MEMORIA

Teoría de los lenguajes de programación

Curso 2019-2020

@mordisko en Telegram| hector@luaces-novo.es

CONTENIDO

INTRODUCCIÓN

Sirva este preámbulo para introducir la memoria de la PEC de este año.

En aras de dejar documentado ante cualquier posible problema técnico, aclaro que la parte práctica de la misma se realizó en un ordenador de arquitectura x64 basado en Linux (Ubuntu 19.10), usando el intérprete GHCi en su versión 8.6.5.

A continuación enumero los ficheros que podrán encontrarse en el archivo comprimido que he entregado:

- **memoria.pdf**: Este archivo.
- **src/**: Directorio que contiene la programación de la aplicación.
- **src/diccionario.txt**: Fichero con todas las palabras del diccionario, proporcionado por el equipo docente.
- **src/Main.hs**: Fichero de programación principal, proporcionado por el equipo docente.
- **src/Dictionary.hs**: Fichero de programación auxiliar con el código del diccionario implementado por el alumno.

A mayores de la presente documentación, el fichero **src/Dictionary.hs** se entrega con todas las funciones documentadas a nivel de código en aras de intentar favorecer la comprensión de las mismas tanto por el equipo docente como por el alumno cuando éste revise la práctica meses después y no recuerde los detalles de la implementación.

DESCRIPCIÓN FUNCIONES PROGRAMADAS

Las funciones programadas se separan en dos categorías principales: funciones requeridas, que son aquellas para las que el equipo docente facilitaba la firma y exigía su implementación, y funciones auxiliares, que son las que el alumno implementa a discreción para que las primeras puedan realizar su trabajo.

A continuación se explicarán los detalles de implementación de las mismas. Además, en las funciones auxiliares se definirán los parámetros y valores de retorno que utilizan, no siendo el caso de las funciones principales, pues dicha información ya está contenida en el propio enunciado de la práctica.

getRootFirst

Esta función auxiliar toma como parámetro un *Dictionary* -el cual se asume que es un *ROOTNODE*, específicamente-, y devuelve el primer elemento de la lista de diccionarios que es contenido por el mismo.

Es principalmente una función de depuración usada por el alumno en algunas pruebas y no tiene utilidad práctica.

getLetterNodeChar

Esta función auxiliar toma como parámetro un *Dictionary* de tipo *LETTERNODE* y devuelve como resultado un único *Char* cuyo valor es el mismo que el *LETTERNODE* tiene asociado.

Es usado en algunas funciones principales para realizar la comparación del carácter actual que se está evaluando para una palabra con los diferentes nodos del árbol que se van recorriendo.

La función es trivial e implementa un único caso base que usa un patrón para extraer y devolver el carácter en cuestión.

getLetterNodeDict

Función auxiliar muy similar a **getLetterNodeChar** que, de la misma forma, toma como parámetro un *Dictionary* de tipo *LETTERNODE* pero que, en lugar de devolver un carácter, devuelve la lista de *Dictionary* que está asociada al propio *LETTERNODE*.

Es usada por algunas funciones de recursión cuando se determina que se ha de profundizar en las distintas ramas hijas de *LETTERNODE* cuyo carácter coincide con alguno que está siendo evaluado.

Se implementa con un único caso trivial que usa un patrón para extraer la lista de diccionarios en cuestión.

findLetterNodeForChar

Función auxiliar de gran importancia que toma como parámetro un *Char* y una lista de *Dictionary* de cualquier tipo y localiza en dicha lista el primer elemento de tipo *LETTERNODE* cuyo carácter asociado coincida con el que se le pasa como parámetro a la función y lo devuelve como resultado en una lista única.

La función se implementa con tres patrones. El primero es un caso elemental que devuelve una lista vacía en el caso de que la lista de *Dictionary* que reciba también sea vacía. Es el caso que termina la recursión.

El segundo patrón es el que implementa la recursión y funciona gracias a hacer *match* a un *LETTERNODE* y su lista de *Dictionary* asociada. Para cada nodo usa una guardia que realiza una de dos operaciones:

1. Si el *Char* buscado coincide con el asociado al *LETTERNODE* actual, termina y devuelve dicho *LETTERNODE*.

2. En cualquier otro caso, corta la cabeza de la lista de *Dictionary* y utiliza la recursión para seguir evaluando la lista en busca de un *match*.

Finalmente un último patrón es usado para hacer *match* de todos aquellos *Dictionary* que no sean un *LETTERNODE* (p.ej.: un *WORDNODE*) con el fin de ignorarlo y parar la recursión. Este patrón es requerido puesto que de no existir la función no sería capaz de recorrer una lista en donde haya variables *Dictionary* de tipo heterogéneo.

Aclarar que la función devuelve una lista porque el alumno necesitaba un valor para emular al "resultado nulo" (que en este caso es la lista vacía) y no estaba del todo claro si definir nuevos valores en el tipo de datos *Dictionary* era algo que la práctica permitiese. De ser así se habría optado por crear un tipo de *Dictionary* nulo, como puede ser *EMPTY_DICTIONARY* que sería usado como este valor "nulo" que representa que la búsqueda fue infructuosa.

replaceLetterNode

Esta última función auxiliar es muy importante y es utilizada en las funciones principales para reemplazar la lista de diccionarios de un *LETTERNODE* con una nueva lista. Específicamente, se usa cuando una palabra es insertada en el diccionario a partir de un carácter ya existente en otra. En estos casos es necesario crear la nueva rama del árbol y, al terminar, sustituir en la rama original la lista de la palabra madre por la nueva lista que también incluye la palabra hijo.

La función toma los siguientes parámetros:

- Una lista de variables *Dictionary*, que se recorrerá para buscar al que queremos reemplazar.
- Un *Char* que será el que será usado para identificar al *LETTERNODE* que queremos reemplazar.
- Una nueva lista de *Dictionary* que será la que será insertada en cualquier *LETTERNODE* encontrado en el árbol que encaje con la búsqueda.

La función devuelve la misma lista de *Dictionary* que recibe pero ya con cualquier reemplazo necesario realizado con éxito.

Para implementarse se vale de tres patrones:

- Un patrón elemental usado para devolver una lista vacía en el caso de que se reciba una lista vacía. Sirve también para terminar la recursión.

- Un patrón que hará *match* a una lista cuya cabeza sea un *LETTERNODE*. Este patrón realiza dos acciones que están definidas con dos guardas:
    1. Si el carácter asociado al *LETTERNODE* es el que buscamos, sustituirá su lista asociada por la pasada como parámetro y continuará la recursión.
    2. En caso contrario continuará la recursión sin hacer modificaciones.
- Finalmente, un tercer patrón es usado para hacer *match* de la lista cuando la cabeza de ésta no es un *LETTERNODE*. En este caso continúa la recursión sin hacer modificaciones.

buildDict

Esta función se define con un único patrón que realiza una única llamada a ***insert***, a la que le pasa la lista de palabras que componen el diccionario y un nuevo *ROOTNODE* que será el usado como acumulador para devolver el diccionario que requiere la práctica.

Insert

Esta función es llamada por ***buildDict*** y se implementa con dos patrones, siendo el primero el caso elemental que devuelve el mismo *Dictionary* recibido como parámetro (que será un *ROOTNODE*) sin modificar cuando la lista de palabras a insertar esté vacía.

El segundo patrón toma la primera palabra a añadir al diccionario y se la envía a la función ***insertInTree*** para que ésta la añada al diccionario donde corresponda. El resultado de esta llamada es concatenado al de utilizar la recursión sobre ***insert*** con el resto de palabras pendientes.

insertInTree

Mi implementación de ***insertInTree*** no es más que un *wrapper* de la función ***insertChild*** y por lo tanto se implementa con dos únicos patrones: uno elemental para salir de la recursión y el segundo que llama a ***insertChild*** pasándole como parámetros la palabra a añadir, su primer carácter y el diccionario acumulado creado hasta el momento.

insertChild

Mi implementación de ***insertChild*** se hizo con dos patrones, ambos apoyándose en una estructura *where* que utiliza la función auxiliar ***findLetterNodeForChar*** para buscar el carácter inicial recibido como parámetro en el diccionario que también se recibe.

El primer patrón es el caso base y es al que se llega cuando no queda ninguna letra de la palabra a insertar. En este caso sencillamente se añade un *LETTERNODE* y *WORDNODE* al diccionario acumulador en el caso de que la palabra añadida tenga que insertarse entera en la raíz. En el caso de que la palabra no tenga que insertarse en la raíz, si no que nazca a partir de algún carácter ya existente, la función usará una llamada a ***replaceLetterNode*** para conseguir tal hecho.

El segundo caso es el dado cuando la lista de caracteres recibida aún no está vacía y realiza las mismas operaciones que el primer caso con la excepción de que realiza una llamada recursiva sobre sí misma para seguir insertando el resto de caracteres.

Search

Mi implementación de esta función se realiza con dos patrones. Uno de ellos es el caso base que devuelve una lista vacía cuando la lista de caracteres a buscar está también vacía.

El segundo patrón realiza una llamada a la función ***searchInTree*** pasándole como parámetros la lista de palabras a buscar, el *Dictionary* en el que se realiza la búsqueda y una cadena vacía, que representa el acumulador que usará la función para determinar las palabras encontradas.

Nótese que la función hace uso de la función nativa ***sort*** para ordenar las palabras encontradas.

searchInTree

Esta función se implementa con tres patrones, siendo dos de ellos casos base.

El primer caso base devuelve una lista vacía cuando los caracteres de búsqueda son también vacíos.

El segundo es similar, pues devuelve una lista vacía cuando no queden palabras en el diccionario que se está examinando en ese momento.

El tercer patrón descompone el *ROOTNODE* recibido u utiliza una llamada a la función ***searchChild*** sobre el primer hijo del *ROOTNODE* a la que concatena la llamada recursiva a ***searchInTree*** sobre la cola de diccionarios asociados al *ROOTNODE* actual.

searchChild

Esta función se implementa con tres patrones, dos de los cuales son casos base.

El primer caso base hace *match* a una lista que termine en *WORDNODE*, es decir, encuentra el final de una palabra. Devuelve, como se espera, la palabra completa que se considera que se ha encontrado.

El segundo caso base devuelve una lista vacía cuando la lista de caracteres de búsqueda restantes esté también vacía.

Finalmente el tercer patrón, encargado, de la recursión, se apoya en un *where* que inicializa la misma cadena de secuencias que se ha recibido con el carácter actual eliminado, así como la variable acumuladora de la palabra que se está encontrando con el carácter actual ya concatenado.

Este tercer patrón descompone un *LETTERNODE* y si su carácter está entre los disponibles para buscarrealiza una operación compleja (***concat ((map ((searchChild x') w') d))***) que se resume en que para cada elemento del diccionario a analizar, realiza una llamada recursiva donde elimina el carácter actual de los disponibles y concatena el encontrado al acumulador**,** usando después **concat** para "aplanar" la lista de listas resultante.

El tercer patrón tiene también un caso elemental que se da cuando el carácter del *LETTERNODE* no está entre los que disponemos para buscar. En este caso devuelve una lista vacía.

RESPUESTAS A CUESTIONES TEÓRICAS DE LA PRÁCTICA

Pregunta 1

> *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?*

Pregunta 2

> *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*

Para poder entender el significado del corte (*!*) es previamente necesario explicar someramente el funcionamiento del intérprete de Prolog.

Cuando éste carga un programa y se le solicita ejecutar una función comenzará buscando todos los predicados que puedan ejecutar la llamada y anotándolos como *puntos de entrada*.

Con la lista de puntos de entrada ya localizada, el intérprete ejecutará el primero de ellos hasta que termine -en cuyo caso se asume que su resultado es el que devolverá la función ejecutada- o hasta que éste falle.

En este segundo caso Prolog seleccionará el siguiente predicado que sirva como punto de entrada de la lista que previamente había seleccionado y lo ejecutará de la misma forma que el anterior: en el caso de que tenga éxito dará por terminada la función y buscará otro punto de entrada en caso contrario.

La ejecución de la función terminará con un error si en el momento en el que un predicado termina de forma no exitosa no hay otro punto de entrada disponible. A todo este proceso se le conoce como *backtracking*.

Con esto aclarado podemos definir el predicado de corte (*!*) como un mecanismo del lenguaje para decirle a Prolog que no debe continuar el proceso de *backtracking* para el predicado en ejecución.

Esto permite, por ejemplo, optimizar el código para no explorar predicados innecesarios, manipular búsquedas, facilitar la lectura del código o crear predicados con una única solución.

Java no implementa este proceso de *backtracking* así que realmente no hay un mecanismo homónimo que podamos mentar como el "*equivalente del predicado* corte". No obstante, un buen símil circunstancial podrían ser las sentencias *return o break*.

La primera nos permite abandonar el cuerpo de una función devolviendo o no un valor. La segunda, si bien no sirve para abandonar una función, permite abortar un bucle y detener su ejecución, lo cual podría ser muy parecido al predicado de corte si la implementación de nuestro algoritmo usase algún bucle para iterar sobre las soluciones (p.ej.: podría hacer que un algoritmo de búsqueda se detuviese).

Una sentencia similar también sería *continue* que nos permite saltarnos pasos en un bucle. Esto permitiría, por ejemplo, hacer que un algoritmo de exploración de un árbol se saltase la búsqueda de determinados nodos.

Insisto en que la similitud de las tres sentencias es situacional debido a que Prolog y Java usan paradigmas distintos, siendo en este último necesario el suponer algún tipo de implementación en particular, como las de búsqueda y exploración ya mentadas. No obstante, y bajo mi punto de vista, creo que los tres ejemplos podrían representar algo similar al corte y, de hecho, cuando aprendí Prolog en otra asignatura me resultó más fácil entenderla si me imaginaba el *backtracking* como un algoritmo que usaba un bucle y comparaba la exclamación con un *break* dentro del mismo.

Pregunta 3

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).

- El tipo de datos *WordsN* tiene un constructor de tipo de datos estructurado que representa un *array* de cadenas (o *Strings*), que en Haskell, a su vez, son un *array* de caracteres (o *Char*).
- El tipo de datos *Words* también tiene un constructor de tipo de datos estructurado que representa un *array* de tipo de datos *WordsN*, cuyo constructor se explicó con anterioridad.
- El tipo de datos *Dictionary* tiene un constructor de tipo de datos estructurado que representa los distintos nodos que pueden ser representados en el árbol de diccionarios del enunciado. Además es un tipo de datos recursivo ya que algunos de sus valores (en concreto *ROOTNODE* y *LETTERNODE*) utilizan a su vez un tipo de datos *Dictionary* en cada uno de sus constructores.