+++ draft = false date = 2025-05-23 title = 'Práctica 3: Paradigma funcional utilizando Haskell' summary = 'Edgar Rafael Garcia Ceseña' +++

Introducción

El paradigma funcional se basa en el uso de funciones matemáticas puras. Este estilo de programación, en vez de enfocarse en cómo se hacen las tareas paso a paso o en modificar objetos y sus estados, se enfoca en el "qué" se quiere lograr, tratando el código como evaluación de funciones.

En esta práctica se explica un programa en Haskell que gestiona una lista de pendientes, donde el usuario puede agregar, eliminar, ver y editar elementos de la lista.

Desarrollo

El primer archivo que vamos a revisar es `Lib.hs`, que contiene toda la lógica del programa.

```
module Lib
  ( prompt,
    editIndex,
  )
where
```

En este bloque se define el módulo `Lib`, donde se exponen las funciones `prompt` y `editIndex` para poder usarlas desde otros archivos, como `Main.hs`.

Luego se importa una función útil de listas como `isPrefixOf`, que sirve para ver si una cadena empieza con otra.

```
import Data.List
```

Ahora pasamos a la lógica del programa, empezando por la función `putTodo`.

```
putTodo :: (Int, String) -> IO ()
putTodo (n, todo) = putStrLn (show n ++ ": " ++ todo)
```

Esta función imprime un elemento de la lista de tareas junto con su número, por ejemplo: `0: Hacer tarea`.

```
prompt :: [String] -> IO ()
prompt todos = do
  putStrLn ""
  putStrLn "Test todo with Haskell. You can use +(create), -(delete), s(show),
e(dit), l(ist), r(everse), c(lear), q(uit) commands."
```

```
  command <- getLine
  if "e" `isPrefixOf` command
    then do
      print "What is the new todo for that?"
      newTodo <- getLine
      editTodo command todos newTodo
    else interpret command todos
```

Esta función muestra un mensaje explicando los comandos y lee uno del usuario. Si el comando empieza con `e`, llama a `editTodo`; si no, llama a `interpret`.

```
interpret :: String -> [String] -> IO ()
interpret ('+' : ' ' : todo) todos = prompt (todo : todos)
interpret ('-' : ' ' : num) todos =
  case deleteOne (read num) todos of
    Nothing -> do
      putStrLn "No TODO entry matches the given number"
      prompt todos
    Just todos' -> prompt todos'
interpret ('s' : ' ' : num) todos =
  case showOne (read num) todos of
    Nothing -> do
      putStrLn "No TODO entry matches the given number"
      prompt todos
    Just todo -> do
      print $ num ++ ". " ++ todo
      prompt todos
interpret "l" todos = do
  let numberOfTodos = length todos
  putStrLn ""
  print $ show numberOfTodos ++ " in total"
  mapM_ putTodo (zip [0 ..] todos)
  prompt todos
interpret "r" todos = do
  let numberOfTodos = length todos
  putStrLn ""
  print $ show numberOfTodos ++ " in total"
  let reversedTodos = reverseTodos todos
  mapM_ putTodo (zip [0 ..] reversedTodos)
  prompt todos
interpret "c" todos = do
  print "Clear todo list."
  prompt []
interpret "q" todos = return ()
interpret command todos = do
  putStrLn ("Invalid command: `" ++ command ++ "`")
  prompt todos
```

Esta función interpreta el comando y realiza la acción adecuada:

- `+`: Agrega una tarea.
- `-`: Elimina una tarea según su número con `deleteOne`.
- `s`: Muestra una tarea con `showOne`.
- `l`: Lista todas las tareas.
- `r`: Invierte el orden de la lista.
- `c`: Borra todas las tareas.
- `q`: Sale del programa.

```haskell
deleteOne :: Int -> [a] -> Maybe [a]
deleteOne 0 (_ : as) = Just as
deleteOne n (a : as) = do
  as' <- deleteOne (n - 1) as
  return (a : as')
deleteOne _ [] = Nothing
```

`deleteOne` elimina un elemento según su posición. Si el índice no es válido, devuelve `Nothing`.

```haskell
showOne :: Int -> [a] -> Maybe a
showOne n todos =
  if (n < 0) || (n > length todos)
    then Nothing
    else Just (todos !! n)
```

`showOne` muestra el elemento de la lista en la posición indicada, si el índice es válido.

```haskell
editOne :: Int -> [a] -> String -> Maybe a
editOne n todos newTodo =
  if (n < 0) || (n > length todos)
    then Nothing
    else do
      Just (todos !! n)
```

`editOne` es similar a `showOne`, pero se usa en la función `editTodo`.

```haskell
editTodo :: String -> [String] -> String -> IO ()
editTodo ('e' : ' ' : num) todos newTodo =
  case editOne (read num) todos newTodo of
    Nothing -> do
      putStrLn "No TODO entry matches the given number"
      prompt todos
    Just todo -> do
      putStrLn ""
      print $ "Old todo is " ++ todo
      print $ "New todo is " ++ newTodo
      let newTodos = editIndex (read num :: Int) newTodo todos
```

```
        let numberOfTodos = length newTodos
        putStrLn ""
        print $ show numberOfTodos ++ " in total"
        mapM_ putTodo (zip [0 ..] newTodos)
        prompt newTodos
```

`editTodo` modifica una tarea. Muestra la tarea anterior, la nueva y actualiza la lista.

```
reverseTodos :: [a] -> [a]
reverseTodos xs = go xs []
  where
    go :: [a] -> [a] -> [a]
    go [] ys = ys
    go (x : xs) ys = go xs (x : ys)
```

`reverseTodos` invierte una lista usando recursión.

Ahora revisamos el archivo `Main.hs`:

```
import Configuration.Dotenv (defaultConfig, loadFile)
import Lib (prompt)
import System.Environment (lookupEnv)
import Web.Browser (openBrowser)
```

Se importan módulos, incluyendo `Lib.hs`, para usar la función `prompt` y abrir un sitio web.

```
main :: IO ()
main = do
  loadFile defaultConfig
  website <- lookupEnv "WEBSITE"

  case website of
    Nothing -> error "You should set WEBSITE at .env file."
    Just s -> do
      result <- openBrowser s
      if result
        then print ("Could open " ++ s)
        else print ("Couldn't open " ++ s)

      putStrLn "Commands:"
      putStrLn "+ <String> - Add a TODO entry"
      putStrLn "- <Int>    - Delete the numbered entry"
      putStrLn "s <Int>    - Show the numbered entry"
      putStrLn "e <Int>    - Edit the numbered entry"
      putStrLn "l          - List todo"
      putStrLn "r          - Reverse todo"
      putStrLn "c          - Clear todo"
```

```
        putStrLn "q           - Quit"
        prompt [] -- Start with the empty todo list.
```

Esta función principal carga un archivo `.env`, intenta abrir un sitio web y luego muestra las instrucciones de uso. Llama a `prompt []` para comenzar con una lista vacía.

Conclusión

En conclusión, me pareció muy interesante esta práctica porque en esta práctica se trabajó el paradigma funcional usando Haskell, creando una aplicación para manejar pendientes. El análisis del código me permitió entender más las funciones puras, recursión y cómo se construye una lógica de control sin cambiar estados.