SINTAXIS Y SEMÁNTICA DE LOS LENGUAJES

CURSO K2006

GRUPO 9

TP N°1 GRUPAL

|  |  |
| --- | --- |
| INTEGRANTES | LEGAJO |
| Roberto Rodas | 2081301 |
| Alex Fiorenza | 2089865 |
| Oscar Mercado | 2083012 |
| Joaquín Fatur | 2079847 |

**SOLIDITY**

**BREVE HISTORIA**

Solidity, un lenguaje de programación de alto nivel orientado a contratos inteligentes se ha convertido en un componente crucial del ecosistema Ethereum desde sus inicios. Solidity fue inicialmente propuesto en agosto de 2014 por Gavin Wood. Más tarde, el lenguaje fue desarrollado por el equipo de Solidity de Ethereum Projects, dirigido por Christian Reitwiesner. Según lo especificado por Wood está diseñado en torno a la sintaxis de ECMAScript para que sea familiar para los desarrolladores web existentes. Aunque su historia está entrelazada con el desarrollo de Ethereum, Solidity destaca por su papel en la habilitación de contratos inteligentes, la automatización y la descentralización.

A medida que Ethereum buscaba llevar la tecnología de la cadena de bloques más allá de las simples transacciones de criptomonedas, surgió la necesidad de ejecutar acuerdos y lógica empresarial en la cadena. Esto llevó a la creación de Solidity, un lenguaje diseñado específicamente para escribir programas que puedan ejecutarse en la cadena de bloques Ethereum.

Uno de los mayores desafíos al escribir contratos inteligentes en Solidity es considerar la falta de reversibilidad y la inmutabilidad inherentes a la cadena de bloques. Los errores pueden ser costosos, y los contratos defectuosos pueden quedar inmovilizados en la cadena de bloques. Esto ha llevado a un enfoque intensivo en auditorías de seguridad y mejores prácticas para minimizar riesgos.

**ALGORITMOS**

**BUBBLE SORT (ORDENAMIENTO BURBUJA)**

Texto

Descripción generada automáticamente

**CONSIDERACIONES**

Solidity como cualquier lenguaje de programación utiliza la memoria para almacenar variables. La diferencia de Solidity frente a JavaScript es que Solidity funciona sobre el algoritmo blockchain y estos contratos son desplegados en ese mismo sistema. Por lo tanto, para tener más eficiencia en la ejecución del contrato y que este sea más liviano se pueden especificar si la función recibe un array de tipo “memory” o “calldata”.

Si es de tipo memory es temporal y volátil y se almacena durante la ejecución de la función. No es persistente en blockchain, pero puede ser modificable. Por lo tanto es más viable utilizar este tipo de variable cuando se quiere procesar datos.

Por otro lado “calldata” es de tipo read-only y no se puede modificar. Se utiliza para parámetros de solo lectura. En este caso el tipo memory es más viable ya que en el BubbleSort se manipulan cambio de elementos y sobreescritura del array pasado por parámetro.

**BINNARY SEARCH (BUSQUEDA BINARIA)**

Texto

Descripción generada automáticamente

Los parámetros que recibe la función es un array del tipo memoria (Se utiliza este tipo de dato por la misma explicación que bubble sort) y un entero de tipo target que sería aquel dato a buscar.

**¿Cómo se ejecuta un script en Solidity?**

Para este TP se utilizó remix.ethereum.org un IDE en la web que permite escribir código Solidity y desplegarlo. Solidity al ser un lenguaje de programación orientado a blockchain la ejecución debe ser en ese entorno. A fines prácticos no se desplegó el script en un entorno de producción sino de pruebas denominado “testnet”.

<https://sepolia.etherscan.io/address/0xe88a405169101a0bce410f8764cf1041b74399a3>

Al momento de realizar las pruebas se utiliza una biblioteca creada en go llamada:

<https://github.com/Giulio2002/bsol>

**JAVASCRIPT**

**Breve Historia:**

JavaScript es un lenguaje de programación que es una de las tecnologías centrales de la web, junto con HTML y CSS. En 2023, el 98.7% de los sitios web usa JavaScript en el lado del cliente para el comportamiento de las páginas web. Todos los principales navegadores tienen un intérprete de JavaScript dedicado para ejecutar el código en los dispositivos de los usuarios.

JavaScript es un lenguaje de alto nivel, compilado en tiempo de ejecución, que conforma el estándar ECMAScript. Tiene tipado dinámico; es multiparadigma, admite: orientado a objetos, funcional, imperativo y dirigido por eventos.

Este lenguaje fue creado en 1995 por Brendan Eich, un programador de Netscape. Su objetivo era optimizar el tiempo de navegación de los usuarios, agregar interactividad y dinamismo a las páginas web, creando un lenguaje que se ejecuta en el mismo navegador. Al principio se llamaba LiveScript pero luego fue renombrado a JavaScript para aprovechar la popularidad de Java. Sin embargo, Java y JavaScript tienen semánticas y propósitos diferentes.

JavaScript se diseñó con una sintaxis similar a C++ y Java,​ aunque adopta nombres y convenciones del lenguaje de programación Java.

En 1997 los autores propusieron JavaScript para que fuera adoptado como estándar de la European Computer Manufacturers Association ECMA, que a pesar de su nombre no es europeo sino internacional. En junio de 1997 fue adoptado como un estándar ECMA, con el nombre de ECMAScript.

Desde entonces, ECMAScript ha evolucionado con varias versiones que han añadido nuevas características y funcionalidades al lenguaje.

En el siglo 21 es cuando se vuelve realmente conocido gracias a la implementación de un conjunto de frameworks y bibliotecas que permiten mejorar las prácticas de programación en JS aumentando su uso fuera de navegadores web

**ALGORITMOS**

**BUBBLE SORTTexto

Descripción generada automáticamente**

**BINARY SEARCH**

**Texto

Descripción generada automáticamente**

Para la realización de las pruebas en JavaScript se utilizó el método **console.time().**

**--RESULTADOS BENCHMARKS—**

**SOLIDITY**

La biblioteca utilizada crea una blockchain local en la computadora donde se ejecuta y realiza un benchmark al script. En este caso se anotan las respuestas en nanosegundos.

**BUBBLE SORT**

Para un array de enteros: [23, 8, 56, 42, 91, 17, 33, 5, 72, 10] el benchmark arrojó un promedio calculado de tiempo de ejecución de **115.983418µs (nanosegundos).**

Para un array de enteros: [42, 17, 89, 53, 24, 67, 11, 36, 75, 28, 5, 91, 62, 19, 81, 47, 9, 32, 69, 53, 14, 73, 39, 84, 22, 61, 3, 50, 95, 7, 31, 68, 26, 70, 15, 79, 44, 10, 56, 90, 29, 66, 1, 58, 35, 77, 21, 45, 87, 2, 37, 82, 13, 59, 38, 85, 23, 71, 6, 94, 34, 64, 16, 78, 43, 8, 92, 27, 63, 12, 49, 88, 18, 54, 30, 74, 20, 46, 80, 4, 51, 96, 72, 33, 76, 25, 60, 86, 40, 83, 55, 97, 65, 48, 98, 57, 41, 99, 52, 93] se obtuvo un promedio calculado de tiempo de ejecución de **7361.929100µs (nanosegundos).**

**BINNARY SEARCH**

Para un array de enteros: [23, 8, 56, 42, 91, 17, 33, 5, 72, 10] donde el numero a buscar es el 17, el benchmark arrojó un promedio calculado de tiempo de ejecución de **41.883300µs (nanosegundos).**

Para el array de 100 enteros donde el número a buscar es el 21 se obtuvo un promedio calculado de tiempo de ejecución de **52.887500µs (nanosegundos).**

**JAVASCRIPT**

**BUBBLE SORT**

Para un array de enteros: [23, 8, 56, 42, 91, 17, 33, 5, 72, 10] el benchmark arrojó un promedio calculado de tiempo de ejecución de **0.06ms=60000ns.**

Para el array de 100 enteros se obtuvo un promedio calculado de tiempo de ejecución de **0.358ms=358000ns.**

**BINNARY SEARCH**

Para un array de enteros: [23, 8, 56, 42, 91, 17, 33, 5, 72, 10] donde el número a buscar es el 17 se obtuvo un promedio calculado de tiempo de ejecución de **0.053ms=53000ns.**

Para el array de 100 enteros donde el número a buscar es el 21 se obtuvo un promedio calculado de tiempo de ejecución de **0.051ms=51000ns.**

**--COMPARACIÓN Y CONCLUSIÓN—**

**COMPARACIÓN RESULTADOS**

En el ordenamiento burbuja, podemos ver que Solidity obtiene un rendimiento muy superior a JavaScript siendo aprox. 500 veces más rápido en tiempo de ejecución para el array de 10 enteros. Cuando analizamos el array de 100 enteros la “superioridad” de Solidity se reduce bastante a razón de 50 veces más rápido en tiempo de ejecución, pero aún así la diferencia sigue siendo considerablemente alta.

Por otro lado, en la búsqueda binaria observamos otra tendencia. Solidity sigue teniendo un rendimiento, en este caso, aún mayor que el de JavaScript siendo 1200 veces más rápido en tiempo de ejecución para el array de 10 enteros. El cambio de tendencia sucede con el array de 100 enteros, vemos que los resultados no cambian mucho individualmente de cada lenguaje pasando del array de 10 al de 100, por lo que la diferencia entre ambos no se reduce tanto como en el caso del ordenamiento burbuja, pasando de 1200 a 1000 veces más rápido.

En conclusión, podemos decir que Solidity es el “ganador” indiscutible en base a las pruebas realizadas.

**FACTORES INFLUYENTES EN LA DIFERENCIA DE RENDIMIENTO**

A continuación, analizaremos algunos de los factores que podrían haber influido en la marcada diferencia de rendimiento.

**Extracto BNF JavaScript:**

LexicalDeclaration:

LetOrConst BindingList ;

LetOrConst:

let

const

VariableStatement:

var VariableDeclarationList ;

Sin entrar en los detalles de "BindingList" y "VariableDeclarationList", podemos ver como JavaScript presenta tres alternativas para la declaración de variables: let, const y var. La diferenciación entre let y const con respecto a var en la BNF del lenguaje se basa en distintos motivos y uno de ellos que cuando se crea una variable con var, esta se inicializa automáticamente en "undefined", lo que permite su uso antes de asignarle un valor concreto. En cambio, las palabras clave let y const no comparten esta característica; al ser creadas, no reciben una asignación de valor por defecto.

La elección entre uno u otro no está determinada por el tipo de dato que vayamos a utilizar, sino por otros factores importantes. Por ejemplo, optamos por const cuando deseemos declarar algo que no pueda ser re-declarado ni alterado de ninguna manera. Esta modalidad de declaración de variables, donde no se especifica el tipo, es precisamente lo que hace que JavaScript sea un lenguaje de tipado dinámico.

**Extracto BNF Solidity:**

PrimaryExpression = BooleanLiteral

| NumberLiteral

| HexLiteral

| StringLiteral

| TupleExpression

| Identifier

| ElementaryTypeNameExpression

ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | 'var'

| Int | Uint | Byte | Fixed | Ufixed

Analizando la BNF de Solidity, vemos que si es necesario declarar el tipo de dato de nuestra variable en cuestión (bool, string, Int, etc). Debido a esto es que Solidity es un lenguaje de tipado estático, a diferencia de JavaScript.

En el tipado dinámico, el tipo está ligado al valor asignado y los chequeos son en tiempo de ejecución. En cambio, en el tipado estático el tipo está ligado a la variable y los tipos se chequean en tiempo de compilación. Esta es una de las razones que pudo haber influido en la diferencia de rendimiento entre ambos lenguajes.

Aun así, el tipado de Solidity es mucho más exhaustivo de lo que se puede ver a simple vista. Para enteros poseemos uint e int, uint es para enteros sin signo es decir el 0 y los positivos mientras que int es enteros de negativos a positivos. Si se asigna uint luego se pueden asignar la cantidad de bytes a utilizar. Para la implementación del algoritmo se han creado arrays de uint8 es decir array que componen enteros de 8 bytes. Por lo tanto al momento de escribir el código se aceptan de 1 a 2 dígitos si se quiere ampliar a más se utilizaran más bytes para los dígitos. Esto por obvias razones agiliza al lenguaje el proceso de compilado y el manejo de memoria. En cambio, JavaScript no le asigna importancia a esto y tiene un peor manejo de memoria, aunque una sintaxis más sencilla.

Solidity ha sido creado e ideado para ser ejecutado en una Virtual Machine, la cual es la de Ethereum. Esta computadora virtual tiene una pésima asignación de recursos y obviamente al ejecutar código se consume poder de cómputo, esto a nosotros se nos cobra mediante un concepto denominado “gas”, el cual se asume al momento de ejecutar el script. Este valor del “gas” depende de factores como congestión de la red (cantidad de transacciones que se realizan en ese momento del día), tamaño en bytes del contrato y tiempo de ejecución del mismo. Es por ello que, si se quiere reducir el gas al mínimo y por lo tanto pagar menos, Solidity provee herramientas que permiten optimizar el código ya sea mediante el tipado de variables, la asignación de bytes y otros operadores especificados en su BNF. Dentro de la comunidad Solidity se empezó a utilizar un término denominado “inline assembly” que permite utilizar la Virtual Machine de Ethereum (EVM) a un bajo nivel tal cual permite assembly, para optimizar aún más los recursos del contrato. Por lo tanto, se requieren conocimientos de assembly, esto resalta aún más la importancia que se le da al benchmarking y optimización del código en Solidity.

JavaScript en cambio no está pensado para correr en un ambiente tan limitado, de hecho, es un lenguaje de alto nivel de ahí a que sea tan laxo y poco restrictivo en su sintaxis. En el pasado, JavaScript era considerado un lenguaje de interpretado puro. Sin embargo, hoy en día los navegadores modernos, mediante el motor de JavaScript, utilizan una “técnica” combinada de interprete-compilador.

El motor de JavaScript es como un programa que interpreta código de este mismo lenguaje. El mismo está presente en mucho de los navegadores que conocemos, el más famoso el motor V8 de Google Chrome que es el que también utiliza Node.js (entorno utilizado para la ejecución del código en JS).

**¿Cómo funciona este motor?**

A diagram of a diagram

Description automatically generated

En primer lugar, se realiza un análisis léxico y sintáctico, el analizador lee línea a línea verificando si su sintaxis es apropiada para JS. El mismo analizador crea lo que se llama un Árbol de análisis que representa la estructura del código. El mismo descompone el código en una estructura jerárquica para facilitar su posterior interpretación y ejecución.

El intérprete ejecuta el código de JS directamente, recorriendo el árbol y ejecutando la instrucción una tras otra. Posteriormente se encarga de convertir el AST en un IR (representación intermedia). Un IR es como una estructura de dato o código que simboliza el código fuente (representación más abstracta del código). El mismo funciona como un puente entre el código fuente y el código de máquina, el IR comúnmente utilizado por JS es bytecode. Una de las razones por las cuales se utiliza esta representación intermedia es para facilitar la optimización del código (ej. eliminación de código muerto) para generar código maquina más eficiente.

Por último, un compilador just-in-time convierte el bytecode en código de máquina para su posterior ejecución. Just-in-time es un tipo de compilador que va tomando porciones del código y la compila en código maquina a medida que se necesita, a diferencia de un compilador tradicional que convierte todo el código fuente a código maquina antes de la ejecución. Este método combinado de interprete-compilador es el utilizado por la mayoría de los navegadores hoy en día, para mejorar el rendimiento de lo que es un interpretado puro.

Estos tres factores: diferencia tipado, manejo de memoria, Solidity siendo compilado y JavaScript utilizando una combinación de interprete-compilador se ven resaltados en las pruebas y vemos como Solidity responde mucho más rápido frente a JavaScript a pesar de ejecutar los mismos algoritmos con los mismos datos.