Programación Declarativa, 2021-1
Nota de clase 12: Programas Interactivos*

Manuel Soto Romero

7 de diciembre de 2020
Facultad de Ciencias UNAM

Hasta la nota pasada, hemos escrito únicamente programas que no interactúan directamente con el usuario, debido principalmente a que sólo se ha hecho a través del intérprete: el usuario proporciona una entrada y el programa la evalúa para devolver un resultado.

Consideraremos *programa interactivo* a aquellos programas que necesitan de información continua por parte del usuario para poder ejecutarse, además de desplegar los datos de salida en un formato más adecuado para la interacción con *humanos*, lo cual nos permite mejorar la *experiencia del usuario*[1] [6]. Por ejemplo, se pueden proporcionar mensajes para solicitar datos o incluso para informar que el programa se encuentra realizando algún tipo de proceso.

Debido a la pureza del lenguaje, en Haskell puede llegar a se inimaginable modelar un programa interactivo, pues en la programación funcional pura, un programa consiste de un conjunto de funciones que bajo la misma entrada, siempre producen el mismo resultado, sin importar cuántas veces sean ejecutadas, es decir, no hay efectos secundarios y se garantiza que se cumple el *principio de trasnparencia referencial*.

En esta nota se presentan los conceptos y las herramientas necesarias para definir programas interactivos, sin dejar de lado la pureza del lenguaje. Esto permite al estudiante escribir programas especializados para interactuar con el usuario, ya sea desde la entrada y salida estándar de datos o mediante la escritura y lectura de archivos para garantizar la persistencia de datos. De la misma forma se ilustra cómo manejar los distintos tipos de excepciones que estos procesos pudieran llegar a generar.

12.1. Entrada y salida de datos

Para definir un *programa interactivo* en Haskell, se definen funciones que toman un estado del *mundo real* como entrada y devuelven un estado modificado del mismo como resultado. Para representar la noción de mundo real, se define el tipo de dato `MundoReal` cuyos valores representan algún estado del mundo real, mismo que puede usarse para obtener algún tipo de dato del usuario o ser modificado[6].

La noción de programa interactivo se da a partir de la definición de una función de la forma `MundoReal` → `MundoReal` llamada `IO`, esto es [6]:

```
type IO = MundoReal → MundoReal
```

*Basada en *Manual de Prácticas para la Asignatura de Programación Declarativa.*
[1] *User Experience*

Todo programa interactivo, necesita además, regresar un valor como resultado, no basta con generar cambios en el estado actual del mundo real. De esta forma, se parametriza la función anterior para que incluya un valor de regreso en su salida[6]:

```
type IO a = MundoReal → (a, MundoReal)
```

Acciones

A las expresiones que forman parte del tipo de dato `IO` se les conoce como *acciones*. Por ejemplo, `IO Char` es el tipo de acciones que devuelven un carácter como resultado, mientras que `IO ()` es el tipo de acciones que regresan la tupla vacía (en otros lenguajes de programación como Java sería equivalente a `void`). Algunas acciones básicas[6]:

`getChar` Lee un carácter del teclado, lo muestra en pantalla y regresa ese carácter como resultado.

```
getChar :: IO Char
```

`putChar` Escribe un carácter en pantalla y no regresa ningún valor.

```
putChar :: Char -> IO ()
```

`return` Regresa un valor, es una función *polimórfica*.

```
return :: a -> IO a
```

Secuencias

Un programa interactivo se forma a partir de la ejecución de distintas acciones en secuencia. Para combinar acciones se usa el *operador de secuencia* que ejecuta una acción después de otra, el estado modificado resultante de la primera acción se toma que como argumento de entrada de la segunda acción y así en lo sucesivo[6].

La firma del operador de secuencia es:

```
(>>=) :: IO a → (a → IO b) → IO b
```

Dada una acción de tipo `a`, y una función que toma como parámetro el resultado de dicha acción, se obtiene una acción de tipo `b`. Por ejemplo, en el Código 1, se pide un número al usuario y lo muestra en pantalla. La función `getChar` espera un valor del usuario y regresa ese valor como resultado. En la ejecución del mismo código, se pasa el valor `5` y regresa una acción de tipo `Char (IO Char)`, el resultado de esta acción se pasa como parámetro de forma secuencial a la función anónima `\x -> return x` que regresa el valor de tipo `Char` como salida.

```
prueba :: IO Char
prueba = getChar >>= (\x -> return x)
```

Código 1: Función que pide un número al usuario y lo muestra en pantalla

```
*Ejemplo> prueba
5'5'
```

Esta forma de escribir programas de forma secuencial, puede volverse algo tediosa a la larga, pues se deben definir distintas funciones para posteriormente pasarlas como argumento al operador de secuencia. Para evitar esto, se provee una versión *endulzada*[2] mediante la primitiva `do`[6].

Empleando la primitiva do, la función del Código 1 se modifica como se muestra en el Código 2.

```
prueba :: IO Char
prueba = do x <- getChar
            return x
```

Código 2: Función que pide un número al usuario y lo muestra en pantalla usando la primitiva do

```
*Ejemplo> prueba
5'5'
```

Acciones derivadas

A partir de las tres acciones básicas descritas anteriormente y el operador de secuencia, se definen algunas acciones derivadas[6]:

`getLine` Lee una cadena de caracteres del teclado.

```
getLine :: IO String
```

`putStr` Escribe una cadena de caracteres en pantalla.

```
putStr :: String -> IO ()
```

`putStrLn` Escribe una cadena de caracteres en pantalla, moviéndose hacia la siguiente línea.

```
putStrLn :: String -> IO ()
```

[2]Con azúcar sintáctica.

Ejemplo 12.1. El Código 3 muestra un programa que pide una cadena al usuario y la convierte en su respectiva versión usando el prefijo correspondiente del lenguaje Efe. Las líneas 41 a 44 muestran el uso de acciones de entrada y salida y su combinación mediante la primitiva `do`.

```
-- ------------------------------------------------------------------ --
-- Programa que pide una cadena al usuario y la convierte en su       --
-- respectiva versión en Efe. Esto es, intercambia las vocales        --
-- como sigue (incluye mayúsculas):                                   --
--                                                                    --
-- a -> afa                                                           --
-- e -> efe                                                           --
-- i -> ifi                                                           --
-- o -> ofo                                                           --
-- u -> ufu                                                           --
-- ------------------------------------------------------------------ --
module Efe where

-- ------------------------------------------------------------------ --
-- Función que toma un caracter y lo convierte a su versión en Efe.   --
-- ------------------------------------------------------------------ --
procesaChar :: Char -> String
procesaChar 'a' = "afa"
procesaChar 'e' = "efe"
procesaChar 'i' = "ifi"
procesaChar 'o' = "ofo"
procesaChar 'u' = "ufu"
procesaChar 'A' = "Afa"
procesaChar 'E' = "Efe"
procesaChar 'I' = "Ifi"
procesaChar 'O' = "Ofo"
procesaChar 'U' = "Ufu"
procesaChar c = [c]

-- ------------------------------------------------------------------ --
-- Función que toma una cadena y la convierte en su respectiva        --
-- versión en Efe.                                                    --
-- ------------------------------------------------------------------ --
procesa :: String -> String
procesa palabra = foldr (++) [] (map (procesaChar) palabra)

-- ------------------------------------------------------------------ --
-- Función que pide una cadena al usuario y regresa la cadena         --
-- procesada.                                                         --
-- ------------------------------------------------------------------ --
pideDatos :: IO ()
pideDatos = do putStr "Introduce una cadena: "
               entrada <- getLine
```

```
                putStrLn $ procesa entrada

-- ------------------------------------------------------------------ --
-- Función encargada de llevar la ejecución de todo el programa.      --
-- ------------------------------------------------------------------ --
run = pideDatos
```

Código 3: Programa interactivo para convertir una cadena a efe

```
*Efe> run
Introduce una cadena: Anita lava la tina
Afanifitafa lafavafa lafa tifinafa
```

12.2. Introducción a las mónadas

El tipo de dato `IO` usado para interactuar con el usuario de forma secuencial a través de la impresión y entrada de datos desde la terminal, es parte de una clase de tipos llamada `Monad` que permite definir *mónadas*. Una mónada permite estructurar un programa en términos de acciones y secuencias[5]. La clase de tipos `Monad` define tipos parámetrizados (familias de la forma `* -> * -> ... -> *`). Para que un tipo forme parte de esta clase, se deben definir las funciones `>>=` y `return` para éste, las mismas que se definen para la mónada de entrada y salida `IO`.

La firma de estas dos funciones es:

```
(>>=) :: m a → (a → m b) → m b

return :: a → m a
```

Observación 12.1. No es estrictamente necesario que una mónada implemente la clase `Monad` para serlo, sin embargo, se recomienda hacerlo, pues Haskell por sí mismo proporciona al programador diversos mecanismos que permiten escribir código de forma legible y robusta, mediante el uso, por ejemplo, de la primitiva `do`[5].

A continuación, se muestran algunos ejemplos de mónadas en Haskell.

La mónada identidad

Esta es una de las mónadas más simples, aplica una función a un parámetro de entrada sin modificar de manera alguna a éste[5]. En el Código 4 se muestra la definición de esta mónada.

```
data Identity a = Identity a

instance Monad Identity where
  (Identity x) >>= f = f x
  return a = Identity a
```

Código 4: Definición de la mónada identidad

La mónada lista

La mónada lista permite ejecutar un conjunto de acciones de forma no determinista mediante la aplicación de distintas operaciones a cada valor posible. Es útil pues al ser no determinista, permite explorar todos los posibles caminos de un cómputo[5]. Las listas por comprensión son una forma de azúcar sintáctica de esta mónada. En el Código 5 se muestra la definición de esta mónada.

```
instance Monad [] where
  m >> f = concatMap f m
  return x = [x]
```

Código 5: Definición de la mónada lista

Ejemplo 12.2. El Código 6 muestra la comparación del uso de una lista por comprensión y otra definida mediante la primitiva `do`. La ejecución del código muestra los mismos resultados para ambas listas.

```
l1 = [1..10]

comprension :: [(Int, Int)]
comprension = [(x,y) | x <- l1, y <- l1, x + y == 7]

monada :: [(Int, Int)]
monada = do x <- l1
            y <- l1
            if (x + y == 7) then return (x,y) else []
```

Código 6: Comparación entre listas por comprensión y mónadas

```
Ejemplo> comprension
[(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)]
Ejemplo> monada
[(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)]
```

La mónada continuación

Una *continuación* es una función de orden superior que representa el resto de la ejecución de un programa en un punto específico[6]. El *estilo de paso de continuaciones*[3], permite definir cómputos como una secuencia de continuaciones anidadas que finaliza con la aplicación de una continuación que representa el final del cómputo (usualmente la función identidad). El uso de continuaciones permite interrumpir o restaurar un programa en un punto intermedio de forma secuencial, debido a esto, el estilo de paso de continuaciones se adapta de forma sencilla a la definición de mónada de Haskell[5]. En el Código 7 se muestra la definición de esta mónada.

```
data Cont r a = Cont {runCont :: ((a -> r) -> r)

instance Monad (Cont r) where
  (Cont c) >>= f = Cont $ (\k -> c (\a -> runCont (f a) k)
  return a = Cont $ (\k -> k a)
```

Código 7: Definición de la mónada continuación

Ejemplo 12.3. El Código 8 muestra el uso de la mónada continuación para calcular la longitud de una lista[5]. La función longitud captura la continuación actual (línea 4), en este caso el único cálculo pendiente es calcular la longitud de la lista correspondiente. Este cálculo no se realiza hasta que la continuación es aplicada (línea 8). La ejecución del código muestra el resultado después de aplicar la continuación.

```
-- Función que dada una lista, regresa una continuación que al
-- aplicarse, calcula la longitud de la lista.
longitud :: [a] -> Cont r Int
longitud l = return (length l)

-- Llamada principal al programa.
-- Aplica la continuación.
main = do runCont (longitud [1,7,2,9]) print
```

Código 8: Uso de la mónada continuación

```
Ejemplo> main
4
```

En el Código 8, se siguen los siguientes pasos para generar la salida correspondiente:

[3] *Continuation Passing Style*

1. La llamada a la función longitud regresa la continuación:

   `Cont $ \k -> k (length [1,7,2,9])`

   Una función que recibe un parámetro `k` y aplica dicho parámetro a `length [1,7,2,9]`.

2. Al ejecutar main se hace una llamada a `runCont` que recibe una continuación (la llamada a `longitud`) y una función, en este caso, se ejecuta la continuación de la siguiente forma:

   ```
   > runCont Cont $ \k -> k (length [1,7,2,9]) print
   > (\k -> k (length [1,7,2,9])) $ print
   > print (length [1,7,2,9])
   > print 4
   4
   ```

12.3. Lectura y escritura de archivos

En las secciones anteriores se revisó la interacción con el usuario mediante impresiones en pantalla y mediante la entrada de datos desde el teclado, sin embargo, hay ocasiones en las que resulta útil almacenar o consultar datos desde archivos para garantizar la persistencia de datos de un programa, esto es, que sin importar que el programa finalice, este podrá recuperar datos que el usuario haya proporcionado o que el programa mismo haya generado, cuantas veces sea necesario.

Para este tipo de programas, HASKELL provee de primitivas que permiten escribir y leer archivos de forma sencilla, conservando la pureza del lenguaje sin generar efectos secundarios. Para ejemplificar este proceso, en el Código 9 se realiza la lectura el archivo `pokemones.txt` con los datos de algunos Pokemones:

`pokemones.txt`

```
1 | Bulbasaur | Planta, Veneno | A Bulbasaur es fácil verle
echándose una siesta al sol. La semilla que tiene en el lomo
va creciendo cada vez más a medida que absorbe los rayos del sol.
2 | Ivysaur | Planta, Veneno | Este Pokémon lleva un bulbo en
el lomo y, para poder con su peso, tiene unas patas y un tronco
gruesos y fuertes. Si empieza a pasar más tiempo al sol, será
porque el bulbo está a punto de hacerse una flor grande.
3 | Venusaur | Planta, Veneno | Venusaur tiene una flor enorme
en el lomo que, según parece, adquiere unos colores muy vivos
si está bien nutrido y le da mucho el sol. El aroma
delicado de la flor tiene un efecto relajante en el ánimo de
las personas.
```

```haskell
import System.IO

-- Función que abre el archivo "pokemones.txt", muestra su contenido
-- en pantalla y cierra el archivo.
lectorDatos :: IO ()
lectorDatos = do contenedor <- openFile "pokemones.txt" ReadMode
                 contenido <- hGetContents contenedor
                 putStr contenido
                 hClose contenedor

-- Función principal, ejecuta las funciones correspondientes en
-- secuencia.
main :: IO ()
main = lectorDatos
```

Código 9: Lectura de archivos

```
Ejemplo> main
1 | Bulbasaur | Planta, Veneno | A Bulbasaur es fácil verle echándose una siesta al
sol. La semilla que tiene en el lomo va creciendo cada vez más a medida que absorbe
los rayos del sol.
2 | Ivysaur | Planta, Veneno | Este Pokémon lleva un bulbo en el lomo y, para poder
con su peso, tiene unas patas y un tronco gruesos y fuertes. Si empieza a pasar más
tiempo al sol, será porque el bulbo está a punto de hacerse una flor grande.
3 | Venusaur | Planta, Veneno | Venusaur tiene una flor enorme en el lomo que, según
parece, adquiere unos colores muy vivos si está bien nutrido y le da mucho el sol.
El aroma delicado de la flor tiene un efecto relajante en el ánimo de las personas.
```

Del Código 9 se puede destacar[5]:

- La línea 1 (`import System.IO`) indica que se hace uso de la biblioteca `System.IO` que permite hacer uso de algunas funciones de entrada y salida del sistema[4].
- En la línea 6, la función `openFile :: FilePath -> IOMode -> IO Handle` se compone de varias partes:
  - `FilePath` es la ruta dónde se encuentra el archivo a procesar. Se representa mediante una cadena, es decir, es un sinónimo de tipo.
    `type FilePath = String`
  - `IOMode` es una enumeración[5] que representa el modo de entrada o salida de la función.
    `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`
  - `IO Handle` es un *contenedor*[6] que almacena datos de forma temporal para ser procesados más adelante.

[4]Consultar `http://hackage.haskell.org/package/base-4.11.1.0/docs/System-IO.html`
[5]Se llama así a aquellos tipos de datos formados únicamente por constructores de datos sin parámetros.
[6]También llamado búfer de datos.

- La función `hGetContents :: Handle -> IO String` (línea 7) toma un contenedor y regresa una acción que tiene como resultado el contenido del archivo. Esta función sigue un régimen de evaluación perezosa, esto es, no se lee todo el contenido de un archivo de golpe, sino conforme se vaya necesitando.
- La función `hClose :: Handle -> IO ()` (línea 9) toma un contenedor y regresa una acción que cierra el archivo. Todo archivo debe cerrarse después de ser abierto y procesado.

Funciones sobre contenedores

A continuación se listan algunas funciones útiles para trabajar con contenedores[5].

`hGetLine` Lee una línea del archivo que almacena el contenedor.

```
hGetLine :: Handle -> IO String
```

`hPutStr` Escribe una cadena en el contenedor.

```
hPutStr :: Handle -> String -> IO ()
```

`hPutStrLn` Escribe una cadena y da un salto de línea en el contenedor.

```
hPutStrLn :: Handle -> String -> IO ()
```

`hGetChar` Lee un carácter del archivo que almacena el contenedor.

```
hGetChar :: Handle -> IO Char
```

Funciones sobre archivos

Además de `openFile`, en HASKELL se tienen otras funciones que facilitan la lectura y escritura de archivos. A continuación se listan las más usadas[5].

`readFile` Toma la ruta del archivo, crea un contenedor y envuelve todo el contenido del mismo en una acción de entrada y salida. Esta función, cierra automáticamente el contenedor asociado.

```
readFile :: FilePath -> IO String
```

`writeFile` Toma la ruta del archivo, y una cadena que escribir en el mismo y devuelve una acción de entrada y salida que se encargará de escribir la cadena en el archivo. Si el archivo ya existe, lo sobreescribe.

```
writeFile :: FilePath -> String -> IO ()
```

`appendFile` Toma la ruta del archivo, y una cadena que escribir en el mismo y devuelve una acción de entrada y salida que se encargará de escribir la cadena en el archivo. A diferencia de `writeFile`, esta función nunca sobreescribe el contenido de un archivo, siempre concatena la cadena pasada como parámetro al final del mismo.

```
appendFile :: FilePath -> String -> IO ()
```

Ejemplo 12.4. El Código 10 muestra un programa interactivo para dar de alta Pokemones y consultarlos.

- La función `cargaBase` (líneas 21 a 23) carga el archivo correspondiente y lo regresa como una lista de listas de cadenas. Se separa primero el contenido por saltos de línea y posteriormente por barras verticales.
- La función `construye` (líneas 28 a 31) reconstruye la información de cada pokémon para mostrar sus datos.
- La función `busca` (líneas 36 a 41) busca una cadena en una lista de listas, esto es, verifica que en cada una de las listas exista una cadena de acuerdo al criterio solicitado.
- La función `despliega` (líneas 47 a 50) muestra los resultados finales, esto es los datos de cada pokémon.
- La función `selecciona` (líneas 56 a 64) integra las funciones busca y despliega. Busca el pokémon correspondiente en la base, en caso de encontrarlo lo despliega, en caso contrario reporta un error.
- La función `nuevoPokemon` (líneas 70 a 80) construye un nuevo pokémon (lo guarda en la base) de acuerdo a los datos que proporciona el usuario.
- La función `muestraMenu` (líneas 89 a 96) muestra el menú de opciones al usuario.
- La función `procesa`(línea 101 a 105) procesa las opciones dadas por el usuario.
- La función `main` (líneas 110 a 116) ejecuta todo el programa como muestra la ejecución correspondiente.

```
-- ------------------------------------------------------------------------ --
-- Progama que permite consultar o modificar los datos sobre distintos      --
-- pokemones.                                                               --
-- ------------------------------------------------------------------------ --
module Pokedex where

import Data.List
import Data.List.Split
import System.IO

-- Archivo que almacena los datos de los pokemones.
baseDatos = "pokemones.txt"
-- Sinónomo para los criterios de búsqueda.
type Criterio = String
-- Sinónimo para los resultados de una consulta.
```

```
type Resultado = [String]

-- ------------------------------------------------------------------------ --
-- Función que carga los datos de los pokemones desde el archivo de texto.  --
-- ------------------------------------------------------------------------ --
cargaBase :: IO [[String]]
cargaBase = do contenido <- readFile baseDatos
               return $ map (splitOn "|") (splitOn "\n" contenido)

-- ------------------------------------------------------------------------ --
-- Aplica formato a cada pokémon.                                           --
-- ------------------------------------------------------------------------ --
construye :: [String] -> String
construye datos = "No. " ++ (datos !! 0) ++ "\n" ++ "Nombre: " ++
                  (datos !! 1) ++ "\n" ++ (datos !! 2) ++ "\n" ++
                  (datos !! 3) ++ "\n"

-- ------------------------------------------------------------------------ --
-- Función que busca una cadena en una lista de listas.                     --
-- ------------------------------------------------------------------------ --
busca :: [[String]] -> Criterio -> Resultado
busca [] _ = []
busca (x:xs) c
   | isInfixOf c (x !! 0) = (construye x):(busca xs c)
   | isInfixOf c (x !! 1) = (construye x):(busca xs c)
   | otherwise = busca xs c

-- ------------------------------------------------------------------------ --
-- Despliega mediante funciones de entrada y salida los datos de cada pokemon --
-- obtenido.                                                                --
-- ------------------------------------------------------------------------ --
despliega :: Resultado -> IO ()
despliega [] = putStrLn ""
despliega (x:xs) = do putStrLn $ "\n" ++ x
                      despliega xs

-- ------------------------------------------------------------------------ --
-- Función que se encarga de seleccionar todos los pokemones cuyo número o  --
-- nombre contienen el criterio dado.                                       --
-- ------------------------------------------------------------------------ --
selecciona :: IO ()
selecciona = do putStr "\n\nNombre o número: "
                criterio <- getLine
                pokemones <- cargaBase
                let res = busca pokemones criterio in
                  if null res then
                      putStr "\nNo encontrado.\n\n"
                   else
                      despliega res

-- ------------------------------------------------------------------------ --
-- Función que pide los datos de un pokemon al usuario y construye la cadena --
-- que lo representa.                                                       --
-- ------------------------------------------------------------------------ --
nuevoPokemon :: IO ()
nuevoPokemon = do putStr "\n\nNúmero: "
                  num <- getLine
                  putStr "Nombre: "
                  nom <- getLine
                  putStr "Tipos (separados por comas): "
```

```
                  tip <- getLine
                  putStr "Descripción: "
                  desc <- getLine
                  appendFile baseDatos ("\n"++num++"|"++nom++"|"++tip ++"|"++desc)
                  putStr "\nPokémon guardado.\n\n"

-- ------------------------------------------------------------------------ --
-- Función que muestra un menú de opciones al usuario:                      --
-- [1] Consultar un pokémon.                                                --
-- [2] Dar de alta un pokémon.                                              --
-- [3] Salir.                                                               --
-- La función regresa la opción del usuario.                                --
-- ------------------------------------------------------------------------ --
muestraMenu :: IO Char
muestraMenu = do putStrLn "Por favor introduce una opción: "
                 putStrLn "[1] Consultar un pokémon"
                 putStrLn "[2] Dar de alta un pokémon"
                 putStrLn "[3] Salir"
                 putStr "Opción: "
                 opcion <- getChar
                 return opcion

-- ------------------------------------------------------------------------ --
-- Función encargada de procesar las opciones elegidas por el usuario.      --
-- ------------------------------------------------------------------------ --
procesa :: Char -> IO ()
procesa '1' = selecciona
procesa '2' = nuevoPokemon
procesa '3' = putStr "\n\nSaliendo...\n"
procesa _ = putStr "\n\nOpción no válida.\n\n"

-- ------------------------------------------------------------------------ --
-- Función encargada de llevar la ejecución de todo el programa interactivo. --
-- ------------------------------------------------------------------------ --
main :: IO ()
main = do c <- muestraMenu
          procesa c
          if c == '3' then
             putStr "Adiós\n"
          else
             main
```

Código 10: Programa interactivo para dar de alta Pokemones y consultarlos

```
*Pokedex> main
Por favor introduce una opción:
[1] Consultar un pokémon
[2] Dar de alta un pokémon
[3] Salir
Opción: 1

Nombre o número: saur

No. 1
Nombre: Bulbasaur
Planta, Veneno
```

```
A Bulbasaur es fácil verle echándose una siesta al sol. La semilla que tiene en
el lomo va creciendo cada vez más a medida que absorbe los rayos del sol.

No. 2
Nombre: Ivysaur
Planta, Veneno
Este Pokémon lleva un bulbo en el lomo y, para poder con su peso, tiene unas
patas y un tronco gruesos y fuertes. Si empieza a pasar más tiempo al sol, será
porque el bulbo está a punto de hacerse una flor grande.

No. 3
Nombre: Venusaur
Planta, Veneno
Venusaur tiene una flor enorme en el lomo que, según parece, adquiere unos
colores muy vivos si está bien nutrido y le da mucho el sol. El aroma delicado
de la flor tiene un efecto relajante en el ánimo de las personas.

Por favor introduce una opción:
[1] Consultar un pokémon
[2] Dar de alta un pokémon
[3] Salir
Opción: 2
Número: 7
Nombre: Squirtle
Tipos (separados por comas): Agua

Descripción: El caparazón de Squirtle no le sirve de protección únicamente. Su
forma redondeada y las hendiduras que tiene le ayudan a deslizarse en el agua y
le permiten nadar a gran velocidad.

Pokémon guardado.

Por favor introduce una opción:
[1] Consultar un pokémon
[2] Dar de alta un pokémon
[3] Salir
Opción: 1

Nombre o número: 7

No. 7
Nombre: Squirtle
Agua
El caparazón de Squirtle no le sirve de protección únicamente. Su forma
redondeada y las hendiduras que tiene le ayudan a deslizarse en el agua y le
permiten nadar a gran velocidad.
```

```
Por favor introduce una opción:
[1] Consultar un pokémon
[2] Dar de alta un pokémon
[3] Salir
Opción: 3

Saliendo...
Adiós
```

12.4. Manejo de excepciones

En todo lenguaje de programación es usual que ocurran diversos errores en tiempo de ejecución. En algunos lenguajes de programación como JAVA o C# se incluyen primitivas para manejar este tipo de errores, usualmente llamados *excepciones*. Una *excepción* es la forma en que algunos lenguajes de programación informan al usuario que ocurrió algún error[5].

La idea detrás del manejo de excepciones consiste en detectar cuando se *lanza* una excepción y *capturar o atrapar* la misma realizando las acciones correspondientes de acuerdo al tipo de excepción obtenido[5].

Control de errores con `Maybe`

Una forma de controlar errores en HASKELL es mediante su sistema de tipos. Esto es, usar un tipo de dato para indicar si un resultado es válido o no[5]. El tipo de dato más usado para representar esta noción es `Maybe`[7].

Por ejemplo, en el Código se muestra la función `primero` que obtiene el primer elemento de una lista, esto es, una definición propia de la función `head` de Haskell.

```
-- Función que obtiene el primer elemento de una lista.
primero :: [a] -> a
primero [] = ¿?
primero (x:xs) = x
```

Código 11: Función primero

¿Que debería regresar la función `primero` del Código 11 cuando reciba la lista vacía?

No es posible regresar la lista vacía por ejemplo, debido a la firma de la función. Se necesita unificar el tipo de regreso de la función para identificar cuando ocurre algún tipo de error.

El tipo de dato `Maybe` permite esto y se define como sigue:

[7] Otro tipo de dato altamente usado es `Either`.

```
date Maybe a = Nothing
             | Just a
```

De esta forma, la definición de la función `primero` puede modificarse como en el Código 12. Las líneas 3 y 4 muestran el uso del mismo tipo de dato para unificar los resultados.

```
-- Función que obtiene el primer elemento de una lista.
primero :: [a] -> Maybe a
primero [] = Nothing
primero (x:xs) = Just x
```

Código 12: Función primero haciendo uso de Maybe

Esta forma definir funciones es útil pues al obtener `Nothing` como resultado, el programador puede inferir que un error ha ocurrido lo cual le permite realizar las acciones correspondientes para mitigarlo.

Observación 12.2. El tipo de dato `Maybe` forma parte de la clase de tipos `Monad` por lo cual puede usarse en programas interactivos.

Manejo de excepciones de entrada/salida

Aún con el uso de tipos de datos como `Maybe` para mitigar errores, HASKELL incluye primitivas para manejar excepciones. Este mecanismo es importante cuando se interactúa con usuarios o cuando se manipulan archivos pues al tratar con el *mundo real* se tienen factores externos que el propio lenguaje de programación no puede controlar[5].

Por ejemplo, para capturar excepciones de entrada/salida, puede usarse la función `catchIOError` de la biblioteca `System.IO.Error`[5]:

```
catchIOError :: IO a -> (IOError -> IO a) -> IO a
```

El primer parámetro de esta función es una acción de entrada/salida, si durante la ejecución de esta acción, ésta llegara a generar una excepción, se pasa al siguiente parámetro llamado *manipulador*. El manipulador recibe un error y regresa una acción que indica qué hacer en caso de que ocurra una excepción. De esta forma `cathIOError` regresa, ya sea el valor de su primer argumento o bien, el valor de regreso del manipulador en caso de que se genere una excepción[5].

Ejemplo 12.5. El Código 13 muestra la modificación a la función `cargaBase` del Código 10 para controlar errores de entrada y salida. La función `cargaBase` (línea 3) ejecuta la función `try`, si la función se ejecuta correctamente, regresa una lista con los datos correspondientes. En caso que se genere una excepción, la función `cacthIOError` buscará un manipulador, en este caso la función con el mismo nombre y regresa una lista vacía pues no se encontró ningún dato.

```
-- Función que carga los datos de los pokemones desde el archivo de
-- texto.
cargaBase = catchIOError try manipulador

try :: IO [[String]]
try = do contenido <- readFile baseDatos
         return $ map (splitOn "|") (splitOn "\n" contenido)

manipulador :: IOError -> IO [[String]]
manipulador ioe = return []
```

Código 13: Función cargaBase controlando excepciones de entrada y salida

Referencias

[1] Manuel Soto, *Manual de Prácticas para la Asignatura de Programación Declarativa,* Facultad de Ciencias UNAM, Proyecto de Titulación, 2019.

[2] Graham Hutton, *Programming in Haskell*, Tercera Edición, Cambridge University Press, 2008.

[3] "All about monads". The Haskell Programming Language. Web. 5 Diciembre. 2020. <https://wiki.haskell.org/All_About_Monads>

[4] Favio E. Miranda, Lourdes del C. González, Leguajes de Programación, Notas de clase, Facultad de Ciencias, Universidad Nacional Autónoma de México. 2016.

[5] Miran Lipovaca, Learn You a Haskell for Great Good! A Beginner's Guide, Primera edición, No Starch Press, 2011.

[6] "Pokedex". Pokemon.ex. Web. 10 julio. 2018. <https://www.pokemon.com/es/pokedex/>