- **Nombre:** Salvador Moreno Sánchez
- **DNI:**
- **Centro asociado:** Motril (Granada)

Índice

Teoría de los Lenguajes de Programación - Memoria PEC

Cuestiones sobre la práctica

1. **Supongamos una implementación de la práctica (usando los mismos tipos de datos aquí presentados) en un lenguaje no declarativo (como Java, Pascal, C...). Comente qué ventajas y qué desventajas tendría frente a la implementación en Haskell. Relacione estas ventajas desde el punto de vista de la eficiencia con respecto a la programación y a la ejecución. ¿Cuál sería el principal punto a favor de la implementación en los lenguajes no declarativos? ¿Y el de la implementación en Haskell? Justifique sus respuestas.**

   En un lenguaje no declarativo como Java, Pascal o C, la implementación de la práctica presentaría algunas diferencias significativas en comparación con Haskell en términos de ventajas y desventajas.

   Ventajas de la implementación en lenguajes no declarativos:

   - **Control más directo sobre la memoria y los recursos:** en lenguajes no declarativos, se posee un control más directo sobre la asignación y liberación de memoria y otros recursos, lo que puede ser ventajoso para optimizar la eficiencia en términos de uso de memoria y rendimiento.

- **Optimización a nivel de hardware:** los lenguajes no declarativos suelen permitir una optimización más directa a nivel de hardware, lo que puede conducir a una ejecución más eficiente en términos de tiempo de ejecución y consumo de recursos, especialmente en sistemas de tiempo real o embebidos.
- **Flexibilidad en la implementación de estructuras de datos:** en lenguajes no declarativos, se tiene más flexibilidad para implementar estructuras de datos y algoritmos específicos para sus necesidades, lo que puede resultar en una optimización más precisa y eficiente para el problema en cuestión.

Desventajas de la implementación en lenguajes no declarativos:

- **Mayor complejidad y propensión a errores:** debido al control más directo sobre la memoria y los recursos, las implementaciones en lenguajes no declarativos suelen ser más propensas a errores y pueden requerir un código más complejo para gestionar correctamente la asignación y liberación de recursos.
- **Menor expresividad y concisión:** en comparación con Haskell, los lenguajes no declarativos suelen tener una sintaxis más verbosa y menos expresiva, lo que puede hacer que el código sea menos legible y mantenible, y requerir más líneas de código para lograr el mismo resultado.
- **Menor capacidad de abstracción:** los lenguajes no declarativos pueden tener una menor capacidad de abstracción en comparación con Haskell, lo que puede hacer que sea más difícil expresar conceptos complejos de manera clara y concisa.

Principal punto a favor de la implementación en lenguajes no declarativos:

El principal punto a favor de la implementación en lenguajes no declarativos sería su capacidad para optimizar directamente el uso de recursos y el rendimiento a nivel de hardware, lo que puede ser crucial en aplicaciones que requieren una alta eficiencia en términos de consumo de recursos y velocidad de ejecución.

Principal punto a favor de la implementación en Haskell:

El principal punto a favor de la implementación en Haskell sería su alto nivel de abstracción y su enfoque en la programación funcional, que permite escribir código de manera más concisa, expresiva y fácilmente comprensible. Además, el sistema de tipos estáticos y la inferencia de tipos en Haskell ayudan a prevenir errores y facilitan el razonamiento sobre el código. Esto puede conducir a un desarrollo más rápido y seguro de aplicaciones, especialmente en proyectos donde la claridad y la mantenibilidad del código son prioritarias sobre la optimización de recursos a nivel de hardware.

En resumen, mientras que la implementación en lenguajes no declarativos puede ofrecer una mayor optimización de recursos a nivel de hardware, la implementación en

Haskell destaca por su alto nivel de abstracción, claridad de código y seguridad en la programación. La elección entre los dos enfoques dependerá de los requisitos específicos del proyecto y las prioridades del equipo de desarrollo.

2. **Indique, con sus palabras, cómo afecta el predicado predefinico no lógico corte (!) al modelo de computación de Prolog. ¿Cómo se realizaría este efecto en Haskell? ¿Considera que sería necesario el uso del corte en una implementación en Prolog de esta práctica? Justifique sus respuestas.**

   El predicado predefinido no lógico "corte" (!) en Prolog tiene un impacto significativo en el modelo de computación de Prolog, ya que interrumpe la búsqueda de soluciones alternativas para una consulta dada y descarta cualquier elección alternativa realizada en las llamadas recursivas previas al punto de corte. Esto implica que una vez que se alcanza el corte en un punto de elección, Prolog no explorará otras ramas de búsqueda, lo que puede mejorar significativamente la eficiencia y el rendimiento del programa, especialmente en casos donde solo se necesita una solución.

   En Haskell, un lenguaje funcional puro, no existe un mecanismo directo equivalente al corte en Prolog, ya que en Haskell no hay efectos secundarios ni estado mutable. En lugar de usar un corte para interrumpir la búsqueda, en Haskell se utilizan conceptos como la evaluación perezosa y la recursión estructural para controlar el flujo del programa y evitar cálculos innecesarios.

   En cuanto a si sería necesario el uso del corte en una implementación en Prolog de esta práctica, la respuesta depende del enfoque de implementación y de los requisitos específicos del programa. En general, el uso del corte puede simplificar y mejorar la eficiencia de la implementación en Prolog al evitar explorar alternativas que no son necesarias para encontrar la solución deseada. Sin embargo, el uso indiscriminado del corte puede ocultar errores en la lógica del programa y dificultar la comprensión y el mantenimiento del código. Por lo tanto, debe usarse con precaución y solo cuando sea realmente necesario para mejorar el rendimiento.

3. **Indique qué clases de constructores de tipos (ver capítulo 5 del libro de la asignatura) se han utilizado para definir los tipos de datos presentes en el módulo StackMachine. Justifique sus respuestas.**

   Se han utilizado principalmente dos clases de constructores de tipos para definir los tipos de datos:

   - **Uniones discriminadas (datos estructurados):** se utilizan para definir tipos de datos compuestos, donde un valor del tipo de datos puede ser construido a partir de otros valores y puede tener diferentes formas. En el módulo `StackMachine`, las uniones discriminadas se utilizan para definir los tipos `BOp`, `UOp` y `SynTree`. Por ejemplo, los constructores `ADD`, `SUB`, `MUL` y `DIV` del

tipo `BOp` representan las diferentes operaciones binarias, mientras que `NEG` del tipo `UOp` representa la operación unaria de negación. En el caso de `SynTree`, los constructores `Binary`, `Unary` y `Operand` representan las diferentes formas de expresiones en el árbol de sintaxis abstracta.

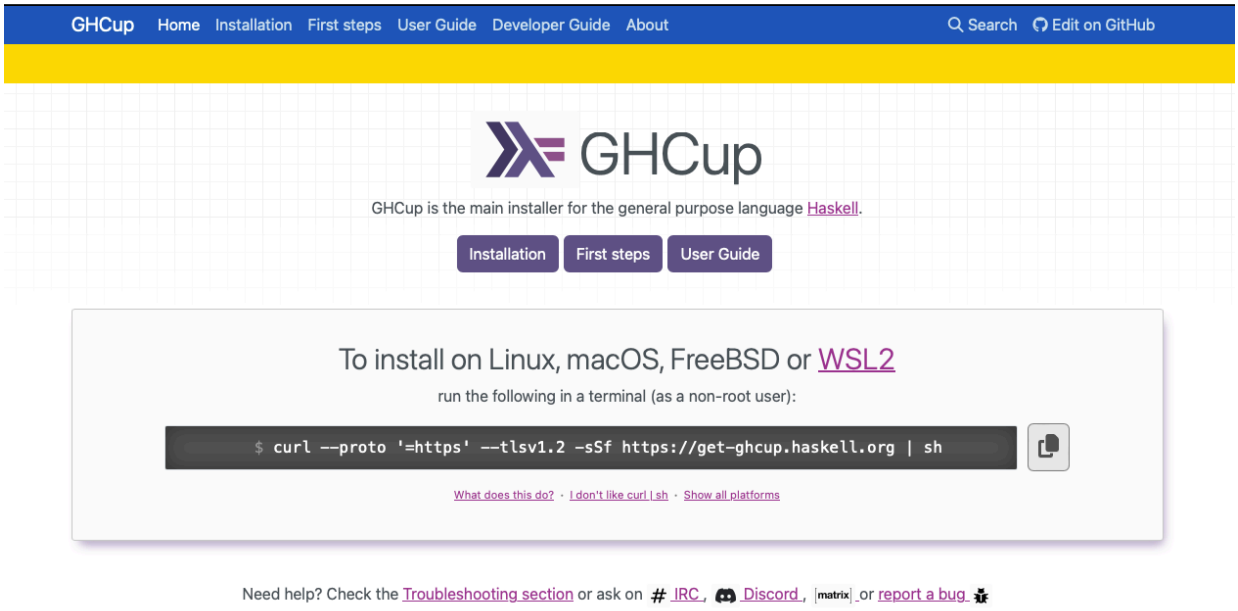
- **Sinónimos de tipos:** se utilizan para proporcionar nombres alternativos a tipos existentes, lo que ayuda a mejorar la legibilidad y la claridad del código. En el módulo `StackMachine`, se define un sinónimo de tipo `Stack` para representar una pila, lo que hace que el código sea más legible y expresivo. Por ejemplo, `Stack Integer` se lee como una pila de enteros.

Ejecución

Instalación de las herramientas necesarias

En mi caso, he instalado GHCup, el cual es el instalador principal del lenguaje de propósito general Haskell. Para ello, debemos seguir los siguientes pasos:

- Ir a la [web oficial del instalador GHCup](#).



- Dependiendo de tu sistema operativo:
  - Para Linux o MacOS, solo habría que ejecutar en la terminal el siguiente comando:

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org |
```

  - Para Windows, debemos instalar previamente WSL (Windows Subsystem for Linux), el cual permite a los desarrolladores ejecutar un entorno GNU/Linux

directamente en Windows, sin modificar, sin la sobrecarga de una máquina virtual tradicional o una configuración de arranque dual. Lo podemos encontrar en el siguiente enlace. Una vez instalado, ejecutamos en él el comando anterior.

- Una vez que lo tenemos instalado. En la terminal nos dirigimos a la carpeta con nuestros desarrollos en Haskell. Ya podremos ejecutar el comando:

```
ghci
```

Y podremos cargar nuestros módulos con el comando:

```
:l Main.hs
```

Ejemplos de ejecución

Procedo a mostrar el funcionamiento de todos los módulos y funciones principales que se albergan en la carpeta de entrega. Para ello, voy a usar de ejemplo la expresión en notación prefija `+ 4 * 5 - 2 -1`

Módulo `StackMachine.hs`

Función de construcción del árbol sintáctico

Función de evaluación

Módulo `Main.hs`

```
[salva PEC % ghci                                                              ]
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
[ghci> :l Main.hs                                                              ]
[1 of 3] Compiling StackMachine     ( StackMachine.hs, interpreted )
[2 of 3] Compiling Main             ( Main.hs, interpreted )
Ok, two modules loaded.
[ghci> main                                                                    ]

Escriba una expresión en notación prefija para ser evaluada
expresión: + 4 * 5 - 2 -1
El árbol de sintaxis abstracta es:
Binary ADD (Operand 4) (Binary MUL (Operand 5) (Binary SUB (Operand 2) (Operand
(-1))))

El resultado de la evaluación es:
19

Escriba una expresión en notación prefija para ser evaluada
expresión:
```

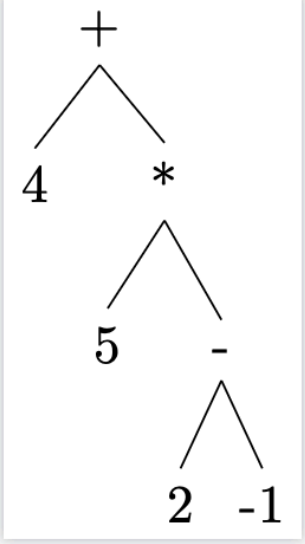
Módulo `ViewSynTree.hs`

Ejecución en consola

```
[salva PEC % ghci                                                              ]
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
[ghci> :l ViewSynTree.hs                                                       ]
[1 of 2] Compiling StackMachine     ( StackMachine.hs, interpreted )
[2 of 2] Compiling ViewSynTree      ( ViewSynTree.hs, interpreted )
Ok, two modules loaded.
[ghci> main                                                                    ]

Escriba un árbol para ser convertido a LaTeX
árbol: Binary ADD (Operand 4) (Binary MUL (Operand 5) (Binary SUB (Operand 2) (O
perand (-1))))
El código LaTeX para visualizar el árbol es:

\documentclass{standalone}
\usepackage{tikz, tikz-qtree}
\begin{document}
\Tree [.+ [.4 ] [.* [.5 ] [.- [.2 ] [.-1 ] ] ] ]
\end{document}

Escriba un árbol para ser convertido a LaTeX
árbol:
```

Vista en LaTeX (ejecutado en Overleaf)

```
\documentclass{standalone}
\usepackage{tikz, tikz-qtree}
\begin{document}
\Tree [.+ [.4 ] [.* [.5 ] [.- [.2 ] [.-1 ] ] ] ]
\end{document}
```



Ejecución exitosa de los tests propuestos por el equipo docente para los estudiantes - Módulo `TestTLP2024Estudiantes.hs`

```
[salva PEC % ghci                                                              ]
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
[ghci> :l TestTLP2024Estudiantes.hs                                            ]
[1 of 2] Compiling StackMachine     ( StackMachine.hs, interpreted )
[2 of 2] Compiling TestTLP2024Estudiantes ( TestTLP2024Estudiantes.hs, interpret
ed )
Ok, two modules loaded.
[ghci> main                                                                    ]
Loading test file: TestTLP2024Estudiantes.txt
Ejecutando el test: Test 1...¡Test superado!
Ejecutando el test: Test 2...¡Test superado!
Ejecutando el test: Test 3...¡Test superado!
Ejecutando el test: Test 4...¡Test superado!
Ejecutando el test: Test 5...¡Test superado!
Ejecutando el test: Test 6...¡Test superado!
Ejecutando el test: Test 7...¡Test superado!
Ejecutando el test: Test 8...¡Test superado!
Ejecutando el test: Test 9...¡Test superado!
Ejecutando el test: Test 10...¡Test superado!
Ejecutando el test: Test 11...¡Test superado!
Ejecutando el test: Test 12...¡Test superado!
Ejecutando el test: Test 13...¡Test superado!
Ejecutando el test: Test 14...¡Test superado!
Ejecutando el test: Test 15...¡Test superado!
Ejecutando el test: Test 16...¡Test superado!
Ejecutando el test: Test 17...¡Test superado!
Ejecutando el test: Test 18...¡Test superado!
Ejecutando el test: Test 19...¡Test superado!
Ejecutando el test: Test 20...¡Test superado!
End of test file
ghci>
```

Ejecución exitosa del test propuesto por el equipo docente en el foro, el cual posee una expresión gigantesca que ocupa casi 6MB de texto - Módulo `TestTLP2024Grande.hs`

🚨 **No se incluye en la entrega de la práctica debido a su enorme tamaño**

```
salva PEC % ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :l TestTLP2024Grande.hs
[1 of 2] Compiling StackMachine     ( StackMachine.hs, interpreted )
[2 of 2] Compiling TestTLP2024Grande ( TestTLP2024Grande.hs, interpreted )
Ok, two modules loaded.
ghci> main
Loading test file: TestTLP2024Grande.txt
Ejecutando el test: Test 1...¡Test superado!
End of test file
ghci>
```