipo, haremos uso de expresiones regulares (ing.: "regular expressions"), también conocidas como *regex*. Una expresión regular es definida por Mitkov (2003, p. 758) como "una expresión que describe un conjunto de cadenas (= un lenguaje regular)". Como señala el mismo autor, cada lenguaje descrito por una expresión regular puede ser representado mediante un autómata finito.

3 En la línea 6 se realiza una reducción del valor de la variable i, para compensar por el acortamiento de la matriz pc realizado en la línea 5.

Groner y Manricks (2015, p. 1), por su parte, explican que una expresión regular es una cadena que describe un patrón utilizando una sintaxis especializada de caracteres. Además, apuntan que las expresiones regulares no son específicas para ningún lenguaje de programación en particular y que han sido implementadas en casi todos los lenguajes modernos.

Primeramente, programaremos una función que busque correspondencias al inicio de las palabras clave. De esta manera, el término de búsqueda "in" daría como resultado todos lemas representados por una palabra clave que empiece con dicha secuencia de caracteres. De aquí en adelante vamos a suponer nuevamente que la lista de palabras clave es equivalente a los lemas registrados en el diccionario, para que el código se mantenga sencillo.

Estas líneas de programación podrían agregarse después de la prueba de equivalencia completa desarrollada en la sección 2.1 (en caso de que se prefieran las correspondencias completas), pero también funcionarían por sí solas. El ejemplo 8 constituye una posible implementación. En la línea 2 se construye una nueva expresión regular, asignada a la variable exp, consistente en el término de búsqueda precedido por un acento circunflejo (^). Este carácter especial representa el inicio de la cadena, por lo que asegura que solamente se tomarán en cuenta las correspondencias en esta posición. Si la palabra clave contiene caracteres adicionales a la derecha, estos serán ignorados. Además, la 'i' añadida al final de la expresión regular, separada por una coma, es un modificador que indica que la comparación entre los términos no sea sensitiva a mayúsculas, de ahí que no importe si el usuario ingresa un término con mayúsculas o minúsculas. Esta diferencia sí constituiría un inconveniente para el código de la sección anterior, ya que no se tomó en cuenta esa posibilidad. El problema de mayúsculas y minúsculas también podría resolverse de otra manera, por ejemplo, mediante la aplicación del método `toLowerCase()` (que convierte todos los caracteres en minúsculas) antes de llevar a cabo la comparación entre el término de búsqueda y la palabra clave[4].

Seguidamente se incluye un bucle *for* (líneas 3–7) que itera sobre la matriz pc, probando si las palabras clave presentan una correspondencia con la expresión regular exp (línea 4). En caso afirmativo, la palabra clave se agrega a la matriz de resultados (línea 5).

```
(8) let resultados = [];
```

4 Por ejemplo, con esta línea al principio de la función: búsqueda = búsqueda.toLowerCase();

```
let exp = new RegExp('^' + búsqueda, 'i');
for (let i = 0; i < pc.length; i++) {
if (pc[i].match(exp)) {
resultados.push(pc[i]);
}
}
```

Cabe señalar que al utilizar esta función también debe incluirse una prueba de que el término de búsqueda no sea una cadena de texto vacía, lo cual podría suceder si el usuario pulsa Intro o el botón de búsqueda sin haber ingresado nada en el campo de texto; esto resultaría en una expresión regular con la que cualquier otra cadena haría correspondencia, y todos los lemas en el diccionario se agregarían a la lista de resultados. Una de varias soluciones es la que se propone en el ejemplo 9. Estas líneas se añadarían al principio de la función de búsqueda, la cual retornaría el valor *false* si el campo de texto está vacío:

(9)
```
if(búsqueda == "") {
 return false;
}
```

Ahora bien, para identificar correspondencias al final de las palabras clave, la segunda línea del ejemplo 8 se sustituiría por la del ejemplo 10. En este caso, la expresión regular consiste en el término de búsqueda seguido por el carácter "$", que representa el final de la cadena. Así, se agregarán a la lista de resultados todas las palabras clave que terminen con el término de búsqueda. Verbigracia, el término "ción" tendría correspondencia con todas las palabras acabadas en dicho sufijo.

(10)
```
let exp = new RegExp(búsqueda + '$', 'i');
```

Por último, para identificar correspondencias en cualquier parte de la cadena, es decir, al inicio, al final o en posición intermedia, se construye una expresión regular que consiste únicamente en el término de búsqueda; es decir, la línea presentada en el ejemplo 11 sustituiría la primera línea del ejemplo 8. De este modo, se incluirán en los resultados todas las palabras clave que contengan este término en cualquier posición.

(11)
```
let exp = new RegExp(búsqueda, 'i');
```

2.3 Correspondencia con comodín

Los buscadores de la mayoría de los diccionarios que hemos analizado usan alguna variante de un tipo de correspondencia que aquí llamamos "correspondencia con comodín". Entre estos, se encuentran el *Diccionario de la lengua española*, *Oxford Dictionaries*, *Cambridge Dictionary*, *Merriam-Webster Online Dictionary*, *WordReference* (Kellogg, 2018) y *Dictionary.com* (Dictionary.com, 2018).

En lo que sigue del presente apartado nos centraremos en el buscador del *Diccionario de la lengua española* y trataremos de reproducir su funcionalidad. Aquí, una palabra clave se considera una correspondencia si es completamente equivalente al término de búsqueda, pero también si es equivalente excepto por el hecho de que

- contiene (máximo) un carácter adicional en cualquier posición,
- contiene (máximo) un carácter menos en cualquier posición,
- o (máximo) uno de los caracteres es diferente.

Para ejemplificar esta funcionalidad, realizamos una búsqueda con el término "tedo", que no corresponde completamente a ninguna de las palabras clave registradas en la obra (aunque en este diccionario también se incluyen formas conjugadas como palabras clave). A continuación se citan algunos de los resultados de la búsqueda, con la explicación de la correspondencia entre paréntesis:

- *aedo* (se cambia la *t* por *a*)
- *ceder* (se cambia la *t* por *c*, lo cual da correspondencia con la forma conjugada *cedo*)
- *dedo* (se cambia la *t* por *d*)
- *-edo* (se elimina la *t*)
- *teda* (se cambia la *o* por *a*)
- *tedio* (se agrega una *i*)
- *tejer* (se cambia la *d* por *j*, lo cual da correspondencia con la forma conjugada *tejo*)
- *tero* (se cambia la *d* por *r*)

• *todo* (se cambia la *e* por *o*)

• *vedar* (se cambia la *t* por *v*, lo cual da correspondencia con la forma conjugada *vedo*)

Es evidente que la función que arroja estos resultados lleva a cabo una comparación entre las palabras clave y distintas versiones del término de búsqueda. Las variaciones se producen por medio de la inserción de un comodín en todas las posiciones. De esa manera, si se halla una correspondencia completa con algunas de las versiones modificadas del término de búsqueda, se agrega a la lista de resultados el lema asociado con la palabra clave en cuestión.

El código de esta búsqueda es más extenso que los anteriores, por lo que lo dividiremos en varias partes. El comienzo se aprecia en el ejemplo 12, donde se declaran cuatro variables y una constante. Las primeras dos variables son matrices vacías, que posteriormente se llenarán; una para los resultados de la búsqueda, y otra para todas las variaciones del término de búsqueda. Las dos siguientes, `parte1` y `parte2`, se utilizarán varias veces en el proceso de creación de las variantes del término de búsqueda: servirán para almacenar temporalmente los dos fragmentos del término entre los que se inserta el comodín antes de volverlos a fusionar. A la constante, denominada `longitud`, se le asigna inmediatamente el número corresponiente a la cantidad de caracteres contenidos en el término de búsqueda, ya que este número será empleado en varias ocasiones más adelante.

(12)
```
z
```

Seguidamente se agregan las líneas del ejemplo 13. Aquí se programa un bucle *for* que itera sobre los caracteres del término de búsqueda y coloca un comodín en todas las posiciones, incluidas las posiciones inicial y final. El comodín, que corresponde a cualquier carácter, es representado por un punto en la expresión regular. Obsérvese la presencia del " ^ " al inicio y del "$" al final de la expresión, caracteres que aseguran que no se registrarán correspondencias con palabras clave que difieran en más de un carácter.

(13)
```
for (let i = 0; i <= longitud; i++) {
    parte1 = búsqueda.substr(0, i);
    parte2 = búsqueda.substr(i, longitud);
    variantes.push(new RegExp('^' + parte1 + '.' + parte2 +
    '$', 'i'));
```

```
}
```

Una palabra clave que contenga exactamente un carácter más que el término de búsqueda original (con todos los demás caracteres iguales), siempre tendrá correspondencia con una de las cadenas de caracteres creadas en el ejemplo 13. De este modo, el término de búsqueda "ela" daría los resultados *tela*, *vela*, *ella*, etcétera.

En segundo lugar, se tienen que considerar todas las variantes del término de búsqueda con un comodín en sustitución de cada uno de los caracteres. Además, en este caso, el comodín también puede representar la ausencia de un carácter, de manera que se detecten correspondencias no solo con palabras clave en las que una letra sea diferente al término de búsqueda, sino también con palabras clave que contengan una letra menos (en cualquier posición). La opcionalidad de un carácter (en este caso, el comodín) se señala en la expresión regular mediante un signo de interrogación, como en la línea 4 del código mostrado a continuación:

```
(14) for (let i = 0; i < longitud; i++) {
     parte1 = búsqueda.substr(0, i);
     parte2 = búsqueda.substr(i+1, longitud);
     variantes.push(new RegExp('^' + parte1 + '.?' + parte2 +
     '$', 'i'));
     }
```

Con el término de búsqueda "ela", las variaciones creadas en el ejemplo 14 harían correspondencia con *ala*, *ola*, *la*, *era*, *esa*, *el*, etcétera.

Lo que queda pendiente de la función es la comparación de todas las palabras clave con todas las variantes del término de búsqueda. Esta se aprecia en el ejemplo 15[5]. Al igual que en los ejemplos anteriores, las correspondencias se agregan a la matriz resultados, la cual es retornada al final de la función.

```
(15) for (let i = 0; i < pc.length; i++) {
     for (let j = 0; j < variantes.length; j++) {
     if (pc[i].match(variantes[j])) {
      resultados.push(pc[i]);
```

5 La declaración `break` se añade para pasar inmediatamente a la palabra clave siguiente después de haber hallado una correspondencia con una de las variantes del término de búsqueda. Si no se incluye esta línea, un mismo lema podría aparecer varias veces en la lista de resultados.

```
 break;
}
}
}
return resultados;
```

Cabe señalar que el buscador del *Diccionario de la lengua española* también incluye algunas instrucciones adicionales que toman en consideración ciertos errores ortográficos frecuentes; por ejemplo, la búsqueda "zalza" solo da los resultados *salsa* y *salso*/*a*, a pesar de que presenta mayor similitud ortográfica con la palabra *zarza*. Esto sucede porque el buscador realiza una sustitución de *z* por *s* en caso de no encontrar ninguna correspondencia completa. Otro ejemplo es la *w*, la cual es tratada como *gu* o *hu* en la búsqueda de correspondencias. Excepciones de este tipo pueden programarse por medio de la estrategia de sustitución de caracteres detallada en el siguiente apartado.

3. Sustitución de caracteres

En esta sección se describe una función adicional, cuyo propósito es sustituir caracteres en el término de búsqueda y en las palabras clave antes de realizar la comparación entre ambas cadenas de texto. La función tiene varios usos; por ejemplo, permitir que dos o más caracteres se consideren equivalentes, ignorar diacríticos o caracteres superfluos e, incluso, realizar conversiones entre un sistema de escritura y otro.

La función toma una cadena de caracteres como argumento y retorna una versión modificada de dicha cadena, por lo que le pondremos el nombre `mod()`. De esta forma, la llamada `mod(búsqueda)` se puede utilizar dentro de la función `buscar()` para obtener la versión modificada de la cadena de texto asignada a la variable `búsqueda`.

En el ejemplo 16 se proporciona una posible implementación de la función `mod()`, donde el argumento (la cadena de texto por modificar) es asignado a la variable `x`. La primera línea dentro de la función realiza sustituciones mediante una expresión regular: cualquier carácter no incluido entre los paréntesis cuadrados (en este caso, el acento circunflejo es un carácter especial que indica negación) es sustituido por una cadena de texto vacía (representada por dos comillas sencillas); es decir, es elidido. De este modo, se eliminan signos de puntuación u otros caracteres que no pertenezcan a la ortografía del español, por si el usuario ingresa alguno de estos elementos por equivocación. La "g" a la derecha de la barra inclinada indica que la sustitución sea global, lo cual quiere decir que no se realice solo una vez, sino para todos los elementos ajenos

encontrados en la cadena. Además, se añade la "i", lo que asegura que también las mayúsculas sean sustituidas.

```
(16) function mod(x) {
     x = x.replace(/[^aábcdeéfghiíjklmnñoópqrstuúvwxyz]/gi,
     '');
     return x;
     }
```

Una función para encontrar correspondencias completas podría, por lo tanto, incluir la línea del ejemplo 17 antes de llevar a cabo la comparación entre las palabras clave y el término de búsqueda. En esta se declara una nueva variable con el valor de una expresión regular consistente en el término de búsqueda modificada y que, además, ignora las diferencias entre mayúsculas y minúsculas:

```
(17) let exp = new RegExp('^' + mod(búsqueda) + '$', 'i');
```

La comparación entre las palabras clave y esta expresión regular se programaría de la siguiente manera (dentro de los paréntesis de la sentencia condicional *if*):

```
(18) pc[i].match(exp)
```

Ahora bien, podría ser conveniente ignorar todos los diacríticos en la búsqueda, sobre todo si es probable que el usuario, por algún motivo, no los ponga. En ese caso, cada carácter con diacrítico puede sustituirse por el correspondiente sin diacrítico. Por tanto, en el caso de un diccionario del español, la función `mod()` podría definirse como en el ejemplo 19. Las primeras líneas se encargan de la sustitución de cada una de las letras tildadas, mientras que la última sustitución es una versión abreviada de la eliminación de caracteres superfluos apreciada anteriormente en el ejemplo 16. En esta ocasión, todas las letras del alfabeto pueden representarse como `[a-z]`, debido a que la cadena de texto no incluirá letras tildadas.

```
(19) function mod(x) {
     x = x.replace(/á/gi, 'a');
     x = x.replace(/é/gi, 'e');
     x = x.replace(/í/gi, 'i');
     x = x.replace(/ó/gi, 'o');
     x = x.replace(/ú/gi, 'u');
```

```
x = x.replace(/ñ/gi, 'n');
x = x.replace(/[^a-z]/gi, '');
return x;
}
```

Dado que las palabras clave sí incluirían diacríticos, aquí también tendrían que eliminarse temporalmente a la hora de realizar la comparación con el término de búsqueda. Por consiguiente, la comparación debe especificarse como en el ejemplo 20:

(20) `mod(pc[i]).match(búsqueda)`

Esto asegura que los diacríticos no se tomen en cuenta en la comparación, pero la modificación no afecta permanentemente el valor de `pc[i]`, por lo que las cadenas de texto agregadas a la lista de resultados sí llevarían todos los diacríticos.

En el caso de la lengua bribri, los diacríticos representan un reto importante para el buscador del diccionario, ya que se utilizan varias combinaciones de diacríticos que no se pueden reproducir con un teclado de configuración estándar. Verbigracia, se dan combinaciones de diéresis con otro diacrítico sobre una vocal, como en la palabra *kö `chi* 'cerdo'. En Unicode, esta combinación puede representarse mediante el carácter ö seguido por el acento grave combinable (U+0300), o bien, por medio de tres caracteres seguidos: la letra *o*, las diéresis combinables (U+0308) y el acento grave combinable. Por su parte, la nasalidad vocálica (la cual es fonémica) es representada por un subrayado, el cual corresponde a alguno de los caracteres combinables U+0320, U+0331 o U+0332. Asimismo, algunos diacríticos representan tonos, los cuales tienden a confundirse a la hora de escribir. Todos estos hechos implican que es muy probable que el término de búsqueda no contenga los mismos diacríticos que la palabra clave correspondiente en el diccionario.

En consecuencia, se decidió ignorar todos los diacríticos en el buscador del diccionario bribri–español, lo que se logra con la función `mod()` presentada en el ejemplo 21. Las primeras cinco líneas sustituyen todas las posibles letras tildadas, mientras que la siguiente elimina todos los caracteres no especificados entre los paréntesis cuadrados, tales como los diacríticos combinables.

(21)
```
function mod(x) {
x = x.replace(/[áàâäã]/gi, 'a');
x = x.replace(/[éèêëẽ]/gi, 'e');
```

```
x = x.replace(/[íìîïĩ]/gi, 'i');
x = x.replace(/[óòôöõ]/gi, 'o');
x = x.replace(/[úùûüũ]/gi, 'u');
x = x.replace(/[^abcdefghijklmnñopqrstuvwxyz]/gi, '');
return x;
}
```

Por otro lado, si el lexicógrafo considera que los usuarios meta tienden a confundir determinadas letras, estas se pueden tratar como si fueran iguales. Suponiendo, por ejemplo, que se desea ignorar la diferencia entre *s* y *z* en el buscador de un diccionario del español, la línea del ejemplo 22 puede incluirse en la función `mod()`. De esta manera, cadenas de texto como *salsa* y *zalza* se considerarían iguales. Como en el caso de los diacríticos, la lista de resultados no sería afectada por la sustitución, de ahí que la búsqueda "zalza" daría *salsa* como resultado.

(22)
```
x = x.replace(/z/gi, 's');
```

Similarmente, la letra muda *h* puede ignorarse a la hora de realizar la búsqueda mediante la siguiente línea en la función mod():

(23)
```
x = x.replace(/h/gi, '');
```

La misma estrategia sirve para permitir dígrafos en lugar de caracteres especiales. Esto es relevante para muchas lenguas, por ejemplo, el alemán, cuya ortografía incluye los caracteres "ä", "ö", "ü" y "ß"; cuando estos no se encuentran disponibles en el teclado, suelen ser sustituidos por "ae", "oe", "ue" y "ss", respectivamente. Las líneas presentadas en el ejemplo 24 permiten que el usuario digite los dígrafos, en lugar de "ä", "ö", "ü" y "ß", en el campo de búsqueda. Al igual que en los demás casos, es importante que tanto el término de búsqueda como la palabra clave sean modificados antes de la comparación entre ambos.

(24)
```
x = x.replace(/ä/gi, 'ae');
x = x.replace(/ö/gi, 'oe');
x = x.replace(/ü/gi, 'ue');
x = x.replace(/ß/gi, 'ss');
```

De hecho, se pueden utilizar expresiones regulares simples para permitir *input* en sistemas de escritura diferentes, siempre y cuando cada carácter del sistema alternativo

invariablemente corresponda al mismo carácter (o conjunto de caracteres) en el sistema utilizado en el diccionario. Por ejemplo, para un diccionario del serbio —idioma escrito tanto con el alfabeto latino como con el cirílico— las entradas podrían registrarse únicamente en el alfabeto latino, puesto que una serie de expresiones regulares como las que se presentan en el ejemplo 25 permitirían que las búsquedas se ingresaran en cualquiera de los dos sistemas de escritura; todos los caracteres cirílicos se convertirían en los correspondientes del alfabeto latino serbio antes de realizarse la búsqueda:

(25)
```
x = x.replace(/ä/gi, 'c');
x = x.replace(/ö/gi, 'č');
x = x.replace(/ü/gi, 'dž');
x = x.replace(/ß/gi, 'š');
...
```

4. Conclusiones

En este artículo se han presentado varias posibles estructuras para la programación de motores de búsqueda en JavaScript. Como fue señalado al principio del escrito, la intención ha sido proporcionar explicaciones lo suficientemente claras para que incluso las personas sin conocimientos previos de la programación logren comprender el procedimiento. De esta manera, el lector también sería capaz de modificar las funciones de acuerdo con sus propias necesidades, puesto que solo sería necesario realizar modificaciones mínimas a los fragmentos de código proporcionados. El conocimiento sobre JavaScript y expresiones regulares también es una base importante para la programación de buscadores en otros lenguajes de programación, ya que, por lo general, la sintaxis no varía excesivamente entre un lenguaje y otro.

Referencias

Dictionary.com. (2018). *Dictionary.com*. Recuperado de http://www.dictionary.com.

Ediciones Larousse. 2017. *Larousse Diccionario Avanzado de Español*. Recuperado de https://www.larousse.mx/app/larousse-diccionario-avanzado-de-espanol/.

Ediciones SM. (2014). *Clave: diccionario de uso del español actual*. Madrid: Ediciones SM.

Flanagan, D. (2011). *JavaScript: The Definitive Guide* (6ª edición). Sebastopol: O'Reilly Media.

Groner, L. & Manricks, G. (2015). *JavaScript Regular Expressions*. Birmingham/Mumbai: Packt Publishing.

Kellogg, M. (2018). *WordReference.com*. Recuperado de http://www.wordreference.com.

Merriam-Webster. (2018). *Merriam-Webster Online Dictionary*. Recuperado de http://www.merriam-webster.com.

Mitkov, R. (Ed.) (2003). *The Oxford Handbook of Computational Linguistics*. Oxford: Oxford University Press.

Oxford University Press. (2018). *Oxford Dictionaries*. Recuperado de http://www.oxforddictionaries.com.

Real Academia Española. (2018). *Diccionario de la lengua española*. Recuperado de http://www.rae.es.

Subramaniam, V. (2018). *Rediscovering JavaScript. Master ES6, ES7, and ES8*. The Pragmatic Progra