# Teoría de los Lenguajes de Programación Práctica curso 2020-2021

# Enunciado

Fernando López Ostenero y Ana García Serrano

## 1. Descripción de la misión: el teorema de los cuatro colores.

El teorema de los cuatro colores fue enunciado en 1852 y establece que:

> *Cualquier mapa geográfico con regiones continuas puede ser coloreado con un máximo de cuatro colores diferentes, de forma que dos regiones adyacentes no compartan color.*

entendiendo por regiones adyacentes aquellas que comparten entre sí una frontera mayor que un punto.

La demostración de este teorema, llevada a cabo en 1976 por Kenneth Appel y Wolfgang Haken, es famosa por ser la primera demostración matemática realizada con la ayuda de un ordenador, lo que también suscitó la desconfianza de muchos ante la imposibilidad de comprobar "a mano" la corrección de dicha demostración.

Pero un grave problema ha sido detectado: alguien está intentando cambiar el pasado y ha convencido a unos jóvenes Appel y Haken acerca de la inutilidad de los ordenadores en sus investigaciones. Esto supondrá que nunca demuestren el teorema de los cuatro colores, hecho que generará una ola temporal que hará que nunca se considere la informática como una ciencia digna de ser estudiada en la universidad.

Su misión, si decide aceptarla, será viajar en el tiempo con un equipo especializado y mostrar a Appel y Haken cómo un ordenador puede ayudarles en su investigación. Para ello, se le pide crear un programa que dado un mapa rectangular de regiones (numeradas desde 1 en adelante), devuelva una lista de los colores a usar en cada región del mapa, con un máximo de cuatro colores diferentes, de forma que dos regiones adyacentes no tengan el mismo color.

Como estudiante de informática, ¿se apunta a salvar el futuro de sus estudios y así no perder todo el esfuerzo invertido en llegar hasta donde está ahora?

## 2. Enunciado de la práctica

La práctica consiste en elaborar un programa en **Haskell** que permita al usuario cargar diferentes ficheros de texto con mapas. Una vez cargado un mapa, el programa buscará una forma de colorearlo con tan solo cuatro colores.

La práctica se divide en dos fases diferentes:

1. **Análisis del mapa**: se cargará un fichero de texto que contiene el mapa (cuyo formato supondremos válido) y se analizará el mapa para obtener la lista de adyacencias entre zonas.
2. **Búsqueda de una solución**: una vez se disponga de la lista de adyacencias entre zonas, el programa buscará una solución al problema de asignar colores a cada una de las zonas, mediante un esquema de backtracking, utilizando para ello un máximo de cuatro colores diferentes.

Dado que la entrada/salida en el paradigma funcional no es parte del temario que se estudia en la asignatura, todas las funciones encargadas de leer el mapa desde un fichero o de la interacción con el usuario se darán programadas para los estudiantes. En este enunciado se dará una explicación del funcionamiento de las mismas.

## 2.1 Formato de los mapas

En primer lugar vamos a ver cuál es el formato de los mapas. Partimos de mapas rectangulares dibujados sobre una cuadrícula como el siguiente:

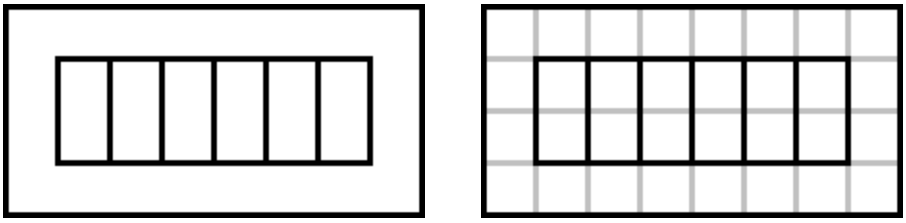

*Figura 1: mapa original (izquierda) y mapa sobre la cuadrícula (derecha).*

Ahora podemos numerar las diferentes zonas, comenzando por el 1 y utilizando números consecutivos. Para codificar el mapa, se asignará a cada elemento de la cuadrícula la zona a la que pertenece. Continuando con el ejemplo:

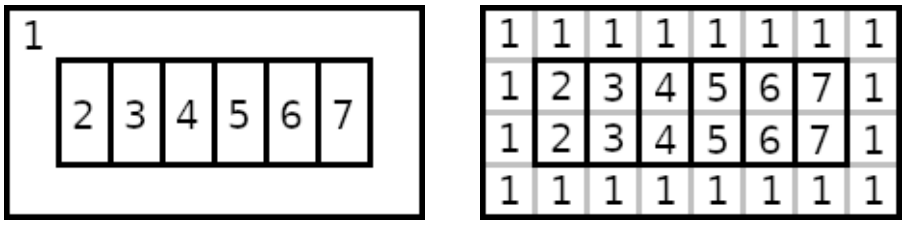


Figura 2: mapa con zonas. Sin cuadricular (izquierda) y sobre la cuadrícula (derecha).

Esto nos lleva a una matriz rectangular como la que sigue:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

*Figura 3: mapa codificado en forma de matriz rectangular.*

la cual codificaremos en **Haskell** mediante una lista de filas donde cada una de las cuales será, a su vez, otra lista:

```
[ [1,1,1,1,1,1,1,1],
  [1,2,3,4,5,6,7,1],
  [1,2,3,4,5,6,7,1],
  [1,1,1,1,1,1,1,1] ]
```

## 2.2 Ejemplos de funcionamiento

A continuación vamos a dar algunos ejemplos del funcionamiento de la práctica. Junto a este enunciado se proporcionan algunos ficheros de mapas que pueden ser cargados desde la práctica.

**Ejemplo 1**: el programa carga el mapa del fichero `small.map` (el usado en el ejemplo anterior), lo muestra, busca una solución y la encuentra:

```
*Main> main
Enter mapfile to load: small.map
Loading map from file small.map
[ [1,1,1,1,1,1,1,1],
  [1,2,3,4,5,6,7,1],
  [1,2,3,4,5,6,7,1],
  [1,1,1,1,1,1,1,1]
]
Solution: [Red,Green,Blue,Green,Blue,Green,Blue]
Enter mapfile to load:
```

Al existir una solución, el programa nos la muestra. Si ahora vamos a nuestro mapa original y coloreamos cada región con el color que nos indica la solución, tendríamos lo siguiente:

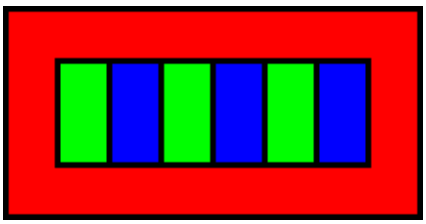

*Figura 4: mapa coloreado*.

En este caso el mapa no necesita cuatro colores diferentes, ya que puede ser coloreado con sólo tres.

**Ejemplo 2**: el programa carga el fichero `nosolution.map` con un mapa en el que todas sus regiones son no contiguas (inclumpliendo los requisitos impuestos por el teorema de los cuatro colores), por lo que no encuentra ninguna solución de coloreado con sólo cuatro colores:

```
Enter mapfile to load: nosolution.map
Loading map from file nosolution.map
[ [1,2,2,3,3,4,4,5],
  [1,3,2,4,3,5,4,1],
  [1,4,2,5,3,1,4,1],
  [3,5,2,1,3,1,4,1]
]
No solution!
Enter mapfile to load:
```

**Ejemplo 3**: el usuario no desea comprobar más mapas, por lo que introduce un nombre de fichero en blanco. Como resultado, el programa termina su ejecución:

```
Enter mapfile to load:
End of Line.
```

## *2.3 Estructura de módulos de la práctica*

La práctica está dividida en dos módulos, cada uno en un fichero independiente cuyo nombre coincide (incluyendo mayúsculas y minúsculas) con el nombre del módulo y cuya extensión es `.hs`:

1. Módulo `FourColors`: dentro de este módulo se programarán las funciones de análisis del mapa y búsqueda de una solución. Incluye las definiciones de todos los tipos de datos utilizados en la práctica.

2 Módulo `Main`: este módulo contiene la función principal y las encargadas de interactuar con el usuario y cargar los mapas de ficheros externos. Este módulo se proporciona ya totalmente programado y no deberá ser modificado.

Dado que un estudio más profundo de los módulos en **Haskell** está fuera del ámbito de la asignatura, sólo indicaremos que el módulo `Main` importa (además del módulo `FourColors`) un módulo adicional para poder trabajar más cómodamente con la entrada/salida y poder acceder a ficheros.

## *2.4 Programa principal (módulo Main)*

El módulo `Main` contiene el programa principal, ya programado por el equipo docente. Hay funciones que utilizan mónadas y están escritas utilizando la "*notación do*", que es una notación para facilitar la escritura de concatenaciones de funciones monádicas. Sin entrar en detalle sobre el funcionamiento de las mónadas, vamos a explicar qué hace cada función:

- `readMap`: carga el fichero de texto indicado que contiene un mapa, lo muestra por pantalla y lo convierte en un dato de tipo `Map` (cuya definición exporta el módulo `FourColors`).
- `loop`: esta función es el "*bucle principal*" del programa. Pregunta al usuario el nombre del fichero con el mapa a cargar, lo carga mediante una llamada a la función `readMap` e imprime el resultado de buscar una forma de colorearlo utilizando un máximo de cuatro colores mediante una llamada a la función `color` (que deberá ser programada en el módulo `FourColors`) del diccionario. Cuando el usuario introduce un nombre de fichero vacío, finaliza la ejecución del programa.
- `main`: es la función principal. Configura la entrada para que se pueda editar (si no, no tendría efecto la tecla de retroceso) y llama a la función `loop`.

Como ya se ha indicado, este módulo se entrega ya programado por el equipo docente. Para su correcto funcionamiento hay que programar las funciones necesarias en el módulo `FourColors`.

## *2.5 Primera fase: análisis del mapa (módulo FourColors)*

En este apartado vamos a tratar sobre el análisis del mapa para obtener las adyacencias de zonas. En primer lugar, veamos los tipos de datos (ya programados):

```
type Zone      = Int
type Row       = [Zone]
type Map       = [Row]
type Adjacency = (Zone,[Zone])
```

- El tipo `Zone` va a representar una zona del mapa mediante un entero. Las zonas deberán numerarse comenzando por el 1 y de forma consecutiva (es decir, no puede existir una zona 6 sin que haya una zona 5, por ejemplo).
- El tipo `Row` va a representar una fila del mapa, compuesta por zonas.
- El tipo `Map` va a representar un mapa, compuesto por filas. Se asumirá que todas las filas del mapa tienen la misma longitud.
- El tipo `Adjacency` representa las adyacencias de una zona `z`, expresadas como una tupla formada por la zona `z` y una **lista ordenada** de todas aquellas zonas (con un número menor que `z`) que son adyacentes a la zona `z`. Así, por ejemplo, el valor `(4,[1,3])` significará que las zonas adyacentes a la 4 (con un número menor que 4) son la 1 y la 3.

El objetivo de esta fase es programar una función `adjacencies`:

```
adjacencies :: Map -> [Adjacency]
```

que recibe un mapa y devuelve una lista de zonas adyacentes, que deberá estar **ordenada por zonas**. Es decir, primero aparecerán las adyacencias de las zonas con un menor índice. Por ejemplo, si queremos obtener las adyacencias del mapa utilizado como ejemplo en la sección 2.1 (contenido en el fichero `small.map`), el resultado de `adjacencies` sería:

```
> adjacencies [[1,1,1,1,1,1,1,1],[1,2,3,4,5,6,7,1],[1,2,3,4,5,6,7,1],[1,1,1,1,1,1,1,1]]
[(2,[1]),(3,[1,2]),(4,[1,3]),(5,[1,4]),(6,[1,5]),(7,[1,6])]
```

nótese que primero aparecen las adyacencias de la zona 2, luego las de la 3 y así sucesivamente en orden hasta las adyacencias de la zona 7. Y, además, las listas de adyacencia de cada zona también están ordenadas.

A continuación describiremos el algoritmo que se deberá seguir para encontrar las adyacencias: para cada una de las filas del mapa, se recorren todos sus elementos y se compara cada uno con el elemento de su izquierda (en su misma fila) y con el elemento superior (que será el que ocupa la misma posición en la fila anterior). Si alguna de las comparaciones resulta ser diferente, se añadirá la nueva adyacencia detectada siempre que no se hubiera añadido con anterioridad.

Para realizar esta tarea serán necesarias algunas funciones auxiliares. Dado que en **Haskell** no se pueden hacer bucles, cada recorrido implicará una función diferente, lo que significa que habrá una función que recorra las filas, otra que recorra los elementos de una fila, otra que recorra las adyacencias ya insertadas para buscar la tupla adyacencia adecuada en cuyo segundo elemento añadir la nueva zona (si es que no había sido añadida ya) mediante otra función.

## *2.6 Búsqueda de una solución (módulo FourColors)*

Una vez tengamos la lista de adyacencias de un mapa, realizaremos una búsqueda mediante backtracking para encontrar una solución que coloree ese mapa con un máximo de cuatro colores. Al igual que en la fase anterior, veamos en primer lugar los tipos de datos que vamos a utilizar:

```
data Color    = Red | Green | Blue | Yellow
  deriving (Enum,Eq,Show)
type Solution = [Color]
type Node     = ([Adjacency],Zone,Zone,Solution)
```

- El tipo color nos define los cuatro colores que vamos a utilizar. Además, lo instanciamos automáticamente en las clases `Enum`, `Eq` y `Show` para poder enumerarlo, dotarlo de la operación de igualdad y poder visualizarlo respectivamente.
- El tipo solución representa, como su propio nombre indica, una solución a nuestro problema, formada por una lista de colores: uno para cada zona del mapa.
- El tipo nodo es el que utilizaremos en nuestro backtracking. Contiene información sobre la lista de adyacencias del mapa, el número de zonas del mismo, la siguiente zona que vamos a considerar y la solución parcial que hayamos construido hasta ese momento.

El objetivo de esta fase es programar una función `color`:

```
color :: Map -> String
```

que dado un mapa nos devuelva una cadena con una lista de colores para las zonas del mapa si existe solución, o bien que nos indique que no hay solución. Esta función ya se encuentra programada en el módulo `FourColors`, al igual que las funciones:

- Función `solution`: es la función que devuelve todos los nodos solución del problema.
- Función `bt`: es una función que implementa el esquema de backtracking.
- Función `colorList`: función constante que devuelve la lista de todos los colores (gracias a haber instanciado el tipo `Color` en la clase `Enum`).

A continuación mostramos una lista de las funciones que utilizan las anteriores y se han de programar, aunque estas funciones pueden necesitar otras funciones auxiliares:

- Función `esSol`: es una función que dado un nodo nos indica si dicho nodo contiene ya una solución o no.
- Función `comp`: es una función que dado un nodo (que no es una solución) calcula las compleciones de dicho nodo, es decir: todos los nodos hijos de dicho nodo que se acercan un paso más a la solución. En este caso, los hijos de un nodo tendrán una zona más coloreada, siempre respetando las restricciones del problema.
- Función `initialNode`: es una función que dado un mapa crea el nodo inicial del mismo. Para ello utilizará la función `adjacencies` creada en la primera fase.

## 3. Cuestiones sobre la práctica

La respuesta a estas preguntas es optativa. Sin embargo, si el estudiante no responde a estas preguntas, la calificación de la práctica sólo podrá llegar a 6 puntos sobre 10.

1 (*1'5 puntos*). Supongamos una implementación de la práctica en un lenguaje no declarativo (como **Java**, **Pascal**, **C**…). Comente qué ventajas y qué desventajas tendría frente a la implementación en **Haskell**. Relacione estas ventajas desde el punto de vista de la eficiencia con respecto a la programación y a la ejecución. ¿Cuál sería el principal punto a favor de la implementación en los lenguajes no declarativos? ¿Y el de la implementación en **Haskell**?

2 (*1'5 puntos*). Indique, con sus palabras, qué permite gestionar el predicado predefinido no lógico, corte (!), en **Prolog**. ¿Cómo se realizaría este efecto en **Java**? Justifique su respuesta.

3 (*1 punto*). Para los tipos de datos del problema definidos en **Haskell**, indique qué clases de constructores de tipos se han utilizado en cada caso (ver capítulo 5 del libro de la asignatura).

## 4. Documentación a entregar

Cada estudiante deberá entregar la siguiente documentación a su tutor de prácticas:

- Código fuente en **Haskell** que resuelva el problema planteado. Para ello se deberán entregar los ficheros `Main.hs` y `FourColors.hs`, con las funciones descritas en este enunciado, así como todas las funciones auxiliares que sean necesarias.
- Una memoria con las respuestas a las cuestiones sobre la práctica.