TEORÍA DE LOS LENGUAJES DE PROGRAMACIÓN

PRÁCTICA CURSO 2015-2016
SKYLINE DE UNA CIUDAD

David Pastor Sanz

Índice

1. Descripción de las funciones/predicados programadas/os

1.1. Funciones Haskell

- `resuelveSkyline` Esta función devuelve la línea de horizonte de un conjunto de edificios. Obtiene como parámetros una seria de edificios, devuelve la línea de horizonte de un edificio. En otro caso divide la lista en dos y llama recursivamente con cada una de las listas resultado. Después se utiliza la función `combina` para combinar los resultados parciales de las dos listas. Se ha utilizado una estructira `let...in...` para definir las dos sublistas y después en el `in` realizar la combinación. Esta es la función principal que utiliza la estrategia de **divide y vencerás**.
  - `div` Hace una llamada a `divide` para dividir la lista parámetro en dos. Se usa esta subfunción para hacer más claro el código.
- `edificioAskyline`: Se encarga de devolver la línea de horizonte de un edificio, como parámetro de entrada toma un tripla de las coordenadas de un edificio *Skyline*. Es una función directa y sencilla de implementar.
- `divide`: Toma una lista de edificios y devuelve una tupla de dos listas, de edificios también, del mismo tamaño o la primera con un edificio más, si esta es impar. Se ha decidido implementarla de manera recursiva en vez de utilizar las funciones predefinidas `length` y `take`, `drop` o `splitAt`.
  - `divideAux` Es la función auxiliar a la que llama `divide` para obtener las dos listas de edificios. Contiene un acumulador donde se irán almacenando los edificios en ambas listas. Divide la lista parámetro en primero, segundo y resto (siempre que haya dos o mas edificios) e introduce en el las listas de la tupla del acumulador primero y segundo, uno en cada una de las listas, seguidamente se hace una llamada recursiva a esta función con el resto y el acumulador como parámetros. Si la lista parámetro contiene un sólo elemento lo introduce en la lista primera de la tupla, si esta vacía devuelve la tupla con dos listas vacías.
- `combina` Como su nombre indica, combina dos listas de coordenadas en una sola. Para realizar esta tarea llama a una subfunción.
  - `combinaAux` Toma como parámetros las dos listas que le pasa `combina` además de las últimas alturas leídas de las cabezas de las listas parámetro y la última altura introducida en la lista de coordenadas de retorno de la función, estas 3 alturas se inician a 0. Si alguna de las listas parámetro es una lista vacía, entonces devolverá la otra lista. En el caso de que ambas sean listas vacías devolverá una lista vacía. En otro caso se compararán las abcisas de las coordenadas de las cabezas de las listas parámetro y si la última altura guardada coincide con la nueva a introducir, dependiendo de esto se introducirá o no la nueva coordenada (que depende si la de la primera lista es mayor, menor o igual que la de la segunda) en la lista solución.
  - `difh` Devuelve verdadero si la nueva coordenada a introducir en la solución tiene distinta altura que la anterior. Esta función se ha introducido para que sea más legible y limpio el código. Se podría haber prescindido de ella.
- `dibujaSkyline` De una lista de coordenadas como parámetro devuelve un `String` compuesto por "*" y " " (asteriscos y espacios) que simbolizan el **Skyline** de la lista parámetro. Se realiza línea a línea, de derecha a izquierda y de arriba hacia abajo empezando por la altura más alta, al final se marca un suelo con "-" (barras). Esta función se apoya en una subfunción auxiliar para devolver dicho resultado junto a `listh` que devuelve una lista con todas las alturas.

- `dibujaSkylineAux` Tomando como parámetros la lista de alturas, la máxima altura de la lista parámetro de `dibujaSkyline` y un acumulador (`result`) en el que se irá almacenando el resultado. Esta función auxiliar irá recorriendo la lista parámetro y comprobará si el siguiente elemento es mayor o igual que la altura, si es cierto, introducirá en el resultado un asterisco, si no un espacio. Cuando la lista se vacíe se introducirá un salto de linea en resultado y se llamará de nuevo a la función con la lista completa y la altura del segundo parámetro se decrementará en uno. Una vez que este segundo parámetro valga 0, se recorrerá la lista pero esta vez introducirá en el resultado barras bajas, indicando que es el suelo. Una vez vacía la lista y el segundo parámetro tenga el valor 0 se devolverá el resultado.
- `listh` Devuelve la lista de alturas. Esta no tiene ningún parámetro, se delega en una subfunción que calculará dicha. Esto es para hacer más legible el código de `dibujaSkylineAux`. Cabe reseñar que esta función solamente se calcula la primera vez que se le llama, gracias a la **memoización**.
  - `listhAux` La utiliza `listh` para calcular la lsita de alturas. Se la introduce como parámetro la lista de coordenadas parámetro de `dibujaSkyline` y un acumulador solución, una lista de enteros, que se inicia con tantos ceros como la coordenada x de la cabeza de la lista parámetro. Si la lista parámetro está vacía o contiene sólo un elemento devuelve el acumulador, `sol`. En otro caso, introduce tantas coordenadas y de la cabeza de la lista como la diferencia de la coordenada x de la siguiente coordenada de la lista menos la coordenada x de la cabeza. Se le llama recursivamente quitando la cabeza a la lista.

1.2. Predicados Prolog

- `resuelveSkyline` Es el predicado principal. Recibe una lista de edificios y devuelve una lista con las coordenadas que simbolizan su linea de horizonte. Si la lista de edificios contiene solo uno simplemente llamará a `edificioAskyline` que devolverá el *Skyline* de ese único edificio. En otro caso se partirá la lista en dos mediante `divide` y se llamará recursivamente con este predicado a cada una de las listas resultado, después se unificarán usando `combina` para devolver el resultado.
- `edificioAskyline` Es un predicado directo y simple. Se le mete un edificio como parámetro y una variable. Traduce la variable a partir del edificio y así esta pasa a tener el valor de su linea de horizonte.
- `divide` Al igual que la función **Haskell** este predicado realiza la misma acción separando la lista parámetro en primero, segundo y resto e introduciendo primero en una de las listas solución y segundo en la otra, y llamando recursivamente hasta que quede un solo elemento, el cual se introducirá en la primera lista, o este vacía la lista.
- `combina` Este predicado junto con su auxiliar son prácticamente idéntico a la función **Haskell**. Recibe dos listas de coordenadas y las unifica en una lista resultado.
  - `combinaAux` Usado por `combina` mediante inmersión toma los mismos parámetros (las dos listas parámetro y la solución) además, al igual que la función **Haskell**, las últimas alturas observadas de cada lista y la última introducida en la lista solución. Su funcionamiento es exactamente el mismo, comparando las coordenadas x de cada cabeza de las listas parámetro y si la que se va a introducir tiene una altura distinta a la última altura introducida, se introduce, y si no no. En ambos casos se llama recursivamente al predicado.
  - `max` Devuelve el mayor de los dos números metidos como parámetros.
- `dibujaSkyline` A partir de una lista de coordenadas devuelve un `String` que simboliza el *Skyline* de dicha lista. Para ello calcula la altura máxima que hay en la lista, la lista de alturas máximas y a partir de ello y gracias al predicado auxiliar `dibujaSkylineAux` calcula una lista con todos los caracteres (*,- y espacios) que "pintarán" el *Skyline*. Después esa lista se transformará en `String` gracias al predicado predefinido de **Prolog** `atomics_to_string` y devolverá dicho `String`.
  - `dibujaSkylineAux` Este predicado realiza la misma función que la función **Haskell**.
  - `listh` A partir de una lsita de coordenadas, devuelve la lista que representa todas sus alturas. Utiliza el siguiente predicado auxiliar para realizar dicha tarea.
    - `listhAux` De manera similar a su homónima en **Haskell** calcula la lista de todas las alturas, va introduciendo el número de alturas restando la coordenadas x de las posiciones segunda a la primera de la lista. Una vez calculada esta, concatena una lista de ceros (tantos como el valor de la coordenada x de la cabeza de la lista) a la lista de alturas.
  - `concatenar` Concatena dos listas y devuelve el resultado.
  - `takeAndRepeat` Devuelve una lista con X veces repetido el elemento H.
  - `maxh` Calcula la altura máxima que hay en la lista que toma como parámetro.

2. Cuestiones sobre la práctica

2.1. Función dibujaSkyline en Haskell

```haskell
-- Dibujar skyline. Imprime un skyline en pantalla
dibujaSkyline :: Skyline -> String
dibujaSkyline p@((a,b):xs) = dibujaSkylineAux listh (maximum listh) " "
  where
    dibujaSkylineAux [] 0 result       = result ++ "-"
    dibujaSkylineAux (ph:phx) 0 result = dibujaSkylineAux phx 0 (result ++ "-")
    dibujaSkylineAux [] 1 result       = dibujaSkylineAux listh 0 (result ++ " \n-")
    dibujaSkylineAux [] n result       = dibujaSkylineAux listh (n-1) (result ++ " \n ")
    dibujaSkylineAux (ph:phx) n result
      | ph >= n   = dibujaSkylineAux phx n (result ++ "*")
      | otherwise = dibujaSkylineAux phx n (result ++ " ")
    listh = (listhAux p (take (a-1) (repeat 0)))
      where
        listhAux [] sol                      = sol
        listhAux (head:[]) sol               = sol
        listhAux ((a1,b1):(a2,b2):tail) sol  = listhAux ((a2,b2):tail)
                                                (sol++(take(a2-a1)(repeat b1)))
```

2.2. Predicado dibujaSkyline en Prolog

```prolog
%Devuelve una cadena de caracteres que simbolizan un skyline
dibujaSkyline(L) :- dibujaSkyline(L,D), write(D).
dibujaSkyline(L,D) :- maxh(L,M), listh(L,LH),
                      dibujaSkylineAux(LH,M,LH,A),
                      atomics_to_string(A,D), !.
dibujaSkylineAux([],0,_,['_\n'|[]])  :- !.
dibujaSkylineAux([_|R],0,_,['-'|S])  :- dibujaSkylineAux(R,0,_,S).
dibujaSkylineAux([],H,LH,['\n'|S])   :- H1 is H-1,
                                        dibujaSkylineAux(LH,H1,LH,S).
dibujaSkylineAux([X|R],H,LH,['*'|S]) :- X >= H, !,
                                        dibujaSkylineAux(R,H,LH,S).
dibujaSkylineAux([X|R],H,LH,[' '|S]) :- X < H, !,
                                        dibujaSkylineAux(R,H,LH,S).

%Devuelve una lista con todas las alturas
listh([],[]) :- !.
listh([_|[]],[]) :- !.
listh([c(X1,H1),c(X2,H2)|R],L) :- X is X2-X1,
                                  listhAux([c(X2,H2)|R],X,H1,L2), !,
                                  takeAndRepeat(X1,0,L1),
                                  concatenar(L1,L2,L).
listhAux([_|[]],1,H,[H|[]])           :- !.
listhAux([c(X1,H1),c(X2,H2)|R],0,_,L) :- X is X2-X1,
                                         listhAux([c(X2,H2)|R],X,H1,L).
listhAux(R,X,H,[H|L])                 :- X1 is X-1,
                                         listhAux(R,X1,H,L).

%Concatena dos listas
concatenar([],L,L) :- !.
concatenar([X|R1],L,[X|R2]) :- concatenar(R1,L,R2).
%Introduce X alturas H en una lista
takeAndRepeat(0,_,[]) :- !.
takeAndRepeat(X,H,[H|LH]) :- X1 is X-1, takeAndRepeat(X1,H,LH).
%Devuele la altura maxima de un Skyline
maxh([c(_,H)|[]],H) :- !.
maxh([c(_,H)|R],H) :- maxh(R,C2), H > C2, !.
maxh([_|R],C) :- maxh(R,C).
```

Tanto la función que dibuja el **Skyline** en **Haskell** como el predicado que hace lo mismo en **Prolog** se han realizado según pide el enunciado de la práctica. Ambos se han explicado en los puntos 1.1 y 1.2 de este documento. Se puede observar su correcto funcionamiento al ejecutar sendos programas, incluidos a parte de este documento.

2.3. Comparación de la eficiencia de la programación en Haskell, Prolog y Java

El término de **eficiencia de la programación** se refiere a la capacidad expresiva del lenguaje: velocidad y facilidad para escribir programas complejos de forma que se relacione de manera sencilla la idea que tiene el programador con el código. Esta relacionado con la potencia y la generalidad de los mecanismos de abstracción y la sintaxis. Dicho con otras palabras, este concepto involucra analizar cuán eficiente es un lenguaje a la hora de escribir programas en él.

El enunciado pide que se compare dicha eficiencia en el algoritmo del `skyline` en los lenguajes **Haskell**, **Prolog** y **Java** y considerar cuál es el más adecuado. Para ello se van a comparar distintos aspectos realizándose una serie de preguntas:

- ¿Es costoso para el programador trabajar con las estructuras de datos?

  Tanto en **Prolog**, como en **Haskell** trabajar con las estructuras de datos, desde el punto de vista del alumno, es más directo que hacerlo desde **Java**. Se pueden tratar directamente sin necesitar métodos los datos de ciertas de estas estructuras de manera directa, en cambio en **Java** se necesitan el uso de estos, como los *getters* y *setters* para simplemente obtener ciertos datos de estas.

- ¿Es costoso para el programador desarrollar el algoritmo completo?

  Si se maneja el uso de los **lenguajes declarativos**, bajo el punto de vista del autor, es mucho más simple de desarrollar que en un **lenguaje imperativo**, ya que se necesitan de muchas menos lineas de código y menos métodos para el acceso a las estructuras de datos que se utilicen, en este caso listas.

- ¿Ofrece facilidades el lenguaje para realizar el programa?

  Los **lenguajes declarativos** muestran mayor facilidad para poder utilizar la recursión que los **lenguajes imperativos** además de lo que se ha hablado anteriormente del uso de las estructuras de datos. Por tanto también se puede decir que es un punto a favor.

- ¿Es fácil de entender el significado del lenguaje observando el algoritmo?

  En este punto es quizás donde un **lenguaje imperativo** como **Java** es más eficiente, respecto a la programación, que los **lenguajes declarativos**. No es tan sencillo de leer un algoritmo que usa solamente recursión como uno que puede utilizar bucles.

Con todas estas premisas se puede decir que es más eficiente el algoritmo escrito en un **lenguaje declarativo** como puede ser en **Haskell** o **Prolog**. Dentro de estos dos la ventaja que tiene el segundo frente al primero es la sintaxis concisa, la ausencia de declaración de variables y la independencia con el mecanismo de ejecución. Por el contrario **Haskell** es más fácil de leer, bajo la opinión del autor, además de ser más eficiente computacionalmente en este tipo de problemas y más fiable. Con todo esto el alumno piensa que el lenguaje más eficiente es **Haskell**, además de parecer más fácil de entender y aprender que **Prolog**.

2.4. Predicado predefinido no lógico, corte (!), en Prolog ¿Cómo se realiza este efecto en Java?

El predicado de corte (!) es un predicado que siempre se cumple, siempre se evalúa como verdadero, y origina fallo cuando se produce el retroceso en el *backtracking*. Podría decirse que no vuelve de la recursión por el camino en que estaba el corte al descender en el árbol de búsqueda. Gracias a esto se evita que se puede acotar el espacio de búsqueda no explorando ramas que no llevarán a una solución. Además también son útiles si sólo se desea encontrar una solución, en cuanto se encuentre la primera solución se "para" la búsqueda.

El predicado corte en **Java** podría decirse que es, para un algoritmo de *vuelta atrás*, añadirle una variable *booleana* para que el algoritmo pare una vez encontrada una solución, si la hay; o aplicarle una serie de condiciones para que no continúe la búsqueda por determinadas ramas donde seguro no habrá una solución y así realizar una poda. También tiene cierta similitud a los "`if`", pues puede servir para indicar ciertas condiciones, por ejemplo:

```
rango(N,1) :- N >= 0, N =<2, !.

rango(N,3).
```

Si el valor de **N** está entre 0 y 2 devolverá 1 y no se evaluará la segunda regla, en otro caso devolverá 3 la segunda regla.

2.5. Tipos de datos del problema en Haskell, Prolog y Java

A parte de los tipos predefinidos por los lenguajes, se han utilizado **edificio**, **coordenada** y **skyline**. Cada uno de ellos se explican a continuación:

- `edificio`: es una tripla de puntos sobre un plano, la primera y segunda muestran el comienzo y el final sobre el eje de coordenadas x y la tercera, sobre el eje y la altura que toma un edificio. En cada uno de los lenguajes de programación son:
  - **Haskell** Se usa un tripla de enteros. El acceso a cada coordenada es directo.
    ```
    type Edificio = (Int,Int,Int)
    ```
  - **Prolog** Se define de la misma manera que en **Haskell**, aunque no sería necesario indicar el tipo.
    ```
    ed(X1,X2,H).
    ```
  - **Java** Para este lenguaje es necesario crear una clase con tres campos de enteros. Se crea un constructor con los tres parámetros para poder crear un objeto edificio. Además contiene métodos *getters* para cada uno de los campos. También contiene un método que simula `edificioAskyline`, es decir devuelve la línea horizonte de un objeto edificio.
    ```
    Edificio(int x1,int x2,int h)
    ```

- `coordenada`: una dupla de enteros que simbolizan un punto sobre el plano de dos dimensiones.
  - **Haskell** Se usa un dupla de enteros. El acceso a cada coordenada es directo.
    ```
    type Coordenada = (Int,Int)
    ```
  - **Prolog** Se define de la misma manera que en **Haskell**, aunque no sería necesario indicar el tipo.
    ```
    c(X,H).
    ```
  - **Java** Es una clase con dos campos de enteros cada uno para punto sobre el plano. Se crea un constructor con los dos parámetros para poder crear un objeto coordenada. Además contiene métodos *getters* para cada uno de los campos.
    ```
    Coordenada(int x,int h)
    ```
- `skyline`: Es una lista de coordenadas que simbolizan la línea horizonte de uno o un grupo de edificios.
  - **Haskell** Una lista de coordenadas, se define en el programa.
    ```
    type Skyline = [Coordenada]
    ```
  - **Prolog** No se define explícitamente en el programa pero podría representarse como:
    ```
    skyline[c(X,H)].
    ```
  - **Java** Un `ArrayList` de objetos coordenada.
    ```
    ArrayList<Coordenada>skyline;
    ```

Herramientas utilizadas.

Para realizar los códigos se ha utilizado el editor de textos **Notepad++**. Como intérpretes en **Haskell** y **Prolog** se han usado **ghci** y **SWI-Prolog** respectivamente. Este documento se ha editado mediante el editor de textos **TEXMaker** utilizando LaTeX.