Manual de Haskell

Instituto tecnológico de Mexicali

Carrera:

Ingeniería en sistemas computacionales

Maestro

Atempa Camacho Jorge Antonio

Materia:

Prog. Lógica y funcional

Tema:

Manual de Haskell

Nombre del alumno:

Navarro Castellanos José Amado #12490001

Mexicali, Baja California 9 de Noviembre de 2016

Manual de Haskell

Tabla de contenido

Manual de Haskell

Introducción

La programación en el área de sistemas computacionales es algo del dia a dia, la tecnología avanza a una velocidad bastante alarmante por lo cual con el paso del tiempo han surgido distintos lenguajes y tipos de programación, en este caso programación funcional, en específico Haskell.

En este manual se presenta suficiente información como para que cualquier desarrollador o entusiasta de la programación pueda comenzar a desarrollar en Haskell con el fin de que la comunidad conozca la programación que está siendo utilizada cada vez más en la industria.

Como dato adicional, esta información va de la mano con una lista de reproducción en la plataforma de YouTube sobre el tema, derechos reservados para el usuario jotajotavm.

Sin nada más que decir, comienza el curso.

Manual de Haskell

Conocimientos previos

¿Qué es la programación funcional?

C, Java, Pascal, Ada, y la gran mayoría de los lenguajes, son todos *imperativos*. Ellos son "imperativos" en el sentido de que consisten en una secuencia de comandos, los cuales son ejecutados estrictamente uno después de otro. Haskell es un lenguaje *funcional*. Un programa funcional es una expresión, la cual es ejecutada mediante su evaluación. (Es/Introduccion, 2007)

¿Qué es Haskell?

Haskell es un moderno lenguaje estándar de programación puramente funcional, no-estricto. Ofrece todas las características explicadas arriba, incluyendo tipado polimórfico, evaluación perezosa y funciones de primera clase. También posee un innovador sistema de tipo el cual soporta una forma sistemática de sobrecarga y un sistema de módulos.

Está especialmente diseñado para manejar una amplia gama de aplicaciones, desde análisis numérico hasta simbólico. Para alcanzar estos objetivos, Haskell posee una sintaxis expresiva, y una rica variedad de tipos primitivos, incluyendo enteros y racionales de precisión arbitraria, también como los tipos de enteros, punto flotante y booleanos más convencionales. (Es/Introduccion, 2007)

Una vez que conocemos Haskell podemos proceder con la descarga e instalación del mismo, descargamos la versión dependiendo del sistema operativo en el que se esté trabajando, a continuación accedemos al siguiente enlace:

https://www.haskell.org/platform/

Al descargarlo y ejecutarlo tendremos una terminal parecida a esto:

```
Prelude>
```

Manual de Haskell

1. Operadores

1.1. Operadores Aritméticos

En la terminal de Haskell podemos manipular valores por medio de operadores, tanto aritméticos como lógicos, y todo esto sin la necesidad de crear ningún archivo, cabe destacar que en Haskell los números pueden o no tener definido su tipo, es decir, podemos trabajar con valores enteros o valores con decimales sin ningún problema. En la siguiente tabla se muestran los principales operadores aritméticos, una pequeña descripción de ellos y un ejemplo aplicable en la terminal.

Símbolo	Descripción	Ejemplo
+	Suma valores numéricos.	Prelude> 5+10.222 15.222
-	Resta valores numéricos.	Prelude> 20-16 4
*	Multiplica valores numéricos.	Prelude > 7*6 42
/	Cociente de valores numéricos.	Prelude > 100/50 2
^	Eleva a una potencia valores numéricos.	Prelude> 4^2 16
++	Concatena listas del mismo tipo.	Prelude>[1,2,3,4]++[5,6,7,8] [1,2,3,4,5,6,7,8]
`div`	Cociente de valores numéricos.	Prelude > 100`div`50 2
`mod`	Residuo del Cociente de valores numéricos.	Prelude > 100`mod`50 0

1.2. Operadores Lógicos

Ya conocemos los operadores aritméticos, ahora es turno de los operadores lógicos los cuales trabajan con salidas de tipo booleano, el resultado final depende si la condición aplicada se cumple o no. En la siguiente tabla se muestran los principales operadores lógicos aplicables a valores aritméticos y booleanos respectivamente.

Símbolo	Descripción	Ejemplo
>	Mayor que.	Prelude> 10>5 True
<	Menor que.	Prelude> 7<2 False
>=	Mayor o igual que.	Prelude> 25>=25 True
<=	Menor o igual que.	Prelude> 100<=76 False
/=	Diferente.	Prelude > 78/=9 True
==	Comparación.	>10==2 False
&&	Compuerta lógica And.	>True && False False
||	Compuerta lógica Or.	>True || False True
not	Negación.	>not True False

2. Funciones

2.1. Funciones prefijas

En Haskell, El peso del valor de salida se encuentra en las funciones, no en las variables, es decir, no es necesario asignar una variable por cada salida en las funciones. Existe una gran cantidad de funciones prefijas que al momento de instalar Haskell ya son perfectamente utilizables. Además Haskell cuanta con listas de números y letras con las cuales este trabaja a la hora de invocar las funciones prefijas, por ejemplo, en esta tabla se muestran las funciones de succ, min y max con su respectiva descripción y un pequeño ejemplo.

Funcion	Descripcion	Ejemplo
succ a	Devuelve el número o caracter siguiente al enviado como parámetro.	Prelude>succ 5 6 Prelude>succ 'a' 'b'
min a b	Devuelve el valor más pequeño entre solamente dos números.	Prelude> 5 6 5
max a b	Devuelve el valor más grande entre solamente dos números.	Prelude>9 2 9

Haskell resuelve las funciones anidadas de derecha a izquierda, por ejemplo aplicando todas las funciones anteriores:

```
Prelude> succ (max 8 (min  34.6  90))
35.6
```

2.2. Creación de funciones

En el editor de texto de nuestra preferencia podemos realizar archivos totalmente funcionales para su uso con Haskell con tan solo guardarlos con la extensión **.hs.**

Es importante conocer la estructura que tienen las funciones en Haskell, las cuales se componen de 3 elementos que son: el nombre, los parámetros y las instrucciones a ejecutar.

```
nombredeFuncion Parámetro = instrucciones

ejemploSuma x y = x+y
```

Una vez se tenga una función en el archivo, ejecuta la terminal de Haskell, nos posicionamos en la carpeta en la que almacenamos nuestra función por medio del comando:

```
$>cd nombredetucarpeta
```

Una vez posicionados en la carpeta ejecutamos el siguiente comando donde ***:l*** se encarga de cargar el archivo.

```
Prelude> :l nombredetuarchivo.hs
```

Una vez cargado el archivo, en nuestra terminal cambiará la palabra ***Prelude>*** a ****Main>,*** esto significa que nuestro archivo ha sido cargada exitosamente y podemos utilizar las funciones que este contiene.

por ejemplo la función ejemploSuma que escribimos hace un momento:

```
Prelude> :l nombredetuarchivo.hs
*Main> ejemploSuma 10 7
17
```

Ahora, si por alguna razón modificamos el archivo, este debe ser recargado para ser utilizado, de lo contrario en la ejecución no se verá el cambio aplicado anteriormente. Para recargar el archivo basta con la simple instrucción:

```
Prelude> :r
```

3. Estructura IF

En una gran cantidad de lenguajes de programación la sentencia IF se escribe casi de la misma manera y no necesita de la instrucción else para funcionar en muchos casos. En Haskell se estructura de la siguiente manera:

```
esMayor x = if x < 10
            then "Es menor a 10"
            else "Es mayor a 10"
```

Es obligatorio tener un **then** y un **else** para que el programa pueda ser compilado, la sentencia **then** se utiliza para establecer el resultado que arroja la función en dado caso de que se cumpla la condición, mientras que el **else** se utiliza para el caso contrario.

Cabe aclarar que en el uso de estas instrucciones es obligatorio saber con el tipo de dato que se utiliza para trabajar, es decir, si se quiere trabajar con datos numéricos, tanto el **then**, como **else** deben arrojar un valor del mismo tipo, en este caso numérico, no se pude colocar en el **then** una cadena y en **else** un número. A continuación se muestra un ejemplo de lo anterior mencionado.

```
sumaDiezAMayoresQueVeinte x = if x >= 20
            then x+10
            else x
```

Simplemente si se cumple la condición se realiza una suma del valor que enviamos y se le suma el valor de 10, caso contrario, se retorna el valor enviado sin ninguna modificación.

4. Listas

Anteriormente conocimos distintas funciones que utilizaban listas que Haskell ya tiene por defecto cargadas, tanto de números como de caracteres. Es posible crear listas con los valores deseados, por ejemplo:

```
Prelude> [4,3,2,5]
[4,3,2,5]
```

Para trabajar con listas es importante recordar que estas deben de ser del mismo tipo de dato, es decir, no podemos hacer algo como esto:

```
Prelude> [4,3,2,5,'f']
ERROR
```

También conocemos un operador para concatenar listas, es importante saber que una cadena de caracteres es considerada una lista:

```
Prelude> ['h','o','l','a']
"hola"
```

Por lo tanto podemos aplicar el operador que ya conocemos con cadenas de caracteres, por ejemplo:

```
Prelude> "hola" ++" mundo"
holamundo
```

Algo interesante es que no podemos agregar un valor a una lista de la siguiente manera:

```
Prelude> [1,2,3,4]++5
ERROR
```

Ya que a una lista solo se le pueden agregar listas, por lo tanto, la manera en la cual podemos agregar el valor deseado es de la siguiente manera:

```
Prelude> [1,2,3,4]++[5]
[1,2,3,4,5]
```

Ahora, si nos damos cuenta, al utilizar el operador "++" para concatenar, el valor enviado se coloca al final de la lista, en el caso de que quisiéramos agregar ese valor al comienzo de la lista podemos optar por la siguiente manera:

```
Prelude> 'H' : "ola mundo"
"Hola mundo"
```

Es importante recordar que las cadenas son consideradas una lista de caracteres, por este motivo se manda un carácter con comillas simples y no uno con comillas dobles.

Manual de Haskell

5. Índice en listas

Si ya conocemos algún otro lenguaje de programación es muy probable que conozcamos el concepto de arrays o arreglos que tienen un índice para acceder a sus datos empezando con que el primero valor se encuentra en la posición 0, el segundo en la posición 1 y así sucesivamente. En las listas esto es igual.

Para comenzar utilizaremos la palabra reservad "let" eso nos permite crear una lista, ponerle un nombre y asociarlo para acceder a la lista por medio de ese nombre:

```
Prelude> let lista = [6,7,8]
```

Ahora si escribimos la lista y le damos entrada nos devolverá la lista que hemos definido:

```
Prelude> lista
[6,7,8]
```

Una vez definimos esta lista podemos acceder a su contenido por medio de los índices, esto lo logramos utilizando el operador "!!", por ejemplo, si queremos acceder al primero número de la lista:

```
Prelude> lista !!0
6
```

Haskell puede incluir listas de listas, esto se hace de la siguiente manera:

```
let lista = [[1,2],[3,4]]
```

Entonces si accedemos al primer elemento obtendremos "[1,2]" ya que es una lista de listas. Ahora, si queremos acceder a un valor en concreto en una de estas listas aplicamos lo siguiente

```
lista ! !0 ! !1
 2
```

6. Funciones de Listas

En la siguiente tabla se aprecian algunas de las funciones con las que podemos interactuar con una lista. Suponiendo que tenemos una lista con los valores [1,2,3,4], es importante saber que estas funciones no alteran el contenido de la lista.

Función	Descripción	Ejemplo
length	Devuelve la longitud de una lista.	*Prelude> length lista 4*
head	Devuelve el primer valor de la lista	*Prelude> head lista 1*
tail	Devuelve todos los valores de la lista a partir de la posición 1	*Prelude> tail lista [2,3,4]*
last	Devuelve el ultimo valor de la lista	*Prelude> last lista 4*
init	Devuelve todos los valores de la lista menos el ultimo	*Prelude> init lista [1,2,3]*
reverse	Devuelve la lista volteada	*Prelude> reverse lista [4,3,2,1]*
take n	Devuelve los primeros n valores de una lista	*Prelude> take 2 lista [1,2]*
drop n	Ignora n números y devuelve el resto de una lista	*Prelude> drop 1 lista [2,3,4]*
minimum	Devuelve el valor minimo de una lista	*Prelude>mínimum lista 1*
maximum	Devuelve el valor máximo de una lista	*Prelude>maximum lista 4*

sum	Suma el contenido de una lista	*Prelude> sum lista* *10*
product	Multiplica un valor tras otro de una lista	*Prelude> product lista* *24*
`elem`	Busca un valor en una lista y devuelve un valor booleano	*Prelude> 2`elem` lista* *True* *Prelude> 5`elem` lista* *False*

7. Rangos

En Haskell existe más de una manera de crear una lista, aquí es donde entran los rangos, un rango de números en Haskell se representa de la siguiente manera:

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

Se ha creado una lista de una manera mas rápida y sencilla. También podemos aplicarle saltos, por ejemplo si queremos una lista de números pares podemos aplicar lo siguiente:

```
Prelude> [2,4..10]
[2,4,6,8,10]
```

Los rangos también se pueden aplicar de manera descendente de la siguiente manera:

```
Prelude> [10..1]
[10,9,8,7,6,5,4,3,2,1]
```

Ademas podemos aplicar lo mismo pero con caracteres, por ejemplo:

```
Prelude> ['a'..'f']
['a','b','c','d','e','f']
```

8. Funciones de listas infinitas

Como en cualquier lenguaje de programación, existen maneras de generar valores de manera continua, y Haskell no es la excepción. A continuación se presenta una tabla con estas funciones:

Función	Descripción	Ejemplo
repeat	Repite de manera indefinida un valor	*Prelude> repeat 7* *[7,7,7,7,7,7…7]*
cycle	Repite de manera indefinida una lista	*Prelude> cycle [1,2,3]* *[1,2,3,1,2,3,…1,2,3]*
replicate	Repite n veces un valor o una lista	*Prelude> replicate 4 7* *[7,7,7,7]*

Para sacarle mayor partido a estas funciones es recomendable utilizar la función take que ya conocemos para evitar problemas, por ejemplo:

```
Prelude> take 4 (repeat 3)
[3,3,3,3]
Prelude> take 4 (cycle “hola”)
 “hola”
```

9. Listas intencionales

Una lista intencional es una lista que hemos visto en los temas anteriores, filtramos qué elementos de las listas queremos o no según las condiciones que buscamos. La estructura de las listas intencionales es la siguiente:

```
[Salida | x <- [Lista de la que filtramos] | Condiciones]
```

A continuación un pequeño ejemplo, esta vez crearemos una lista con los números impares del 1 al 20:

```
Prelude>let lista = [ x | x <-[1..20], x`mod`2 == 1 ]
Prelude> lista
[1,3,5,7,9,11,13,15,17,19]
```

Con listas intencionales podemos hacer muchas cosas, en este caso aplicaremos este tema pero con un poco más de dificultad, tomaremos una lista como parámetro, filtraremos los números impares y además mostraremos un mensaje que diga si cada número es de una cifra o dos respectivamente. Para realizar este ejercicio utilizaremos un archivo .hs como vimos anteriormente, en este caso lo nombraremos haskell.hs

```
cuentaCifras lista = [if x<10 then "una cifra" else "dos cifras"
 | x <- lista,odd x]
```

En esta ocacion utilizamos una función llamada **odd ,** la cual verifica si un valor es imapar y retorna un valor booleano. Ahora cargamos el archivo que contiene nuestra función:

```
Prelude> :l haskell.hs
```

Cargamos la función y le enviamos una lista, en este caso de 1 a 30:

```
*Main> cuentaCifras [1..30]
["una cifra","una cifra","una cifra","una cifra","una
cifra","dos cifras","dos cifras","dos cifras","dos
cifras","dos cifras","dos cifras","dos cifras","dos
cifras","dos cifras","dos cifras"]
```

Al final la función nos devuelve una lista de números impares y además nos dice si estos tienen una o dos cifras respectivamente.

10. Listas Intencionales Dobles

Ya conocemos que es una lista intencional, ahora podemos probar combinar listas intencionales con el fin de comprender un poco mas el que tanto podemos lograr con Haskell.

Primero creamos la lista con la estructura que ya hemos usado solo que esta vez agregaremos otra lista y usaremos un nombre para cada una, en este caso **x** y **y**. La lista de **x** será del 1 al 20 y la lista de **y** será del 1 al 100:

```
Prelude>
let lista = [Salida| x <- [1..20], y <- [1..100],condición]
```

La condición que usaremos será que tomara valores de **x** siempre que sean valores menores a 10 y además tomaremos de **y** los valores que sean múltiplos de 10:

```
Prelude>
let lista = [ | x<-[1..20], y<-[1..100], x<10, y`mod`10== 0]
```

Como salida queremos los valores que resulten de la suma de **x** y **y**:

```
Prelude>
let lista = [x+y| x<-[1..20],y<-[1..100], x<10, y`mod`10== 0]
```

Ya tenemos nuestra lista terminada, ahora ejecutamos y muestra como resultado:

```
Prelude> lista
[11,21,31,41,51,61,71,81,91,101,12,22,32,42,52,62,72,82,92,10
2,13,23,33,43,53,63,73,83,93,103,14,24,34,44,54,64,74,84,94,1
04,15,25,35,45,55,65,75,85,95,105,16,26,36,46,56,66,76,86,96,
106,17,27,37,47,57,67,77,87,97,107,18,28,38,48,58,68,78,88,98
,108,19,29,39,49,59,69,79,89,99,109]
```

Podemos ver como las listas se combinaron ya que primero pasa el 1 que viene de la lista llamada **x** con el 10 de la lista llamada **y** mostrando y así sucesivamente hasta el 9. En total son 90 elementos.

Otro ejemplo que podemos analizar es una función que muestre cuantas vocales tiene una frase.

Creamos un archivo .hs como ya sabemos y aplicamos esta lista intencional

```
mostrarVocales frase = [ letra | letra<-frase, letra `elem`
['a', 'e', 'i', 'o', 'u'] ]
```

Guardamos y ejecutamos, en este caso pasaremos la cadena “Hola mundo” y nos muestra:

```
*Main> mostrarVocales "Hola Mundo"
"oauo"
```

11. Duplas VS Listas

Una lista es una colección de datos de un mismo tipo. En Haskell se declaran con corchetes como ya vimos anteriormente, es decir, [1, 2, 3,4] o [´a´, ´b´, ´c´, ´d´] por ejemplo.

```
Prelude> let lista = [1,2,3,4]
```

Una dupla es una colección de datos que permite almacenar varios tipos de datos sin ningún problema, en Haskell se declaran con paréntesis, es decir, (1,"dos"), en este caso es una dupla que contiene un número y una cadena.

```
Prelude> let dupla = (1,"dos")
```

De momento parece que las duplas tienen una gran ventaja sobre las listas pero tienen un gran detalle, una lista de listas puede contener listas de diferente longitud:

```
Prelude> let superlista = [[1,2],[3,4,5,6],[7,8,9]]
```

En cambio, una lista de duplas debe tener duplas de una misma longitud y además el mismo patrón de tipos, como por ejemplo un valor numérico y una cadena:

```
Prelude> let listaDupla = [(1,"dos"),(3,"cuatro")]
```

12. Funciones de duplas

En esta sección vamos a trabajar con únicamente las duplas, recordando que en listas logramos acceder al dato cualquier posición por medio de la siguiente instrucción:

```
Prelude> let lista = [1,2,3,4]
Prelude> lista !!0
1
```

Ahora, con las duplas esto no funciona, al poder tener solo dos elementos se utilizan solo dos instrucciones las cuales son **fst** y **snd**, funcionan de la siguiente manera:

```
Prelude> let dupla = (1,"dos")
Prelude> fst dupla
1
```

fst muestra el elemento que tenemos en la primera posición y como se pude suponer, **snd** muestra el otro elemento:

```
Prelude> snd dupla
 "dos"
```

13. Listas de Duplas ZIP

Tenemos una nueva función llamada **zip,** la cual funciona de la siguiente manera, recibe dos listas de diferente tipo y genera una lista de duplas con sus elementos.

Poe ejemplo tenemos dos listas, una almacena cuatro cadenas y la otra lista almacena números con decimales:

```
Prelude> let nombres=[“juan”, “iker”, “pedro”, “luis”]
Prelude> let estaturas=[1.78,1.98,1.67,1.80]
```

Aplicamos **zip**:

```
zip nombres estaturas
[(“juan”,1.78),(“iker”1.98),(“pedro”1.67),
(“luis”,1.8)]
```

Adicionalmente, cuando aplicamos **zip** a dos listas de diferente longitud, la salida se ajusta a la lista de longitud más corta. Gracias a esto le podemos dar uso a las listas infinitas, por ejemplo:

```
Prelude> let nombres=[“juan”, “alberto”, “manolo”,
 “luis”]
Prelude> zip [1..] nombres
[(1, “juan”), (2, “alberto”), (3, “manolo”), (4, “luis”)]
```

Se puede apreciar que a pesar de tener una lista infinita, esta se detuvo al contar cuatro elementos en la lista nombres.

14. Comando :t

Este comando es indispensable a la hora de comprender las funciones en Haskell, tanto las prefijas como las de nuestra autoría ya que con este podemos ver los tipos de datos que recibe una función y a su vez que tipo de valor regresa. Incluso funciona para simplemente ver de qué tipo es cualquier parámetro que ingresemos.

```
Prelude> :t "a"
"a" :: [Char]

Prelude> :t "hola"
"hola" :: [Char]

Prelude> :t True
True :: Bool

Prelude> :t 6
6 :: Num t => t

Prelude> :t 6.8
6.8 :: Fractional t => t
```

Este comando también se conoce como firma de una función ya que lo puedes agregar en tus archivos .hs para mejorar la comprensión del mismo.

15. Conversores show y read

Por último, conoceremos dos funciones muy interesantes y útiles, la función **show** y la función **read**.

La función **show** lo que hace es tomar como parámetro cualquier cosa, ya sea número, booleano, lista o dupla y lo convierte a una cadena de texto:

```
Prelude> show 3
"3"
Prelude> show 5.35346
"5.35346"
Prelude> show False
"False"
Prelude> show [1,2,3]
"[1,2,3]"
Prelude> show (1,2,3)
"(1,2,3)"
```

La función **read** es muy interesante, lo que hace es tomar dos parámetros, verifica el tipo del segundo y convierte el primer parámetro a dicho tipo con el fin de poder realizar operaciones:

```
Prelude> read "5.6" + 8.1
13.7
Prelude> read "False" && True
False
Prelude> read "[4,8,2,4]"++[9]
[4,8,2,4,9]
```

Conclusión

A veces es difícil comprender las cosas nuevas, más aun cuando ya estamos acostumbrados a una forma de hacer las cosas, en cuanto al desarrollo de software aplica igual, de momento la programación orientada a objetos ha estado presente en todos lados y de hecho es muy buena pero ha surgido una excelente alternativa que poco a poco está haciéndose camino para ser uno de los lenguajes más populares en el área de desarrollo de software ya que es muy eficiente y no necesita de grandes bloques de código para realizar tareas complejas.

Mi opinión personal sobre este tema es que al inicio fue bastante extraño pero interesante el conocer este lenguaje, realmente como estudiante no esperaba encontrarme con algo asi en un salón de clase, realmente agradezco esta introducción ya que cambio un poco la dinámica de al programación, quizá no logre crear funciones que lograran ser compiladas al primer intento pero realmente logre comprender en base a la experimentación y a los retos de la materia hasta ahora. Le doy el visto bueno a este lenguaje llamado Haskell.

Espero que este manual funcione para que otras personas se introduzcan de una manera amigable a esta forma de desarrollar.

Referencias

wiki.haskell.org (1º de noviembre 2006) Introducción. Obtenido de https://wiki.haskell.org/Es/Introduccion

Jotajotavm (productor). (16 mayo. 2014). Tutoriales Haskell. Obtenido de https://www.youtube.com/playlist?list=PLraIUviMMM3fbHLdBJDmBwcNBZd_1Y_hC