
FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

# Desarrollo de una librería Haskell sobre códigos QR

Realizado por
Javier Delgado Cruces

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Junio de 2021

## Abstract

QR codes are the most common way of storing and exchanging information. In this project they will be studied in order to create a Haskell package with necessary functions and data types to implement them.

## Agradecimientos

*A todas las personas que, de un modo u otro, han ayudado a que este trabajo salga adelante.*

# Índice general

# CAPÍTULO 0

## Introducción

Los Códigos QR (del inglés Quick Response Code, código de respuesta rápida) son un tipo de código de barras bidimensional diseñado por la multinacional japonesa Denso Wave en 1994. Sus principales características son: la gran cantidad de datos que son capaces de almacenar en comparación con otros códigos similares como los códigos de barras, su rapidez de lectura, la capacidad de leer códigos girados, dañados o incompletos y ser de código abierto. Todo esto ha contribuido a que su uso transcienda el propósito original que se les dio y se hayan convertido en la opción más popular para almacenar y transmitir información.

El objetivo de este trabajo es crear una librería en el lenguaje de programación Haskell que contenga las funciones y los tipos necesarios para poder operar con los código QR. Para ello, vamos a dividir el trabajo de la siguiente manera:

- En el primer capítulo, estudiaremos las características y especificaciones técnicas de los códigos QR. La principal referencia que seguiremos será [3].
- En el segundo capítulo, comentaremos la implementación de la librería. El orden que seguiremos será similar al del capítulo anterior. Esto permitirá encontrar rápidamente la explicación teórica de la parte que estemos implementando en cada momento o leer ambos capítulos en paralelo. En cualquier caso, todos los archivos del código pueden encontrarse de manera íntegra y ordenada en

  `https://github.com/AlStinson/QR` .
- El tercer capítulo es el manual de uso de la librería. En él, encontraremos la explicación de como usar la librería, de forma que puede utilizarse sin tener que leer el trabajo completo.
- En el último capítulo, presentaremos las conclusiones y el trabajo futuro.

# CAPÍTULO 1

# Desarrollo teórico

Un QR es una matriz cuadrada de módulos de dos colores distintos, normalmente uno claro y otro oscuro, de forma que se puedan distinguir de forma sencilla. Se utilizará blanco para hacer referencia a los módulos de color claro y negro para los oscuros.

## 1.1– Versiones

Hay 44 versiones de códigos QR. Las 4 primeras (desde la M1 a M4) se denominan Micro QR y funcionan de forma un poco distinta a las 40 versiones restantes (desde la 1 hasta la 40), que se denominan QR. Esto puede dar lugar a confusión, pues por código QR puede entenderse tanto todas las versiones, como solo las versiones de la 1 a la 40. Por tanto, cuando no haya ninguna referencia a Micro QR, debe entenderse que se hace mención a todas las versiones, mientras que, cuando sí la haya, solo se hace mención a las versiones de la 1 a la 40.

Las dimensiones de la matriz que representa al QR dependen de la versión escogida. En la tabla 1.1 se puede encontrar las dimensiones de cada versión. Hay que tener en cuenta que existe una fuerte relación entre las dimensiones de la matriz y la cantidad de datos que es capaz de almacenar el QR, de forma que se podrán codificar más datos con una versión con un número mayor.

| Versión | Dimensiones | Versión | Dimensiones |
|---|---|---|---|
| M1 | 11×11 módulos | 1 | 21×21 módulos |
| M2 | 13×13 módulos | 2 | 25×25 módulos |
| M3 | 15×15 módulos | n | (17+4n)×(17+4n) módulos |
| M4 | 17×17 módulos | 40 | 177×177 módulos |

Tabla 1.1: Dimensiones del QR según la versión

## 1.2– Estructura

Algunos de los módulos de la matriz del QR forman parte de los patrones requeridos y siempre toman el mismo valor, mientras que los demás forman parte de la zona de codificación, que es la que se encarga de transmitir los datos.

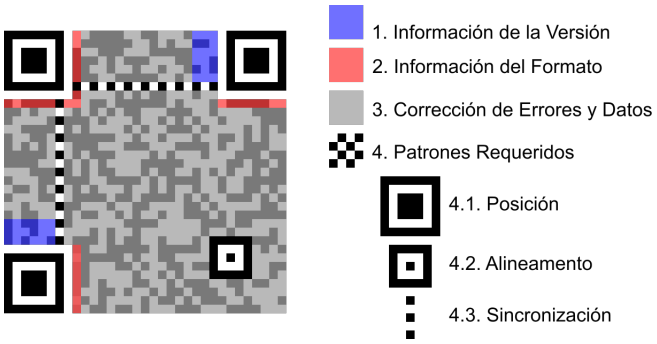


Figura 1.1: Estructura de un QR
*Autor: Luis Atala*

### 1.2.1. Patrones requeridos

Los patrones requeridos son una serie de módulos que siempre toman los mismos valores y que permiten la correcta detección de los demás módulos.

#### Patrón de posición

Consiste en un cuadrado de 7×7 módulos, de forma que el cuadrado de 3×3 módulos central y los módulos del borde son negros y los demás blancos. Se colocan un total de 3, uno en la esquina superior derecha, otro en la esquina superior izquierda y el último en la esquina inferior izquierda. Además, se separan del resto del QR dejando en blanco los módulos adyacentes a cada patrón de posición. En los Micro QR solo se coloca uno en la esquina superior izquierda.

#### Patrón de alineamiento

Parecido al patrón de posición, pero más pequeño. Consiste en un cuadrado de 5×5 módulos, de forma que el módulo central y los módulos del borde son negros y los demás blancos. La cantidad que se coloca y su posición dependen de la versión escogida. Los valores concretos se pueden encontrar en la tabla E.1 de [3]. Los Micro QR no incluyen patrones de alineamiento.

#### Patrón de sincronización

Hay dos tipos: uno horizontal, con una única fila de alto, que se coloca en la séptima fila y otro vertical, con una única columna de ancho, que se coloca en la séptima columna. En ambos está formado por módulos blancos y negros, colocados de forma alterna y

empezando por uno negro. En los módulos donde coincide con alguno de los patrones de posición, tiene prioridad lo que dicta el patrón de posición. En los Micro QR se colocan en la primera columna y primera fila respectivamente.

### 1.2.2. Zona de codificación

Esta es la zona encargada de almacenar la información. Esta compuesta por todos los módulos que no forman parte de los patrones requeridos.

#### Información de versión

Solo está presente en la versión 7 y posteriores. Consiste en dos bloques, uno de 6×3 módulos, situado a la izquierda del patrón de posición de la esquina superior derecha, y el otro de 3×6 módulos, situado encima del patrón de posición de la esquina inferior izquierda. En esta zona se encuentra codificada, por repetido en cada bloque, la versión del QR. Esto ayuda a que se detecte de forma correcta en las versiones con muchos módulos.

#### Información de formato

Se encuentra repartida en la siguientes zonas:

- 8 módulos en horizontal, justo debajo de patrón de posición situado en la esquina superior derecha,
- todos los módulos adyacentes al patrón de posición situado en la esquina superior izquierda (sin contar los ya ocupados por los patrones de sincronización) y
- 8 módulos en vertical, justo a la derecha del patrón de posición situado en la esquina inferior izquierda.

En los Micro QR solo son los módulos adyacentes al patrón de posición. En esta zona se encuentra codificado por duplicado el nivel de correción de errores y la máscara utilizada.

#### Zona de datos y corrección de errores

Todos los módulos que no se han mencionado anteriormente forman parte de esta zona. En ella va codificada la información que se quiere transmitir junto a los bits de corrección de errores. Cada módulo representa un bit, de forma que los módulos blancos representa un cero y los negros, un uno.

### 1.2.3. Zona tranquila

Consiste en la zona adyacente al QR, fuera del mismo. Esta zona debe estar despejada de otro tipo de simbología (y, por tanto, debe quedar en blanco) para que se puedan detectar correctamente los patrones de posición. Debe tener al menos el ancho de 4 módulos. En los Micro QR solo es necesario que tenga el ancho de 2 módulos.

## 1.3– Modos

Para transformar la información que se quiere transmitir en los bits que finalmente se colocan en el QR, se utilizan los modos. Cada modo es una serie de reglas que permiten transformar una serie de caracteres en una cadena de bits. Para saber qué modo se está utilizando, la cadena de bits debe ir precedida por una cabecera. La cabecera está compuesta por el indicador de modo y el indicador de caracteres. Cada modo tiene un indicador de modo distinto y puede consultarse en la tabla 1.2. El indicador de caracteres indica, en binario, cuantos caracteres se han codificado con ese modo. El número de bits que se reservan para el indicador de caracteres puede consultarse en la tabla 1.2.

| Versiones | Indicador de modo | | | | | Longitud del indicador de caracteres | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | M1 | M2 | M3 | M4 | 1 a 40 | M1 | M2 | M3 | M4 | 1 a 9 | 10 a 26 | 27 a 40 |
| Modo | | | | | | | | | | | | |
| Numeric | - | 0 | 00 | 000 | 0001 | 3 | 4 | 5 | 6 | 10 | 12 | 14 |
| Alphanumeric | n/a | 1 | 01 | 001 | 0010 | n/a | 3 | 4 | 5 | 9 | 11 | 13 |
| Byte | n/a | | 10 | 010 | 0100 | n/a | | 4 | 5 | 8 | 16 | 16 |
| Kanji | n/a | | 11 | 011 | 1000 | n/a | | 3 | 4 | 8 | 10 | 12 |
| ECI | | | | | 0111 | | | | | | | |
| Structured Append | n/a | | | | 0011 | n/a | | | | | 0 | |
| FNC1 (1ª posición) | | | | | 0101 | | | | | | | |
| FNC1 (2ª posición) | | | | | 1001 | | | | | | | |

Tabla 1.2: Indicadores de modo y longitud del indicador de caracteres según la versión

Es posible codificar una cadena de caracteres utilizando un cierto modo y luego cambiar a otro. Para ello se deben concatenar las codificaciones obtenidas usando cada modo en los caracteres correspondientes.

### 1.3.1. Numeric

Este modo codifica únicamente caracteres numéricos (desde **0** hasta **9**). Para codificar una cadena de caracteres:

1. se divide en grupos de 3 caracteres,
2. cada grupo se pasa a binario utilizando 10 dígitos (en caso de que el último grupo sea de 1 o 2 caracteres únicamente, solo se utilizan 4 o 7 dígitos respectivamente),
3. se concatenan todos los grupos de nuevo en el mismo orden y
4. se incluye la cabecera al principio de la cadena.

El número de bits necesarios para codificar una cadena de caracteres con este modo es

$$B = C + 10(DIV\ N\ 3) + R$$

siendo

$N$ el número de caracteres en la cadena

$B$ el número de bits en la cadena codificada

$C$ el número de bits de la cabecera

$R$ toma cada valor según el resto de dividir $N$ entre 3, siendo 0 en el caso de 0, 4 en el caso de 1 y 7 en el caso de 2.

### 1.3.2. Alphanumeric

Este modo codifica 45 caracteres, que comprenden caracteres numéricos (desde **0** hasta **9**), el alfabeto inglés en mayúsculas (desde **A** hasta **Z**) y otros 9 caracteres (**Espacio**, **$**, **%**, *, +, -, ., / y :). A cada carácter se le asocia un número entre el 0 y el 44 según el orden en el que se han descrito antes. Para codificar una cadena de caracteres

1. se divide en grupos de 2 caracteres,
2. cada grupo se cambia por el número en binario de 11 dígitos que resulta de multiplicar por 45 el número del primer carácter y sumar el número del segundo carácter (en caso de que el último grupo quede de 1 único caráter, se cambia por su número asociado en binario utilizando 6 dígitos),
3. se concatenan todos los grupos en el mismo orden en el que estaban sus caracteres asociados y
4. se incluye la cabecera al principio de la cadena.

El número de bits necesarios para codificar una cadena de caracteres con este modo es

$$B = C + 11(DIV\ N\ 2) + R$$

siendo

$N$ el número de caracteres de la cadena

$B$ el número de bits en la cadena codificada

$C$ el número de bits de la cabecera

$R$ toma el valor 0 si $N$ es par y 6 en caso contrario

Este modo no se puede utilizar en la versión M1.

### 1.3.3. Byte

Este modo codifica todos los caracteres de la codificación *latin-1*. Para codificar una cadena de caracteres

1. cada carácter se cambia por su equivalente binario de 8 dígitos correspondiente dentro de la codificación *latin-1*,

2. se concatenan todos los números en el mismo orden en el que estaban sus caracteres asociados y
3. se incluye la cabecera al principio de la cadena.

El número de bits necesarios para codificar una cadena de caracteres con este modo es

$$B = C + 8N$$

siendo

$N$ el número de caracteres de la cadena

$B$ el número de bits en la cadena codificada

$C$ el número de bits de la cabecera

Este modo no se puede utilizar en las versiones M1 y M2.

### 1.3.4. Kanji

Este modo codifica los caracteres kanji de la codificación *Shift JIS*. Para codificar una cadena de caracteres

1. cada carácter
   *a*) se cambia por su equivalente hexadecimal de 4 dígitos correspondiente dentro de la codificación *Shift JIS*,
   *b*) se le resta $8140_{\text{HEX}}$ ó $C140_{\text{HEX}}$, según en que rango esté:
      1) si está entre $8140_{\text{HEX}}$ y $9FFX_{\text{HEX}}$, se le resta $8140_{\text{HEX}}$ o
      2) si está entre $E040_{\text{HEX}}$ y $EBBF_{\text{HEX}}$, se le resta $C140_{\text{HEX}}$;
   *c*) los 2 primeros dígitos se multiplican por $C0_{\text{HEX}}$ y a dicho producto se suman los dos últimos dígitos
   *d*) se pasa a binario utilizando 13 dígitos,
2. se concatenan todos los números en el mismo orden en el que estaban sus caracteres asociados,
3. se incluye la cabecera al principio de la cadena.

El número de bits necesarios para codificar una cadena de caracteres con este modo es

$$B = C + 11N$$

siendo

$N$ el número de caracteres de la cadena

$B$ el número de bits en la cadena codificada

$C$ el número de bits de la cabecera

Este modo no se puede utilizar en las versiones M1 y M2.

### 1.3.5. ECI

Este modo no codifica caracteres, sino que los transforma en otros para que se puedan codificar con los modos que se han descrito anteriormente. Para ello, debe encontrarse una codificación donde se encuentren los caracteres que hay que codificar. Una vez encontrada, hay que cambiar cada carácter por su codificación en esta y, después, cambiar dicha codificación por su carácter asociado en *latin-1*. Una vez hecho esto, ya se podrán codificar con los modos que se han descrito anteriormente. El efecto de este modo abarca a todos los caracteres desde que aparece hasta que aparece un nuevo modo ECI o, si no aparece ninguno, hasta el final. Para volver a usar los modos anteriores como están descritos, solo hay que usar este mismo modo de nuevo, pero con la codificación *latin-1*.

En el QR aparecerá con su indicador de modo y de 8 a 24 bits, donde irá el valor de asignación de la codificación. El número de bytes (8 bits) usados para describir el valor de asignación vendrá dado por el número de unos antes del primer cero en el primer byte. Los demás bits se utilizan para almacenar el valor de asignación en binario.

Este modo no está disponible para los Micro QR y su implementación en las aplicaciones que crean y leen QR es opcional.

| Valor de asignación | Codificación | Valor de asignación | Codificación |
|---|---|---|---|
| 0 | Code Page 437 | 16 | Latin-8 (Celtic) |
| 1 | Latin-1 | 17 | Latin-9 |
| 2 | Code Page 437 | 18 | Latin-10 |
| 3 | Latin-1 | 20 | Shift JIS |
| 4 | Latin-2 | 21 | Windows-1250 |
| 5 | Latin-3 | 22 | Windows-1251 |
| 6 | Latin-4 | 23 | Windows-1252 |
| 7 | Latin/Cyrillic | 24 | Windows-1256 |
| 8 | Latin/Arabic | 25 | UTF-16 |
| 9 | Latin/Greek | 26 | UTF-8 |
| 10 | Latin/Hebrew | 27 | ASCII |
| 11 | Latin-5 | 28 | Big5 |
| 12 | Latin-6 | 29 | GB18030 |
| 13 | Latin/Thai | 30 | EUC-KR |
| 15 | Latin-7 | | |

Tabla 1.3: Valores de asignación de diferentes codificaciones

### 1.3.6. Structured Append

Este modo no codifica caracteres, sino que indica que el QR forma parte de una serie de sucesivos códigos QR, de forma que se deben leer todos para poder obtener el texto codificado. Solo puede aparecer una vez por QR y, en caso de aparecer, debe ser al principio, antes de los modos que codifican caracteres. El orden en el que se lean los diferentes códigos QR no importa para obtener el texto codificado de manera ordenada, pues en el propio modo se indica el orden y el número total de ellos.

Este modo no está disponible para los Micro QR y su implementación en las aplicaciones que crean y leen QR es opcional.

### 1.3.7. FNC1

Este modo no codifica caracteres, sino que indica que los datos codificados en el QR siguen algún estándar, ya sea el definido por GS1 Application Identifier en el caso de FNC1 en primera posición, u otro, en caso de FNC1 en segunda posición. Solo puede aparecer una vez por QR y, en dicho caso, debe ser el primer modo que aparezca.

Este modo no esta disponible para los Micro QR y su implementación en las aplicaciones que crean y leen QR es opcional.

### 1.3.8. Terminator

No es un modo en sí mismo, sino que se utiliza para señalizar el final de los datos. Consiste en 4 ceros para las versiones de la 1 a la 40 y en $2n + 1$ ceros para las versiones M$n$. Estos ceros se concatenan al final de los datos codificados ya como bits. Sin embargo, se debe omitir, si ya se ha alcanzado la capacidad máxima de bits de datos del QR, o acortar, si la capacidad restante es menor que la cantidad de ceros que se deben incluir.

## 1.4– Corrección de errores

Para detectar y corregir errores se utilizan códigos de Reed-Solomon sobre el cuerpo $\mathbb{F}_{256}$. Se puede seleccionar entre cuatro niveles de corrección de errores, aunque algunos no están disponibles para algunas versiones de Micro QR.

| Nivel | Capacidad de recuperación de datos (aprox.) | Versiones de Micro QR disponibles | Indicador |
|---|---|---|---|
| L | 7 % | Todas | 01 |
| M | 15 % | M2, M3 y M4 | 00 |
| Q | 25 % | M4 | 11 |
| H | 30 % | Ninguna | 10 |

Tabla 1.4: Niveles de corrección de errores

Un código de Reed-Solomon $(n, k, e)$ es un código que transmite $n$ elementos de un cierto cuerpo finito (en este caso $\mathbb{F}_{256}$) de los cuales $k$ son de datos que se quiere transmitir y $n - k$ son elementos correctores de errores. Este código puede detectar $n - k$ errores y corrige hasta $e$ errores. Teóricamente, el número de errores que se puede corregir es la mitad de los que se puede detectar; sin embargo, es habitual, cuando el número de errores que se pueden corregir es pequeño, corregir menos de la mitad. Esto evita que se corrija de manera errónea cuando la cantidad de errores sobrepasa el número de errores que se puede detectar.

Cuando en los siguientes apartados aparezca una serie de elementos de $\mathbb{F}_{256}$ como un polinomio (o viceversa), se debe entender que el primer elemento se corresponde con el

coeficiente del monomio de mayor grado y el último con el término independiente.

### 1.4.1. Codificación sobre $\mathbb{F}_{256}$

El mensaje codificado está formado por dos partes: la primera, que son los datos que se quieren transmitir, y la segunda, que esta compuesta por los elementos correctores de errores. Los elementos correctores de errores son los coeficientes del polinomio $p(x) \cdot x^{(n-k)} \bmod g(x)$, donde $p(x)$ es el polinomio que corresponde a los datos que se quieren transmitir y $g(x) = \prod_{r=0}^{n-k-1} (x - 2^r)$.

### 1.4.2. Decodificación sobre $\mathbb{F}_{256}$

Para decodificar los elementos recibidos, hay que verlos como un polinomio $r(x)$ de grado $n-1$. Después, hay que calcular los síndromes, que son

$$S_i = r(2^i) \quad \forall\, i = 0 \ \ldots \ (n-k-1)$$

Si todos los síndromes son 0, entonces el mensaje recibido no tiene errores. En caso contrario, se han producido errores y hay que corregirlos. Para ello, hay que averiguar en que elementos se han producido y el tamaño de cada error.

Para hallar las posiciones de los errores, hay que resolver el siguiente sistema de ecuaciones

$$\begin{pmatrix} S_0 & S_1 & \ldots & S_{v-1} \\ S_1 & S_2 & \ldots & S_v \\ \vdots & \vdots & \ddots & \vdots \\ S_{v-1} & S_v & \ldots & S_{v-2} \end{pmatrix} \begin{pmatrix} \Lambda_0 \\ \Lambda_1 \\ \vdots \\ \Lambda_{v-1} \end{pmatrix} = \begin{pmatrix} S_v \\ S_{v+1} \\ \vdots \\ S_{2v-1} \end{pmatrix}$$

donde $v$ es el número de errores que se han cometido. Aunque el valor de $v$ es desconocido, se puede calcular, pues es el mayor $v$ (empezando en $(n-k)$ DIV 2) tal que la matriz izquierda del sistema es invertible, lo que implica que el sistema tiene solución única. Si $v > e$, no se continua, pues se podría estar decodificando erróneamente. En caso contrario, las posiciones de los errores vendrán dadas por los $i$ tales que

$$\sum_{r=0}^{v-1} \Lambda_r (2^i)^r = 0$$

y el tamaño de cada error por la solución al sistema

$$\begin{pmatrix} (2^{i_0})^1 & (2^{i_1})^1 & \ldots & (2^{i_{v-1}})^1 \\ (2^{i_0})^2 & (2^{i_1})^2 & \ldots & (2^{i_{v-1}})^2 \\ \vdots & \vdots & \ddots & \vdots \\ (2^{i_0})^v & (2^{i_1})^v & \ldots & (2^{i_{v-1}})^v \end{pmatrix} \begin{pmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{v-1} \end{pmatrix} = \begin{pmatrix} S_0 \\ S_1 \\ \vdots \\ S_{v-1} \end{pmatrix}$$

Para corregir los errores, solo hay que sustraer el tamaño de cada error al elemento correspondiente en el que se haya producido un error. Por último, haya sido necesario o no corregir errores, hay que seleccionar los $k$ primeros elementos, cuyo contenido corresponde con el mensaje original que se trasmitió.

## 1.5– Máscaras

Para ayudar a que los módulos blancos y negros estén bien distribuidos a lo largo del QR y evitar patrones indeseados, que puedan dificultar la correcta decodificación, se utilizan las máscaras. Aplicar una máscara consiste en invertir una serie de módulos del QR, de forma que los módulos blancos se cambian por negros y viceversa. Hay 8 máscaras distintas para las versiones de la 1 a la 40, pero solo 4 de ellas están disponibles para los Micro QR.

En la tabla 1.5 se puede encontrar que módulos hay qué cambiar para cada máscara. Dicha condición solo se aplica a los módulos de la zona de datos y corrección de errores. Los módulos de los patrones requeridos y de las zonas de información de versión y de formato no cambian, independientemente de la máscara que se utilice.

| Referencia de máscara | | Condición para cambiar |
|---|---|---|
| QR | Micro QR | cada módulo $(i, j)$ |
| 000 | n/a | $(i + j)$ mód $2 = 0$ |
| 001 | 00 | $i$ mód $2 = 0$ |
| 010 | n/a | $j$ mód $3 = 0$ |
| 011 | n/a | $(i + j)$ mód $3 = 0$ |
| 100 | 01 | $((i \div 2) + (j \div 3))$ mód $2 = 0$ |
| 101 | n/a | $(i \cdot j)$ mód $2 + (i \cdot j)$ mód $3 = 0$ |
| 110 | 10 | $((i \cdot j)$ mód $2 + (i \cdot j)$ mód $3)$ mód $2 = 0$ |
| 111 | 11 | $((i + j)$ mód $2 + (i \cdot j)$ mód $3)$ mód $2 = 0$ |

Tabla 1.5: Máscaras disponibles y sus referencias

## 1.6– Puntuación

Los QR tienen una puntuación que nos indica lo fácil o dificil que va a ser detectarlos correctamente. Esto permite que se pueda escoger la máscara que cree un QR más fácil de detectar.

Para los Micro QR la puntuación viene dada por

$$\text{mín}\{\text{SUM}_F, \text{SUM}_C\} \cdot 16 + \text{máx}\{\text{SUM}_F, \text{SUM}_C\}$$

donde $\text{SUM}_F$ y $\text{SUM}_C$ son el número de módulos negros en la última fila y en la última columna respectivamente.

Para las demás versiones, la puntuación consiste en sumar las penalizaciones por encontrar los siguientes patrones en el QR:

- cada vez que se encuentren $5 + n$ módulos consecutivos del mismo color en una fila o columna se penaliza con $3 + n$ puntos;
- cada bloque de $m \times n$ módulos del mismo color se penaliza con $3 \times (m - 1) \times (n - 1)$;
- cada vez que se encuentra una sucesión de módulos en una fila o columna con los colores BBBBNBNNNBN ó NBNNNBNBBBB, donde N representa un módulo negro y B un módulo blanco, se penaliza con 40 puntos;

- cada 5 % que el porcentaje de módulos negros se desvíe del 50 % se penaliza con 10 puntos.

## 1.7– Procedimiento de creación

En esta sección se describe el algoritmo que hay que llevar a cabo para convertir una cadena de caracteres de entrada en un QR.

### 1.7.1. Análisis de los datos

Primero, hay que analizar los caracteres para ver qué modos se van a utilizar para codificar cada parte de la cadena. Hay que tener en cuenta que hay caracteres que se pueden codificar con varios modos distintos. La elección de cuál escoger vendrá dada por aquella que minimice el número de bits de la cadena final. Muchas veces, esto va a significar mantener el modo que se usa en el carácter anterior, pues cambiar de modo implica incluir la cabecera del nuevo modo.

Tras esto, hay que escoger la menor versión que sea capaz de almacenar los datos codificados. La capacidad en bits de cada versión puede encontrarse en la tabla 7 de [3].

### 1.7.2. Codificación de los datos

Una vez escogida la versión, hay que codificar la entrada en una cadena de bits según los modos que se han seleccionado en el paso anterior. No se puede olvidar: incluir al principio los modos que aportan información general del QR, incluir la cabecera de cada modo al principio y cada vez que se cambia de modo y terminar con el modo *Terminator*.

Después, hay que dividir la cadena en tramos de 8 bits y considerar cada tramo como un elemento de $\mathbb{F}_{256}$. En caso de que el último tramo esté compuesto por menos de 8 bits, se deben incluir los ceros que hagan falta hasta llegar a los 8 bits. Además, hay que añadir tantos elementos de relleno como sean necesarios para completar la capacidad de la versión. En la tabla 7 de [3] se encuentra una tabla con las capacidades en bytes (8 bits) de cada versión. Los elementos de relleno son 237 y 17. Estos deben añadirse de manera alternada al final de la cadena, hasta completar la capacidad.

En las versiones M1 y M3, el último tramo es de únicamente 4 bits. Si los datos no son suficientes para llenar la capacidad de estas versiones, el último elemento de relleno es 0.

### 1.7.3. Generación de los elementos correctores de errores

Para generar los elementos correctores de errores, primero hay que dividir la cadena de elementos de datos en distintos bloques, para posteriormente poder generar los elementos correctores de errores de cada bloque por separado. En cuántos bloques debe separarse y qué código de Reed-Solomon se utiliza en cada bloque puede encontrarse en la tabla 9 de [3].

### 1.7.4. Construcción de la secuencia de datos final

Para obtener la secuencia final hay que ir concatenando uno a uno los elementos de los distintos bloques (primero el primer elemento de cada bloque, después el segundo de cada bloque, etc.) y, a continuación, hacer lo mismo con los elementos correctores de errores de cada bloque. Una vez hecho esto, hay que volver a convertir cada elemento de $\mathbb{F}_{256}$ en un número binario de 8 bits. Esta será la secuencia final de bits que va incluida en el QR.

### 1.7.5. Colocación de los datos en la matriz

Primero hay que colocar en la matriz los patrones requeridos. Las zonas de información de versión e información de formato deben quedar en blanco, pero reservadas para ser completadas más tarde. En la zona de datos y corrección de errores debe ir la secuencia de datos final obtenida en el apartado anterior, colocada en columnas de 2 módulos de ancho de derecha a izquierda y alternando de abajo a arriba y de arriba a abajo, empezando por la esquina inferior derecha. La columna destinada al patrón de sincronización no cuenta a la hora de alternar la dirección vertical, sino que, cuando se encuentre, debe ignorarse. Cuando colocando la secuencia, se encuentre un módulo reservado o algún patrón requerido, debe saltarse dicho módulo y se colocará el bit correspondiente en el siguiente módulo disponible, siguiendo las reglas descritas anteriormente. Cada uno en la cadena de bits implica que el módulo correspondiente debe ser negro y cada cero, que debe ser blanco. Por último, se dejan en blanco todos los módulos que no se haya rellenado anteriormente. El número de módulos que no se rellenan con la cadena de bits pueden ser 3, 4 o 7, dependiendo de la versión, y se les llama bits restantes.

### 1.7.6. Selección de máscara

Al QR se le puede aplicar cualquiera de las máscaras disponibles. Sin embargo, se recomienda escoger aquella con menor penalización o, si es un Micro QR, con mayor puntuación. Esto hará que el QR se pueda leer con la mayor rapidez posible.

### 1.7.7. Información de formato y versión

Por último, solo falta incluir la información de formato y de versión en las zonas reservadas para ello.

#### Información de formato

La información de formato consiste en 5 bits que indican el nivel de corrección de errores y la máscara. Estos 5 bits preceden a otros 10 bits que son correctores de errores. Los bits de corrección de errores son los coeficientes del polinomio $p(x) \cdot x^{10} \mod g(x)$, donde $p(x)$ es el polinomio cuyos coeficientes son los bits a los que se les quiere calcular los bits de corrección de errores (con el primer bit siendo el termino de mayor grado y el último el término independiente) y $g(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$. Una vez hecho este cálculo, se les suma a los 15 bits una máscara de formato (suma en $\mathbb{F}_2$, es decir, la operación disyunción exclusiva) y se colocan en la zona reservada para la información de

formato, de forma que el módulo es negro si el bit correspondiente es un uno y es blanco, si el bit es un cero.

En el caso de los Micro QR, las 3 primeros bits dependen de la versión y el nivel de corrección de errores. Los bits que corresponden a cada uno se pueden encontrar en la tabla 1.6. Los 2 últimos son el indicador de la máscara que se haya utilizado. Su valor puede consultarse en la tabla 1.5. La máscara de formato utilizada es 100010001000101. Por último los bits se colocan en la zona reservada para ello, empezando por el módulo que se encuentra más a la izquierda, continuando con los módulos adyacentes a su derecha y terminando por los módulos superiores.

| Version y nivel de corrección de errores | Indicador |
|---|---|
| M1 L | 000 |
| M2 L | 001 |
| M2 M | 010 |
| M3 L | 011 |
| M3 M | 100 |
| M4 L | 101 |
| M4 M | 110 |
| M4 Q | 111 |

Tabla 1.6: Indicadores de formato para Micro QR

En las demás versiones, los 2 primeros bits son el indicador del nivel de corrección de errores, cuyo valor se puede encontrar en la tabla 1.4. Los 3 últimos bits son el indicador de la máscara que se haya utilizado, que se pueden encontrar en la tabla 1.5. La máscara de formato utilizada es 101010000010010. Por último, los bits se colocan por duplicado en la zona reservada para ello, poniendo una copia en los módulos que se encuentran en la novena columna, empezando por los módulos que se encuentran abajo y la otra copia en los módulos de la novena fila, empezando por los módulos que se encuentran a la izquierda. El módulo de la intersección de la fila y la columna mencionadas anteriormente se considera que pertenece a la columna, ya que en el lugar donde debería ir el octavo bit en los módulos que van en la columna, va siempre un módulo negro en su lugar.

### Información de versión

La información de versión solo está presente en las versión 7 y posteriores. Consiste en 6 bits en los que se incluye la versión. Estos 6 bits preceden a otros 12 bits, que son correctores de errores. Los 6 primeros bits corresponden con el número de la versión en binario y los bits de corrección de errores son los coeficientes del polinomio $p(x) \cdot x^{12}$ mód $g(x)$, donde $p(x)$ es el polinomio cuyos coeficientes son los bits a los que se les quiere calcular los bits de corrección de errores (con el primer bit siendo el termino de mayor grado y el último el término independiente) y $g(x) = x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^5 + x^2 + 1$.

Los bits se colocan por duplicado en la zona reservada para ello, poniendo una copia en los módulos cercanos al patrón de posición inferior izquierdo, de abajo a arriba y de derecha a izquierda, y la otra copia en los módulos cercanos al patrón de posición superior

derecho, de derecha a izquierda y de abajo a arriba.

## 1.8– Procedimiento de lectura

El algoritmo para obtener una matriz de módulos blancos y negros a través de una imagen es bastante complejo, por lo que se va a estudiar el caso en el que ya se tengan identificados los módulos y, por tanto, el QR venga representado por una matriz de módulos blancos y negros. La idea general es seguir el proceso inverso al que se sigue a la hora de crear el código QR.

### 1.8.1. Versión

Primero, hay que identificar la versión. Esta será aquella que coincida en dimensiones con la matriz.

### 1.8.2. Nivel de corrección de errores y máscara

Hay mirar los módulos que corresponden a la información de formato en el mismo orden en el que se colocan y obtener los bits correspondientes, de forma que cada módulo negro corresponde con un uno y cada módulo blanco con un cero. Después, se aplica la máscara de formato y se corrigen los errores que pudiese haber.

Para corregir los errores, se puede utilizar fuerza bruta. Sabiendo que se pueden corregir hasta 3 errores, basta con probar a codificar las 32 combinaciones de nivel corrección de errores y máscara y ver cuál difiere en menos de 3 bits con la cadena. Una vez hecho esto, se selecciona el nivel de corrección de errores y la máscara que corresponda con los indicadores que se han obtenido, según las tablas 1.4, 1.5 y 1.6.

### 1.8.3. Cadena de datos codificados

Antes que nada, se debe aplicar la máscara que se ha obtenido en el apartado anterior al QR. Después, hay que seleccionar los módulos en el mismo orden en el que se coloca la secuencia de bits en la sección 1.7.5 y transformarlos en una cadena de bits, de forma que cada módulo negro se convierte en un uno y cada módulo blanco en un cero.

### 1.8.4. Corrección de errores

Para corregir errores, hay que dividir la cadena en tramos de 8 bits y considerar cada tramo como un elemento de $\mathbb{F}_{256}$. Debe recordarse que, en la versiones M1 y M3, el último tramo correspondiente a los datos era únicamente de 4 bits. Por tanto, en dichas versiones, el tramo correspondiente debe ser de 4 bits. Posteriormente, se divide la cadena en bloques con el proceso inverso al utilizado en la sección 1.7.4. En cada bloque, por separado, se corrigen los errores y se obtiene la cadena original. Finalmente, se concatenan todas las cadenas así obtenidas y se transforman de nuevo en una cadena de bits, cambiando cada elemento por los 8 bits correspondientes, excepto para el último elemento de las versiones M1 y M3, que se transforma únicamente en 4 bits.

### 1.8.5. Decodificación de los datos

El último paso consiste en repetir el siguiente proceso. Si no quedan bits en la cadena, todos los bits que quedan son ceros o si los primeros bits corresponden con el modo `Terminator`, entonces ya se ha recibido la cadena de caracteres completa y, por tanto, se termina. En caso contrario, se mira en la tabla 1.2 con qué modo se corresponden los primeros bits. Se cogen tantos bits como se indica en esa misma tabla para ver el indicador de caracteres. Este indica cuantos caracteres se han codificado con ese modo. Se cogen tantos bits como correspondan según el número de caracteres codificados y aplicamos el proceso contrario de codificación según el modo que se haya detectado. Una vez repetido este proceso hasta que se haya terminado, se habrá obtenido la cadena que codifica el QR.

## 1.9– Variantes

Para cubrir otras necesidades que no son capaces de abarcar los códigos QR, se han creado diversas variantes de los códigos QR.

### 1.9.1. Model 1

Un modelo más antiguo y en desuso, aunque muy parecido al actual, pero con menos versiones y algunos cambios en los patrones requeridos (lo que imposibilita que se pueda leer con un lector de códigos QR ordinario).

Figura 1.2: QR Model 1
*Autor: Bobmath*

### 1.9.2. iQR

Una alternativa que permite ahorrar espacio, ya que es capaz de almacenar más información. Además, se pueden crear con forma rectangular, además de la forma cuadrada, y se puede utilizar un nuevo nivel de corrección de errores, que es capaz de recuperar hasta el 50 % de la información. Sin embargo, su licencia es privada, por lo que solo se puede crear y leer con aplicaciones de Denso Wave.

### 1.9.3. Secure QR

Muy parecido a un código QR usual, pero con la diferencia de que incluye una parte de datos públicos y una parte de datos privados. La parte de datos públicos se puede leer

Figura 1.3: iQR
*Autor: Tom Knox*

con un lector de códigos QR usual; sin embargo, la parte de datos privados necesita un lector específico y una clave de desencriptado.

Figura 1.4: Secure QR
*Autor: Bang Bang50*

### 1.9.4. Frame QR

Una variante que incluye un área, que puede ser de distintas formas, para incluir imágenes o logos. No se debe confundir con códigos QR convencionales a los que se les ha incluido un logo sustituyendo algunos de los módulos, pues en dicho caso un lector de códigos QR convencional es suficiente, mientras que un Frame QR necesita un lector propio, pues tienen características distintas.

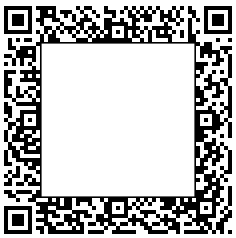

Figura 1.5: Frame QR
*Autor: Tom Knox*

### 1.9.5. HCC2D

Esta variante se diferencia en que los módulos pueden tomar más de dos colores. Esto permite que cada módulo pueda contener más de 1 bit (2 bits y 4 bits por módulo es lo

usual), con lo que se incrementa considerablemente la cantidad de información que puede contener.

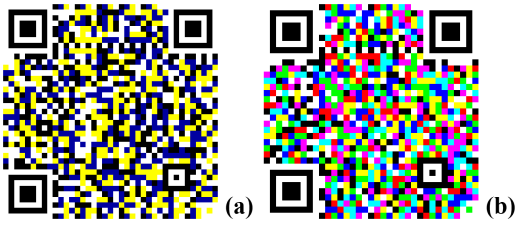


Figura 1.6: HCC2D. a) 4 colores, b) 8 colores
*Autor: Marco Querini*

# CAPÍTULO 2

## Diseño de la aplicación

Antes de empezar con los tipos y las funciones propiamente de los códigos QR, vamos a presentar una serie de tipos, instancias y funciones auxiliares que nos van a ayudar posteriormente.

### 2.0.1. `Bool`

Primero vamos a definir instancias del tipo `Bool` de las clases `Num` y `Fractional`. Gracias a esto, vamos a poder trabajar con este tipo como si fuera un tipo numérico sobre el cuerpo $\mathbb{F}_2$, de forma que `True` es uno y `False` es cero.

Código 2.1: Nuevas instancias de `Bool`

```
instance Num Bool where
   (+) = xor
   (-) = xor
   (*) = (&&)
   fromInteger x = mod x 2 == 1
   abs = id
   signum = id

instance Fractional Bool where
   recip x = if x then x else error "divide by zero"
   fromRational q = on (/) fromInteger (numerator q) (denominator q)
```

### 2.0.2. `BitString`

Para manejar las cadenas de bits, vamos a utilizar el tipo `BitString`, que consiste en una lista de booleanos. Esta lista la vamos a entender como una lista de ceros y unos.

Código 2.2: Tipo `BitString`

```
type BitString = [Bool]
```

Las funciones `integralToBitString` e `integralsToBitString` nos permiten transformar números (uno solo en el primer caso y varios en el segundo) con una instancia de la clase `Integral` a una `BitString`, como si pasásemos números en base decimal a su forma en binario. El primer argumento indica cuántos bits vamos a utilizar para hacer la cadena de bits. Esto es necesario pues, por ejemplo, el número 2 puede convertirse tanto en 10, como en 0010, dependiendo del número de bits que utilicemos.

Código 2.3: Funciones `integralsToBitString` e `integralToBitString`

```
integralsToBitString :: (Integral a) => [Int] -> [a] -> BitString
integralsToBitString ns = concat . zipWith (integralToBitString) ns

integralToBitString :: (Integral a) => Int -> a  -> BitString
integralToBitString n x = go n x []
   where go 0 0 xs = xs
         go 0 _ xs = error "integralToBitString needs more bits"
         go n x xs = go (n-1) d $ (m==1):xs
            where (d,m) = divMod x 2
```

Las funciones `bitStringToNum` y `bitStringToNums` son las funciones contrarias. Estas nos permiten trasformar una `BitString` en números (uno solo en el primer caso y varios en el segundo) con una instancia de la clase `Num`, como si pasásemos números de su forma en binario a base decimal. El primer argumento de `bitStringToNums` indica cuántos elementos de la cadena de bits van a transformarse en cada número.

Código 2.4: Funciones `bitStringToNums` y `bitStringToNums`

```
bitStringToNums :: (Num a) => [Int] -> BitString -> [a]
bitStringToNums ns xs = go ns xs $ length xs
   where go (n:ns) xs m
           | m == 0    = []
           | m <  n    = [bitStringToNum $ xs ++ take (n-m)(repeat False)]
           | otherwise = let (a,b) = splitAt n xs
                         in (bitStringToNum a):(go ns b (m-n))

bitStringToNum :: (Num a) => BitString -> a
bitStringToNum = foldl (a b ->  2*a+ if b then 1 else 0) 0
```

Por último, vamos a definir la función `count`, que cuenta cuántos valores `True` (o cuántos unos) hay en nuestra `BitString`. Aunque hayamos definido la instancia de la clase `Num`, no podemos utilizar la función `sum`, ya que solo devuelve uno o cero en lugar del número de unos que hay en la lista.

Código 2.5: Función `count`

```
count :: Num a => BitString -> a
count = foldl (x y -> x + if y then 1 else 0) 0
```

### 2.0.3. GF256

Para los elementos de $\mathbb{F}_{256}$ vamos a utilizar el tipo `GF256`, que consiste en el constructor `GF256` con un único argumento de tipo `Word8`. El motivo de hacerlo de esta forma, y no como un tipo sinónimo a `Word8`, es para poder definir nuestras propias instancias para la suma, multiplicación, etcétera, y no confundirlas con las de `Word8` en ningún momento.

Código 2.6: Tipo `GF256`

```
newtype GF256 = GF256 {unpack :: Word8}
   deriving (Eq, Ord)
```

Primero vamos a definir todas las operaciones que necesitamos usando solo el tipo `Word8`. La exponencial y el logaritmo las hemos implementado con una matriz de búsqueda del tipo `Array`.

Código 2.7: Operaciones sobre `Word8`

```
gfSum :: Word8 -> Word8 -> Word8
gfSum = xor

gfMul :: Word8 -> Word8 -> Word8
gfMul = go 0
   where go p 0 _ = p
         go p _ 0 = p
         go p a b = go p' a' b'
            where p' | testBit b 0 = gfSum p a
                     | otherwise   = p
                  a' | testBit a 7 = gfSum 0x1d (shiftL a 1)
                     | otherwise   = shiftL a 1
                  b' = shiftR b 1

gfExpArray :: Array Int Word8
gfLogArray :: Array Word8 Int

gfExp :: Int -> Word8
gfExp n = gfExpArray ! (mod n 255)

gfLog :: Word8 -> Int
gfLog x = gfLogArray ! x
```

Con las funciones definidas anteriormente, podemos escribir todas las instancias que necesitamos que tenga el tipo `GF256`.

Código 2.8: Instancias de `GF256`

```
instance Num GF256 where
   (+) x y = GF256 $ on gfSum unpack x y
   (*) x y = GF256 $ on gfMul unpack x y
   (-) = (+)
   fromInteger = GF256 . fromInteger
   abs = id
   signum = GF256 . signum . unpack

```

```haskell
instance Fractional GF256 where
   recip = GF256 . gfExp . (255-) . gfLog . unpack
   fromRational q = on (/) (GF256 . fromInteger) (numerator q) (denominator q)

instance Real GF256 where
   toRational = toRational . unpack
   maxBound = GF256 maxBound

instance Enum GF256 where
   toEnum = GF256 . toEnum
   fromEnum = fromEnum . unpack

instance Integral GF256 where
   toInteger = toInteger . unpack
   quotRem x y = (GF256 q, GF256 r)
      where (q,r) = on quotRem unpack x y
   divMod = quotRem
```

Lo último que nos queda por definir son las funciones `log` y `exp`. Sin embargo, estas funciones están diseñadas para ser usadas con tipos numéricos de coma flotante, es decir, instancias de la clase `Floating`, y no de forma discreta como queremos ahora. Por tanto, tenemos que crear otras funciones fuera de la clase anterior para poder obtener el logaritmo discreto y la exponencial.

Código 2.9: Funciones `discLog` y `discExp`

```haskell
discExp :: Int -> GF256
discExp = GF256 . gfExp

discLog :: GF256 -> Int
discLog = gfLog . unpack
```

### 2.0.4. `Poly`

Para los polinomios vamos a utilizar el tipo `Poly`, que es un tipo algebraico en el que en cada nodo se corresponde con un monomio con coeficiente no nulo y la hoja final con el termino independiente del polinomio, que si puede ser cero. Los nodos deben estar ordenados en orden decreciente de grado.

Código 2.10: Tipo Poly

```haskell
data Poly a = P a Int (Poly a) | C a
```

Vamos a definir una instancia del tipo `Poly` de las clases `Num` y `Functor` para poder trabajar con los polinomios de manera más cómoda. Sin embargo, hay que tener cuidado con la función `fmap`, pues solo debemos usarla con funciones que dejen al cero como punto fijo y no manden ningún otro elemento al cero. Si no, puede funcionar de manera incorrecta. Un ejemplo de función con la que sí podemos usar `fmap` es `negate`.

Código 2.11: Instancias de Poly

```haskell
instance Functor Poly where
   fmap f (C a) = C . f $ a
   fmap f (P a n q) = P (f a) n $ fmap f q

instance (Eq a, Num a) => Num (Poly a) where
   (+) p@(P a n r) q@(P b m s) | n==m = (if (a+b)==0 then id
                                         else P (a+b) n) r+s
                               | n<m  = P b m (p+s)
                               | n>m  = P a n (r+q)
   (+) (P a n p)   q                  = P a n (p+q)
   (+) p           (P a n q)          = P a n (p+q)
   (+) (C a)       (C b)              = C $ a+b

   (*) p (P b m s) = (go p b m) + p*s
      where go (P a n r) b m = P (a*b) (n+m) (go r b m)
            go (C a) b m | a == 0    = C 0
                         | otherwise = P (a*b) m $ C 0
   (*) p (C 0) = C 0
   (*) (P a n r) q@(C b) = P (a*b) n $ r*q
   (*) (C a) (C b) = C $ a*b
   negate =  fmap negate
   abs = id
   signum _ = 1
   fromInteger = C . fromInteger
```

Para la división de polinomios tendríamos que crear la instancia de la clase `Integral`. Sin embargo, para ello, hace falta que la clase de los polinomios tenga también definida una instancia de la clase `Real`, que nos pide una función que a cada polinomio le asocie un número racional. Como esta función tiene poco sentido, vamos a hacer una funciones independientes de esta clase para la división de polinomios.

Código 2.12: Función `polyDivMod`

```haskell
polyDivMod :: (Eq a, Fractional a) => Poly a -> Poly a -> (Poly a, Poly a)
polyDivMod p (C b) = (p*(C . recip $ b),C 0)
polyDivMod (C a) q = (0, C a)
polyDivMod p@(P a n r) q@(P b m s) | n>m  = (cocienteParcial + x,y)
                                  | n==m = (cocienteIndepe, restoIndepe)
                                  | n<m  = (0, p)
      where cocienteParcial = P (a/b) (n-m) 0
            restoParcial = p-cocienteParcial*q
            (x,y) = polyDivMod restoParcial q
            cocienteIndepe = C $ a/b
            restoIndepe = p-cocienteIndepe*q

polyDiv :: (Eq a, Fractional a) => Poly a -> Poly a -> Poly a
polyDiv p q = c where (c,_) = polyDivMod p q

polyMod :: (Eq a, Fractional a) => Poly a -> Poly a -> Poly a
polyMod p q = r where (_,r) = polyDivMod p q
```

Para construir polinomios de manera más sencilla, sin tener que recurrir a sus constructores, vamos a definir las siguientes funciones:

- `x` que es el polinomio $x$.
- `constPoly` para crear polinomios constantes.
- `monomial` para crear monomios dados el coeficiente y el grado.
- `makePoly` para crear un polinomio desde una lista de coeficientes.
- `fromLists` para crear un polinomio desde dos listas, una con los coeficientes en cualquier orden y otra con el grado del monomio al que corresponden esos coeficientes.

Código 2.13: Funciones para crear polinomios

```
1  x :: Num a => Poly a
2  x = P 1 1 $ C 0
3
4  constPoly :: Num a => a -> Poly a
5  constPoly = C
6
7  monomial :: Num a => a -> Int -> Poly a
8  monomial a n = P a n $ C 0
9
10 makePoly :: (Eq a, Num a) => [a] -> Poly a
11 makePoly xs = go xs $ length xs
12    where go _      0 = C 0
13          go (x:[]) 1 = C x
14          go (0:xs) n = go xs (n-1)
15          go (x:xs) n = P x (n-1) $ go xs (n-1)
16
17 fromLists :: (Eq a, Num a) => [a] -> [Int] -> Poly a
18 fromLists xs = sum . zipWith f xs
19    where f x n = if n==0 then C x else P x n 0
```

Por último, vamos a definir las siguientes funciones auxiliares:

- `grade` indica el grado del polinomio.
- `coefs` devuelve una lista con los coeficientes del polinomio en orden decreciente de grado.
- `coefsn` toma un argumento más que la función anterior y devuelve lo mismo, pero como si el polinomio fuera del grado del argumento adicional, es decir, con tantos ceros delante como la diferencia entre su grado y el argumento adicional.
- `eval` evalúa un polinomio en un número.
- `isZero` comprueba si el número es raíz del polinomio.
- `searchZerosBy` busca los ceros de un polinomio siguiendo un orden que definamos.

Código 2.14: Otras funciones de polinomios

```
grade :: Poly a -> Int
grade (C _)     = 0
grade (P _ n _) = n

coefs :: Num a => Poly a -> [a]
coefs p = go p $ grade p
   where go (P a n q) m = replicate (m-n) 0 ++ a:(go q (n-1))
         go (C a) m = (replicate m 0) ++ [a]

coefsn :: (Eq a, Num a) => Int -> Poly a -> [a]
coefsn n p = replicate (n - grade p - 1) 0 ++ coefs p

eval :: Num a => Poly a -> a -> a
eval p x = foldl (a c -> x*a+c) 0 $ coefs p

isZero :: (Eq a,Num a) => Poly a -> a -> Bool
isZero p = (0==) . (eval p)

searchZerosBy :: (Eq a, Num a) => (a -> a) -> a -> Poly a -> [a]
searchZerosBy s i p = go (grade p) $ i:(nexts $ s i)
   where go 0 _  = []
         go _ [] = []
         go n (x:xs) = if isZero p x then x:(go (n-1) xs) else go n xs
         nexts y = if y==i then [] else y:(nexts $ s y)
```

### 2.0.5. `QR`

Aunque antes de crear el tipo que represente a los QR propiamente, vamos a crear el tipo adicional `QRArray`. Este tipo consta únicamente de un constructor con un único argumento de tipo `Array`. La ventaja de crear este tipo y no utilizar directamente el tipo `Array`, es que podemos declarar nuestras propias instancias de ese tipo sin que nos afecten las declaradas para el tipo `Array`, cosa que vamos a necesitar para poder mostrar el QR por pantalla al crearse. Además, creando una instancia de la clase `IArray`, podemos trabajar con nuestra clase de la misma forma que si fuese una `Array` ordinaria.

Código 2.15: Tipo `QRArray` e instancia de la clase `IArray`

```
newtype QRArray i e = QR {getArray :: Array i e}

instance IArray QRArray e where
   bounds = bounds . getArray
   numElements = numElements . getArray
   unsafeArray i = QR . unsafeArray i
   unsafeAt = unsafeAt . getArray
```

La posición de cada módulo la vamos a representar con un par de números enteros. Para recalcar su significado, lo vamos a renombrar con el tipo `Module`.

Código 2.16: Tipo `Module`

```
type Module = (Int,Int)
```

Finalmente, para los QR vamos a utilizar el tipo `QRArray Module Bool`, que pasamos a renombrar `QR`. El valor `True` va a representar un módulo negro o un uno en la cadena de bits y, el valor `False`, un módulo blanco o un cero.

Código 2.17: Tipo `QR`

```
type QR = QRArray Module Bool
```

## 2.1– Versiones

Para las versiones vamos a utilizar el tipo `Version`. Este tipo tiene dos constructores, uno para los Micro QR y otro para las demás versiones. También vamos a definir la función `isMicro`, que nos dice si la versión corresponde a un Micro QR o no, y `micros`, que nos enumera todas las versiones de Micro QR.

Código 2.18: Tipo `Version` y funciones `isMicro` y `micros`

```
data Version = V {number :: Int} |  MV {number :: Int}
   deriving (Eq)

isMicro :: Version -> Bool
isMicro MV{} = True
isMicro  _     = False

micros :: [Version]
micros = range (MV 1, MV 4)
```

Muchos de los datos necesarios para crear un código QR dependen de la versión. Por ello, queremos guardarlos en tablas de búsqueda (del tipo `Array`) donde la versión funcione como índice. Para poder hacerlo, vamos a declarar una instancia del tipo `Version` de la clase `Ix`.

Código 2.19: Instancias de `Version`

```
instance Ord Version where
   compare (V  _) (MV _) = GT
   compare (MV _) (V  _) = LT
   compare v w = compare (number v) (number w)

instance Ix Version where
   range (V  n, V  m) = [V  a | a<-[n..m]]
   range (MV n, MV m) = [MV a | a<-[n..m]]
   range (MV n, V  m) = [MV a | a<-[n..4]] ++
                        [V  a | a<-[1..m]]
   range (_   , _   ) = []
   index p c = go 0 $ range p
      where go n [] = error "Index out of range"
```

```
14              go n (x:xs) | c==x = n
15                          | otherwise = go (n+1) xs
16      inRange (a,b) v = a <= v && v<= b
```

Dejando a un lado que ciertos datos cambian según la version, hay diferencias significativas en el algoritmo para crear los Micro QR y las demás. Para resolver este problema, vamos a definir la función `versionCase`. Su primer argumento es el algoritmo a seguir para los Micro QR y el segundo, el algoritmo para las demás versiones. El resultado es aplicar el algoritmo correspondiente según la versión. Esta función tiene un comportamiento y una finalidad muy parecidos a las funciones `maybe` o `either`.

Código 2.20: Función `versionCase`

```
1 versionCase :: (Version -> a) -> (Version -> a) -> Version -> a
2 versionCase f g v | isMicro v = f v
3                  | otherwise = g v
```

Derivadas de esta función, vamos a definir otras para cuando nuestros algoritmos no dependan del tipo `Versión`, sino de otro derivado.

- `numberVersionCase` solo pasa a los algoritmos el número de la versión. Esta será la más habitual, pues una vez separadas las versiones Micro QR de las demás, lo único importante es el número de la versión.
- `kindVersionCase` no pasa ningún argumento. Esta será útil cuando el algoritmo sea distinto entre Micro QR y las demás versiones, pero este no dependa del número de la versión.

Código 2.21: Funciones `numberVersionCase` y `kindVersionCase`

```
1 numberVersionCase :: (Int -> a) -> (Int -> a) -> Version -> a
2 numberVersionCase f g = versionCase (f . number) (g . number)
3
4 kindVersionCase :: a -> a -> Version -> a
5 kindVersionCase f g = versionCase (const f) (const g)
```

La función `size` indica el tamaño de la matriz según la versión. Hay que tener en cuenta que el número que devuelve esta función es uno menos que lo que indica la tabla 1.1. Esto se debe a que la matriz que define al QR comienza por la fila y la columna cero y, por tanto, la última fila y columna, que es lo que querremos saber normalmente cuando utilicemos esta función, será uno menos que lo que se indica en la tabla 1.1. También vamos a definir la función `sizeVersionCase` para cuando el algoritmo que vayamos a utilizar solo dependa del tamaño de la matriz.

Código 2.22: Funciones `size` y `sizeVersionCase`

```
1 sizeVersionCase ::  (Int -> a) -> (Int -> a) -> Version -> a
2 sizeVersionCase f g = numberVersionCase (f . sizeMV) (g . sizeV)
3
4 size :: Version -> Int
5 size = numberVersionCase sizeMV sizeV
```

```
6
7  sizeV :: Int -> Int
8  sizeV x = 16+4*x
9
10 sizeMV :: Int -> Int
11 sizeMV x = 8+2*x
```

De forma inversa, vamos a definir la función `unSize`, que devuelve la versión que corresponda a cada tamaño de matriz, o `Nothing` si dicho tamaño no coincide con ninguna versión. Cuando estemos seguros de que el tamaño que vayamos a pasarle a la función se corresponde con el tamaño asociado a una versión, podremos utilizar `unsafeUnSize`, que tiene el mismo comportamiento que la función anterior, pero devuelve el tipo `Version` en lugar de `Maybe Version`.

Código 2.23: Funciones `unSize` y `unsafeUnSize`

```
1  unSize :: Int -> Maybe Version
2  unSize x | x<=16     = if and [mMV==0,1<=dMV,4 >=dMV]
3                        then Just $ MV dMV else Nothing
4           | otherwise = if and [mV ==0,1<=dV ,40>=dV ]
5                        then Just $ V  dV  else Nothing
6     where (dMV,mMV) = divMod (x-8) 2
7           (dV,mV) = divMod (x-16) 4
8
9  unsafeUnSize :: Int -> Version
10 unsafeUnSize x = if x<=16 then MV $ div (x-8) 2 else V $ div (x-16) 4
```

## 2.2– Estructura

Como los módulos que vamos a ver ahora no son un QR completo, sino solo una parte, no los vamos a tratar con el tipo `QR`. En su lugar, los vamos a describir como listas de pares de módulos y booleanos, con el objetivo de incluirlos en el QR cuando sea necesario. Cada par representa un módulo y el valor que debe tomar.

### 2.2.1. Patrones requeridos

Primero, vamos a definir la función auxiliar `placeModules`, cuyo objetivo es definir los patrones requeridos de una forma más sencilla. Esta función toma como argumentos: una función, que a cada módulo le asocia un booleano; dos enteros, que corresponden con el ancho y el alto; y otros dos enteros, que corresponden a la posición del módulo superior izquierdo. La función devuelve una lista de pares de módulos y booleanos que corresponden a los módulos del patrón requerido en cuestión.

Código 2.24: Función `placeModules`

```
1  placeModules :: (Int -> Int -> Bool) -> Int -> Int -> Int -> Int ->
2                  [(Module,Bool)]
3  placeModules f n m i j = [((a+i,b+j),f a b) | a<-[0..n-1], b<-[0..m-1]]
```

Gracias a esta función, podemos definir de manera sencilla todos los patrones requeridos.

- `finderPattern` define los patrones de posición.
- `hWhite` y `vWhite` definen las zonas que separan los patrones de posición del resto del QR.
- `alignmentPattern` define los patrones de alineamiento.
- `blackAndWhite` define los patrones de sincronización.

Evidentemente estas funciones dependen de unos argumentos, pues la posición de los patrones requeridos depende de la versión y en el caso de los patrones de sincronización, su tamaño también depende de la versión.

Código 2.25: Funciones de los patrones requeridos

```
finderPattern :: Int -> Int -> [(Module,Bool)]
finderPattern = placeModules f 7 7
   where f x y = or [x==0, x==6, y==0, y==6, x>=2 && x<=4 && y>=2 && y<=4]

alignmentPatter :: Int -> Int -> [(Module,Bool)]
alignmentPatter = placeModules f 5 5
   where f x y = or [x==0, x==4, y==0, y==4, x==2 && y==2]

vWhite, hWhite :: Int -> Int -> [(Module,Bool)]
vWhite = placeModules (const $ const $ False) 7 1
hWhite = placeModules (const $ const $ False) 1 8

blackAndWhite :: Int -> Int -> Int -> Int -> [(Module,Bool)]
blackAndWhite = placeModules $ x y -> mod (x+y) 2 == 0
```

La función que indica los centros de los patrones de posición es `alignmentPatternLocationV`. El algoritmo que hemos seguido para determinarlos se puede encontrar en [14].

Código 2.26: Función `alignmentPatternLocationV`

```
alignmentPatternLocationV :: Int -> [(Module)]
alignmentPatternLocationV 1 = []
alignmentPatternLocationV n = go $ list lastPos
   where npatterns = div n 7
         npatterns1 = npatterns + 1
         fstPos = 6
         lastPos = sizeV n - 6
         sndLastPos = roundEven $ div (fstPos +
                      lastPos*npatterns + div npatterns1 2)
                      npatterns1
         step = lastPos-sndLastPos
         roundEven x | even x = x
                     | otherwise = x-1
         list x | x<=8 = [6]
                | otherwise = x:(list $ x-step)
         go xs = let l = last xs
                     f = head xs
                 in (foldr (x ys -> zip (repeat x) xs ++ ys)
                     [] xs) \\ [(l,l),(l,f),(f,l)]
```

### 2.2.2. Zona de codificación

Las funciones `versionLocation` y `formatLocation` indican los módulos que corresponden a la información de versión y a la información de formato respectivamente según la versión.

Código 2.27: Funciones `versionLocation` y `formatLocation`

```
versionLocation :: Version -> [(Module)]
versionLocation = numberVersionCase (const []) versionLocationV

versionLocationV :: Int -> [(Module)]
versionLocationV n | n<7 = []
                   | otherwise =  [(i,j) | j<-[5,4..0], i<-map (s-) [8,9,10]]
                                ++ [(i,j) | i<-[5,4..0], j<-map (s-) [8,9,10]]
   where s = sizeV n

formatLocation :: Version -> [(Module)]
formatLocation = numberVersionCase f g
   where f _ = [(8,j) | j<-[1..8]] ++ [(i,8) | i<-[7,6..1]]
         g v = let s = sizeV v
               in [(i,8) | i<-(map (s-) [0..6])++[8,7,5,4,3,2,1,0]]
               ++ [(8,j) | j<-[0,1,2,3,4,5,7]++ (map (s-) [7,6..0])]
```

La zona de datos y corrección de errores viene dada por aquellos módulos que no forman parte ni de los patrones requeridos ni de los módulos de la información de versión y la información de formato. A la conjunción de todos los módulos anteriores que no forman parte de la zona de datos y corrección de errores se les suele llamar también módulos reservados. En la tabla de búsqueda `reservedMod` podemos encontrar qué módulos son reservados y cuáles son de la zona de datos y corrección de errores.

Código 2.28: Tabla de busqueda `reservedMod`

```
reservedMod :: Version -> Array Module Bool
reservedMod v = array b [(p, f p) | p<-range b]
   where b = ((0,0),(size v,size v))
         f = numberVersionCase reservedModMV reservedModV v

reservedModV :: Int -> Module -> Bool
reservedModV n (x,y) = or $ (gur x y):(gdl x y):(x==6):(y==6):(x<=8 && y<=8):
                        (x>=(s-7) && y<=8):(x<=8 && y>=(s-7)):
                        [x>=i-2 && x<=i+2 && y>=j-2 && y<=j+2
                        | (i,j) <- alignmentPatternLocationV n]
   where s = sizeV n
         gur x y | n<7 = False
                 | otherwise = x>=(s-10) && y<=5
         gdl x y | n<7 = False
                 | otherwise = x<=6 && y>=(s-10)

reservedModMV :: Int -> Module -> Bool
reservedModMV _ (x,y) = or [x==0, y==0, x<=8 && y<=8]
```

## 2.3– Modos

Los modos que se utilizan para codificar caracteres los vamos a manejar con el tipo `DataSet`, que es una enumeración donde cada constructor se corresponde con un modo.

Código 2.29: Tipo `DataSet`

```
1 data DataSet = Numeric | Alphanumeric | Byte
2    deriving (Eq,Ord,Enum,Show)
```

Para cada uno de estos modos hemos definido las siguientes funciones:

- `is` indica si un carácter se puede codificar.
- `minVersion` es la menor versión con la que se puede utilizar cada modo.
- `modeIndicator` es el indicador del modo.
- `characterCountLenght` es la longitud en bits del indicador de caracteres.
- `toBitString` codifica una cadena de caracteres y devuelve la cadena de bits correspondiente.
- `fromBitString` decodifica una cadena de bits y devuelve los caracteres correspondientes.
- `charCost` indica cuántos bits se necesitan para codificar un carácter. Hay que tener en cuenta que el coste no es constante, sino que depende de cuántos caracteres se hayan codificado anteriormente.

Estas funciones están pensadas para que se importen de manera cualificada, de forma que el nombre de cada función hay que precederlo con el nombre del modo que se esté utilizando.

Supuesto que tenemos la funciones anteriores definidas e importadas de forma cualificada, vamos a definir una serie de funciones que vamos a necesitar luego. Primero, de muchas de las funciones que hemos mencionado antes para cada modo, vamos a crear una versión que tome un argumento más. El objetivo es poder aplicar a los demás argumentos la función del modo que se indica en el argumento adicional.

Código 2.30: Funciones genéricas asociadas a los modos

```
1  charCost :: DataSet -> Int -> Int
2  charCost x = case x of
3     Numeric -> Numeric.charCost
4     Alphanumeric -> Alphanumeric.charCost
5     Byte -> Byte.charCost
6
7  modeMinVersion :: DataSet -> Version
8  modeMinVersion x = case x of
9    Numeric -> Numeric.minVersion
10   Alphanumeric -> Alphanumeric.minVersion
11   Byte -> Byte.minVersion
12
```

```
13 modeIndicator :: DataSet -> Version -> BitString
14 modeIndicator Numeric = Numeric.modeIndicator
15 modeIndicator Alphanumeric = Alphanumeric.modeIndicator
16 modeIndicator Byte = Byte.modeIndicator
17
18 characterCountLength :: DataSet -> Version -> Int
19 characterCountLength Numeric      = Numeric.characterCountLength
20 characterCountLength Alphanumeric = Alphanumeric.characterCountLength
21 characterCountLength Byte         = Byte.characterCountLength
22
23 toBitString :: DataSet -> String -> BitString
24 toBitString Numeric = Numeric.toBitString
25 toBitString Alphanumeric = Alphanumeric.toBitString
26 toBitString Byte = Byte.toBitString
27
28 fromBitString :: DataSet -> BitString -> Maybe String
29 fromBitString Numeric = Numeric.fromBitString
30 fromBitString Alphanumeric = Alphanumeric.fromBitString
31 fromBitString Byte = Byte.fromBitString
```

Para ver con qué modo podemos codificar un carácter tenemos las funciones `exclusiveSet` y `posibleSet`. La primera indica el modo óptimo según el número de bits que se necesitan para codificar cada carácter, mientras que la segunda indica todos los modos con los que se puede codificar. Esta segunda también tiene en cuenta la versión, pues no todos los modos están disponibles en todas las versiones.

Código 2.31: Funciones `exclusiveSet` y `posibleSet`

```
1  exclusiveSet :: Char -> DataSet
2  exclusiveSet c | Numeric.is c = Numeric
3                 | Alphanumeric.is c = Alphanumeric
4                 | Byte.is c = Byte
5                 | otherwise = error $ "Non valid character: "++show c
6
7  posibleSet :: DataSet -> Version -> [DataSet]
8  posibleSet d = numberVersionCase
9                 (posibleSetMV d)
10                (const $ posibleSetV d)
11
12 posibleSetV :: DataSet -> [DataSet]
13 posibleSetV d = [d .. Byte]
14
15 posibleSetMV :: DataSet -> Int -> [DataSet]
16 posibleSetMV Numeric 1 = [Numeric]
17 posibleSetMV d 2 = [d .. Alphanumeric]
18 posibleSetMV x _ = posibleSetV x
```

Por último, para conocer las longitudes de la cabecera de cada modo tenemos la función `headerCost`, y para saber la longitud del indicador de modo tenemos la función `modeIndicatorLenght`.

Código 2.32: Funciones `headerCost` y `modeIndicatorLength`

```
modeIndicatorLength :: Version -> Int
modeIndicatorLength = numberVersionCase f g
   where f n = n-1
         g _ = 4

headerCost :: Version -> DataSet -> Int
headerCost v d = modeIndicatorLength v + characterCountLength d v
```

### 2.3.1. Numeric

Código 2.33: Funciones del modo Numeric

```
is :: Char -> Bool
is c = let o = ord c in o>=48 && o<=57

minVersion :: Version
minVersion = MV 1

modeIndicator :: Version -> BitString
modeIndicator = numberVersionCase f g
   where f n = integralToBitString (n-1) 0
         g _ = integralToBitString 4 1

characterCountLength :: Version -> Int
characterCountLength = numberVersionCase f g
   where f n = n+2
         g n | n<=  9 = 10
             | n<= 26 = 12
             | n<= 40 = 14

toBitString :: String -> BitString
toBitString [] = []
toBitString (x:y:z:xs) = (integralToBitString 10 (read [x,y,z] :: Int))
                         ++ toBitString xs
toBitString (x:y:_) = integralToBitString 7 (read [x,y] :: Int)
toBitString (x:_) = integralToBitString 4 (read [x] :: Int)

fromBitString :: BitString -> Maybe String
fromBitString xs = go xs (length xs) []
   where go :: BitString -> Int -> String -> Maybe String
         go xs n ys | n==0  = Just ys
                    | n==4  = Just $ (ys++) $ show $ bitStringToNum xs
                    | n==7  = Just $ (ys++) $ showInt 2 $ bitStringToNum xs
                    | n>=10 = let (a,b) = splitAt 10 xs
                              in go b (n-10) $ (ys++) $ showInt 3 $
                                 bitStringToNum a
                    | otherwise = Nothing

charCost :: Int -> Int
charCost d = case mod d 3 of
```

```
39    0 -> 3
40    1 -> 4
41    2 -> 3
42
43
44  showInt :: Int -> Int -> String
45  showInt k n = let s = show n in replicate (k-(length s)) '0' ++ s
```

### 2.3.2. Alphanumeric

Código 2.34: Funciones del modo Alphanumeric

```
1   is :: Char -> Bool
2   is c = elem c set
3
4   minVersion :: Version
5   minVersion = MV 2
6
7   modeIndicator :: Version -> BitString
8   modeIndicator = numberVersionCase f g
9     where f 1 = error "Alphanumeric not available in version M1"
10          f n = integralToBitString (n-1) 1
11          g _ = integralToBitString 4 2
12
13  characterCountLength :: Version -> Int
14  characterCountLength = numberVersionCase f g
15    where f 1 = error "Alphanumeric mode not available in version M1"
16          f n = n+1
17          g n | n<= 9 = 9
18              | n<=26 = 11
19              | n<=40 = 13
20
21  toBitString :: String -> BitString
22  toBitString [] = []
23  toBitString (x:y:xs) = integralToBitString 11 ((toInt x)*45+(toInt y)) ++
24                         toBitString xs
25  toBitString (x:_) = integralToBitString 6 $ toInt x
26
27  fromBitString :: BitString -> Maybe String
28  fromBitString xs = go xs (length xs) []
29    where go xs n ys | n==0  = Just $ reverse ys
30                     | n==6  = go [] 0 $ (fromInt $ bitStringToNum xs):ys
31                     | n>=11 = go b (n-11) $ (fromInt m):(fromInt d):ys
32                     | otherwise = Nothing
33            where (a,b) = splitAt 11 xs
34                  (d,m) = divMod (bitStringToNum a) 45
35
36  charCost :: Int -> Int
37  charCost d = if even d then 5 else 6
38
39  set :: String
40  set = ['0'..'9']++['A'..'Z']++" $%*+-./:"
```

```
41
42 toInt :: Char -> Int
43 toInt = go set 0
44    where go (x:xs) n a | x==a = n
45                        | otherwise = go xs (n+1) a
46
47 fromInt :: Int -> Char
48 fromInt n = set !! n
```

### 2.3.3. Byte

Código 2.35: Funciones del modo Byte

```
1  is :: Char -> Bool
2  is c = ord c < 256
3
4  minVersion :: Version
5  minVersion = MV 3
6
7  modeIndicator :: Version -> BitString
8  modeIndicator = numberVersionCase f g
9     where f 1 = error "Byte mode not available in MV-1"
10          f 2 = error "Byte mode not available in MV-2"
11          f n = integralToBitString (n-1) 2
12          g _ = integralToBitString 4 4
13
14
15 characterCountLength :: Version -> Int
16 characterCountLength = numberVersionCase f g
17    where f 1 = error "Byte mode not available in MV-1"
18          f 2 = error "Byte mode not available in MV-2"
19          f n = n+1
20          g n | n<= 9 =  8
21              | n<=40 = 16
22
23 toBitString :: String -> BitString
24 toBitString = integralsToBitString (repeat 8) . map ord
25
26
27 fromBitString :: BitString -> Maybe String
28 fromBitString xs | mod (length xs) 8 == 0 = Just $ map chr $
29                                             bitStringToNums (repeat 8) xs
30                  | otherwise              = Nothing
31
32
33 charCost :: Int -> Int
34 charCost = const 8
```

### 2.3.4. Terminator

El modo Terminator, a diferencia de los modos anteriores, solo tiene definidas las funciones `terminator` y `terminatorLength`. Estas funciones devuelven la cadena de bits correspodiente al modo Terminator y su longitud respectivamente, según la versión que vayamos a utilizar.

Código 2.36: Funciones del modo Terminator

```
terminator :: Version ->  BitString
terminator v = integralToBitString (terminatorLength v) 0

terminatorLength :: Version -> Int
terminatorLength = numberVersionCase (v -> 1+2*v) (const 4)
```

## 2.4– Corrección de errores

Los niveles de corrección de errores vienen dados por el tipo `ECLevel`, que consiste en una enumeración, donde cada constructor es un nivel de corrección de errores distinto.

Código 2.37: Tipo `ECLevel`

```
data ECLevel = L | M | Q | H
   deriving (Eq, Ord, Enum, Show, Ix)
```

El número de elementos correctores de errores que se utiliza en cada versión con cada nivel de corrección de errores podemos encontrarlo en la tabla de búsqueda `ecCwCount`

Código 2.38: Tabla de búsqueda `ecCwCount`

```
ecCwCount :: Array (ECLevel,Version) Int
```

Los distintos tipos de códigos de Reed-Solomon los vamos a expresar con el tipo `RSCode`, que consta de un constructor con tres valores. El primero, que se obtiene con `totalCwCount`, es el número de elementos que transmite el código ($n$); el segundo, con `dataCwCount`, es el número de datos que se quiere transmitir ($k$); y el tercero, con `correctionCapacity`, es el número de errores que puede corregir ($e$). También vamos a definir `ecCwCount`, que es el número de elementos de corrección de datos del código ($n - k$).

Código 2.39: Tipo `RSCode`

```
data RSCode = RS { rsTotalCwCount :: Int,
                   rsDataCwCount :: Int,
                   rsCorrectionCapacity :: Int}

rsEcCwCount :: RSCode -> Int
rsEcCwCount rs = (rsTotalCwCount rs) - (rsDataCwCount rs)
```

Sin embargo, para construir los códigos de Reed-Solomon que vamos a necesitar, no vamos a tener estos datos directamente, sino que tendremos cuántos elementos queremos

transmitir ($k$), cuántos elementos son correctores de errores ($n-k$) y cuántos elementos correctores no queremos que se utilicen para corregir errores. Por tanto, vamos a construir la función `rsCode`, que nos genere el código que corresponde a los datos que hemos mencionado anteriormente.

Código 2.40: Función `rsCode`

```
rsCode :: Int -> Int -> Int -> RSCode
rsCode d e m = RS (d+e) d $ div (e-m) 2
```

### 2.4.1. Codificación sobre $\mathbb{F}_{256}$

Para obtener los elementos de corrección de errores utilizamos la función `ecCodewords`

Código 2.41: Función `ecCodewords`

```
ecCodewords :: RSCode -> [GF256] -> [GF256]
ecCodewords rs ps = coefsn e $ polyMod ((makePoly ps)*(monomial 1 e)) $
                                product [x+ constPoly s | s<-solutions e]
   where e = rsEcCwCount rs

solutions :: Int -> [GF256]
solutions n = [discExp k | k<-[0 .. n-1]]
```

### 2.4.2. Decodificación sobre $\mathbb{F}_{256}$

Uno de los problemas que había para decodificar era que el número de errores era desconocido. Esto se solventaba calculando la mayor submatriz principal invertible de una cierta matriz y, después, calculando su inversa para resolver un sistema lineal. Sin embargo, cuando el número máximo de errores es grande, comprobar si muchas matrices son invertibles puede suponer mucho tiempo. Para evitar eso, podemos ir calculando la inversa de la matriz original con la eliminación de Gauss-Jordan. En el momento en el que el algoritmo de Gauss-Jordan no pueda continuar, la parte diagonalizada será la mayor submatriz principal invertible. Su inversa será la submatriz que ocupe su misma posición en la parte derecha. Todo este trabajo lo hace la función `minorInvertible`. Esto permite ahorrar mucho tiempo, pues pasamos de intentar calcular muchas inversas, tantas como el número de errores que se puede corregir, a calcular solo una.

Código 2.42: Función `menorInvertible`

```
minorInvertible :: (Eq a,Fractional a) => Matrix a -> Matrix a
minorInvertible a = sup 1 (a <|> (identity n))
   where n = nrows a
         sup m a | m > n     = inf n n a
                 | isJust k  = sup (m+1) $ cleanDown n m $
                               if m==k' then a else combineRows m 1 k' a
                 | otherwise = inf (m-1) (m-1) a
            where k = findNoNull n m a
                  k' = fromJust k
         inf 0 m = submatrix 1 m (n+1) (n+m)
```

```
11          inf k m = (inf (k-1) m) . (cleanUp k)
12
13  cleanDown :: Fractional a => Int -> Int -> Matrix a -> Matrix a
14  cleanDown n m a = go (m+1) $ scaleRow (recip $ a ! (m,m)) m a
15     where go k a | k > n     = a
16                  | otherwise = go (k+1) $ combineRows k (negate $ a ! (k,m)) m a
17
18  cleanUp :: Fractional a => Int -> Matrix a -> Matrix a
19  cleanUp m a = go (m-1) a
20     where go 0 a = a
21           go k a = go (k-1) $ combineRows k (negate $ a ! (k,m)) m a
22
23  findNoNull :: (Eq a, Num a) => Int -> Int -> Matrix a -> Maybe Int
24  findNoNull n m a = go m a
25     where go k a | k>n = Nothing
26                  | otherwise = if (a!(k,m))/=0 then Just k else go (k+1) a
```

Con la ayuda de la función anterior, la función `rsDecode` nos devuelve el mensaje original a partir de la codificación del mismo.

Código 2.43: Función `decode`

```
1   rsDecode :: RSCode -> [GF256] -> Maybe [GF256]
2   rsDecode rs xs = do restored <- restore rs $ makePoly xs
3                       return $ take (rsDataCwCount rs) $
4                              coefsn (rsTotalCwCount rs) restored
5
6   restore :: RSCode -> Poly GF256 -> Maybe (Poly GF256)
7   restore rs p | all (0==) syn = Just p
8                | isNothing mExpPos = Nothing
9                | numErrors > rsCorrectionCapacity rs = Nothing
10               | otherwise = Just $ p - (fromLists errors $ map discLog expPos)
11     where syn = [eval p s | s<-solutions $ rsEcCwCount rs]
12           mExpPos = expPosErrors rs syn
13           expPos = fromJust mExpPos
14           numErrors = length expPos
15           errors = toList $ m*(fromList numErrors 1 syn)
16              where (Right m) = inverse $ matrix numErrors numErrors
17                                ((x,y) -> (expPos !! (y-1))^(x-1))
18
19  expPosErrors :: RSCode -> [GF256] -> Maybe [GF256]
20  expPosErrors rs s = if length zeros < d2 then Nothing else Just zeros
21     where d1 = div (rsEcCwCount rs) 2
22           d2 = nrows inv
23           inv = minorInvertible $ (matrix d1 d1) $
24                 ((x,y) -> (s !! (x+y-2)))
25           b = matrix d2 1 ((x,_) -> -(s !!(d2+x-1)))
26           zeros = searchZerosBy (2*) 1 $ makePoly $ (1:) $
27                   reverse $ toList $ inv * b
```

## 2.5– Máscaras

Para las máscaras vamos a usar el tipo `Mask`, que es un sinónimo de entero. El entero que representa cada máscara es su referencia.

Código 2.44: Tipo `Mask`

```haskell
type Mask = Int
```

Dados un identificador de máscara, una versión y un módulo, la función `mask` indica si en la máscara definida por ese identificador hay que cambiar dicho módulo o no. Normalmente la utilizaremos sin indicar el módulo, de forma que sea una función que indique los módulos que hay que cambiar. También vamos a definir la función `maxMask`, que nos dice cuál es el mayor indicador de máscara disponible según la versión.

Código 2.45: Funciones `mask` y `maxMask`

```haskell
mask :: Int -> Version -> Module -> Bool
mask n = kindVersionCase (maskMV n) (maskV n)

maskV :: Int  -> Module -> Bool
maskV 0 (i,j) = mod (i+j) 2 == 0
maskV 1 (i,j) = mod i 2 == 0
maskV 2 (i,j) = mod j 3 == 0
maskV 3 (i,j) = mod (i+j) 3 == 0
maskV 4 (i,j) = mod ((div i 2)+(div j 3)) 2 == 0
maskV 5 (i,j) = (mod (i*j) 2) + (mod (i*j) 3) == 0
maskV 6 (i,j) = mod ((mod (i*j) 2) + (mod (i*j) 3)) 2 == 0
maskV 7 (i,j) = mod ((mod (i+j) 2) + (mod (i*j) 3)) 2 == 0
maskV n _ = error $ "Undefined mask: "++ show n

maskMV :: Int -> Module -> Bool
maskMV 0 = maskV 1
maskMV 1 = maskV 4
maskMV 2 = maskV 6
maskMV 3 = maskV 7
maskMV n = const $ error $ "Undefined micro mask: "++show n

maxMask :: Version -> Mask
maxMask = kindVersionCase 3 7
```

Para aplicar la máscara a un QR tenemos dos funciones distintas. La primera, `applyMask`, devuelve un nuevo `QR` donde se han cambiado los módulos que indica la máscara. La segunda, `maskModule`, devuelve una función que a cada módulo le asocia el valor que debería tener tras aplicarle la máscara. Esta segunda función es mucho más rapida, ya que no requiere crear un `Array`. Por tanto, cuando queramos ver la puntuación del QR tras aplicarle una máscara, utilizaremos la segunda, ya que no tenemos necesidad de crear el QR para cada máscara. Ésta, además, tiene un argumento adicional que corresponde al `Array` de los módulos reservados. Hacemos eso para estar seguros de que esa tabla se calcula una única vez, aunque queramos usar esa función múltiples veces con diferentes máscaras.

Para aplicar la máscara podemos utilizar la función `xor`, pues `xor False == id` y

`xor True == not`. Esto es justamente lo que queremos al aplicar la máscara, cambiar los módulos que indica la máscara y dejar igual los demás.

Código 2.46: Funciones `applyMask` y `maskModule`

```
applyMask :: Int -> Version -> QR -> QR
applyMask n v qr = array b [(i, maskModule n v qr t i) | i<-range b]
   where b = bounds qr
         t = reservedMod v

maskModule :: Int -> Version -> QR -> Array Module Bool -> Module -> Bool
maskModule n v qr t p = (if t ! p then id else xor $ mask n v p) $ qr ! p
```

## 2.6– Puntuación

Para obtener la puntuación asociada a cada máscara tenemos la función `score`. El argumento principal de esta función no es un QR, sino una función que, a cada módulo, devuelve el valor del módulo. Si se quiere utilizar con un QR, basta usarlo con la función `(qr !)`. Sin embargo, lo más habitual será utilizarla con la función `maskModule n v qr (reservedMod v)`, que tiene justamente el tipo `Module -> Bool`.

Código 2.47: Función `score`

```
score :: (Module -> Bool) -> Version -> Int
score f = sizeVersionCase (scoreMV f) (scoreV f)

scoreV :: (Module -> Bool) -> Int -> Int
scoreV f s = negate $
               sum [sameColorRow f s (i,0) 0 True | i<-[0..s]] +
               sum [sameColorColumn f s (0,i) 0 True | i<-[0..s]] +
               sum [block f (i,j) | i<-[0..s-1], j<-[i..s-1]] +
               sum [pattern [ f (i,j) | j<-[0..s]] | i<-[0..s]] +
               sum [pattern [ f (i,j) | i<-[0..s]] | j<-[0..s]] +
               proportion f s

scoreMV :: (Module -> Bool) -> Int -> Int
scoreMV f s = (min sr sl)*16 + (max sr sl)
   where sr = count [f (n,s) | n<-[0..s]]
         sl = count [f (s,n) | n<-[0..s]]


sameColorRow :: (Module -> Bool) -> Int -> Module -> Int -> Bool -> Int
sameColorRow f e (x,y) c b | y>e       = s
                           | b'==b     = n (c+1) b'
                           | otherwise = s + n 1 b'
   where b' = f (x,y)
         s  = if c>=5 then c-2 else 0
         n  = sameColorRow f e (x,y+1)

sameColorColumn :: (Module -> Bool) -> Int -> Module -> Int -> Bool -> Int
sameColorColumn f e (x,y) c b | x>e      = s
                              | b'==b     = n (c+1) b
```

```
                              | otherwise = s+ (n 1 b')
   where b' = f (x,y)
         s  = if c>=5 then c-2 else 0
         n  = sameColorColumn f e (x+1,y)

block :: (Module -> Bool) -> Module -> Int
block f (x,y) | and [b1==b2,b2==b3,b3==b4] = 3
              | otherwise                  = 0
   where b1 = f (x  , y  )
         b2 = f (x+1, y  )
         b3 = f (x  , y+1)
         b4 = f (x+1, y+1)

pattern :: [Bool] -> Int
pattern xs = (if isPrefixOf pattern1 xs then 40 else 0)+
             (if isPrefixOf pattern2 xs then 40 else 0)+
             (if null xs then 0 else pattern $ tail xs)

pattern1, pattern2 :: [Bool]
pattern1 = [False,False,False,False,True,False,True,True,True,False,True]
pattern2 = [True,False,True,True,True,False,True,False,False,False,False]

proportion :: (Module -> Bool) -> Int -> Int
proportion f s = (10*) $ round $ (20*) $ abs $ (0.5-) $
  (count $ [f p | p<- range((0,0),(s,s))]) / fromIntegral ((s+1)^2)
```

## 2.7– Procedimiento de creación

La función para crear QR de forma más generica es `makeQRWith`. Esta función tiene como argumentos: un nivel de corrección de errores; una cadena de caracteres, que es el mensaje que se quiere codificar en el QR; una versión o ninguna; y una máscara o ninguna. Si no se especifica ninguna versión, se utilizará la más pequeña en la que quepan los datos y, si se especifica una pero no caben, se utilizará la más pequeña donde quepan. Si no se especifica ninguna máscara, se utilizará la que tenga una menor penalización o, en el caso de los Micro QR, una mayor puntuación.

Código 2.48: Función `makeQRWith`

```
makeQRWith :: ECLevel -> String -> Maybe Version -> Maybe Mask -> QR
makeQRWith e s b c =  (qr //) $
                    (zip (versionLocation v) $ (vi++vi)) ++
                    (zip (formatLocation v) $ (fi++fi))
   where minV = max (capacityMinVersion e s) (fromMaybe (MV 1) b)
         costsModes = costsPerVersion s minV
         v = findVersion e minV $ map snd costsModes
         bitString = (++remainderBits v) $ encodeData e v $ encodeString v $
                      fst $ costsModes !! (index (minV, V 40) v)
         unMaskQR = array ((0,0),(size v,size v)) $ (blankQR v ++) $
                    zip (nonReservedMod v) $ bitString
         mask = maybe (betterScore unMaskQR v) id c
         qr = applyMask mask v unMaskQR
```

```
14          fi = encodeFormat e v mask
15          vi = encodeVersion v
```

Vamos a presentar poco a poco las funciones que se utilizan en este código.

### 2.7.1. Análisis de datos

Lo primero que tenemos que hacer es ver cuál es la menor versión que podemos utilizar, teniendo en cuenta el nivel de corrección de errores seleccionado y los modos que se necesitan para codificar el texto. Esto se tiene que hacer porque no todas las versiones tienen todos los modos ni todos los niveles de corrección de errores disponibles. Para ello vamos a utilizar la función `capacityMinVersion`. Esta función no solo devuelve la menor versión en la que están definidos todos los elementos que vamos a necesitar, sino que puede devolver una versión mayor si, sin hacer ningún cálculo adicional, se puede estar completamente seguro de que el texto no cabe en una versión menor. Esto podemos saberlo gracias a la tabla `capacityMinVersionTable`, que nos dice el número máximo de caracteres que se puede codificar en cada versión y cada nivel de corrección de errores.

Código 2.49: Función `capacityMinVersion`

```
1 capacityMinVersion :: ECLevel -> String -> Version
2 capacityMinVersion e s = max (modeMinVersion $ maximum $ map exclusiveSet s)
3                          $ go 0 (length s) (range (MV 1, V 40))
4    where table = capacityMinVersionTable
5          go n l (v:vs) = if l <= table ! (e,v) then v else go (n+1) l vs
6          go _ _ [] = error "QR Codes cant hold that amount of data"
7
8 capacityMinVersionTable :: Array (ECLevel,Version) Int
```

Para ayudarnos en las funciones que vamos a ver más adelante, vamos a definir los siguientes tipos sinónimos:

- `ModeSelecting` es un par formado por un modo y un entero. El entero indica cuántos caracteres vamos a codificar con ese modo.
- `Segment` es un par formado por un modo y una cadena de caracteres. Este tipo indica que la cadena de caracteres la vamos a codificar con ese modo.

Código 2.50: Tipos `ModeSelecting` y `Segment`

```
1 type ModeSelecting = (DataSet,Int)
2 type Segment = (DataSet,String)
```

Ahora, tenemos que ver cuál es la manera óptima de codificar nuestra cadena de caracteres, es decir, cuál es la elección de modos que, al codificar, resulte en una cadena de bits más corta. Este problema no es sencillo, pues algunos caracteres puede codificarse con múltiples modos y el modo que codifique con menos bits no siempre es la mejor opción, ya que cada vez que cambiemos de modo hay que incluir la cabecera de ese modo.

Para resolver este problema, lo primero que debemos ver es que si ya hemos decidido con qué modos vamos a codificar los primeros caracteres, entonces, lo único que afecta en la codificación de los demás es el modo que hayamos seleccionado para el último carácter, pues es el que necesitamos comprobar para saber si hay que incluir la cabecera, ya que esta solo hay que incluirla cuando cambiemos de modo. De forma equivalente a lo anterior, cuando tengamos dos elecciones distintas de modo para los primeros caracteres, y el último carácter se vaya a codificar con el mismo modo, entonces los modos óptimos para codificar los caracteres restantes serán iguales para ambas elecciones y, por tanto, la que hasta ahora necesite menos bits será finalmente la óptima.

Siguiendo el argumento que hemos explicado en el párrafo anterior, la función `selectModes` nos devuelve la forma óptima de codificar la cadena de entrada. Para ello, utiliza la fuerza bruta, creando todas las posibles opciones para codificar todos los caracteres, pero, en cada paso, elimina todas aquellas que coinciden en el modo que utilizan en el último carácter, dejando solo aquella que minimice el número de bits utilizado. De esta forma, estamos completamente seguros de que la elección que devuelva la función será la óptima. Además, es lo suficientemente rápida, pues tras cada paso solo mantiene como mucho tres opciones distintas de modos, aquellas que minimicen los bits que se van a utilizar pero utilicen distintos modos en el último carácter.

Código 2.51: Función `selectModes`

```
1  selectModes :: String -> Version -> ([Segment],Int)
2  selectModes s v = (recoverString s $ reverse modes,cost)
3     where (modes,cost) = minCostModes v $ map exclusiveSet s
4
5  minCostModes :: Version -> [DataSet] -> ([ModeSelecting],Int)
6  minCostModes v s =  minimumBy f $ foldl go [([],0)] s
7     where go xs c = [minimumBy f [include v a b | b<-xs] | a<-posibleSet c v]
8           f = compare `on` snd
9
10 recoverString :: String -> [ModeSelecting] -> [(DataSet,String)]
11 recoverString [] [] = []
12 recoverString xs ((d,n):ys) = (d,a):(recoverString b ys)
13    where (a,b) = splitAt n xs
14
15 include :: Version -> DataSet -> ([ModeSelecting],Int) -> ([ModeSelecting],Int)
16 include v c ([],_) = ([(c,1)], charCost c 1 + headerCost v c)
17 include v c (ys@((d,m):xs),n)
18    | d == c = ((d,m+1):xs, n + charCost c (m+1))
19    | otherwise = ((c,1):ys, n + charCost c 1 + headerCost v c)
```

Por desgracia, el número de bits que se necesitan para codificar depende de la versión que se esté utilizando, pues el número de bits para el encabezado de cada modo cambia según la versión, por lo que la elección óptima puede cambiar de una versión a otra. Sin embargo, no podemos seleccionar una versión hasta que no sepamos cuántos bits vamos a necesitar utilizar y veamos las capacidades de las versiones. Para resolver este problema, vamos a utilizar la función `costsPerVersion`. Esta función indica el número de bits necesarios para codificar la cadena en cada versión y la elección de modos óptima en cada caso. Como muchas de las versiones sí que comparten el número de bits de la cabecera, la función se encarga de hacer los cálculos solo en aquellas versiones en las que pueda cambiar, que son

en los Micro QR y en las versiones 1, 10 y 27.

Código 2.52: Función `costsPerVersion`

```
costsPerVersion :: String -> Version -> [([Segment],Int)]
costsPerVersion s minV = drop (index (MV 1, V 40) minV) $
                     [selectModes s $ MV n | n<-[1..4]] ++
                      replicate 9  (selectModes s $ V 1 ) ++
                      replicate 17 (selectModes s $ V 10) ++
                      replicate 14 (selectModes s $ V 27)
```

Ahora tenemos que ver cuántos bits hay disponibles en cada versión. Para ello, vamos a utilizar una serie de funciones auxiliares que nos permitan conocer cuántos módulos se utilizan para diferentes zonas:

- `totalModCount` indica cuántos módulos tiene en total cada versión.
- `reservedModCount` es una tabla de búsqueda con la cantidad de módulos reservados.
- `nonReservedModCount` indica cuántos módulos son para datos y bits de corrección de errores.
- `dataModCount` indica cuántos módulos son para datos.
- `remainderBitsCount` indica cuántos módulos de los destinados a datos y a bits de corrección de errores no se usan nunca y quedan siempre en blanco.

Código 2.53: Funciones para calcular el número de módulos

```
totalModCount :: Version -> Int
totalModCount v = (1 + size v)^2

reservedModCount :: Array Version Int

nonReservedModCount :: Version -> Int
nonReservedModCount v = totalModCount v - (reservedModCount ! v)

dataModCount :: ECLevel -> Version -> Int
dataModCount e v = nonReservedModCount v - (ecCwCount ! (e,v))*8 -
                   remainderBitsCount v

remainderBitsCount :: Version -> Int
remainderBitsCount = versionCase (const 0) f
   where f v = mod (nonReservedModCount v) 8
```

Gracias a todo lo anterior, podemos ver qué versión vamos a utilizar con la función `findVersion`. Esta función toma como argumentos el nivel de corrección de errores, la minima versión donde todo está bien definido y la lista del número de bits necesarios para codificar la cadena de caracteres en cada versión, empezando por la versión que se ha indicado en el argumento anterior, y devuelve la menor versión en la que caben nuestros datos.

Código 2.54: Función `findVersion`

```
findVersion :: ECLevel -> Version -> [Int] -> Version
findVersion e m = go $ range (m,V 40)
   where go [] [] = error "QR codes cant hold that amount of data"
         go (v:vs) (c:cs) | c <= dataModCount e v = v
                          | otherwise = go vs cs
```

### 2.7.2. Codificación de datos

Una vez conocida la versión, los modos que vamos a utilizar para codificar son aquellos que ha devuelto la función `costsPerVersion` para esa versión. Para codificar según esta elección usamos `encodeString`.

Código 2.55: Función `encodeString`

```
encodeString :: Version -> [Segment] -> BitString
encodeString v = foldr f $ terminator v
   where f (x,y) bs = modeIndicator x v ++ integralToBitString
                      (characterCountLength x v) (length y) ++
                       toBitString x y ++ bs
```

Para pasar nuestra cadena de bits a elementos de $\mathbb{F}_{256}$ vamos a utilizar la función `bitStringToNums`. El número de bits que corresponden a cada elemento es normalmente 8. Sin embargo, en las versiones M1 y M3, el último elemento viene determinado solo por los 4 últimos bits de la cadena de bits. Para trabajar con esto de manera más simple, vamos a definir la función `shortLastword`, que indica en qué versiones ocurre este fenómeno. También vamos a definir la función `shortLastwordVersionCase`, que funciona como la función `versionCase`, es decir, toma dos algoritmos y una versión y, si en esa versión ocurre el fenómeno anterior, se utiliza el primer algoritmo, mientras que en caso contrario, se utiliza el segundo.

Código 2.56: Función `shorLastwordVersionCase`

```
shortLastwordVersionCase :: (Version -> a) -> (Version -> a) -> Version -> a
shortLastwordVersionCase f g = versionCase f' g
   where f' v | shortLastword v = f v
              | otherwise = g v

shortLastword :: Version -> Bool
shortLastword = numberVersionCase odd (const False)
```

El número de elementos que van a corresponder a datos nos lo da la función `dataCwCount`.

Código 2.57: Función `dataCwCount`

```
dataCwCount :: ECLevel -> Version -> Int
dataCwCount e v = div (dataModCount e v) 8 + if shortLastword v then 1 else 0
```

Con la ayuda de las funciones anteriores, podemos definir la funciones `wordsLength` y `padCodewords`, que indican cuántos bits definen cada elemento y cuáles son los elementos de relleno respectivamente.

Código 2.58: Funciones `wordsLength` y `padCodewords`

```
wordsLength :: ECLevel -> Version -> [Int]
wordsLength e = shortLastwordVersionCase f (const c)
   where f v = take (dataCwCount e v - 1) c ++ (4:c) ++ repeat 8
         c = repeat 8


padCodewords :: Int -> Version -> [GF256]
padCodewords n =  shortLastwordVersionCase
                   (const $ if n==0 then [] else (take (n-1) c) ++ [0])
                   (const $ take n c)
   where c = cycle [236,17]
```

### 2.7.3. Generación de los elementos correctores de errores

El número de bloques en los que hay que dividir la lista de elementos que hemos calculado anteriormente podemos encontrarlo en la tabla de búsqueda `ecBlocks`. Para saber qué código de Reed-Solomon vamos a utilizar en cada bloque, necesitamos saber cuántos elementos son de datos, cuántos son correctores de errores y cuántos elementos correctores de errores no vamos a utilizar para corregir errores.

Código 2.59: Tabla de búsqueda `ecBlocks`

```
ecBlocks :: Array (ECLevel, Version) Int
```

La función `dataCwCount` indica el número de elementos que corresponden a datos en cada versión. Para saber cuántos elementos de datos van a cada bloque, hay que dividir este número entre el número de bloques. En caso de que no sea divisible, el resto se reparte entre los últimos bloques. El número de elementos correctores de errores en cada bloque lo tenemos en la tabla de búsqueda `ecCwPerBlock`, y el número de éstos que no se utiliza para corregir errores lo devuelve la función `miscodeProtectionCw`.

Código 2.60: Tablas de búsqueda `ecCwPerBlock` y `miscodeProtectionCW`

```
miscodeProtectionCw :: ECLevel -> Version -> Int
miscodeProtectionCw e v | number v > 4 = 0
                        | otherwise = miscodeProtectionCw4 ! (e,v)

miscodeProtectionCw4 :: Array (ECLevel,Version) Int
ecCwPerBlock :: Array (ECLevel,Version) Int
```

Una vez conocido todo esto, la función `rsCodes` nos devuelve la lista de códigos de Reed-Solomon que hay que utilizar en los distintos bloques.

Código 2.61: Función `rsCodes`

```
rsCodes :: ECLevel -> Version -> [RSCode]
rsCodes e v = [rsCode (d + if n>b-r then 1 else 0) f m  | n<-[1..b]]
   where (d,r) = divMod (dataCwCount e v) b
         f = ecCwPerBlock ! (e,v)
```

```
5           m = miscodeProtectionCw e v
6           b = ecBlocks ! (e,v)
```

Después, utilizamos la función `blockDivision` para hacer la división de los elementos en los distintos bloques, cada uno con su código de Reed-Solomon correspondiente.

Código 2.62: Función `blockDivision`

```
1 blockDivision :: [RSCode] -> [a] -> [(RSCode, [a])]
2 blockDivision [] [] = []
3 blockDivision (r:rs) xs = (r,i):(blockDivision rs f)
4    where (i,f) = splitAt (rsDataCwCount r) xs
```

Finalmente, usamos `ecBlocks` para calcular los elementos de corrección de errores de cada bloque.

Código 2.63: Función `ecBlocks`

```
1 ecBlock :: (RSCode,[GF256]) -> [GF256]
2 ecBlock (r,s) = ecCodewords r s
```

### 2.7.4. Construcción de la secuencia de datos final

Para juntar en el orden correcto todos los bloques que hemos dividido antes y para añadir al final los elementos correctores de errores, también en el orden correcto, vamos a utilizar la función `assemble`. Por último, utilizamos la función `numsToBitString` para volver a convertir cada elemento en una cadena de bits.

Código 2.64: Función `Assemble`

```
1 assemble :: [[a]] -> [[a]] -> [a]
2 assemble xs ys = go [] xs ++ go [] ys
3    where go [] [] = []
4          go xss [] = go [] $ reverse xss
5          go xss ([]:yss) = go xss yss
6          go xss ((y:ys):yss) = y:(go (ys:xss) yss)
```

Es la función `encodeData` la que se encarga de hacer el conjunto de las acciones definidas desde que se pasa la cadena de bits a elementos de $\mathbb{F}_{256}$ hasta ahora, que se vuelve a transformar en una cadena de bits.

Código 2.65: Función `encodeData`

```
1 encodeData :: ECLevel -> Version -> BitString -> BitString
2 encodeData e v xs = integralsToBitString w $
3                     assemble (map snd blocks) (map ecBlock blocks)
4    where blocks = blockDivision (rsCodes e v) $
5                   cw++(padCodewords (c-length cw) v)
6          c = dataCwCount e v
7          cw = fst $ splitAt c $ bitStringToNums w xs
8          w = wordsLength e v
```

### 2.7.5. Colocación de los datos en la matriz

Para especificar qué valor debe tener cada módulo, vamos a crear una lista de pares que asocien cada módulo con su valor. Después, usaremos la función `array` para transformar la lista anterior al tipo `QR`. Como la información de formato y de versión todavía no debe aparecer, debemos dejar sus módulos en blanco. La función `reservedArea` deja los módulos del área especificada en blanco.

Código 2.66: Función `reservedArea`

```
reservedArea :: Int -> Int -> Int -> Int -> [(Module,Bool)]
reservedArea = placeModules $ const $ const $ False
```

Con la función `blankQR` tenemos ya una única lista de todos los módulos reservados con sus valores.

Código 2.67: Función `blankQR`

```
blankQRV :: Int -> [(Module,Bool)]
blankQRV v = let s = sizeV v in concat $
   [finderPattern 0 0, finderPattern 0 (s-6), finderPattern (s-6) 0,
    vWhite 0 7, vWhite 0 (s-7), vWhite (s-6) 7,
    hWhite 7 0, hWhite 7 (s-7), hWhite (s-7) 0,
    blackAndWhite 1 (s-15) 6 8, blackAndWhite (s-15) 1 8 6,
    blackAndWhite 1 1 (s-7) 8, reservedArea 6 1 0 8,
    reservedArea 2 1 7 8, reservedArea 1 1 8 7, reservedArea 1 6 8 0,
    reservedArea 1 8 8 (s-7), reservedArea 7 1 (s-6) 8,
    if v >= 7 then reservedArea 6 3 0 (s-10) else [],
    if v >= 7 then reservedArea 3 6 (s-10) 0 else []] ++
    [alignmentPatter (i-2) (j-2) | (i,j)<- alignmentPatternLocationV v]

blankQRMV :: Int -> [(Module,Bool)]
blankQRMV v = let s = sizeMV v in concat $
   [finderPattern 0 0, vWhite 0 7, hWhite 7 0,
    blackAndWhite 1 (s-7) 0 8, blackAndWhite (s-7) 1 8 0,
    vWhite 1 8, hWhite 8 1]
```

Para los datos, la función `nonReservedMod` se encarga de poner en una lista todos los módulos que no están reservados en el orden en el que se deben colocar los bits.

Código 2.68: Función `nonReservedMod`

```
nonReservedMod :: Version -> [Module]
nonReservedMod v = (s,s):(go (s,s))
   where t = reservedMod v
         s = size v
         go (x,y) | (snd n) < 0    = []
                  | t ! n     = go n
                  | otherwise = n:(go n)
            where n = nextModule
                        (isMicro v, y>6, mod (y+k) 4, x, x==s) (x,y)
                  k = if isMicro v && odd (number v) then 2 else 0

           -- (isMicro v, y>6 , mod y 4, x  , x==s-1)
```

```
13 nextModule :: (Bool     , Bool, Int    , Int, Bool  ) -> Module -> Module
14
15 nextModule (_, True , 0, _, _    ) (x,y) = (x  , y-1)
16 nextModule (_, True , 3, 0, _    ) (x,y) = (x  , y-1)
17 nextModule (_, True , 3, _, _    ) (x,y) = (x-1, y+1)
18 nextModule (_, True , 2, _, _    ) (x,y) = (x  , y-1)
19 nextModule (_, True , 1, _, True ) (x,y) = (x  , y-1)
20 nextModule (_, True , 1, _, False) (x,y) = (x+1, y+1)
21
22 nextModule (False, False, _, a, b) (x,y) =
23             nextModule (False, True, mod (y+1) 4, a, b) (x,y)
24 nextModule (True , False, a, b, c) (x,y) =
25             nextModule (False , True, a, b, c) (x,y)
```

Gracias a esto, solo necesitamos usar la función `zip` con la lista devuelta por esta función y los bits que debemos colocar para tener asociado cada bit con su módulo. Sin embargo, antes de hacerlo, debemos añadir a la cadena de bits un cero por cada módulo que falte por rellenar. De esto se encarga la función `remainderBits`.

Código 2.69: Función `remainderBits`

```
1 remainderBits :: Version -> BitString
2 remainderBits v = take (remainderBitsCount v) $ repeat False
```

### 2.7.6. Selección de la máscara

Si no se ha especificado una máscara, deberemos ver cual tiene mejor puntuación. Para ello, usaremos la función `betterScore`, que devuelve la máscara que obtiene mayor puntuación. Tras esto, usaremos la función `applyMask` para aplicar al QR la máscara que corresponda, ya sea la que se hubiera especificado o la que hemos obtenido con la función anterior.

Código 2.70: Función `betterScore`

```
1 betterScore :: QR -> Version -> Int
2 betterScore qr v = snd $ maximumBy (compare `on` fst)
3        [(score (maskModule n v qr t) v, n) | n<-[0 .. maxMask v]]
4    where t = reservedMod v
```

### 2.7.7. Información de formato y versión

La forma de obtener los bits de corrección de errores es muy parecida tanto en la información de formato como en la información de versión; la única diferencia es el polinomio que se utiliza. Por tanto, vamos a crear un tipo común, muy parecido al tipo de los códigos de Reed-Solomon, que nos ayude al calcular los bits de corrección de errores en ambos casos. El tipo `BHCCode` solo va a tener un constructor con tres valores. El primero, que se obtiene con `bhcTotalCwCount`, es el número de bits totales, tanto los que queremos transmitir como los de corrección de errores; el segundo, con `bhcDataCwCount`, es el número

de bits que queremos transmitir; y el tercero, con `bhcCorrectionCapacity`, es el número de errores que podemos corregir. También vamos a definir `bhcEcCwCount`, que devuelve el número de bits de corrección de errores.

Código 2.71: Tipo `BHCCode`

```
data BHCCode = BHC { bhcTotalCwCount :: Int,
                     bhcDataCwCount :: Int,
                     bhcCorrectionCapacity :: Int}

bhcEcCwCount :: BHCCode -> Int
bhcEcCwCount rs = bhcTotalCwCount rs - bhcDataCwCount rs
```

El polinomio que se utiliza en cada caso nos lo da la función `polyGen`. Solo hemos definido los polinomios que nos hacen falta para los casos que vamos a tratar, que son 10 y 12 bits de corrección de errores.

Código 2.72: Función `polyGen`

```
polyGen :: BHCCode -> Poly Bool
polyGen rs = makePoly $ case bhcEcCwCount rs of
   10 -> [1,0,1,0,0,1,1,0,1,1,1]
   12 -> [1,1,1,1,1,0,0,1,0,0,1,0,1]
   _  -> error "BHCCode non defined"
```

La cadena de bits final, con los bits que se quieren transmitir y los bits de corrección de errores, nos la da la función `bhcEncode`.

Código 2.73: Función `bhcEncode`

```
bhcEncode :: BHCCode -> BitString -> BitString
bhcEncode rs xs = (xs ++) $ coefsn e $
                  polyMod ((monomial 1 e)*(makePoly xs)) $ polyGen rs
   where e = bhcEcCwCount rs
```

### Información de formato

Para la información de formato, necesitamos conocer el indicador del nivel de corrección de errores y la máscara de información de formato. Para el indicador, usaremos dos funciones: `vIndicatorMV`, para los Micro QR y `eclIndicator`, para las demás versiones. Para la máscara usaremos la función `infoMask`.

Código 2.74: Funciones `vIndicatorMV`, `ecLevelIndicator`, e `infoMask`

```
infoMask :: Version -> BitString
infoMask = kindVersionCase p q
   where p = [1,0,0,0,1,0,0,0,1,0,0,0,1,0,1]
         q = [1,0,1,0,1,0,0,0,0,0,1,0,0,1,0]

ecLevelIndicator :: ECLevel -> BitString
ecLevelIndicator ecl = case ecl of
                    L -> [0,1]
```

```
                    M -> [0,0]
                    Q -> [1,1]
                    H -> [1,0]

vIndicatorMV :: ECLevel -> Version -> BitString
vIndicatorMV e v = case (e,v) of
   (L, MV 1) -> [0,0,0]
   (L, MV 2) -> [0,0,1]
   (M, MV 2) -> [0,1,0]
   (L, MV 3) -> [0,1,1]
   (M, MV 3) -> [1,0,0]
   (L, MV 4) -> [1,0,1]
   (M, MV 4) -> [1,1,0]
   (Q, MV 4) -> [1,1,1]
```

Para obtener los bits correctores de errores utilizamos el código que genera 15 bits, de los cuales 5 son de datos y corrige hasta 3 errores.

Código 2.75: Código de corrección de errores para la Información de formato

```
bhcCodeFormat :: BHCCode
bhcCodeFormat = BHC 15 5 3
```

Con ayuda de todo lo anterior, la función que nos da la información de formato es `encodeFormat`.

Código 2.76: Función `encodeFormat`

```
encodeFormat :: ECLevel -> Version -> Mask -> BitString
encodeFormat e v m = zipWith (+) (infoMask v) $ bhcEncode bhcCodeFormat $
                     versionCase (vIndicatorMV e) (const $ ecLevelIndicator e) v
                     ++ integralToBitString (kindVersionCase 2 3 v) m
```

### Información de versión

Para obtener los bits correctores de errores se utiliza el código que genera 18 bits, de los cuales 6 son de datos y corrige hasta 3 errores.

Código 2.77: Código de corrección de errores para la Información de versión

```
bhcCodeVersion :: BHCCode
bhcCodeVersion = BHC 18 6 3
```

La función que nos da la información de versión es `encodeVersion`

Código 2.78: Función `encodeVersion`

```
encodeVersion :: Version -> BitString
encodeVersion = bhcEncode bhcCodeVersion . integralToBitString 6 . number
```

### QR final

Para obtener el QR final, solo tenemos que actualizar del QR, con las cadenas de bits correspondientes, los módulos que previamente habíamos reservado para la información de versión y para la información de formato.

## 2.8– Procedimiento de lectura

Para obtener el texto que se ha codificado en un QR vamos a utilizar la función `decodeQR`. El resultado será `Nothing` si no hemos sido capaces de decodificar el QR o `Just` y una cadena de texto si hemos podido hacerlo. Hacemos uso de la mónada booleana para manejar de forma más sencilla los casos de error, pues, a diferencia de cuando queremos crear un QR, que no deberían saltar errores, a la hora de leerlos, es muy posible que haya fallos.

Código 2.79: Función `decodeQR`

```
1 decodeQR :: QR -> Maybe String
2 decodeQR qr = do
3    v <- getVersion qr
4    (e,m) <- getFormatInformation qr v
5    bitString <- decodeData e v $ take
6                 (nonReservedModCount v - remainderBitsCount v) $
7                 [ xor (qr ! p) $ mask m v p | p<-nonReservedMod v]
8    decodeString v $ bitString
```

Vamos a analizar este código poco a poco.

### 2.8.1. Versión

Como no vamos a tratar con imágenes, sino con el tipo `QR`, la versión nos la dirá el tamaño de la matriz. La funciones `getSize` y `getVersion` nos dicen el tamaño y la versión del QR respectivamente. No vamos a utilizar la información de formato porque, con el tipo `QR`, no hay dudas de cuantos módulos forman el QR; a diferencia de cuando se decodifica a través de una imágen, que es más complicado acertar el número exacto de módulos cuando hay muchos.

Código 2.80: Funciones `getSize` y `getVersion`

```
1 getVersion :: QR -> Maybe Version
2 getVersion qr = unSize $ getSize qr
3
4 getSize :: QR -> Int
5 getSize =  snd . snd . bounds
```

### 2.8.2. Nivel de corrección de errores y máscara

Para obtener el nivel de corrección de errores y la máscara, debemos decodificar los bits que están en los módulos de la información de formato. Para decodificar, vamos a utilizar

la función `bhcDecode`. Esta función utiliza la fuerza bruta, de forma que, dadas todas las codificaciones posibles, devuelve aquella que difiera con la de entrada en 3 bits o menos.

Código 2.81: Función `bhcDecode`

```
bhcDecode :: BHCCode -> BitString -> [(BitString,a)] -> Maybe a
bhcDecode rs s [] = Nothing
bhcDecode rs s ((x,y):xs) | (e>=) $ count $ zipWith (+) s x = Just y
                          | otherwise = bhcDecode rs s xs
   where e = bhcCorrectionCapacity rs
```

La función `decodeFormat` codifica todos los posibles niveles de corrección de errores y máscaras.

Código 2.82: Función `decodeFormat`

```
decodeFormat :: BitString -> Version -> Maybe (ECLevel, Mask)
decodeFormat s v = bhcDecode bhcCodeFormat s [(encodeFormat e v m,(e,m)) |
                   e<-[L .. maxECLevel v], m<-[0..maxMask v]]
```

Finalmente, la función `getFormatInformation` selecciona los módulos correspondientes a la información de formato y los decodifica gracias a las funciones anteriores. En el caso de las versiones de la 1 a la 40, donde la información de formato está duplicada, también comprueba que, si las dos se han decodificado correctamente, no indiquen máscaras o un nivel de corrección de errores distintos.

Código 2.83: Función `getFormatInformation`

```
getFormatInformation :: QR -> Version -> Maybe (ECLevel,Mask)
getFormatInformation qr = versionCase
                          (getFormatInformationMV qr)
                          (getFormatInformationV qr)

getFormatInformationMV :: QR -> Version -> Maybe (ECLevel,Mask)
getFormatInformationMV qr v = decodeFormat (formatModules qr v) v

getFormatInformationV :: QR -> Version -> Maybe (ECLevel,Mask)
getFormatInformationV qr v = go (decodeFormat f1 v) (decodeFormat f2 v)
   where (f1,f2) = splitAt 15 $ formatModules qr v
         go Nothing y       = y
         go x       Nothing = x
         go x       y       = if x==y then x else Nothing

formatModules :: QR -> Version -> BitString
formatModules qr v = [qr ! p | p<-formatLocation v]
```

### 2.8.3. Cadena de datos codificados

La cadena de bits codificados la obtenemos aplicando de nuevo la máscara y seleccionando los módulos no reservados, es decir, aquellos que no forman parte de los patrones requeridos, la información de versión o la información de formato. También hay que descartar los bits restantes, que son los que no se rellenan en algunas versiones.

### 2.8.4. Corrección de errores

Para corregir los errores, primero tenemos que volver a ver la cadena como elementos de $\mathbb{F}_{256}$, para lo cual utilizamos la función `bitStringToNums`. Después, debemos separar los elementos en los bloques en los que se codificaron originalmente. Para ello, utilizamos la función `unAssemble`, que toma una lista con los elementos que transmiten datos y otra con los elementos correctores de errores y devuelve una lista con los distintos bloques.

Código 2.84: Función `unAssemble`

```
unAssemble :: Int -> [a] -> [a] -> [[a]]
unAssemble n xs ys = zipWith (++) (unAssemble' xs) (unAssemble' ys)
   where unAssemble' zs = go (replicate n []) zs (length zs)
         go zss zs m
          | m == 0 = zss
          | m <  n = let (a,b) = splitAt (n-m) zss
                     in (a++) $ zipWith (++) b $ transpose [zs]
          | otherwise = let (a,b) = splitAt n zs
                        in go (zipWith (++) zss $ transpose [a]) b (m-n)
```

Después, decodificamos cada bloque con el código de Reed-Solomon que le corresponde con la función `rsDecode` y concatenamos los resultados de los bloques, previamente transformados de nuevo a una cadena de bits con la función `integralsToBitString`. Todo este trabajo en su conjunto lo realiza la función `decodeData`.

Código 2.85: Función `decodeData`

```
decodeData :: ECLevel -> Version -> BitString -> Maybe BitString
decodeData e v xs = fmap (integralsToBitString words) $ foldl1 (liftM2 (++)) $
            zipWith rsDecode (rsCodes e v) $ unAssemble (ecBlocks ! (e,v)) a b
   where (a,b) = splitAt (dataCwCount e v) $ bitStringToNums words xs
         words = wordsLength e v
```

### 2.8.5. Decodificación de los datos

La función `decodeString` lleva a cabo el último paso y transforma la cadena de bits en la cadena de caracteres correspondientes a la decodificación.

Código 2.86: Función `decodeString`

```
decodeString :: Version -> BitString -> Maybe String
decodeString v xs = do modes <- unSelectModes v xs
                       let list = map ((x,y) -> fromBitString x y) modes
                       if any isNothing list then Nothing
                       else Just $ concat $ catMaybes list

unSelectModes :: Version -> BitString -> Maybe [(DataSet,BitString)]
unSelectModes v xs = go [] xs
   where go ys xs | isPrefixOf (terminator v) xs = Just $ reverse ys
                  | all not xs = Just $ reverse ys
                  | null mode = Nothing
                  | otherwise = let (a,b) = splitBitString v zs d
```

```
                                in go ((d,a):ys) b
            where mode = detectMode v xs
                  (d,zs) = head mode

detectMode :: Version -> BitString -> [(DataSet,BitString)]
detectMode v xs = catMaybes $ fmap
   ( x -> fmap ( y -> (x,y)) $ stripPrefix (modeIndicator x v) xs)
   [Numeric,Alphanumeric,Byte]


splitBitString :: Version -> BitString -> DataSet ->
                  (BitString, BitString)
splitBitString v xs d = flip splitAt b $
                        sum [charCost d i | i<-[1 .. bitStringToNum a]]
   where (a,b) = splitAt (characterCountLength d v) xs
```

## 2.9– Visualización

Para visualizar los QR creados, es decir, el tipo `QR`, hemos creado una instancia de la clase `Show`. Hay que tener en cuenta que el espacio para los caracteres es el doble de alto que de ancho. Por lo tanto, cada dos espacios se corresponden con un módulo del QR.

Código 2.87: Instancia de la clase `Show` del tipo `QR`

```
instance Show QR where
   show qr = let nlines = replicate 4 'n'
                 spaces = replicate 8 ' '
                 s = getSize qr
                 g n m | m == s && n==s = c ++ nlines
                       | m == s         = c ++ ('n':spaces) ++ g (n+1) 0
                       | otherwise      = c ++ g n (m+1)
                    where c = if qr!(n,m) then "■" else "  "
             in nlines ++ spaces ++ g 0 0
```

## 2.10– Guardado

Para guardar los QR creados vamos a ayudarnos del paquete Juicy Pixels. Con la función `createImage` vamos a poder crear una imagen de dicho paquete a partir de un QR. El primer argumento es el número de píxeles que queramos que tenga la imagen de lado y, el segundo, el número mínimo de módulos que queremos que haya en la zona tranquila.

Código 2.88: Función `createImage`

```
createImage :: Int -> Int -> QR -> DynamicImage
createImage size minBorder qr = if size<minSize
                                then error $ "Min size is "++show
                                     minSize++" pixels"
                                else ImageY8 $ generateImage f size size
   where minSize = qrSize+2*minBorder
```

```
7           qrSize = getSize qr
8           (scale,restBorder) = divMod size minSize
9           border = minBorder*scale + (div restBorder 2)
10          f x y | and [x'>=0,y'>=0,x'<=qrSize,y'<=qrSize] =
11                      if qr ! (y',x') then 0 else 255
12                | otherwise = 255
13              where x' = div (x-border) scale
14                    y' = div (y-border) scale
```

Para poder guardar una imagen creada con la función anterior, necesitamos especificar una extensión en la que la vayamos a guardar. El tipo `Extension` es una enumeración de las extensiones en las que se puede guardar.

Código 2.89: Tipo `Extension`

```
1 data Extension = PNG | JPG | BMP | GIF | TIFF | HDR
```

Para finalmente guardar un QR como imagen vamos a utilizar la función `saveImage`. Esta función tiene como argumentos:

- La extensión con la que se quiere guardar el archivo.
- Una ruta donde guardar el archivo. Para adentrarnos en las carpetas debemos usar la barra lateral (/) en lugar de la barra invertida (\). Además, la última carpeta debe acabar sin barra.
- Un nombre para poner al archivo, que no debe llevar extensión.
- El número de pixeles que queremos que tenga la imagen.
- El número mínimo de módulos que queremos que tenga la zona tranquila.

El resultado es que la imagen se guarda con los parámetros que hemos establecido.

Código 2.90: Función `saveImage`

```
1 saveImage :: Extension -> FilePath -> String -> Int -> Int -> QR -> IO()
2 saveImage ext path name size minBorder qr = case ext of
3         PNG -> savePngImage (path++name++".png") i
4         JPG -> saveJpgImage 100 (path++name++".jpg") i
5         BMP -> saveBmpImage (path++name++".bmp") i
6         GIF -> either error id $ saveGifImage (path++name++".gif") i
7         TIFF -> saveTiffImage (path++name++".tiff") i
8         HDR -> saveRadianceImage (path++name++".hdr") i
9    where i = createImage size minBorder qr
```

## 2.11– Otras funciones útiles

En esta sección vamos a definir las funciones dirigidas al usuario que no hayamos definido anteriormente.

Para saber cuál es la menor versión en la que se puede codificar un texto con un nivel de corrección de errores vamos a utilizar la función `minVersion`. Funciona de manera similar a la parte de `makeQRWith` que selecciona la versión que se va a utilizar. Esto quiere decir que, si no hay ninguna versión que pueda almacenar esos datos, devuelve error. Para las situaciones en la que no queremos que salte un error, vamos a definir la función `hasCapacity`, que nos dice si hay alguna versión que pueda codificar el texto o no.

Código 2.91: Funciones `minVersion` y `hasCapacity`

```
minVersion :: ECLevel -> String -> Version
minVersion e s = findVersion e minV $ map snd $ costsPerVersion s minV
   where minV = capacityMinVersion e s

hasCapacity :: ECLevel -> String -> Bool
hasCapacity e s = dataModCount e (V 40) >= (snd $ selectModes s $ (V 40))
```

Por otro lado, vamos a definir una serie de funciones para crear QR, variantes de la función `makeQRWith`. Las funciones `makeQR`, `makeJustQR` y `fastQR` crean el QR seleccionando algunos de los parámetros de `makeQRWith`. Las funciones `url` y `email` añaden el contenido necesario para que el lector de QR detecte que está leyendo una dirección web o un correo electrónico respectivamente y optimiza la cadena todo lo posible.

Código 2.92: Otras funciones para crear QR

```
makeQR :: ECLevel -> String -> QR
makeQR e s = makeQRWith e s Nothing Nothing

makeJustQR :: ECLevel -> String -> QR
makeJustQR e s = makeQRWith e s (Just $ V 1) Nothing

fastQR :: ECLevel -> String -> QR
fastQR e s = makeQRWith e s Nothing (Just 0)

url :: ECLevel -> String -> QR
url e s = makeQR e $ scheme ++ map toUpper a ++ b
   where (a,b) = splitAt (fromMaybe maxBound $ elemIndex '/' web) web
         d1 = stripPrefix "http://" s
         d2 = stripPrefix "https://" s
         (scheme,web) = if isJust d1 then ("HTTP://",fromJust d1) else
                        ("HTTPS://", fromMaybe s d2)

email :: ECLevel -> String -> QR
email e s = makeQR e $ "mailto:" ++ a ++ map toUpper b
   where (a,b) = splitAt (fromMaybe maxBound $ elemIndex '@' s) s
```

## 2.12– Comprobación

Por último, vamos a comprobar que el proceso de crear y decodificar QR se realiza de forma correcta. Para ello, necesitamos funciones que generen de manera aleatoria los elementos para crear un QR aleatorio.

Para elegir una versión y un nivel de corrección de errores vamos a usar las funciones `arbitraryVersion` y `arbitraryECLevel` respectivamente. La segunda tiene como argumento una versión, pues no todas las versiones tienen disponibles todos los niveles de corrección de errores.

Código 2.93: Funciones `arbitraryVersion` y `arbitraryECLevel`

```
arbitraryECLevel :: Version -> Gen ECLevel
arbitraryECLevel v = elements $ case v of
   (MV 1) -> [L]
   (MV 2) -> [L,M]
   (MV 3) -> [L,M]
   (MV 4) -> [L .. Q]
   _      -> [L .. H]

arbitraryVersion :: Gen Version
arbitraryVersion = do n<-chooseInt (0,43)
                      return $ (range (MV 1, V 40)) !! n
```

Generar una cadena de caracteres es algo más complicado, pues debemos asegurarnos de que la cadena que generemos se pueda codificar en un QR con la versión y el nivel de errores que se hayan seleccionado anteriormente. Para ello, vamos a ir generando los caracteres hasta que se complete la capacidad. Cada carácter no se genera de manera uniforme, sino que se favorece que se genere un cáracter del mismo conjunto que el anterior. De esta forma, se favorece que salga una cadena en la que se combinen varios modos, en lugar de que se utilice únicamente el modo Byte. En ocasiones, escogeremos solo una parte de la cadena en lugar de la cadena completa, evitando así que siempre se complete la capacidad del QR.

Código 2.94: Función `arbitraryString`

```
arbitraryChars :: DataSet -> [DataSet] -> Int -> Gen (DataSet,String)
arbitraryChars d _  0 = return (d,"")
arbitraryChars d ds n = do m <- frequency [(95,elements [d]),
                                           (5 ,elements ds )]
                           c <- arbitraryChar m
                           nexts <- arbitraryChars m ds (n-1)
                           return $ fmap (c:) nexts

arbitraryChar :: DataSet -> Gen Char
arbitraryChar ds = case ds of
      Numeric -> arbitraryNumeric
      Alphanumeric -> arbitraryAlphanumeric
      Byte -> arbitraryByte

arbitraryNumeric :: Gen Char
arbitraryNumeric = elements ['0' .. '9']

arbitraryAlphanumeric :: Gen Char
arbitraryAlphanumeric = elements $ ['0'..'9']++['A'..'Z']++" $%*+-./:"

arbitraryByte :: Gen Char
arbitraryByte = elements ['0' .. '255']
```

```
23
24 arbitraryString :: ECLevel -> Version -> Gen String
25 arbitraryString e v = do m <- elements $ dataSets v
26                         fstChar <- arbitraryChar m
27                         xs <- go m [fstChar]
28                         total <- chooseEnum (False,True)
29                         n <- chooseInt (1,length xs)
30                         elements $ pure $ if total then xs else take n xs
31    where dataSets (MV 1) = [Numeric]
32          dataSets (MV 2) = [Numeric,Alphanumeric]
33          dataSets _      = [Numeric .. Byte]
34          capacity = dataModCount e v
35          go m xs = let cost = snd $ selectModes xs v
36                    in if capacity < cost then return $ init xs
37                       else do
38               (m',ys) <- arbitraryChars m (dataSets v) $ max 1 $ flip div 8 $
39                          capacity - cost - (headerCost v $ maximum $ dataSets v)
40               go m' (xs++ys)
```

También debemos generar una lista aleatoria de errores. Para ello, vamos a utilizar la función `arbitraryErrors`. Esta función toma como argumentos el nivel de corrección de errores, una versión y un número entre 0 y 1, y devuelve una lista aleatoria de posiciones de los errores. El número entre 0 y 1 reduce el número máximo de errores, de forma que con un 1, el número máximo es el máximo teórico que se puede corregir, mientras que con 0 no devuelve ningún error.

Código 2.95: Función `arbitraryErrors`

```
1  arbitraryErrors :: Float -> ECLevel -> Version -> Gen [Module]
2  arbitraryErrors f e v = do n <- chooseInt (0,max)
3                             go n
4     where max = round $ (f*) $ fromIntegral $ sum $
5                 map rsCorrectionCapacity $ rsCodes e v
6           go 0 = return []
7           go n = do x <- chooseInt (0,size v)
8                     y <- chooseInt (0,size v)
9                     nexts <- go $ n-1
10                    return $ (x,y):nexts
```

Debemos introducir este número, pues tenemos un problema a la hora de corregir errores. El número máximo de errores que un QR puede corregir lo sabemos por el nivel de corrección de errores y los códigos de Reed-Solomon que se utilizan. Sin embargo, si estos errores se distribuyen de manera aleatoria por el QR, puede ocurrir que en un bloque haya más errores de los que se puede corregir en ese bloque y, por tanto, hacer que el QR no se pueda decodificar. Todavía más, pues, con la suficiente mala suerte, 8 errores bastan para hacer que cualquier QR no se pueda decodificar. Si caen 4 errores en cada zona de información de formato, no podremos recuperar el nivel de corrección de errores ni la máscara, por lo que el QR no se puede decodificar.

Podemos reducir la probabilidad de que esto ocurra reduciendo el máximo de errores que vamos a aplicar, cosa que hacemos multiplicando el máximo teórico por un número

entre 0 y 1 que se pide como argumento a la función. Sin embargo, no vamos a poder utilizar el paquete `QuickCheck`, pues este fenómeno puede seguir ocurriendo. En su lugar, vamos a utilizar nuestras propias funciones para comprobar que los algoritmos están bien implementados. A estas comprobaciones les vamos a exigir que siempre decodifiquen bien cuando no apliquemos los errores, mientras que vamos a permitir que fallen a veces cuando apliquemos los errores en el QR.

La primera función de comprobación es `iosample`, que toma como argumentos una versión y un número para reducir el número máximo de errores, y devuelve un par de booleanos, donde el primero indica una correcta codificación y decodificación sin aplicar errores y el segundo, aplicando los errores. Si no se especifica una versión, se utiliza una al azar. Por lo que hemos dicho anteriormente, exigimos que el primero siempre sea `True`, mientras que consideramos aceptable, sobre todo si no reducimos considerablemente el número máximo de errores, que el segundo valga `False`.

Código 2.96: Función `iosample`

```
iosample :: Maybe Version -> Float -> IO (Bool,Bool)
iosample v1 f = do
  v2 <- generate arbitraryVersion
  let v = fromMaybe v2 v1
  e <- generate $ arbitraryECLevel v
  string <- generate $ arbitraryString e v
  err <- generate $ arbitraryErrors f e v
  let qr = makeQRWith e string (Just v) Nothing
      qre = qr // [(p, not $ qr ! p) | p<-err]
      b1 = decodeQR qr == Just string
      b2 = decodeQR qre == Just string
  putStrLn $ "ECLevel: " ++ show e ++ "nVersion: " ++ show v ++
     " nErrors: "++ show (length err) ++"nDecoded correctly without errors: " ++
     show b1 ++ "nDecoded correctly using errors: " ++ show b2
  return (b1,b2)
```

Para usar la función anterior múltiples veces vamos a usar la función `generateSamples`, cuyo primer argumento nos dice el número de repeticiones. Por último, vamos a definir la función `quickCheck`, que consiste en una abreviatura para aplicar la función anterior con 100 repeticiones en diferentes versiones escogidas al azar y reduciendo el número máximo de errores a dos tercios del teórico.

Código 2.97: Funciones `generateSamples` y `quickCheck`

```
generateSamples :: Int -> Maybe Version -> Float -> IO()
generateSamples n v f = do {go 1 (0,0); return ()}
   where go i (a,b)
           | i>n = do
              putStrLn $ "Number of Examples: " ++ show n ++
                 "nDecoded correctly without errors: " ++ show a ++
                 "nDecoded correctly using errors: " ++ show b ++ "n"
              return (a,b)
           | otherwise = do
              putStrLn $ "Example " ++ show i
              (a',b') <- iosample v f
              putStrLn "n"
```

```
13              go (i+1) (if a' then a+1 else a, if b' then b+1 else b)
14
15  quickCheck :: IO ()
16  quickCheck = generateSamples 100 Nothing 0.66
```

# CAPÍTULO 3

## Manual de uso

El módulo `Codec.QR` tiene definidos los tipos y funciones para poder crear códigos QR y, en menor medida, leerlos. A la hora de crearlos, los modos se seleccionan siempre de manera automática de la mejor forma posible, por lo que el usuario no tiene que prestar atención en ellos. Sin embargo, solo están definidos los modos Numeric, Alphanumeric y Byte, por lo que solo se pueden codificar caracteres que se encuentren en la codificación *latin-1*.

### 3.1– Tipos

`type QR`

Tipo que representa los QR creados. Se puede utilizar como si fuera del tipo `Array Module Bool`, gracias a que su tipo sinónimo tiene definida una instancia de la clase `IArray`.

Instancias

`Show QR`

`type Module = (Int,Int)`

Tipo que representa los módulos de los QR.

`data ECLevel`

Tipo que representa el nivel de corrección de errores. Hay 4 niveles, cada uno es capaz de recuperar una cantidad de datos distinta. Sin embargo, cuantos más datos se quiera recuperar, más grande va a ser la versión que se tenga que utilizar. Lo recomendable es usar el nivel `M`.

Constructores

`L` Recupera hasta un 7 % de datos perdidos

`M` Recupera hasta un 15 % de datos perdidos (recomendado)

`Q` Recupera hasta un 25 % de datos perdidos

`H` Recupera hasta un 30 % de datos perdidos

Instancias

`Eq ECLevel`

`Ord ECLevel`

`Enum ECLevel`

`Ix ECLevel`

`Show ECLevel`

`data Version`

Tipo que representa la versión del QR. Cuanto más grande sea la versión, mayor será la matriz del QR. Lo recomendable es utilizar siempre la menor versión posible, pues esto ayuda a que se decodifique de manera correcta más fácilmente. Sin embargo, no todos los lectores de códigos QR soportan las versiones Micro QR.

Constructores

`V :: Int -> Version` Para las versiones de la 1 a la 40

`MV :: Int -> Version` Para las versiones de la M1 a la M4

Instancias

`Eq Version`

`Ord Version`

`Ix Version`

`Show Version`

`type Mask = Int`

Tipo para las máscaras. En las versiones de la 1 a la 40 hay 8 máscaras disponibles, desde la 0 a la 7; mientras que en las versiones de la M1 a la M4 solo hay 4, desde la 0 a la 3.

## 3.2– Creación

`makeQR :: ECLevel -> String -> QR`

Codifica el texto en un QR con el nivel de corrección de errores indicado. Esta es la función que debe utilizarse para crear códigos QR si no entendemos muy bien su funcionamiento.

`makeJustQR :: ECLevel -> String -> QR`

Codifica el texto en un QR con el nivel de corrección de errores indicado sin utilizar las versiones correspondientes a los Micro QR. Esto es útil cuando no estemos seguros de si el decodificador que se vaya a utilizar para leer el QR soporta los Micro QR.

```
fastQR :: ECLevel -> String -> QR
```

Codifica el texto en un QR con el nivel de corrección de errores y utilizando la máscara especificada. Esta función es más rápida que `makeQR`; sin embargo, puede resultar en un QR más difícil de detectar. No se recomienda su uso.

```
url :: ECLevel -> String -> QR
```

Codifica una dirección web con el nivel de corrección de errores indicado. Aprovecha que los dominios no son sensibles a las mayúsculas para reducir la versión utilizada e incluye el encabezado `https://` en caso de no tenerlo.

```
email :: ECLevel -> String -> QR
```

Codifica una dirección de correo electrónico con el nivel de corrección de errores indicado. Aprovecha que los dominios no son sensibles a las mayúsculas para reducir la versión utilizada, e incluye el encabezado para que la aplicación que lo lea lo entienda como una dirección de correo electrónico.

```
makeQRWith :: ECLevel -> String -> Maybe Version -> Maybe Mask -> QR
```

Codifica el texto en un QR con el nivel de corrección de errores indicado. Opcionalmente se puede indicar una versión, de forma que el QR utilizará como mínimo esa versión, y una máscara. Especificar una versión puede ser útil para que, al crear un grupo de QR, todos utilicen la misma versión, o para evitar que se utilicen Micro QR. Especificar una máscara (entre 0 y 7 si sabemos que no se va a utilizar un Micro QR o entre 0 y 3 si no estamos seguros) hace que la función sea más rápida; sin embargo, puede resultar en un QR más difícil de detectar. No se recomienda especificar una máscara.

## 3.3– Lectura

```
decodeQR :: QR -> Maybe String
```

Decodifica un QR y devuelve el texto, en caso de haberlo decodificado con éxito, o `Nothing`, si algún paso ha fallado.

Se cumple que

```
decodeQR (makeQR e s) == Just s
```

Esto también ocurrre con las funciones `makeJustQR`, `fastQR` y `makeQRWith`. Sin embargo, no ocurre con las funciones `url` y `email`, pues estas modifican el texto de entrada.

```
>>> decodeQR (makeQR Q "Codigo QR") == Just "Codigo QR"
True
>>> decodeQR (url M "https://github.com/AlStinson/QR")
Just "HTTPS://GITHUB.COM/AlStinson/QR"
```

## 3.4– Funciones auxiliares

`minVersion :: ECLevel -> String -> Version`

Devuelve la menor versión que se puede utilizar para codificar un QR dados el texto que se quiere codificar y el nivel de corrección de errores.

```
>>> minVersion L (replicate 150 'a')
V  7
>>> minVersion L (replicate 15000 'a')
*** Exception: QR Codes cant hold that amount of data
CallStack (from HasCallStack): error, called at src\Codec\QR\String\Encode.
    hs:54:22 in qr-microqr-code-0.1.0.0-7VfI9aXvUr670Dwg4puGUo:Codec.QR.
    String.Encode
```

`hasCapacity :: ECLevel -> String -> Bool`

Indica si hay alguna versión que sea capaz de codificar el texto con el nivel de corrección de errores indicado.

```
>>> hasCapacity L (replicate 150 'a')
True
>>> hasCapacity L (replicate 15000 'a')
False
```

## 3.5– Guardado en archivos externos

`data Extension = PNG | JPG | BMP | GIF | TIFF | HDR`

Tipo que representa los distintos formatos disponibles para guardar un QR en un archivo externo.

`saveImage`

- `:: Extension` Extensión que se quiere utilizar
- `-> FilePath` Carpeta donde se quiere guardar
- `-> String` Nombre del archivo
- `-> Int` Dimensiones de la imagen
- `-> Int` Módulos de la zona tranquila
- `-> QR` QR que se quiere guardar
- `-> IO()`

Guarda un QR en un archivo externo según las opciones especificadas. El nombre del archivo no debe llevar extensión; esta se pone de manera automática según la extensión escogida. Para adentrarnos en las carpetas debemos usar la barra lateral (/) en lugar de la barra invertida (\) y la última carpeta debe acabar sin barra. Se recomienda un mínimo de 2 módulos para la zona tranquila en el caso de Micro QR y 4 módulos para los QR.

```
>>> saveImage PNG "c:/QR" "QR" 200 4 (makeQR L "QR")
>>>
```

## 3.6– Comprobaciones

Para utilizar las funciones de comprobación, hay que importar el módulo `Test.QR`.

`iosample`

> `:: Maybe Version` Versión del QR
>
> `-> Float` Reducción del número de errores máximo
>
> `-> IO(Bool,Bool)` Resultados

Genera un QR aleatorio y comprueba, en primer lugar, si se codifica y decodifica bien y, luego, si se decodifica bien aplicándole errores generados aleatoriamente. El primer resultado siempre debe ser `True`, mientras que el segundo puede ser `False`. Si no se especifica ninguna versión, se generará una aleatoria.

```
>>> iosample Nothing 1
ECLevel: Q
Version: V  37
Errors: 357
Decoded correctly without errors: True
Decoded correctly using errors: True
(True,True)
```

`generateSamples :: Int -> Maybe Version -> Float -> IO()`

Genera tantos QR aleatorios con la función `iosample` como se indique en su primer argumento. El número total de QR que se decodifican bien se muestra por pantalla.

```
>>> generateSamples 10 (Just $ MV 4) 1
...
Number of Examples: 10
Decoded correctly without errors: 10
Decoded correctly using errors: 10
```

`quickCheck :: IO`

Para comprobar si la librería funciona bien sin tener que entender cómo funciona. Genera 100 QR aleatorios y comprueba si se decodifican bien; primero, sin aplicar errores, y luego, aplicándoselos. Consideramos que la librería funciona correctamente si el segundo número que se muestra por pantalla cuando acaba la función es 100 y el último es cercano a 100.

Es un sinónimo para `generateSamples 100 Nothing 0.66`.

```
>>> quickCheck
...
Number of Examples: 100
Decoded correctly without errors: 100
Decoded correctly using errors: 97
```

## 3.7– Ejemplos

`makeQR L "CODIGO QR"`
(0.05 secs, 1,170,696 bytes)

`makeJustQR L "CODIGO QR"`
(0.06 secs, 4,367,304 bytes)

`makeQR H "Código QR"`
(0.08 secs, 8,492,456 bytes)

`fastQR H "Código QR"`
(0.06 secs, 5,164,616 bytes)

`url M "github.com/AlStinson/QR"`
(0.06 secs, 8,221,472 bytes)

`email M "javidelgado1997@gmail.com"`
(0.11 secs, 12,737,264 bytes)

# CAPÍTULO 4

## Conclusiones

Como se ha visto en los capítulos anteriores, hemos sido capaces de crear una librería que nos brinda las funciones y los tipos necesarios para poder crear códigos QR, mostrarlos por pantalla, almacenarlos como archivos independientes en muchos de los formatos de imágenes más habituales y leerlos para recuperar el texto codificado, incluso aunque se produzcan cambios en algunos de los módulos del QR. Para ello hemos estudiado en profundidad el funcionamiento y la estructura de los códigos QR.

El principal problema que hemos encontrado a la hora de estudiar de manera teórica los códigos QR ha sido que fueron diseñados para utilizarse con una codificación orientada a los caracteres japoneses, como es *Shift Jis*, y no con la codificación *latin-1*. Esto hace que, por ejemplo, el modo Kanji pierda sentido y sea complicado entender su funcionamiento. Además, aunque el estándar dice que la codificación por defecto es *latin-1*, muchos desarrolladores han optado por usar *utf-8* en su lugar. Esto tiene la ventaja de que se puede codificar cualquier carácter sin usar el modo ECI. Sin embargo, puede dar problemas de decodificación si la aplicación con la que se creo el QR y la que lo decodifica usan codificaciones distintas de manera predeterminada. En este trabajo hemos seguido el estańdar y hemos usado *latin-1* como codificación predeterminada.

En la implementación, por un lado, hemos tenido problemas relacionados con las codificaciones de caracteres, por lo que decidimos no implementar los modos Kanji y ECI, aunque sí los hubiéramos estudiado de manera teórica. También tuvimos problemas con los modos que no iban destinados a codificar caracteres, por lo que también decidimos no implementarlos. Además, en este caso, su implementación es opcional, según indica el propio estándar, por lo que no es usual usar estos modos.

A pesar de lo anterior, podemos concluir que hemos cumplido con los objetivos que se habían marcado para el trabajo. Como trabajo futuro, nos queda por implementar: los modos que no se han implementado y el algoritmo para decodificar un QR a partir de una imagen. También hay que seguir intentando optimizar las funciones ya existentes todo lo posible.

# Bibliografía


[1] Denso Wave. *Official QR Website.* https://www.qrcode.com/en/.

[2] Wikipedia, the free encyclopedia. *QR Code.* https://en.wikipedia.org/wiki/QR_code.

[3] *ISO/IEC 18004:2015 — Information technology — Automatic identification and data capture techniques — QR Code bar code symbology specification.*

[4] Thonky. *QR Code Tutorial.* https://www.thonky.com/qr-code-tutorial/.

[5] *Project Nayuki.* https://github.com/nayuki/QR-Code-generator.

[6] Chris Yuen. *qrcode: QR Code library in pure Haskell.* https://hackage.haskell.org/package/qrcode.

[7] Alex Kazik. *qrcode-core: QR code library in pure Haskell.* https://hackage.haskell.org/package/qrcode-core.

[8] Michael Jahn. *ZXing.Net.* https://github.com/micjahn/ZXing.Net/tree/master/Source/lib/qrcode.

[9] Vincent Berthoux. *JuicyPixels: Picture loading/serialization.* https://hackage.haskell.org/package/JuicyPixels.

[10] Wikiversity. *Reed–Solomon codes for coders.* https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders.

[11] Justin Handville. *AES Reference Implementation in Haskell.* https://unconceived.net/blog/2015/01/29/aes-reference-haskell.html.

[12] Wikipedia, the free encyclopedia. *Finite field arithmetic.* https://en.wikipedia.org/wiki/Finite_field_arithmetic.

[13] Wikipedia, the free encyclopedia. *Reed–Solomon error correction.* https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction.

[14] Stackoverflow question. *Calculating the position of QR Code alignment patterns.* https://stackoverflow.com/questions/13238704/calculating-the-position-of-qr-code-alignment-patterns.

# Índice de tablas

# Índice de código

# Índice alfabético de funciones y tipos