Listas y cadenas

Programación funcional para la física computacional

10 de marzo de 2023

Listas y cadenas

Quizá en este capítulo me alejaré un poco de la ruta tradicional en un curso de programación para física. por lo regular tratamos con números y lo demás queda de lado. Pero realmente es una dualidad: es cierto que las computadoras funcionan con puros números, aún si es una imagen, un sonido, es codificado a números. Pero, por otro lado, estos números deben incluir más información, en código binario ¿cómo incluimos el signo $+$ y $-$? Estos números son leídos como cadenas y se opera en ellos para identificar la codificación.

Veremos la parte de listas y cadenas justo en esta idea, del principio de funcionamiento de una computadora y que es una de los bloques básicos para su uso.

Tipos de listas y su creación

Como su nombre lo dice, una lista es un conjunto ordenado de objetos, no necesariamente del mismo tipo, a los que se puede acceder por una referencia y un índice. En los lenguajes de programación suele ser uno de los objetos más versátiles (como ya vimos en nuestro ejemplo en $python$ para la criba de Eratostenes) y que son base para otro tipo de estructuras, como los arreglos (a diferencia de los arreglos las reglas de operación sobre listas son más relajadas[1]).

Dependerá de cada aplicación, de cada programa y cada programador como le conviene formar listas. Como ya han de estar acostumbrados, en esto de la programación hay mucho de prueba y error, a veces conviene definir una lista de alguna forma y en el desarrollo del programa nos damos cuenta que nos conviene más otra forma. Quizá esa es la parte desafiante de esta labor, no siempre se disfruta (seguro le han pegado más de una vez a su pantalla o teclado), pero lo hace interesante.

Aquí requeriremos de un *pull-up* al estilo de los *soundsystems* de *reggae*. Vamos a regresarnos un poco a cosas básicas de Haskell ante de entrar a listas.

Ya hemos mostrado algunas de las características más curiosas de Haskell, como sumar en una sola linea y con pocos caracteres los números del 1 al 100

```
sum [1..100]
```

La función sum y la forma de definir la lista de los primeros 100 naturales está dentro de las liberrías de Haskell. Pero sería bueno saber como fue definida esta función, lo cual mostramos a continuación (cmbiando el nombre para notar que no se hace trampa):

[1] Los arreglos ocupan menos memoria, ya que el acceso es secuencial, eso provoca que sea más lento operar sobre sus elementos pero más rápoido acceder a ellos. En cambio la lista tiene apuntadores al inicio y final de la lista, los hace más pesados en espacio, pero es mucho más rápido crear y agregar datos, aunque más lento operar en ellos.

```
sumalista [] = 0
sumalista (n:ns) = n + sumalista ns
```

Es importante notar que esto debe estar escrito en un archivo $*.hs$, al menos el compilador de ghc no comprende si esto se le da en el $idle$. Notemos que la función es definida por casos, cuando la lista es vacía y luego de manera recursiva sobre los elementos. El tipo de los argumentos y la salida es inferido de esta sintaxis, pero en ciertos casos se le puede dar de forma explícita.

Prueba definir una lista en Haskell $a = [1, 2, 3, 4]$ y aplicar la función. Ahora define una lista de caracteres $b = [\prime a\prime, \prime b\prime, \prime c\prime, \prime d\prime]$ y trata de sumarla ¿qué obtienes?

Algunas cosas básicas de *Haskell*

Antes de continuar debemos tener en cuenta como se hace la aplicación de funciones en Haskell. En el cuadro 1 comparamos como se haría en la notación matemática tradicional, y cómo se escribiría en Haskell. Recuerden la importancia de los espacios para Haskell

Matemática	Haskell
$f(x)$	$f\ x$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(y)$	$f\ x\ *\ g\ y$

Cuadro 1: Aplicación de funciones en *Haskell*

Pero para definir una función como la anterior llamada $sumalista$ debemos definirla dentro de un *script*, si intentamos hacerlo desde el *idle* de $Haskell$ nos dará errores. Para poder escribir un script debemos seguir las indicaciones:

- Editar en su editor preferido guardando como $nombre.hs$.
- Cargar desde el $idle$ con : $l\ nombre.hs$.
- Llamar las funciones como si las hubiéramos definido antes, sólo lamar por el nombre y dar los argumentos (ej. $sumalista\ [1, 2, 3, 4]$).
- Si se cambia algo en el $script\ Haskell$ no lo detecta inmediatamente, deben guardar el archivo y recargar con : $reload$.

Unos comandos útiles sobre $scripts$ en $Haskell$ se pueden ver en el cuadro 2. El comando : $type\ expr$ da información sobre el dominio y el rano de la expresión que se da, es útil para saber en donde está definida la función y evitar errores. Si se ha especificado el editor de texto se puede editar el $script$ desde el mismo $idle$.

Algunas cadenas exclusivas del lenguaje (que no pueden ser usadas para nombrar variables o funciones):
case class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where

Otra parte importante son los comentarios, de los que hay dos tipos:

Comando	Accion
: $load\ nombre$	Cargar $script$
: $reload$	Recargar script
: $edit\ nombre$	Cambiar nombre del $script$
: $edit$	Editar el $script$
: $type\ expr$	Mostrar el tipo de la expresión
:?	Mostrar los comandos
: q	Salir de $GHCi$

Cuadro 2: Comandos del $idle$ de $Haskell$ para $scripts$.

- Comentarios ordinarios, se hacen en línea y su longitud máxima es hasta donde termina esa línea, empiezan con $--$.
- Comentarios anidados, más de una línea, empiezan con $\{-$ y terminan con $-\}$

Regresando a operar sobre listas

Regresemos a trabajar con listas en $Haskell$. Definamos una función más interesante, la que ordena una lista:

```
ordenar [] = []
ordenar (x:xs) = ordenar chico ++ [x] ++ ordenar grande
  where
    chico =[a | a<-xs, a<=x]
    grande = [b | b<- xs, b>x]
```

Notemos que de nuevo tenemos una definición recursiva, pero ahora es un poco más complicada. Esta función ordena la lista de objetos, haciendo uso del diseño de *arriba-abajo* (¿recuerdan de la introducción?) hacemos uso de cosas que definimos después, $chico$ y $grande$. Para ver que esto funciona definan una lista de números desordenada y una de letras igual desordenada y aplícala a la función. $where$ es utilizado para definiciones locales, sirven para esta función y ya.

Veamos como opera ordenar. Para el caso de la lista de un solo elemento:

$$\begin{aligned} ordenar[x] = & \\ & = ordenar[] + +[x] + + ordenar[] \\ & = [] + +[x] + +[] \\ & = [x] \end{aligned}$$

Para una lista con más elementos

$$\begin{aligned} ordenar[3,5,4,2,1] = & \\ & = ordenar[1,2] + +[3] + +[4,5] \\ & = [] + +[1] + +[2] + +[3] + +[4] + +[5] \\ & = [1,2,3,4,5] \end{aligned}$$

Ahora sí, regresando al hilo de ideas de antes, y haciendo de manera más formal cosas que ya hemos hecho, en Haskell podemos definir una lista como:

```
let miLista = []
```

Que es una lista vacía. Aquí let es similar a $where$ salvo que no es una definición local, hasta este momento no la hemos puesto, lo que implica que lo que definimos es por default un $where$ para nuestro código. Hay que ser cuidadosos con estas definiciones para no tener errores. Las variables siempre deben empèzar con letras minúsculas, de lo contrario dará un error.

Como ya vimos rápido hay características especiales para listas en Haskell, ya vimos como definir una lista sin necesidad de dar todos los elementos, pero además podemos usar comprensión de listas:

```
let a = [1,2,3,4]
let b = [1..10]
let a = [x*2 | x<- a]
```

Por otro lado en $python$ también ya vimos como crear listas

$$Milista = [1, 2, 3, 4]$$

Para crear una lista de manera perezosa, como en $Haskell$:

$$b = list(range(1, 10))$$

Y para la comprensión de listas, como en $Haskell$

$$c = [a * 2 \ for \ a \ in \ range(1, 5)].$$

No es exactamente como definir un conjunto, realmente depende de una estructura de programación que es el for. Tiene el mismo resultado pero a nivel conceptual no está en un nivel tan básico en la abstracción, como sí lo está en $Haskell$.

Algunas operaciones sobre listas en $Haskell$:

$$\begin{aligned} a &= [1, 2, 3, 4, 5] \\ head\ a \\ tail\ a \\ init\ a \\ last\ a \\ a!!2 \\ take\ 3\ a \\ drop\ 3\ a \end{aligned}$$

$$\begin{aligned} length\ a \\ sum\ a \\ product\ a \\ [1, 2, 3] + +[4, 5] \\ reverse\ a \\ null\ a \\ minimum\ a \\ head\ [] \ (¿?) \end{aligned}$$

A partir de estas funciones ya por $default$ en $Haskell$ podemos definir nuevas por nuestra cuenta:

```
prome = \x -> sum x `div` length x
```

Algunas de estas mismas operaciones sobre cadenas se pueden hacer en $python$:

$$
\begin{aligned}
a &= [1,2,3,4,5] \\
a[0] & \\
a[1:] & \\
a[:-1] & \\
a[-1] & \\
a[2] & \\
a[:3] &
\end{aligned}
\qquad\qquad
\begin{aligned}
a[:-3] & \\
a[-3:] & \\
len\ a & \\
not\ a & \\
min\ a & \\
max\ a & \\
[1,2]+[3,4] & \\
sum\ a & \\
b &= [] \\
b[0]\ (¿?) &
\end{aligned}
$$

Y de la misma manera podemos definir nuevas funciones sobre listas a partir de estas listadas:

$$
\begin{aligned}
prome &= lambda\ x : sum(x) // len(x) \\
inver &= lambda\ x : x[::-1]
\end{aligned}
$$

Numerales de Church

Esta sección debería estar en el capítulo anterior, pero por aquello de los salones divididos lo di en medio del tema de cadenas. Si en algún momento actualizo estas notas, reacomodaré esto.

Ya hablamos de como a partir de cálculo λ podemos generalizar cualquier función, pero ¿cómo se traduce eso ahora a un lenguaje de programación completo? ¿Cómo construimos todo lo que se necesita? Pues veamos un poco.

Lo primero que necesitamos para operar son los números naturales, se definen a partir de los numerales de Church:

$$
\begin{aligned}
\overline{0} &\stackrel{def}{=} \lambda f.\lambda x.x \\
\overline{1} &\stackrel{def}{=} \lambda f.\lambda x.fx \\
\overline{2} &\stackrel{def}{=} \lambda f.\lambda x.f(fx) \\
\overline{3} &\stackrel{def}{=} \lambda f.\lambda x.f(f(fx)) \\
&\vdots \\
\overline{n} &\stackrel{def}{=} \lambda f.\lambda x.f^n x
\end{aligned}
$$

Como pueden ver construimos el cero como una función inicial que no se aplica, sólo nos regresa x, el valor 0 y a partir de ahí de manera recursiva se construyen los números naturales aplicando la función al anterior. Esta función la podemos pensar como nuestra ya conocida función sucesor.

Pero ¿cómo se construye esa función sucesor? Lo vemos a continuación:

$$s \overset{def}{=} \lambda m.\lambda f.\lambda x.f(mfx).$$

Para entender un poco como es que esta es la definición de la función sucesor probemos aplicarla a $\overline{n}$ para obtener el siguiente natural $\overline{n+1}$:

$$\begin{aligned}
s\overline{n} =& (\lambda m.\lambda f.\lambda x.f(mfx))(\lambda f.\lambda x.f^n x) \\
\xrightarrow{\alpha}& (\lambda m.\lambda g.\lambda y.g(mgy))(\lambda f.\lambda x.f^n x), \text{ haciendo reducción } \alpha \\
\xrightarrow{\beta}& \lambda g.\lambda y.g((\lambda f.\lambda x.f^n x)gy), \text{ haciendo reducción } \beta \\
\xrightarrow{\beta}& \lambda g.\lambda y.g((\lambda x.g^n x)y) \\
\xrightarrow{\beta}& \lambda g.\lambda y.g(g^n y) \\
=& \lambda g.\lambda y.g^{n+1} y \\
\xrightarrow{\alpha}& \lambda f.\lambda x.f^{n+1} x \\
=& \overline{n+1}.
\end{aligned}$$

Noten que la reducción α solo fue cambiar nombres para evitar confusiones y la reducción β es sustituir los λ términos para aplicar las funciones.

De esta forma podemos construir funciones similares a las funciones recursivas μ, y comprobar que así como esas funciones se traducen a una máquina de Turing, lo mismo sucede para el cálculo λ. No me detengo mucho en construirlas cada una, pues quizá cambia un poco la notación, pero espero en una versión posterior de las notas ponerlo más claro.

$$\begin{aligned}
V_{bool} &\overset{def}{=} \lambda xy.x \\
F_{bool} &\overset{def}{=} \lambda xy.y \\
[M,N] &\overset{def}{=} \lambda z.zMN \\
\overline{0} &\overset{def}{=} \lambda x.x \\
\overline{n+1} &\overset{def}{=} [F, \overline{n}] \\
S &\overset{def}{=} \lambda x.[F,x] \\
P &\overset{def}{=} \lambda x.xF \\
C &\overset{def}{=} \lambda x.\overline{0} \\
Cero &\overset{def}{=} \lambda x.xV \\
\pi_i^n &\overset{def}{=} \lambda x_1, ..., x_n.x_i \\
Y &\overset{def}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))
\end{aligned}$$

Lo importante, y en lo que me detendré un poco más es el último término, llamado el **combinador Y**, en ese término es donde se encierra la recursión para todo el cálculo λ. Es uin término importante aunque se vea un poco horrible. Lo relevante es notar que es la aplicación del λ término $(\lambda x.f(xx))$ a si mismo. Si realizan la β reducción se

darán cuenta de que llegan exactamente al mismo término. Así hasta el infinito, es la recursión.

Pero si no les impresiona tanto quizá prefieran ver la explicación de, y me pongo de pie, Graham Hutton, en un video de youtube:

`https://www.youtube.com/watch?v=9T8A89jgeTI`

Regresando a operar sobre listas y cadenas

¿Cuáles son las operaciones básicas para transformar listas, o al menos las que más usamos? Hemos visto como definirlas, obtener información de ellas, pero ahora vamos a meterles mano. Listo las que me parecen las tareas que más solemos usar en listas

- Concatenación ($python$: $+$, $haskell$: $++$)
- Repetición
- Pertenencia
- Iteración
- Editar

Ahora veamos como se hace en cada uno de los lenguajes de programación en los que nos hemos centrado para este curso.

En el caso de Haskell:

$let\ a\ = [1,2,3,4,5]$, definimos una lista
$let\ b\ = take\ 3\ a$, una nueva sublista de los primeros tres miembros
$let\ c\ = filter\ odd\ a$, sólo toma los miembros impares
$let\ d\ = zip\ a\ ['a','b','c','d','e']$, crea eneadas

Debemos notar que, como hemos mencionado, los datos en Haskell son inmutables. Si se aplica $take\ 3\ a$ desde el *idle* de haskell nos devuelve los primeros tres miembros de la lista a, **pero no los guarda en ningún lado y mucho menos altera la lista** a. Para poder guardarlos es que se asignan a nuevas listas b, c y d en cada caso.

En cambio en python, los datos no son inmutables y en caso de usar las funciones *punto*[2] las listas serán alteradas, como sucedería al aplicar:

$$\begin{aligned} a =& [1,2,3,4,5] \\ a.reverse&() \end{aligned}$$

Si se vuelve a llamar a a ahora se tiene la cadena invertida y la cadena original se ha perdido. Para evitar esto es mejor definir una nueva función *no-punto* a partir de cálculo λ (de hecho, una función *curriada*)

$$\begin{aligned} revertir\ &= lambda\ x:\ x[::-1] \\ b\ &= revertir(a) \end{aligned}$$

[2]Es decir, las funciones que se aplican a las lista a de la forma $a.funcion()$, como se suele hacer en programación orientada a objetos.

Pueden probar que aplicarla deja sin alterar a la lista a y genera una nueva b. A final de cuentas, aunque python puede tener estos problema de valores mutables, es posible corregirlo al hacerlo en cálculo λ, y esa es la finalidad de este curso.

Otras estructuras similares o derivadas

De las listas se derivan otras estructuras útiles en diversas tareas. Una de esas estructuras son los **diccionarios**.los diccionarios son listas en las que el acceso a los miembros es por medio de una *llave* especificada en la misma definición. En python se define como:

$$dict = \{\text{“Uno”} : 1, \text{“Dos”} : 2, \text{“Tres”} : 3\}$$
$$print(dict[\text{“Uno”}])$$

En Haskell

```
import qualified Data.Map as M
  let myDict = M.fromList[("First", 1), ("Second", 2),
                          ("Third", 3)]
```

Necesitamos importar una nueva librería. Tal parece que para Haskell la definición de listas es tan dinámica que no es necesaria desde raíz la definición de diccionarios, pero siempre se puede agregar una librería extra.

Otra estructura que puede ser útil son los **conjuntos**, un tanto en el sentido matemático. En python:

$$conj = frozenset([1, 2, 3, 4, 5])$$

En haskell

```
import Data.Set as Set
  let mySet = Set.fromList[1, 2, 3, 4, 5, 6]
```

Solo lo menciono de rápido, en caso de necesitarlas regresaremos a ellas, pero la idea será armar los programas para aplicación en física desde las partes más básicas.

Cadenas

Y finalmente llegamos al momento estrella, al que hemos esperado todos estos días, el conocimiento final que nos dará la clave para toda nuestra labor como programadores, por si queremos entrar a la cosa esa de datos y la otra cosa de datos, y lo demás de datos: **cadenas**.

Las cadenas son listas de caracteres, fin. Todo lo que se puede hacer en listas, se puede hacer para cadenas, en verdad no hay más que ver.

Solo para ver un ejemplo, comprobemos un palíndromo en Haskell

$$let\ a\ = \text{anita lava la tina}$$
$$let\ b\ = reverse\ a$$

Listo, ya vimos cadenas.

Referencias

[1] Mueller, John P. “Functional programming (for dummies)” John Willey & Sons Inc. (2019).

[2] Hutton, Graham. “Programming in Haskell” Cambridge University Press, 2a edición (2016).

[3] Kozen, Dexter C. “Automata and Computability” Springer (1997)