Introducción a los contratos inteligentes

Introduction to smart contract

Por
María González Gutiérrez

UNIVERSIDAD COMPLUTENSE MADRID

Grado en Matemáticas
Facultad de Ciencias Matemáticas

Dirigido por
Adrián Riesco Rodríguez
Departamento de Sistemas Informáticos y Computación

Madrid, 2020-2021

Índice general

Página

Capítulo 1

Introducción

¿Alguna vez ha pensado cómo alquilar una casa sin estar pendiente de los pagos del alquilado? ¿O cómo repartir el premio de una apuesta entre todos los apostantes de una vez si ninguno confía en los otros?

Por medio de contratos inteligentes podemos hacer esto y más. Los contratos inteligentes son una forma de crear contratos o acuerdos entre varios actores sin necesidad de intermediarios o de una autoridad central, lo que hace que sea un gran ahorro de tiempo y dinero. Estos contratos son programas que ejecutan y hacen cumplir un acuerdo entre las partes conformantes del mismo. Están almacenados en una determinada plataforma; en nuestro caso es Ethereum [18, 2], que es una plataforma global digital de código abierto para aplicaciones descentralizadas [2], es decir, el código de las aplicaciones está disponible para todos los usuarios y no depende de un sistema central que regule y controle su funcionamiento.

Estas aplicaciones, también llamadas DApps, interactúan con uno o varios contratos inteligentes. Si lo comparásemos con una aplicación web tradicional, la diferencia se encontraría en que en vez de usar un servidor que accede a servicios web y tiene una base de datos, utilizan un *blockchain* que almacena y ejecuta los contratos inteligentes, mientras que los datos son almacenados en forma de transacciones.

El hecho de que los contratos se encuentren en un entorno descentralizado hace que ninguno de los participantes del contrato pueda influir en el mismo y, por tanto, el contrato no debería beneficiar a ninguna parte, debería de ser neutral. Esto sería cierto siempre y cuando el contrato estuviese bien hecho, por lo que aquí entra en juego el lenguaje de programación que usemos. Si el lenguaje es fácil de entender, realizar contratos que no están bien hechos es más complicado. En nuestro caso vamos a usar un lenguaje fácil de entender como

es Vyper [7], ya que un principio importante de este lenguaje es que cualquier persona pueda entender el contrato, lo que hace que realizar un contrato bien hecho sea relativamente sencillo. Otros lenguajes, como Solidity, son más complicados de entender, lo que puede dar lugar a realizar contratos mal hechos si no se comprende muy bien el lenguaje con el que se está trabajando.

Como he comentado antes, estos contratos están alojados en Ethereum, una plataforma de código abierto en la que se crean y se ejecutan contratos inteligentes. Esta plataforma funciona con cadenas de bloques o *blockchain*, que son un registro único, consensuado y distribuido por los ordenadores que usan esta plataforma, es decir, es como si fuese un registro de todas las transacciones que se realizan. Estos ordenadores son los nodos de la red de Ethereum. Una forma de uso muy popular de la cadena de bloques es la criptomoneda [12], que es un medio de intercambio digital que usa criptografía para asegurar las transacciones. El funcionamiento de las mismas no está controlado por ningún banco central al estar en un entorno descentralizado. La criptomoneda nativa de Ethereum es el Ether. Podemos usar los contratos inteligentes para realizar transacciones de Ether y así mandar el Ether correspondiente de una cuenta a otra si el contrato así lo dictamina.

Estos contratos se ejecutan en la máquina virtual de Ethereum (EMV), que es el conjunto de todos los nodos de la red de Ethereum. Para realizar esta ejecución se realiza un esfuerzo computacional. Este esfuerzo está monitorizado por una unidad de medida llamada gas, la cual se mide en Ether. Si no existiese el gas y directamente se monitorizara con Ether, como este fluctúa durante todo el día, variaría también el coste de ejecutar un contrato, una transacción o una operación, pero la energía computacional para realizarlo es siempre constante. Por este motivo se creo el gas, para separar el coste computacional del económico.

La intención de este trabajo es aprender cómo crear contratos inteligentes escritos en Vyper. Para ello se ha realizado un repositorio con diversos tipos de contratos inteligentes y, a través de ellos, podremos aprender los conocimientos básicos para crearlos. Este repositorio cuenta con 44 contratos inteligentes, que se reparten en 26 temáticas o funcionalidades distintas. Esto es porque hay contratos que tienen variantes, bien porque usen distintos métodos para realizar la misma función, bien porque se hayan realizado modificaciones que hacen que el contrato gaste más o menos gas, o bien porque se hayan introducido más funciones a parte de las básicas para hacer el contrato más completo. A parte de estos contratos, en el repositorio podemos encontrar 43 ficheros de pruebas a estos contratos, un fichero para cada contrato excepto en el caso de la interfaz ya que la información encontrada para realizar estos test es muy escasa e insuficiente. Cada fichero cuenta con al menos 3 test para realizar al contrato. Así, empezaremos con los más básicos e iremos avanzando por distintos niveles de complejidad.

También veremos cómo realizar test unitarios a estos contratos; en este caso lo vamos a hacer con Brownie, que es un marco para crear pruebas para contratos inteligentes basado en Python. De esta forma, sabremos si el contrato hace lo que tiene que hacer, si se revierte si no se cumplen las clausulas iniciales y si los datos son correctos. Si los contratos pasan estos test diremos que son aptos para su uso.

Antes de empezar a aprender cómo realizar contratos inteligentes, vamos a ver el entorno en el que se desarrollan, ya que juega un papel fundamental. Este entorno son las cadenas de bloques y Ethereum. También vamos a ver las ideas básicas de Vyper, así como los programas necesarios para poder usar Vyper y testear los contratos.

También vamos a debatir o a hablar sobre el gasto de gas que se produce en el contrato y si nos interesa gastar más por una lectura más clara o, por el contrario, si nos interesa gastar menos y tener una lectura menos clara.

El resto de la memoria se divide como sigue:

Un primer capítulo explicativo de los elementos que rodean a los contratos inteligentes, así como como las ideas básicas de estos y de Vyper.

Un segundo capítulo explicativo del lenguaje Vyper y sus características usando diversos ejemplos.

Un tercer capítulo explicativo de dos contratos inteligentes escritos en Vyper.

Un cuarto capítulo explicativo de la creación de test con Brownie para estos contratos.

Un quinto capítulo explicativo de los gastos de gas dependiendo de cómo se escriba el contrato.

Un sexto capítulo exponiendo las conclusiones al realizar este trabajo y el posible trabajo futuro.

Un capítulo final con un resumen de la memoria en inglés.

El código de los contratos inteligentes y de los test realizados para este trabajo está disponible en `https://github.com/maria988/Contratos-inteligentes`.

Capítulo 2

Preliminares

En esta sección vamos a explicar y entrar en más detalle en los conceptos mencionados en el capítulo anterior. Primero vamos a hablar de la cadena de bloques, de cómo funciona y de sus componentes. Seguidamente hablaremos de la forma que tienen los contratos inteligentes y de su almacenamiento en Ethereum. A continuación explicaremos qué es un test y la necesidad de realizarlos en contratos inteligentes. Con estas bases nos adentraremos en el lenguaje de programación de contratos inteligentes Vyper, veremos qué permite y qué no, y daremos una comparación con Solidity, otro lenguaje de contratos inteligentes.

2.1. Cadena de bloques

La cadena de bloques [16, 1], como vimos en el capítulo anterior, es una especie de libro de registros, por ejemplo, es como el libro de movimientos de un banco. Esta cadena de bloques está alojada en cada nodo de la red, es decir, cada usuario de Ethereum tiene el mismo libro. Sigamos con el ejemplo anterior, el libro de movimientos del banco está formado por distintos registros, los cuales se encuentran cifrados por seguridad del cliente, ya que cada registro cuenta con información del cliente y del dinero transferido. Imaginémonos que queremos pagar el alquiler de nuestra vivienda, entonces quedaría registrado que nos quitan de nuestra cuenta el valor del alquiler y en la cuenta del arrendador le sumarían este valor. Por tanto, ha habido una transacción económica, la cual ha sido realizada y verificada por los bancos del arrendador y del arrendatario. Nuestro símil a estos registros serían los bloques. Un bloque, al igual que un registro del banco, almacena información sobre la transacción realizada, donde la transacción [11] no es más que un mensaje que se manda de una cuenta a otra.

En Ethereum en vez de tener una cuenta bancaria, como es en el caso del banco, tenemos una cuenta a la cual se le asocia una cantidad de Ether. Esta cuenta funciona como una cartera personal e intransferible, la cual esta controlada por una clave privada. Existen otro tipo de cuentas, las cuentas de contrato, que están controladas por el contrato que está almacenado con la cuenta.

Imaginemos que queremos pagar el alquiler de la vivienda utilizando un contrato inteligente de Ethereum. Primero se comprueba que tenemos suficiente Ether para realizar la operación. Si no es el caso, no se realiza, pero si hay suficiente se almacena en un bloque la acción que se quiere realizar. Esta operación tiene que validarse, es decir, se tiene que almacenar en todos los nodos de la red. Se transmite el bloque con la información a todos los nodos y estos verifican que lo han almacenado. En este momento, el bloque se puede añadir a la cadena que había y completarse la operación añadiendo el Ether correspondiente a la cuenta de la cartera del arrendador. Pero esta operación no está registrada de forma definitiva. En el bloque que se encuentra nuestra operación, se van a ir almacenando más transacciones hasta llegar a llenar el bloque (la cantidad de transacciones en un bloque depende del tamaño de las mismas). Cuando el bloque está lleno hay que sellarlo, para que quede almacenado de forma definitiva y no se pueda modificar ninguna transacción del bloque. A esto se le conoce como minería de Ethereum, que es la creación de nuevos bloques en la cadena de bloques.

Esta acción recibe este nombre por su similitud a la minería de oro. Por tanto, validar o sellar un bloque es minarlo y los mineros son los usuarios que minan. Estos mineros en vez de tener una batea o un detector de metales tienen ordenadores potentes que usan funciones Hash o funciones resumen [3]. Estas funciones utilizan un algoritmo para crear una cadena finita de caracteres, un Hash. Este se crea a partir de los datos de entrada de la función , es decir, la función transforma los datos de entrada en una cadena alfanumérica. Por ejemplo, usando un creador de Hash en línea, si codificamos “contrato inteligente” con el algoritmo SHA1 [17] obtenemos “906097c1183caf2f17ccbb9d3c84e6c71ea7abfb”. Estas funciones se caracterizan por ser de un solo sentido, es decir, del Hash no vamos a poder obtener los datos de entrada. Otra característica muy importante es la asignación de un Hash único a cada conjunto de datos, por poco que varíen estos.

Los mineros [12] son responsables de crear tokens de Ether al realizar esta operación. Un token es una unidad de valor que carece de valor de curso legal. Veamos un ejemplo para entender esto de forma más clara: las fichas de un casino solo tienen valor dentro de este, las puedes canjear por dinero dentro de él, pero fuera del casino las fichas carecen de valor, esto mismo pasa con los tokens. Así, los mineros pueden recibir una cantidad de tokens de Ether. No todos los mineros reciben una compensación por intentar minar un bloque, solo lo consigue el primero que lo haga. Así, dependiendo del esfuerzo computacional usado, medido con gas, se les compensa con más o menos tokens. Cuando el

bloque esté minado, se enviará a los demás nodos de la red esta información y quedará registrado en todos.

Estos tokens [15] no tienen porqué tener un valor monetario, pueden se una acción de una empresa, una casa, cualquier cosa que queramos. Estos tokens se registran en la cadena de bloques de forma que se almacena quien posee dicho token y lo que representa. Pueden ser creados para la necesidad de cada persona, por ejemplo, una empresa quiere vender acciones. En este caso, esta empresa crea un token para que quién compre sus acciones tenga uno o varios de estos tokens, que serían equivalentes a la compra realizada. De esta forma, el propietario de estos tokens sabe que tiene unas acciones las cuales pueden ser vendidas y, en este caso, este token pasaría a ser de la persona que las ha comprado.

El Ether es una unidad divisible y consta de distintas unidades. La unidad más pequeña e indivisible es el Wei, un wei es equivalente a 10^-18 Ether. Otra unidad del Ether importante es el Gwei, ya que es la unidad en la que se mide el gas. Un Gwei equivale a 10^9 Wei o a 10^{-9} Ether. Hay otras unidades del Ether, la escala en orden ascendente sería Wei, Kilowei, Megawei, Gigawei (GWei), Microether, Milliether, Ether y Kiloether. La diferencia entre cada unidad es de 10^3, entre cada una en el orden anterior, es decir 1 Wei es 10^3 Kilowei, 1 Kilowei es 10^3 Megawei, y así sucesivamente.

Cada bloque de la cadena cuenta con el conjunto de transacciones, el Hash del bloque, una marca de tiempo y un puntero Hash que enlaza al bloque anterior, es decir, están unidos por el Hash del bloque anterior. De esta manera cambiar, eliminar o modificar el bloque es una tarea imposible, puesto que si se varía un bloque se han de variar los bloques anteriores, por tanto, la cadena de bloques es inmutable. Podemos pensar qué pasa si tenemos dos acciones que se contradicen, ¿pueden registrarse siendo que se están contradiciendo? No pueden, se registra la que se ha realizado en primer lugar y se descarta la segunda. Por lo que son seguros y, al ser una tecnología distribuida, la información está en todos los nodos de la red. Por tanto, la cadena de bloques contiene información perpetua, la cual no se puede eliminar, cambiar o mover pero sí consultar.

Para terminar, vamos a hablar un poco sobre Bitcoin y su diferencia con Ethereum. Bitcoin [14] es una red descentralizada de igual a igual, es decir, todos los nodos de esta red son iguales no hay ninguno que sea especial. El propósito principal de Bitcoin es la descentralización del comercio creando una criptomoneda, el bitcoin, que no dependiese de ningún Banco central o gobierno, es decir, que no hubiese una entidad central que lo controlase. Por tanto, Bitcoin es un sistema de pago alternativo y descentralizado, mientras que el propósito de Ethereum es quitar intermediarios en las transacciones. La cantidad de bitcoins que puede haber en circulación tiene un tope de 21 millones mientras que la cantidad de Ether es ilimitada. Bitcoin no cuenta con la posibilidad de crear contratos inteligentes ya que, por su propósito, funciona como si fuese un

monedero: se puede enviar y recibir bitcoins [4] y tenerlos como inversión.

2.2. Contratos inteligentes

Un contrato inteligente es un programa informático que consta de sus funciones y del estado de sus datos. Ambas partes están alojadas en una determinada dirección en la cadena de bloques de Ethereum, en concreto se alojan en una dirección de contrato. Veamos un contrato muy simple para ver la estructura global de estos. Este ejemplo consta de una variable global, que va a ser un número natural, que indica las veces que se ha llamado a la función `anadir` del contrato. Si se ha llamado más de 3 veces a esta función el valor de la variable será 0 y en otro caso se sumará 1 a la variable global.

El contrato quedaría de esta forma:

```
llamadas: uint256
def __init__(valor: uint256):
    self.llamadas = valor

def anadir():
    assert self.llamadas > 0
    if self.llamadas >= 3:
        self.llamadas = 0
    else:
        self.llamadas = self.llamadas + 1
```

Estos datos van a ir variando dependiendo de la evolución del contrato, la cual es proporcionada por sus funciones. Los datos o variables globales del contrato se declaran al principio de este. En nuestro ejemplo, sólo tenemos la variable `llamadas`.

Después de las variables se crea el constructor, mediante el cual podemos dar un valor inicial a determinadas variables globales. Quién inicializa un contrato tiene que elegir los valores de inicio de este. En nuestro ejemplo, vamos a inicializar este contrato con el valor 0, es decir, vamos a dar a `llamadas` el valor inicial de 0.

Después del constructor están las demás funciones del contrato. En nuestro caso solo tenemos la función `anadir`. Podemos dividir la estructura de las funciones en dos. Una parte comprueba que se puede usar la función mediante asertos o afirmaciones y otra parte es el cuerpo de la función. Gracias a estos

asertos podemos hacer que la función sólo se pueda usar bajo determinadas circunstancias o por determinadas personas, por tanto, los asertos son optativos. En nuestro ejemplo, el aserto es `assert self.llamadas >0`, y tenemos que el valor de `self.llamadas` es 0, por tanto tenemos $0 > 0$ que es falso. En este caso, se revertirá la función, es decir, no nos dejará seguir y si se hubiese realizado algún cambio, se desharía.

Imaginemos que en vez de haber inicializado el contrato a 0 lo inicializamos a 1 y llamamos a la función `anadir`. En este caso, sí se cumple el aserto ya que $1 > 0$ es cierto, ahora sí que podemos acceder al cuerpo de la función. Nuestro cuerpo, en esta función, es una declaración `if-else`, por tanto hay dos opciones. Como el valor de `llamadas` es 1, entraríamos en el `else` y se incrementaría el valor de esta variable a 2. Si llamásemos una vez más a `anadir`, haría el mismo recorrido y el valor de `llamadas` sería de 3. Si volviésemos a llamar, como `llamadas` tiene asignado el valor 3, se cumpliría la condición del `if` y se actualizaría el valor de `llamadas` a 0.

El contrato se va ejecutando y sus acciones se van registrando en la cadena de bloques. De esta forma, queda registrado lo que ha ocurrido en el contrato, ya sea un pago, la recepción de un objeto, el cambio de valor de una variable, etc. Para que un contrato pueda ser ejecutado, uno o varios participantes de este deben de pagar una cantidad de Ether en forma de gas para realizar esta ejecución. Esto es para suplir el coste computacional de ejecutarlo, sería como el equivalente a pagar la electricidad para que funcionen el ordenador, la televisión, el microondas,etc. Estos contratos y sus transacciones son seguros, ya que el bloque en el que se encuentran es inamovible al tener este vinculación con el bloque anterior. Hay una única posibilidad de borrar código de la cadena de bloques y es cuando el contrato llama a la función `selfdestruct`. En este caso, el Ether que hay en el contrato se manda a la cuenta que se haya designado previamente y se eliminan el código y el almacenamiento de los datos.

2.3. Test

El testing [13] de un programa informático es la realización de pruebas sobre este para poder obtener datos de la calidad del mismo. Se realiza para poder encontrar fallos en el código y poder arreglarlos. Dentro de los test hay distintos tipos de pruebas como las de integración, de aceptación, de seguridad, de rendimiento, de unidad, etc. Nosotros nos vamos a centrar en los test de unidad que son los que usaremos en nuestros contratos.

El test de unidad [10] es una comprobación sobre una unidad de código, es decir, no se comprueba todo el programa, sino que se comprueba que una parte del código está bien hecha, funciona correctamente y tiene lógica. Este tipo de pruebas son ventajosas ya que sabemos que la parte del código que hemos

comprobado funciona correctamente, así no tendremos que esperar a tener todo el código completo, y si añadimos más código a lo que ya habíamos probado, tendremos la certeza de que la unidad comprobada está bien construida. Los fallos se encuentran más fácilmente puesto que el código que se prueba es una parte y no todo. Estas pruebas pueden ser consideradas como documentación del código ya que, con ellas, podemos ver como funciona el mismo. A comparación de los demás test, son muy rápidas. Pueden realizar la prueba en milisegundos, por lo que las podemos ejecutar a menudo y realizar pruebas asiduamente.

Los contratos se han de verificar o se han de probar antes de almacenarlos en Ethereum. Esto es porque, al almacenarse en la cadena de bloques, el contrato no puede ser modificado. Si tiene algún error o no funciona correctamente, el contrato quedará almacenado con este error. Por tanto, es muy importante realizar las comprobaciones necesarias. Como vimos anteriormente, vamos a usar Brownie [6] para comprobar que los contratos sean correctos, es decir, hacen lo que tienen que hacer y funcionarán en todos los casos posibles. Con Brownie vamos a realizar pruebas unitarias, por tanto, si el test da error podemos corregir el contrato fácilmente. Debemos probar que los datos iniciales son correctos, que se revierte la acción si no se cumplen las afirmaciones iniciales de cada función y que los datos se actualizan correctamente. Brownie cuenta con una consola para interactuar con los contratos directamente y así probar su uso. Trabaja con Pytest que es un marco para crear pruebas simples. De esta forma, los test que se crean son más sencillos. Esto es porque Pytest cuenta con funciones que simplifican el código del test y la prueba del contrato.

2.4. Vyper

Vyper [7] está influenciado por Python, pero aunque la sintaxis de Vyper sea válida en Python, no se da el caso contrario. Una diferencia para ver que toda la sintaxis de Python, o sus funciones, no son aceptadas en Vyper, es que los bucles `while` no están definidos y los bucles `for` tienen que estar definidos en un determinado rango. El tamaño del intervalo tiene que ser finito y no puede depender de variables globales. Con este ejemplo vemos que Vyper no permite bucles infinitos, por tanto ¿es Turing completo? A diferencia de Solidity, Vyper no es Turing-completo aunque no es necesario que lo sea. Por una parte no es Turing-completo ya que no deja que se ejecute una función por tiempo ilimitado cosa que, una máquina de Turing universal, sí puede. Por otra parte, no es necesario que lo sea, puesto que la EMV no es Turing-completo ya que tiene limite de gas para cada bloque.

Este lenguaje es simple y seguro. Es simple ya que su objetivo es ser lo más legible y entendible posible para que cualquier lector, esté o no familiarizado con la programación, pueda comprender lo que está haciendo el contrato en cada momento. Es seguro gracias a esta sencillez. Según los desarrolladores de

Vyper [7]:

> "Prohibirá deliberadamente las cosas o dificultará las cosas si lo considera oportuno con el objetivo de aumentar la seguridad."

Para poder garantizar esta sencillez cada contrato se almacena en un único fichero; no se puede escribir un contrato formado por distintos ficheros para reutilizar código, ya que no sería simple. Mientras que Solidity [11, 9] es un lenguaje de programación orientado a objetos, Vyper carece de herencia de clases. Tampoco cuenta con sobrecargas de operadores ni de funciones, cosa que Solidity sí, ya que así, el lenguaje sería más complicado o incluso engañoso. Por ejemplo, si sobrecargásemos el operador +, siendo la suma usual para todos los números excepto para los múltiplos de 5, que en este caso se suma 5 al valor de la suma usual(por ejemplo 5 + 10 = 20). Si usamos este operador en una declaración `if-else` para verificar que la suma de dos variables es mayor que un determinado valor (por ejemplo, `if a + b >= 20`, donde a y b son variables globales), se puede dar el caso de que se ejecute una parte de esta declaración sin realmente verificarse la cláusula. Por ejemplo, en el caso de que a y b fueran 5 y 10, respectivamente, y tuviésemos la declaración `if a + b >= 20`, se ejecutaría el cuerpo del `if` aunque realmente no debería. Esto podría dar lugar a que se realizasen transacciones o cambios en los datos que no debiesen producirse. También podría producir un error si este operador se usase para enviar Ether, podría mandar una cantidad de Ether mayor de la que dispone el contrato, lo cual es imposible y produciría un error.

Este lenguaje comprueba los limites de las listas y bucles que usa, es capaz de calcular un límite superior para cada función y, así, ajustar el gas necesario para llamarla. De esta forma, no permite llamadas recursivas a funciones, ya que no podría establecer estos límites y podría sufrir falta de gas. En cambio, se permite realizar llamadas a determinadas funciones que están definidas dentro del mismo contrato. Estas funciones no pueden ser llamadas externamente, solo pueden ser llamadas por otras funciones del contrato. Cuenta con soporte para enteros y números de coma fija decimal, haciendo que los cálculos sean más exactos. Mientras que en Solidity los números están en punto binario, lo que hace que se trabaje siempre con aproximaciones.

Acabamos de ver que por un lado Vyper cuenta con sencillez y seguridad, mientras que Solidity cuenta con libertad. Esta libertad puede ser útil para realizar contratos más complejos pero tendríamos que tener cuidado con los bucles, ya que estos no pueden ejecutarse de forma ilimitada. Esto es porque el gas que se les proporciona es limitado, si hubiese un bucle infinito, habría una excepción de falta de gas y, en este caso, se revertirían las modificaciones realizadas en esta actualización.
Los programas que vamos a usar y necesitamos instalar son:

- Vyper.
- Pytest.
- Brownie.
- Node.js es un entorno de ejecución para JavaScript.
- Ganache-cli es un emulador de blockchain rápido y sin los gastos de ejecutar un nodo en Ethereum.

Con todo esto, ya tenemos las bases necesarias para conocer las determinadas características de Vyper y poder escribir contratos inteligentes. Esto lo desarrollaremos en el capítulo siguiente.

Capítulo 3

Vyper mediante ejemplos

En esta sección vamos a aprender a realizar contratos inteligentes en Vyper. Vamos a ver las distintas características de este lenguaje ilustrándolo con ejemplos, como la estructura, los tipos, las funciones, etc. Todos los ejemplos que van a aparecer son fragmentos de código de contratos inteligentes, mediante la palabra previa subrayada podemos acceder al contrato completo. Estos fragmentos son partes de código de algunos contratos inteligentes del repositorio. Además de estos contratos que vamos a ver, hay más contratos inteligentes como alquilar una casa, un testamento, alquiler de un vehículo,etc. En el repositorio también están los test de cada contrato, que realizan distintas pruebas para comprobar que el contrato funciona bien y como debería. Habrá algunos ejemplos que se indique que son modificaciones; en este caso no habrá contrato que contenga dicho ejemplo.

3.1. Estructura

En el capítulo anterior esbozamos la estructura de los contratos inteligentes. Veamos ahora todas las partes y detalles de la estructura de estos contratos.

Al principio del contrato hemos de poner la versión pragma, que es una especificación para que el compilador sepa con qué versión se ha de compilar. Esta información evita que el contrato se compile en versiones nuevas que puedan introducir cambios incompatibles con el contrato. El siguiente ejemplo de versión pragma compilaría en la versión 0.2.8

```
1 # @version ^0.2.8
```

Las variables globales o de estado del contrato se declaran al principio poniendo del tipo qué son cada una. En la siguiente sección veremos los distintos tipos que hay. Por ejemplo:

```
tope: uint256
```

La variable `_tope` es del tipo `uint256`, que son los números naturales. Accedemos a las variables mediante el objeto `self` para consultar su valor o para cambiarlo.

```
self.tope = 12
```

También podemos personalizar el tipo de datos que queremos y agrupar distintos tipos ya existentes. Lo realizaremos usando la palabra clave `struct` de la siguiente forma:

```
struct Comida:
    nombre : String[10]
    descripcion : String[30]
    precio: uint256
```

Así hemos creado un tipo de datos llamado `Comida` que almacena dos cadenas de caracteres de, a lo sumo, 10 caracteres y un natural.

Si vamos a usar este tipo de datos personalizado en alguna variable global, lo debemos declarar y definir antes de usarlo.

También podemos crear eventos, que son datos que queremos registrar. Los clientes del contrato pueden encontrar fácilmente estos eventos ya que se indexan las cuentas a las que va dirigido. Para realizar los eventos tenemos la palabra clave `event`. De esta forma, un evento podría ser:

```
event Transaccion:
    receptor: indexed(address)
    emisor: indexed(address)
    valor: uint256
```

En este evento se han indexado dos cuentas, la del emisor y la del receptor de la transferencia, por tanto estos dos clientes del contrato podrán encontrar fácilmente que se ha realizado una `Transferenca` con un determinado `valor`.

Podemos crear interfaces para que dos contratos interactúen. Una interfaz es un conjunto de definiciones de funciones que no están definidas en el contrato en el que se van a usar, si no que pertenecen a otro contrato externo, y sirve para poder llamar desde el contrato actual al externo. Para ello usamos la palabra clave `interface`. Por ejemplo :

```
interface Puntos:
    def acumularpuntos(puntos:uint256,cliente: address):
        nonpayable
    def usarpuntos(cliente:address) -> uint256:nonpayable
    def usarlitros(cliente:address,puntos: uint256): nonpayable
    def nuevocliente(cliente:address): nonpayable
    def dejardesercliente(cliente:address): nonpayable
```

En este caso se han definido cinco funciones que pertenecen a un contrato externo para usarlas en el contrato actual.

Después de declarar las variables, eventos e interfaces se definen las funciones. En el capítulo anterior vimos la estructura de estas. Hay distintos tipos de funciones como la pura, la pagable y de vista que determinan la mutabilidad de la función. Funciones que son visibles al usuario o que no lo son, así como funciones predefinidas en Vyper o funciones especiales.

Iremos viendo estas funciones y conceptos mencionados en esta sección más adelante.

3.2. Tipos

En está sección vamos a ver los distintos tipos que podemos asignar a las variables. Al declarar las variables debemos de indicar del tipo que son ya que Vyper usa tipado estático y necesita saber el tipo de las variables en tiempo de compilación. Así, un lenguaje con tipado estático es aquel que necesita indicar el tipo de las variables antes de que sean usadas. No hace falta definir al inicio todas las variables que se van a usar, sólo las globales, es decir, las que se van a usar en distintas funciones del contrato.

Podemos separar los tipos en dos grandes grupos: los tipos de valor y los tipos de referencia. Los tipos de valor tienen almacenado en su memoria el valor

correspondiente. A este tipo pertenecen los booleanos, los números, las cuentas, los *arrays* y los *strings*. Mientras que los tipos de referencia son demasiado grandes como para almacenar todos sus valores, este tipo transmite el lugar de almacenamientos de estos. A este tipo pertenecen las listas de tamaño fijo, las estructuras y las asignaciones.

3.2.1. Tipos de valor

Booleanos

El valor booleano almacena el valor lógico el cual puede ser verdadero o falso. Podemos asignar este valor a una variable usando la palabra `bool`. Por ejemplo:

```
1 vacunado: bool
```

Este tipo puede usar distintos operadores lógicos. Por un lado, tenemos los que usan solo un valor booleano, en este caso tenemos la negación `not`. Por otro lado tenemos los que usan dos valores booleanos, estos son la conjunción lógica `and`, la disyunción lógica `or`, la igualdad `==` y la desigualdad `!=`.

El valor inicial de este tipo es `False`, es decir, antes de asignarle un valor tiene el valor falso. En nuestro ejemplo, el valor de `vacunado` cuando es declarado es `False`.

Números

Tenemos tres tipos de números: enteros con signo, enteros sin signo o positivos y decimales.

Los enteros con signo asignan un valor entero con signo. Este valor comprende entre -2^{127} y $2^{127} - 1$, ambos incluidos. Para asignar a una variable este tipo usamos `int128`. El valor inicial de este tipo es 0. Por ejemplo:

```
1 indice: int128
```

Los enteros sin signo o enteros positivos almacenan un valor entero comprendido entre 0 y $2^{256} - 1$, ambos incluidos. Asignaremos este tipo a una variable usando `uint256`. El valor inicial de este tipo es 0. Por ejemplo:

```
tope: uint256
```

No podemos escribir ninguno de estos dos tipos usando el "." aunque la parte decimal sea 0, es decir, no podemos asignar a estos tipos el valor 12.0.

Los decimales almacenan un valor decimal entre -2^{127} y $2^{127} - 1$, ambos incluidos. Este valor decimal tiene una precisión de 10 decimales, es decir, puede almacenar hasta diez decimales. El valor inicial de este tipo es 0.0 . Por ejemplo:

```
maximo: public(decimal)
```

Los operadores para los tres tipos son los mismos. Tenemos dos tipos de operadores: las comparaciones, que devuelven un valor booleano, y los operadores aritméticos, que devuelven un valor del mismo tipo de las variables que lo usan.

Los operadores que devuelven un valor booleano comparan dos tipos numéricos iguales. Los distintos operadores son el menor que `<`, el menor o igual que `<=`, el igual `==`, el distinto `!=`, el mayor que `>` y el mayor o igual que `>=`.

Los operadores que devuelven un valor son la adición `+`, la sustracción `-`, la multiplicación `*`, la división `/`, la exponenciación `**` y el módulo `%`. El valor que devuelven es del mismo tipo que las variables que usan el operador. Estas variables deben ser del mismo tipo,es decir, no podemos comparar ni usar un operador aritmético con dos tipos numéricos distintos.

Cuenta

Este tipo almacena una dirección de Ethereum, la cual tiene que estar en notación hexadecimal precedida con `0x`. Las direcciones deben tener una suma de verificación, es decir, deben tener una función que compruebe que la cadena no haya podido sufrir cambios y sea correcta. Para asignar este tipo a una variable usamos `address`. Por ejemplo:

```
protectora : public(address)
```

El valor inicial de este tipo es 0x0000000000000000000000000000000000000000. El tipo `address` consta de las siguientes propiedades:

- Cantidad de Ether que tiene la cuenta. Para ello usamos `.balance` y nos

devuelve un valor del tipo `uint256`.

- Tamaño del código en la dirección. Usamos `.codesize` y nos devuelve un valor del tipo `uint256`, que es la cantidad de bytes que usa el código.
- Saber si un contrato está implementado en está dirección. Usamos `.is_contract` y devuelve un valor booleano.
- Saber el hash de una cuenta de contrato. Para ello usamos `.codehash` que devuelve un tipo `byte32`, que es el hash del contrato. En el caso de que no haya un contrato implementado en la dirección nos devuelve el valor inicial del tipo `byte32`.

Arrays

Los *arrays* pueden ser de dos tipos: de 32 bits o de longitud máxima fijada. En estos tipos se almacenan cadenas de bytes, cadenas de hexadecimales o cadenas binarias.

Los *arrays* de 32 bits, son cadenas de 32 bits de tamaño. Para asignar este tipo usamos `bytes32`. El valor inicial de este tipo es 0. Los operadores que tiene este tipo son:

- `keccak256(a)`, dada un *array* de 32 bits devuelve el hash creado por una determinada función criptográfica llamada keccak256, el hash de salida es del tipo *array* de 32 bits.
- `concat(a,b, ...)`, toma distintos arrays de 32 bits y los concatena.
- `slice(a, start=inicio, len=tamaño)`, dado un *array* de 32 bits devuelve una porción de este que comienza en `start` y tiene tamaño `len`. Estos dos valores deben ser del tipo `uint256`.

Los *arrays* de longitud máxima fijada se asignan con `Bytes[máximo]`, donde `máximo` es un número natural que indica el máximo número de bytes que va a contener la variable. Este tipo se inicializa con los bytes configurados en $\backslash x00$.

Strings

El tipo *string* le usamos asignando `String[máximo]`, donde `máximo` es un natural que representa el número máximo de caracteres que puede contener este *string*. Por ejemplo:

```
microchip : public(String[15])
```

3.2.2. Tipos de referencia

Estructuras

Las estructuras, las cuales he mencionado anteriormente en la sección 3.1, son un tipo de datos personalizado el cual puede contener cualquier otro tipo excepto asignaciones, las cuales veremos más adelante.

Veamos un ejemplo de cómo se asigna un valor de este tipo. En este ejemplo queremos almacenar en cada número de la carta un plato. Para ello creamos un tipo personalizado llamado `Comida` que consta del nombre del plato, de una breve descripción y del precio de este.

```
struct Comida:
    nombre : String[10]
    descripcion : String[30]
    precio: uint256
carta : public(HashMap[uint256,Comida])
...
def __init__(plato: String[10],ingredientes:String[30],_precio:
    uint256,...):
    ...
    self.carta[self.indice]=Comida({nombre:plato,descripcion:
        ingredientes,precio:_precio})
    ...
```

En `self.carta[self.indice]` almacenamos el plato, concretamente este plato será el número `self.indice`. Sabemos donde lo vamos a guardar, pero debemos crear el plato. Para ello debemos asignar a cada argumento del tipo `Comida` su valor correspondiente, y esto lo debemos de encapsular dentro del tipo. Para asignar a cada parámetro su valor, escribimos el nombre del parámetro seguido de : y el valor que le corresponda. Esto lo hacemos así `Comida(nombre:plato,descripcion:ingredientes,precio:_precio)`. Ahora, igualando ambas partes asignamos el plato creado al número de la carta. De esta forma se ha creado un plato del tipo `Comida` que tiene un `nombre` con valor `plato`, una `descripcion` con valor `ingredientes` y un `precio` con valor `_precio`.

Si queremos consultar el valor de un parámetro de una variable que es de

este tipo usamos `.parametro` donde `parametro` es el parámetro de la estructura que queremos consultar. Veámoslo con un ejemplo:

```
def pedir(numero: uint256, terminado: bool):
    plato:String[10] = self.carta[numero].nombre
    ...
    self.preciototal += self.carta[numero].precio
    ...
```

En este ejemplo consultamos el nombre mediante `self.carta[numero].nombre` y el precio mediante `self.carta[numero].precio` del plato número `numero`.

Listas de tamaño fijo

Las listas de tamaño fijo contienen un único tipo de elementos, el cual es especificado a la hora de su declaración. Para asignar esta lista a nuestra variable usamos `tipo_elemento[máximo]` donde `tipo_elemento` es el tipo de los elementos de la lista y `máximo` es el máximo de elementos de la lista. El primer elemento de la lista se encuentra almacenado en el índice 0 de esta. El valor inicial de la lista es el valor inicial de sus elementos. Un ejemplo de este tipo puede ser:

```
struct Calles:
    uso: bool
    cliente: address
    tope: uint256
    combustible: String[3]
    pagado: uint256
    selec: uint256
surtidores: public(Calles[2])
seleccion: public(uint256[7])
```

Aquí tenemos dos listas distintas `surtidores` que almacena dos elementos del tipo `Calles` y `seleccion` almacena 7 elementos del tipo `uint256`.

Asignaciones

Las asignaciones son tablas hash, que es una estructura de datos en la que a cada clave le asigna un valor y utiliza una función hash para calcular el índice en el que está almacenado el valor de la clave. Este tipo se inicia virtualmente, lo

que hace que todas las claves posibles existan y estas tengan asociadas un valor inicial, el cual es el valor inicial del tipo asociado a la clave. Para usar la asignación usamos `HashMap[tipo_clave,tipo_valor]`. Por ejemplo, vamos a almacenar en cada valor posible del tipo `uint256` un tipo `Juego`.

```
struct Juego:
    apostador: address
    equipo1: uint256
    equipo2: uint256
    apuesta: uint256
apostadores: HashMap[uint256, Juego]
```

El valor inicial del tipo `Juego` está compuesto por el valor inicial de cada uno de sus componentes, es decir, tendrá el valor inicial (0x0...,0,0,0). Mientras que el valor inicial de esta asignación será el valor inicial de Juego, para cada una de sus claves.

3.3. Estructuras de control

Las estructuras de control son los elementos que vamos a usar para hacer que el contrato haga lo que queremos. Las estructuras de control van a ser las funciones, las declaraciones `if-else` y los bucles `for`, que van a dar funcionalidad a nuestro contrato haciendo que se tome una dirección u otra.

3.3.1. Funciones

Las funciones son unidades de código que se pueden ejecutar dentro de un contrato. Las funciones se pueden llamar interna o externamente. Para indicar si una función es interna o externa debemos poner un decorador antes de declarar la función.

Las funciones externas utilizan el decorador `@external`, esto indica que estas funciones solo pueden ser llamadas externamente o desde otros contratos, es decir, forman parte de la interfaz del contrato. Por tanto, no podemos llamar a una función externa desde una función externa dentro del mismo contrato. Así vemos que Vyper, de verdad, no permite las llamadas recursivas dentro de un contrato.

Por otro lado, tenemos las funciones internas las cuales solo pueden ser llamadas por funciones del contrato en el que estén. Para declarar una función interna

usamos el decorador `@internal`. Podemos acceder a estas funciones utilizando el objeto `self`. Veamos el siguiente ejemplo:

```
@view
@internal
def _terminado() -> bool:
    return block.timestamp > self.termina

@view
@external
def terminado()-> bool:
    return self._terminado()
```

Podemos ver que la función externa `terminado` llama a la función interna `_terminado`. Como notación a las funciones internas las ponemos un _ delante del nombre. Las funciones internas han de estar definidas antes de que sean llamadas, es decir, si en el ejemplo anterior estuviese la externa antes de la interna, se produciría un error a la hora de compilar.

Estos decoradores son obligatorios, es decir, cada vez que definamos una función tenemos que elegir si queremos que sea externa o interna. Mientras que los que vamos a ver a continuación son opcionales.

Estos decoradores son para asignar la mutabilidad de las funciones. Tenemos:

- Funciones puras, no leen ni el estado del contrato ni las variables de entorno por tanto no se pueden modificar dichas variables. Asignamos este tipo con el decorador `@pure`.
- Funciones de vista, no altera el contrato pero puede ver el estado del mismo. Para estas funciones usamos el decorador `@view`.
- Funciones no pagables, estas funciones pueden modificar las variables de entorno pero no pueden recibir Ether. Podemos utilizar el decorador `@nonpayable` o no poner nada, ya que las funciones son de este tipo por defecto.
- Funciones pagables, estas funciones pueden modificar variables y recibir Ether. Para que una función sea pagable usamos el decorador `@payable`.

Por ejemplo:

```
@view
```

```
@internal
def _terminado() -> bool:
    return block.timestamp > self.termina

@view
@external
def terminado()-> bool:
    return self._terminado()

@external
@payable
def apostar(eq1: uint256,eq2: uint256):
    ...
```

Como en este caso, la función externa `terminado` es de vista y llama a la interna `_terminado`, la interna ha de ser de vista. Si las interna no lo fuese, se estaría llamando a una función que puede cambiar el estado desde una función que no puede cambiarlo. Si no nos damos cuenta y hacemos esto, a la hora de compilar nos dará un error. La función `apostar` es externa y pagable.

Tenemos otro tipo de decoradores, los de no rentada. Este decorador permite que se realiza una sola llamada desde un contrato externo, y si se vuelve a llamar se revertirá la transacción. Este decorador es `@nonreentrant(clave)`, el cual bloquea la función y todas las demás funciones del contrato que tengan ese mismo valor de `clave`.

Podemos elegir cualquier nombre para las funciones, excepto los nombres de las funciones predeterminadas como `send`, `log`, etc. , porque, como dijimos anteriormente, Vyper no admite sobrecarga de funciones. Por tanto, cada función tiene que tener un nombre distinto.

Tenemos dos funciones especiales que, para usarlas, las debemos de escribir con el mismo nombre. Estas funciones son la función `__default__`, que es la función por defecto y la función `__init__`, que es la constructora.

La función `__default__`, si está definida, es llamada si se llama a una función que no existe en el contrato. En cambio, si no tenemos definida esta función se llamará a la función `REVERT` la cual generará una excepción y se revertirá la llamada al igual que pasa con las afirmaciones. A la hora de definir esta función tenemos que tener en cuenta varias cosas:

- El nombre de la función ha de ser `__default__`, puesto que si no tiene este nombre no sería la función por defecto, si no que sería otra función más del contrato.
- No puede tomar argumentos de entrada.

- No puede devolver valores.
- El decorador de la función tiene que ser `@external`, ya que solo se la puede llamar externamente.
- La función puede ser pagable, usando `@payable`, y, en este caso, si se llama al contrato enviando Ether y sin más datos se llamará a esta función.

La función `__init__` la tenemos que definir siempre, puesto que esta función es la que inicializa el contrato, por tanto, solo se puede llamar cuando se implementa el mismo. Para definir esta función tenemos que tener en cuenta que:

- El nombre de la función debe ser `__init__` para que sea el constructor, si no sería otra función.
- Puede tener argumentos de entrada ya que podemos inicializar algunas o todas las variables globales en esta función.
- No puede devolver valores.
- Debemos de poner el decorador `@external` ya que esta función es llamada a la hora de implementar el contrato a través de una llamada externa.
- La función puede ser pagable, usando `@payable`, y así almacenar en la cuenta del contrato una cantidad de Ether.
- No puede llamar a otras funciones y no puede ser llamada por ninguna.
- Ha de ser la primera función del contrato, es decir, se debe escribir antes que todas las demás funciones del contrato, pero después de todas las variables globales.

En este ejemplo vamos a ver una función `__init__` que es pagable para almacenar inicialmente una cantidad de Ether e inicializa algunas variables.

```
1  casa:public(address)
2  inicial:public(uint256)
3  empieza:public(uint256)
4  termina: public(uint256)
5  ...
6  @payable
7  @external
8  def __init__( tiempo_inicio: uint256,duracion: uint256):
9      assert tiempo_inicio > 0
10     assert duracion > 0
11     self.inicial = msg.value
```

```
12      self.casa = msg.sender
13      self.empieza = block.timestamp + tiempo_inicio
14      self.termina = self.empieza + duracion
```

3.3.2. Declaraciones `if`

Estas estructuras nos permiten hacer que el contrato vaya por un camino u otro, dependiendo de las condiciones que se cumplan. La declaración `if` comprueba una condición, la cual debe ser booleana, es decir, no podemos escribir algo como `if 0` ya que Vyper no contiene conversiones implícitas de tipos. Si la condición es cierta, se ejecutará el cuerpo del `if`, mientras que si no lo es, no entrará a este. Podemos usar también la estructura `if-else` o también `if-elif-else`, para poder comprobar distintos casos y en el caso de no cumplirse ninguno de los indicados, que haga otra cosa. De esta forma podemos tener multitud de caminos diferentes que puede tomar nuestro contrato. Por ejemplo, queremos crear una función que recorra todas las apuestas realizadas y que si se ha acertado la apuesta, se mande el Ether a la cuenta correspondiente.

```
1  @external
2  def devolver():
3      ...
4      nive:uint256 = self.sigindice
5      for i in range (nive,nive+30):
6          if i >= self.indice:
7              selfdestruct(self.casa)
8          else:
9              if self._ganado(self.apostadores[i]):
10                 send(self.apostadores[i].apostador,
11                     self._ganar(self.apostadores[i]))
12             self.apostadores[i]= empty(Juego)
13     self.sigindice = nive + 30
```

En esta función podemos ver que dentro del bucle `for`, del cual hablaremos en la siguiente sección, se comprueba que el índice `i` sea mayor o igual que el `self.indice`, que es la cantidad de apuestas hechas mas una. Si esto ocurre, el contrato terminará, y si no ocurre, comprobará si ha ganado o no la apuesta i-ésima. En el caso de haber ganado se le manda el Ether correspondiente.

3.3.3. Declaraciones `for`

La declaración `for i in <iterable>` nos permite realizar operaciones y comprobaciones sobre cada elemento del iterable elegido. Este iterable puede ser una lista estática, definida previamente en la función o escribimos directamente la lista, o usando la función rango, en la cual hay que definir los limites de este. En ningún caso el iterable puede contener alguna variable global, puesto que así no se podría determinar la cantidad de gas que va a gastar dicha función.

Si usamos la primera opción, la `i` irá tomando los distintos valores de la lista. Esta lista no puede ser multidimensional, es decir, no puede ser una lista de listas.

Si usamos la segunda opción tenemos tres opciones.

- Podemos usar `for i in range(PARAR)`, donde `PARAR` es un literal natural mayor que 0. De esta forma i tomará los valores de uno en uno desde 0 hasta el valor que tenga `PARAR`.
- Podemos usar `for i in range(EMPEZAR,PARAR)`, donde `EMPEZAR` y `PARAR` son literales enteros y `PARAR` es mayor que `EMPEZAR`. Con esta forma `i` tomará los valores de uno en uno desde `EMPEZAR` hasta `PARAR`.
- Podemos usar `for i in range(índice, índice + N)`, donde `índice` es una variable de tipo entero y `N` es un literal natural mayor que 0.

En el ejemplo anterior teníamos está ultima opción ya que queríamos empezar en el primer índice no visitado.

3.4. Variables y constantes del entorno

Las variables y constantes del entono son valores a los que tenemos acceso en todo el contrato.

3.4.1. Variables de entorno

Las variables del entorno se utilizan para saber información de la cadena de bloques, de las transacciones y hacer referencia a variables del contrato.

Tenemos acceso a distintas propiedades para obtener información de la cadena de bloques y de las transacciones. Estas propiedades son:

- Dirección del minero del bloque. Usamos `block.coinbase`, esta llamada nos devuelve un `address`.
- Dificultad de bloque actual. Usamos `block.difficulty` y nos devuelve un `uint256`.
- Número del bloque actual. Usamos `block.number`, que nos devuelve un `uint256`.
- Hash del bloque anterior. Para esto usamos `block.prevhash`, que devuelve un `byte32`.
- Marca de tiempo del bloque actual. Para esto usamos `block.timestamp` que nos devuelve un `uint256`.
- ID de la cadena. Usamos `chain.id` que nos devuelve un `uint256`.
- Gas restante en el contrato. Para acceder a esta propiedad usamos `msg.gas` la cual no devuelve un `uint256`.
- Remitente de la llamada actual. Usamos `msg.sender` la cual nos devuelve un `address`.
- Número de wei enviados en el mensaje. Para acceder a este valor usamos `msg.value` y nos devuelve un `uint256`.
- Dirección de inicialización del contrato. Usamos `tx.origin` y nos devuelve una `address`.

Vamos a ver un ejemplo que usa algunas de estas propiedades. En este ejemplo se usa el `block.timestamp` para saber la hora del bloque actual, el `msg.sender` para saber la dirección desde donde se está llamando a la función y el `msg.value` para saber la cantidad de Ether que se ha mandado en esta llamada.

```
@external
@payable
def apostar(eq1: uint256,eq2: uint256):
    assert block.timestamp <= self.empieza,"Antes de empezar"
    assert msg.sender != self.casa,"Jugador"
    assert msg.value > 0 ,"Apuesta positiva"
    nfi: uint256 = self.indice
    self.apostadores[nfi] = Juego({apostador: msg.sender,
                                   equipo1: eq1,
                                   equipo2:eq2,
                                   apuesta:msg.value})
    self.indice = nfi + 1
```

En este caso, se comprueba que el tiempo del bloque actual es menor que `self.empieza`, que es el tiempo en el que empieza el partido, se comprueba que la dirección que llama a la función no sea `self.casa` y que el valor enviado sea mayor que 0.

A las propiedades `msg.sender` y `msg.value` sólo se puede acceder a través de funciones externas, si se necesita su valor en una función interna, se llama a esta función tomando como argumentos estos valores.

Para acceder a la dirección de contrato desde dentro del mismo se utiliza el objeto `self` y para saber el saldo de este contrato se llama a `self.balance`, que devuelve un `uint256`. También utilizamos `self` para acceder a las variables globales del contrato, saber su valor y poderlas cambiar. Del mismo modo se pueden hacer llamadas a funciones internas, es decir, llamamos a la función interna igual que si fuese una variable del estado sólo que, al ser una función tendremos que añadir unos paréntesis vacíos al final, si no toma argumentos o pasarle los argumentos. Por ejemplo:

```
apuntados:bool
...
@view
@internal
def _ganado(apos: Juego)-> bool:
    valor: uint256 = 0
    return(apos.equipo1 == self.pequipo1)
           and (apos.equipo2 == self.pequipo2)

@view
@external
def ganado(apos: Juego)-> bool:
    assert self.apuntados,"Apuntados"
    return self._ganado(apos)
```

Se llama a la función `ganado` para saber si se ha ganado la apuesta `apos`. Esta verifica que se han introducido los puntos de cada equipo mediante la variable de estado `self.apuntados`. Por último, llama a la función interna `self._ganado(apos)` y esta realiza las comparaciones. La función `_ganado` devuelve un `bool` a la función `ganado` y esta devuelve el mismo valor.

3.4.2. Constantes de entorno

Las constantes de entorno son valores fijos durante todo el contrato. Contamos con dos tipos de constantes las predefinidas y las personalizadas.

Las constantes predefinidas son:

Comando	Devuelve	Valor
ZERO_ADDRESS	address	Valor inicial del tipo
EMPTY_BYTES32	bytes32	Valor inicial del tipo
MAX_INT128	int128	Valor máximo del tipo, $2^{127} - 1$
MIN_INT128	int128	Valor minimo del tipo, -2^{127}
MAX_DECIMAL	decimal	Valor máximo del tipo, $2^{127} - 1$
MIN_DECIMAL	decimal	Valor minimo del tipo, -2^{127}
MAX_UINT256	uint256	Valor maximo del tipo, $2^{256} - 1$

Podemos definir una constante personalizada usando `constant()` al declarar la variable. Por ejemplo:

```
duracion : constant(uint256)= 200
```

3.5. Declaraciones

Las distintas declaraciones de Vyper las podemos separar en flujo de control, que hacen variar el flujo de la función, registro de eventos y afirmaciones, que permiten o no entrar a una función.

3.5.1. Flujo de control

Dentro de este tipo de declaraciones tenemos las declaraciones `break`, `continue`, `pass` y `return`. Todas estas son útiles en las estructuras de control.

La declaración `break` hace que se termine un bucle, aunque no se hayan recorrido todos los elementos. Veamos un ejemplo en el recorremos todos los elementos disponibles con un bucle `for`, comprobando primero si el índice que vamos a consultar es menor que el total de datos. En el caso de serlo comprueba si ha recibido el producto, y en el caso de no haberlo recibido usamos un `break` para salir del bucle.

```
@external
def comprobar():
    ...
    ind: uint256 = self.indi
```

```
    for i in range(ind,ind+20):
        if i >= self.indice and not self.fallo:
            self.siguiente = block.timestamp + self.periodo
            self.indi = empty(uint256)
            self.todos = True
            return
        else:
            if not self.clientes[i].recibido:
                self.fallo = True
                self.indi = i
                break
            elif ...
    if not self.fallo:
        self.indi = ind + 20
```

En este caso si sale del bucle `for` por el `break` no se actualizaría el valor de `self.indi` al siguiente inicio de intervalo correspondiente.

Mientras que `break` te saca del bucle, la declaración `continue` pasa al siguiente valor del elemento a iterar del bucle.

La declaración `pass` es la operación nula, es decir, no hace nada cuando se ejecuta. Esta declaración puede servir como marcador de posición cuando se requiere una declaración sintáctica, es decir, manda la anotación hecha en esta declaración para saber si se ha pasado por ese punto o no.

La declaración `return` devuelve el valor o nada, dependiendo de lo que le siga. Después de un `return` no podemos tener más código en el mismo nivel, ya que no se va acceder a este al haber hecho el `return`. En el ejemplo que vimos para el `break` tenemos también un `return` dentro del bucle `for`, el cual finaliza la función si el valor de `i` es la cantidad de abonados más uno. En este caso sí tenemos más código escrito después del `return`, pero este sí es accesible ya que está a otro nivel.

3.5.2. Registro de eventos

Como vimos al inicio de este capítulo podemos crear eventos para realizar registros, ahora bien, ¿cómo realizamos estos registros? Utilizamos `log NombreEvento(parámetros)`. Por ejemplo:

```
event Pago:
    emisor: indexed(address)
    receptor: indexed(address)
```

```
    valor: uint256
...
@external
def findelcontrato():
    assert msg.sender == self.cliente,"Cliente"
    assert self.terminada,"Terminada"
    log Pago(self.cliente,self.constructora, self.balance)
    selfdestruct(self.constructora)
```

En este ejemplo registramos el evento `Pago` en la función `findelcontrato`, donde `self.cliente` es el emisor, `self.constructora` es el receptor y `self.balance` es el valor que se va enviar. En este caso, se envía el Ether al llamar a la función `selfdestruct()`, la cual manda todo el Ether almacenado en la cuenta del contrato a la cuenta `self.constructora`.

Los datos que podemos incluir en el evento son todos aquellos que no ocupen más de 32 bits, es decir, todos los tipos de valor menos los arrays personalizados con más de 32 bits de longitud. Los parámetros de los eventos pueden ser de dos tipos, los indexados y los de valor.

Los parámetros indexados del evento son las direcciones que queremos que reciban este evento. Para indexar una dirección usamos `indexed(address)`. En el ejemplo anterior tenemos dos direcciones indexadas, el `emisor` y el `receptor`.

Los parámetros de valor son los valores que queremos que reciban las direcciones indexadas. Para declarar estos valores solamente tenemos que poner el nombre del parámetro seguido del tipo que queremos que sea el parámetro. En el ejemplo anterior tenemos el parámetro `valor`, que es de tipo `uint256`.

Es muy importante que el orden de los parámetros al registrar un evento sea el correcto, es decir, si en nuestro ejemplo hubiésemos puesto `log Pago(self.balance,self.constructora, self.cliente)` nos daría un error a la hora de compilarlo y si permutamos `self.cliente` con `self.construtora`, el parámetro `emisor` y `receptor` no tendrán el valor que les correspondería tener.

3.5.3. Afirmaciones y excepciones

Las afirmaciones, como vimos en la sección 2.2, las usamos al inicio de la función para poder acceder a esta, por que nos puede interesar que sólo se pueda acceder a la función bajo determinadas circunstancias y que, en el caso de no cumplirse, se revierta la función. Para usarlas utilizamos `assert`. Continuando con el ejemplo anterior tenemos:

```
cliente: public(address)
terminada: bool
...
@external
def findelcontrato():
    assert msg.sender == self.cliente,"Cliente"
    assert self.terminada,"Terminada"
    ...
```

La variable `self.cliente` es la dirección de la cuenta del cliente y `self.terminada` es un booleano que, cuando se ha terminado la obra, pasa a ser `True`. Para poder realizarse el pago y que termine el contrato se ha de dar que se está llamando a la función desde la cuenta del cliente y se debe de haber terminado la obra. Si alguna de las dos no se cumple se revierte la función, y se devuelve la cadena de error correspondiente. Si llamamos a esta función desde cualquier otra dirección que no sea la del cliente, se revertirá la función y nos devolverá la cadena de error "`Cliente`", de igual forma con la otra afirmación y su correspondiente cadena de error.

La afirmación es equivalente a realizar un condicional y un `raise`, método que revierte la llamada. El ejemplo anterior modificado sería:

```
@external
def findelcontrato():
    if  msg.sender != self.cliente:
        raise "Cliente"
    if not self.terminada:
        raise "Terminada"
    ...
```

El error que se devuelve está creado por una excepción de reversión, es decir, se crea una excepción la cual llama al código `REVERT`. Este hace que se revierta la operación y la frase que acompaña a la afirmación se convierta en el mensaje de error. Cuando esto sucede, el gas restante se devuelve al emisor de la llamada. Podemos hacer que no se realice un reembolso de gas. En lugar de escribir el mensaje de error que queremos que muestre, escribimos `UNREACHABLE`, esto hace que en vez de llamarse al código `REVERT` se llame al código `INVALID`, el cual revierte la acción pero no devuelve el gas restante. Usando el ejemplo anterior tendríamos:

```
@external
```

```
def findelcontrato():
    assert msg.sender == self.cliente,UNREACHABLE
    assert self.terminada,UNREACHABLE
    ...
```

3.6. Alcance y declaraciones

La declaración de variables es lo primero que se ha de hacer en el contrato. Pero no todas las variables tienen que ser globales, podemos tener variables locales, las que están definidas dentro de la función, y las variables de los argumentos de las funciones. En estos tres tipos de variables hemos de indicar su tipo en el momento de declararlas, sin embargo dependiendo de qué tipo sean debemos de indicar un valor inicial o no, así como indicar si es pública o privada.

Al declarar una variable global, o de almacenamiento, debemos indicar si queremos que esta sea pública. Para ello usamos `public()`; si no queremos que lo sea no ponemos nada y el valor de esta solo lo podrá saber el contrato. Al declarar una variable pública el compilador creará una función que se llamará igual que la variable y devolverá un valor del tipo de la variable. De esta forma podemos saber el valor de estas variables llamándolas como si fuesen una función más. En este ejemplo se creará una función llamada `adoptado`, que devolverá un booleano.

```
adoptado: public(bool)
```

Los otros dos tipos de variables solo pueden ser privadas, por lo que no pueden usar esta asignación. Sin embargo, a las variables globales no las podemos asignar un valor inicial, mientras que a las variables locales debemos asignarlas un valor inicial a la hora de declararlas.

Las variables de los argumentos pueden tener asignado un valor, que será el valor por defecto. Es decir, si llamamos a la función y no escribimos el valor del argumento, se tomará el valor asignado por defecto.

Podemos asignar tuplas aunque no exista este tipo. Se podrán asignar cuando una función devuelva una tupla y se tome el valor de este. Un ejemplo de esto es crear una función interna que devuelva una tupla con dos elementos. En la función externa que va a llamar a esta interna debemos declarar las dos variables que van a tomar los valores de la tupla. Cada variable tiene que ser del mismo tipo que el valor de la tupla que va a almacenar. Creamos una tupla introduciendo estas variables entre paréntesis. El orden de estas dos variables

debe ser el mismo orden de los elementos de la tupla que se devuelven. Es decir, si la función interna devuelve `(bool,uint256)`, debemos de colocar la variable booleana primero y la numérica después.

```
@view
@internal
def _ganar(apos: Juego)-> (bool,uint256):
    booleano : bool = ((apos.equipo1 == self.pequipo1) and
              (apos.equipo2 == self.pequipo2))
    return (booleano,apos.apuesta + (apos.apuesta/2))

@view
@external
def ganado(apos: Juego)-> bool:
    assert self.apuntados,"Apuntados"
    gana: bool = False
    cantidad: uint256 = 0
    (gana, cantidad) = self._ganar(apos)
    return gana
```

La función interna `_ganar` devuelve una tupla y la función externa `ganado`, cuando llama a la interna, almacena el valor en la tupla `(gana,cantidad)`. Después esta función solo devuelve un elemento de la tupla, que es `gana`. En el contrato completo otras funciones llaman a esta función interna y se usan los dos o uno de los dos valores. Así, sólo se ha creado una función interna que se puede usar para varias funciones.

El alcance de una variable es la parte del programa en el cual esta variable está definida. Las variables globales están definidas en todo el contrato mientras que las locales y las de argumento sólo en la función donde son declaradas. Como hemos visto en la sección 3.4.1, para acceder a las variables globales usamos el objeto `self`, pero aunque se acceda a ella con este objeto, no podemos crear una variable local o de argumento que tenga el mismo nombre que una función o una variable global. Veamos el siguiente ejemplo, en el que vamos a declarar una variable local fuera de la declaración `if-else`. De esta forma, la variable tendrá el valor asignado dentro de la declaración fuera de esta.

```
@payable
@external
def darse_de_alta():
   assert not self.registrado[msg.sender].reg,"No registrado"
   assert msg.value >= self.cuota,"Valor suficiente"
   index: uint256 = self.indi
```

```
    self.registrado[msg.sender] .reg= True
    if self.indilibres > 0:
        self.registrado[msg.sender].num = self.libres[self.indi]
        self.indilibres -= 1
    else:
        self.registrado[msg.sender].num = self.indice
        index = self.indice
        self.indice += 1
    log Pagado(msg.sender,self.cuota)
    self.clientes[index] = Subcriptor({...})
```

Hemos definido la variable local `index` fuera del `if-else` y se modifica dentro de este. Si la hubiésemos declarado en el `if-else`, se produciría un error a la hora de compilar, ya que esta variable se utiliza fuera de esta declaración.

Pero podemos repetir el nombre de una variable local dentro de la misma función. Realizando modificaciones en el ejemplo anterior:

```
@payable
@external
def darse_de_alta():
    assert not self.registrado[msg.sender].reg,"No registrado"
    assert msg.value >= self.cuota,"Valor suficiente"
    self.registrado[msg.sender] .reg= True
    if self.indilibres > 0:
        ...
        index: uint256 = self.indi
        log Pagado(msg.sender,self.cuota)
        self.clientes[index] = Subcriptor({...})
    else:
        ...
        index: uint256 = self.indi
        log Pagado(msg.sender,self.cuota)
        self.clientes[index] = Subcriptor({...})
```

Aquí no se produce ningún error puesto que el alcance de cada variable es desde que se declara hasta que se termina el cuerpo del `if` o del `else`.

3.7. Funciones integradas

Las funciones integradas son funciones predefinidas en Vyper. Tenemos a nuestro alcance distintas funciones predefinidas para usar en los contratos. Las podemos usar cuando queramos y no podemos crear una función o variable con el nombre de estas. Hay distintos tipos de funciones como las que realizan operaciones bit a bit, criptográficas, matemáticas, de utilidad, de manipulación de datos y de interacción con la cadena de bloques.

Las más importantes o las que son más útiles para realizar un contrato pueden ser las siguientes, aunque hay muchas más funciones predeterminadas [8] que pueden ser útiles o no, para el contrato que se quiera realizar.

La función `selfdestruct`, que hemos mencionado anteriormente, sirve para destruir el contrato en el cual se encuentra. La forma de esta función es `selfdestruct(to:address)⟶None`. Nótese que toma como argumento la dirección a la que queremos que se envié el Ether que hay en la cuenta del contrato y no devuelve nada.

La función `send`, de la forma `send( a: address , valor: uint256 )⟶None`, sirve para enviar una cantidad de Ether a la dirección que queramos. Por tanto, tenemos que pasar como argumentos una dirección y un valor que ha de ser `uint256`.

La función `empty`, de la forma `empty(tipo)⟶ ValorInicial`, da a la variable a la que se asigna esta función el valor inicial del `tipo` que pasamos como argumento. El tipo de la variable y el argumento han de ser iguales.

La función `as_wei_value` es de la forma `as_wei_value(valor,unidad: String) ⟶uint256`. Con esta función podemos saber la cantidad de wei a la que equivale el valor de entrada, es decir, pasamos un valor que puede ser de cualquier tipo numérico pero positivo y la unidad que es, si son "ether","gwei",etc, y nos devuelve el valor en wei.

La función `convert` es de la forma `convert(a:valor,tiposalida)⟶tiposalida`, donde a es de un tipo distinto de `tiposalida`. Con esta función podemos realizar una conversión de tipos, pasando como argumentos la variable que queremos convertir y el tipo de datos que queremos de salida. En la siguiente tabla vemos algunas de las distintas conversiones que se pueden hacer. Esta función es necesaria porque Vyper no cuenta con conversiones implícitas sino que tenemos que realizarlas explícitamente. Existen muchas conversiones [5], algunas de ellas se muestran en la tabla 3.1.

La función len tiene la forma `len(b:tipo)⟶uint256`. Esta función sirve para calcular la longitud de un `String` o de un `Bytes`.

Entrada	Salida	Valores permitidos
bool	int128	Todos. Devuelve 0
bool	decimal	Todos. Devuelve 0.0
decimal	uint256	a >= 0.0. Devuelve el valor truncado
int128	uint256	a >= 0
uint256	int128	a <= MAX_INT128
bytes32	uint256	Todos.

Cuadro 3.1: Conversiones entre valores

La función `concat` es de la forma `concat(a:tipo,b: tipo,...)⟶tipo`. Esta función concatena `String` o `Bytes`, pero todas las que se introducen tienen que ser del mismo tipo.

La función `blockhash` es de la forma `blockhash(número_bloque:uint256) ⟶bytes32`. Dado un número de bloque devuelve el hash de este. Sólo se permite el acceso a 256 bloques anteriores, si damos un número que está por debajo de ese límite o damos uno que está por encima del actual nos devolverá el valor inicial de `bytes32`.

3.8. Interfaces

Como vimos anteriormente en la sección 3.1, podemos hacer que dos contratos interactúen entre sí, para lo que es necesario usar interfaces. En la interfaz se definen las funciones que se van a usar en el contrato y no están en este. Tenemos distintas formas de definir una interfaz, que veremos a continuación.

Podemos definir una interfaz usando `interface` al principio del contrato. Definimos las funciones que vamos a usar y declaramos una variable para llamar al contrato externo a través de ella, como en el siguiente ejemplo, el cual es una ligera modificación del contrato del repositorio.

```
interface Puntos:
    def acumularpuntos(puntos:uint256,cliente: address):
        nonpayable
    def usarpuntos(cliente:address)->uint256:nonpayable
    def usarlitros(cliente:address,puntos: uint256): nonpayable
    def nuevocliente(cliente:address): nonpayable
    def dejardesercliente(cliente:address): nonpayable
...
@payable
```

```
@external
def echargasolina(...,usar_puntos: bool,direccion: address):
    ...
    litros: uint256 = 0
    if usar_puntos:
        litros = Puntos(direccion).usarpuntos(msg.sender)
        ...
```

Definimos la interfaz `Puntos` al principio y para realizar la llamada al contrato externo en la función `echargasolina` debemos de pasar la dirección del contrato como argumento. En este ejemplo llamamos a la función `usarpuntos()`.

Al definir la interfaz también tenemos que indicar la mutabilidad de la función que se va a usar. En nuestro ejemplo todas son `nonpayable`.

Para no pasar la dirección en todas las funciones que llamen al contrato externo, podemos crear una variable global de tipo el nombre de la interfaz. Así, al inicializar el contrato, asignamos la dirección del contrato y así acceder a la interfaz, esto sería como un "acceso directo" al contrato externo. En este ejemplo hemos definido una variable global `puntos_contract`, la cual se ha inicializado en el constructor pasando como argumento la dirección del contrato externo. Después se usa en la función `echargasolina` como si estuviésemos llamando a una variable global, solo que después tenemos que poner a que función queremos que llame.

```
interface Puntos:
    ...
puntos_contract: Puntos
...
@external
def __init__(...,_puntos:address):
    ...
    self.puntos_contract = Puntos(_puntos)

@payable
@external
def echargasolina(...,usar_puntos: bool):
    ...
    litros: uint256 = 0
    if usar_puntos:
        litros = self.puntos_contract.usarpuntos(msg.sender)
        ...
```

En vez de definirlas en el contrato, podemos importar una interfaz. Es decir, creamos externamente una interfaz y la importamos en el contrato que la vaya a

usar. Para crear una interfaz externamente debemos usar la sintaxis de Vyper, es decir, debemos de definir las funciones de la interfaz como si fuese otro contrato, pero en el cuerpo de las funciones puede haber cualquier cosa, ya que se va a obviar. Usando el ejemplo anterior escribiríamos en otro fichero llamado `puntos`:

```
@external
def acumularpuntos(puntos:uint256,cliente: address):
    pass

@external
def usarpuntos(cliente:address)->uint256:
    return
```

Ahora tenemos dos opciones para importar la interfaz, una con `import...` y otra con `from...import...`.

Si usamos la primera forma, en nuestro contrato escribiríamos `import puntos as Puntos`. Debemos poner el alias, es decir el `as Puntos`, ya que si no lo pusiésemos nos daría un error a la hora de compilar. En este caso llamaríamos al contrato externo como en el primer ejemplo visto en esta sección.

De igual modo, si usásemos la segunda forma, en nuestro contrato escribiríamos `from contract import puntos` o `from contract import puntos as Puntos`. Si no escribimos el alias debemos llamar a la interfaz por el nombre del fichero, es decir, en el primer ejemplo de la sección se llamaría a al contrato externo `puntos(direccion).usarpuntos(msg.sender)`, mientras que si lo escribimos el ejemplo de llamada sería el mismo.

Capítulo 4

Ejemplos

En este capítulo vamos a ver algunos ejemplos de código completo. En ellos vamos a ver todo lo que hemos ido viendo en los anteriores capítulos.

4.1. Hora con descuento

Este contrato es para que una tienda ponga una cantidad de stock a la venta y durante un periodo de tiempo el precio del producto esté rebajado.

Primero declaramos las variables. Estas variables son, en orden, el precio del producto, el precio con descuento, la dirección del vendedor, tiempo a esperar para que empiece la oferta, cuando finaliza la oferta y la cantidad de stock.

```
# @version ^0.2.8

precio: public(uint256)
pdescuento:public(uint256)
vendedor: public(address)
inicio: public(uint256)
fin :public(uint256)
stock:public(uint256)
```

Ahora vamos a crear el constructor y en él vamos a iniciar todas las variables, ya que es la tienda quien elige cada una de ellas.

```
@external
def __init__(_stock: uint256,_precio: uint256 ,
             _inicio: uint256 ,_duracion: uint256 ,
             _pdescuento: uint256):
    assert _stock > 0
    assert _precio > 0
    assert _pdescuento > 0
    self.vendedor = msg.sender
    self.stock = _stock
    self.precio = _precio
    self.pdescuento = _pdescuento
    self.inicio = block.timestamp + _inicio
    self.fin = self.inicio + _duracion
```

Observemos los asertos, el contrato se creará si los valores del `_stock`, del `_precio` y del `_pdescuento` son mayores que 0. También debemos ver que, en vez de tomar como argumento el tiempo final, toma la duración del descuento. Por tanto, el fin del tiempo de descuento será el inicio más la duración. El inicio que se pasa como argumento es el tiempo de espera desde el momento actual hasta que empiece la oferta.

Creemos la función `comprar`, la cual ha de ser externa pagable para comprar el producto.

```
@external
@payable
def comprar():
    assert self.stock > 0
    precio: uint256 = self.precio
    if (block.timestamp <= self.fin and
        block.timestamp >= self.inicio):
        precio = self.pdescuento
    assert msg.value >= precio
    cantidad: uint256 = msg.value / precio
    assert self.stock >= cantidad
    self.stock -= cantidad
    send(self.vendedor,cantidad*precio)
```

Primero se comprueba que hay stock. Si lo hay, se determina el precio actual del producto y se comprueba que el valor enviado es suficiente para pagarlo. También se comprueba que el stock que hay actualmente es mayor o igual que la cantidad que se quiere. Si es el caso, se manda el Ether al vendedor y el stock

disminuye.

Creamos la función `terminar` para finalizar el contrato, esta función solo la puede llamar el vendedor y si queda algo de Ether en la cuenta del contrato, se le manda a la cuenta del vendedor.

```
@external
def terminar():
    assert msg.sender == self.vendedor,"Vendedor"
    selfdestruct(self.vendedor)
```

4.2. Apuestas en un partido

En este otro contrato vamos a ir almacenando las apuestas de los usuarios. Cuando empiece el partido, no se podrá apostar más y la casa de apuestas debe poner la mitad de la cantidad de Ether que haya en la cuenta del contrato. Esto es por si se diera el caso de que todas las apuestas sean ciertas, ya que en este caso se devuelve a los clientes 3/2 de su apuesta.

```
struct Juego:
    apostador: address
    equipo1: uint256
    equipo2: uint256
    apuesta: uint256

casa:public(address)
inicial:public(uint256)
empieza:public(uint256)
termina: public(uint256)
pequipo1: uint256
pequipo2: uint256
indice : uint256
apostadores: HashMap[uint256, Juego]
sigindice : uint256
invertido: bool
apuntados:bool

@payable
@external
def __init__( tiempo_inicio: uint256,duracion: uint256):
```

```
22      assert tiempo_inicio > 0
23      assert duracion > 0
24      self.inicial = msg.value
25      self.casa = msg.sender
26      self.empieza = block.timestamp + tiempo_inicio
27      self.termina = self.empieza + duracion
```

Declaramos las variables iniciales, que son, en orden, la cuenta de la casa de apuestas, la cantidad abonada por la casa al iniciar el contrato, el tiempo para que empiece el partido, el tiempo para que termine el partido, la puntuación del equipo 1, la puntuación del equipo 2, el total de apuestas más una, una tabla Hash con las apuestas, el siguiente índice no visitado para devolver el Ether, si la casa ha metido el Ether correspondiente y si se ha anotado la puntuación de cada equipo. Inicializamos las que necesitamos al principio del contrato.

Veamos como crear la función `apostar` para realizar una apuesta.

```
1  @external
2  @payable
3  def apostar(eq1: uint256,eq2: uint256):
4      assert block.timestamp <= self.empieza,"Antes de empezar"
5      assert msg.sender != self.casa,"Jugador"
6      assert msg.value > 0 ,"Apuesta positiva"
7      nfi: uint256 = self.indice
8      self.apostadores[nfi] = Juego({apostador: msg.sender,
9                                     equipo1: eq1, equipo2:eq2,
10                                    apuesta:msg.value})
11     self.indice = nfi + 1
```

Primero debemos comprobar que no ha empezado el partido, que la función no es llamada por la casa de apuestas y que el valor enviado es mayor que 0. Con esto almacenamos en el primer índice libre la estructura `Juego` con los datos que esta necesita y cambiamos el índice al siguiente.

Ahora crearemos una función llamada `necesario` para que la casa de apuestas sepa el Ether que tiene que introducir al contrato y otra función llamada `mitad` para introducirlo.

```
1  @view
2  @external
3  def necesario()-> uint256:
4      assert msg.sender == self.casa,"Casa"
```

```
5       assert  block.timestamp > self.empieza
6                           ,"Despues de empezar"
7       return (self.balance - self.inicial) / 2
8
9   @payable
10  @external
11  def mitad():
12      assert block.timestamp > self.empieza
13                          ,"Despues de empezar"
14      assert self.casa == msg.sender,"Casa"
15      assert (msg.value + self.inicial  >=
16          ((self.balance - self.inicial -msg.value) / 2))
17                                      ,"Valor suficiente"
18      self.invertido = True
```

Estas dos funciones solo pueden ser llamadas por la dirección de la casa de apuestas y cuando haya empezado el partido ya que en este momento no se pueden realizar más apuestas.

Ahora debemos crear la función `ganadores` para almacenar el resultado final.

```
1   @external
2   def ganadores(_eq1: uint256, _eq2: uint256):
3       assert msg.sender == self.casa,"Casa"
4       assert block.timestamp > self.termina,"Despues de terminar"
5       assert not self.apuntados,"No apuntados"
6       self.apuntados = True
7       self.pequipo1 = _eq1
8       self.pequipo2 = _eq2
```

La función `ganadores` solo puede ser llamada por la casa de apuestas, cuando el partido haya terminado y si no se ha llamado previamente.

Nos queda la parte de devolver, si han ganado, el Ether que les corresponde. Para ello vamos a crear la función `devolver`.

```
1   @external
2   def devolver():
3       assert self.apuntados,"Apuntados"
4       assert self.casa == msg.sender,"Casa"
5       assert self.invertido,"Ha invertido"
6       nive:uint256 = self.sigindice
```

```
7       for i in range (nive,nive+30):
8           if i > self.indice:
9               selfdestruct(self.casa)
10          else:
11              if ((self.apostadores[i].equipo1 == self.pequipo1)
12                and (self.apostadores[i].equipo2 == self.pequipo2
                      )):
13
14                  send(self.apostadores[i].apostador,
15                  self.apostadores[i].apuesta +
16                         (self.apostadores[i].apuesta/2))
17      self.sigindice = nive + 30
```

Debemos comprobar que se hayan guardado las puntuaciones de los equipos mediante la variable `self.apuntados`. También debemos de comprobar que sea la casa de apuestas la que esté llamando a esta función, `self.casa == msg.sender`, y que haya invertido el ether correspondiente mediante la variable global `self.invertido`. Primero, creamos una variable local para almacenar el primer índice no visitado. El bucle `for` se va a dar a lo sumo 30 vueltas. Si el valor de la `i` es la cantidad de apuestas hechas más una, se termina el bucle y se destruye el contrato devolviendo a la casa de apuestas la cantidad de Ether en el contrato. Si no es el caso, se comprueba la apuesta del apostante y, en el caso de haber acertado, se le manda la cantidad correspondiente. Cuando se sale del bucle se actualiza el valor del siguiente índice a visitar.

En el contrato hay unas funciones de menor complejidad que no aportan nada novedoso, ya que sirven para devolver valores como si se ha ganado la apuesta, cuanto se ha ganado o se puede ganar, si se ha terminado el partido y si ha empezado. Se pueden consultar estas funciones en el repositorio .

Capítulo 5

Pruebas con Brownie

Como mencionamos en la sección 2.3, los contratos se han de testear para comprobar que funcionan como deben. Para realizar estas pruebas vamos a usar Brownie. En primer lugar presentamos cómo realizar un test sencillo, por ejemplo para el contrato de `Hora con descuento` el cual hemos visto en la sección 4.1 y de esta forma saber como realizar estos test.

```
1  import pytest
2
3  import brownie
4
5  import time
6
7  STOCK = 50
8  PRECIO = 20
9  INICIO = 3
10 DURACION = 5
11 PRECIODESCUENTO = 10
12
13 @pytest.fixture
14 def descuento1_contract(descuento1, accounts):
15     yield descuento1.deploy(STOCK,PRECIO,INICIO,DURACION,
       PRECIODESCUENTO,{'from': accounts[0]})
```

Primero, debemos importar pytest y Brownie ya que son los que vamos a usar para realizar el test. En nuestro caso, también necesitamos importar la biblioteca `time` ya que este contrato tiene una duración de tiempo. Asignamos valores a

las variables que vayamos a introducir como argumentos del constructor, para tenerlos almacenados y después comprobar que el valor del contrato de estas variables es correcto.

Debemos usar el decorador `@pytest.fixture` en la función que llama al contrato para que se inicie. Este decorador sirve para que se ejecute el contrato antes de cada función de prueba y, así, cada prueba será independiente de la anterior. Pasamos como argumentos el nombre del fichero del contrato, esto le dará acceso al objeto contenedor de contratos, en el cual están compilados los contratos. También pasamos `accounts` que nos da acceso al contenedor de direcciones locales. Tanto el nombre que nos da acceso al contenedor de contratos y el `accounts` que nos da acceso al contenedor de direcciones son accesorios de Brownie. Usamos el método `deploy()` pasando los valores que necesita la función constructora y al final entre corchetes ponemos la cuenta que ha realizado la llamada.

Ahora realicemos una primera prueba para saber si los valores almacenados en el contrato son correctos. El nombre de estos test tiene que ser de la forma test_nombre o nombre_test para que Pytest los pueda recoger.

```
def test_inicial(descuento1_contract,accounts):
    assert descuento1_contract.vendedor() == accounts[0]
    assert descuento1_contract.precio() == PRECIO
    assert descuento1_contract.stock() == STOCK
    assert descuento1_contract.pdescuento() == PRECIODESCUENTO
```

En este `test_inicial` pasamos como argumento el nombre de la función que inicia el contrato, ya que así se nos va a dar acceso a este y también las direcciones. El cuerpo es muy simple ya que compara que sea el mismo valor, si lo es se completa el test y en el caso contrario, en el momento en el que alguno no es igual se produce un `fail` y no sigue comprobando. Solo podemos comprobar las variables que sean públicas, ya que estas, como vimos, tienen una función que devuelve su valor.

Veamos la siguiente función para comprobar que funciona correctamente.

```
def test_compra(descuento1_contract,accounts):
    descuento1_contract.comprar({'from':accounts[1],'value':20}
        )
    assert descuento1_contract.stock() == 49
    time.sleep(5)
    descuento1_contract.comprar({'from':accounts[2],'value':20}
        )
```

```
6       assert descuento1_contract.stock() == 47
7       time.sleep(6)
8       descuento1_contract.comprar({'from':accounts[1],'value':20}
            )
9       assert descuento1_contract.stock() == 46
10      descuento1_contract.terminar({'from':accounts[0]})
```

En este test se realizan 3 compras: dos fuera de la hora de descuento y una en ella. El uso de `time.sleep()` es para detener el test durante un tiempo ya que la oferta empieza en un tiempo determinado. Se van realizando comprobaciones del stock restante y podemos observar que cuando no hay oferta solo podemos comprar un producto, mientras que cuando estamos en el tiempo de descuento podemos comprar dos.

También debemos comprobar las afirmaciones ya que genera una acción `REVERT`. Para comprobarlas vamos a usar `brownie.reverts` ya que si no usásemos esto, como el contrato genera un error, tendríamos un error en el test, cuando realmente está haciendo lo que debe hacer. Podemos pasar un argumento, el cual debe coincidir con la cadena de error.

```
1   def test_failed_transactions(descuento1_contract, accounts):
2
3       with brownie.reverts("Precio adecuado"):
4           descuento1_contract.comprar({'from':accounts[1],'value'
                :10})
5
6       time.sleep(5)
7
8       descuento1_contract.comprar({'from':accounts[3],'value':250
            })
9
10      with brownie.reverts("Hay stock suficiente"):
11          descuento1_contract.comprar({'from':accounts[3],'value'
                :500})
12
13      descuento1_contract.comprar({'from':accounts[3],'value':250
            })
14
15      with brownie.reverts("Hay stock"):
16          descuento1_contract.comprar({'from':accounts[3],'value'
                :100})
17
18      with brownie.reverts("Vendedor"):
19          descuento1_contract.terminar({'from':accounts[1]})
```

Con este test comprobamos que se revierte si no se paga la cantidad suficiente, si no hay stock o si otra cuenta llama a la función que solo puede usar el vendedor.

Con estos test hemos probado que se almacenan correctamente los valores, que los cambios de las variables son correctos y que se revierten las funciones.

También podemos comprobar que un evento se ha realizado correctamente, para ver esto vamos a usar otro test ya que este contrato no cuenta con eventos. Vamos a ver el test de evento de el ejemplo visto en la sección 3.5.2.

```
PRESUPUESTO = 20000
TIEMPO_OBRA = 2
MES = 1
INICIO = 4
ALQUILER = 300
...
def test_events(obra_contract,accounts):
    obra_contract.pagarobra(ALQUILER,{'from':accounts[1],'value
        ':PRESUPUESTO})
    time.sleep(4)
    tx1 = obra_contract.cobraralquiler({'from':accounts[1]})

    assert len(tx1.events)==1
    assert tx1.events[0]['emisor'] == accounts[0]
    assert tx1.events[0]['receptor'] == accounts[1]
    assert tx1.events[0]['valor'] == ALQUILER

    obra_contract.finobra({'from':accounts[0]})
    tx2 = obra_contract.findelcontrato({'from':accounts[1]})
    assert len(tx2.events) == 1
    assert tx2.events[0]['emisor'] == accounts[1]
    assert tx2.events[0]['receptor'] == accounts[0]
    assert tx2.events[0]['valor'] == 19700
```

Este contrato solo tiene un evento pero es llamado en dos funciones distintas. El evento visto en la sección 3.5.2 corresponde a la segunda comprobación. Para comprobar eventos almacenamos el registro del evento. Comprobamos que es solo un evento, `assert len(tx1.events)==1`, ya que la función solo almacena un evento, pero se puede dar el caso de que una función registre más de uno. Comprobamos que las distintas partes del evento, en este caso el `emisor`, el `receptor` y el `valor`, tengan registrado el valor correspondiente. Esto lo hacemos usando el método `events` seguido de `[0]` para indicar que queremos consultar el primer evento registrado, si hubiese más el siguiente evento registrado sería

`[1]`. Después de esto, elegimos la parte que queramos comprobar. En nuestro ejemplo comprobamos que `emisor`, `receptor` y `valor` sean correctos.

Para ejecutar estos test y que realicen las comprobaciones pertinentes en los contratos debemos, primero, inicializar un proyecto en Brownie. Para esto necesitamos crear una carpeta vacía, acceder a ella desde la linea de comandos y escribir `brownie init`, a partir de ahora todo lo que hagamos se realizará en la linea de comandos. Esto creará un nuevo proyecto vacío, con varias carpetas. Almacenamos el contrato en la carpeta *contract* y los test en la carpeta test. Los contratos deben tener la extensión "vy" mientras que los test, al ser programas de Python su extensión debe ser "py".

Podemos compilar el contrato primero o directamente realizar la prueba, ya que si el contrato no ha sido compilado previamente lo compilará.

Para compilarlo escribimos `vyper nombreficherocontrato.vy` estando en la carpeta donde está almacenado. Si el contrato se puede compilar compilará y si no dará un error el cual habrá que corregir.

Para realizar las pruebas usamos `brownie test`, primero compilará el contrato y si no se puede compilar, se producirá un error. Después creará la máquina virtual local y realizará los test. Si todo ha salido bien, todo estará en verde, mientras que, si algo ha salido mal, debemos de comprobar donde está el fallo y solucionarlo. Es muy fácil encontrar el fallo ya que pytest te dice donde se está produciendo el fallo.

Capítulo 6

Gas

En esta sección vamos a hablar sobre el gas, ya que es una parte muy importante. Ejecutar contratos cuesta gas, que es una cantidad de Ether. Por tanto, ejecutar un contrato nos está costando dinero, entonces sería más conveniente realizar los contratos de forma eficiente para que consumiesen el menos gas posible.

Hay distintas formas de evitar que se gaste más gas. Vamos a ver distintas maneras de realizar el mismo código y cómo, con ligeras variaciones, se gasta mas o menos gas. Tenemos que tener en cuenta de que va a haber veces que sea mas fácil de entender que de otras, pero esto nos puede suponer una variación de gas, es decir, se puede dar que la función que escribamos sea menos legible pero gaste menos gas y que si la hacemos más legible, gaste más gas. En este caso debemos de elegir que es los más conveniente.

Los bucles `for i in range(a, a+N)` gastan más o menos gas dependiendo de la cantidad de elementos que se recorran, es decir, si la `N` es 10 gastará menos gas que si es 20. Entonces podemos pensar que poner un valor menor es mejor, pero tampoco es el caso, ya que si se pone un numero menor y el contrato que estamos realizando necesita llamar muchas veces a esta función para que se complete el bucle, se producirá un gasto de gas mayor. Por tanto, debemos tener en cuenta que sería lo mejor para cada contrato.

Por ejemplo, esta función gasta 3678448 Gwei si N = 30, y si N = 10 gasta 1252608 Gwei. El costo en euros de la primera función sería de 6 euros mientras que la segunda tiene un costo de 2 euros. En este caso deberíamos de pensar la cantidad de personas que la podrían usar y así poder intentar gastar menos gas.

```
@external
def devolver(_eq1:int128, _eq2:int128):
    ...
    nive:int128 = self.niv
    for i in range (nive,nive+N):
        ...
    self.niv = nive + N
```

Podemos reducir el costo de una función almacenando el valor de una variable global en una local si esta variable se usa en más de un sitio, así no se realizarían llamadas a las variables globales, lo cual aumenta la cantidad de gas necesaria. Veamos dos funciones, que son la misma, pero que realizan esta acción. Por un lado tenemos la función 1:

```
@external
def parar(calle: uint256, litros: decimal, lleno: bool):
    ...
    surti: Calles = self.surtidores[calle]
    surti.uso = False
    self.gasolinera[surti.combustible].litros -= litros
    if surti.selec == 6:
        precio: int128 = ceil((self.gasolinera[surti.
            combustible].precio_litro )*(surti.tope - litros))
        valor : uint256 = convert(precio, uint256)
        send(surti.cliente,valor)
        precio = ceil(litros*self.gasolinera[surti.combustible]
            .precio_litro)
        valor  = convert(precio, uint256)
        send(self.empresa,valor)

    else:
        send(self.empresa,self.surtidores[calle].pagado)
    self.surtidores[calle] = empty(Calles)
```

Podemos observar que creamos una variable local del tipo `Calles` llamada `surti`, sobre la cual vamos a realizar hasta siete consultas si la selección es la `6`. Después de realizar el `if-else` se inicializa el valor de la variable global, aunque esto también lo podíamos haber hecho después de almacenar su valor en la local.

Por otro lado tenemos la función 2:

```
@external
```

```
def parar(calle: uint256, litros: decimal,lleno: bool):
    assert calle == 0 or calle == 1,"Numero calle correcto"
    assert self.surtidores[calle].uso,"El surtidor se esta
        usando"
    assert (self.surtidores[calle].tope == litros or lleno),"
        Esta lleno o tope"
    self.surtidores[calle].uso = False
    self.gasolinera[self.surtidores[calle].combustible].litros
        -= litros
    if self.surtidores[calle].selec == 6:
        precio: int128 = ceil((self.gasolinera[self.surtidores[
            calle].combustible].precio_litro )*(self.surtidores[
            calle].tope - litros))
        valor : uint256 = convert(precio, uint256)
        send(self.surtidores[calle].cliente,valor)
        precio = ceil(litros*self.gasolinera[self.surtidores[
            calle].combustible].precio_litro)
        valor  = convert(precio, uint256)
        send(self.empresa, valor)
    else:
        send(self.empresa,self.surtidores[calle].pagado)
    self.surtidores[calle] = empty(Calles)
```

Esta otra función no almacena el valor en una variable local, si no que se consulta las siete veces a la global. Esta función tiene más dificultad de leer que la función 1, por tanto puede ser más difícil de entender que la primera. Si comprobamos la cantidad de gas que gasta cada una tenemos que la función 1 gasta 243570 Gwei mientras que la función 2 gasta 275682 Gwei. El costo, equivalente en euros, de la función 1 es de 40 céntimos y de la función 2 es de 45 céntimos. En este caso tenemos que la opción más limpia y clara es la que menos gas gasta. Por tanto, a la hora de realizar nuestros contratos, sería preferible almacenar en una variable local una global si se va a usar varias veces.

Otra cosa que puede influir en la variación de gas es llamar a una función interna o, no llamarla y escribir lo de esta función en la externa directamente. En este caso es más limpio el llamar a una función interna, pero esta limpieza nos va a llevar a un aumento del gas que se necesita usar. Entonces, aquí debemos elegir si un código más limpio y que gaste más gas o un código menos limpio y claro pero que gaste menos gas. Veamos dos funciones y cómo varía el gasto de gas de una a la otra.

Esta será nuestra función 3:

```
@external
```

```
def devolver():
    ...
    if (self.apostadores[i].equipo1 == self.pequipo1)
    and (self.apostadores[i].equipo2 == self.pequipo2):
     send(self.apostadores[i].apostador,
           self.apostadores[i].apuesta +
          (self.apostadores[i].apuesta/2) )

    self.sigindice = nive + 30
```

Y esta otra nuestra función 4 aunque realmente son 2:

```
@view
@internal
def _ganar(apos: Juego)-> (bool,uint256):
    booleano : bool = (apos.equipo1 == self.pequipo1) and (apos
        .equipo2 == self.pequipo2)
    return (booleano,apos.apuesta + (apos.apuesta/2))

@external
def devolver():
    ...
      ganado: bool = False
      cantidad: uint256 = 0
      (ganado, cantidad) = self._ganar(self.apostadores[i])
      if ganado:
          send(self.apostadores[i].apostador, cantidad)
    self.sigindice = nive + 30
```

Podemos ver que la condición en la función 3 es mucho más difícil de entender que la que se usa en la función 4, ya que esta llama a la función interna `_ganar` para saber si ha ganado el i-esimo apostante. En este caso la función 3 gasta menos gas, en concreto gasta 1317293 Gwei mientras que la de la función 4 gasta 1355181 Gwei. El costo, equivalente en euros, de la función 3 es de 2,17 euros y de la función 4 es de 2,23. Sería más conveniente el llamar a una función interna aunque gaste más gas, pero al fin y al cabo se está diciendo que se gasta más dinero ya que el gas hay que pagarlo.

Habiendo visto esto ya depende de cada uno cómo prefiera escribir el contrato si mas limpio y claro o que gaste menos, es decir, tendrá que elegir que conviene más realizar.

En el repositorio hay contratos que están "repetidos", estos son variaciones unos de otros en los que hay cambios que hacen que se gaste más o menos gas.

Dentro de cada fichero están ordenados para saber cual gasta más que otro, los que tienen un 1 gastan menos, mientras que los que tienen un 2 gastan más.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo vamos a desarrollar una conclusión al trabajo realizado, lo que se ha aprendido y como puede influir en nuestras vidas. Vamos a ver también posibles líneas de trabajo futuro que sería interesante realizar basándonos en lo que hemos aprendido.

7.1. Conclusiones

En este trabajo hemos aprendido un lenguaje de programación de contratos inteligentes como es Vyper. Este lenguaje no es del todo nuevo puesto que su sintaxis procede de Python, y este lenguaje se ha estudiado en el grado de matemáticas en la asignatura de informática. Hemos visto cómo son estos contratos inteligentes y su funcionamiento, así como el entorno que los rodea: la plataforma en la que se implementan y se usan, que es Ethereum, y las bases de esta plataforma, que es la cadena de bloques, la cual es segura gracias a que cada bloque esta encriptado y unido al anterior, por lo que es muy difícil modificarlo. Aparte de esto, están distribuidos entre potencialmente millones de usuarios, lo cual hace mucho más difícil variar o eliminar algún bloque.

El concepto de distribuido se ha visto con más detalle en la asignatura de programación paralela del grado, la cual ha resultado muy útil. La estructura de la cadena de bloques también resulta muy similar a la estructura de las listas enlazadas estudiada en la asignatura de estructuras de datos, ya que se enlaza cada elemento de la lista mediante un puntero al elementos siguiente. Pero en este caso las listas se pueden variar sin variar los anteriores elementos.

Hemos aprendido cuál es el trabajo de los mineros, la validación de los bloques y creación de nuevos tokens, y que este trabajo es imprescindible para que Ethereum siga funcionando. Hemos aprendido que no vale lo mismo escribir de una forma o de otra, ya que puede salir más caro. Que hay que tener cuidado con el gas para no sufrir una falta de gas y que no se revierta la llamada de forma inesperada. También hemos aprendido a realizar test de unidad al código creado usando pytest y Brownie y así asegurarnos de que estos contratos estén bien hechos y sean funcionales. Por tanto, distintas nociones aprendidas durante el grado son una base y un comienzo de este trabajo, ya que sin estas el trabajo a realizar hubiese sido mayor. En consecuencia, este trabajo puede ser una extensión de los conocimientos adquiridos durante el grado.

Como hemos visto durante toda la memoria, Vyper ofrece una seguridad para realizar estos contratos de forma casi intuitiva, ya que, por como está diseñado el lenguaje, es muy difícil realizar un contrato en el que no te des cuenta que algo puede fallar, es imposible realizar bucles infinitos por lo que es imposible que se quede una función sin gas si se le ha mandado el necesario. Y si realizamos las pruebas pertinentes si que es imposible que hagamos un contrato que no funcione adecuadamente.

Hemos aprendido cómo realizar contratos inteligentes con este lenguaje, desde cosas muy simples como puede ser un contrato para enviar o recibir Ether a realizar un contrato de suscripción de clientes en el que se envía un determinado producto, o un contrato para echar gasolina y que te den puntos los cuales puedes canjear por litros gratis, pasando por realizar un contrato de adopción de un animal de una protectora de animales, o compras de billetes de avión, que si este no sale a tiempo, el contrato devuelve a los clientes una parte de dinero por el retraso.

Como podemos observar, estos contratos pueden ser útiles en nuestro día a día y no sabíamos que había una forma más fácil de realizar estas acciones, ya que en estos casos no se necesita ningún intermediario que tenga que verificar cada condición, sino que el contrato lo hace solo. Gracias a estos contratos y a las cadenas de bloques podemos hacer que la vida sea más fácil, haciendo que las transacciones cotidianas se realicen más rápidamente, de forma más eficiente y tengamos registros de estas transacciones.

De esta forma, los contratos inteligentes, en un futuro no muy lejano, serán nuestro día a día. Porque gracias a ellos no necesitaríamos un intermediario que verifique que se cumplan los acuerdos o que realice las transacciones. El contrato inteligente lo hace directamente, de forma rápida y segura gracias a la cadena de bloques y a la simpleza de estos contratos, porque, como hemos visto en los ejemplos, estos contratos son fáciles de leer y entender. Con estos contratos se pueden hacer infinidad de cosas que agilizarían nuestro día a día. Desde realizar la votación del presidente a través de una DApp, que sería más rápido y sin el coste que tiene actualmente, hasta pagos sin necesidad de que el

banco los verifique o la contratación de un seguro en el que si se da parte los cobros sea inmediatos. Nuestra vida sería algo más fácil, no dependeríamos de tantos intermediarios y nos ahorraríamos dinero.

7.2. Trabajo futuro

Este tipo de contratos están en evolución constante, tanto por la creación de nuevos y distintos contratos, como por que cada vez se usan más. Una muy buena continuación de los contratos inteligentes, sería el desarrollo de una DApp que use uno o varios de los contratos que se han realizado y se encuentran en el repositorio. De esta forma estos contratos podrían usarse por muchas personas y tendrían un uso real.

Se podrían realizar los contratos del repositorio en Solidity y realizar comprobaciones de los gastos entre un lenguaje y otro. Estos contratos se realizarían usando las características de Solidity y veríamos si las diferencias entre un lenguaje y otro, así como sus posibles defectos, son muy grandes y por tanto con un lenguaje puedes crear contratos que con el otro no o, sin embargo, puedes realizar exactamente los mismos contratos. Esta comprobación entre lenguajes puede ser útil, ya que actualmente Solidity es el lenguaje más usado por su flexibilidad y puede que Vyper, que es más seguro y fácil, pueda realizar los mismos contratos. Aunque uno sea Turing completo y otro no, los contratos no deben tener bucles infinitos ni precisar recursos ilimitados ya que la cadena de bloques tiene gas limitado para cada bloque.

Otro posible trabajo sería la implementación de una cadena de bloques. De esta forma nos podríamos adentrar más a fondo en estas estructuras ya que son las responsables de la seguridad de los datos, así como la base de estas plataformas descentralizadas. También podríamos realizar la validación de bloques y así entender mejor como funciona esta e incluso realizar los programas necesarios para minar en Ethereum.

También podríamos realizar un token propio que solo tuviese sentido dentro de una DApp y un contrato determinados, y que este se pudiese usar sin necesidad de usar directamente Ether. Por ejemplo, podríamos crear un aplicación que usase el contrato del bingo creado y que cada el token fuese el cartón o usar el contrato de las apuestas y que el token fuese la apuesta realizada o crear una DApp que use estos dos contratos y al inicio se cambie el Ether por tokens que representan una determinada cantidad, como las fichas de un casino, y usarlas en el contrato que quisiésemos, bien para comprar un cartón o bien para pagar una apuesta.

Capítulo 8

Summary

Smart contracts are computer programs that execute an agreement between two or more persons. These smart contracts are housed in Ethereum. Ethereum is a digital platform decentralized, there is no person or entity that control it. This platform use blockchain, that it is made up of a large number of block where each block are connected with its previous block. The best known use for this is the cryptocurrency which is a digital money that use cryptography to secure transactions. The cryptocurrency of Ethereum is Ether. Each person's Ether are associated with accounts that use a private password, and the smart contract's Ether are associated with a contract account.

Each block have an amount of transactions that they can be a send of ether, a change in value of a global variable in a smart contract,... It are very secure because you need to change all the blocks before the block that you want to change, and it is impossible. The blockchain is in all the computers that use Ethereum. These computers are the nodes of the Ethereum network. Some of this nodes are mining nodes and this nodes do Ethereum mining. The miners validate the blocks creating alphanumeric chains called Hash, to protect all the transactions of the block. This done using Hash function, which cryptography the transactions to create the Hash. This validation has a computational cost which is measured by gas, and its unit of measurement is the Ether. Smart contract run in the Ethereum Virtual Machine(EMV), which is the Ethereum's set of nodes.

Vyper is a language to program smart contract. It do in safe way because it very easy to program and to understand these smart contracts. First of all, you need to declare the global variables saying the type that each one is and saying if the variable are public or no. You can define a custom data type using `struct` and you can interact with other contracts using `interface`, that it is the set of functions of the other contract that you are going to use in this one. You can

also define `events` to register some information for an account, and you use `log NameEvent(..)` in a function to register it.

After this, you need to define the constructor of the smart contract. This function are called `__init__`. This function cannot call other functions and it cannot return anything. But you can put value in some or all of the global variables.

The functions that you are going to program can use asserts to check some conditions. If one of these conditions are false, the call will be reverted. Before defining a function you need to write the the visibility of the function, it can be external, you can call this function externally, or internal, only the other functions of the contract can call it. You use `self` to call internal functions or to change or consult the value of the global variables. You might indicate the mutability of the function. You have four different types of mutability, with the pure you cannot read from the contract state and you cannot change the value of global variables, with the view you cannot change it either, but you can consulting the state of the contract, With the payable you can send Ether to the contract accounts. By default the functions are no payable.

You can do loop only using `for i iterable`, where the range of iterable must be finite. It can be a finite list or the function `in range` to made a finite interval. The itereable cannot depend of global variables. With the function `in range` only can say the end, the start and the end or an interval of type (a,a+N) where a is a variable of type numeric and N is a literal number. With the use of only this loop and with the restriction of the calls between functions you cannot do any recursive function or infinite loop. This is thanks to the fact that Vyper can calculate the upper limit of the gas to be used. In functions you can use the `if-else` statement to create different contract paths. The condition of the `if` statement must be a boolean. The statement to control the flow of functions are `break`, `return`, `pass`, `continue`.

You have different environment variables to provide information about the blockchain or current transactions like msg.sender or `block.timestamp` and environment constants that are a convenience constants or a custom constants. You also have built in functions like `convert` or `send` that you can use in the smart contract.

After create a smart contract you need to test it before implement it in Ethereum because if you implement it you will not be able to change it. You test the smart contract with Pytest and Brownie, and if the smart contract pass the test you can implement it in Ethereum.

Apéndice A

Instalación

Es mejor instalar estos programas en un entorno virtual para que no haya conflictos y se pueda trabajar sin preocupación de qué pueda ocurrir. Este entorno lo podemos crear eligiendo una carpeta desde la linea de comandos y después poniendo "`python3 -m venv tutorial-env`". Se descargarán las carpetas necesarias en la carpeta previamente seleccionada. Para activarlo debemos poner a continuación "`tutorial-env.bat`" si estamos trabajando en Windows, y si estamos trabajando en Unix o MacOS debemos escribir "`source tutorial-env/bin/activate`".

Los test se realizarán usando Pytest y Brownie. Debemos usar Python 3, puesto que es donde se puede usar Brownie, y no versiones anteriores. Ganache-cli también es una dependencia de Brownie, puesto que el contrato se ejecutaría en Ethereum y queremos probar que funcionan de manera local. Para poder instalar Ganache-cli debemos tener instalado Node.js. Por tanto, el orden de instalación más apropiado sería el que vamos a seguir a continuación. Mencionar que todos excepto Node los instalaremos desde la línea de comandos usando el comando `pip instal`.

1. Para instalar Vyper escribimos "`pip install vyper`".
2. Para instalar Pytest escribimos "`pip install -U pytest`".
3. Para instalar Brownie escribimos "`pip install eth-brownie`".
4. Node lo tenemos que descargar e instalar.
5. Para instalar Ganache-cli escribimos "`npm install -g ganache-cli`".

Bibliografía

[1] Alexander Preukschat (Coordinador). *Blockchain: la revolución industrial de internet*. Planeta, 2019. ISBN: 8498754895.

[2] Andreas Antonopoulos y Gavin Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly Media, 2018. ISBN: 1491971940.

[3] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 2017. ISBN: 1593278268.

[4] *Bitcoin*. URL: https://bitcoin.org/es/faq#que-es-bitcoin.

[5] *Conversiones posibles en Vyper*. 2020. URL: https://vyper.readthedocs.io/en/stable/types.html#type-conversions.

[6] *Documentación de Brownie*. 2020. URL: https://eth-brownie.readthedocs.io/en/stable/.

[7] *Documentación Vyper*. 2020. URL: https://vyper.readthedocs.io/en/stable/.

[8] *Funciones integradas en Vyper*. 2020. URL: https://vyper.readthedocs.io/en/stable/built-in-functions.html#chain-interaction.

[9] Liqun Huang Gavin Zheng Longxiang Gao y Jian Guan. *Ethereum Smart Contract Development in Solidity*. Springer, 2020. ISBN: 9811562172.

[10] Paul Hamill. *Unit Test Frameworks*. O'Reilly Media, 2004. ISBN: 0596006896.

[11] *Introducción a los Contratos Inteligentes*. 2017. URL: https://solidity-es.readthedocs.io/es/latest/introduction-to-smart-contracts.html.

[12] Herbert Jones. *Criptomonedas: Una guía esencial para principiantes sobre la Tecnología de Cadenas de Bloques, la Inversión en Criptomonedas, y Bitcoin, incluyendo Minería, Ethereum y Comercio*. Independently published, 2019. ISBN: 1097822451.

[13] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. Auerbach Publications, 2013. ISBN: 1466560681.

[14] Andreas M.Antonopoulos. *Mastering Bitcoin*. O'Reilly Media, 2017. ISBN: 1491954388.

[15] Eugenio Noyola. *Ethereum, tokens smart contracts: Notes on getting started.* Independently published, 2017. ISBN: 1973442558.

[16] Javier Pastor. *Qué es blockchain: la explicación definitiva para la tecnología más de moda*. 2017. URL: https://www.xataka.com/especiales/que-es-blockchain-la-explicacion-definitiva-para-la-tecnologia-mas-de-moda.

[17] Farhana Sheikh y Leonel Sousa. *Circuits and Systems for Security and Privacy: 57 (Devices, Circuits, and Systems)*. CRC Press, 2016. ISBN: 1482236885.

[18] *Web de Ethereum*. 2021. URL: https://ethereum.org/es/.