Teoría de los Lenguajes de Programación Práctica curso 2022-2023

Enunciado

Fernando López Ostenero y Ana García Serrano

1. '*El cliente siempre sabe lo que quiere*'.

La empresa "Tecleamos Los Programas" (TLP) siempre vela por el bienestar de sus clientes, ofreciéndoles productos de calidad basados en sus necesidades. Recientemente, uno de estos clientes les ha encargado un programa que controle el stock de productos del catálogo de su tienda, así que se han puesto manos a la obra para llevarlo a cabo.

El cliente ha sido muy específico en la descripción del funcionamiento del programa y ha pedido poder tener cuanto antes un *prototipo funcional* del mismo. Como el lema de la empresa es "*El cliente siempre sabe lo que quiere*", han decidido crear exactamente eso: un prototipo del programa de control de stocks escrito en un lenguaje de programación funcional.

El problema es que en la empresa no hay programadores que sepan trabajar bajo el paradigma de programación funcional, por eso le ofrecen un contrato para la programación del prototipo funcional en Haskell debido a su experiencia con los lenguajes funcionales.

¿Dejará pasar esta oportunidad única de trabajo o demostrará su conocimiento de Haskell a la empresa TLP?

2. Enunciado de la práctica

La práctica consiste en elaborar un programa en **Haskell** que permita al usuario almacenar el stock de los productos de un catálogo, incluyendo operaciones de modificación de stock (debido a órdenes de compra o venta), consulta de stock y listado de productos en stock.

Consideraremos que los productos se definen mediante cadenas de caracteres y que asociado a cada producto existirá un valor entero, siempre mayor o igual a cero, que indicará el número de unidades de ese producto de que dispone la tienda. Es decir, la estructura de datos a utilizar va a comportarse de manera similar a una estructura indexada por cadenas de caracteres. Por ello, para ilustrar los ejemplos utilizaremos una notación de estructura indexada como la siguiente:

```
s["vaso"]==10
```

que significará que en el stock `s`, el número de unidades del producto `"vaso"` es 10.

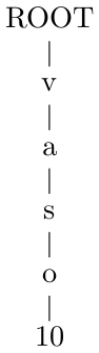
Se implementarán cuatro operaciones para trabajar con esta estructura:

1. **`createStock`**: que devolverá un stock vacío.
2. **`updateStock`**: recibirá un stock `s`, un producto `p` y un número de unidades `u` y devolverá un stock resultado de insertar en `s` el producto `p` con un número de unidades `u`. Si antes de la inserción ya hubiera en `s` un número de unidades del producto `p`, entonces el valor anterior se perdería, siendo substituido por `u`.
3. **`retrieveStock`**: que recibirá un stock `s` y un producto `p` y devolverá el número de unidades del producto `p` que estén almacenadas en el stock `s`. Si el producto no estuviese almacenado en el stock `s`, devolverá un valor `-1` (ver apartado de implementación).
4. **`listStock`**: que recibirá un stock `s` y una cadena de caracteres `p` y devolverá la lista de pares `<pr,u>` tales que `s["pr"]==u` para todos los productos `pr` almacenados en `s` que comiencen por la cadena `p`. Se devolverá, además, ordenada alfabéticamente por los nombres de los productos.

Realizaremos la implementación de la estructura utilizando un Trie, que consiste en un árbol general con las siguientes características:

1. El nodo raíz no contiene ningún valor.
2. Todos los nodos intermedios contienen un carácter.
3. Cada nodo hoja contiene un elemento del tipo almacenado en el Trie, que se asocia al índice formado por la cadena de caracteres que se obtiene siguiendo el camino desde la raíz hasta dicho nodo hoja.

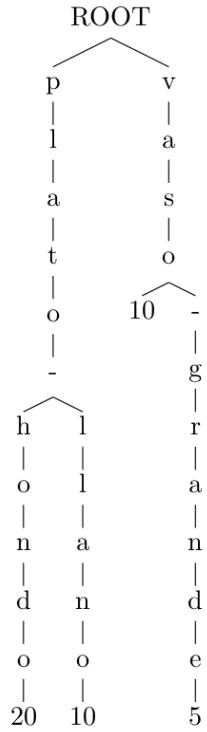
Así, en nuestro ejemplo anterior, una representación visual de la estructura utilizada para almacenar `s["vaso"] = 10` sería la siguiente:



Si ahora añadimos los siguientes valores al stock:

```
s["vaso grande"] = 5
s["plato llano"] = 10
s["plato hondo"] = 20
```

la representación gráfica de la estructura sería:



donde los espacios en blanco se han sustituido por guiones para mostrar el lugar que ocupan .

2.1 Ejemplo de funcionamiento

A continuación vamos a dar un ejemplo del funcionamiento de la práctica.

```
ghci> main

Menú
----
1-Comprar productos
2-Vender productos
3-Consultar stock
4-Listado del stock
0-Fin del programa

Seleccione opción:
```

La función main inicia la ejecución del programa. Se carga el estado del stock de un fichero (el cual supondremos que contiene el stock de nuestro ejemplo anterior) y se muestra un menú con las opciones disponibles:

1. La opción “Comprar productos” nos preguntará por el nombre de un producto (distinto de la cadena vacía) y un valor numérico (un valor entero mayor que cero) que será el número de unidades que se adquieren. El stock se modifica sumando las unidades compradas a las ya existentes.

   ```
   Seleccione opción: 1

   Compra de producto
   ------------------

   Introduzca el nombre del producto: vaso grande
   Introduzca el número de unidades: 10
   ```

2. La opción “Vender productos” nos preguntará por el nombre de un producto (distinto de la cadena vacía) y un valor numérico (un valor entero mayor que cero) que será el número de unidades que se van a vender. Si este número de unidades no es mayor que el de unidades existentes en el catálogo, se realizará la operación, y se actualizará el stock restando las unidades existentes menos las vendidas.

   ```
   Seleccione opción: 2

   Venta de producto
   -----------------

   Introduzca el nombre del producto: vaso grande
   Introduzca el número de unidades: 5
   ```

   En caso contrario, no se realizará la operación de venta y no se modificará el stock.

   ```
   Introduzca el nombre del producto: vaso grande
   Introduzca el número de unidades: 100
   No hay tantas unidades para vender.
   ```

3. La opción “Consultar stock” nos preguntará por el nombre de un producto (distinto de la

cadena vacía) y devolverá el número de unidades almacenadas de ese producto.

```
Seleccione opción: 3

Consulta de stock
-----------------

Introduzca el nombre del producto: vaso grande
El producto "vaso grande" está en nuestro catálogo y tenemos 10
unidades.
```

Si el producto nunca hubiera estado en el catálogo de la tienda, se indicará con un mensaje.

```
Introduzca el nombre del producto: vaso pequeño
El producto "vaso pequeño" nunca ha estado en nuestro catálogo.
```

4. La opción "Listado del stock" nos preguntará por un prefijo de búsqueda (que puede ser la cadena vacía) y devolverá la lista de todos los productos almacenados que comiencen por dicho prefijo y el número de unidades existentes para cada uno de ellos.

```
Listado de stock
----------------

Introduzca el prefijo de la búsqueda: plato

Listado del catálogo comenzando por "plato":

plato hondo: 20
plato llano: 10
```

5. La opción "Fin del programa" finalizará la ejecución del programa tras guardar el estado del stock en un fichero para ser cargado con posterioridad.

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 `StockControl`: dentro de este módulo se programarán las funciones indicadas para poder operar con los stocks. Incluye las definiciones de todos los tipos de datos utilizados en la práctica, además de una función que implementa el esquema de bactracking que se empleará para la función `listStock`.
2 Módulo `Main`: este módulo contiene la función principal y las encargadas de interactuar con el usuario y cargar los ficheros de test. 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 `StockControl`) tres módulos adicionales para poder trabajar más cómodamente con la entrada/salida, convertir cadenas a enteros y poder acceder a ficheros.

2.3.1 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:

- `fileName`: define el nombre del fichero donde se almacena el stock.
- `loadStock`: carga el stock en memoria (ver nota al final de este apartado).
- `saveStock`: guarda el stock en el fichero indicado.
- `errorCadenaVacia`: muestra que el usuario ha introducido una cadena vacía cuando no debía y vuelve al "*bucle principal*" del programa.
- `productoDesconocido`: construye el mensaje que se muestra cuando se busca un producto desconocido.
- `inputString`: lee del teclado una cadena de caracteres y la devuelve convertida a minúsculas.
- `inputInt`: lee del teclado una cadena de caracteres y devuelve un entero (si la cadena de caracteres contenía un entero) o un valor Nothing (si la cadena de caracteres no contenía un entero).
- `inputData`: lee del teclado el producto y el número de unidades para las operaciones de compra y venta, comprobando que lo introducido por el usuario cumpla las restricciones indicadas.
- `compra`: ejecuta una operación de compra de productos.
- `venta`: ejecuta una operación de venta de productos.
- `consulta`: realiza una consulta al stock.
- `mainLoop`: esta función es el "*bucle principal*" del programa. Muestra el menú al usuario, espera que se elija una opción y realiza la operación del menú indicada.
- `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 `mainLoop`.

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 `StockControl`.

Nota: versión de Haskell para ejecutar la práctica

La función `loadStock` hace uso de la función `readFile'` para la lectura del fichero de stock mediante evaluación estricta y sólo es compatible con versiones recientes de Haskell (posteriores a enero de 2021).

Junto a los dos módulos de la práctica se incluye el fichero `Test_readFile.hs` que permite comprobar si dicha función es compatible con la versión de Haskell que tenemos instalada. En caso de que no fuese compatible, sería necesario modificar la función `loadStock` dentro del fichero `Main.hs` como sigue:

1. En la línea 17 se elimina la comilla tras `readFile'` para utilizar la función `readFile`, que realiza el mismo trabajo, pero mediante evaluación perezosa.
2. Cambiamos la línea 21 (respetando la indentación previa) por el siguiente código:

```
[s] -> mainLoop (read s::Stock)
```

Estos cambios permiten que la práctica se ejecute sin necesidad de la función `readFile'`, pero a

cambio pueden provocar un error de entrada/salida si el fichero donde se graba el stock tiene más de una línea, debido a la forma de evaluación de la función `readFile`.

2.3.2 Implementación (módulo StockControl)

En este apartado vamos a mostrar las cuatro funciones que deberán ser programadas para gestionar los stocks. En primer lugar, veamos los tipos de datos (ya programados):

```
data Stock = ROOTNODE [Stock] |
             INNERNODE Char [Stock] |
             INFONODE Int
  deriving (Show,Read,Eq)
```

El tipo `Stock` se implementa como un trie, que es un árbol que puede ser:

- Un nodo `ROOTNODE`, con una lista de hijos, que son, a su vez de tipo `Stock`.
- Un nodo `INNERNODE`, que contiene un carácter y tiene una lista de hijos también de tipo `Stock`.
- Un nodo `INFONODE`, que contiene un valor entero.

Para que el programa pueda realizar su trabajo es necesario programar las siguientes cuatro funciones:

1. **`createStock :: Stock`**: que deberá devolver un stock vacío
2. **`retrieveStock :: Stock -> String -> Int`**: que recibe un `Stock s`, un `String p` (no vacío) y devuelve un `Int` con el número de productos `p` que están almacenados en `s`. Si en `s` no hubiera información sobre el producto `p`, devolverá el valor entero `-1`.
3. **`updateStock :: Stock -> String -> Int -> Stock`**: que recibe un `Stock s`, un `String p` (no vacío) y un `Int u` (mayor que cero) y devuelve un `Stock` en el que el valor asociado a `p` es `u`.
4. **`listStock :: Stock -> String -> [(String,Int)]`**: un `Stock s`, un `String p` (que puede ser vacío) y devuelve una lista de tuplas `(String,Int)` que cumple:
    a) Para toda tupla `(pr,u)`, se cumple que `s[pr]==u`, es decir, el valor asociado a `p` en `s` es `u`.
    b) Contiene todos los productos `pr` de `s` que comienzan por la cadena `p`.
    c) Está ordenada según el orden alfabético de los productos.

Recomendamos que la última función se programe utilizando un backtracking, para lo que se incluye la función `bt`:

```
bt :: (a -> Bool) -> (a -> [a]) -> a -> [a]
bt    eS             c             n
  | eS n      = [n]
  | otherwise = concat (map (bt eS c) (c n))
```

que implementa un esquema genérico de backtracking. Esta función recibe una función `eS` que determina si un nodo es una solución a nuestro problema, una función `c` que devuelve los hijos válidos de un nodo y el nodo actual de exploración `n`. Devuelve una lista de todos los nodos solución a nuestro problema a los que se llega desde el nodo `n`.

Por lo tanto, programar la función `listStock` va a implicar:

a) Localizar el punto de la estructura al que se llega siguiendo el prefijo `p`.

b) Definir el problema de localizar todos los productos que comiencen por el prefijo `p` en forma de exploración en un grafo y programar las dos funciones que requiere el esquema de backtracking para resolver el problema definido, a partir del punto localizado en el paso anterior.

2.4 Módulo ViewStock

Este módulo adicional se ha incluido como ayuda para poder visualizar gráficamente el aspecto del `Trie` que contiene el stock. El módulo contiene la definición del tipo `Stock` del módulo `StockControl`, por lo que no necesita importarlo y funcionará aunque no se haya implementado completamente la práctica.

Al ejecutar la función `main`, el programa carga el fichero de stock con el que trabaja el módulo `Main` interactivo y lo traduce a un fichero `LaTeX` (llamado "`stock.tex`"), el cual puede ser compilado a PDF (entre otros formatos) con cualquier herramienta que permita trabajar con `LaTeX`.

3. Cuestiones sobre la práctica

La respuesta a estas preguntas es optativa. Sin embargo, si no se 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 (usando la misma estructura aquí presentada) 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**? Justifique sus respuestas.

2 (*1'5 puntos*). Indique, con sus palabras, cómo afecta el predicado predefinico no lógico corte (`!`) al modelo de computación de **Prolog**. ¿Cómo se realizaría este efecto en **Haskell**? ¿Considera que sería necesario el uso del corte en una implementación en **Prolog** de esta práctica? Justifique sus respuestas.

3 (*1 punto*). Indique qué clases de constructores de tipos (ver capítulo 5 del libro de la asignatura) se han utilizado para definir el tipo `Stock` y para definir los nodos del backtracking utilizado en la función `listStock`. Justifique sus respuestas.

4. Documentación a entregar

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

- Código fuente en **Haskell** que resuelva el problema planteado. Para ello se deberá entregar el ficheros `StockControl.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.