Teoría de los Lenguajes de Programación

Nombre: Jorge Martinez Pazos

DNI: 39510046W

Correo: jmartinez5741@alumno.uned.es

Centro Asociado: Vigo (Pontevedra)

Índice

Apartado 1 3
Eficiencia en la programación 3
Eficiencia en la ejecución 3
Beneficios de la semántica de variables 4
Ventaja principal de un lenguaje no declarativo 4
Ventaja principal de Haskell 4
Apartado 2 5
Constructores de tipo usados 5
Bibliografía 6

Apartado 1

Eficiencia en la programación

La implementación en Haskell muestra ventajas significativas durante la fase de desarrollo debido a su paradigma funcional. La naturaleza declarativa del lenguaje permite expresar las operaciones necesarias para el juego de forma concisa y directa. Funciones como newTry y updateLS, que son centrales en la lógica del juego, se implementan con pocas líneas de código gracias al uso de pattern matching y funciones de orden superior.

En contraste, lenguajes como Java o C++ requerirían estructuras más complejas: clases para gestionar el estado del juego, bucles explícitos para comparar las palabras, y mecanismos adicionales para manejar la mutabilidad. Esta mayor verbosidad no solo incrementa el tiempo de desarrollo, sino que también introduce más oportunidades para cometer errores sutiles, especialmente en la gestión de estados y la sincronización en entornos concurrentes.

Eficiencia en la ejecución

Desde el punto de vista del rendimiento en tiempo de ejecución, los lenguajes no declarativos como C o Java suelen ser más eficientes. Esto se debe a su compilación directa a código máquina y a las optimizaciones específicas para el hardware subyacente. Haskell, aunque poderoso, introduce cierta carga debido a su evaluación perezosa y al manejo de estructuras inmutables, lo que puede afectar levemente el rendimiento en comparación con lenguajes más cercanos al metal.

Sin embargo, para un problema como el planteado en esta práctica donde no se requieren operaciones intensivas en tiempo real, la diferencia de rendimiento sería prácticamente imperceptible. La elección entre uno u otro lenguaje dependerá más de los requisitos del proyecto que de consideraciones puramente de rendimiento.

Beneficios de la semántica de variables

Haskell se beneficia enormemente de la semántica de sus variables inmutables. Esta característica elimina problemas comunes en lenguajes imperativos, como los errores derivados de estados compartidos o modificaciones accidentales de variables. La inmutabilidad facilita la escritura de programas concurrentes y paralelos, ya que evita condiciones de carrera y otros efectos colaterales.

En lenguajes como Java o C, las variables mutables pueden introducir bugs difíciles de rastrear, especialmente en proyectos de gran escala o con alta concurrencia. La necesidad de sincronizar accesos y modificar estados manualmente añade una capa adicional de complejidad que Haskell evita por diseño.

Ventaja principal de un lenguaje no declarativo

La principal ventaja de implementar esta práctica en un lenguaje no declarativo radica en el mayor control sobre el hardware y la gestión de memoria. Lenguajes como C permiten optimizaciones muy finas, lo que es crucial en aplicaciones donde el rendimiento es crítico. Además, la amplia adopción de lenguajes imperativos en la industria facilita la colaboración entre equipos y el mantenimiento del código a largo plazo.

Ventaja principal de Haskell

La mayor fortaleza de Haskell es su capacidad para expresar soluciones de manera clara y concisa, minimizando la probabilidad de errores lógicos. La inmutabilidad y las funciones puras no solo simplifican la verificación formal del código, sino que también hacen que las pruebas sean más robustas y predecibles. Estas características convierten a Haskell en una opción ideal para aplicaciones donde la corrección y la mantenibilidad son prioritarias.

Apartado 2

Constructores de tipo usados

1. **Enumerado**: Clue es un tipo de datos enumerado. Los constructores C, I, N y U son constructores de datos que representan los valores del tipo Clue. El ser un tipo enumerado quiere decir que Clue puede tomar exactamente uno de los valores definidos y no hay parámetros adicionales asociados a estos valores.

2. **Listas**: El tipo Try se define como una lista de tuplas, lo que lo convierte en un tipo recursivo que representa una secuencia de elementos. Las listas son estructuras fundamentales que reemplazan a los arrays tradicionales de otros lenguajes, ofreciendo flexibilidad y facilidad de manipulación.

3. **Producto cartesiano**: Cada elemento de la lista Try es una tupla (Char, Clue), que combina un carácter (Char) con una pista (Clue). Esta tupla es un ejemplo claro de producto cartesiano, ya que agrupa dos tipos distintos en un par ordenado sin añadir nombres a los campos (a diferencia de los registros).

Técnicamente type Try = [(Char, Clue)] es un sinónimo de tipo, su propósito principal es mejorar la legibilidad del código, pero no se considera un constructor porque solo da nombre a una estructura existente.

Bibliografía

-Aprende Haskell por tu bien

-Diapositivas Haskell

-Resúmenes sobre la asignatura