+++ date = '2025-05-22T10:20:24-08:00' draft = false title = 'Práctica 3: Aplicación TODO en Haskell' +++

## Tabla de Contenido



---

# Introducción

En este reporte voy a deescribir el desarrollo de una aplicación de consola para la gestión de listas de tareas (TODO list) utilizando el lenguaje de programación funcional Haskell. El objetivo principal fue comprender y estar familiarizados con los conceptos fundamentales de Haskell, su sintaxis, el manejo de entrada/salida (IO), la manipulación de listas y el uso de herramientas de desarrollo como Stack. Esta práctica se basó en la adaptación y comprensión de tutoriales y ejemplos que ya existen para construir una aplicación funcional paso a paso.

# Descripción de la Aplicación "TODO"

La aplicación desarrollada permite a los usuarios administrar una lista de tareas pendientes a través de una interfaz de línea de comandos. Proporciona funcionalidades básicas para crear, visualizar, modificar y eliminar tareas.

La aplicación soporta los siguientes comandos principales:

- **Añadir tarea (`+ texto_tarea`)**: Agrega una nueva tarea a la lista.
- **Listar tareas (`l`)**: Muestra todas las tareas pendientes con un índice numérico.
- **Eliminar tarea (`- índice`)**: Borra la tarea correspondiente al índice proporcionado.
- **Editar tarea (`e índice`)**: Permite modificar el texto de una tarea existente.
- **Mostrar tarea (`s índice`)**: Visualiza una tarea específica.
- **Limpiar lista (`c`)**: Elimina todas las tareas de la lista.
- **Salir (`q`)**: Termina la ejecución de la aplicación.

# Entorno de Desarrollo: Haskell y Stack

Haskell es un lenguaje de programación puramente funcional. Para este proyecto, se utilizó el compilador GHC (Glasgow Haskell Compiler). **Stack** fue la herramienta empleada para la gestión del proyecto. Stack facilita la creación de proyectos, la gestión de dependencias, la compilación y la ejecución, asegurando un entorno de desarrollo consistente al manejar la instalación de GHC y las bibliotecas necesarias de forma aislada por proyecto.

# Proceso de Desarrollo

## 1. Creación del Proyecto

Se inició un nuevo proyecto Haskell utilizando Stack con el comando:

```
stack new MiProyectoTodo
cd MiProyectoTodo
```

## 2. Estructura del Proyecto

La estructura generada por Stack organiza el código fuente, los archivos de prueba y la configuración del proyecto. Los directorios y archivos más relevantes para esta práctica fueron:

- `app/Main.hs`: Contiene el punto de entrada principal de la aplicación.
- `src/`: Directorio para los módulos de la biblioteca. Aquí se creó `Crud.hs` para la lógica de la aplicación.
- `test/Spec.hs`: Archivo para las pruebas unitarias.
- `package.yaml`: Archivo de configuración del paquete, desde el cual Stack genera el archivo `.cabal`.
- `stack.yaml`: Archivo de configuración específico de Stack para el proyecto.

# Implementación de Funcionalidades Clave

La lógica de la aplicación se centralizó en un módulo `Crud.hs`, mientras que `Main.hs` sirvió como iniciador.

## 1. Módulo Principal de Lógica (`Crud.hs`)

Este módulo contiene las funciones que manejan el estado de la lista de tareas (representada como `[String]`) y la interacción con el usuario.

## 2. Bucle de Interacción y Procesamiento de Comandos

La función principal `prompt` en `Crud.hs` gestiona el ciclo de vida de la aplicación:

1. Muestra un mensaje al usuario solicitando un comando.
2. Lee la entrada del usuario.
3. Delega el procesamiento del comando a una función `interpret`.
4. Se llama recursivamente con el estado actualizado de la lista de tareas hasta que el usuario ingresa 'q'.

```
-- Ejemplo simplificado del bucle en Crud.hs
module Crud (prompt) where

import System.IO (hFlush, stdout) -- Para flush de salida
```

```haskell
type Tarea = String
type ListaTareas = [Tarea]

prompt :: ListaTareas -> IO ()
prompt tareas = do
    putStr "\nComandos (+, l, e, -, c, q): "
    hFlush stdout -- Asegura que el prompt se muestre antes de getLine
    comando <- getLine
    if comando == "q"
        then putStrLn "Adiós!"
        else procesarComando comando tareas

procesarComando :: String -> ListaTareas -> IO ()
procesarComando cmd tareas = do
    -- Aquí iría la lógica para interpretar 'cmd' y actualizar 'tareas'
    -- Por ejemplo:
    let nuevasTareas = case cmd of
                        ('+' : ' ' : desc) -> desc : tareas -- Añadir
                        "l"                -> tareas -- Listar no cambia la lista
aquí
                        _                  -> tareas -- Comando no reconocido
    if cmd == "l" then imprimirTareas nuevasTareas else return ()
    prompt nuevasTareas -- Llamada recursiva

imprimirTareas :: ListaTareas -> IO ()
imprimirTareas ts = mapM_ putStrLn $ zipWith (\n t -> show n ++ ". " ++ t) [0..]
ts
```

### 3. Gestión de Tareas (Añadir, Listar, Eliminar, Editar)

- **Añadir**: Se implementó un patrón para `'+' : ' ' : task` que añade la nueva tarea al principio de la lista.
- **Listar**: Se utilizó `zip` para emparejar tareas con índices y `mapM_` para imprimirlas.
- **Eliminar**: Requiere convertir el índice (String) a `Int` (usando `readMaybe` para seguridad) y luego una función auxiliar para quitar el elemento de la lista.

  ```haskell
  -- Ejemplo simplificado de función para eliminar
  eliminarTarea :: Int -> ListaTareas -> Maybe ListaTareas
  eliminarTarea idx tareas
      | idx < 0 || idx >= length tareas = Nothing -- Índice inválido
      | otherwise = Just (take idx tareas ++ drop (idx + 1) tareas)
  ```

- **Editar**: Similar a eliminar, pero después de validar el índice, se solicita el nuevo texto y se reemplaza el elemento en la lista.

```haskell
-- Ejemplo simplificado de función para editar
editarTarea :: Int -> String -> ListaTareas -> Maybe ListaTareas
editarTarea idx nuevoTexto tareas
    | idx < 0 || idx >= length tareas = Nothing
    | otherwise = Just (take idx tareas ++ [nuevoTexto] ++ drop (idx + 1)
tareas)
```

- El manejo de `Maybe` es necesario para operaciones que pueden fallar (índice inválido, formato incorrecto).

### 4. Punto de Entrada de la Aplicación (`Main.hs`)

El archivo `app/Main.hs` importa la función `prompt` de `Crud.hs` y la inicia con una lista de tareas vacía, mostrando un mensaje de bienvenida.

```haskell
-- Ejemplo simplificado de Main.hs
module Main where

import Crud (prompt) -- Asumiendo que Crud.hs está en src/

main :: IO ()
main = do
    putStrLn "--- Aplicación TODO en Haskell ---"
    putStrLn "Comandos disponibles: + tarea, l, e idx, - idx, s idx, c, q"
    prompt [] -- Iniciar con lista vacía
```

## Pruebas Unitarias

Se utilizó `test/Spec.hs` para probar funciones puras, como las de manipulación de listas (`eliminarTarea`, `editarTarea`). Las pruebas verifican que estas funciones se comportan como se espera al usar entradas válidas e inválidas.

```haskell
-- Ejemplo simplificado de una prueba en Spec.hs
import Control.Exception (assert)
-- import Lib (editarTarea)

-- Función de prueba para editarTarea
testEdicion :: IO ()
testEdicion =
    let listaInicial = ["comprar pan", "estudiar Haskell"]
        resultadoEsperado = Just ["comprar leche", "estudiar Haskell"]
        resultadoReal = editarTarea 0 "comprar leche" listaInicial -- Usando una
función hipotética
    in assert (resultadoReal == resultadoEsperado) (putStrLn "Prueba de edición:
PASSED")

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

```
    putStrLn "Ejecutando conjunto de pruebas..."
    testEdicion
    putStrLn "Pruebas finalizadas."
```

## Compilación y Ejecución

La compilación del proyecto se realiza con:

```
stack build
```

Para ejecutar la aplicación:

```
stack exec MiProyectoTodo-exe
```

O de forma combinada:

```
stack run
```

Para ejecutar las pruebas:

```
stack test
```

## Conclusión

Esta práctica fue importante para obtener una comprensión básica pero sólida de Haskell y su ecosistema. La aplicación TODO, aunque simple, permitió explorar conceptos cruciales del lenguaje. Fue un reto al considerar el uso de recursos que requerian capacidades mayores a las que tenia en ese momento (como el almacenamiento disponible). Haskell ofrece una perspectiva diferente para la resolución de problemas, y aunque su adopción inicial puede ser algo compleja de entender, los beneficios en términos de robustez y expresividad son notables. La experiencia adquirida, aunque introductoria, sienta las bases para futuras exploraciones en la programación funcional.