Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información
CI3661 - Taller de Lenguajes de Programación I
Enero-Marzo 2011

Programación Funcional - Entrega 2
Sudoku

En esta etapa del proyecto Ud. ampliará el módulo de resolución de Sudoku con algunas funciones adicionales, para luego construir un programa en Haskell que recibe la especificación del Sudoku a resolver, mostrando el resultado, si existe, en pantalla. El programa ha de correr desde la línea de comandos, por lo que necesitará ser compilado empleando GHC, bien sea con la ayuda de `hmake` o mediante la construcción de un `Makefile` a la medida.

Leyendo y escribiendo Sudokus

El programa tendrá que leer la especificación de uno o más Sudokus a resolver a partir de un archivo de texto simple, de manera que sea fácil suministrar varios acertijos para resolver en una sola operación.

Adoptaremos una representación textual simple y fácil de procesar. Así, en un archivo de texto, el Sudoku de nuestro ejemplo inicial[1] se representará como

```
36..712..
.5....18.
..92.47..
....13.28
4..5.2..9
27.46....
..53.89..
.83....6.
..769..43
```

esto es, cada Sudoku se representará con nueve líneas seguidas, cada una de las cuales tiene nueve caracteres representando los valores para cada columna. Si hubiera espacios en blanco *antes* o *después* del cuerpo de una fila, estos deben ser ignorados; sin embargo, **nunca** habrá espacios en blanco *entre* dos posiciones de una fila.

Si un archivo contiene más de un Sudoku, habrá una línea vacía entre cada pareja de Sudoku. El primer Sudoku en el archivo *siempre* comenzará en la primera línea, y el último Sudoku en el archivo *nunca* tendrá líneas en blanco después.

Cualquier representación que se desvíe de lo especificado anteriormente debe ser reportada como un error razonablemente preciso (caracteres inválidos, estructura inválida, etc.).

Con esta especificación, usted debe implantar la función

```
readSudokus :: FilePath -> IO [Either String Sudoku]
```

que dado un nombre de archivo, prepara la secuencia de acciones de I/O que procesan el archivo analizando su contenido para determinar si hay uno o más Sudokus válidos, y en caso de error proveerá información acerca de las causas.

[1] El mismo que aparece en el enunciado del Proyecto 1.

El programa también mostrará los Sudoku en pantalla, así que usted debe implantar la función

```
printSudoku :: Sudoku -> IO ()
```

cuyo resultado en pantalla para nuestro ejemplo debe lucir como

```
36.|.71|2..
.5.|...|18.
..9|2.4|7..
---+---+---
...|.13|.28
4..|5.2|..9
27.|46.|...
---+---+---
..5|3.8|9..
.83|...|.6.
..7|69.|.43
```

Mejorando el desempeño de solve

Hasta ahora, el método que ha implantado para resolver Sudoku no es más que una exploración por "fuerza bruta" de todo el espacio de soluciones. Habrá notado que el tiempo para resolver algunos Sudokus es considerable, y que demora aún más en determinar que un Sudoku no tiene solución.

Esto tiene que ver con la inocencia con la cual se está seleccionando el espacio en blanco a rellenar. Ud. debe mejorar el desempeño de la función `solve` escribiendo una nueva función

```
smarter :: Sudoku -> Maybe Sudoku
```

que incorpore funcionalidad adicional tal que:

- En lugar de escoger "el primer espacio disponible", encuentre la fila, columna o bloque de 3x3 que tenga **menos** espacios libres por llenar. Si hay más de una posibilidad, seleccione la primera.
- Al escoger un espacio libre entre varios de un mismo bloque, debe escoger aquel que tenga menos valores posibles por seleccionar.
- Si algún espacio libre sólo tiene **un** valor posible por asignar, la asignación de ese espacio tiene prioridad sobre cualquier otra asignación, incluso si pertenece a un bloque que tiene más celdas vacías que el resto.[2]

así, si aplicamos esas técnicas sobre el siguiente Sudoku en curso de solución

[2]Esta heurística es conocida como *propagación* y es la que emplea la mayoría de los humanos cuando resuelven Sudoku.

```
36..712..
.52...18.
.192.47..
596.1.428
4..5.2..9
27.46....
..53.89..
.83....6.
..769..43
```

su implantación debería preferir la cuarta fila [3] y además notar que para la cuarta columna solamente hay un valor posible y comenzar a llenar por allí.[4]

El orden en el cual aplicar ambas técnicas es *su* problema, sin embargo resulta obvio que si sólo queda un valor posible en una celda esa tenga prioridad, así que le conviene escribir la función

```
propagate :: Sudoku -> Sudoku
```

que dado un Sudoku cualquiera determina si hay filas, columnas o bloques de 3x3 en los cuales solamente haya un valor posible y los completa, repitiendo ese proceso hasta que el Sudoku esté completamente lleno, o hasta que no queden ocurrencias de celdas con único valor posible por bloque. Esta función deberá ser incorporada en el punto adecuado de la función `smarter`.

El programa principal

El programa principal se encargará de la interacción con el sistema operativo anfitrión. Para ello debe determinar el método de resolución a aplicar (*solve* o *smarter*) y el archivo que contiene los Sudoku a resolver. Esta información se obtiene a través de los argumentos de la línea de comando, e.g.

```
$ sudoku solve sudokus.txt
```

empleará la función `solve` desarrollada en la primera entrega, mientras que

```
$ sudoku smarter sudokus.txt
```

empleará la función `smarter` desarrollada en esta entrega. Note que se puede usar cualquier prefijo del método, esto es, se puede utilizar `solve`, `solv`, `sol` o `so` para indicar la función inocente, mientras que `smarter`, `smarte`, `smart`, `smar`, `sma` o `sm` para indicar la función inteligente; usar solamente una `s` es ambiguo y debe generar un mensaje de error apropiado.

Para cada Sudoku contenido en el archivo indicado, el programa deberá aplicar la función solicitada de resolución hasta obtener una solución o determinar que no hay solución. En caso de haber solución, el programa debe mostrar el Sudoku resuelto, y en caso de no haber solución, indicarlo con el mensaje `"No tiene solución"` (sin las comillas). En cualquier caso, inmediatamente después debe mostrar una línea que indique el tiempo de resolución en horas, minutos, segundos y fracciones de segundo, e.g. `"42 mins, 42 secs"` (sin las comillas).

[3] Porque solamente tiene dos espacios vacíos por llenar.

[4] El valor posible es el 7, pues en la sexta columna se podría poner 3 o 7. Obviamente la sexta columna se verá forzada tan pronto se asigne 7 en la cuarta.

Entrega de la Implementación

Ud. debe entregar un archivo `.tar.gz` o `.tar.bz2` (**no** puede ser `.rar`, ni `.zip`) que al expandirse genere un directorio con el nombre de su grupo (e.g. `G42`) dentro del cual encontrar **solamente**:

- El archivo `Sudoku.hs` conteniendo el código fuente Haskell para implantar el tipo de datos abstractos `Sudoku`, además de las funciones que operan sobre el. El archivo debe seguir la estructura de un módulo Haskell.
- El archivo `sudoku.hs` conteniendo el código fuente Haskell para el programa principal. Todas las operaciones de I/O deben estar contenidas en este archivo, con el necesario manejo de excepciones.
- Se evaluará el correcto estilo de programación:
  - Indentación adecuada y consistente en *cualquier* editor – la excusa "profe, en el mío se ve bien" es inaceptable.
  - El módulo solamente debe exportar los tipos de datos y funciones definidas en este enunciado. Cualquier función auxiliar debe aparecer bien sea en el contexto local de un `where` o como una función privada al módulo.
  - Todas las funciones deben tener su firma, con el tipo más general posible.
- Todas las funciones deben escribirse aprovechando constructores de orden superior evitando el uso de recursión directa.
- Puede aprovechar funciones del `Prelude` y de los módulos `Data.List`, `Data.Char`, `Data.Either`, `Data.Maybe`, `System.Time`, `System.Environment` y `System.IO`.
- Los archivos **deben** estar completa y correctamente documentados utilizando la herramienta Haddock. Ud. **no** entregará los documentos HTML generados, sino que deben poder **generarse** de manera automática incluyendo acentos y símbolos especiales. Es **inaceptable** que la documentación tenga errores ortográficos.
- El programa será compilado utilizando `hmake` o el `Makefile` suministrado por Ud. hasta construir el programa ejecutable final, el cual será invocado desde la línea de comandos como se describió anteriormente.

  Si el archivo no puede ser leído o tiene errores en alguna de las definiciones que contiene, el programa debe indicar la falla con un mensaje de error apropiado y finalizar la ejecución. Es **inaceptable** que el programa aborte sin control.
- **Fecha de Entrega.** Viernes 2011-02-11 (Semana 5) hasta las 18:00 VET
- **Valor de Evaluación.** Veinticinco (25) puntos.