Universidad Autónoma de Baja California

Facultad de Ingeniería, Arquitectura y Diseño

Paradigmas de la programación

Práctica 3

Instalación y funcionamiento de Haskell

Arturo Rafael Cornejo Escobar

31 de abril del 2025

INTRODUCCIÓN

Haskell es un lenguaje de programación funcional puro, tipificado estáticamente y perezoso, distinto de la mayoría. Recibe su nombre de Haskell Brooks Curry.

Programación Funcional A diferencia de los lenguajes imperativos que ejecutan comandos secuenciales, un programa funcional es una expresión que se evalúa. El enfoque es en qué computar, no en cómo, permitiendo al sistema optimizar el orden de las operaciones.

Esto significa que no se especifica el orden exacto de las operaciones. El sistema determina la mejor secuencia para evaluar las dependencias. La asignación de variables no es tan relevante, ya que el foco está en qué se debe computar, no en cómo.

GHCi significa Glasgow Haskell Compiler interactive.

Es el intérprete interactivo del lenguaje de programación Haskell.

DESARROLLO

Instalación de entorno de desarrollo

En la página oficial de Haskell, nos encontramos con el comando del instalador.

```
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; try { &
([ScriptBlock]::Create((Invoke-WebRequest
https://www.haskell.org/ghcup/sh/bootstrap-haskell.ps1 -
UseBasicParsing))) -Interactive -DisableCurl } catch { Write-Error $_ }
```

Después de confirmar los directorios de instalación y la instalación de modulos, se abre una ventana de MSY64 que terminará la descarga de los elementos necesarios.

Una vez terminada la instalación, hacemos verificación de esta, con el comando `gch --version` y regresa:

```
The Glorious Glasgow Haskell Compilation System, version 9.6.7
```

Se procede a instalar Docker que es una herramienta que te permite **crear, ejecutar y compartir aplicaciones en contenedores**.

Una vez que WSL y Docker Desktop estén configurados, se procede a preparar el entorno para desarrollar en la máquina y compilar dentro de un contenedor.

Obtenemos la imagen de Docker `relaxed_buck.tar` y la montamos ejecutando el comando `docker load -i relaxed_buck.tar`.

```
uedia@DianaLaura MINGW64 ~
$ docker load -i relaxed_buck.tar
Loaded image: vsc-practica3-c5386672936dab905a26c5c91ed988543d546598d7125e642683
74c3ed288144:latest
```

Instalamos la extension **Dev Containers** en VScode y se clona el folder `.devcontainer` del repositorio del portafolio, hacemos el primer programa de prueba para comprender como funciona.

```
uedia@DianaLaura MINGW64 /c/portafolio/hsworkshop (main)
$ ghc helloworld.hs
[1 of 2] Compiling Main             ( helloworld.hs, helloworld.o )
[2 of 2] Linking helloworld.exe

uedia@DianaLaura MINGW64 /c/portafolio/hsworkshop (main)
$ ls
helloworld.exe*  helloworld.hi  helloworld.hs  helloworld.o  README.md

uedia@DianaLaura MINGW64 /c/portafolio/hsworkshop (main)
$ ./helloworld.exe
Hello, World! from docker container ... yeah!
```

Aplicación todo

Con el comando `stack new todo`, creamos un nuevo proyecto desde las plantillas de **stack**.

```
Looking for Cabal or package.yaml files to use to initialise Stack's
project-level YAML configuration file.

Using the Cabal packages:
* todo\

Selecting the best among 13 snapshots...


Note: Matches
      https://raw.githubusercontent.com/commercialhaskell/stackage-snapshots/mas
ter/lts/23/24.yaml

Selected the snapshot
https://raw.githubusercontent.com/commercialhaskell/stackage-snapshots/master/lt
s/23/24.yaml.
Initialising Stack's project-level configuration file using snapshot
https://raw.githubusercontent.com/commercialhaskell/stackage-snapshots/master/lt
s/23/24.yaml.
Considered 1 user package.
Writing configuration to todo\stack.yaml.
Stack's project-level configuration file has been initialised.
```

```
uedia@DianaLaura MINGW64 /c/portafolio (main)
$ ls todo
CHANGELOG.md  LICENSE  README.md  Setup.hs  app/  package.yaml  src/  stack.yaml  test/  todo.cabal
```

Comenzaremos con el package.yaml para usar los paquetes de Haskell en su proyecto y agregamos dotenv y el paquete open-browser al proyecto.

```
dependencies:
- base >= 4.7 && < 5
- dotenv
- open-browser

ghc-options:
- -Wall
- -Wcompat
```

Para comprobar que funciona correctamente, se corre una prueba del proyecto con el comando `stack test`.

> Su función es compilar y ejecutar las pruebas (unitarias, integración, etc.) definidas en un proyecto de software escrito en el lenguaje de programación Haskell.

```
todo> build (lib + exe + test) with ghc-9.8.4
Preprocessing library for todo-0.1.0.0..
Building library for todo-0.1.0.0..
[1 of 2] Compiling Lib
[2 of 2] Compiling Paths_todo
Preprocessing executable 'todo-exe' for todo-0.1.0.0..
Building executable 'todo-exe' for todo-0.1.0.0..
[1 of 2] Compiling Main
[2 of 2] Compiling Paths_todo
[3 of 3] Linking .stack-work\dist\f24b2e15\build\todo-exe\todo-exe.exe
Preprocessing test suite 'todo-test' for todo-0.1.0.0..
Building test suite 'todo-test' for todo-0.1.0.0..
[1 of 2] Compiling Main
[2 of 2] Compiling Paths_todo
[3 of 3] Linking .stack-work\dist\f24b2e15\build\todo-test\todo-test.exe
todo> copy/register
Installing library in C:\portafolio\todo\.stack-work\install\64a2b18e\lib\x86_64-windows-ghc-9.8.4\t
odo-0.1.0.0-JP3k2Jx4uwiCWIPvn1nyn6
Installing executable todo-exe in C:\portafolio\todo\.stack-work\install\64a2b18e\bin
Registering library for todo-0.1.0.0..
todo> test (suite: todo-test)

Test suite not yet implemented



todo> Test suite todo-test passed
Completed 2 action(s).
```

Compilamos con el comando `stack run`, este es el resultado esperado:

```
uedia@DianaLaura MINGW64 /c/portafolio/todo (main)
$ stack run
someFunc
```

Estructura del Proyecto

La aplicación se divide en tres archivos fundamentales, cada uno con una función específica:

- app/Main.hs: Es la entrada de la aplicación, manejando la interfaz y de iniciar el bucle de la aplicación.
- src/Lib.hs: Contiene toda la lógica de negocio de la aplicación: cómo se añaden, eliminan, muestran, editan y listan las tareas pendientes.
- test/Spec.hs: Contiene las pruebas unitarias para verificar que las funciones de la lógica funcionan correctamente.

Main.hs

```haskell
module Main where

import Lib (prompt)

main :: IO ()
main = do
    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.
```

Lib.hs

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

import Data.List

-- import Data.Char (digitToInt)

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

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

interpret :: String -> [String] -> IO ()
interpret ('+' : ' ' : todo) todos = prompt (todo : todos) -- append todo to the
empty or previous todo list [] here.
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

  -- Move the functions below to another file.

  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

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

  editIndex :: Int -> a -> [a] -> [a]
  editIndex i x xs = take i xs ++ [x] ++ drop (i + 1) xs

  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 index = head (map digitToInt num)
```

```
        -- let index = read num::Int
        -- print index

        let newTodos = editIndex (read num :: Int) newTodo todos -- Couldn't match
expected type ‘Int’ with actual type ‘[Char]
        let numberOfTodos = length newTodos
        putStrLn ""
        print $ show numberOfTodos ++ " in total"
        mapM_ putTodo (zip [0 ..] newTodos)

        prompt newTodos

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

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

Contiene la lógica central de la aplicación To-Do en Haskell. Define la función prompt, que es el bucle interactivo para procesar los comandos del usuario (+, -, l, etc.), y todas las funciones auxiliares para manipular la lista de tareas (añadir, borrar, editar, mostrar, invertir). Usa pattern matching, recursión y el tipo Maybe para gestionar las operaciones y posibles errores.

Una vez con la logica y la interfaz, se procede a probar las funciones iniciando con el comando `stack repl`:

Cuando es ejecutado en el directorio raiz del proyecto, stack repl hace algunas cosas antes de iniciar GHCi:

- Resuelve y descarga dependencias.
- Compila el proyecto.
- Carga los módulos de tu proyecto: Una vez que GHCi se inicia, automáticamente carga todos los módulos (`Lib.hs`) del proyecto.

```
uedia@DianaLaura MINGW64 /c/portafolio/todo (main)
$ stack repl
Using main module:
1.  Package todo, component todo:exe:todo-exe, with main-is file: C:\portafolio\todo\app\Main.hs.

todo> initial-build-steps (lib + exe)

Warning: The following GHC options are incompatible with GHCi and have not been passed to it: -threaded.

Configuring GHCi with the following packages: todo.

Warning: Multiple files use the same module name:
         * Paths_todo found at the following paths
           * C:\portafolio\todo\.stack-work\dist\f24b2e15\build\autogen\Paths_todo.hs (todo:lib)
           * C:\portafolio\todo\.stack-work\dist\f24b2e15\build\todo-exe\autogen\Paths_todo.hs (todo:exe:todo-exe)

GHCi, version 9.8.4: https://www.haskell.org/ghc/  :? for help
[1 of 4] Compiling Lib              ( C:\portafolio\todo\src\Lib.hs, interpreted )
```

```
[3 of 4] Compiling Paths_todo       ( C:\portafolio\todo\.stack-work\dist\f24b2e15\build\autogen\Paths_todo.hs, interpreted )
Ok, three modules loaded.
Loaded GHCi configuration from C:\Users\uedia\AppData\Local\stack\ghci-script\1f52abca\ghci-script
ghci>
```

Probamos las funciones en el GHCi:

```
Ok, one module loaded.
ghci> deleteOne 0 ["item1", "item2", "item3"]
Just ["item2","item3"]
ghci> reverseTodos ["hola", "mundo"]
["mundo","hola"]
ghci> showOne 1 ["a", "b", "c"]
Just "b"
```

Ya que vemos que esta lista para ser usada, probamos con `stack run` y se muestra esto:

```
uedia@DianaLaura MINGW64 /c/portafolio/todo (main)
$ stack run
Commands:
+ <String> - Add a TODO entry
- <Int>    - Delete the numbered entry
s <Int>    - Show the numbered entry
e <Int>    - Edit the numbered entry
l          - List todo
r          - Reverse todo
c          - Clear todo
q          - Quit

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
+

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
+

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
l

"2 in total"
0:
1:

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
```

Probaremos las funciones:

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

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
+ primer_post

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
+ segundo_post

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
l

"2 in total"
0: segundo_post
1: primer_post

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
- 1

Test todo with Haskell. You can use +(create), -(delete), s(show), e(dit), l(ist), r(everse), c(lear), q(uit) commands.
l

"1 in total"
0: segundo_post
```

Specs.hs

```haskell
import Control.Exception

import Lib (editIndex)

main :: IO ()
main = do
    putStrLn "Test:"
    let index = 1
    let new_todo = "two"
    let todos = ["Write", "a", "blog", "post"]
    let new_todos = ["Write", "two", "blog", "post"]

    let result = editIndex index new_todo todos == new_todos

    -- assert :: Bool -> a -> a
    putStrLn $ assert result "editIndex worked."
```

Esta es una prueba unitaria para ver si la aplicación esta funcionando con el comando `stack test`.

```
Installing executable todo-exe in C:\portafolio\todo\.stack-work\install\64a2b18e\bin
Registering library for todo-0.1.0.0..
todo> test (suite: todo-test)

Test:
editIndex worked.



todo> Test suite todo-test passed
Completed 2 action(s).
```

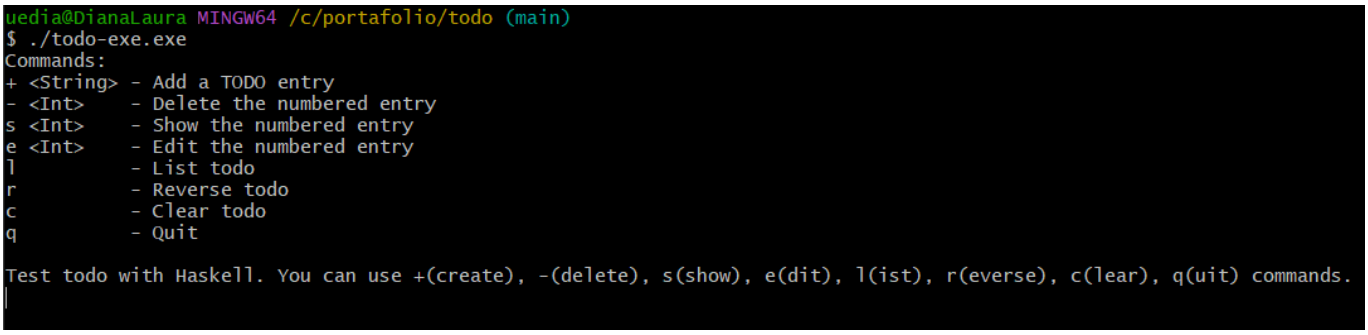
.

Crear y usar un ejecutable de la app Todo

Este proceso te permite convertir tu proyecto Haskell en un programa independiente que puedes ejecutar directamente desde tu terminal, sin necesidad de stack run o GHCi.

El comando `stack install --local-bin-path .` compila la aplicación y la convierte en un archivo ejecutable directamente en la carpeta del proyecto.

Ya con el ejecutable creado, se puede ejecutar directamente desde la terminal.



.

Se puede hacer un comando global con las siguientes instrucciones: `mv todo-exe todo` y `mv todo /usr/local/bin`. Así se podra hacer uso de la aplicación Haskell **todo** con solo un comando `todo`.

Referencias

Steadylearner. (2023, 23 junio). **How to use Haskell to build a todo app with Stack**. *DEV Community*. https://dev.to/steadylearner/how-to-use-stack-to-build-a-haskell-app-499j?authuser=0

Dirección del repositorio:

https://github.com/roixarturo/portafolio1

Dirección de la página de GitHub:

https://roixarturo.github.io/portafolio1/