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

Programación Funcional - Entrega 1
Sudoku

El Sudoku es un pasatiempo lógico originario del Japón, que ha adquirido una popularidad enorme en el mundo occidental en los últimos años. A menos que viva debajo de una roca, habrá visto algún Sudoku en periódicos o revistas de pasatiempos.

Consiste en una retícula de 9 filas y 9 columnas, algunas de cuyas celdas contienen dígitos entre 1 y 9, y el resto está en blanco. El objetivo del pasatiempo es completar la retícula de manera que en cada fila, cada columna y cada bloque de 3 por 3 celdas, aparezcan los dígitos entre 1 y 9 exactamente una vez.

Si aún así nunca has visto uno, el siguiente es un ejemplo de un Sudoku

	6		**1**		**4**		**5**	
		8	**3**		**5**	**6**		
2								**1**
8			**4**		**7**			**6**
		6				**3**		
7			**9**		**1**			**4**
5								**2**
		7	**2**		**6**	**9**		
	4		**5**		**8**		**7**	

Figura 1: Un Sudoku de ejemplo

y su solución, de manera que pueda apreciarse como se satisface la propiedad descrita anteriormente en cuanto a la ocurrencia de los dígitos en cada fila, columna y bloque de 3 por 3.

9	**6**	3	**1**	7	**4**	2	**5**	8
1	7	**8**	**3**	2	**5**	**6**	4	9
2	5	4	6	8	9	7	3	**1**
8	2	1	**4**	3	**7**	5	9	**6**
4	9	**6**	8	5	2	**3**	1	7
7	3	5	**9**	6	**1**	8	2	**4**
5	8	9	7	1	3	4	6	**2**
3	1	**7**	**2**	4	**6**	**9**	8	5
6	**4**	2	**5**	9	**8**	1	**7**	3

Figura 2: Nuestro ejemplo, resuelto

En esta etapa del proyecto Ud. va a construir un conjunto de funciones en Haskell para modelar Sudokus y resolverlo por "fuerza bruta". Por el momento, solamente nos preocuparemos por modelarlos como estructuras de datos directamente legibles en Haskell, de manera que todas las operaciones sean realizadas a través del interpretador GHCi.

Le sugiero que lea las referencias[1, 2, 3] al final de este documento, tanto para encontrar aclaratorias y estudios sobre el Sudoku, así como ejemplos de Sudokus con sus soluciones y técnicas para resolverlos.

Desarrollo de la implementación

Comenzaremos por modelar el Sudoku como una matriz de dígitos o blancos. La manera natural de modelar una matriz es como una lista de listas, en la cual la lista exterior representa las filas y cada una de las listas contiene los elementos correspondientes a las columnas. Para modelar la presencia de dígitos o blancos, utilizaremos el tipo `Maybe` de Haskell. Los dígitos serán presentados con enteros pequeños, i.e.

```
data Sudoku = Grid { rows :: [[Maybe Int]] }
```

Con esa estructura de datos, el ejemplo quedaría representado como

Algorithm 1 El ejemplo, modelado con el tipo `Sudoku`

```
ex1 :: Sudoku
ex1 = Grid
  [
    [ Nothing, Just 6,  Nothing, Just 1,  Nothing,
      Just 4,  Nothing, Just 5,  Nothing ],
    [ Nothing, Nothing, Just 8,  Just 3,  Nothing,
      Just 5,  Just 6,  Nothing, Nothing ],
    [ Just 2,  Nothing, Nothing, Nothing, Nothing,
      Nothing, Nohting, Nothing, Just 1  ],
    [ Just 8,  Nothing, Nothing, Just 4,  Nothing,
      Just 7,  Nothing, Nothing, Just 6  ],
    [ Nothing, Nothing, Just 6,  Nothing, Nothing,
      Nothing, Just 3,  Nothing, Nothing ],
    [ Just 7,  Nothing, Nothing, Just 9,  Nothing,
      Just 1,  Nothing, Nothing, Just 4  ],
    [ Just 5,  Nothing, Nothing, Nothing, Nothing,
      Nothing, Nothing, Nothing, Just 2  ],
    [ Nothing, Nothing, Just 7,  Just 2,  Nothing,
      Just 6,  Just 9,  Nothing, Nothing ],
    [ Nothing, Just 4,  Nothing, Just 5,  Nothing,
      Just 8,  Nothing, Just 7,  Nothing ]
  ]
```

Así, harán falta una serie de funciones de apoyo sobre las cuales desarrollar nuestra función que resuelve el Sudoku por "fuerza bruta".

Funciones de apoyo

Para comenzar, es necesario contar con algunas funciones básicas de manipulación que nos permitan construir y verificar la estructura de los Sudoku.

1. Implante la función

   ```
   emptySudoku :: Sudoku
   ```

   que genera un Sudoku compuesto exclusivamente por celdas vacías. Note que **no** puede escribir la lista explícitamente, sino que debe generarla con funciones.

2. El tipo de datos `Sudoku` es demasiado general y permite construir cosas que no cumplan con la estructura de un Sudoku, así que es necesario implantar la función

   ```
   isValid :: Sudoku -> Bool
   ```

   que verifica la estructura del Sudoku tanto en forma como composición, produciendo el valor booleano según el caso.

3. Es conveniente tener una función que nos indique si hemos terminado de llenar la retícula del Sudoku o no. Note que solamente queremos saber si ya está completamente lleno; determinar su validez es un problema diferente. Así, implante la función

   ```
   isFull :: Sudoku -> Bool
   ```

   que produzca el valor booleano adecuado según la retícula esté llena o no.

4. Aplicando la estrategia *bottom-up*, típica de la programación funcional, vamos a descomponer el problema de determinar si un Sudoku está "resuelto" o no en varias funciones. Suponiendo que tenemos un Sudoku lleno, deben cumplirse tres condiciones:

   - Ninguna fila ha de contener dígitos repetidos.
   - Ninguna columna ha de contener dígitos repetidos.
   - Ningún recuadro de 3x3 ha de contener dígitos repetidos.

   A efectos prácticos, cualquiera de estos tres conjuntos de dígitos puede ser considerado como un "bloque" sobre el cual verificar las restricciones, así que definiremos un tipo de datos auxiliar

   ```
   type Block = [Maybe Int]
   ```

   y ahora es necesario que Ud.:

   a) Implante la función

   ```
   blockIsOk :: Block -> Bool
   ```

   que produce el valor booleano adecuado si el bloque cumple con la restricción de dígitos contenidos en él.

   b) Implante la función

   ```
   blocks :: Sudoku -> [Block]
   ```

que genere una lista con todos los bloques que pueden extraerse de un Sudoku particular. Esto es, la lista resultante debe tener las 9 filas, las 9 columnas y los 9 recuadros de 3x3.

c) Implante la función

```
isOk :: Sudoku -> Bool
```

que dado un Sudoku cualquiera produzca el valor booleano adecuado según éste cumpla con las restricciones de dígitos contenidos en todos los bloques que le constituyen.

¿Como resolver el Sudoku?

Para este punto, debe resultar natural que la conjunción de `isValid`, `isFull` e `isOk`, indica cuando una retícula Sudoku cumple con la restricción de forma, está completamente llena y además cumple con las restricciones de contenido.

Ahora, nuestro problema se centra en llenar la retícula. Para ello necesitaremos una serie de funciones para encontrar y manipular las celdas vacías, y eso resultará más natural si representamos la posición de cada celda con un tipo de datos

```
type Position = (Int,Int)
```

que indique la fila y la columna a la que hacemos referencia, i.e. `(4,2)` corresponde a la segunda columna de la cuarta fila.

Con esta representación, es necesario que Ud.

1. Implante la función

   ```
   blanks :: Sudoku -> [Position]
   ```

   que dado un Sudoku cualquiera produzca la lista de posiciones que aún están vacías. El orden en el cual aparezcan las posiciones en la lista es absolutamente arbitrario y depende de su astucia en cuanto a la estrategia de resolución de Sudokus.

2. Implante la función

   ```
   (!!=) :: [a] -> (Int,a) -> [a]
   ```

   que dada una lista cualquiera y una tupla indicando una posición en la lista y un nuevo valor, produzca una nueva lista idéntica a la anterior, pero sustituyendo el nuevo valor en la posición indicada por la tupla.

3. Implante la función

   ```
   update :: Sudoku -> Position -> Maybe Int -> Sudoku
   ```

   que dado un Sudoku cualquiera, una posición y el valor para la celda, produzca un nuevo Sudoku con el valor en la posición indicada.

4. Implante la función

   ```
   possibleValues :: Sudoku -> Position -> [Int]
   ```

que dado un Sudoku cualquiera y una posición, produzca la lista de valores numéricos plausibles para esa posición. Esto es, la función debe analizar la posición y determinar, según la fila, columna y recuadro al que pertenece, cuales valores podrían colocarse en ella.

Contando con toda la maquinaria necesaria, podemos resolver el Sudoku por "fuerza bruta", con un algoritmo muy sencillo el cual estará implantado en la función

```
solve :: Sudoku -> Maybe Sudoku
```

La función debe determinar si el Sudoku suministrado cumple con las condiciones de estructura y restricciones de un Sudoku, y en caso negativo retornar `Nothing`. En caso que la estructura y restricciones se cumplan, hemos de operar de manera recursiva:

- Si el Sudoku está completo y se verifican todas las restricciones, ya lo hemos resuelto.
- En caso contrario, existe al menos una celda vacía sobre la cual probar recursivamente cada uno de los valores posibles que puedan asignarse. El primer intento recursivo que produzca un resultado diferente a `Nothing`, corresponde a la solución; si todos los intentos recursivos producen `Nothing`, entonces la invocación recursiva falla hacia su llamada anterior.

Esto es, "fuerza bruta" es un eufemismo por *backtracking[4]* que es una estrategia básica de exploración de espacio de soluciones.

Por último, resultará conveniente la existencia de una función

```
isSolutionOf :: Sudoku -> Sudoku -> Bool
```

tal que dados dos Sudoku, produzca el valor booleano correspondiente si el segundo Sudoku es solución del primer Sudoku.

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 `Sudoku` y las funciones que operan sobre el. El archivo debe seguir la estructura de un módulo Haskell.
- Se evaluará el correcto estilo de programación:
  - Indentación adecuada y consistente en *cualquier* editor – la excusa "profe, pero 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` o `Data.List` que resulten convenientes.
- El archivo **debe** estar completa y correctamente documentado 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.
- **Fecha de Entrega.** Viernes 2011-01-28 (Semana 3) hasta las 18:00 VET
- **Valor de Evaluación.** Diez (10) puntos.

Referencias

[1] Sudoku según WikiPedia
http://en.wikipedia.org/wiki/Sudoku

[2] The Daily Sudoku
http://www.dailysudoku.com/sudoku/index.shtml

[3] Montones de Sudoku para resolver
http://www.sudoku.com

[4] Backtracking según WikiPedia
http://en.wikipedia.org/wiki/Backtracking