Teoría de los Lenguajes de Programación
Práctica curso 2019-2020

Enunciado

Fernando López Ostenero y Ana García Serrano

1. Introducción: el problema de las letras.

El conocido concurso "*Cifras y Letras*" propone a sus participantes dos tipos de pruebas: una prueba de cifras en la que deben aproximarse lo más posible a un número objetivo utilizando operaciones básicas sobre una serie de números y una prueba de letras, en la que deben encontrar la palabra (válida) más larga que sea posible escribir con una serie de letras.

En esta práctica vamos a crear un programa que resolverá la segunda de esas pruebas. Concretamente el programa recibirá una serie de letras y nos devolverá todas las palabras válidas que se puedan construir con ellas. Para comprobar si una palabra es válida o no, es imprescindible el uso de un diccionario.

Para ello, se proporcionará un fichero con el diccionario en forma de lista de 79517 palabras pertenecientes al Diccionario de la RAE para que los estudiantes puedan probar su programa. Para evitar problemas debidos a la codificación de caracteres, se han borrado todas las palabras que contienen la ñ y se han eliminado los acentos gráficos.

Así pues, la práctica se dividirá en dos fases diferentes:

1. **Creación del diccionario**: se cargará un fichero de texto que contiene las palabras que vamos a considerar válidas (una palabra por línea) y se insertará cada una de ellas en una estructura de diccionario sobre la cual se consultarán las posibles palabras.
2. **Búsqueda de palabras**: una vez se disponga del diccionario, el programa preguntará al usuario la secuencia de letras a analizar y devolverá todas las posibles palabras válidas que se puedan construir a partir de esas letras, ordenadas alfabéticamente y agrupadas por tamaño de mayor a menor.

Comoquiera 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 diccionario 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 dichas funciones.

2. Enunciado de la práctica

La práctica consiste en elaborar un programa en **Haskell** que cargue un fichero de texto con palabras válidas y cree un diccionario. Una vez realizado esto, el programa preguntará al usuario por diferentes secuencias de letras y devolverá las palabras válidas (aquellas presentes en el diccionario) que se puedan construir en base a las letras disponibles.

Ejemplo 1: el programa busca el fichero del diccionario por palabras (`diccionario.txt`), construye el diccionario y espera que el usuario introduzca una secuencia de letras:

```
*Main> main
Cargando lista de palabras desde el fichero diccionario.txt
79517 palabras leídas

Introduce secuencia de letras:
```

Ejemplo 2: con el diccionario anterior ya cargado, el usuario introduce la secuencia de letras "`aosc`" y el programa devuelve todas las palabras del diccionario que pueden construirse utilizando esas letras:

```
Introduce secuencia de letras: aosc
-Palabras de 4 letras: asco, caos, caso, cosa, saco, soca
-Palabras de 3 letras: cao, cas, coa, oca, osa, sao
-Palabras de 2 letras: as, ca, oc, os, so
-Palabras de 1 letra: a, c, o, s

Introduce secuencia de letras:
```

Ejemplo 3: con el diccionario anterior ya cargado, el usuario introduce la secuencia de letras "`riomwalfo`" y el programa devuelve todas las palabras del diccionario que pueden construirse utilizando esas letras:

```
Introduce secuencia de letras: riomwalfo
-Palabras de 9 letras: wolframio
-Palabras de 7 letras: wolfram
-Palabras de 6 letras: almori, amorfo, amorio, ariolo, filmar, firmal, foliar,
formal, formol, marfil, mariol, moflir
-Palabras de 5 letras: afilo, aforo, alimo, arfil, arilo, aromo, farol, filar,
firma, flora, folia, folio, foral, forma, limar, maori, marlo, mirla, mirlo,
mofar, molar, moral, morfa, oriol, ramio
-Palabras de 4 letras: afro, alim, almo, amir, amol, amor, ario, arlo, falo,
faro, fiar, fila, film, filo, fimo, flor, foro, fria, frio, liar, lima, limo,
lira, loar, loma, lomo, loor, lora, loro, malo, maro, miar, mira, mofa, mola,
molo, mora, moro, olio, olma, olmo, olor, oral, orfo, orla, orlo, rail, ralo,
ramo, rial, rifa, rima, rola, rolo, roma, romi, romo
-Palabras de 3 letras: ali, ami, amo, aro, far, fia, fil, ira, lar, lia, lio,
loa, mal, mar, mia, mil, mio, moa, mol, mor, oil, oir, ola, ora, ori, oro, ria,
rio, roa, rol
-Palabras de 2 letras: al, am, ar, fa, fi, fo, io, ir, la, lo, mi, ro
-Palabras de 1 letra: a, f, i, l, m, o, r, w

Introduce secuencia de letras:
```

En este ejemplo vemos que con las letras de la secuencia introducida no se puede formar ninguna palabra de 8 letras.

2.1 Estructura del diccionario

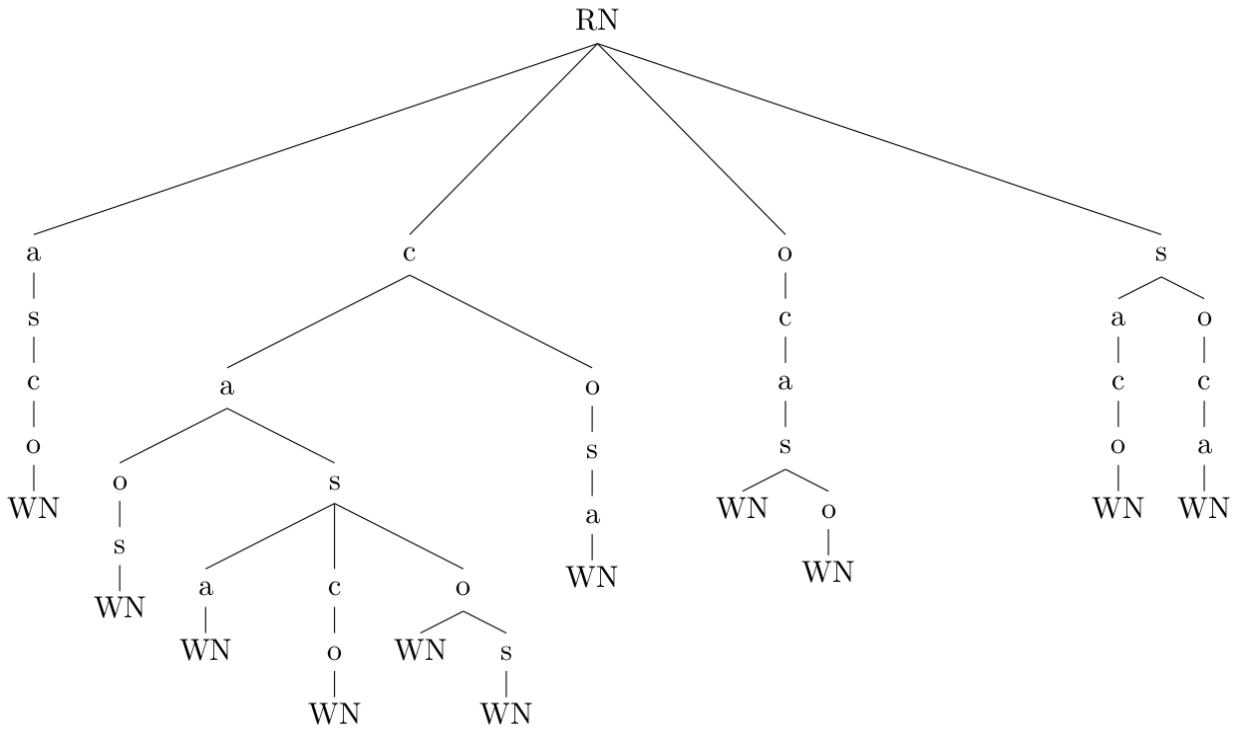
En este apartado vamos a explicar cuál será la estructura que va a tener nuestro diccionario, lo que va a condicionar cómo se han de introducir en él las palabras leídas.

La idea va a ser utilizar un árbol de caracteres en el que cada nodo intermedio (sin incluir la raíz, que es un caso especial) va a contener un carácter y los nodos hoja van a darnos información acerca de las palabras que contiene el diccionario, de forma que la concatenación de los caracteres que vayamos encontrando en el camino desde la raíz hasta un nodo hoja será una palabra del diccionario.

A continuación vamos a ver un ejemplo gráfico de un pequeño diccionario (diferente al utilizado en los ejemplos anteriores) construido a partir de la siguiente lista de palabras:

```
["casa","caso","casco","caos","saco",
"casos","cosa","ocas","asco","soca","ocaso"]
```

Figura 1: Diccionario representado en forma de árbol



Aquí podemos ver el diccionario resultante representado gráficamente en forma de árbol. Como se indicó previamente, los nodos intermedios (salvo la raíz, representada como RN) son los únicos que contienen caracteres. Y la concatenación de los caracteres que forman el camino desde la raíz a cualquiera de los nodos hoja (representados como WN) nos indica las palabras presentes en el diccionario.

Para una mejor comprensión de la estructura de datos a utilizar, sugerimos a los estudiantes que inviertan un tiempo recorriendo el diccionario mostrado en la ***Figura 1*** para comprobar que contiene exactamente las once palabras indicadas en la lista que se ha utilizado para crearlo.

2.2 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 `Dictionary`: dentro de este módulo se programarán las funciones de construcción y acceso al diccionario. Incluye, además, la definición del tipo de datos `Dictionary`.
2. Módulo `Main`: este módulo contiene el programa principal y todas las funciones encargadas de cargar la lista de palabras e interactuar con el usuario.

Dado que un estudio más profundo de los módulos en Haskell está fuera del ámbito de la asignatura, sólo indicaremos que cada módulo importa una serie de módulos que son necesarios para su funcionamiento. En nuestra práctica el módulo `Main` importa (además del módulo `Dictionary`) módulos adicionales para poder trabajar con la salida, ficheros y convertir a minúsculas todas las letras. Además, la lista de identificadores entre paréntesis que acompaña al nombre del módulo `Dictionary` representa las definiciones de tipos y funciones que el módulo exporta. Cualquier otra definición se considerará privada, siendo sólo accesible por el módulo.

2.3 Tipo de dato Dictionary (módulo Dictionary)

En este apartado vamos a introducir el tipo de datos que vamos a utilizar para almacenar un diccionario en la práctica. Está definido en la parte del módulo `Dictionary` que proporciona el Equipo Docente. Tal y como hemos visto en el apartado anterior, un diccionario será un árbol en el que los nodos intermedios contendrán un carácter, mientras que los nodos hoja nos darán la información de qué palabras pertenecen al diccionario. Así pues, la definición del tipo `Dictionary` será la siguiente:

```
data Dictionary = ROOTNODE [Dictionary] |
                  LETTERNODE Char [Dictionary] |
                  WORDNODE
```

es decir, un diccionario es o bien un `ROOTNODE` (nodo raíz) que contiene una lista de diccionarios, un `LETTERNODE` (nodo letra) que contiene un `Char` y una lista de diccionarios, o bien un `WORDNODE` (nodo palabra) que será un nodo hoja que nos marcará que la concatenación de todas las letras contenidas en los nodos desde la raíz (que no contiene ninguno) hasta el padre del nodo hoja forman una palabra válida del diccionario.

Este tipo de datos se añade como instancia de la clase `Show`, para poder visualizar el contenido de un elemento de tipo `Dictionary` y así facilitar la tarea de programación de las funciones de creación del diccionario.

La representación interna del diccionario de la *Figura 1* sería la siguiente:

```
ROOTNODE [LETTERNODE 'a' [LETTERNODE 's' [LETTERNODE 'c' [LETTERNODE 'o'
[WORDNODE]]]],LETTERNODE 'c' [LETTERNODE 'a' [LETTERNODE 'o' [LETTERNODE 's'
[WORDNODE]],LETTERNODE 's' [LETTERNODE 'a' [WORDNODE],LETTERNODE 'c' [LETTERNODE
'o' [WORDNODE]],LETTERNODE 'o' [WORDNODE,LETTERNODE 's' [WORDNODE]]]],LETTERNODE
'o' [LETTERNODE 's' [LETTERNODE 'a' [WORDNODE]]]],LETTERNODE 'o' [LETTERNODE 'c'
[LETTERNODE 'a' [LETTERNODE 's' [WORDNODE,LETTERNODE 'o'
[WORDNODE]]]]],LETTERNODE 's' [LETTERNODE 'a' [LETTERNODE 'c' [LETTERNODE 'o'
[WORDNODE]]],LETTERNODE 'o' [LETTERNODE 'c' [LETTERNODE 'a' [WORDNODE]]]]]
```

la cual es un poco más complicada de leer que su representación gráfica.

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 demasiado detalle sobre el funcionamiento de las mónadas, vamos a explicar los tipos de datos definidos y qué hace cada función.

Tipos de Datos

- `WordsN`: representa una lista de palabras. Utilizaremos este tipo para referirnos a una lista de palabras todas de la misma longitud y ordenadas alfabéticamente.
- `Words`: representa una lista de `WordsN`. Utilizaremos este tipo para referirnos a una lista de palabras agrupadas en listas de palabras de igual longitud y ordenadas alfabéticamente.

Funciones

- `splitWords`: esta función recibe una lista de palabras devuelta por una consulta del diccionario y se encarga de separarla en una lista de listas de palabras, agrupándolas por tamaño. Es decir, esta función nos devuelve un elemento de tipo Words.
- `showWords`: esta función nos construye una cadena de texto para mostrar el contenido de un elemento de tipo `Words`. Para ello llama a la siguiente función.
- `showWordsN`: esta función nos construye una cadena de texto para mostrar el contenido de un elemento de tipo `WordsN`.
- `deNletras:` esta función recibe una longitud x y construye la cadena de caracteres `" de x letras"` (y si `x` es igual a `1` sería `" de 1 letra"`)
- `createDict`: carga el fichero de palabras en formato texto y crea el diccionario llamando a la función `buildDict` (que deberá ser programada).
- `mainLoop`: esta función es el "*bucle principal*" de la segunda fase. Pregunta al usuario una secuencia de letras y muestra todas las palabras válidas que se pueden formar con ellas, agrupadas por tamaño y en orden alfabético. Para ello llama a la función `search` (que deberá ser programada) del diccionario. Cuando el usuario introduce una secuencia vacía, finaliza la ejecución del programa.
- `main`: es la función principal. Llama a la función `createDict` para cargar la lista de palabras y crear el diccionario, para a continuación llamar a `mainLoop`.

Además, también está la función constante `textFile` que devuelve el nombre del fichero con la lista de palabras en texto plano.

Como ya se ha indicado, este módulo se entrega ya programado por el equipo docente. Para su correcto funcionamiento es necesario programar una serie de funciones en el módulo `Dictionary`.

2.5 Construcción del diccionario (módulo Dictionary)

En este apartado vamos a tratar sobre la construcción del diccionario. El objetivo final será programar la función `buildDict`, que es llamada por `createDict` tras cargar el fichero de palabras en formato texto. En primer lugar, vamos a ver cuál ha de ser el tipo de la función:

```
buildDict :: [String] -> Dictionary
```

es decir, la función recibe una lista de palabras (que son cadenas de caracteres) y devuelve un valor de tipo `Dictionary` (ya en forma de árbol).

Para ello serán necesarias algunas funciones auxiliares (que el módulo no va a exportar, por lo que no serán accesibles desde el módulo principal). A continuación describiremos las más importantes, lo cual no significa que no se puedan programar más en caso de necesitarlas.

Función insert

Cuyo tipo sería `insert :: [String] -> Dictionary -> Dictionary`. Esta función será llamada por `buildDict` y recibe la lista de palabras y un diccionario con una serie de palabras ya introducidas (este parámetro actuará, pues, como un parámetro acumulador). El cometido de esta función será ir introduciendo cada palabra de la lista en el diccionario que recibe como segundo parámetro.

Función insertInTree

Su tipo sería `insertInTree :: String -> Dictionary -> Dictionary`. Esta función es la que se encarga de insertar una palabra en el diccionario pero ya pensado como un árbol. El primer parámetro debería ser el resto de palabra que aún queda por insertar y el segundo parámetro el nodo del árbol a partir del cuál se debería insertar ese resto de palabra.

Función insertChild

De tipo `insertChild :: Char -> [Char] -> [Dictionary] -> [Dictionary]`. En el primer parámetro recibe el primer carácter del resto de palabra a insertar, el segundo parámetro el resto de la palabra a insertar (sin ese primer carácter), el tercer parámetro una lista de elementos de tipo `Dictionary` (que serán los hijos de un nodo). Devuelve la lista de hijos del nodo correctamente modificada.

Esta función se encarga de buscar el hijo del nodo actual donde se ha de continuar la inserción de los caracteres que aún quedan de la palabra actual. En caso de que dicho hijo no exista, la función lo deberá crear. Una vez localizado (o creado) el hijo correcto, la función deberá proseguir la inserción del resto de caracteres de la palabra a partir de ese hijo. En este punto sugerimos a los estudiantes crear “a mano” el diccionario representado en la ***Figura 1*** para una mejor comprensión del proceso de inserción de palabras en el diccionario.

2.6 Consultas al diccionario (módulo Diccionario)

En este apartado trataremos sobre las consultas al diccionario. Nuestro objetivo será programar la función `search` que recibe una secuencia de caracteres y un diccionario. Nos devuelve la lista de palabras, ordenadas alfabéticamente, que están contenidas en el diccionario y que se pueden construir con los caracteres de la secuencia.

```
search :: String -> Dictionary -> [String]
```

Por ejemplo, si buscamos la secuencia “`cascao`” sobre el diccionario representado en la ***Figura 1***, deberíamos obtener la siguiente lista de palabras:

```
["asco","caos","casa","casco","caso","cosa","ocas","saco","soca"]
```

Nótese que “`casos`” y “`ocaso`”, presentes en el diccionario, no pueden formarse con las letras de la secuencia, ya que ésta sólo contiene una ‘`s`’ (y “`casos`” tiene dos) y una ‘`o`’ (y “`ocaso`” tiene dos).

¿Cómo vamos a realizar la búsqueda? Vamos a suponer que ya hemos hecho un recorrido previo que nos ha llevado a un nodo del árbol (que no es una hoja). Por lo tanto, tendremos una secuencia de caracteres aún no utilizados y el comienzo de una palabra formado por los caracteres presentes en el camino desde la raíz al nodo actual.

Ahora analizamos **todos** los hijos del nodo actual:

- Si un hijo es un `WORDNODE`, significará que la palabra que ya tenemos formada pertenece al diccionario, con lo cual, habrá que añadirla.
- Si un hijo es un `LETTERNODE`, contendrá un carácter. Únicamente si dicho carácter pertenece a la secuencia aún no explorada (en cualquier posición, no necesariamente el primero), deberemos explorar recursivamente este hijo, añadiendo a la palabra ya formada el carácter del `LETTERNODE` y eliminando dicho carácter de la secuencia aún no explorada.

Nuevamente volvemos a recomendar que los estudiantes inviertan un tiempo en realizar este recorrido sobre el árbol de la *Figura 1* para la secuencia "`cascao`" y comprueben así que se obtienen exactamente las palabras indicadas.

Al igual que en el apartado anterior, la función `search` va a necesitar una serie de funciones auxiliares (que tampoco serán visibles desde el exterior del módulo). Las más importantes son las dos siguientes:

Función searchInTree

Su tipo sería `searchInTree :: String -> String -> Dictionary -> [String]`. En el primer parámetro recibe los caracteres de la secuencia aún no utilizados, en el segundo recibe la palabra que ya se ha formado hasta llegar al nodo del árbol que se recibe como tercer parámetro.

Deberá devolver todas las palabras contenidas en el diccionario cuyo comienzo sea el contenido del segundo parámetro y continúen a partir del nodo actual (tercer parámetro) empleando únicamente caracteres de la secuencia aún no explorada (primer parámetro).

Función searchChild

De tipo `searchChild :: String -> String -> Dictionary -> [String]`. La función recibe la secuencia aún no explorada, el comienzo de palabra ya construido (en base al recorrido empleado para llegar a este nodo desde la raíz) y un nodo (hijo del nodo que se está explorando actualmente). Esta es la función que va a aplicar el algoritmo explicado anteriormente para devolver la lista de palabras que se pueden formar con el comienzo ya construido, y con los caracteres que aún no se han utilizado.

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** (en ambos módulos), 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 `Diccionario.hs`, con las funciones descritas en este enunciado, así como todas las funciones auxiliares que sean necesarias.
- Una memoria con:
    - Una pequeña descripción de las funciones programadas.
    - Las respuestas a las cuestiones sobre la práctica.