Baltazar Coyotl Miguel Angel

Práctica número 1

Funciones puras, de primer orden y de orden superior.

Todas las opciones

14 de octubre del 2021

Grupo 3CM2

Materia: Paradigmas de Programación

Indice.-

Introducción.

Los lenguajes Haskell tanto Lisp son lenguajes los cuales su forma de programación es funcional lo cual significa que son un lenguaje de programación declarativo donde el programador especifica lo que quiere hacer, en lugar de lidiar con el estado de los objetos. Es decir, las funciones estarían en un primer lugar y nos centraremos en expresiones que pueden ser asignadas a cualquier variable.

Primero me enfocare en Haskell:

Haskell es un lenguaje de programación puramente funcional, cuya primera versión fue lanzada en 1990. Su creador es Haskell Brooks Curry, que sentó las bases de los lenguajes de programación funcional con su trabajo sobre lógica combinatoria. Haskell se basa en el cálculo lambda (lenguaje formal para la investigación de funciones). Los programas escritos en Haskell se representan siempre como funciones matemáticas, pero estas funciones nunca tienen efectos secundarios ni derivados. De este modo, cada función utilizada siempre devuelve el mismo resultado con la misma entrada, y el estado del programa nunca cambia. Por esto, el valor de una expresión o el resultado de una función dependen exclusivamente de los parámetros de entrada en el momento. En Haskell no pueden hacerse construcciones de lenguaje imperativo para programar una secuencia de declaraciones.

Ahora en Lisp:

Lisp es un lenguaje que muestra la información estructurada en listas en las que se pueden gestionar la información que estas contienen. Lisp (List-Processing), porque fue creado principalmente para el procesamiento de listas. Es un lenguaje funcional que se apoya en la utilización de funciones matemáticas para el control de los datos. Cada función de Lisp y cada programa que se genera con él vienen dado en forma de lista. Por esta razón los datos no se pueden diferenciar sintácticamente de los programas.

A este tipo de lenguaje se les denomina aplicativos o funcionales porque se basan en la aplicación de funciones a los datos. El Lisp diferencia dos tipos de elementos básicos: El átomo, datos elementales de varios tipos como números, símbolos, caracteres y cadenas de caracteres. Y las Listas, entre las que podemos nombrar a una lista nula que no tiene ningún elemento.

Desarrollo

Para la primera parte de pide codificar dos funciones que obtengan el segundo y tercer elemento de una lista.

Se comenzó con la función que pide el segundo elemento el cual el primer paso es definir y dar nombre a la función.

```
(define selem
```

A esta función se le dio el nombre de selem debido a que nos regresa el segundo elemento, posterior a esto es importante recordar que es lo que hace car y cdr. Teniendo claro su función se puede aplicar para poder obtener el segundo elemento de una lista, tal como se hace en la siguiente línea de código.

```
(car(cdr l))))
```

Lo anterior nos dice que nos dé el primer elemento del resto de la lista.

Así mismo para obtener el tercer elemento de la lista es un proceso similar, sin embargo, el principal cambio que sufre es el siguiente.

```
 (car(cdr(cdr l)))))
```

Con lo anterior se obtiene el tercer elemento, como se puede ver lo que cambio para obtener el tercer elemento con respecto al otro es que se agregó cdr.

Finalmente, el resultado de compilar este código es el siguiente:

```
(print "Segundo Elemento")
(selem '(A B C D E F H I J K))
(print "Tercer Elemento")
(telem '(1 2 3 4 5 6 7 8 9 10))
Segundo Elemento
B
Tercer Elemento
3
```

Continuando con la codificación se cambia de lenguaje de programación por Haskell del cual ya se ha dado una breve introducción sobre este mismo.

Lo primero por realizar son las funciones para obtener el segundo y tercer elemento de una lista, con lo cual para comenzar:

Función que pide el segundo elemento de una lista

```
selem :: [a] -> a
```

Como se puede notar en la funcion se le da el nombre de selem por segundo elemento y se define que tipo de parametro recibira y el valor que se dara como tipo de retorno.

```
selem (x:xs) = (x:xs) !! 1
```

Consecuentemente se ve lo siguiente (x:xs) donde x es la cabeza de la lista y xs es el resto de la lista, lo cual gracias a !! es que obtiene el segundo elemento de la lista.

Para obtener el tercer elemento es similar a lo anterior solo que hay un cambio pequeño.

```
telem (x:xs) = (x:xs) !! 2
```

Por ultimo se pide generalizar la función twice a una función repetir, de modo que la repetir f n x aplique f a x un total de n veces. Es importante recordar que la función twice fue vista y explicada en el libro Learn you Haskell for Great Good en la pagina 63, con lo cual al generalizar la función su estructura de la nueva función será similar pero con algunos cambios como se podrá ver a continuación

Función Twice:

```
applyTwice :: (a -> a) -> a -> a
```

Funcion Repetir:

```
repetir :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

Como se puede ver en la funcion repetir se agrega un paramtero mas esto se debe a las condiciones que nos ponen las instrucciones sobre la funcion donde

repetir f 0 x produce x
repetir f 1 x produce f x
repetir f 2 x produce f (f x) (que es lo mismo que twice f x)
repetir f 3 x produce f (f (f x))

Continuando con la codificación se establece un caso en el cual parara la recursividad.

```
repetir f 0 x = x
```

Es importante saber que la recursividad es esencial en estos casos. Por lo cual lo último de esta función es la recursividad

```
repetir f y x = f (repetir f (y-1) x )
```

Al haber concretado la función repetir se pide que se use para implementar una función que eleve un numero x a una potencia n, lo cual se ve a continuación:

```
pot :: Integer -> Integer -> Integer
pot x y = repetir (*x) (y-1) x
```

Primero se definen los parámetros que recibirá la función el valor que retornara, posteriormente como se nota en la segunda línea se usa la función repetir que se había hecho anteriormente con los respectivos datos, (*x) esto se usara para multiplicar por sí mismo al número que se dé, (y-1) es para saber las veces que este número se está elevando y por último x es el numero que se va a elevar. Con la explicación anterior a continuación se ve la compilación.

```
putStr "Second element"
putStr "\n"
print(selem [1,2,3,4,5,6,7,8])
putStr "Third element"
putStr "\n"
print(telem [9,8,7,6,5,4,3])
putStr "\n"
putStr "9 a la 5 potencia es: "
print(pot 9 5)
```

```
Second element
2
Third element
7

9 a la 5 potencia es: 59049
```

Conclusión

En conclusión sobre la practica se logro identificar y aplicar lo aprendido cobre Haskell y Lisp dándole un uso didáctico e interesante que nos permite identificar debilidades que se tiene sobre la forma de codificar y al identificar esas debilidades se puede buscar información o consultar para poder entender por completo el tema. Así mismo, se puede notar la forma en la que Haskell y Lisp actúan ya que no es de la misma forma en la que estamos habituados al programar con C o C++, lenguajes en los cuales su forma de programar es diferente.

Bibliografía

(2019). Obtenido de EcuRed:

https://www.ecured.cu/LISP

Digital Guide. (9 de octubre de 2020). Obtenido de

https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/que-es-haskell/