Codificación UTF-8

Diego José Abengózar Vilar, Julio César Castro López

26/3/2021

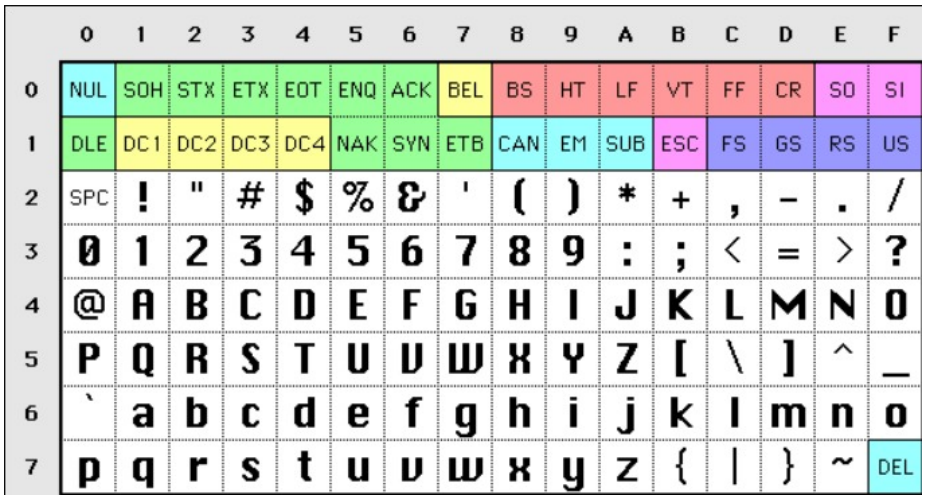
Tabla de contenidos

1 Contexto histórico

En los años 60 y 70, cuando el uso de los ordenadores no era tan universal como ahora, se utilizaban unos pocos caracteres: letras del alfabeto romano sin tildes. Eran tan pocos símbolos que se podían codificar perfectamente en 7 bits, en un código llamado ASCII.

De hecho, como sobraba 1 bit para llegar a la unidad tradicional de 1 byte, los vendedores de PC de la época empezaron a darle uso a este último bit para permitir, por ejemplo, el uso de letras con tilde en el caso de muchos países europeos, o caracteres hebreos en los ordenadores vendidos en Israel. En muchos PCs europeos el carácter 130 se visualizaba como 'é" mientras que en los ordenadores de Israel se visualizaba como 'ג". Esto tenía el problema de que, si se enviaba un documento codificado en un ordenador con una cierta especificación a otro con una especificación distinta, se producían traducciones erróneas. El problema era aún mayor en países asiáticos, ya que el número de símbolos de sus lenguajes no cabían en 8 bits.

Con la llegada de Internet, esta codificación era insuficiente y en 1991 surgió *Unicode* para includir todos los caracteres de cualquier sistema de escritura del planeta. Cada símbolo está asociado a un punto de código (*code point*), que se suele escribir de la forma $U+XXXX$, donde $XXXX$ son dígitos hexadecimales,



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 1: Tabla ASCII

y que luego se trasladará a un cierto número de bytes. Al principio se decidió utilizar 2 bytes para cada punto de código, pero pronto se dieron cuenta de que, en muchas ocasiones, las cadenas estaban llenas de ceros pues sobraba mucho espacio para los caracteres de uso común. Además, al utilizar 2 bytes podía haber problemas para determinar el orden de los mismos (*endianness*).

Para solucionar estos problemas se inventó UTF-8, que utiliza un número variable de bloques de 1 byte.

2 Desarrollo Teórico

2.1 Especificación

UTF-8 es una codificación de tamaño variable: codifica la información utilizando entre 1 y 4 octetos. Está pensado además para ser una correspondencia perfecta con el código ASCII, es decir, un texto codificado con ASCII tendría la misma codificación en UTF-8 y no hay ningún otro texto codificado en UTF-8 que pudiera ser interpretado como ASCII.

Los caracteres Unicode se designan con la notación $U+XXXX$, donde $XXXX$ es una cadena de entre 4 y 6 dígitos hexadecimales.

El número de octetos que se emplean dependen del caracter a codificar:

- Los caracteres con números desde $U + 0000$ hasta $U + 007F$ se codifican con un octeto de bits que va desde el valor $0x00$ hasta $0x7F$ (0 - 127 en decimal). Nótese que con 7 bits se pueden codificar 128 valores ($2^7 = 128$), por lo que la cifra más significativa del octeto toma valor 0 y con las otras

siete se codifica el valor correspondiente. Esto es una correspondencia exacta con ASCII.

- Los caracteres con números desde $U + 0080$ hasta $U + 07FF$ se codifican con dos octetos (128 - 2047 en decimal). Los primeros tres bits del primer octeto son 110 y los dos primeros bits del segundo octecto son 10. Con los 11 bits restantes basta para codificar el valor correspondiente.
- Los caracteres con números desde $U + 0800$ hasta $U + FFFF$ se codifican con tres octetos (2048 - 65535 en decimal). Los primeros cuatro bits del primer octeto son 1110 y los dos primeros bits de los octetos siguientes son 10. Con los 16 bits restantes se codifica el valor correspondiente.
- Los caracteres con números desde $U + 10000$ hasta $U + 10FFFF$ se codifican con tres octetos (65536 - 1114111 en decimal). Los primeros cinco bits del primer octeto son 11110 y los dos primeros bits de los octetos siguientes son 10. Con los 21 bits restantes se codifica el valor correspondiente.

En la siguiente tabla presentamos un resumen de los disintos formatos.

Rango de números	Tipo de codificación
$0x00$ - $0x7F$	$0xxxxxxx$
$0x80$ - $0x7FF$	$110xxxxx$ $10xxxxxx$
$0x800$ - $0xFFFF$	$1110xxxx$ $10xxxxxx$ $10xxxxxx$
$0x10000$ - $0x10FFFF$	$11110xxx$ $10xxxxxx$ $10xxxxxx$ $10xxxxxx$

Como vemos, el primer octeto empieza con tantos 1 como número de octetos tiene la codificación (salvo cuando solo hay uno), y los posteriores octetos empiezan con 10. Se codifican todos los valores del rango $0x0$ hasta $0x10FFFF$, lo que da cabida a más de un millón de caracteres.

2.1.1 Codificación

1. Se determina el número de octetos a partir del número de caracter.
2. Se colocan los bits más significativos según la tabla anterior.
3. Para rellenar el resto de bits se toma el bit menos signifcativo del número de caracter a codificar y se coloca en la posición menos significativa del último de los octetos. A continuación se toma el segundo bit menos significativo del número de caracter a codificar y se coloca en la segunda posición menos significativa del último de los octetos. Se sigue con este procedimiento hasta rellenar todos los bits, pasando al octeto siguiente cuando corresponda y completando con 0 si es necesario.

Ejemplo

Si queremos codificar $U + 2764$, primero observamos que $0x0800 < 0x2764 <$

$0xFFFF$, y por tanto tiene 3 octetos:

$$1110xxxx\ 10xxxxxx\ 10xxxxxx$$

$0x2764$ en binario es 0010011101100100 y colocamos los bits menos significativos en la posiciones menos significativas:

$$1110\mathbf{0010}\ 10\mathbf{011101}\ 10\mathbf{100100}$$

2.1.2 Decodificación

1. Se inicializa un número binario con todos los bits a 0. Nótese que no hacen falta más de 21 bits.
2. Se determina el número de octetos y qué posiciones contienen la información del caracter codificado.
3. Se colocan los bits desde la posición menos significativa del último octeto hasta la más sigificativa del primer octeto en nuestro número binario en ese mismo orden, es decir, empezando por los bits menos significativos.

Ejemplo

Si nos dan 111000101000100010011110, determinamos que tiene 3 bloques ya que empieza por 111 e identificamos los bits que codifican la información:

$$1110\mathbf{0010}\ 10\mathbf{001000}\ 10\mathbf{011110}$$

Nos queda el número 0010001000011110, lo pasamos a hexadecimal:

$$0010\ 0010\ 0001\ 1110 = 0x221E$$

Por tanto el caracter UTF-8 codificado es $U + 221E$.

Supongamos ahora que nos dan la secuencia $0xC0AF = 1100000010101111$. Si la decodificamos siguiendo el procedimiento anterior:

$$110\mathbf{00000}\ 10\mathbf{101111}$$

$$000\ 0010\ 1111 = 0x02F = U + 002F$$

Sin embargo, si nos fijamos bien, el caracter $U + 002F$ ($0x2F = 101111$) lo habríamos codificado de la siguiente manera:

$$0xxxxxxx$$

$$00101111 = 0x2F$$

¿Hay acaso dos secuencias tales que al decodificarlas dan el mismo caracter? No, la implementación solo debe tomar como secuencia válida la más pequeña que codifica el caracter. La secuencia $0xC0AF$ del ejemplo es conocida como una *overlong sequence* y no es aceptada como válida por la especificación. El estándar también prohibe la codificación de los caracteres entre $U + D800$ y $U + DFFF$.

Otra consideración a hacer es el carácter $U + FEFF$, conocido como *Byte Order Mark*. Si se encuentra al comienzo, sirve para indicar el orden de los bytes (*endianness*). Esto es relevante para otras codificaciones Unicode que utilizan bloques de 16 o 32 bits, aunque no tiene mayor importancia en el caso de UTF-8.

3 Desarrollo Informático

Hemos implementado una aplicación web de visualización de los procesos de codificación y decodificación en UTF-8.

En cuanto a las tecnologías usadas, hemos empleado Angular como framework para las interfaces gráficas. Por otra parte, hemos usado GitLab como servicio de control de versiones. El código fuente se puede encontrar en el siguiente repositorio público: https://gitlab.com/codificacion-criptografia/utf.

Hemos aprovechado el servicio gratuito de hospedaje de aplicaciones web de GitLab, para que así esta sea accesible por cualquier persona que disponga de un navegador web. Para poder usarla basta ir a la siguiente dirección: https://codificacion-criptografia.gitlab.io/utf/.

3.1 Implementación

Más allá de las tareas relacionadas con las propias interfaces gráficas (maquetación de las páginas, aplicación de los estilos, navegación, etc), en lo que respecta al propio proceso de codificación y decodificación, nos hemos apoyado en la librería utf8.js [3], la cual hemos utilizado principalmente para facilitar la tarea de obtención de la representación binaria de los caracteres codificados y, también, para detectar errores en el proceso de decodificación relacionados con secuencias no asignadas por Unicode. No obstante, toda la parte visual correspondiente al coloreado de posiciones importantes, detección de otros errores en la decodificación, entre otras tareas, se han llevado a cabo por implementación propia, la cual se puede apreciar en el repositorio público que acabamos de comentar.

3.2 Visualización

La aplicación cuenta principalmente con dos funcionalidades: codificación y decodificación.

Empezando por la parte de codificación, contamos con una pantalla en la que nos pide ingresar un carácter cualquiera. Este proceso cuenta con validación: solo se permite ingresar un único carácter. Además, como en el resto de partes de la aplicación, el contenido es dinámico, si no hay carácter introducido o no es válido la información correspondiente no se va a mostrar y las opciones asociadas se deshabilitarán.

Una vez introducimos un carácter se nos mostrará el código Unicode asociado y se habilitará la opción de codificar, esto lo podemos apreciar en la figura 2. Un punto interesante es que, junto con el código Unicode, damos la posibilidad al usuario a consultar más información sobre el carácter introducido, esto lo conseguimos redirigiendo, si hace click en la caja en cuestión, a la entrada correspondiente de la aplicación Unicode Character Table [4].

Si hacemos click en el botón de codificar, podremos ver, mediante una serie de pasos, el proceso para representar el carácter introducido en utf-8, que, dependiendo del carácter se requerirán más o menos bytes. En cualquier caso, salvo cuando el carácter entra dentro del rango de ASCII, podremos ver resaltados con colores las posiciones importantes del proceso, así como pequeñas explicaciones de lo que se está haciendo. Podemos encontrar un ejemplo en la figura 3

Por otra parte, en cuanto a la sección de decodificación, se dispone otra vez con un campo para introducir información, esta vez una secuencia de bits. Dicho campo, al igual que antes, cuenta con su propia lógica de validación: no se podrán introducir caracteres que no sean ceros o unos y contamos con una longitud máxima de hasta 32 bits (el campo de texto como tal no tiene límite de caracteres, pero, al momento de empezar el proceso de decodificación, el programa se queda con los primeros 32, desechando el resto). Cabe destacar también que está implementado un sencillo sistema de formateo de los bits, para agruparlos en bytes, es decir, de 8 en 8 para que sean más fáciles de leer y editar. Además, también se permite pegar alguna secuencia de bits que tengamos en el portapapeles. De hecho, para comprobar que la aplicación se comportaba correctamente hemos probado a codificar un carácter, copiado la secuencia de bits devuelta por el programa y comprobado que la decodificación de esa misma secuencia devolvía el carácter del principio. Podemos apreciar la página en la figura 4.

Aparte de las validaciones puede ocurrir que la secuencia de bits introducida no corresponda con una secuencia utf-8 válida, esto se puede manifestar de diferentes formas: el número de bits no es una secuencia de bytes (no es múltiplo de 8), el primer byte no tiene la cabecera correcta, los bytes intermedios no tienen la cabecera correcta, las secuencias de un byte (correspondientes a ASCII) no tienen su primer bit a 0 o bien que directamente el código Unicode que se puede extraer de la secuencia no se encuentra asignado a algún carácter. Todos y cada uno de estos errores, salvo el último, cuentan con una pequeña explicación para poder solucionarlo, así como un pequeño apoyo visual, el cual varía dependiendo del error. Podemos apreciar un ejemplo en la figura 5.

Finalmente, cuando la secuencia es correcta, y, de forma análoga al proceso de codificación se ilustra el proceso con una serie de pasos y colores que resaltan las posiciones importantes. Por otra parte, mostramos el carácter asociado y, de nuevo, una referencia a la entrada correspondiente en Unicode Table Source [4]. Podemos apreciar un ejemplo en la figura 6. Para finalizar, puede ocurrir que el carácter correspondiente no esté soportado por la fuente que usamos en la aplicación, especialmente si es un carácter menos habitual, esto provoca que se muestre, como sustitución, un símbolo parecido a este: □. Esto lo indicamos también al usuario con un pequeño botón de información junto con la botonera de funcionalidades.

4 Aportaciones personales y conclusiones

Este es el título de la sección que propone el profesor. A saber qué poner xD.

Referencias

[1] F. Yergeau, *RFC 3629 - UTF-8, un formato de transformación de ISO 10646*, Internet Society, 2003

[2] Joel Spolsky, *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*, 2003

[3] Librería utf8.js, https://github.com/mathiasbynens/utf8.js.

[4] Unicode Character Table, https://unicode-table.com/.

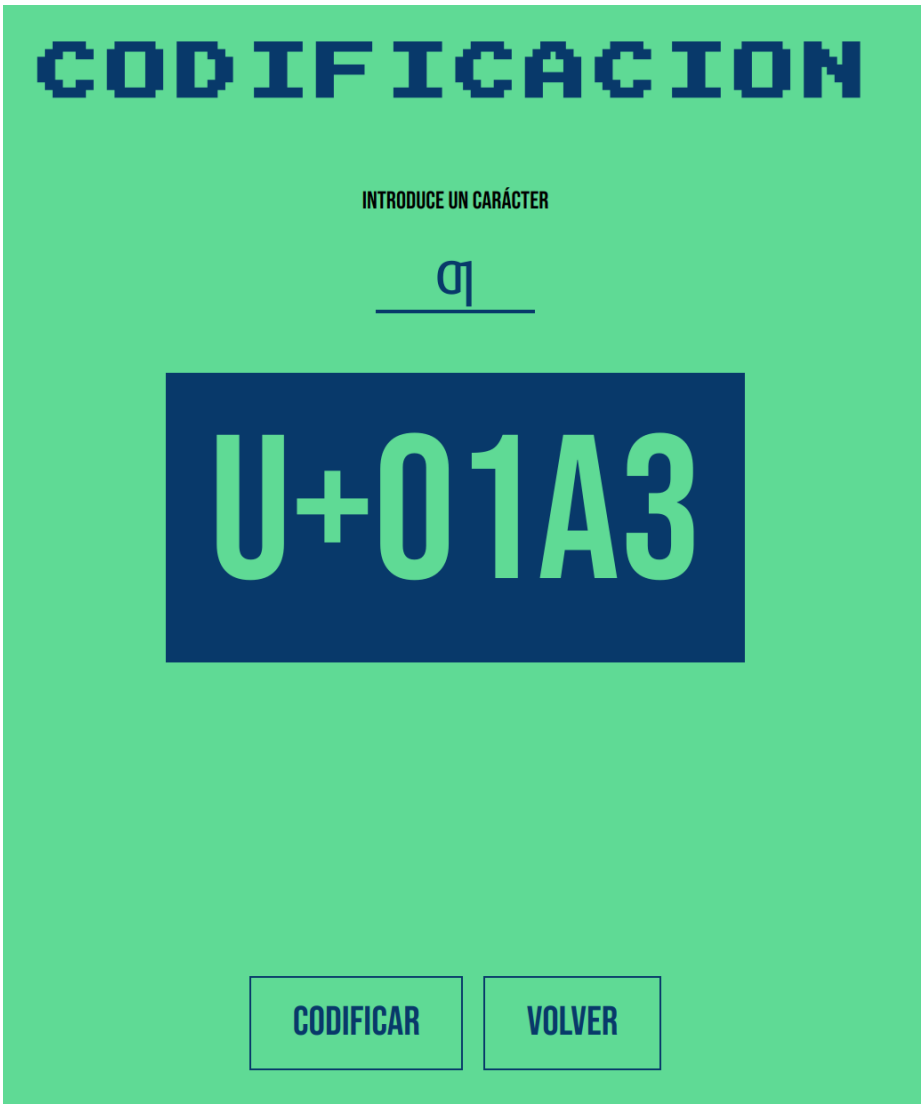


Figure 2: Página de Codificación

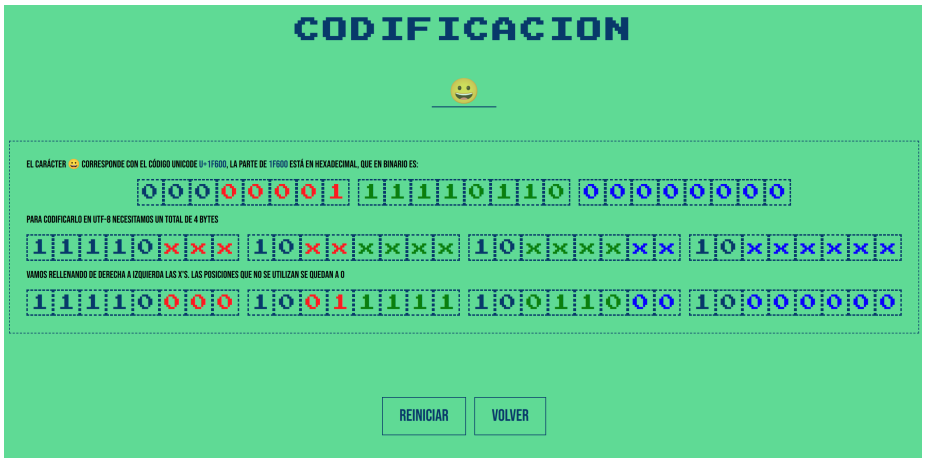


Figure 3: Codificación de un carácter

Figure 4: Página de decodificación

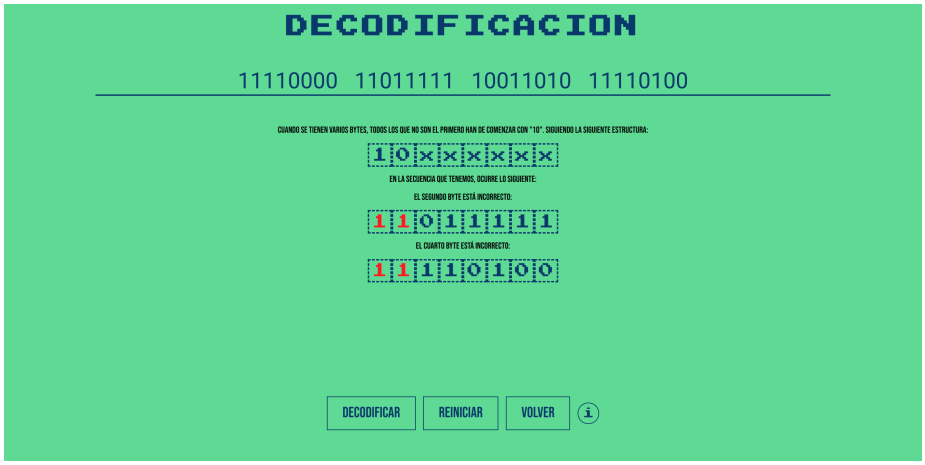


Figure 5: Error en decodificación de un carácter

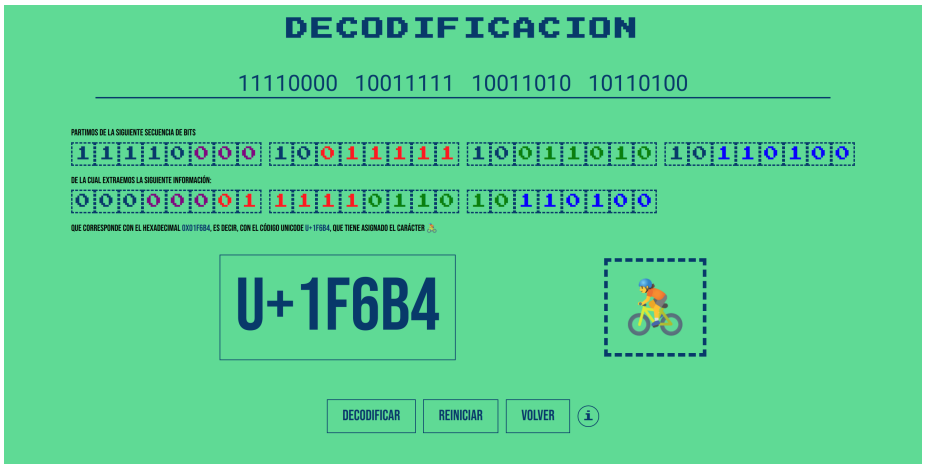


Figure 6: Decodificación de un carácter