# Solidity Documentation

***Versión 0.8.20***

**Ethereum**

**05 de marzo de 2023**

# Conceptos Básicos

Solidity es un lenguaje de alto nivel orientado a objetos, para implementar contratos inteligentes (smart contracts). Los contratos inteligentes son programas que rigen el comportamiento de las cuentas dentro del ecosistema de Ethereum.

Solidity es un lenguaje de llaves, diseñado para la Máquina Virtual de Ethereum (Ethereum Virtual Machine). Está influenciado por los lenguajes de programación C++, Python y JavaScript. Para más detalles sobre los lenguajes que han inspirado Solidity visite la sección *influencia de lenguajes*.

Solidity es estáticamente escrito, soporta herencia, librerías, y tipos complejos definidos por el usuario, entre otras características.

Con Solidity se pueden crear contratos para votaciones, financiación colectiva (crowdfunding), subastas a ciegas, y monederos (wallets) multifirma.

Al momento de desplegar un contrato, debes utilizar la versión más reciente de Solidity. Aparte de casos excepcionales, solamente la última versión recibe correciones de seguridad. Además, cambios importantes y nuevas funciones se introducen periódicamente. Actualmente se utiliza el número de versión 0.y.z para indicar estos rápidos cambios.

**Advertencia:** Solidity lanzó recientemente la versión 0.8.x, que introdujo varios cambios significantes. Asegúrese de leer *la lista completa de cambios*.

Ideas para mejorar Solidity o esta misma documentación son siempre bienvenidas, lea nuestra *guía de contribución* para más detalles.

**Nota:** Puede descargar esta documentación como PDF, HTML o Epub haciendo click en el menú desplegable que se encuentra en la esquina inferior izquierda y seleccionando su formato preferido de descarga.

# CAPÍTULO 1

# Para Comenzar

**1. Comprender los Conceptos Básicos de los Contratos Inteligentes**

Si usted es nuevo y todavía no está familiarizado con el concepto de los contratos inteligentes, le recomendamos iniciar con la sección «Introducción a los Contratos Inteligentes», que cubre:

- *Un ejemplo sencillo de un contrato inteligente* escrito en Solidity.
- *Conceptos Básicos de las Cadenas de Bloques (Blockchain)*.
- *La Máquina Virtual de Ethereum*.

**2. Conozca Solidity**

Una vez que esté relacionado con los conceptos básicos, le recomendamos leer las secciones de *«Solidity con Ejemplos»* y «Descripción del Lenguaje» para comprender los conceptos fundamentales del lenguaje.

**3. Instalar el Compilador de Solidity**

Hay distintas maneras de instalar el compilador de Solidity, simplemente elija su opción preferida y siga los pasos indicados en la *página de instalación*.

---

**Nota:** Puede probar algunos ejemplos de código directamente en su navegador con Remix IDE. Remix es un entorno de desarrollo integrado (IDE) basado en el navegador web, que permite a cualquier usuario escribir, desplegar y administrar contratos inteligentes de Solidity; sin la necesidad de instalar Solidity localmente.

---

**Advertencia:** Ya que el software está escrito por humanos, puede contener errores. Usted debe seguir e implementar las mejores prácticas de desarrollo de software establecidas al escribir sus contratos inteligentes. Esto incluye la revisión, pruebas, auditorías y la correcta validez del código. Algunas veces los usuarios de los contratos inteligentes tienen más confianza el el código que sus mismos autores; ya que las cadenas de bloques y los contratos inteligentes tienes sus respectivos problemas, se recomienda que lea la sección *Consideraciones de Seguridad* antes de trabajar en el código de producción.

**4. Conocer Más**

Si desea obtener más información sobre la creación de aplicaciones descentralizadas en Ethereum, los recursos para desarrolladores de Ethereum pueden ayudarlo con más documentación general sobre Ethereum y una amplia selección de tutoriales, herramientas y marcos de desarrollo (frameworks).

Si tiene alguna duda o pregunta, puede intentar buscar consultas o respuestas en Ethereum StackExchange, o en nuestro Canal de Gitter.

CAPÍTULO 2

# Traducciones

La comunidad de contribuidores es la encargada de la traducción de esta documentación en diferentes lenguajes. Por favor, tenga en cuenta que se tienen diversos grados de integridad y actualización. La versión en Inglés es tomada como referencia.

Puede elegir entre distintos idiomas haciendo click en el menú desplegable que se encuentra en la esquina inferior izquierda y seleccionando su lenguaje preferido.

- Chino
- Francés
- Indonesio
- Japonés
- Coreano
- Persa
- Ruso
- Español
- Turco
- Alemán
- Portugués

**Nota:** Recientemente se creó una nueva organización en GitHub y un flujo de trabajo para las traducciones, con el motivo de agilizar los esfuerzos de la comunidad. Por favor, consulte la guía de traducción para obtener información sobre cómo iniciar un nuevo idioma o contribuir en las traducciones de la comunidad. Hemos establecido una organización de Github y flujo de trabajo de traducción para ayudar a optimizar los esfuerzos de la comunidad. Consulte la guīa de traducción en solidity-docs org para obtener información sobre cómo iniciar una nuevo idioma o contribuir a la traducciones de la comunidad.

# CAPÍTULO 3

# Contenidos

Índice de palabras clave, Página de búsqueda

## 3.1 Introducción a los Smart Contracts

### 3.1.1 Un Smart Contract Simple

Iniciemos con un ejemplo básico, que establece el valor de una variable y la expone para que otros contratos puedan acceder a ella. Está bien si no comprende todo en este momento, entraremos en más detalles más adelante.

#### Ejemplo de Almacenamiento

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

La primera línea nos indica que el código fuente está bajo la licencia GPL versión 3.0. Los especificadores de licencia, que son legibles por las computadoras, son importantes en una configuración donde la publicación del código fuente es predeterminada.

La siguiente línea especifica que el código fuente está escrito para la versión 0.4.16 de Solidity, o una versión más reciente del lenguaje, hasta, pero sin incluir, la versión 0.9.0. Esto es para garantizar que el contrato no pueda ser compilado con una versión nueva del compilador, donde el código podría comportarse de manera diferente. Los *Pragmas* son instrucciones comunes para los compiladores, indican cómo se debe tratar el código fuente (por ejemplo, pragma once).

Un contrato, en el sentido de Solidity, es una colección de código (sus *funciones*) y datos (su *estado*) que reside en una dirección específica en el blockchain de Ethereum. La línea `uint storedData;` declara una variable de estado denominada `storedData`, de tipo `uint` (*u*nsigned *int*eger de *256* bits). Podemos considerarlo como un espacio único en una base de datos que se puede consultar y modificar llamando a funciones del código que administran la base de datos.

---

You can think of it as a single slot in a database that you can query and alter by calling functions of the code that manages the database. In this example, the contract defines the functions `set` and `get` that can be used to modify or retrieve the value of the variable.

To access a member (like a state variable) of the current contract, you do not typically add the `this.` prefix, you just access it directly via its name. Unlike in some other languages, omitting it is not just a matter of style, it results in a completely different way to access the member, but more on this later.

This contract does not do much yet apart from (due to the infrastructure built by Ethereum) allowing anyone to store a single number that is accessible by anyone in the world without a (feasible) way to prevent you from publishing this number. Anyone could call `set` again with a different value and overwrite your number, but the number is still stored in the history of the blockchain. Later, you will see how you can impose access restrictions so that only you can alter the number.

**Advertencia:** Be careful with using Unicode text, as similar looking (or even identical) characters can have different code points and as such are encoded as a different byte array.

---

**Nota:** All identifiers (contract names, function names and variable names) are restricted to the ASCII character set. It is possible to store UTF-8 encoded data in string variables.

---

### Subcurrency Example

The following contract implements the simplest form of a cryptocurrency. The contract allows only its creator to create new coins (different issuance schemes are possible). Anyone can send coins to each other without a need for registering with a username and password, all you need is an Ethereum keypair.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping(address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

This contract introduces some new concepts, let us go through them one by one.

The line `address public minter;` declares a state variable of type *address*. The `address` type is a 160-bit value that does not allow any arithmetic operations. It is suitable for storing addresses of contracts, or a hash of the public half of a keypair belonging to *external accounts*.

The keyword `public` automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable. The code of the function generated by the compiler is equivalent to the following (ignore `external` and `view` for now):

```
function minter() external view returns (address) { return minter; }
```

You could add a function like the above yourself, but you would have a function and state variable with the same name. You do not need to do this, the compiler figures it out for you.

The next line, `mapping(address => uint) public balances;` also creates a public state variable, but it is a more complex datatype. The *mapping* type maps addresses to *unsigned integers*.

Mappings can be seen as hash tables which are virtually initialised such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros. However, it is neither possible to obtain a list of all keys of a mapping, nor a list of all values. Record what you added to the mapping, or use it in a context where this is

not needed. Or even better, keep a list, or use a more suitable data type.

The *getter function* created by the `public` keyword is more complex in the case of a mapping. It looks like the following:

```
function balances(address account) external view returns (uint) {
    return balances[account];
}
```

You can use this function to query the balance of a single account.

The line `event Sent(address from, address to, uint amount);` declares an *«event»*, which is emitted in the last line of the function `send`. Ethereum clients such as web applications can listen for these events emitted on the blockchain without much cost. As soon as it is emitted, the listener receives the arguments `from`, `to` and `amount`, which makes it possible to track transactions.

To listen for this event, you could use the following JavaScript code, which uses web3.js to create the `Coin` contract object, and any user interface calls the automatically generated `balances` function from above:

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

The *constructor* is a special function that is executed during the creation of the contract and cannot be called afterwards. In this case, it permanently stores the address of the person creating the contract. The `msg` variable (together with `tx` and `block`) is a *special global variable* that contains properties which allow access to the blockchain. `msg.sender` is always the address where the current (external) function call came from.

The functions that make up the contract, and that users and contracts can call are `mint` and `send`.

The `mint` function sends an amount of newly created coins to another address. The *require* function call defines conditions that reverts all changes if not met. In this example, `require(msg.sender == minter);` ensures that only the creator of the contract can call `mint`. In general, the creator can mint as many tokens as they like, but at some point, this will lead to a phenomenon called «overflow». Note that because of the default *Checked arithmetic*, the transaction would revert if the expression `balances[receiver] += amount;` overflows, i.e., when `balances[receiver] + amount` in arbitrary precision arithmetic is larger than the maximum value of `uint` (`2**256 - 1`). This is also true for the statement `balances[receiver] += amount;` in the function `send`.

*Errors* allow you to provide more information to the caller about why a condition or operation failed. Errors are used together with the *revert statement*. The `revert` statement unconditionally aborts and reverts all changes similar to the `require` function, but it also allows you to provide the name of an error and additional data which will be supplied to the caller (and eventually to the front-end application or block explorer) so that a failure can more easily be debugged or reacted upon.

The `send` function can be used by anyone (who already has some of these coins) to send coins to anyone else. If the sender does not have enough coins to send, the `if` condition evaluates to true. As a result, the `revert` will cause the operation to fail while providing the sender with error details using the `InsufficientBalance` error.

**Nota:** If you use this contract to send coins to an address, you will not see anything when you look at that address on a blockchain explorer, because the record that you sent coins and the changed balances are only stored in the data storage

of this particular coin contract. By using events, you can create a «blockchain explorer» that tracks transactions and balances of your new coin, but you have to inspect the coin contract address and not the addresses of the coin owners.

### 3.1.2 Blockchain Basics

Blockchains as a concept are not too hard to understand for programmers. The reason is that most of the complications (mining, hashing, elliptic-curve cryptography, peer-to-peer networks, etc.) are just there to provide a certain set of features and promises for the platform. Once you accept these features as given, you do not have to worry about the underlying technology - or do you have to know how Amazon's AWS works internally in order to use it?

#### Transactions

A blockchain is a globally shared, transactional database. This means that everyone can read entries in the database just by participating in the network. If you want to change something in the database, you have to create a so-called transaction which has to be accepted by all others. The word transaction implies that the change you want to make (assume you want to change two values at the same time) is either not done at all or completely applied. Furthermore, while your transaction is being applied to the database, no other transaction can alter it.

As an example, imagine a table that lists the balances of all accounts in an electronic currency. If a transfer from one account to another is requested, the transactional nature of the database ensures that if the amount is subtracted from one account, it is always added to the other account. If due to whatever reason, adding the amount to the target account is not possible, the source account is also not modified.

Furthermore, a transaction is always cryptographically signed by the sender (creator). This makes it straightforward to guard access to specific modifications of the database. In the example of the electronic currency, a simple check ensures that only the person holding the keys to the account can transfer money from it.

#### Blocks

One major obstacle to overcome is what (in Bitcoin terms) is called a «double-spend attack»: What happens if two transactions exist in the network that both want to empty an account? Only one of the transactions can be valid, typically the one that is accepted first. The problem is that «first» is not an objective term in a peer-to-peer network.

The abstract answer to this is that you do not have to care. A globally accepted order of the transactions will be selected for you, solving the conflict. The transactions will be bundled into what is called a «block» and then they will be executed and distributed among all participating nodes. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

These blocks form a linear sequence in time, and that is where the word «blockchain» derives from. Blocks are added to the chain at regular intervals, although these intervals may be subject to change in the future. For the most up-to-date information, it is recommended to monitor the network, for example, on Etherscan.

As part of the «order selection mechanism» (which is called «mining») it may happen that blocks are reverted from time to time, but only at the «tip» of the chain. The more blocks are added on top of a particular block, the less likely this block will be reverted. So it might be that your transactions are reverted and even removed from the blockchain, but the longer you wait, the less likely it will be.

**Nota:** Transactions are not guaranteed to be included in the next block or any specific future block, since it is not up to the submitter of a transaction, but up to the miners to determine in which block the transaction is included.

If you want to schedule future calls of your contract, you can use a smart contract automation tool or an oracle service.

### 3.1.3 The Ethereum Virtual Machine

#### Overview

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts.

#### Accounts

There are two kinds of accounts in Ethereum which share the same address space: **External accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called «nonce»).

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage**.

Furthermore, every account has a **balance** in Ether (in «Wei» to be exact, `1 ether` is `10**18 wei`) which can be modified by sending transactions that include Ether.

#### Transactions

A transaction is a message that is sent from one account to another account (which might be the same or empty, see below). It can include binary data (which is called «payload») and Ether.

If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is not set (the transaction does not have a recipient or the recipient is set to `null`), the transaction creates a **new contract**. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the «nonce»). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output data of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code when executed.

---

**Nota:** While a contract is being created, its code is still empty. Because of that, you should not call back into the contract under construction until its constructor has finished executing.

---

#### Gas

Upon creation, each transaction is charged with a certain amount of **gas** that has to be paid for by the originator of the transaction (`tx.origin`). While the EVM executes the transaction, the gas is gradually depleted according to specific rules. If the gas is used up at any point (i.e. it would be negative), an out-of-gas exception is triggered, which ends execution and reverts all modifications made to the state in the current call frame.

This mechanism incentivizes economical use of EVM execution time and also compensates EVM executors (i.e. miners / stakers) for their work. Since each block has a maximum amount of gas, it also limits the amount of work needed to validate a block.

The **gas price** is a value set by the originator of the transaction, who has to pay `gas_price * gas` up front to the EVM executor. If some gas is left after execution, it is refunded to the transaction originator. In case of an exception that reverts changes, already used up gas is not refunded.

Since EVM executors can choose to include a transaction or not, transaction senders cannot abuse the system by setting a low gas price.

### Storage, Memory and the Stack

The Ethereum Virtual Machine has three areas where it can store data: storage, memory and the stack.

Each account has a data area called **storage**, which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract, it is comparatively costly to read, and even more to initialise and modify storage. Because of this cost, you should minimize what you store in persistent storage to what the contract needs to run. Store data like derived calculations, caching, and aggregates outside of the contract. A contract can neither read nor write to any storage apart from its own.

The second data area is called **memory**, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (i.e. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

The EVM is not a register machine but a stack machine, so all computations are performed on a data area called the **stack**. It has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end in the following way: It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack. Of course it is possible to move stack elements to storage or memory in order to get deeper access to the stack, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

### Instruction Set

The instruction set of the EVM is kept minimal in order to avoid incorrect or inconsistent implementations which could cause consensus problems. All instructions operate on the basic data type, 256-bit words or on slices of memory (or other byte arrays). The usual arithmetic, bit, logical and comparison operations are present. Conditional and unconditional jumps are possible. Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

For a complete list, please see the *list of opcodes* as part of the inline assembly documentation.

### Message Calls

Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn can create further message calls.

A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain. If an out-of-gas exception happens in the inner call (or any other exception), this will be signaled by an error value put onto the stack. In this case, only the gas sent together with the call is used up. In Solidity, the calling contract causes a manual exception by default in such situations, so that exceptions «bubble up» the call stack.

As already said, the called contract (which can be the same as the caller) will receive a freshly cleared instance of memory and has access to the call payload - which will be provided in a separate area called the **calldata**. After it has

finished execution, it can return data which will be stored at a location in the caller's memory preallocated by the caller. All such calls are fully synchronous.

Calls are **limited** to a depth of 1024, which means that for more complex operations, loops should be preferred over recursive calls. Furthermore, only 63/64th of the gas can be forwarded in a message call, which causes a depth limit of a little less than 1000 in practice.

### Delegatecall and Libraries

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context (i.e. at the address) of the calling contract and `msg.sender` and `msg.value` do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

This makes it possible to implement the «library» feature in Solidity: Reusable library code that can be applied to a contract's storage, e.g. in order to implement a complex data structure.

### Logs

It is possible to store data in a specially indexed data structure that maps all the way up to the block level. This feature called **logs** is used by Solidity in order to implement *events*. Contracts cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain. Since some part of the log data is stored in bloom filters, it is possible to search for this data in an efficient and cryptographically secure way, so network peers that do not download the whole blockchain (so-called «light clients») can still find these logs.

### Create

Contracts can even create other contracts using a special opcode (i.e. they do not simply call the zero address as a transaction would). The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller / creator receives the address of the new contract on the stack.

### Deactivate and Self-destruct

The only way to remove code from the blockchain is when a contract at that address performs the `selfdestruct` operation. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state. Removing the contract in theory sounds like a good idea, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether is forever lost.

**Advertencia:** From version 0.8.18 and up, the use of `selfdestruct` in both Solidity and Yul will trigger a deprecation warning, since the `SELFDESTRUCT` opcode will eventually undergo breaking changes in behaviour as stated in EIP-6049.

**Advertencia:** Even if a contract is removed by `selfdestruct`, it is still part of the history of the blockchain and probably retained by most Ethereum nodes. So using `selfdestruct` is not the same as deleting data from a hard disk.

**Nota:** Even if a contract's code does not contain a call to `selfdestruct`, it can still perform that operation using `delegatecall` or `callcode`.

If you want to deactivate your contracts, you should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

#### Precompiled Contracts

There is a small set of contract addresses that are special: The address range between `1` and (including) `8` contains «precompiled contracts» that can be called as any other contract but their behaviour (and their gas consumption) is not defined by EVM code stored at that address (they do not contain code) but instead is implemented in the EVM execution environment itself.

Different EVM-compatible chains might use a different set of precompiled contracts. It might also be possible that new precompiled contracts are added to the Ethereum main chain in the future, but you can reasonably expect them to always be in the range between `1` and `0xffff` (inclusive).

## 3.2 Instalando el compilador de Solidity

### 3.2.1 Versionado

Solidity utiliza Versionado Semántico. Además, los parches publicados con versión 0 (ej. 0.x.y) no contendrán cambios con rupturas. Eso significa que si un código compila con una versión 0.x.y puede esperar que compile con una versión 0.x.z donde z > y.

Adicionalmente a los lanzamientos, proporcionamos **builds nocturnos** con la intención de facilitar a los desarrolladores probar próximas funcionalidades y proveer retroalimentación temprana. Note, sin embargo, que aunque los builds nocturnos son bastante estables, contienen código reciente de la rama de desarrollo y no podemos garantizar que siempre funcionen. A pesar de nuestros mejores esfuerzos, pueden contener cambios fallidos o sin documentar que no formarán parte del lanzamiento final. No están pensados para uso en producción.

Al desplegar contratos, debería usar el último lanzamiento de Solidity. Esto es porque regularmente se introducen cambios con rupturas, funcionalidades nuevas y correcciones de errores. Actualmente usamos un número de versión 0.x para indicar el rápido ritmo de cambio.

### 3.2.2 Remix

*Recomendamos Remix para pequeños contratos y para aprender Solidity rápidamente.*

Entre a Remix en línea, no necesita instalar nada. Si quiere usarlo sin conexión a internet, vaya a https://github.com/ethereum/remix-live/tree/gh-pages y descargue el archivo `.zip` como se explica en la página. Remix también es una buena opción para probar builds nocturnos sin instalar múltiples versiones de Solidity.

Las siguientes opciones en esta página detallan cómo instalar el software para compilar Solidity mediante línea de comandos en su computadora. Use un compilador de línea de comandos si está trabajando en un contrato más largo o si requiere más opciones de compilación.

### 3.2.3 npm / Node.js

Use `npm` para instalar `solcjs`, un compilador de Solidity, de una manera portable y sencilla. El programa *solcjs* tiene menos funcionalidades que el resto de maneras de compilar detalladas más abajo en esta página. La documentación del compilador de línea de comandos asume que está usando el compilador con funcionalidad completa, `solc`. El uso de `solcjs` está documentado dentro de su propio repositorio.

Nota: El proyecto solc-js fue extraído del programa *solc* escrito en C++ usando Emscripten, lo que significa que ambos usan el mismo código fuente. *solc-js* puede usarse directamente en proyectos JavaScript (como Remix). Por favor diríjase al repositorio de solc-js para más instrucciones.

```
npm install -g solc
```

---

**Nota:** El ejecutable de línea de comandos es llamado `solcjs`.

Las opciones de línea de comandos de `solcjs` no son compatibles con `solc` y las herramientas (como `geth`) que esperen el comportamiento de `solc` no funcionarán con `solcjs`.

---

### 3.2.4 Docker

Las imágenes de Docker de Solidity están disponibles usando la imagen `solc` de la organización `ethereum`. Busque la etiqueta `stable` para la versión más reciente, y `nightly` para versiones con cambios potencialmente inestables de la rama de desarrollo.

La imagen de Docker corre el ejecutable del compilador, así que puede pasarle todos los argumentos del compilador. Por ejemplo, el comando de abajo toma la versión estable de la imagen de `solc` (si no la tiene todavía), y la corre en un contenedor nuevo, pasando el argumento `--help`.

```
docker run ethereum/solc:stable --help
```

También puede solicitar versiones de lanzamientos específicas, por ejemplo la versión 0.5.4.

```
docker run ethereum/solc:0.5.4 --help
```

Para usar una imagen de Docker para compilar archivos Solidity en la máquina huésped, monte una carpeta local para entrada y salida, y especifique el contrato a compilar. Por ejemplo.

```
docker run -v /local/path:/sources ethereum/solc:stable -o /sources/output --abi --bin /
↪sources/Contract.sol
```

También puede usar la interfaz standard de JSON (que es la recomendada cuando se usa el compilador con otras herramientas). Cuando use esta interfaz, no es necesario montar ningún directorio mientras el input de JSON sea autocontenido (ej. cuando no haga referencia a archivos externos que tendrían que ser cargados con un *import callback*).

```
docker run ethereum/solc:stable --standard-json < input.json > output.json
```

### 3.2.5 Paquetes de Linux

Los binarios de Solidity están disponibles en solidity/releases.

También tenemos PPAs para Ubuntu, puede obtener la versión estable más reciente usando los siguientes comandos:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

La versión nocturna puede ser instalada usando estos comandos:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install solc
```

Más aún, algunas distribuciones de Linux proveen sus propios paquetes. Estos paquetes no son mantenidos directamente por nosotros, sino por los desarrolladores de dichos paquetes.

Por ejemplo, Arch Linux tiene paquetes para la última versión de desarrollo:

```
pacman -S solidity
```

También hay un paquete para snap, sin embargo actualmente **no se le da mantenimiento**. Se puede instalar en todas las distros de Linux respaldadas. Para instalar la última versión estable de solc:

```
sudo snap install solc
```

Si usted quiere ayudar a probar la última versión de desarrollo de Solidity con los cambios más recientes, por favor utilice el siguiente comando:

```
sudo snap install solc --edge
```

**Nota:** El snap de `solc` usa confinamiento estricto. Este es el modo más seguro para paquetes de snap pero tiene sus limitaciones, como por ejemplo, el solo acceder a los archivos en sus directorios `/home` y `/media`. Para más información, vaya a Demystifying Snap Confinement.

### 3.2.6 Paquetes para macOs

Distribuimos el compilador de Solidity a través de Homebrew como una versión build-from-source. Los binarios pre-compilados no son respaldados actualmente.

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

Para instalar la versión 0.4.x / 0.5.x más reciente de Solidity también puede usar `brew install solidity@4` y `brew install solidity@5`, respectivamente.

Si necesita una versión específica de Solidity puede instalar una fórmula de Homebrew directamente desde Github.

Ver commits de solidity.rb en Github.

Copie el commit hash de la versión que quiere y haga check out en su máquina.

```
git clone https://github.com/ethereum/homebrew-ethereum.git
cd homebrew-ethereum
git checkout <el-hash-va-aquí>
```

Instalar usando `brew`:

```
brew unlink solidity
# ej. para instalar 0.4.8
brew install solidity.rb
```

### 3.2.7 Binarios estáticos

Mantenemos un repositorio con todos los binarios estáticos y versiones actuales del compilador para todas las plataformas respaldadas en solc-bin. En esta locación también puede encontrar las compilaciones nocturnas.

Este repositorio no es solamente una manera fácil y rápida para que los usuarios finales puedan obtener los archivos binarios listos para usarse; también está pensado para ser amigable para herramientas de terceros:

- El contenido está respaldado en https://binaries.soliditylang.org donde puede ser descargado por HTTPS sin autenticación, limitaciones de velocidad o la necesidad de usar git.
- El contenido es servido con las cabeceras de tipo *Content-Type* correctas y una configuración de CORS permisiva para que pueda ser cargado directamente por herramientas que corran en navegadores.
- Los binarios no requieren instalación o desempaquetado (con la excepción de algunas compilaciones antiguas para Windows que vienen con DLLs necesarios).
- Nos esforzamos por una alta compatibilidad con versiones anteriores. Una vez que añadimos un archivo, no lo movemos o borramos sin proveer un symlink/redirect a la locación anterior. Nunca los modificamos en su lugar y el checksum siempre debería coincidir. La única excepción es con archivos dañados o corrompidos que potencialmente podrían causar más daño de dejarse como están.
- Los archivos son servidos por HTTP y HTTPS. Mientras obtenga la lista de archivos de manera segura (via git, HTTPS, IPFS o esté cacheado localmente) y verifique los hashes de los binarios después de descargarlos, no necesita usar HTTPS para los binarios en sí mismos.

Los mismos binarios están disponibles la mayoría de las ocasiones en la página de releases de Solidity en Gihtub. La diferencia es que normalmente no actualizamos viejos releases en esa página. Eso significa que no los renombramos si las convenciones para nombrar archivos cambian, y tampoco añadimos compilaciones para plataformas que no estaban disponibles cuando se hizo el release. Esto solo sucede en `solc-bin`.

El repositorio de `solc-bin` contiene varios directorios de primer nivel, cada uno representando una plataforma. Cada uno contiene un archivo `list.json` que enlista los binarios disponibles. Por ejemplo, en `emscripten-wasm32/list.json` encontrará la siguiente información sobre la versión 0.7.4:

```
{
  "path": "solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js",
  "version": "0.7.4",
  "build": "commit.3f05b770",
  "longVersion": "0.7.4+commit.3f05b770",
  "keccak256": "0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3",
  "sha256": "0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2",
  "urls": [
    "bzzr://16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1",
```

(continué en la próxima página)

(proviene de la página anterior)

```
    "dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS"
  ]
}
```

Esto significa que:

- Puede encontrar el binario en el mismo directorio bajo el nombre solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. Note que el archivo puede ser un symlink, y necesitará resolverlo usted mismo si no está usando git para bajarlo a su sistema de archivos o su sistema de archivos no soporta symlinks.
- El binario también está respaldado en https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. En este caso no se necesita git y los symlinks son resueltos transparentemente, ya sea sirviendo una copia del archivo o regresando una redirección de HTTP.
- El archivo también está disponible en IPFS en QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS.
- El archivo puede estar disponible en Swarm en un futuro en 16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c95
- Usted puede verificar la integridad del binario comparando su hash keccak256 con `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3`. El hash puede ser computado en la línea de comandos usando la utilería `keccak256sum` que provee sha3sum o la función keccak256() de ethereumjs-util en JavaScript.
- Usted también puede verificar la integridad del binario comparando su hash sha256 con `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2`.

**Advertencia:** Debido a los fuertes requerimientos de compatibilidad con versiones anteriores, el repositorio contiene algunos elementos descontinuados que debería evitar usar al crear herramientas nuevas:

- Use `emscripten-wasm32/` (o en su defecto `emscripten-asmjs/`) en lugar de `bin/` si quiere el mejor rendimiento. Hasta la versión 0.6.1 solo proveíamos binarios de asm.js. A partir de la versión 0.6.2 hicimos el cambio a WebAssembly builds que tiene mucho mejor rendimiento. Hemos recompilado las versiones más antiguas para wasm pero los archivos originales asm.js se mantienen en `bin/`. Los nuevos tuvieron que ser puestos en un directorio separado para evitar conflictos de nombramiento.
- Use los directorios `emscripten-asmjs/` y `emscripten-wasm32/` en lugar de `bin/` y `wasm/` si quiere estar seguro de si está descargando un binario de wasm o de asm.js.
- Use `list.json` en vez de `list.js` y `list.txt`. El formato JSON contiene toda la información de los binarios antiguos y más.
- Use https://binaries.soliditylang.org en vez de https://solc-bin.ethereum.org. Para mantener las cosas simples, movimos casi todo lo relacionado al compilador al nuevo dominio `soliditylang.org` y esto también aplica para `solc-bin`. Aunque recomendamos el nuevo dominio, el antiguo sigue siendo mantenido y se garantiza que apunte a la misma locación.

**Advertencia:** Los binarios también están disponbiles en https://ethereum.github.io/solc-bin/ pero esta página dejó de ser actualizada justo después del lanzamiento de la versión 0.7.2, no va a recibir nuevos releases ni compilaciones nocturnas para ninguna plataforma y no sirve la nueva estructura del directorio, incluyendo compilaciones no-emscripten.

Si usted la está usando, por favor cambie a https://binaries.soliditylang.org, que es su reemplazo. Esto nos permite hacerle cambios al hosting subyacente de una manera transparente y minimizar disrupciones. Al contrario del dominio `ethereum.github.io`, el cual no controlamos, `binaries.soliditylang.org` está garantizado para funcionar y mantener la misma estructura de URLs en el largo plazo.

### 3.2.8 Compilando del código fuente

#### Prerrequisitos - Todos los Sistemas Operativos

A continuación las dependencias para todas las compilaciones de Solidity:

<<<<<<< HEAD | CMake (version 3.13+) | Generadord de código fuente multiplataforma. | +———————————+———————————————————-+ | Boost (version 1.77+ en | Librerías de C++. | | Windows, 1.65+ en otros) | | ======= | CMake (version 3.21.3+ on | Cross-platform build file generator. | | Windows, 3.13+ otherwise) | | +———————————+———————————————————-+ | Boost (version 1.77 on | C++ libraries. | | Windows, 1.65+ otherwise) | | >>>>>>> english/develop +———————————+———————————————————-+ | Git | Software de línea de comandos para obtener código | | | fuente. | +———————————+———————————————————-+ | z3 (versión 4.8+, Opcional) | Para usar con SMT checker. | | z3 (version 4.8.16+, Opcional) | Para usar con SMT checker. | +———————————+———————————————————-+ | cvc4 (Opcional) | Para usar con SMT checker. | +———————————+———————————————————-+

---

**Nota:** Las versiones de Solidity anteriores a la 0.5.10 pueden no funcionar correctamente con versiones de Boost 1.70+. Una posible solución es renombrar temporalmente `<Boost install path>/lib/cmake/Boost-1.70.0` antes de correr el comando de cmake para configurar Solidity.

A partir de la versión 0.5.10, debería funcionar correctamente con las versiones de Boost 1.70+ sin necesidad de ajustes manuales.

---

---

**Nota:** La configuración de la compilación por defecto requiere una versión de Z3 específica (la última en el momento que el código se actualizó por última vez). Los cambios que hay entre versiones de Z3 generalmente dan como resultado lanzamientos ligeramente distintos (pero válidos). Nuestras pruebas SMT no toman en cuenta estas diferencias y probablemente fallarán con una versión diferente de la versión para la que fueron escritas. Esto no significa que una compilación que use una versión diferente está dañada. Si usted pasa la opción `-DSTRICT_Z3_VERSION=OFF` a CMake, puede compilar con cualquier versión que satisfaga los requerimientos de la tabla de arriba. Si hace esto, sin embargo, recuerde pasar la opción `--no-smt` a `scripts/tests.sh` para saltar las pruebas SMT.

---

### 3.2.9 Versiones Mínimas del Compilador

---

**Nota:** De forma predeterminada, la compilación se realiza en *modo pedante*, lo que activa advertencias adicionales y le indica al compilador para tratar todas las advertencias como errores. Esto obliga a los desarrolladores a corregir las advertencias a medida que surgen, de modo que no acumulen que se solucionen más tarde. Si sólo le interesa crear una versión de lanzamiento y no desea modificar el código fuente para tratar estas advertencias, puede pasar la opción ``-DPEDANTIC=OFF` a CMake para deshabilitar este modo. No se recomienda hacerlo para uso general, pero puede ser necesario cuando se utiliza una cadena de herramientas que estamos no probar o intentar crear una versión anterior con herramientas más recientes. Si encuentra este tipo de advertencias, tenga en cuenta reporting them.

---

Los siguientes compiladores de C++ y sus versiones mínimas pueden compilar el código de Solidity:

- GCC, versión 8+
- Clang, versión 7+
- MSVC, versión 2019+

### Prerrequisitos - macOS

Para compilar en macOS, asegúrese que tenga la versión más reciente de Xcode instalada. Ésta contiene el compilador Clang C++, el IDE de Xcode y otras herramientas de desarrollo de Apple que se requieren para compilar aplicaciones C++ en OS X. Si está instalando Xcode por primera vez, o acaba de instalar una nueva versión necesitará aceptar la licencia antes de poder compilar desde la línea de comandos:

```
sudo xcodebuild -license accept
```

Nuestro script de compilación para OS X usa el gestor de paquetes Homebrew para instalar dependencias externas. Aquí puede ver cómo desinstalar Homebrew, si necesita empezar desde cero.

### Prerrequisitos - Windows

Necesitará instalar las siguientes dependencias para compilar Solidity en Windows:

| Software | Notas |
|---|---|
| Visual Studio 2019 Build Tools | Compilador de C++ |
| Visual Studio 2019 (Opcional) | Compilador de C++ y entorno de desarrollo |

<<<<<<< HEAD | Boost (versión 1.77+) | Librerías de C++ | ======= | Boost (version 1.77) | C++ libraries. | >>>>>>> english/develop +———————————+———————————————————————-+

Si ya tiene un IDE y solo necesita instalar el compilador y las librerías, puede instalar solamente Visual Studio 2019 Build Tools.

Visual Studio 2019 provee un IDE y el compilador y librerías necesarias. Si usted todavía no cuenta con un IDE de su preferencia y desea programar en Solidity, Visual Studio 2019 puede ser una opción para configurar todo de manera sencilla.

A continuación, una lista de componentes que deben ser instalados ya sea que use Visual Studio 2019 o Visual Studio 2019 Build Tools.

- Visual Studio C++ core features
- VC++ 2019 v141 toolset (x86,x64)
- Windows Universal CRT SDK
- Windows 8.1 SDK
- C++/CLI support

Tenemos un script de soporte que puede usar para instalar todas las dependencias externas necesarias:

```
scripts\install_deps.ps1
```

Esto instalará `boost` y `cmake` al subdirectorio `deps`.

### Clonar el repositorio

Para clonar el código fuente ejecute el siguiente comando:

```
git clone --recursive https://github.com/ethereum/solidity.git
cd solidity
```

Si usted quiere ayudar en el desarrollo de Solidity, le recomendamos hacer "fork" de Solidity y añadir su "fork" personal como segundo remoto:

```
git remote add personal git@github.com:[username]/solidity.git
```

---

**Nota:** Este método generará una precompilación que llevará a, por ejemplo, que se genere una bandera en cada bytecode producido por el compilador. Si quiere recompilar un compilador de Solidity ya lanzado, por favor use el archivo tarball en la página de lanzamientos de github:

https://github.com/ethereum/solidity/releases/download/v0.X.Y/solidity_0.X.Y.tar.gz

(no el «código fuente» de github).

---

### Compilación por la línea de comandos

**Asegúrese de instalar todas las dependencias externas (ver arriba) antes de compilar.**

El proyecto de Solidity utiliza CMake para configurar la compilación. Se recomienda instalar ccache para acelerar el proceso si requiere compilar en repetidas ocasiones. CMake lo detectará automaticamente. Compilar Solidity es bastante parecido en Linux, macOS y otras versiones de Unix:

```
mkdir build
cd build
cmake .. && make
```

En Linux y macOS es todavía más fácil, puede correr:

```
#nota: esto instalará los binarios solc y soltest en usr/local/bin
./scripts/build.sh
```

**Advertencia:** Las compilaciones en BSD deberían funcionar, pero no han sido probadas por el equipo de Solidity.

Y para Windows:

```
mkdir build
cd build
cmake -G "Visual Studio 16 2019" ..
```

En caso de querer usar la versión de boost instalada por `scripts\install_deps.ps1`, deberá pasar adicionalmente las banderas `-DBoost_DIR="deps\boost\lib\cmake\Boost-*"` y `-DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded` como argumentos cuando corra `cmake`.

Esto debería crear un archivo **solidity.sln** en ese directorio. Al hacer doble click en ese archivo Visual Studio debería abrirse. Sugerimos compilar con la configuración **Release** pero todas las demás funcionan.

De manera alternativa, puede compilar en Windows desde la línea de comandos:

```
cmake --build . --config Release
```

### 3.2.10 Opciones de CMake

Para ver las opciones de Cmake disponibles solo corra `cmake .. -LH`.

#### Solvers de SMT

Solidity puede ser compilado contra solvers SMT y lo hará por defecto si se encuentran en el sistema. Cada solver puede ser deshabilitado con una opción de *cmake*.

*Nota: En algunos casos, esto también puede ser una solución alterna para compilaciones fallidas.*

En la carpeta donde compile puede deshabilitarlos, ya que están habilitados por defecto:

```
# deshabilita solo el solver SMT para Z3
cmake .. -DUSE_Z3=OFF

# deshabilita solo el solver SMT para CVC4
cmake .. -DUSE_CVC4=OFF

# deshabilita los solvers para CVC4 y Z3
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

### 3.2.11 La Cadena de Versión en Detalle

La cadena de versión de Solidity contiene cuatro partes:

- el número de versión
- la etiqueta de pre-release, generalmente fijada en `develop.YYYY.MM.DD` o `nightly.YYYY.MM.DD`
- commit en el formato `commit.GITHASH`
- la plataforma, que contiene un número arbitrario de elementos, e incluye detalles sobre la plataforma y el compilador

Si hay modificaciones locales, el commit tendrá el sufijo de `.mod`.

Estas partes se combinan según requerimientos de SemVer, donde la etiqueta de pre-release de Solidity equivale al pre-release de SemVer y el commit de Solidity y la plataforma combinados nos dan la metadata del compilado de SemVer.

Un ejemplo de release: `0.4.8+commit.60cc1668.Emscripten.clang`.

Un ejemplo de pre-release: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

### 3.2.12 Información Importante Sobre el Versionado

Después de que se hace un lanzamiento, se aumenta el número de versión de parche, porque asumimos que solo siguen cambios a nivel de parches. Cuando los cambios se fusionan, la versión debería ser aumentada de acuerdo a SemVer y a la severidad del cambio. Finalmente, un lanzamiento siempre se hace con la versión de la compilación nocturna vigente, pero sin el especificador de `prerelease`.

Ejemplo:

1. Se hace el release de la versión 0.4.0.
2. La compilación nocturna tiene una versión de 0.4.1 de ahora en adelante.
3. Se introducen cambios sin rupturas –> no hay cambio en la versión.
4. Se introduce un cambio con rupturas –> la versión se aumenta a 0.5.0.
5. Se hace el release de la versión 0.5.0.

Este método funciona bien con la *versión de pragma*.

## 3.3 Solidity con Ejemplos

### 3.3.1 Voting

The following contract is quite complex, but showcases a lot of Solidity's features. It implements a voting contract. Of course, the main problems of electronic voting is how to assign voting rights to the correct persons and how to prevent manipulation. We will not solve all problems here, but at least we will show how delegated voting can be done so that vote counting is **automatic and completely transparent** at the same time.

The idea is to create one contract per ballot, providing a short name for each option. Then the creator of the contract who serves as chairperson will give the right to vote to each address individually.

The persons behind the addresses can then choose to either vote themselves or to delegate their vote to a person they trust.

At the end of the voting time, `winningProposal()` will return the proposal with the largest number of votes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of `proposalNames`.
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Give `voter` the right to vote on this ballot.
    // May only be called by `chairperson`.
    function giveRightToVote(address voter) external {
        // If the first argument of `require` evaluates
        // to `false`, execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use `require` to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
```

(continué en la próxima página)

(proviene de la página anterior)

```
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) external {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "You have no right to vote");
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in delegation.");
        }

        Voter storage delegate_ = voters[to];

        // Voters cannot delegate to accounts that cannot vote.
        require(delegate_.weight >= 1);

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender]`.
        sender.voted = true;
        sender.delegate = to;

        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) external {
        Voter storage sender = voters[msg.sender];
```

(continué en la próxima página)

(proviene de la página anterior)

```
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() external view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

#### Possible Improvements

Currently, many transactions are needed to assign the rights to vote to all participants. Moreover, if two or more proposals have the same number of votes, `winningProposal()` is not able to register a tie. Can you think of a way to fix these issues?

### 3.3.2 Blind Auction

In this section, we will show how easy it is to create a completely blind auction contract on Ethereum. We will start with an open auction where everyone can see the bids that are made and then extend this contract into a blind auction where it is not possible to see the actual bid until the bidding period ends.

### Simple Open Auction

The general idea of the following simple auction contract is that everyone can send their bids during a bidding period. The bids already include sending money / Ether in order to bind the bidders to their bid. If the highest bid is raised, the previous highest bidder gets their money back. After the end of the bidding period, the contract has to be called manually for the beneficiary to receive their money - contracts cannot activate themselves.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
    bool ended;

    // Events that will be emitted on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Errors that describe failures.

    // The triple-slash comments are so-called natspec
    // comments. They will be shown when the user
    // is asked to confirm a transaction or
    // when an error is displayed.

    /// The auction has already ended.
    error AuctionAlreadyEnded();
    /// There is already a higher or equal bid.
    error BidNotHighEnough(uint highestBid);
    /// The auction has not ended yet.
    error AuctionNotYetEnded();
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    /// Create a simple auction with `biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `beneficiaryAddress`.
    constructor(
        uint biddingTime,
        address payable beneficiaryAddress
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ) {
        beneficiary = beneficiaryAddress;
        auctionEndTime = block.timestamp + biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() external payable {
        // No arguments are necessary, all
        // information is already part of
        // the transaction. The keyword payable
        // is required for the function to
        // be able to receive Ether.

        // Revert the call if the bidding
        // period is over.
        if (block.timestamp > auctionEndTime)
            revert AuctionAlreadyEnded();

        // If the bid is not higher, send the
        // money back (the revert statement
        // will revert all changes in this
        // function execution including
        // it having received the money).
        if (msg.value <= highestBid)
            revert BidNotHighEnough(highestBid);

        if (highestBid != 0) {
            // Sending back the money by simply using
            // highestBidder.send(highestBid) is a security risk
            // because it could execute an untrusted contract.
            // It is always safer to let the recipients
            // withdraw their money themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() external returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `send` returns.
            pendingReturns[msg.sender] = 0;

            // msg.sender is not of type `address payable` and must be
```

(continué en la próxima página)

(proviene de la página anterior)

```
            // explicitly converted using `payable(msg.sender)` in order
            // use the member function `send()`.
            if (!payable(msg.sender).send(amount)) {
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() external {
        // It is a good guideline to structure functions that interact
        // with other contracts (i.e. they call functions or send Ether)
        // into three phases:
        // 1. checking conditions
        // 2. performing actions (potentially changing conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could call
        // back into the current contract and modify the state or cause
        // effects (ether payout) to be performed multiple times.
        // If functions called internally include interaction with external
        // contracts, they also have to be considered interaction with
        // external contracts.

        // 1. Conditions
        if (block.timestamp < auctionEndTime)
            revert AuctionNotYetEnded();
        if (ended)
            revert AuctionEndAlreadyCalled();

        // 2. Effects
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
        beneficiary.transfer(highestBid);
    }
}
```

### Blind Auction

The previous open auction is extended to a blind auction in the following. The advantage of a blind auction is that there is no time pressure towards the end of the bidding period. Creating a blind auction on a transparent computing platform might sound like a contradiction, but cryptography comes to the rescue.

During the **bidding period**, a bidder does not actually send their bid, but only a hashed version of it. Since it is currently considered practically impossible to find two (sufficiently long) values whose hash values are equal, the bidder commits to the bid by that. After the end of the bidding period, the bidders have to reveal their bids: They send their values unencrypted, and the contract checks that the hash value is the same as the one provided during the bidding period.

Another challenge is how to make the auction **binding and blind** at the same time: The only way to prevent the bidder from just not sending the money after they won the auction is to make them send it together with the bid. Since value transfers cannot be blinded in Ethereum, anyone can see the value.

The following contract solves this problem by accepting any value that is larger than the highest bid. Since this can of course only be checked during the reveal phase, some bids might be **invalid**, and this is on purpose (it even provides an explicit flag to place invalid bids with high-value transfers): Bidders can confuse competition by placing several high or low invalid bids.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    // Errors that describe failures.

    /// The function has been called too early.
    /// Try again at `time`.
    error TooEarly(uint time);
    /// The function has been called too late.
    /// It cannot be called after `time`.
    error TooLate(uint time);
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    // Modifiers are a convenient way to validate inputs to
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // functions. `onlyBefore` is applied to `bid` below:
    // The new function body is the modifier's body where
    // `_` is replaced by the old function body.
    modifier onlyBefore(uint time) {
        if (block.timestamp >= time) revert TooLate(time);
        _;
    }
    modifier onlyAfter(uint time) {
        if (block.timestamp <= time) revert TooEarly(time);
        _;
    }

    constructor(
        uint biddingTime,
        uint revealTime,
        address payable beneficiaryAddress
    ) {
        beneficiary = beneficiaryAddress;
        biddingEnd = block.timestamp + biddingTime;
        revealEnd = biddingEnd + revealTime;
    }

    /// Place a blinded bid with `blindedBid` =
    /// keccak256(abi.encodePacked(value, fake, secret)).
    /// The sent ether is only refunded if the bid is correctly
    /// revealed in the revealing phase. The bid is valid if the
    /// ether sent together with the bid is at least "value" and
    /// "fake" is not true. Setting "fake" to true and sending
    /// not the exact amount are ways to hide the real bid but
    /// still make the required deposit. The same address can
    /// place multiple bids.
    function bid(bytes32 blindedBid)
        external
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: blindedBid,
            deposit: msg.value
        }));
    }

    /// Reveal your blinded bids. You will get a refund for all
    /// correctly blinded invalid bids and for all bids except for
    /// the totally highest.
    function reveal(
        uint[] calldata values,
        bool[] calldata fakes,
        bytes32[] calldata secrets
    )
        external
        onlyAfter(biddingEnd)
```

(continué en la próxima página)

(proviene de la página anterior)

```
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(values.length == length);
        require(fakes.length == length);
        require(secrets.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (values[i], fakes[i], secrets[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,␣
↪secret))) {
                // Bid was not actually revealed.
                // Do not refund deposit.
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Make it impossible for the sender to re-claim
            // the same deposit.
            bidToCheck.blindedBid = bytes32(0);
        }
        payable(msg.sender).transfer(refund);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() external {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `transfer` returns (see the remark above about
            // conditions -> effects -> interaction).
            pendingReturns[msg.sender] = 0;

            payable(msg.sender).transfer(amount);
        }
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd()
        external
        onlyAfter(revealEnd)
    {
        if (ended) revert AuctionEndAlreadyCalled();
        emit AuctionEnded(highestBidder, highestBid);
```

(continué en la próxima página)

(proviene de la página anterior)

```
        ended = true;
        beneficiary.transfer(highestBid);
    }

    // This is an "internal" function which means that it
    // can only be called from the contract itself (or from
    // derived contracts).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }
}
```

### 3.3.3 Safe Remote Purchase

Purchasing goods remotely currently requires multiple parties that need to trust each other. The simplest configuration involves a seller and a buyer. The buyer would like to receive an item from the seller and the seller would like to get money (or an equivalent) in return. The problematic part is the shipment here: There is no way to determine for sure that the item arrived at the buyer.

There are multiple ways to solve this problem, but all fall short in one or the other way. In the following example, both parties have to put twice the value of the item into the contract as escrow. As soon as this happened, the money will stay locked inside the contract until the buyer confirms that they received the item. After that, the buyer is returned the value (half of their deposit) and the seller gets three times the value (their deposit plus the value). The idea behind this is that both parties have an incentive to resolve the situation or otherwise their money is locked forever.

This contract of course does not solve the problem, but gives an overview of how you can use state machine-like constructs inside a contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // The state variable has a default value of the first member, `State.created`
    State public state;

    modifier condition(bool condition_) {
        require(condition_);
```

(continué en la próxima página)

(proviene de la página anterior)

```
        _;
    }

    /// Only the buyer can call this function.
    error OnlyBuyer();
    /// Only the seller can call this function.
    error OnlySeller();
    /// The function cannot be called at the current state.
    error InvalidState();
    /// The provided value has to be even.
    error ValueNotEven();

    modifier onlyBuyer() {
        if (msg.sender != buyer)
            revert OnlyBuyer();
        _;
    }

    modifier onlySeller() {
        if (msg.sender != seller)
            revert OnlySeller();
        _;
    }

    modifier inState(State state_) {
        if (state != state_)
            revert InvalidState();
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();
    event SellerRefunded();

    // Ensure that `msg.value` is an even number.
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd number.
    constructor() payable {
        seller = payable(msg.sender);
        value = msg.value / 2;
        if ((2 * value) != msg.value)
            revert ValueNotEven();
    }

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        external
        onlySeller
        inState(State.Created)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    {
        emit Aborted();
        state = State.Inactive;
        // We use transfer here directly. It is
        // reentrancy-safe, because it is the
        // last call in this function and we
        // already changed the state.
        seller.transfer(address(this).balance);
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
    function confirmPurchase()
        external
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = payable(msg.sender);
        state = State.Locked;
    }

    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        external
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Release;

        buyer.transfer(value);
    }

    /// This function refunds the seller, i.e.
    /// pays back the locked funds of the seller.
    function refundSeller()
        external
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
```

(continué en la próxima página)

(proviene de la página anterior)

```
        state = State.Inactive;

        seller.transfer(3 * value);
    }
}
```

### 3.3.4 Micropayment Channel

In this section, we will learn how to build an example implementation of a payment channel. It uses cryptographic signatures to make repeated transfers of Ether between the same parties secure, instantaneous, and without transaction fees. For the example, we need to understand how to sign and verify signatures, and setup the payment channel.

#### Creating and verifying signatures

Imagine Alice wants to send some Ether to Bob, i.e. Alice is the sender and Bob is the recipient.

Alice only needs to send cryptographically signed messages off-chain (e.g. via email) to Bob and it is similar to writing checks.

Alice and Bob use signatures to authorize transactions, which is possible with smart contracts on Ethereum. Alice will build a simple smart contract that lets her transmit Ether, but instead of calling a function herself to initiate a payment, she will let Bob do that, and therefore pay the transaction fee.

The contract will work as follows:

1. Alice deploys the `ReceiverPays` contract, attaching enough Ether to cover the payments that will be made.
2. Alice authorizes a payment by signing a message with her private key.
3. Alice sends the cryptographically signed message to Bob. The message does not need to be kept secret (explained later), and the mechanism for sending it does not matter.
4. Bob claims his payment by presenting the signed message to the smart contract, it verifies the authenticity of the message and then releases the funds.

#### Creating the signature

Alice does not need to interact with the Ethereum network to sign the transaction, the process is completely offline. In this tutorial, we will sign messages in the browser using web3.js and MetaMask, using the method described in EIP-712, as it provides a number of other security benefits.

```
/// Hashing first makes things easier
var hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log("Signed
↪"); });
```

**Nota:** The `web3.eth.personal.sign` prepends the length of the message to the signed data. Since we hash first, the message will always be exactly 32 bytes long, and thus this length prefix is always the same.

### What to Sign

For a contract that fulfils payments, the signed message must include:

1. The recipient's address.
2. The amount to be transferred.
3. Protection against replay attacks.

A replay attack is when a signed message is reused to claim authorization for a second action. To avoid replay attacks we use the same technique as in Ethereum transactions themselves, a so-called nonce, which is the number of transactions sent by an account. The smart contract checks if a nonce is used multiple times.

Another type of replay attack can occur when the owner deploys a `ReceiverPays` smart contract, makes some payments, and then destroys the contract. Later, they decide to deploy the `RecipientPays` smart contract again, but the new contract does not know the nonces used in the previous deployment, so the attacker can use the old messages again.

Alice can protect against this attack by including the contract's address in the message, and only messages containing the contract's address itself will be accepted. You can find an example of this in the first two lines of the `claimPayment()` function of the full contract at the end of this section.

### Packing arguments

Now that we have identified what information to include in the signed message, we are ready to put the message together, hash it, and sign it. For simplicity, we concatenate the data. The ethereumjs-abi library provides a function called `soliditySHA3` that mimics the behaviour of Solidity's `keccak256` function applied to arguments encoded using `abi.encodePacked`. Here is a JavaScript function that creates the proper signature for the `ReceiverPays` example:

```
// recipient is the address that should be paid.
// amount, in wei, specifies how much ether should be sent.
// nonce can be any unique number to prevent replay attacks
// contractAddress is used to prevent cross-contract replay attacks
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + abi.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

    web3.eth.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

### Recovering the Message Signer in Solidity

In general, ECDSA signatures consist of two parameters, `r` and `s`. Signatures in Ethereum include a third parameter called `v`, that you can use to verify which account's private key was used to sign the message, and the transaction's sender. Solidity provides a built-in function *ecrecover* that accepts a message along with the `r`, `s` and `v` parameters and returns the address that was used to sign the message.

### Extracting the Signature Parameters

Signatures produced by web3.js are the concatenation of `r`, `s` and `v`, so the first step is to split these parameters apart. You can do this on the client-side, but doing it inside the smart contract means you only need to send one signature parameter rather than three. Splitting apart a byte array into its constituent parts is a mess, so we use *inline assembly* to do the job in the `splitSignature` function (the third function in the full contract at the end of this section).

### Computing the Message Hash

The smart contract needs to know exactly what parameters were signed, and so it must recreate the message from the parameters and use that for signature verification. The functions `prefixed` and `recoverSigner` do this in the `claimPayment` function.

### The full contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// This will report a warning due to deprecated selfdestruct
contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature)␣
↪external {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // this recreates the message that was signed on the client
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce,␣
↪this)));

        require(recoverSigner(message, signature) == owner);

        payable(msg.sender).transfer(amount);
    }

    /// destroy the contract and reclaim the leftover funds.
    function shutdown() external {
        require(msg.sender == owner);
        selfdestruct(payable(msg.sender));
    }

    /// signature methods.
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix.
            r := mload(add(sig, 32))
            // second 32 bytes.
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes).
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

### Writing a Simple Payment Channel

Alice now builds a simple but complete implementation of a payment channel. Payment channels use cryptographic signatures to make repeated transfers of Ether securely, instantaneously, and without transaction fees.

### What is a Payment Channel?

Payment channels allow participants to make repeated transfers of Ether without using transactions. This means that you can avoid the delays and fees associated with transactions. We are going to explore a simple unidirectional payment channel between two parties (Alice and Bob). It involves three steps:

1. Alice funds a smart contract with Ether. This «opens» the payment channel.
2. Alice signs messages that specify how much of that Ether is owed to the recipient. This step is repeated for each payment.
3. Bob «closes» the payment channel, withdrawing his portion of the Ether and sending the remainder back to the sender.

**Nota:** Only steps 1 and 3 require Ethereum transactions, step 2 means that the sender transmits a cryptographically signed message to the recipient via off chain methods (e.g. email). This means only two transactions are required to

support any number of transfers.

Bob is guaranteed to receive his funds because the smart contract escrows the Ether and honours a valid signed message. The smart contract also enforces a timeout, so Alice is guaranteed to eventually recover her funds even if the recipient refuses to close the channel. It is up to the participants in a payment channel to decide how long to keep it open. For a short-lived transaction, such as paying an internet café for each minute of network access, the payment channel may be kept open for a limited duration. On the other hand, for a recurring payment, such as paying an employee an hourly wage, the payment channel may be kept open for several months or years.

### Opening the Payment Channel

To open the payment channel, Alice deploys the smart contract, attaching the Ether to be escrowed and specifying the intended recipient and a maximum duration for the channel to exist. This is the function `SimplePaymentChannel` in the contract, at the end of this section.

### Making Payments

Alice makes payments by sending signed messages to Bob. This step is performed entirely outside of the Ethereum network. Messages are cryptographically signed by the sender and then transmitted directly to the recipient.

Each message includes the following information:

- The smart contract's address, used to prevent cross-contract replay attacks.
- The total amount of Ether that is owed to the recipient so far.

A payment channel is closed just once, at the end of a series of transfers. Because of this, only one of the messages sent is redeemed. This is why each message specifies a cumulative total amount of Ether owed, rather than the amount of the individual micropayment. The recipient will naturally choose to redeem the most recent message because that is the one with the highest total. The nonce per-message is not needed anymore, because the smart contract only honours a single message. The address of the smart contract is still used to prevent a message intended for one payment channel from being used for a different channel.

Here is the modified JavaScript code to cryptographically sign a message from the previous section:

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.solditySHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

function signMessage(message, callback) {
    web3.eth.personal.sign(
        "0x" + message.toString("hex"),
        web3.eth.defaultAccount,
        callback
    );
}

// contractAddress is used to prevent cross-contract replay attacks.
// amount, in wei, specifies how much Ether should be sent.
```

(continué en la próxima página)

(proviene de la página anterior)

```
function signPayment(contractAddress, amount, callback) {
    var message = constructPaymentMessage(contractAddress, amount);
    signMessage(message, callback);
}
```

### Closing the Payment Channel

When Bob is ready to receive his funds, it is time to close the payment channel by calling a `close` function on the smart contract. Closing the channel pays the recipient the Ether they are owed and destroys the contract, sending any remaining Ether back to Alice. To close the channel, Bob needs to provide a message signed by Alice.

The smart contract must verify that the message contains a valid signature from the sender. The process for doing this verification is the same as the process the recipient uses. The Solidity functions `isValidSignature` and `recoverSigner` work just like their JavaScript counterparts in the previous section, with the latter function borrowed from the `ReceiverPays` contract.

Only the payment channel recipient can call the `close` function, who naturally passes the most recent payment message because that message carries the highest total owed. If the sender were allowed to call this function, they could provide a message with a lower amount and cheat the recipient out of what they are owed.

The function verifies the signed message matches the given parameters. If everything checks out, the recipient is sent their portion of the Ether, and the sender is sent the rest via a `selfdestruct`. You can see the `close` function in the full contract.

### Channel Expiration

Bob can close the payment channel at any time, but if they fail to do so, Alice needs a way to recover her escrowed funds. An *expiration* time was set at the time of contract deployment. Once that time is reached, Alice can call `claimTimeout` to recover her funds. You can see the `claimTimeout` function in the full contract.

After this function is called, Bob can no longer receive any Ether, so it is important that Bob closes the channel before the expiration is reached.

### The full contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// This will report a warning due to deprecated selfdestruct
contract SimplePaymentChannel {
    address payable public sender;      // The account sending payments.
    address payable public recipient;   // The account receiving the payments.
    uint256 public expiration;  // Timeout in case the recipient never closes.

    constructor (address payable recipientAddress, uint256 duration)
        payable
    {
        sender = payable(msg.sender);
        recipient = recipientAddress;
        expiration = block.timestamp + duration;
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    /// the recipient can close the channel at any time by presenting a
    /// signed amount from the sender. the recipient will be sent that amount,
    /// and the remainder will go back to the sender
    function close(uint256 amount, bytes memory signature) external {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

        recipient.transfer(amount);
        selfdestruct(sender);
    }

    /// the sender can extend the expiration at any time
    function extend(uint256 newExpiration) external {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// if the timeout is reached without the recipient closing the channel,
    /// then the Ether is released back to the sender.
    function claimTimeout() external {
        require(block.timestamp >= expiration);
        selfdestruct(sender);
    }

    function isValidSignature(uint256 amount, bytes memory signature)
        internal
        view
        returns (bool)
    {
        bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));

        // check that the signature is from the payment sender
        return recoverSigner(message, signature) == sender;
    }

    /// All functions below this are just taken from the chapter
    /// 'creating and verifying signatures' chapter.

    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
            // second 32 bytes
```

(continué en la próxima página)

(proviene de la página anterior)

```
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes)
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

**Nota:** The function `splitSignature` does not use all security checks. A real implementation should use a more rigorously tested library, such as openzepplin's version of this code.

### Verifying Payments

Unlike in the previous section, messages in a payment channel aren't redeemed right away. The recipient keeps track of the latest message and redeems it when it's time to close the payment channel. This means it's critical that the recipient perform their own verification of each message. Otherwise there is no guarantee that the recipient will be able to get paid in the end.

The recipient should verify each message using the following process:

1. Verify that the contract address in the message matches the payment channel.
2. Verify that the new total is the expected amount.
3. Verify that the new total does not exceed the amount of Ether escrowed.
4. Verify that the signature is valid and comes from the payment channel sender.

We'll use the ethereumjs-util library to write this verification. The final step can be done a number of ways, and we use JavaScript. The following code borrows the `constructPaymentMessage` function from the signing **JavaScript code** above:

```
// this mimics the prefixing behavior of the eth_sign JSON-RPC method.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
```

(continué en la próxima página)

(proviene de la página anterior)

```
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}

function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}
```

### 3.3.5 Modular Contracts

A modular approach to building your contracts helps you reduce the complexity and improve the readability which will help to identify bugs and vulnerabilities during development and code review. If you specify and control the behaviour of each module in isolation, the interactions you have to consider are only those between the module specifications and not every other moving part of the contract. In the example below, the contract uses the `move` method of the `Balances` *library* to check that balances sent between addresses match what you expect. In this way, the `Balances` library provides an isolated component that properly tracks balances of accounts. It is easy to verify that the `Balances` library never produces negative balances or overflows and the sum of all balances is an invariant across the lifetime of the contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address to,
↪ uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping(address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function transfer(address to, uint amount) external returns (bool success) {
        balances.move(msg.sender, to, amount);
```

(continué en la próxima página)

(proviene de la página anterior)

```
        emit Transfer(msg.sender, to, amount);
        return true;

    }

    function transferFrom(address from, address to, uint amount) external returns (bool␣
↪success) {
        require(allowed[from][msg.sender] >= amount);
        allowed[from][msg.sender] -= amount;
        balances.move(from, to, amount);
        emit Transfer(from, to, amount);
        return true;
    }

    function approve(address spender, uint tokens) external returns (bool success) {
        require(allowed[msg.sender][spender] == 0, "");
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }

    function balanceOf(address tokenOwner) external view returns (uint balance) {
        return balances[tokenOwner];
    }
}
```

## 3.4 Composición de un archivo fuente en Solidity

Los archivos fuente pueden contener un número arbitrario de *definiciones de contratos*, sentencias *import*, *pragma* y directrices *using for* y *struct*, *enum*, *function*, *error* y definiciones de *constant variable*.

### 3.4.1 Identificador de licencia SPDX

La confianza en los contratos inteligentes se puede establecer mejor si sus código fuente está disponible. Debido a que disponer del código fuente disponible siempre roza problemas legales con respecto a derechos de autor, el compilador de Solidity fomenta el uso de identificadores de licencia SPDX legibles por máquinas. Cada archivo fuente debería comenzar con un comentario que indique su licencia:

```
// SPDX-License-Identifier: MIT
```

El compilador no valida que la licencia sea parte de la lista permitida por SPDX pero sí incluye el string provisto en los metadatos bytecode.

Si no quiere especificar una licencia o si el código fuente no es de código abierto, por favor use el valor especial `UNLICENSED`. Note que `UNLICENSED` (uso no permitido, no presente en la lista de licencias de SPDX) es diferente de `UNLICENSE` (concede todos los derechos). Solidity sigue la recomendación de npm.

Proporcionar este comentario, por supuesto, no lo exime de otras obligaciones relacionadas con licencias como tener que mencionar un encabezado específico de licencias en cada archivo fuente o el titular de los derechos originales.

El comentario es reconocido por el compilador en cualquier parte del archivo al nivel de archivos, pero se recomienda ponerlo en la parte superior del archivo.

Más información sobre cómo usar los identificadores de licencias SPDX puede ser encontrado en el sitio web de SPDX.

### 3.4.2 Pragmas

La palabra reservada `pragma` se usa para permitir ciertas características del compilador o verificaciones. Una directriz pragma siempre es local a un archivo fuente, de modo que tiene que agregar la directriz pragma a todos sus archivos si quiere habilitarlo en el proyecto entero. Si usted *import* otro archivo, la directriz pragma desde ese archivo *no* se aplica automáticamente al archivo de importación.

#### Version Pragma

Los archivos fuente pueden (y deberían) ser anotados con una versión del pragma para rechazar la compilación con versiones futuras del compilador que podrían introducir cambios incompatibles. Nosotros intentamos mantenerlos al mínimo estrictamente necesario e introducirlos de una manera que los cambios en la semántica también requieran cambios en la sintaxis, pero esto no es siempre posible. Debido a ello, siempre es una buena idea leer el registro de modificaciones al menos para los lanzamientos que contengan cambios de ruptura. Estos lanzamientos siempre tienen versiones de la forma `0.x.0` o `x.0.0`.

La versión del pragma se usa de la siguiente manera: `pragma solidity ^0.5.2;`

Un archivo fuente con la línea de arriba no compila con un compilador anterior a la versión 0.5.2 y tampoco funciona en un compilador que inicie con la versión 0.6.0 (esta segunda condición se agrega al usar `^`). Debido a que no habrá cambios de ruptura hasta la versión `0.6.0`, puede estar seguro que su código compila de la forma que esperaba. La versión exacta del compilador no es fija, por lo tanto, las versiones de corrección aun son posibles.

Es posible especificar reglas más complejas para la versión del compilador, estas siguen la misma sintaxis usaba por npm.

---

**Nota:** El uso de la versión del pragma *no* cambia la versión del compilador *ni* habilita o deshabilita características del compilador. Simplemente indica al compilador que verifique si su versión corresponde a la requerida por el pragma. Si no corresponde, el compilador emite un error.

---

#### ABI Coder Pragma

Al usar `pragma abicoder v1` o `pragma abicoder v2` puedes seleccionar entre las dos implementaciones del codificador y decodificador ABI.

El nuevo codificador ABI (v2) es capaz de codificar y decodificar arrays y structs anidadas arbitrariamente. Además de admitir más tipos, implica una validación y comprobaciones de seguridad más amplias, lo que puede dar como resultado mayores costes de gas, pero también una mayor seguridad. Se considera no experimental a partir de Solidity 0.6.0 y está habilitado de forma predeterminada iniciando con Solidity 0.8.0. El antiguo codificador ABI todavía puede ser seleccionado usando `pragma abicoder v1;`.

El conjunto de tipos soportados por el nuevo codificador es un superset de aquellos soportados por el viejo. Los contratos que lo usan pueden interactuar con aquellos que no lo usan sin limitaciones. Lo opuesto es posible solo siempre y cuando el contrato no-`abicoder v2` no intente hacer llamadas que requerirían decodificador tipos solamente soportados por el nuevo codificador. El compilador puede detectar esto y emitirá un error. Simplemente con activar `abicoder v2` para su contrato es suficiente para hacer que estos errores desaparezcan.

---

**Nota:** Este pragma aplica a todo el código definido en el archivo donde está activado, sin reparar en donde ese código termina finalmente. Esto significa que un contrato cuyo archivo fuente está seleccionado para compilar con ABI coder

v1 aun puede contener código que usa el nuevo codificador al heredarlo de otro contrato. Esto se permite si los nuevos tipos son solamente usados internamente y no en signaturas de funciones externas.

**Nota:** Hasta Solidity 0.7.4 fue posible seleccionar el ABI coder v2 al usar `pragma experimental ABIEncoderV2`, pero no era posible seleccionar el codificador v1 explícitamente porque estaba por defecto.

#### Pragma Experimental

El segundo pragma es el pragma experimental. Puede ser usado para habilitar características del compilador o lenguaje que todavía no están activadas por defecto. Los siguientes pragmas experimentales están actualmente soportados:

#### ABIEncoderV2

Debido a que el codificador ABI v2 ya no es considerado experimental, puede ser seleccionado por medio de `pragma abicoder v2` desde Solidity 0.7.4 (véase más arriba).

#### SMTChecker

Este componente tiene que ser habilitado cuando el compilador de Solidity es construido y, por lo tanto, no está disponible en todos los binarios Solidity. Las *instrucciones de construcción* explican cómo activar esta opción. Está activado para todas los lanzamientos PPA de Ubuntu en la mayoría de las versiones, pero no para las imágenes de Docker, los binarios de Windows o los binarios de Linux construidos estáticamente. Se puede activar para solc-js a través de smtCallback si tiene un SMT solver instalado localmente y ejecute solc-js por medio de node (no a través del navegador).

Si usa `pragma experimental SMTChecker;`, entonces obtiene *avisos de seguridad* adicionales los cuales se obtienen al consultar un SMT solver. El componente todavía no soporta todas las características del lenguaje Solidity y probablemente genera muchas advertencias. En caso de que señale características no soportadas, el análisis pudiese no ser enteramente sólido.

### 3.4.3 Importación de Otros Archivos Fuente

#### Sintaxis y Semántica

Solidity soporta sentencias import para ayudar a modularizar su código, similar a aquellas disponibles en JavaScript (a partir de ES6). Sin embargo, Solidity no soporta el concepto default export.

A un nivel global, puede usar sentencias import de la siguiente forma:

```
import "filename";
```

La parte `filename` se llama *ruta de importación*. Esta sentencia importa todos los símbolos globales desde «filename» (y símbolos importados allí) a el scope global actual (diferente de ES6 pero retrocompatible para Solidity). No se recomienda usar esta forma porque corrompe de modo impredecible el espacio de nombres. Si agregas nuevos elementos de alto nivel dentro de «filename», aparece autoáticamente en todos los archivos que importan de esta manera desde «filename». Es mejor importar símbolos específicos explícitamente.

El siguiente ejemplo crea un nuevo símbolo global `symbolName` cuyos miembros son todos los símbolos globañes desde «filename»:

```
import * as symbolName from "filename";
```

lo cual resulta en todos los símbolos globales disponibles en el formato `symbolName.symbol`.

Una variante de esta sintaxis que no es parte de ES6, pero posiblemente útil es:

```
import "filename" as symbolName;
```

lo cual es equivalente a `import * as symbolName from "filename";`.

Si hay una colisión de nombres, puede renombrar símbolos durante la importación. Por ejemplo, el código debajo crea símbolos globales nuevos `alias` y `symbol2` los cuales referencian `symbol1` y `symbol2` desde dentro de «filename» respectivamente.

```
import {symbol1 as alias, symbol2} from "filename";
```

#### Rutas de Importación

A fin de ser capaz de soportar construcciones reproducibles en todas las plataformas, el compilador de Solidity tiene que abstraer los detalles del sistema de archivos en donde los archivos fuente están almacenados. Por esta razón las rutas de importación no hacen referencia directamente a los archivos en el sistema de archivos host. En lugar de ello, el compilador mantiene una base de datos interna (*sistema de archivos virtual* o *VFS* de forma resumida) donde cada unidad fuente se asigna a un único *nombre de unidad fuente* el cual es un identificador opaco y desestructurado. La ruta de importación especificada en una sentencia import se traduce a un nombre de unidad fuente y usada para encontrar la unidad de fuente correspondiente en esta base de datos.

Al usar la API *Standard JSON* es posible proveer directamente los nombres y contenido de todos los archivos fuentes como parte de la entrada del compilador. En este caso, los nombres de unidad fuente son verdaderamente arbitrarios. Si, de todas maneras, quiere que el compilador encuentre y cargue el código fuente en el VFS automáticamente, sus nombres de unidad fuente necesitan estar estructurados de una manera que permita a un *import callback* localizarlos. Cuando se usa el compilador en la línea de comandos el import callback por defecto soporta solamente el código fuente que se carga desde el sistema de archivos host, lo cual significa que sus nombres de unidad fuente deben ser rutas. Algunos ambientes proveen custom callbacks que son más versátiles. Por ejemplo, el Remix IDE provee una que te permite importar archivos desde HTTP, IPFS y Swarm URLs o referir directamente a paquetes el el registro de NPM.

Para una descripción completa del sistema de archivos virtuales y la lógica de resolución de rutas usadas por el compilador véase *Resolución de Rutas*.

### 3.4.4 Comentarios

Comentarios de una sola línea (`//`) y comentarios de múltiples líneas (`/*...*/`) son posibles.

```
// Esto es un comentario de una sola línea.

/*
Esto es un
comentario de múltiples líneas.
*/
```

**Nota:** Un comentario de una sola línea finaliza por cualquier terminador de línea unicode (LF, VF, FF, CR, NEL, LS, o PS) en codificación UTF-8. El terminador es aún parte del código fuente luego del comentario, así que si no es un símbolo ASCII (se trata de NEL, LS y PS), conducirá a un parser error.

Adicionalmente, hay otro tipo de comentario llamado comentario NatSpec, el cual se detalla en la *guía de estilo*. Son escritos con una triple barra diagonal (`///`) o un bloque de asteriscos dobles (`/** ... */`) y deberían ser usados directamente sobre las declaraciones de funciones o sentencias.

## 3.5 Estructura de un Contrato

Los contratos en Solidity son similares a las clases en los lenguajes orientados a objetos. Cada contrato puede contener declaraciones de variables de estado, funciones, modificadores de funciones, eventos, errores, tipos struct y enum. Además, los contratos pueden heredar de otros contratos.

También hay tipos de contratos especiales llamados *libraries* e *interfaces*.

La sección sobre *contratos* contiene más detalles que esta sección, el cual sirve para proveer un rápido resumen.

### 3.5.1 Variables de Estado

Las variables de estado son variables cuyos valores se almacenan permanentemente en el almacenamiento del contrato.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract SimpleStorage {
    uint storedData; // Variable de estado
    // ...
}
```

Véase la sección de tipos para tipos válidos de variables de estado y la sección visibilidad-y-getters para posibles opciones de visibilidad. See the *Tipos* section for valid state variable types and *Visibility and Getters* for possible choices for visibility.

### 3.5.2 Funciones

Las funciones son las unidades de código ejecutables. Las funciones están definidas usualmente dentro de los contratos, pero tambíen pueden ser definidas fuera de los contratos.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract SimpleAuction {
    function bid() public payable { // Función
        // ...
    }
}

// Función auxiliar definida fuera de un contrato
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

Las-llamadas-a-funciones pueden suceder interna o externamente y tienen diferentes niveles de *visibilidad* hacia otros contratos. Las *funciones* aceptan *parámetros y retornan variables* para pasar parámetros y valores entre ellos.

### 3.5.3 Modificadores de Funciones

Los modificadores de funciones se pueden usar para modificar la semántica de las funciones de una forma declarativa (véase modificadores en la sección de contratos).

La sobrecarga, es decir, tener el mismo nombre del modificador con diferentes parámetros, no es posible.

Como las funciones, los modificadores se pueden *anular*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modificador
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public view onlySeller { // Uso del modificador
        // ...
    }
}
```

### 3.5.4 Eventos

Los eventos son interfaces convenientes con las facilidades de registro de la EVM.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Evento

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // Desencadenando un evento
    }
}
```

Véase eventos en la sección de contratos para información sobre cómo los eventos se declaran y pueden ser usados desde dentro de una dapp.

### 3.5.5 Errores

Los errores le permiten definir nombres y datos descriptivos para situaciones de falla. Los errores se pueden usar en *sentencias revert*. En comparaciñon a las descripciones de caradenas de caracteres, los errores son mucho más baratos y le permiten codificar datos adicionales. Puede usar NatSpec para describir el error al usuario.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Sin suficientes fondos para transferir.  Solicitado `requested`,
/// pero solo disponible `available`.
error NotEnoughFunds(uint requested, uint available);

contract Token {
    mapping(address => uint) balances;
    function transfer(address to, uint amount) public {
        uint balance = balances[msg.sender];
        if (balance < amount)
            revert NotEnoughFunds(amount, balance);
        balances[msg.sender] -= amount;
        balances[to] += amount;
        // ...
    }
}
```

Véase errores en la sección de contratos para más información.

### 3.5.6 Tipos de Structs

Los structs son tipos personalizados que pueden agrupar varias variables (véase *Structs* en la sección tipos).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

### 3.5.7 Tipos Enum

Los enums pueden ser usados para crear tipos personalizados con un conjunto finito de "valores constantes" (véase *Enums* en la sección de tipos).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

## 3.6 Tipos

Solidity es estáticamente escrito, lo que significa que se debe especificar el tipo de cada variable (de estado y local). Solidity proporciona varios tipos elementales que se pueden combinar para formar tipos complejos.

Adicionalmente, los tipos pueden interactuar entre sí en expresiones que contienen operadores. Para obtener una referencia rápida de los distintos operadores, consulte orden.

El concepto de valores «indefinidos (undefined)» o «nulos (null)» no existe en Solidity, pero las variables recién declaradas siempre tienen un *valor por defecto* que depende de su tipo. Para cualquier valor inesperado, debe usar la *función revert*, para revertir toda la transacción, o devolver una tupla con un segundo valor bool que indique el éxito de la operación.

### 3.6.1 Value Types

The following are called value types because their variables will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

#### Booleans

`bool`: The possible values are constants `true` and `false`.

Operators:

- `!` (logical negation)
- `&&` (logical conjunction, «and»)
- `||` (logical disjunction, «or»)
- `==` (equality)
- `!=` (inequality)

The operators `||` and `&&` apply the common short-circuiting rules. This means that in the expression `f(x) || g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

### Integers

`int` / `uint`: Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of `8` (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
- Shift operators: << (left shift), >> (right shift)
- Arithmetic operators: +, -, unary - (only for signed integers), *, /, % (modulo), ** (exponentiation)

For an integer type `X`, you can use `type(X).min` and `type(X).max` to access the minimum and maximum value representable by the type.

**Advertencia:** Integers in Solidity are restricted to a certain range. For example, with `uint32`, this is `0` up to `2**32 - 1`. There are two modes in which arithmetic is performed on these types: The «wrapping» or «unchecked» mode and the «checked» mode. By default, arithmetic is always «checked», meaning that if an operation's result falls outside the value range of the type, the call is reverted through a *failing assertion*. You can switch to «unchecked» mode using `unchecked { ... }`. More details can be found in the section about *unchecked*.

### Comparisons

The value of a comparison is the one obtained by comparing the integer value.

### Bit operations

Bit operations are performed on the two's complement representation of the number. This means that, for example `~int256(0) == int256(-1)`.

### Shifts

The result of a shift operation has the type of the left operand, truncating the result to match the type. The right operand must be of unsigned type, trying to shift by a signed type will produce a compilation error.

Shifts can be «simulated» using multiplication by powers of two in the following way. Note that the truncation to the type of the left operand is always performed at the end, but not mentioned explicitly.

- `x << y` is equivalent to the mathematical expression `x * 2**y`.
- `x >> y` is equivalent to the mathematical expression `x / 2**y`, rounded towards negative infinity.

**Advertencia:** Before version `0.5.0` a right shift `x >> y` for negative `x` was equivalent to the mathematical expression `x / 2**y` rounded towards zero, i.e., right shifts used rounding up (towards zero) instead of rounding down (towards negative infinity).

**Nota:** Overflow checks are never performed for shift operations as they are done for arithmetic operations. Instead, the result is always truncated.

### Addition, Subtraction and Multiplication

Addition, subtraction and multiplication have the usual semantics, with two different modes in regard to over- and underflow:

By default, all arithmetic is checked for under- or overflow, but this can be disabled using the *unchecked block*, resulting in wrapping arithmetic. More details can be found in that section.

The expression `-x` is equivalent to `(T(0) - x)` where `T` is the type of `x`. It can only be applied to signed types. The value of `-x` can be positive if `x` is negative. There is another caveat also resulting from two's complement representation:

If you have `int x = type(int).min;`, then `-x` does not fit the positive range. This means that `unchecked { assert(-x == x); }` works, and the expression `-x` when used in checked mode will result in a failing assertion.

### Division

Since the type of the result of an operation is always the type of one of the operands, division on integers always results in an integer. In Solidity, division rounds towards zero. This means that `int256(-5) / int256(2) == int256(-2)`.

Note that in contrast, division on *literals* results in fractional values of arbitrary precision.

**Nota:** Division by zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

**Nota:** The expression `type(int).min / (-1)` is the only case where division causes an overflow. In checked arithmetic mode, this will cause a failing assertion, while in wrapping mode, the value will be `type(int).min`.

### Modulo

The modulo operation `a % n` yields the remainder `r` after the division of the operand `a` by the operand `n`, where `q = int(a / n)` and `r = a - (n * q)`. This means that modulo results in the same sign as its left operand (or zero) and `a % n == -(-a % n)` holds for negative `a`:

- `int256(5) % int256(2) == int256(1)`
- `int256(5) % int256(-2) == int256(1)`
- `int256(-5) % int256(2) == int256(-1)`
- `int256(-5) % int256(-2) == int256(-1)`

**Nota:** Modulo with zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

### Exponentiation

Exponentiation is only available for unsigned types in the exponent. The resulting type of an exponentiation is always equal to the type of the base. Please take care that it is large enough to hold the result and prepare for potential assertion failures or wrapping behaviour.

**Nota:** In checked mode, exponentiation only uses the comparatively cheap `exp` opcode for small bases. For the cases of `x**3`, the expression `x*x*x` might be cheaper. In any case, gas cost tests and the use of the optimizer are advisable.

**Nota:** Note that `0**0` is defined by the EVM as 1.

### Fixed Point Numbers

**Advertencia:** Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from.

`fixed` / `ufixed`: Signed and unsigned fixed point number of various sizes. Keywords `ufixedMxN` and `fixedMxN`, where `M` represents the number of bits taken by the type and `N` represents how many decimal points are available. `M` must be divisible by 8 and goes from 8 to 256 bits. `N` must be between 0 and 80, inclusive. `ufixed` and `fixed` are aliases for `ufixed128x18` and `fixed128x18`, respectively.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Arithmetic operators: +, -, unary -, *, /, % (modulo)

**Nota:** The main difference between floating point (`float` and `double` in many languages, more precisely IEEE 754 numbers) and fixed point numbers is that the number of bits used for the integer and the fractional part (the part after the decimal dot) is flexible in the former, while it is strictly defined in the latter. Generally, in floating point almost the entire space is used to represent the number, while only a small number of bits define where the decimal point is.

### Address

The address type comes in two largely identical flavors:

- `address`: Holds a 20 byte value (size of an Ethereum address).
- `address payable`: Same as `address`, but with the additional members `transfer` and `send`.

The idea behind this distinction is that `address payable` is an address you can send Ether to, while you are not supposed to send Ether to a plain `address`, for example because it might be a smart contract that was not built to accept Ether.

Type conversions:

Implicit conversions from `address payable` to `address` are allowed, whereas conversions from `address` to `address payable` must be explicit via `payable(<address>)`.

Explicit conversions to and from `address` are allowed for `uint160`, integer literals, `bytes20` and contract types.

Only expressions of type `address` and contract-type can be converted to the type `address payable` via the explicit conversion `payable(...)`. For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a *receive* or a payable fallback function. Note that `payable(0)` is valid and is an exception to this rule.

---

**Nota:** If you need a variable of type `address` and plan to send Ether to it, then declare its type as `address payable` to make this requirement visible. Also, try to make this distinction or conversion as early as possible.

The distinction between `address` and `address payable` was introduced with version 0.5.0. Also starting from that version, contracts are not implicitly convertible to the `address` type, but can still be explicitly converted to `address` or to `address payable`, if they have a receive or payable fallback function.

---

Operators:

- <=, <, ==, !=, >= and >

**Advertencia:** If you convert a type that uses a larger byte size to an `address`, for example `bytes32`, then the `address` is truncated. To reduce conversion ambiguity, starting with version 0.4.24, the compiler will force you to make the truncation explicit in the conversion. Take for example the 32-byte value `0x111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFFCCCC`.

You can use `address(uint160(bytes20(b)))`, which results in `0x1111222233334444555566667777888899999aAaa`, or you can use `address(uint160(uint256(b)))`, which results in `0x777788889999AaAAbBbbCcccddDdeeeEfFFfCcCc`.

---

**Nota:** Mixed-case hexadecimal numbers conforming to EIP-55 are automatically treated as literals of the `address` type. See *Address Literals*.

---

### Members of Addresses

For a quick reference of all members of address, see *Miembros de tipos de direcciones*.

- `balance` and `transfer`

It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to a payable address using the `transfer` function:

```
address payable x = payable(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

The `transfer` function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The `transfer` function reverts on failure.

---

**Nota:** If `x` is a contract address, its code (more specifically: its *Receive Ether Function*, if present, or otherwise its *Fallback Function*, if present) will be executed together with the `transfer` call (this is a feature of the EVM and cannot be prevented). If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

---

- `send`

Send is the low-level counterpart of `transfer`. If the execution fails, the current contract will not stop with an exception, but `send` will return `false`.

**Advertencia:** There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: use a pattern where the recipient withdraws the money.

- `call`, `delegatecall` and `staticcall`

In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions `call`, `delegatecall` and `staticcall` are provided. They all take a single `bytes memory` parameter and return the success condition (as a `bool`) and the returned data (`bytes memory`). The functions `abi.encode`, `abi.encodePacked`, `abi.encodeWithSelector` and `abi.encodeWithSignature` can be used to encode structured data.

Example:

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

**Advertencia:** All these functions are low-level functions and should be used with care. Specifically, any unknown contract might be malicious and if you call it, you hand over control to that contract which could in turn call back into your contract, so be prepared for changes to your state variables when the call returns. The regular way to interact with other contracts is to call a function on a contract object (`x.f()`).

**Nota:** Previous versions of Solidity allowed these functions to receive arbitrary arguments and would also handle a first argument of type `bytes4` differently. These edge cases were removed in version 0.5.0.

It is possible to adjust the supplied gas with the `gas` modifier:

```
address(nameReg).call{gas: 1000000}(abi.encodeWithSignature("register(string)", "MyName
↪"));
```

Similarly, the supplied Ether value can be controlled too:

```
address(nameReg).call{value: 1 ether}(abi.encodeWithSignature("register(string)", "MyName
↪"));
```

Lastly, these modifiers can be combined. Their order does not matter:

```
address(nameReg).call{gas: 1000000, value: 1 ether}(abi.encodeWithSignature(
↪"register(string)", "MyName"));
```

In a similar way, the function `delegatecall` can be used: the difference is that only the code of the given address is used, all other aspects (storage, balance, ...) are taken from the current contract. The purpose of `delegatecall` is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for delegatecall to be used.

**Nota:** Prior to homestead, only a limited variant called `callcode` was available that did not provide access to the original `msg.sender` and `msg.value` values. This function was removed in version 0.5.0.

Since byzantium `staticcall` can be used as well. This is basically the same as `call`, but will revert if the called function modifies the state in any way.

All three functions `call`, `delegatecall` and `staticcall` are very low-level functions and should only be used as a *last resort* as they break the type-safety of Solidity.

The `gas` option is available on all three methods, while the `value` option is only available on `call`.

**Nota:** It is best to avoid relying on hardcoded gas values in your smart contract code, regardless of whether state is read from or written to, as this can have many pitfalls. Also, access to gas might change in the future.

- `code` and `codehash`

You can query the deployed code for any smart contract. Use `.code` to get the EVM bytecode as a `bytes memory`, which might be empty. Use `.codehash` to get the Keccak-256 hash of that code (as a `bytes32`). Note that `addr.codehash` is cheaper than using `keccak256(addr.code)`.

**Nota:** All contracts can be converted to `address` type, so it is possible to query the balance of the current contract using `address(this).balance`.

### Contract Types

Every *contract* defines its own type. You can implicitly convert contracts to contracts they inherit from. Contracts can be explicitly converted to and from the `address` type.

Explicit conversion to and from the `address payable` type is only possible if the contract type has a receive or payable fallback function. The conversion is still performed using `address(x)`. If the contract type does not have a receive or payable fallback function, the conversion to `address payable` can be done using `payable(address(x))`. You can find more information in the section about the *address type*.

**Nota:** Before version 0.5.0, contracts directly derived from the address type and there was no distinction between `address` and `address payable`.

If you declare a local variable of contract type (`MyContract c`), you can call functions on that contract. Take care to assign it from somewhere that is the same contract type.

You can also instantiate contracts (which means they are newly created). You can find more details in the *"Contracts via new"* section.

The data representation of a contract is identical to that of the `address` type and this type is also used in the *ABI*.

Contracts do not support any operators.

The members of contract types are the external functions of the contract including any state variables marked as `public`.

For a contract `C` you can use `type(C)` to access *type information* about the contract.

### Fixed-size byte arrays

The value types `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` hold a sequence of bytes from one to up to 32.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
- Shift operators: << (left shift), >> (right shift)
- Index access: If `x` is of type `bytesI`, then `x[k]` for `0 <= k < I` returns the `k` th byte (read-only).

The shifting operator works with unsigned integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a signed type will produce a compilation error.

Members:

- `.length` yields the fixed length of the byte array (read-only).

---

**Nota:** The type `bytes1[]` is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the `bytes` type instead.

---

---

**Nota:** Prior to version 0.8.0, `byte` used to be an alias for `bytes1`.

---

### Dynamically-sized byte array

**`bytes`:**
Dynamically-sized byte array, see *Arrays*. Not a value-type!

**`string`:**
Dynamically-sized UTF-8-encoded string, see *Arrays*. Not a value-type!

### Address Literals

Hexadecimal literals that pass the address checksum test, for example `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` are of `address` type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

---

**Nota:** The mixed-case address checksum format is defined in EIP-55.

---

### Rational and Integer Literals

Integer literals are formed from a sequence of digits in the range 0-9. They are interpreted as decimals. For example, `69` means sixty nine. Octal literals do not exist in Solidity and leading zeros are invalid.

Decimal fractional literals are formed by a `.` with at least one number after the decimal point. Examples include `.1` and `1.3` (but not `1.`).

Scientific notation in the form of `2e10` is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal `MeE` is equivalent to `M * 10**E`. Examples include `2e10`, `-2e10`, `2e-10`, `2.5e1`.

Underscores can be used to separate the digits of a numeric literal to aid readability. For example, decimal `123_000`, hexadecimal `0x2eff_abde`, scientific decimal notation `1_2e345_678` are all valid. Underscores are only allowed between two digits and only one consecutive underscore is allowed. There is no additional semantic meaning added to a number literal containing underscores, the underscores are ignored.

Number literal expressions retain arbitrary precision until they are converted to a non-literal type (i.e. by using them together with anything other than a number literal expression (like boolean literals) or by explicit conversion). This means that computations do not overflow and divisions do not truncate in number literal expressions.

For example, `(2**800 + 1) - 2**800` results in the constant `1` (of type `uint8`) although intermediate results would not even fit the machine word size. Furthermore, `.5 * 8` results in the integer `4` (although non-integers were used in between).

**Advertencia:** While most operators produce a literal expression when applied to literals, there are certain operators that do not follow this pattern:

- Ternary operator (`... ? ... : ...`),
- Array subscript (`<array>[<index>]`).

You might expect expressions like `255 + (true ? 1 : 0)` or `255 + [1, 2, 3][0]` to be equivalent to using the literal 256 directly, but in fact they are computed within the type `uint8` and can overflow.

Any operator that can be applied to integers can also be applied to number literal expressions as long as the operands are integers. If any of the two is fractional, bit operations are disallowed and exponentiation is disallowed if the exponent is fractional (because that might result in a non-rational number).

Shifts and exponentiation with literal numbers as left (or base) operand and integer types as the right (exponent) operand are always performed in the `uint256` (for non-negative literals) or `int256` (for a negative literals) type, regardless of the type of the right (exponent) operand.

**Advertencia:** Division on integer literals used to truncate in Solidity prior to version 0.4.0, but it now converts into a rational number, i.e. `5 / 2` is not equal to `2`, but to `2.5`.

---

**Nota:** Solidity has a number literal type for each rational number. Integer literals and rational number literals belong to number literal types. Moreover, all number literal expressions (i.e. the expressions that contain only number literals and operators) belong to number literal types. So the number literal expressions `1 + 2` and `2 + 1` both belong to the same number literal type for the rational number three.

---

**Nota:** Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions. Disregarding types, the value of the expression assigned to `b` below evaluates to an integer. Because `a` is of type `uint128`, the expression `2.5 + a` has to have a proper type, though. Since there is no common type for the type of `2.5` and `uint128`, the Solidity compiler does not accept this code.

---

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

### String Literals and Types

String literals are written with either double or single-quotes (`"foo"` or `'bar'`), and they can also be split into multiple consecutive parts (`"foo" "bar"` is equivalent to `"foobar"`) which can be helpful when dealing with long strings. They do not imply trailing zeroes as in C; `"foo"` represents three bytes, not four. As with integer literals, their type can vary, but they are implicitly convertible to `bytes1`, ..., `bytes32`, if they fit, to `bytes` and to `string`.

For example, with `bytes32 samevar = "stringliteral"` the string literal is interpreted in its raw byte form when assigned to a `bytes32` type.

String literals can only contain printable ASCII characters, which means the characters between and including 0x20 .. 0x7E.

Additionally, string literals also support the following escape characters:

- `\<newline>` (escapes an actual newline)
- `\\` (backslash)
- `\'` (single quote)
- `\"` (double quote)
- `\n` (newline)
- `\r` (carriage return)
- `\t` (tab)
- `\xNN` (hex escape, see below)
- `\uNNNN` (unicode escape, see below)

`\xNN` takes a hex value and inserts the appropriate byte, while `\uNNNN` takes a Unicode codepoint and inserts an UTF-8 sequence.

---

**Nota:** Until version 0.8.0 there were three additional escape sequences: `\b`, `\f` and `\v`. They are commonly available in other languages but rarely needed in practice. If you do need them, they can still be inserted via hexadecimal escapes, i.e. `\x08`, `\x0c` and `\x0b`, respectively, just as any other ASCII character.

---

The string in the following example has a length of ten bytes. It starts with a newline byte, followed by a double quote, a single quote a backslash character and then (without separator) the character sequence `abcdef`.

```
"\n\"\'\\abc\
def"
```

Any Unicode line terminator which is not a newline (i.e. LF, VF, FF, CR, NEL, LS, PS) is considered to terminate the string literal. Newline only terminates the string literal if it is not preceded by a `\`.

### Unicode Literals

While regular string literals can only contain ASCII, Unicode literals – prefixed with the keyword `unicode` – can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

```
string memory a = unicode"Hello ";
```

### Hexadecimal Literals

Hexadecimal literals are prefixed with the keyword `hex` and are enclosed in double or single-quotes (`hex"001122FF"`, `hex'0011_22_FF'`). Their content must be hexadecimal digits which can optionally use a single underscore as separator between byte boundaries. The value of the literal will be the binary representation of the hexadecimal sequence.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal: `hex"00112233" hex"44556677"` is equivalent to `hex"0011223344556677"`

Hexadecimal literals behave like *string literals* and have the same convertibility restrictions.

### Enums

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a *Panic error* otherwise. Enums require at least one member, and its default value when declared is the first member. Enums cannot have more than 256 members.

The data representation is the same as for enums in C: The options are represented by subsequent unsigned integer values starting from `0`.

Using `type(NameOfEnum).min` and `type(NameOfEnum).max` you can get the smallest and respectively largest value of the given enum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function getLargestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).max;
    }

    function getSmallestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).min;
    }
}
```

**Nota:** Enums can also be declared on the file level, outside of contract or library definitions.

### User-defined Value Types

A user-defined value type allows creating a zero cost abstraction over an elementary value type. This is similar to an alias, but with stricter type requirements.

A user-defined value type is defined using `type C is V`, where `C` is the name of the newly introduced type and `V` has to be a built-in value type (the «underlying type»). The function `C.wrap` is used to convert from the underlying type to the custom type. Similarly, the function `C.unwrap` is used to convert from the custom type to the underlying type.

The type `C` does not have any operators or attached member functions. In particular, even the operator == is not defined. Explicit and implicit conversions to and from other types are disallowed.

The data-representation of values of such types are inherited from the underlying type and the underlying type is also used in the ABI.

The following example illustrates a custom type `UFixed256x18` representing a decimal fixed point type with 18 decimals and a minimal library to do arithmetic operations on the type.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

// Represent a 18 decimal, 256 bit wide fixed point type using a user-defined value type.
type UFixed256x18 is uint256;

/// A minimal library to do fixed point operations on UFixed256x18.
library FixedMath {
    uint constant multiplier = 10**18;

    /// Adds two UFixed256x18 numbers. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function add(UFixed256x18 a, UFixed256x18 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) + UFixed256x18.unwrap(b));
    }
    /// Multiplies UFixed256x18 and uint256. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function mul(UFixed256x18 a, uint256 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
    }
    /// Take the floor of a UFixed256x18 number.
    /// @return the largest integer that does not exceed `a`.
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function floor(UFixed256x18 a) internal pure returns (uint256) {
        return UFixed256x18.unwrap(a) / multiplier;
    }
    /// Turns a uint256 into a UFixed256x18 of the same value.
    /// Reverts if the integer is too large.
    function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(a * multiplier);
    }
}
```

Notice how `UFixed256x18.wrap` and `FixedMath.toUFixed256x18` have the same signature but perform two very different operations: The `UFixed256x18.wrap` function returns a `UFixed256x18` that has the same data representation as the input, whereas `toUFixed256x18` returns a `UFixed256x18` that has the same numerical value.

### Function Types

Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - *internal* and *external* functions:

Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.

External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Function types are notated as follows:

```
function (<parameter types>) {internal|external} [pure|view|payable] [returns (<return␣
↪types>)]
```

In contrast to the parameter types, the return types cannot be empty - if the function type should not return anything, the whole `returns (<return types>)` part has to be omitted.

By default, function types are internal, so the `internal` keyword can be omitted. Note that this only applies to function types. Visibility has to be specified explicitly for functions defined in contracts, they do not have a default.

Conversions:

A function type `A` is implicitly convertible to a function type `B` if and only if their parameter types are identical, their return types are identical, their internal/external property is identical and the state mutability of `A` is more restrictive than the state mutability of `B`. In particular:

- `pure` functions can be converted to `view` and `non-payable` functions
- `view` functions can be converted to `non-payable` functions
- `payable` functions can be converted to `non-payable` functions

No other conversions between function types are possible.

The rule about `payable` and `non-payable` might be a little confusing, but in essence, if a function is `payable`, this means that it also accepts a payment of zero Ether, so it also is `non-payable`. On the other hand, a `non-payable` function will reject Ether sent to it, so `non-payable` functions cannot be converted to `payable` functions. To clarify, rejecting ether is more restrictive than not rejecting ether. This means you can override a payable function with a non-payable but not the other way around.

Additionally, When you define a `non-payable` function pointer, the compiler does not enforce that the pointed function will actually reject ether. Instead, it enforces that the function pointer is never used to send ether. Which makes it possible to assign a `payable` function pointer to a `non-payable` function pointer ensuring both types behave the same way, i.e, both cannot be used to send ether.

If a function type variable is not initialised, calling it results in a *Panic error*. The same happens if you call a function after using `delete` on it.

If external function types are used outside of the context of Solidity, they are treated as the `function` type, which encodes the address followed by the function identifier together in a single `bytes24` type.

Note that public functions of the current contract can be used both as an internal and as an external function. To use `f` as an internal function, just use `f`, if you want to use its external form, use `this.f`.

A function of an internal type can be assigned to a variable of an internal function type regardless of where it is defined. This includes private, internal and public functions of both contracts and libraries as well as free functions. External function types, on the other hand, are only compatible with public and external contract functions.

---

**Nota:** External functions with `calldata` parameters are incompatible with external function types with `calldata` parameters. They are compatible with the corresponding types with `memory` parameters instead. For example, there is no function that can be pointed at by a value of type `function (string calldata) external` while `function (string memory) external` can point at both `function f(string memory) external {}` and `function g(string calldata) external {}`. This is because for both locations the arguments are passed to the function in the same way. The caller cannot pass its calldata directly to an external function and always ABI-encodes the arguments into memory. Marking the parameters as `calldata` only affects the implementation of the external function and is meaningless in a function pointer on the caller's side.

---

Libraries are excluded because they require a `delegatecall` and use *a different ABI convention for their selectors*. Functions declared in interfaces do not have definitions so pointing at them does not make sense either.

Members:

External (or public) functions have the following members:

- `.address` returns the address of the contract of the function.
- `.selector` returns the *ABI function selector*

---

**Nota:** External (or public) functions used to have the additional members `.gas(uint)` and `.value(uint)`. These were deprecated in Solidity 0.6.2 and removed in Solidity 0.7.0. Instead use `{gas: ...}` and `{value: ...}` to specify the amount of gas or the amount of wei sent to a function, respectively. See *External Function Calls* for more information.

---

Example that shows how to use the members:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.4 <0.9.0;

contract Example {
    function f() public payable returns (bytes4) {
        assert(this.f.address == address(this));
        return this.f.selector;
    }

    function g() public {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        this.f{gas: 10, value: 800}();
    }
}
```

Example that shows how to use internal function types:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library ArrayUtils {
    // internal functions can be used in internal library functions because
    // they will be part of the same code context
    function map(uint[] memory self, function (uint) pure returns (uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }

    function reduce(
        uint[] memory self,
        function (uint, uint) pure returns (uint) f
    )
        internal
        pure
        returns (uint r)
    {
        r = self[0];
        for (uint i = 1; i < self.length; i++) {
            r = f(r, self[i]);
        }
    }

    function range(uint length) internal pure returns (uint[] memory r) {
        r = new uint[](length);
        for (uint i = 0; i < r.length; i++) {
            r[i] = i;
        }
    }
}


contract Pyramid {
    using ArrayUtils for *;

    function pyramid(uint l) public pure returns (uint) {
        return ArrayUtils.range(l).map(square).reduce(sum);
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function square(uint x) internal pure returns (uint) {
        return x * x;
    }

    function sum(uint x, uint y) internal pure returns (uint) {
        return x + y;
    }
}
```

Another example that uses external function types:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;


contract Oracle {
    struct Request {
        bytes data;
        function(uint) external callback;
    }

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // Here goes the check that the reply comes from a trusted source
        requests[requestID].callback(response);
    }
}


contract OracleUser {
    Oracle constant private ORACLE_CONST =␣
↪Oracle(address(0x00000000219ab540356cBB839Cbe05303d7705Fa)); // known contract
    uint private exchangeRate;

    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(
            msg.sender == address(ORACLE_CONST),
            "Only oracle can call this."
        );
        exchangeRate = response;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }
}
```

**Nota:** Lambda or inline functions are planned but not yet supported.

### 3.6.2 Tipos de Referencia

El valor de los «tipos de referencia» (reference type) puede ser modificado a través de distintos nombres. Esto contrasta con los «tipos de valor» (value type) donde obtienes una copia independiente siempre que se usan estos tipos de valor. Por esta razón, los tipos de referencia tienen que ser manejados con más cuidado que los tipos de valor. Actualmente, existen los siguientes tipos de referencia: structs, arrays y mappings. Si usas un tipo de referencia, siempre tienes que indicar de forma explícita el lugar en el que está almacenado el tipo: `memory` (cuyo tiempo de almacenamiento está limitado a una llamada externa que se hace a una función), `storage` (el lugar en el cual se alojan las variables de estado, y cuyo tiempo de almacenamiento está limitado a la propia vida del contrato) or `calldata` (un lugar especial para el guardado de datos que contiene los argumentos de una función).

Una asignación o una conversión de un tipo, el cual implique un cambio en la localización de los datos, siempre incurrirá en una operación de copia automática. Por otro lado, las asignaciones hechas sobre la misma localización de los datos, tan solo se copiarán en algunos casos de tipos almacenados en el storage.

#### Data location

Every reference type has an additional annotation, the «data location», about where it is stored. There are three data locations: `memory`, `storage` and `calldata`. Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory.

**Nota:** If you can, try to use `calldata` as data location because it will avoid copies and also makes sure that the data cannot be modified. Arrays and structs with `calldata` data location can also be returned from functions, but it is not possible to allocate such types.

**Nota:** Prior to version 0.6.9 data location for reference-type arguments was limited to `calldata` in external functions, `memory` in public functions and either `memory` or `storage` in internal and private ones. Now `memory` and `calldata` are allowed in all functions regardless of their visibility.

**Nota:** Prior to version 0.5.0 the data location could be omitted, and would default to different locations depending on the kind of variable, function type, etc., but all complex types must now give an explicit data location.

### Data location and assignment behaviour

Data locations are not only relevant for persistency of data, but also for the semantics of assignments:

- Assignments between `storage` and `memory` (or from `calldata`) always create an independent copy.
- Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
- Assignments from `storage` to a **local** storage variable also only assign a reference.
- All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        uint[] storage y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.pop(); // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // Similarly, "delete y" is not valid, as assignments to local variables
        // referencing storage objects can only be made from existing storage objects.
        // It would "reset" the pointer, but there is no sensible location it could␣
↪point to.
        // For more details see the documentation of the "delete" operator.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

### Arrays

Arrays can have a compile-time fixed size, or they can have a dynamic size.

The type of an array of fixed size `k` and element type `T` is written as `T[k]`, and an array of dynamic size as `T[]`.

For example, an array of 5 dynamic arrays of `uint` is written as `uint[][5]`. The notation is reversed compared to some other languages. In Solidity, `X[3]` is always an array containing three elements of type `X`, even if `X` is itself an array. This is not the case in other languages such as C.

Indices are zero-based, and access is in the opposite direction of the declaration.

For example, if you have a variable `uint[][5] memory x`, you access the seventh `uint` in the third dynamic array using `x[2][6]`, and to access the third dynamic array, use `x[2]`. Again, if you have an array `T[5] a` for a type `T` that can also be an array, then `a[2]` always has type `T`.

Array elements can be of any type, including mapping or struct. The general restrictions for types apply, in that mappings can only be stored in the `storage` data location and publicly-visible functions need parameters that are *ABI types*.

It is possible to mark state variable arrays `public` and have Solidity create a *getter*. The numeric index becomes a required parameter for the getter.

Accessing an array past its end causes a failing assertion. Methods `.push()` and `.push(value)` can be used to append a new element at the end of a dynamically-sized array, where `.push()` appends a zero-initialized element and returns a reference to it.

---

**Nota:** Dynamically-sized arrays can only be resized in storage. In memory, such arrays can be of arbitrary size but the size cannot be changed once an array is allocated.

---

### `bytes` and `string` as Arrays

Variables of type `bytes` and `string` are special arrays. The `bytes` type is similar to `bytes1[]`, but it is packed tightly in calldata and memory. `string` is equal to `bytes` but does not allow length or index access.

Solidity does not have string manipulation functions, but there are third-party string libraries. You can also compare two strings by their keccak256-hash using `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` and concatenate two strings using `string.concat(s1, s2)`.

You should use `bytes` over `bytes1[]` because it is cheaper, since using `bytes1[]` in `memory` adds 31 padding bytes between the elements. Note that in `storage`, the padding is absent due to tight packing, see *bytes and string*. As a general rule, use `bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of the value types `bytes1` to `bytes32` because they are much cheaper.

---

**Nota:** If you want to access the byte-representation of a string `s`, use `bytes(s).length` / `bytes(s)[7] = 'x';`. Keep in mind that you are accessing the low-level bytes of the UTF-8 representation, and not the individual characters.

---

### The functions `bytes.concat` and `string.concat`

You can concatenate an arbitrary number of `string` values using `string.concat`. The function returns a single `string memory` array that contains the contents of the arguments without padding. If you want to use parameters of other types that are not implicitly convertible to `string`, you need to convert them to `string` first.

Analogously, the `bytes.concat` function can concatenate an arbitrary number of `bytes` or `bytes1 ... bytes32` values. The function returns a single `bytes memory` array that contains the contents of the arguments without padding. If you want to use string parameters or other types that are not implicitly convertible to `bytes`, you need to convert them to `bytes` or `bytes1`/.../`bytes32` first.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.12;

contract C {
    string s = "Storage";
    function f(bytes calldata bc, string memory sm, bytes16 b) public view {
        string memory concatString = string.concat(s, string(bc), "Literal", sm);
        assert((bytes(s).length + bc.length + 7 + bytes(sm).length) ==␣
↪bytes(concatString).length);

        bytes memory concatBytes = bytes.concat(bytes(s), bc, bc[:2], "Literal",␣
↪bytes(sm), b);
        assert((bytes(s).length + bc.length + 2 + 7 + bytes(sm).length + b.length) ==␣
↪concatBytes.length);
    }
}
```

If you call `string.concat` or `bytes.concat` without arguments they return an empty array.

### Allocating Memory Arrays

Memory arrays with dynamic length can be created using the `new` operator. As opposed to storage arrays, it is **not** possible to resize memory arrays (e.g. the `.push` member functions are not available). You either have to calculate the required size in advance or create a new memory array and copy every element.

As all variables in Solidity, the elements of newly allocated arrays are always initialized with the *default value*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

### Array Literals

An array literal is a comma-separated list of one or more expressions, enclosed in square brackets (`[...]`). For example `[1, a, f(3)]`. The type of the array literal is determined as follows:

It is always a statically-sized memory array whose length is the number of expressions.

The base type of the array is the type of the first expression on the list such that all other expressions can be implicitly converted to it. It is a type error if this is not possible.

It is not enough that there is a type all the elements can be converted to. One of the elements has to be of that type.

In the example below, the type of `[1, 2, 3]` is `uint8[3] memory`, because the type of each of these constants is `uint8`. If you want the result to be a `uint[3] memory` type, you need to convert the first element to `uint`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] memory) public pure {
        // ...
    }
}
```

The array literal `[1, -1]` is invalid because the type of the first expression is `uint8` while the type of the second is `int8` and they cannot be implicitly converted to each other. To make it work, you can use `[int8(1), -1]`, for example.

Since fixed-size memory arrays of different type cannot be converted into each other (even if the base types can), you always have to specify a common base type explicitly if you want to use two-dimensional array literals:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure returns (uint24[2][4] memory) {
        uint24[2][4] memory x = [[uint24(0x1), 1], [0xffffff, 2], [uint24(0xff), 3],␣
↪[uint24(0xffff), 4]];
        // The following does not work, because some of the inner arrays are not of the␣
↪right type.
        // uint[2][4] memory x = [[0x1, 1], [0xffffff, 2], [0xff, 3], [0xffff, 4]];
        return x;
    }
}
```

Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

// This will not compile.
contract C {
    function f() public {
        // The next line creates a type error because uint[3] memory
```

(continué en la próxima página)

(proviene de la página anterior)

```
        // cannot be converted to uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

It is planned to remove this restriction in the future, but it creates some complications because of how arrays are passed in the ABI.

If you want to initialize dynamically-sized arrays, you have to assign the individual elements:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        uint[] memory x = new uint[](3);
        x[0] = 1;
        x[1] = 3;
        x[2] = 4;
    }
}
```

### Array Members

**length:**
Arrays have a `length` member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

**push():**
Dynamic storage arrays and `bytes` (not `string`) have a member function called `push()` that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like `x.push().t = 2` or `x.push() = b`.

**push(x):**
Dynamic storage arrays and `bytes` (not `string`) have a member function called `push(x)` that you can use to append a given element at the end of the array. The function returns nothing.

**pop():**
Dynamic storage arrays and `bytes` (not `string`) have a member function called `pop()` that you can use to remove an element from the end of the array. This also implicitly calls *delete* on the removed element. The function returns nothing.

---

**Nota:** Increasing the length of a storage array by calling `push()` has constant gas costs because storage is zero-initialised, while decreasing the length by calling `pop()` has a cost that depends on the «size» of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling *delete* on them.

---

**Nota:** To use arrays of arrays in external (instead of public) functions, you need to activate ABI coder v2.

---

**Nota:** In EVM versions before Byzantium, it was not possible to access dynamic arrays return from function calls. If

you call functions that return dynamic arrays, make sure to use an EVM that is set to Byzantium mode.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract ArrayContract {
    uint[2**20] aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of length two).
    // In Solidity, T[k] and T[] are always arrays with elements of type T,
    // even if T itself is an array.
    // Because of that, bool[2][] is a dynamic array of elements
    // that are bool[2]. This is different from other languages, like C.
    // Data location for all state variables is storage.
    bool[2][] pairsOfFlags;

    // newPairs is stored in memory - the only possibility
    // for public contract function arguments
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // assignment to a storage array performs a copy of ``newPairs`` and
        // replaces the complete array ``pairsOfFlags``.
        pairsOfFlags = newPairs;
    }

    struct StructType {
        uint[] contents;
        uint moreInfo;
    }
    StructType s;

    function f(uint[] memory c) public {
        // stores a reference to ``s`` in ``g``
        StructType storage g = s;
        // also changes ``s.moreInfo``.
        g.moreInfo = 2;
        // assigns a copy because ``g.contents``
        // is not a local variable, but a member of
        // a local variable.
        g.contents = c;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // access to a non-existing index will throw an exception
        pairsOfFlags[index][0] = flagA;
        pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // using push and pop is the only way to change the
        // length of an array
        if (newSize < pairsOfFlags.length) {
            while (pairsOfFlags.length > newSize)
```

(continué en la próxima página)

(proviene de la página anterior)

```
                pairsOfFlags.pop();
        } else if (newSize > pairsOfFlags.length) {
            while (pairsOfFlags.length < newSize)
                pairsOfFlags.push();
        }
    }

    function clear() public {
        // these clear the arrays completely
        delete pairsOfFlags;
        delete aLotOfIntegers;
        // identical effect here
        pairsOfFlags = new bool[2][](0);
    }

    bytes byteData;

    function byteArrays(bytes memory data) public {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        byteData = data;
        for (uint i = 0; i < 7; i++)
            byteData.push();
        byteData[3] = 0x08;
        delete byteData[2];
    }

    function addFlag(bool[2] memory flag) public returns (uint) {
        pairsOfFlags.push(flag);
        return pairsOfFlags.length;
    }

    function createMemoryArray(uint size) public pure returns (bytes memory) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        // Inline arrays are always statically-sized and if you only
        // use literals, you have to provide at least one type.
        arrayOfPairs[0] = [uint(1), 2];

        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = bytes1(uint8(i));
        return b;
    }
}
```

### Dangling References to Storage Array Elements

When working with storage arrays, you need to take care to avoid dangling references. A dangling reference is a reference that points to something that no longer exists or has been moved without updating the reference. A dangling reference can for example occur, if you store a reference to an array element in a local variable and then `.pop()` from the containing array:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[][] s;

    function f() public {
        // Stores a pointer to the last array element of s.
        uint[] storage ptr = s[s.length - 1];
        // Removes the last array element of s.
        s.pop();
        // Writes to the array element that is no longer within the array.
        ptr.push(0x42);
        // Adding a new element to ``s`` now will not add an empty array, but
        // will result in an array of length 1 with ``0x42`` as element.
        s.push();
        assert(s[s.length - 1][0] == 0x42);
    }
}
```

The write in `ptr.push(0x42)` will **not** revert, despite the fact that `ptr` no longer refers to a valid element of `s`. Since the compiler assumes that unused storage is always zeroed, a subsequent `s.push()` will not explicitly write zeroes to storage, so the last element of `s` after that `push()` will have length 1 and contain `0x42` as its first element.

Note that Solidity does not allow to declare references to value types in storage. These kinds of explicit dangling references are restricted to nested reference types. However, dangling references can also occur temporarily when using complex expressions in tuple assignments:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[] s;
    uint[] t;
    constructor() {
        // Push some initial values to the storage arrays.
        s.push(0x07);
        t.push(0x03);
    }

    function g() internal returns (uint[] storage) {
        s.pop();
        return t;
    }

    function f() public returns (uint[] memory) {
        // The following will first evaluate ``s.push()`` to a reference to a new element
```

(continué en la próxima página)

(proviene de la página anterior)

```
        // at index 1. Afterwards, the call to ``g`` pops this new element, resulting in
        // the left-most tuple element to become a dangling reference. The assignment␣
↪still
        // takes place and will write outside the data area of ``s``.
        (s.push(), g()[0]) = (0x42, 0x17);
        // A subsequent push to ``s`` will reveal the value written by the previous
        // statement, i.e. the last element of ``s`` at the end of this function will have
        // the value ``0x42``.
        s.push();
        return s;
    }
}
```

It is always safer to only assign to storage once per statement and to avoid complex expressions on the left-hand-side of an assignment.

You need to take particular care when dealing with references to elements of `bytes` arrays, since a `.push()` on a bytes array may switch *from short to long layout in storage*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

// This will report a warning
contract C {
    bytes x = "0123456789012345678901234567890123456789";

    function test() external returns(uint) {
        (x.push(), x.push()) = (0x01, 0x02);
        return x.length;
    }
}
```

Here, when the first `x.push()` is evaluated, `x` is still stored in short layout, thereby `x.push()` returns a reference to an element in the first storage slot of `x`. However, the second `x.push()` switches the bytes array to large layout. Now the element that `x.push()` referred to is in the data area of the array while the reference still points at its original location, which is now a part of the length field and the assignment will effectively garble the length of `x`. To be safe, only enlarge bytes arrays by at most one element during a single assignment and do not simultaneously index-access the array in the same statement.

While the above describes the behaviour of dangling storage references in the current version of the compiler, any code with dangling references should be considered to have *undefined behaviour*. In particular, this means that any future version of the compiler may change the behaviour of code that involves dangling references.

Be sure to avoid dangling references in your code!

### Array Slices

Array slices are a view on a contiguous portion of an array. They are written as `x[start:end]`, where `start` and `end` are expressions resulting in a uint256 type (or implicitly convertible to it). The first element of the slice is `x[start]` and the last element is `x[end - 1]`.

If `start` is greater than `end` or if `end` is greater than the length of the array, an exception is thrown.

Both `start` and `end` are optional: `start` defaults to `0` and `end` defaults to the length of the array.

Array slices do not have any members. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice.

Array slices do not have a type name which means no variable can have an array slices as type, they only exist in intermediate expressions.

---

**Nota:** As of now, array slices are only implemented for calldata arrays.

---

Array slices are useful to ABI-decode secondary data passed in function parameters:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.5 <0.9.0;
contract Proxy {
    /// @dev Address of the client contract managed by proxy i.e., this contract
    address client;

    constructor(address client_) {
        client = client_;
    }

    /// Forward call to "setOwner(address)" that is implemented by client
    /// after doing basic validation on the address argument.
    function forward(bytes calldata payload) external {
        bytes4 sig = bytes4(payload[:4]);
        // Due to truncating behaviour, bytes4(payload) performs identically.
        // bytes4 sig = bytes4(payload);
        if (sig == bytes4(keccak256("setOwner(address)"))) {
            address owner = abi.decode(payload[4:], (address));
            require(owner != address(0), "Address of owner cannot be zero.");
        }
        (bool status,) = client.delegatecall(payload);
        require(status, "Forwarded call failed.");
    }
}
```

### Structs

Solidity provides a way to define new types in the form of structs, which is shown in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// Defines a new type with two fields.
// Declaring a struct outside of a contract allows
// it to be shared by multiple contracts.
// Here, this is not really needed.
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
    // Structs can also be defined inside contracts, which makes them
    // visible only there and in derived contracts.
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping(uint => Funder) funders;
    }

    uint numCampaigns;
    mapping(uint => Campaign) campaigns;

    function newCampaign(address payable beneficiary, uint goal) public returns (uint␣
↪campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // We cannot use "campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)"
        // because the right hand side creates a memory-struct "Campaign" that contains␣
↪a mapping.
        Campaign storage c = campaigns[campaignID];
        c.beneficiary = beneficiary;
        c.fundingGoal = goal;
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // Creates a new temporary memory struct, initialised with the given values
        // and copies it over to storage.
        // Note that you can also use Funder(msg.sender, msg.value) to initialise.
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
```

(continué en la próxima página)

(proviene de la página anterior)

```
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays.

It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite.

Note how in all the functions, a struct type is assigned to a local variable with data location `storage`. This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

Of course, you can also directly access the members of the struct without assigning it to a local variable, as in `campaigns[campaignID].amount = 0`.

---

**Nota:** Until Solidity 0.7.0, memory-structs containing members of storage-only types (e.g. mappings) were allowed and assignments like `campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)` in the example above would work and just silently skip those members.

---

### 3.6.3 Tipos Mapping

Los tipos mapping usan la sintaxis `mapping(KeyType KeyName? => ValueType ValueName?)` y las variables del tipo mapping son declaradas usando la sintaxis `mapping(KeyType KeyName? => ValueType ValueName?) VariableName`. Donde `KeyType` puede ser cualquier tipo estándar, `bytes`, `string`, cualquier contrato o un tipo enum. Sin embargo, tipos definidos por el propio usuario o tipos complejos, como mappings, structs o arrays no están permitidos para `KeyType`. Por otro lado, para `ValueType`, puede ser cualquier tipo, incluyendo mappings, arrays o structs. `KeyName` y `ValueName` son opcionales (por lo que `mapping(KeyType => ValueType)` también funciona) y pueden ser cualquier identificador válido que no sea un tipo.

Puede pensar en los mappings como tablas hash, las cuales se inicializan virtualmente, asumiendo que ya existen todas las posibles claves, y donde se mapea cada clave a un valor cuya representación byte está formada por todo ceros, equivalente al *valor por defecto* de un tipo. Aunque la similitud termina aquí, dado que las claves no se almacenan realmente en el mapping, sino tan solo su hash `keccak256`, el cual se usa para buscar su valor.

Por esto mismo, los mappings no tienen una longitud (length), ni tampoco son exactamente una tabla la cual relaciona claves con valores. Por tanto, no puede eliminarse información respecto a cada clave asociada. (ver *Limpieza de Mappings*).

Los mappings solo pueden almacenar información en el `storage`, por lo que solo están permitidos para variables de estado, como tipos que referencian al storage en funciones, o como parámetros de funciones de librerías. Pero no se pueden usar como parámetros o como retorno de funciones públicamente visibles de un contrato. Y estas mismas restricciones también se aplican para arrays y structs los cuales contengan mappings.

Puede marcar variables de estado de tipo mapping como `public` y así Solidity creará una *función get (getter)* por ti. El `KeyType` se convertirá en un parámetro de la función get. Si `ValueType` es un tipo de valor o un struct, la función get

retornará `ValueType`. Si `ValueType` es un array o un mapping, la función get tendrá un parámetro por cada `KeyType`, recursivamente.

En el siguiente ejemplo, el contrato `MappingExample` define un mapping público `balances`, donde el tipo de clave es un tipo `address`, y el tipo de valor es un tipo `uint`, mapeando así una dirección Ethereum a un número entero sin signo. Como `uint` es un tipo de valor, la función get devolverá un valor que encajará con este tipo, tal como puede ver aquí en el contrato `MappingUser`, el cual retorna el valor para la address especificada.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

El siguiente ejemplo es una versión simplificada de un ERC20 token.

`_allowances` es un ejemplo de un tipo de asignación dentro de otro tipo de asignación.

En el siguiente ejemplo, los campos opcionales `KeyName` y `ValueName` son proporcionados para el mapeo. No afecta a ninguna funcionalidad del contrato ni al bytecode, sólo establece el campo `name` para las entradas y salidas en la ABI para el getter de la asignación.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.18;

contract MappingExampleWithNames {
    mapping(address user => uint balance) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}
```

El siguiente ejemplo utiliza `_allowances` para registrar la cantidad que otra persona puede retirar de su cuenta.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract MappingExample {

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public␣
↪returns (bool) {
        require(_allowances[sender][msg.sender] >= amount, "ERC20: Allowance not high␣
↪enough.");
        _allowances[sender][msg.sender] -= amount;
        _transfer(sender, recipient, amount);
        return true;
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");
        require(_balances[sender] >= amount, "ERC20: Not enough funds.");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

### Mappings iterables

No se puede iterar un mapping. Es decir, no se puede numerar las claves. Sin embargo, sí es posible implementar una estructura de datos por encima para poder iterar esta estructura contenedora. Por ejemplo, el siguiente código implementa una librería `IterableMapping` donde luego el contrato `User` añade información, y la función `sum` puede iterar sobre la suma de todos los valores.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

struct itmap {
    mapping(uint => IndexValue) data;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    KeyFlag[] keys;
    uint size;
}

type Iterator is uint;

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool␣
↪replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            keyIndex = self.keys.length;
            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

    function remove(itmap storage self, uint key) internal returns (bool success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterateStart(itmap storage self) internal view returns (Iterator) {
        return iteratorSkipDeleted(self, 0);
    }

    function iterateValid(itmap storage self, Iterator iterator) internal view returns␣
↪(bool) {
        return Iterator.unwrap(iterator) < self.keys.length;
    }

    function iterateNext(itmap storage self, Iterator iterator) internal view returns␣
↪(Iterator) {
        return iteratorSkipDeleted(self, Iterator.unwrap(iterator) + 1);
    }

    function iterateGet(itmap storage self, Iterator iterator) internal view returns␣
↪(uint key, uint value) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        uint keyIndex = Iterator.unwrap(iterator);
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
    }

    function iteratorSkipDeleted(itmap storage self, uint keyIndex) private view returns␣
↪(Iterator) {
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return Iterator.wrap(keyIndex);
    }
}

// Modo de funcionamiento
contract User {
    // Un simple struct mantiene nuestros datos
    itmap data;
    // Se aplican las funciones de la librería para este tipo de datos
    using IterableMapping for itmap;

    // Añadimos algo
    function insert(uint k, uint v) public returns (uint size) {
        // Esto llama a IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // Todavía podemos acceder a las partes del struct
        // pero debemos hacerlo con cuidado para no desorganizar la lógica
        return data.size;
    }

    // Se calcula la suma de todos los datos almacenados
    function sum() public view returns (uint s) {
        for (
            Iterator i = data.iterateStart();
            data.iterateValid(i);
            i = data.iterateNext(i)
        ) {
            (, uint value) = data.iterateGet(i);
            s += value;
        }
    }
}
```

### 3.6.4 Operators

Arithmetic and bit operators can be applied even if the two operands do not have the same type. For example, you can compute `y = x + z`, where `x` is a `uint8` and `z` has the type `uint32`. In these cases, the following mechanism will be used to determine the type in which the operation is computed (this is important in case of overflow) and the type of the operator's result:

1. If the type of the right operand can be implicitly converted to the type of the left operand, use the type of the left operand,
2. if the type of the left operand can be implicitly converted to the type of the right operand, use the type of the right operand,
3. otherwise, the operation is not allowed.

In case one of the operands is a *literal number* it is first converted to its «mobile type», which is the smallest type that can hold the value (unsigned types of the same bit-width are considered «smaller» than the signed types). If both are literal numbers, the operation is computed with effectively unlimited precision in that the expression is evaluated to whatever precision is necessary so that none is lost when the result is used with a non-literal type.

The operator's result type is the same as the type the operation is performed in, except for comparison operators where the result is always `bool`.

The operators ** (exponentiation), << and >> use the type of the left operand for the operation and the result.

#### Ternary Operator

The ternary operator is used in expressions of the form `<expression> ? <trueExpression> : <falseExpression>`. It evaluates one of the latter two given expressions depending upon the result of the evaluation of the main `<expression>`. If `<expression>` evaluates to `true`, then `<trueExpression>` will be evaluated, otherwise `<falseExpression>` is evaluated.

The result of the ternary operator does not have a rational number type, even if all of its operands are rational number literals. The result type is determined from the types of the two operands in the same way as above, converting to their mobile type first if required.

As a consequence, `255 + (true ? 1 : 0)` will revert due to arithmetic overflow. The reason is that `(true ? 1 : 0)` is of `uint8` type, which forces the addition to be performed in `uint8` as well, and 256 exceeds the range allowed for this type.

Another consequence is that an expression like `1.5 + 1.5` is valid but `1.5 + (true ? 1.5 : 2.5)` is not. This is because the former is a rational expression evaluated in unlimited precision and only its final value matters. The latter involves a conversion of a fractional rational number to an integer, which is currently disallowed.

#### Compound and Increment/Decrement Operators

If `a` is an LValue (i.e. a variable or something that can be assigned to), the following operators are available as shorthands:

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`, `<<=` and `>>=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1 / a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change.

### delete

`delete a` assigns the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`, but it can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value. `delete a[x]` deletes the item at index `x` of the array and leaves all other elements and the length of the array untouched. This especially means that it leaves a gap in the array. If you plan to remove items, a *mapping* is probably a better choice.

For structs, it assigns a struct with all members reset. In other words, the value of `a` after `delete a` is the same as if `a` would be declared without assignment, with the following caveat:

`delete` has no effect on mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings. However, individual keys and what they map to can be deleted: If `a` is a mapping, then `delete a[x]` will delete the value stored at `x`.

It is important to note that `delete a` really behaves like an assignment to `a`, i.e. it stores a new object in `a`. This distinction is visible when `a` is reference variable: It will only reset `a` itself, not the value it referred to previously.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[] is a␣
↪complex object, also
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as assignments to local variables
        // referencing storage objects can only be made from existing storage objects.
        assert(y.length == 0);
    }
}
```

### Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

| Precedence | Description | Operator |
|---|---|---|
| *1* | Postfix increment and decrement | ++, -- |
| | New expression | new <typename> |
| | Array subscripting | <array>[<index>] |
| | Member access | <object>.<member> |
| | Function-like call | <func>(<args...>) |
| | Parentheses | (<statement>) |
| *2* | Prefix increment and decrement | ++, -- |
| | Unary minus | - |
| | Unary operations | delete |
| | Logical NOT | ! |
| | Bitwise NOT | ~ |
| *3* | Exponentiation | ** |
| *4* | Multiplication, division and modulo | *, /, % |
| *5* | Addition and subtraction | +, - |
| *6* | Bitwise shift operators | <<, >> |
| *7* | Bitwise AND | & |
| *8* | Bitwise XOR | ^ |
| *9* | Bitwise OR | \| |
| *10* | Inequality operators | <, >, <=, >= |
| *11* | Equality operators | ==, != |
| *12* | Logical AND | && |
| *13* | Logical OR | \|\| |
| *14* | Ternary operator | <conditional> ? <if-true> : <if-false> |
| | Assignment operators | =, \|=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %= |
| *15* | Comma operator | , |

## 3.6.5 Conversions between Elementary Types

### Implicit Conversions

An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators. In general, an implicit conversion between value-types is possible if it makes sense semantically and no information is lost.

For example, `uint8` is convertible to `uint16` and `int128` to `int256`, but `int8` is not convertible to `uint256`, because `uint256` cannot hold values such as `-1`.

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands to the type of the other (the same is true for assignments). This means that operations are always performed in the type of one of the operands.

For more details about which implicit conversions are possible, please consult the sections about the types themselves.

In the example below, `y` and `z`, the operands of the addition, do not have the same type, but `uint8` can be implicitly converted to `uint16` and not vice-versa. Because of that, `y` is converted to the type of `z` before the addition is performed in the `uint16` type. The resulting type of the expression `y + z` is `uint16`. Because it is assigned to a variable of type `uint32` another implicit conversion is performed after the addition.

```
uint8 y;
uint16 z;
uint32 x = y + z;
```

### Explicit Conversions

If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behaviour and allows you to bypass some security features of the compiler, so be sure to test that the result is what you want and expect!

Take the following example that converts a negative `int` to a `uint`:

```
int  y = -3;
uint x = uint(y);
```

At the end of this code snippet, `x` will have the value `0xfffff..fd` (64 hex characters), which is -3 in the two's complement representation of 256 bits.

If an integer is explicitly converted to a smaller type, higher-order bits are cut off:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end). The result of the conversion will compare equal to the original integer:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

Fixed-size bytes types behave differently during conversions. They can be thought of as sequences of individual bytes and converting to a smaller type will cut off the sequence:

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b will be 0x12
```

If a fixed-size bytes type is explicitly converted to a larger type, it is padded on the right. Accessing the byte at a fixed index will result in the same value before and after the conversion (if the index is still in range):

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b will be 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

Since integers and fixed-size byte arrays behave differently when truncating or padding, explicit conversions between integers and fixed-size byte arrays are only allowed, if both have the same size. If you want to convert between integers and fixed-size byte arrays of different size, you have to use intermediate conversions that make the desired truncation and padding rules explicit:

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b will be 0x00001234
uint32 c = uint32(bytes4(a)); // c will be 0x12340000
uint8 d = uint8(uint16(a)); // d will be 0x34
uint8 e = uint8(bytes1(a)); // e will be 0x12
```

`bytes` arrays and `bytes` calldata slices can be converted explicitly to fixed bytes types (`bytes1`/.../`bytes32`). In case the array is longer than the target fixed bytes type, truncation at the end will happen. If the array is shorter than the target type, it will be padded with zeros at the end.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;

contract C {
    bytes s = "abcdefgh";
    function f(bytes calldata c, bytes memory m) public view returns (bytes16, bytes3) {
        require(c.length == 16, "");
        bytes16 b = bytes16(m);  // if length of m is greater than 16, truncation will␣
↪happen
        b = bytes16(s);  // padded on the right, so result is "abcdefgh\0\0\0\0\0\0\0\0"
        bytes3 b1 = bytes3(s); // truncated, b1 equals to "abc"
        b = bytes16(c[:8]);  // also padded with zeros
        return (b, b1);
    }
}
```

### 3.6.6 Conversions between Literals and Elementary Types

#### Integer Types

Decimal and hexadecimal number literals can be implicitly converted to any integer type that is large enough to represent it without truncation:

```
uint8 a = 12; // fine
uint32 b = 1234; // fine
uint16 c = 0x123456; // fails, since it would have to truncate to 0x3456
```

**Nota:** Prior to version 0.8.0, any decimal or hexadecimal number literals could be explicitly converted to an integer type. From 0.8.0, such explicit conversions are as strict as implicit conversions, i.e., they are only allowed if the literal fits in the resulting range.

#### Fixed-Size Byte Arrays

Decimal number literals cannot be implicitly converted to fixed-size byte arrays. Hexadecimal number literals can be, but only if the number of hex digits exactly fits the size of the bytes type. As an exception both decimal and hexadecimal literals which have a value of zero can be converted to any fixed-size bytes type:

```
bytes2 a = 54321; // not allowed
bytes2 b = 0x12; // not allowed
bytes2 c = 0x123; // not allowed
bytes2 d = 0x1234; // fine
bytes2 e = 0x0012; // fine
bytes4 f = 0; // fine
bytes4 g = 0x0; // fine
```

String literals and hex string literals can be implicitly converted to fixed-size byte arrays, if their number of characters matches the size of the bytes type:

```
bytes2 a = hex"1234"; // fine
bytes2 b = "xy"; // fine
bytes2 c = hex"12"; // not allowed
bytes2 d = hex"123"; // not allowed
bytes2 e = "x"; // not allowed
bytes2 f = "xyz"; // not allowed
```

#### Addresses

As described in *Address Literals*, hex literals of the correct size that pass the checksum test are of `address` type. No other literals can be implicitly converted to the `address` type.

Explicit conversions to `address` are allowed only from `bytes20` and `uint160`.

An `address a` can be converted explicitly to `address payable` via `payable(a)`.

**Nota:** Prior to version 0.8.0, it was possible to explicitly convert from any integer type (of any size, signed or unsigned) to `address` or `address payable`. Starting with in 0.8.0 only conversion from `uint160` is allowed.

## 3.7 Unidades y variables disponibles globalmente

### 3.7.1 Unidades de Ether

Un número literal puede tomar un sufijo de `wei`, `gwei` o `ether` para especificar una subdenominación de Ether, donde números de Ether sin un postfix se supone que son Wei.

```
assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 ether == 1e18);
```

El único efecto del sufijo subdenominación es una multiplicación por una potencia de diez.

**Nota:** Las denominaciones `finney` y `szabo` se han eliminado en la versión 0.7.0.

### 3.7.2 Unidades de tiempo

Sufijos como `seconds`, `minutes`, `hours`, `days` y `weeks` después de números literales se pueden utilizar para especificar unidades de tiempo donde segundos son la unidad base y las unidades se consideran ingenuamente de la siguiente manera:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`

Tenga cuidado si realiza cálculos de calendario utilizando estas unidades, porque no todos los años equivalen a 365 días y ni siquiera todos los días tienen 24 horas debido a segundos bisiestos. Debido a que los segundos bisiestos no se pueden predecir, una librería de calendario exacta tiene que ser actualizado por un oráculo externo.

**Nota:** El sufijo `years` se ha quitado en la versión 0.5.0 debido a las razones anteriores.

Estos sufijos no se pueden aplicar a las variables. Por ejemplo, si quieres interpretar un parámetro de función en días, puede hacerlo de las siguientes maneras:

```
function f(uint start, uint daysAfter) public {
    if (block.timestamp >= start + daysAfter * 1 days) {
      // ...
    }
}
```

### 3.7.3 Variables y funciones especiales

Hay variables y funciones especiales que siempre existen en el espacio de nombres global y se utilizan principalmente para proporcionar información sobre el blockchain o son funciones de utilidad de uso general.

#### Propiedades de bloques y transacciones

- `blockhash(uint blockNumber) returns (bytes32)`: hash del bloque dado cuando `blocknumber` es uno de los 256 bloques más recientes; de lo contrario devuelve cero
- `block.basefee` (`uint`): tarifa base del bloque actual (EIP-3198 y EIP-1559)
- `block.chainid` (`uint`): ID de cadena actual
- `block.coinbase` (`address payable`): dirección actual del minero del bloque
- `block.difficulty` (`uint`): dificultad del bloque actual. Para otras versiones de EVM se comporta como un alias obsoleto de

`block.prevrandao` (EIP-4399 ) - `block.gaslimit` (`uint`): límite de gas del bloque actual - `block.number` (`uint`): número de bloque actual - `block.timestamp` (`uint`): marca de tiempo de bloque actual como segundos desde la época unix - `gasleft() returns (uint256)`: gas restante - `msg.data` (`bytes calldata`): calldata completo - `msg.sender` (`address`): remitente del mensaje (llamada actual) - `msg.sig` (`bytes4`): primeros cuatro bytes de calldata (i.e. identificador de función) - `msg.value` (`uint`): número de wei enviado con el mensaje - `tx.gasprice` (`uint`): precio del gas de la transacción - `tx.origin` (`address`): remitente de la transacción (cadena de llamadas completa)

**Nota:** Los valores de todos los miembros de `msg`, incluyendo `msg.sender` y `msg.value` pueden cambiar para cada llamada a una función **externa**. Esto incluye llamadas a funciones de librería.

**Nota:** Cuando se evalúan los contratos fuera de la cadena en lugar de en contexto de una transacción incluido en un bloque, no debería asumir que `block.*` y `tx.*` se refieren a valores de cualquier bloque o transacción específica. Estos valores son proporcionados por la implementación de EVM que ejecuta el contrato y pueden ser arbitrarios.

**Nota:** No confíe en `block.timestamp` o `blockhash` como fuente de aleatoriedad, a menos que sepas lo que estás haciendo.

Tanto la marca de tiempo y el hash de bloque pueden ser influenciados por los mineros hasta cierto punto. Malos actores en la comunidad minera pueden, por ejemplo, ejecutar una función de pago de casino en un hash elegido y simplemente reintentar un hash diferente si no recibieron dinero.

La marca de fecha del bloque actual debe ser estrictamente más grande que la marca de fecha del último bloque, pero la única garantía es que estará entre las marcas de fecha de dos bloques consecutivos en la cadena canónica.

**Nota:** Los hashes de bloque no están disponibles para todos los bloques por razones de escalabilidad. Solo puede acceder a los hashes de los 256 bloques más recientes, todos los demás valores serán cero.

**Nota:** La función `blockhash` se conocía anteriormente como `block.blockhash`, que quedó en desuso en la versión 0.4.22 y eliminado en la versión 0.5.0.

**Nota:** La función `gasleft` se conocía anteriormente como `msg.gas`, que quedó en desuso en la versión 0.4.21 y eliminado en la versión 0.5.0.

**Nota:** En versión 0.7.0, el alias `now` (para `block.timestamp`) se quitó.

### Funciones de codificación y decodificación ABI

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI-decodifica los datos dados, mientras que los tipos se dan entre paréntesis como segundo argumento. Ejemplo: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: ABI codifica los argumentos dados
- `abi.encodePacked(...) returns (bytes memory)`: Realiza *packed encoding* de los argumentos dados. ¡Tenga en cuenta que la codificación empaquetada puede ser ambigua!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: ABI codifica los argumentos dados a partir del segundo y antepone el selector de cuatro bytes dado
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Equivalente a `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)`: ABI codifica una llamada a `functionPointer` con los argumentos encontrados en la tupla. Realiza una comprobación de tipo completa, garantizar que los tipos coincidan la firma de la función. El resultado es igual a `abi.encodeWithSelector(functionPointer.selector, (...))`

**Nota:** Estas funciones de codificación se pueden utilizar para crear datos para llamadas a funciones externas sin llamar realmente a una función externa. Además, `keccak256(abi.encodePacked(a, b))` es una forma de calcular el hash de los datos estructurados (aunque tenga en cuenta que es posible crear una «colisión de hash» usando diferentes tipos de parámetros de función).

Consulte la documentación sobre la *ABI* y la *codificación tightly packed* para obtener más información sobre la codificación.

### Miembros de bytes

- `bytes.concat(...) returns (bytes memory)`: *Concatena el número de variable de bytes y bytes1, ..., argumentos de bytes32 para una matriz de bytes*

### Miembros de cadena

- `string.concat(...) returns (string memory)`: *Concatena el número variable de argumentos de cadena en una matriz de cadenas*

### Manejo de errores

Consulte la sección dedicada a *assert y require* para obtener más información sobre el control de errores y cuándo usar qué función.

**`assert(bool condition)`**
provoca un error de pánico y, por lo tanto, cambiar el estado de reversión si no se cumple la condición - para ser utilizado para errores internos.

**`require(bool condition)`**
se revierte si no se cumple la condición - para ser utilizado para errores en componentes internos o externos.

**`require(bool condition, string memory message)`**
se revierte si no se cumple la condición - para ser utilizado para errores en componentes internos o externos. También proporciona un mensaje de error.

**`revert()`**
anular la ejecución y revertir los cambios de estado

**`revert(string memory reason)`**
anular la ejecución y revertir los cambios de estado, proporcionar una cadena explicativa

### Funciones matemáticas y criptográficas

**`addmod(uint x, uint y, uint k) returns (uint)`**
calcular `(x + y) % k` donde se realiza la adición con precisión arbitraria y no se envuelve en `2**256`. Afirman que `k != 0` a partir de la versión 0.5.0.

**`mulmod(uint x, uint y, uint k) returns (uint)`**
calcular `(x * y) % k` donde se realiza la multiplicación con precisión arbitraria y no se envuelve en `2**256`. Afirman que `k != 0` a partir de la versión 0.5.0.

**`keccak256(bytes memory) returns (bytes32)`**
computadora el hash Keccak-256 de la entrada

---

**Nota:** Solía haber un alias para `keccak256` llamado `sha3`, que se eliminó en la versión 0.5.0.

---

**`sha256(bytes memory) returns (bytes32)`**
computa el hash SHA-256 de la entrada

**`ripemd160(bytes memory) returns (bytes20)`**
computa el hash RIPEMD-160 de la entrada

**ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)**
recupera la dirección asociada con la clave pública de la firma de curva elíptica o devuelva cero en el error. Los parámetros de la función corresponden a los valores de ECDSA de la firma:

- r = primeros 32 bytes de firma
- s = segundo 32 bytes of signature
- v = 1 byte final de firma

ecrecover devuelve una address, y no una address payable. Ver *dirección payable* para la conversión, en caso de que necesite transferir fondos a la dirección recuperada.

Para más detalles, lea ejemplo de uso.

**Advertencia:** Si utiliza ecrecover, tenga en cuenta que una firma válida se puede convertir en una firma válida diferente sin requerir el conocimiento de la clave privada correspondiente. En la bifurcación dura de Homestead, este problema se ha corregido para las firmas de _transaction_ (see EIP-2), pero la función ecrecover permaneció sin cambios.

Por lo general, esto no es un problema a menos que requiera que las firmas sean únicas o utilícelos para identificar elementos. OpenZeppelin tiene una biblioteca auxiliar de ECDSA que puede usar como envoltorio para ecrecover sin este problema.

**Nota:** Al ejecutar sha256, ripemd160 o ecrecover en un *blockchain privado*, es posible que se encuentre sin gas. Esto se debe a que estas funciones se implementan como «contratos precompilados» y solo existen realmente después de recibir el primer mensaje (aunque su código de contrato está codificado). Los mensajes a contratos inexistentes son más caros y, por lo tanto, la ejecución podría encontrarse con un error de falta de gas. Una solución para este problema es enviar primero Wei (1 por ejemplo) a cada uno de los contratos antes de utilizarlos en sus contratos reales. Esto no es un problema en la red principal o de prueba.

### Miembros de tipos de direcciones

**<address>.balance (uint256)**
balance de la dirección en Wei

**<address>.code (bytes memory)**
código en la dirección (puede estar vacío)

**<address>.codehash (bytes32)**
el codehash de la dirección

**<address payable>.transfer(uint256 amount)**
envía la cantidad dada de Wei a dirección, revierte en caso de error, adelanta 2300 estipendios de gas, no ajustable

**<address payable>.send(uint256 amount) returns (bool)**
envìa la cantidad dada de Wei a dirección, devuelve false en caso de error, adelanta 2300 estipendios de gas, no ajustable

**<address>.call(bytes memory) returns (bool, bytes memory)**
emite CALL de bajo nivel con la carga útil dada, devuelve la condición de éxito y devuelve datos, reenvía todo el gas disponible, ajustable

**<address>.delegatecall(bytes memory) returns (bool, bytes memory)**
emite DELEGATECALL de bajo nivel con la carga útil dada, devuelve la condición de éxito y devuelve datos, reenvía todo el gas disponible, ajustable

`<address>.staticcall(bytes memory) returns (bool, bytes memory)`
: emite `STATICCALL` de bajo nivel con la carga útil dada, devuelve la condición de éxito y devuelve datos, reenvía todo el gas disponible, ajustable

Para obtener más información, consulte la sección sobre dirección.

**Advertencia:** Debe evitar usar `.call()` siempre que sea posible al ejecutar otra función de contrato, ya que omite la comprobación de tipo, comprobación de existencia de funciones y empaquetado de argumentos.

**Advertencia:** Hay algunos peligros en el uso de `send`: La transferencia falla si la profundidad de la pila de llamadas está en 1024 (esto siempre puede ser forzado por el autor de la llamada) y también falla si el receptor se queda sin gasolina. Entonces, para hacer transferencias seguras de Ether, compruebe siempre el valor devuelto de `send`, usar `transfer` o incluso mejor: Use un patrón en el que el destinatario retire el dinero.

**Advertencia:** Debido a que la EVM considera que una llamada a un contrato inexistente siempre tiene éxito, Solidity incluye una comprobación adicional utilizando el opcode `extcodesize` al realizar llamadas externas. Esto asegura que el contrato que está a punto de ser llamado realmente existe (contiene código) o se genera una excepción.

Las llamadas de bajo nivel que operan en direcciones en lugar de instancias de contrato (i.e. `.call()`, `.delegatecall()`, `.staticcall()`, `.send()` y `.transfer()`) **no** incluya esta comprobación, lo que los hace más baratos en términos de gas, pero también menos seguros.

**Nota:** Antes de la versión 0.5.0, Solidity permitió que se accediera a los miembros de la dirección mediante una instancia de contrato, por ejemplo `this.balance`. Esto ahora está prohibido y se debe hacer una conversión explícita a la dirección: `address(this).balance`.

**Nota:** Si se accede a las variables de estado a través de una delegatecall de bajo nivel, el diseño de almacenamiento de los dos contratos debe alinearse para que el contrato llamado tenga acceso correctamente a las variables de almacenamiento del contrato de llamada por su nombre. Por supuesto, este no es el caso si los punteros de almacenamiento se pasan como argumentos de función como en el caso de las bibliotecas de alto nivel.

**Nota:** Antes de la versión 0.5.0, `.call`, `.delegatecall` y `.staticcall` solo devolvieron la condición de éxito y no los datos de devolución.

**Nota:** Antes de la versión 0.5.0, había un miembro llamado `callcode` con una semántica similar pero ligeramente diferente a la de `delegatecall`.

### Relacionados con el contrato

**`this` (tipo de contrato actual)**
: el contrato actual, explícitamente convertible en dirección

**`selfdestruct(address payable recipient)`**
: Destruir el contrato actual, enviando sus fondos a la dirección dada y finalizar la ejecución. Tenga en cuenta que `selfdestruct` tiene algunas peculiaridades heredadas del EVM:

  - la función de recepción del contrato receptor no se ejecuta.
  - el contrato solo se destruye realmente al final de la transacción y `revert` podría «deshacer» la destrucción.

Además, todas las funciones del contrato actual son llamables directamente, incluida la función actual.

**Advertencia:** A partir de la versión 0.8.18, el uso de `selfdestruct` tanto en Solidity como en Yul provocará una advertencia de obsoleto, ya que el opcode `SELFDESTRUCT` sufrirá eventualmente cambios en su comportamiento. deprecation warning, ya que el opcode `SELFDESTRUCT` sufrirá eventualmente cambios en su comportamiento como se indica en EIP-6049.

**Nota:** Antes de la versión 0.5.0, había una función llamada `suicide` con la misma semántica que `selfdestruct`.

### Información de tipo

La expresión `type(X)` se puede utilizar para recuperar información sobre el tipo `X`. Actualmente, la compatibilidad con esta característica es limitada (`X` puede ser un contrato o un tipo entero) pero podría ampliarse en el futuro.

Las siguientes propiedades están disponibles para un tipo de contrato `C`:

**`type(C).name`**
: El nombre del contrato.

**`type(C).creationCode`**
: Matriz de bytes de memoria que contiene el código de bytes de creación del contrato. Esto se puede utilizar en el ensamblaje en línea para crear rutinas de creación personalizadas, especialmente mediante el uso del opcode `create2`. **No** se puede acceder a esta propiedad en el propio contrato o en cualquier contrato derivado. Hace que el bytecode se incluya en el bytecode del sitio de la llamada y, por lo tanto, las referencias circulares como esa no son posibles.

**`type(C).runtimeCode`**
: Matriz de bytes de memoria que contiene el código de bytes en tiempo de ejecución del contrato. Este es el código que suele implementar el constructor de `C`. Si `C` tiene un constructor que utiliza un conjunto en línea, esto podría ser diferente del bytecode realmente implementado. Tenga en cuenta también que las librerías modifican su bytecode en tiempo de ejecución en el momento de la implementación para protegerse contra las llamadas regulares. Las mismas restricciones que con `.creationCode` también se aplican a esta propiedad.

Además de las propiedades anteriores, las siguientes propiedades están disponibles para un tipo de interfaz `I`:

**`type(I).interfaceId:`**
: Un valor `bytes4` que contiene el EIP-165 identificador de interfaz de la interfaz dada `I`. Este identificador se define como el `XOR` de todos los selectores de funciones definidos dentro de la propia interfaz - excluyendo todas las funciones heredadas.

Las siguientes propiedades están disponibles para un tipo entero `T`:

`type(T).min`
: El valor más pequeño representable por el tipo `T`.

`type(T).max`
: El valor más grande representable por el tipo `T`.

### 3.7.4 Palabras clave reservadas

Estas palabras clave están reservadas en Solidity. Podrían formar parte de la sintaxis en el futuro:

`after`, `alias`, `apply`, `auto`, `byte`, `case`, `copyof`, `default`, `define`, `final`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof`, `var`.

## 3.8 Expresiones y Estructuras de Control

### 3.8.1 Estructuras de Control

La mayoría de estructuras de control conocidas de los lenguages que usan corchetes está disponsible en Solidity:

Existen: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, con la semántica habitual conocida de C o JavaScript.

Solidity tambien admite control de excepciones en la forma de instrucciones `try/catch`, pero solo para *llamadas a funciones externas* y las llamadas de la creación de contratos. Errores se pueden crear usando la *sentencia revert*.

Los paréntesis no se pueden omitir para condicionales, pero sí los corchetes alrededor de los cuerpos de las declaraciones sencillas.

Hay que tener en cuenta que no hay conversión de tipos no boolean a boolean como hay en C y JavaScript, por lo que `if (1) { ... }` *no* es válido en Solidity.

### 3.8.2 Llamadas a funciones

#### Llamadas a funciones internas

Las funciones del contrato actual pueden ser llamadas directamente («internamente») y, también, recursivamente como se puede ver en este ejemplo sin sentido funcional:

```
.. code-block:: solidity
```

```
// SPDX-License-Identifier: GPL-3.0 pragma solidity >=0.4.22 <0.9.0;

// Esto reportará un aviso contract C {

    function g(uint a) public pure returns (uint ret) { return a + f(); } function f() internal pure
    returns (uint ret) { return g(7) + f(); }

}
```

Estas llamadas a funciones son traducidas en simples saltos dentro de la máquina virtual de Ethereum (EVM). Esto tiene como consecuencia que la memoria actual no se limpia, así que pasar referencias de memoria a las funciones llamadas internamente es muy eficiente. Solo las funciones del mismo contrato pueden ser llamadas internamente.

Todavía hay que evitar recursion excesiva, como todos las llamadas de funciones internas usan al menos una ranura de pila y solo hay 1024 ranuras disponible.

### Llamadas a funciones externas

Las funciones también pueden llamadas usando la notación this.g(8); and c.g(2); donde c es la instancia de un contrato y g es la funcion que pertenece c. Llamando la funcion `g` de cualquier manera resulta en una llamada externa, usando una llamada de mensaje y no por saltos directamente. Hay que tener cuenta que las llamadas de funciones en `this` no se puede usar en el constructor, ya que el contrato actual aún no se ha creado todavía.

Funciones de otros contratos tienen que ser llamado externamente. Para una llamada externa, todos los arguments de función deben copiarse a la memoria.

---

**Nota:** Una llamada de la función de un contrato a otro no crea su propia transacción, es una llamada de mensaje como parte de la transacción completa.

---

Cuando se llama a funciones de otros contratos, la cantidad de Wei enviada con la llamada y el gas pueden especificarse con las opciones especiales `{value: 10, gas: 10000}`. Hay que tener cuenta que se desaconseja especifar valores de gas explícitamente, ya que los costos de gas de opcodes pueden cambiar en el futuro. Cualquier Wei que envíe se agrega al saldo total de ese contrato.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info{value: 10, gas: 800}(); }
}
```

Debe utilizar el modificador `payable` con la función `info` porque de lo contrario la opción `valor` no estaría disponible.

**Advertencia:** Hay que tener cuidado que `feed.info{value: 10, gas: 800}` solo establezca localmente el `value` y la cantidad de `gas` enviado con la llamada de la función, y los paréntesis al final realizan la llamda actual. Por lo tanto, `feed.info{value: 10, gas: 800}` no ejecuta la función y se pierden los ajustes del `value` y `gas`, solo `feed.info{value: 10, gas: 800}()` realiza la llamada de la función.

Debido al hecho que el EVM considera una llamada a un contrato inexistente siempre tenga éxito, Solitiy utiliza el `extcodesize` opcode para comprobar que el contrato que está a punto de ser llamado existe realmente (contiene codigo) y causa excepción si no lo hace. Esta comprobación se omite si los datos devueltos se descodifican después de la llamada y, por tanto, el descodificador ABI detectará el caso de un contrato no existente.

Hay que tener cuenta que esta comprobación no se realiza en caso de *llamadas de bajo nivel* que operan en las direcciones en lugar de instancias de contrato.

---

**Nota:** Hay que tener cuidado al utilizar llamadas de alto nivel para *contratos precompilados*, dado que el compilador los considera no existentes según la lógica de arriba aunque ejecuten código y puedan devolver datos.

---

Las llamadas a funciones también causan excepciones si el propio contrato llamado arroja una excepción o se queda sin gas.

**Advertencia:** Cualquier interacción con otro contrato supone un peligro potencial, especialmente si el código fuente del contrato no se conoce por adelantado. El contrato actual entrega el control al contrato llamado y eso puede potencialmente hacer casi cualquier cosa. Incluso si el contrato llamado hereda de un contrato principal conocido, el contrato de herencia solo es necesario para tener una interfaz correcta. Sin embargo, la ejecución del contrato puede ser completamente arbitraria y, por lo tanto, plantea un peligro. Además, esté preparado en caso de que convoque otros contratos de su sistema o incluso volver al contrato de llamada antes de que retorne la primera llamada. Esto significa que el contrato llamado puede cambiar las variables de estado del contrato de llamada a través de sus funciones. Escriba sus funciones de forma que, por ejemplo, las llamadas a las funciones externas se produzcan después de cualquier cambio en las variables de estado en su contrato, por lo tanto, su contrato no es vulnerable a una explotación de reentrada.

**Nota:** Antes de Solidity 0.6.2, la forma recomendada de especificar el valor y el gas era use `f.value(x).gas(g)()`. Esto se volvió obsoleto en Solidity 0.6.2 y ya no es posible desde Solidity 0.7.0.

### Llamadas a funciones con parámetros con nombre

Los argumentos de llamada a funciones pueden darse por nombre, en cualquier orden, si están encerrados entre { }, como se puede ver en el siguiente ejemplo. La lista de argumentos tiene que coincidir por nombre con la lista de parámetros de la declaración de función, pero puede estar en orden arbitrario.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }

}
```

### Nombres omitidos en definiciones de funciones

Se pueden omitir los nombres de los parámetros y los valores retornos en la declaración de la función. Los elementos con nombres omitidos seguirán presentes en la pila, pero no se puede acceder a ellos por su nombre. Un nombre de valor retorno omitido todavía puede devolver un valor al llamador mediante la instrucción `return".

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    // nombre omitido para parámetro
    function func(uint k, uint) public pure returns(uint) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        return k;
    }
}
```

### 3.8.3 Creando contratos mediante `new`

Un contrato puede crear otros contratos usando la palabra reservada `new`. El código completo del contrato que se está creando tiene que ser conocido de antemano, por lo que no son posibles las dependencias de creación recursivas.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // Se ejecutará como parte del constructor de C

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
         // Envía Ether junto con la creación
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}
```

Como se ve en el ejemplo, es posible traspasar Ether a la creación usando la opción `.value()`, pero no es posible limitar la cantidad de gas. Si la creación falla (debido al desbordamiento de la pila, falta de balance o cualquier otro problema), se dispara una excepción.

#### Creaciones de contratos salted / create2

Al crear un contrato, la dirección del contrato se calcula a partir de la dirección del contrato de creación y un contador que se incrementa con cada creación de contrato.

Si especifica la opción `salt` (un valor bytes32), la creación de contratos utilizará un mecanismo diferente para encontrar la dirección del nuevo contrato:

Calculará la dirección a partir de la dirección del contrato de creación, el valor de salt dado, el código de bytes (de creación) del contrato creado y los argumentos del constructor.

En particular, no se utiliza el contador (“nonce”). Esto permite una mayor flexibilidad en la creación de contratos: Puede derivar la dirección del nuevo contrato antes de crearlo. Además, puede confiar en esta dirección también en caso de que la creación de contratos cree otros contratos mientras tanto.

El principal caso de uso aquí son los contratos que actúan como jueces para las interacciones fuera de la cadena, que solo deben crearse si hay una disputa.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        // Esta complicada expresión solo te dice cómo la dirección
        // se puede precalcular. Solo está ahí para ilustrar.
        // En realidad, solo necesita ``new D{salt: salt}(arg)``.
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                abi.encode(arg)
            ))
        )))));

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

**Advertencia:** Hay algunas peculiaridades en relación con la creación salted. Un contrato puede ser recreado en la misma dirección después de haber sido destruido. Sin embargo, es posible que ese contrato recién creado tuviera un código de bytes diferente incluso aunque el código de byte de creación ha sido el mismo (lo que es un requisito porque de lo contrario, la dirección cambiaría). Esto se debe al hecho de que el constructor puede consultar el estado externo que podría haber cambiado entre las dos creaciones e incorporarlo al código de bytes implementado antes de que se almacene.

### 3.8.4 Orden de la evaluación de expresiones

El orden de evaluación de expresiones no se especifica (más formalmente, el orden en el que los hijos de un nodo en el árbol de la expresión son evaluados no es especificado. Eso sí, son evaluados antes que el propio nodo). Solo se garantiza que las sentencias se ejecutan en orden y que se hace un cortocircuito para las expresiones booleanas. Ver *Order of Precedence of Operators* para más información.

### 3.8.5 Asignación

#### Asignaciones para desestructurar y retornar múltiples valores

Solidity internamente permite tipos tupla, i.e.: una lista de objetos de, potencialmente, diferentes tipos cuyo tamaño es constante en tiempo de compilación. Esas tuplas pueden ser usadas para retornar múltiples valores al mismo tiempo. Pueden asignarse a variables recién declaradas o variables preexistentes ( o LValues en general).

Las tuplas no son tipos propios en Solidity, Se pueden usar para formar agrupaciones sintácticas de expresiones.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    uint index;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declaradas con tipo y asignadas desde la tupla devuelta,
        // no es necesario especificar todos los elementos (pero el número debe␣
↪coincidir).
        (uint x, , uint y) = f();
        // Truco común para intercambiar valores: no funciona para tipos de␣
↪almacenamiento que no son de valor.
        (x, y) = (y, x);
        // Los componentes se pueden omitir (también para declaraciones de variables).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

No es posible mezclar declaraciones de variables y asignaciones sin declaración, i.e., lo siguiente no es válido: `(x, uint y) = (1, 2);`

**Nota:** Antes de la versión 0.5.0 era posible asignar a tuplas de menor tamaño, ya sea llenando a la izquierda o en el lado derecho (que alguna vez estaba vacío). Esto ahora no está permitido, por lo que ambas partes tienen que tener el mismo número de componentes.

**Advertencia:** Hay que tener cuidado al asignar a varias variables al mismo tiempo cuando se involucran tipos de referencia, ya que podría provocar un comportamiento de copia inesperado.

#### Complicaciones en Arrays y Structs

La sintaxis de asignación es algo más complicada para tipos sin valor como arrays y structs, incluyendo `bytes` y `string`, mira *localización de datos y comportamiento de asignaciones* para detalles.

En el siguiente ejemplo la llamada a `g(x)` no tiene ningún efecto en `x` porque crea una copia independiente del valor de almacenamiento en la memoria. Sin embargo, `h(x)` modifica con éxito `x` porque solo se pasa una referencia y no una copia.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

### 3.8.6 Scoping y declaraciones

Una variable cuando se declara tendrá un valor inicial por defecto que, representado en bytes, será todo ceros. Los valores por defecto de las variables son los típicos «estado-cero» cualquiera que sea el tipo. Por ejemplo, el valor por defecto para un `bool` es `false`. El valor por defecto para un `uint` o `int` es `0`. Para arrays de tamaño estático y `bytes1` hasta `bytes32`, cada elemento individual será inicializado a un valor por defecto según sea su tipo. Para arrays de tamaño dinámico, `bytes``y ``string`, el valor por defecto es un array o string vacío. Para el tipo `enum`, el valor por defecto es su primer miembro.

El alcance en Solidity sigue las reglas de alcance generalizadas de C99 (y muchos otros lenguajes): Las variables son visibles desde el punto justo después de su declaración hasta el final del bloque más pequeño `{ }` que contiene la declaración. Como excepción a esta regla, las variables declaradas en la parte de inicialización de un for-loop solo son visibles hasta el final del for-loop.

Las variables que son similares a los parámetros (parámetros de función, parámetros modificadores, parámetros de captura, ...) son visibles dentro del bloque de código que sigue - el cuerpo de la función/modificador para una función y parámetro modificador y el bloque de captura para un parámetro de captura.

Las variables y otros elementos declarados fuera de un bloque de código, por ejemplo, funciones, contratos, tipos definidos por el usuario, etc., son visibles incluso antes de que se declararan. Esto significa que puede usar variables de estado antes de que se declaren y llamar a funciones de forma recursiva.

Como consecuencia, los siguientes ejemplos se compilarán sin advertencias, ya que las dos variables tienen el mismo nombre pero ámbitos disjuntos.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
    }
}
```

Como ejemplo especial de las reglas de alcance de C99, tenga en cuenta que en lo siguiente, la primera asignación a `x` en realidad asignará la variable externa y no la interna. En cualquier caso, recibirá una advertencia sobre la variable externa que se sombrea.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// Esto informará de una advertencia
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // esto se asignará a la variable externa
            uint x;
        }
        return x; // x tiene valor 2
    }
}
```

**Advertencia:** Antes de la versión 0.5.0, Solidity seguía las mismas reglas de ámbito que JavaScript, es decir, una variable declarada en cualquier lugar dentro de una función estaría en el ámbito para toda la función, independientemente de dónde se haya declarado. En el ejemplo siguiente se muestra un fragmento de código que solía compilar pero conduce a un error a partir de la versión 0.5.0.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// Esto no se compilará
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

### 3.8.7 Aritmética comprobada o no comprobada

Un desbordamiento o subflujo es la situación en la que el valor resultante de una operación aritmética, cuando se ejecuta en un entero sin restricciones, cae fuera del rango del tipo de resultado.

Antes de Solidity 0.8.0, las operaciones aritméticas siempre se envolvían en caso de desbordamiento o desbordamiento, lo que llevaría a un uso generalizado de bibliotecas que introducen comprobaciones adicionales.

Desde Solidity 0.8.0, todas las operaciones aritméticas se revierten por defecto en el subdesbordamiento o desbordamiento, lo que hace innecesario el uso de estas bibliotecas.

Para obtener el comportamiento anterior, se puede utilizar un bloque `unchecked`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // Esta resta se envolverá en el desbordamiento.
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // Esta resta se revertirá en caso de subdesbordamiento.
        return a - b;
    }
}
```

La llamada a `f(2, 3)` devolverá `2**256-1`, mientras que `g(2, 3)` causará una aserción fallida.

El bloque `unchecked` se puede usar en todas partes dentro de un bloque, pero no como reemplazo de un bloque. Tampoco se puede anidar.

La configuración solo afecta a las instrucciones que se encuentran sintácticamente dentro del bloque. Las funciones llamadas desde un bloque `unchecked` no heredan la propiedad.

---

**Nota:** Para evitar la ambigüedad, no puede usar `_;` dentro de un bloque `unchecked`.

---

Los siguientes operadores provocarán un error de aserción en el desbordamiento o subdesbordamiento y se ajustarán sin error si se utilizan dentro de un bloque descomprobada:

++, --, +, binario -, unario -, *, /, %, **

+=, -=, *=, /=, %=

**Advertencia:** No es posible desactivar la comprobación de división por cero o módulo por cero utilizando el bloque `unchecked`.

---

**Nota:** Los operadores Bitwise no realizan comprobaciones de desbordamiento o subdesbordamiento. Esto es particularmente visible cuando se usan desplazamientos bitwise (<<, >>, <<=, >>=) en lugar de la división entera y la multiplicación por una potencia de 2. Por ejemplo, `type(uint256).max << 3` no se revierte aunque `type(uint256).max * 8` lo haría.

---

**Nota:** La segunda instrucción en `int x = type(int).min; -x;` dará lugar a un desbordamiento porque el rango

negativo puede contener un valor más que el rango positivo.

Las conversiones de tipos explícitos siempre se truncarán y nunca causarán una aserción errónea con la excepción de una conversión de un entero a un tipo enum.

### 3.8.8 Manejo de errores: Afirmar, Requerir, Revertir y Excepciones

Solidity utiliza excepciones de reversión de estado para controlar los errores. Tal excepción deshace todos los cambios realizados en el estado de la llamada actual (y todas sus subllamadas) y marca un error al autor de la llamada.

Cuando las excepciones ocurren en una subllamada, «burbujean» (i.e., las excepciones se vuelven a lanzar) automáticamente a menos que se detecten en una instrucción `try/catch`. Las excepciones a esta regla son `send` y las funciones de bajo nivel `call`, `delegatecall` y `staticcall`: devuelven `false` como su primer valor devuelto en caso de una excepción en lugar de «burbujear».

**Advertencia:** Las funciones de bajo nivel `call`, `delegatecall` y `staticcall` devuelve `true` como su primer valor de retorno si la cuenta llamada no existe, como parte del diseño del EVM. Debe comprobarse la existencia de la cuenta antes de llamar.

Las excepciones pueden contener datos de error que se devuelven al autor de la llamada en forma de *error instances*. Los errores incorporados `Error(string)` y `Panic(uint256)` son utilizados por funciones especiales, como se explica a continuación. `Error` se usa para condiciones de error «regular», mientras que `Panic` se usa para errores que no deberían estar presentes en el código libre de errores.

#### Panic a través de `assert` y Error a través de `require`

Las funciones de conveniencia `assert` y `require` se pueden usar para verificar las condiciones y lanzar una excepción si no se cumple la condición.

La función `assert` crea un error del tipo `Panic(uint256)`. El compilador crea el mismo error en ciertas situaciones que se enumeran a continuación.

Assert solo debe usarse para detectar errores internos y para verificar invariantes. El correcto funcionamiento del código nunca debe crear un pánico, ni siquiera en una entrada externa no válida. Si esto sucede, entonces hay un error en su contrato que debe corregir. Las herramientas de análisis de lenguaje pueden evaluar su contrato para identificar las condiciones y las llamadas a funciones que causarán pánico.

Se genera una excepción de pánico en las siguientes situaciones. El código de error proporcionado con los datos de error indica el tipo de pánico.

1. 0x00: Se utiliza para pánicos insertados en el compilador genérico.
2. 0x01: Si llama a `assert` con un argumento que se evalúa como false.
3. 0x11: Si una operación aritmética resulta en subdesbordamiento o desbordamiento fuera de un bloque `unchecked { ... }`.
4. 0x12; Si divide o modulo por cero (por ejemplo `5 / 0` o `23 % 0`).
5. 0x21: Si convierte un valor demasiado grande o negativo en un tipo de enumeración.
6. 0x22: Si accede a una matriz de bytes de almacenamiento que está codificada incorrectamente.
7. 0x31: Si llama a `.pop()` en un array vacío.
8. 0x32: Si accede a una matriz, `bytesN` o a un segmento de matriz en un índice negativo o fuera de los límites (i.e. `x[i]` donde `i >= x.length` o `i < 0`).

9. 0x41: Si asigna demasiada memoria o crea una matriz que es demasiado grande.
10. 0x51: Si llama a una variable inicializada en cero de tipo de función interna.

La función``"require`` crea un error sin ningún dato o un error del tipo `Error(string)`. Debe utilizarse para garantizar condiciones válidas que no se puedan detectar hasta el momento de la ejecución. Esto incluye condiciones sobre entradas o valores devueltos de llamadas a contratos externos.

---

**Nota:** Actualmente no es posible utilizar errores personalizados en combinación con *require*`. Utilice `if (! condition) revert CustomError();` en su lugar.

---

El compilador genera una excepción `Error(string)` (o una excepción sin datos) en las siguientes situaciones:

1. Llamar a `require(x)` donde `x` se evalúa como `false`.
2. Si se utiliza `revert()` o `revert("description")`.
3. Si se realiza una llamada de función externa apuntando a un contrato que no contiene código.
4. Si un contrato recibe Ether mediante una función sin el modificador `payable` (incluyendo el constructor y la función de fallback).
5. Si un contrato recibe Ether mediante una función getter pública.

Para los siguientes casos, se reenvían los datos de error de la llamada externa (si se proporcionan). Esto significa que puede causar un *Error* o un *Pánico* (o cualquier otra cosa que se haya dado):

1. Si un `.transfer()` falla.
2. Si llama a una función a través de una llamada de mensaje pero no termina correctamente (i.e., se queda sin gas, no tiene una función coincidente, o lanza una excepción en sí misma), excepto cuando se utiliza una operación de bajo nivel `call`, `send`, `delegatecall`, `callcode` or `staticcall`. Las operaciones de bajo nivel nunca arrojan excepciones, sino que indican errores devolviendo `false`.
3. Si crea un contrato utilizando la palabra clave `new` pero la creación del contrato *no finaliza propiamente*.

Opcionalmente, puede proporcionar una cadena de mensaje para `require`, pero no para `assert`.

---

**Nota:** Si no proporciona un argumento de cadena a `require`, se revertirá con datos de error vacíos, sin siquiera incluir el selector de errores.

---

En el ejemplo siguiente se muestra cómo puede utilizar `require` para comprobar las condiciones de las entradas y `assert` para la comprobación interna de errores.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // Dado que la transferencia arroja una excepción en caso de fallo y
        // no puedo volver a llamar aquí, no debería haber forma de que lo hagamos
        // todavía tienen la mitad del dinero.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }
}
```

Internamente, Solidity realiza una operación de reversión (instrucción `0xfd`). Esto hace que el EVM revierta todos los cambios realizados en el estado. La razón para revertir es que no hay una forma segura de continuar la ejecución, porque no se produjo un efecto esperado. Debido a que queremos mantener la atomicidad de las transacciones, la acción más segura es revertir todos los cambios y dejar (o al menos llamar) sin efecto toda la transacción.

En ambos casos, quien llama puede reaccionar ante tales fallos usando `try/catch`, pero los cambios en quien está siendo llamado siempre se revertirán.

---

**Nota:** Las excepciones de pánico solían usar el código de operación `invalid` antes de Solidity 0.8.0, que consumía todo el gas disponible para la llamada. Las excepciones que usan `require` solían consumir todo el gas hasta antes del lanzamiento de Metropolis.

---

### `revert`

Se puede activar una reversión directa utilizando la instrucción `revert` y la función `revert`.

La instrucción `revert` accepta un error personalizado como argumento directo sin paréntesis:

    revert CustomError(arg1, arg2);

Por razones de compatibilidad con versiones anteriores, también existe la función `revert()`, que utiliza paréntesis y acepta una cadena:

    revert(); revert(«description»);

Los datos de error se devolverán a la persona que llama y se pueden capturar allí. El uso de `revert()` causa una reversión sin ningún dato de error, mientras que `revert("description")` creará un error `Error(string)`.

El uso de una instancia de error personalizada generalmente será mucho más barato que una descripción de cadena, por que puede usar el nombre del error para describirlo, que está codificado en solo cuatro bytes. Se puede proporcionar una descripción más larga a través de NatSpec que no incurre en ningún costo.

En el ejemplo siguiente se muestra cómo utilizar una cadena de error y una instancia de error personalizada junto con `revert` y el equivalente `require`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Forma alternativa de hacerlo:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Realice la compra.
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        payable(msg.sender).transfer(address(this).balance);
    }
}
```

Las dos formas `if (!condition) revert(...);` y `require(condition, ...);` son equivalentes siempre y cuando los argumentos para `revert` y `require` no tengan efectos secundarios, por ejemplo, si son solo cadenas.

---

**Nota:** La función `require` se evalúa como cualquier otra función. Esto significa que todos los argumentos se evalúan antes de ejecutar la función en sí. En particular, en `require(condition, f())` la función `f` se ejecuta incluso si `condition` es true.

---

La cadena proporcionada es *abi-encoded* como si fuera una llamada a una función `Error(string)`. En el ejemplo anterior, ``revertir(«No se proporciona suficiente éter.»); `` devuelve el siguiente hexadecimal como datos de retorno de error:

```
0x08c379a0                                                         // Selector de␣
↪funciones para Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Desplazamiento de␣
↪datos
0x000000000000000000000000000000000000000000000000000000000000001a // Longitud de la␣
↪cadena
0x4e6f7420656e6f756768204574686572207072f76696465642e000000000000 // Dato de cadena
```

El mensaje proporcionado puede ser recuperado por la persona que llama usando `try/catch` como se muestra a continuación.

---

**Nota:** Solía haber una palabra clave llamada `throw` con la misma semántica que `revert()` que estaba en desuso en la versión 0.4.13 y eliminada en la versión 0.5.0.

---

### try/catch

Un error en una llamada externa se puede detectar mediante una instrucción try/catch, de la siguiente manera:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Desactivar permanentemente el mecanismo si hay
        // mas de 10 errores.
        require(errorCount < 10);
```

(continué en la próxima página)

(proviene de la página anterior)

```
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // Esto se ejecuta en caso de que
            // se llamó un revert dentro de getData
            // y se proporcionó una cuerda de razón.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // Esto se ejecuta en caso de pánico,
            // i.e. un error grave como la división por cero
            // o desbordamiento. Se puede utilizar el código de error
            // para determinar el tipo de error.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // Esto se ejecuta en caso de que se haya utilizado revert().
            errorCount++;
            return (0, false);
        }
    }
}
```

La palabra clave `try``tiene que ser seguida de una expresión que represente una llamada a una función externa o una creación de contrato (``new ContractName()`). Los errores dentro de la expresión no se detectan (por ejemplo, si se trata de una expresión compleja que también implica llamadas a funciones internas), solo una reversión dentro de la propia llamada externa. La parte `returns` (que es opcional) que sigue declara variables de retorno que coinciden con los tipos devueltos por la llamada externa. En caso de que no hubiera error, se asignan estas variables y la ejecución del contrato continúa dentro del primer bloque de éxito. Si se llega al final del bloque de éxito, la ejecución continúa después de los bloques de «captura».

Solidity admite diferentes tipos de bloques de captura dependiendo del tipo de error:

- **`catch Error(string memory reason) { ... }`: Esta cláusula catch se ejecuta si el error fue causado por `revert("reasonString")` o**
  `require(false, "reasonString")` (o un error interno que causa tal excepción).
- **`catch Panic(uint errorCode) { ... }`: Si el error fue causado por un pánico, i.e, por un error `assert`, división por cero,**
  acceso a matriz no válido, desbordamiento aritmético y otros, se ejecutará esta cláusula catch.
- **`catch (bytes memory lowLevelData) { ... }`: Esta cláusula se ejecuta si la firma de error**
  no coincide con ninguna otra cláusula, si hubo un error al decodificar el mensaje de error, o si ningunos datos de error se proveyeran de la excepción. La variable declarada proporciona acceso a los datos de error de bajo nivel en ese caso.
- `catch { ... }`: Si no está interesado en los datos de error, puede usar `catch { ... }` (incluso como la única cláusula de captura) en lugar de la cláusula anterior.

Está planeado para soportar otros tipos de datos de error en el futuro. Las cadenas `Error` y `Pánico` se analizan actualmente tal cual y no se tratan como identificadores.

Para detectar todos los casos de errores, debe tener al menos las cláusas `catch { ...}` o `catch (bytes memory lowLevelData) { ... }`.

Los variables declarado en en la cláusula `returns` y `catch` solo están en el ámbito en el bloque que sigue.

**Nota:** Si se produce un error durante la decodificación de los datos devueltos dentro de una sentencia try/catch, esto provoca una excepción en el actual ejecución de contrato y por eso, no se captura en la cláusula catch. Si hay un error durante la decodificación de `catch Error(string memory reason)` y hay una cláusula catch de bajo nivel, este error se detecta allí.

**Nota:** Si la ejecución alcanza un bloque catch, entonces los efectos de cambio de estado de la llamada externa han sido revertidos. Si la ejecución alcanza el bloque de éxito, los efectos no se revertieron. Si los efectos se han revertido, la ejecución continuará en un bloque catch o la ejecución de la propia sentencia try/catch se revierte (por ejemplo, debido a errores de descodificación como se ha indicado anteriormente o debido a que no se proporciona una cláusula catch de bajo nivel).

**Nota:** El motivo de una llamada fallida puede ser múltiple. No asuma que el mensaje de error procede directamente del contrato llamado: El error podría haberse producido en una fase más profunda de la cadena de llamadas y el contrato llamado acaba de reenviarlo. Además, podría deberse a un situación de falta de gas y no una condición de error deliberada: La persona que llama siempre retiene al menos 1/64th del gas en una llamada y, por lo tanto, incluso si el contrato llamado se queda sin gas, la persona que llama aún le queda algo de gas.

## 3.9 Contractos

Los contratos en Solidity son similares a las clases en lenguajes orientados a objetos. Contienen datos persistentes en variables de estado y funciones que pueden modificar estas variables. Llamar a una función en un contrato diferente (una instancia) realizará una llamada a una función en el EVM y, por lo tanto, cambiará el contexto, de modo que las variables de estado en el contrato llamado se vuelven inaccesibles. Un contrato y sus funciones deben ser llamados para que algo suceda. No existe un concepto «cron» en Ethereum para llamar automáticamente a una función en un evento en particular.

### 3.9.1 Creating Contracts

Contracts can be created «from outside» via Ethereum transactions or from within Solidity contracts.

IDEs, such as Remix, make the creation process seamless using UI elements.

One way to create contracts programmatically on Ethereum is via the JavaScript API web3.js. It has a function called web3.eth.Contract to facilitate contract creation.

When a contract is created, its *constructor* (a function declared with the `constructor` keyword) is executed once.

A constructor is optional. Only one constructor is allowed, which means overloading is not supported.

After the constructor has executed, the final code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.

Internally, constructor arguments are passed *ABI encoded* after the code of the contract itself, but you do not have to care about this if you use `web3.js`.

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;


contract OwnedToken {
    // `TokenCreator` is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // This is the constructor which registers the
    // creator and the assigned name.
    constructor(bytes32 name_) {
        // State variables are accessed via their name
        // and not via e.g. `this.owner`. Functions can
        // be accessed directly or through `this.f`,
        // but the latter provides an external view
        // to the function. Especially in the constructor,
        // you should not access functions externally,
        // because the function does not exist yet.
        // See the next section for details.
        owner = msg.sender;

        // We perform an explicit type conversion from `address`
        // to `TokenCreator` and assume that the type of
        // the calling contract is `TokenCreator`, there is
        // no real way to verify that.
        // This does not create a new contract.
        creator = TokenCreator(msg.sender);
        name = name_;
    }

    function changeName(bytes32 newName) public {
        // Only the creator can alter the name.
        // We compare the contract based on its
        // address which can be retrieved by
        // explicit conversion to address.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;

        // We ask the creator contract if the transfer
        // should proceed by using a function of the
        // `TokenCreator` contract defined below. If
        // the call fails (e.g. due to out-of-gas),
        // the execution also fails here.
        if (creator.isTokenTransferOK(owner, newOwner))
```

(continué en la próxima página)

(proviene de la página anterior)

```
            owner = newOwner;
    }
}


contract TokenCreator {
    function createToken(bytes32 name)
        public
        returns (OwnedToken tokenAddress)
    {
        // Create a new `Token` contract and return its address.
        // From the JavaScript side, the return type
        // of this function is `address`, as this is
        // the closest type available in the ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
        // Again, the external type of `tokenAddress` is
        // simply `address`.
        tokenAddress.changeName(name);
    }

    // Perform checks to determine if transferring a token to the
    // `OwnedToken` contract should proceed
    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // Check an arbitrary condition to see if transfer should proceed
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

### 3.9.2 Visibility and Getters

#### State Variable Visibility

**public**
: Public state variables differ from internal ones only in that the compiler automatically generates *getter functions* for them, which allows other contracts to read their values. When used within the same contract, the external access (e.g. `this.x`) invokes the getter while internal access (e.g. `x`) gets the variable value directly from storage. Setter functions are not generated so other contracts cannot directly modify their values.

**internal**
: Internal state variables can only be accessed from within the contract they are defined in and in derived contracts. They cannot be accessed externally. This is the default visibility level for state variables.

**private**
: Private state variables are like internal ones but they are not visible in derived contracts.

**Advertencia:** Making something `private` or `internal` only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

### Function Visibility

Solidity knows two kinds of function calls: external ones that do create an actual EVM message call and internal ones that do not. Furthermore, internal functions can be made inaccessible to derived contracts. This gives rise to four types of visibility for functions.

**`external`**
: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works).

**`public`**
: Public functions are part of the contract interface and can be either called internally or via message calls.

**`internal`**
: Internal functions can only be accessed from within the current contract or contracts deriving from it. They cannot be accessed externally. Since they are not exposed to the outside through the contract's ABI, they can take parameters of internal types like mappings or storage references.

**`private`**
: Private functions are like internal ones but they are not visible in derived contracts.

**Advertencia:** Making something `private` or `internal` only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

The visibility specifier is given after the type for state variables and between parameter list and return parameter list for functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

In the following example, `D`, can call `c.getData()` to retrieve the value of `data` in state storage, but is not able to call `f`. Contract `E` is derived from `C` and, thus, can call `compute`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }
    function setData(uint a) public { data = a; }
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
```

(continué en la próxima página)

(proviene de la página anterior)

```
}

// This will not compile
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // error: member `f` is not visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: member `compute` is not visible
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // access to internal member (from derived to parent␣
↪contract)
    }
}
```

### Getter Functions

The compiler automatically creates getter functions for all **public** state variables. For the contract given below, the compiler will generate a function called `data` that does not take any arguments and returns a `uint`, the value of the state variable `data`. State variables can be initialized when they are declared.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

The getter functions have external visibility. If the symbol is accessed internally (i.e. without `this.`), it evaluates to a state variable. If it is accessed externally (i.e. with `this.`), it evaluates to a function.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // internal access
```

(continué en la próxima página)

(proviene de la página anterior)

```
        return this.data(); // external access
    }
}
```

If you have a `public` state variable of array type, then you can only retrieve single elements of the array via the generated getter function. This mechanism exists to avoid high gas costs when returning an entire array. You can use arguments to specify which individual element to return, for example `myArray(0)`. If you want to return an entire array in one call, then you need to write a function, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract arrayExample {
    // public state variable
    uint[] public myArray;

    // Getter function generated by the compiler
    /*
    function myArray(uint i) public view returns (uint) {
        return myArray[i];
    }
    */

    // function that returns entire array
    function getArray() public view returns (uint[] memory) {
        return myArray;
    }
}
```

Now you can use `getArray()` to retrieve the entire array, instead of `myArray(i)`, which returns a single element per call.

The next example is more complex:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping(uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping(uint => mapping(bool => Data[])) public data;
}
```

It generates a function of the following form. The mapping and arrays (with the exception of byte arrays) in the struct are omitted because there is no good way to select individual struct members or provide a key for the mapping:

```
function data(uint arg1, bool arg2, uint arg3)
    public
    returns (uint a, bytes3 b, bytes memory e)
{
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
    e = data[arg1][arg2][arg3].e;
}
```

### 3.9.3 Function Modifiers

Modifiers can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function.

Modifiers are inheritable properties of contracts and may be overridden by derived contracts, but only if they are marked `virtual`. For details, please see *Modifier Overriding*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;
// This will report a warning due to deprecated selfdestruct

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;

    // This contract only defines a modifier but does not use
    // it: it will be used in derived contracts.
    // The function body is inserted where the special symbol
    // `_;` in the definition of a modifier appears.
    // This means that if the owner calls this function, the
    // function is executed and otherwise, an exception is
    // thrown.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract destructible is owned {
    // This contract inherits the `onlyOwner` modifier from
    // `owned` and applies it to the `destroy` function, which
    // causes that calls to `destroy` only have an effect if
    // they are made by the stored owner.
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // Modifiers can receive arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, destructible {
    mapping(address => bool) registeredAddresses;
    uint price;

    constructor(uint initialPrice) { price = initialPrice; }

    // It is important to also provide the
    // `payable` keyword here, otherwise the function will
    // automatically reject all Ether sent to it.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint price_) public onlyOwner {
        price = price_;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// This function is protected by a mutex, which means that
    /// reentrant calls from within `msg.sender.call` cannot call `f` again.
    /// The `return 7` statement assigns 7 to the return value but still
    /// executes the statement `locked = false` in the modifier.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

If you want to access a modifier `m` defined in a contract `C`, you can use `C.m` to reference it without virtual lookup. It is only possible to use modifiers defined in the current contract or its base contracts. Modifiers can also be defined in libraries but their use is limited to functions of the same library.

Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.

Modifiers cannot implicitly access or change the arguments and return values of functions they modify. Their values can only be passed to them explicitly at the point of invocation.

In function modifiers, it is necessary to specify when you want the function to which the modifier is applied to be run. The placeholder statement (denoted by a single underscore character _) is used to denote where the body of the function being modified should be inserted. Note that the placeholder operator is different from using underscores as leading or trailing characters in variable names, which is a stylistic choice.

Explicit returns from a modifier or function body only leave the current modifier or function body. Return variables are assigned and control flow continues after the _ in the preceding modifier.

**Advertencia:** In an earlier version of Solidity, `return` statements in functions having modifiers behaved differently.

An explicit return from a modifier with `return;` does not affect the values returned by the function. The modifier can, however, choose not to execute the function body at all and in that case the return variables are set to their *default values* just as if the function had an empty body.

The _ symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

### 3.9.4 Variables de estado constantes e inmutables

Las variables de estado pueden ser declaradas como `constant` o `immutable`. En ambos casos, estas variables ya no podrán ser modificadas una vez se haya creado el contrato.

Las variables `constant` fijarán su valor ya directamente en el propio proceso de compilación, mientras que las variables `immutable`, podrán hacerlo cuando el contrato sea construido.

También se pueden definir variables `constant` a nivel de archivo.

El compilador no reservará un espacio de almacenamiento (storage slot) para estas variables. En su lugar, cada una de estas variables será reemplazada por su respectivo valor.

En comparación con las variables de estado convencionales, el coste de gas de las variables constantes e inmutables es mucho más bajo. Una variable declarada como constante tiene un valor fijo, el cual es copiado directamente en todos los lugares que aparece o es accedido. Y gracias a esto, se obtiene una mayor optimización de los recursos computacionales a nivel local. En el caso de las variables declaradas como inmutables, dichas variables se evaluarán tan solo una única vez, justo cuando se realice la construcción del contrato. Será en ese momento, cuando se copiarán todos los respectivos valores, en todos aquellos lugares del código que existan referencias a estas variables. Para estos valores inmutables, se reservan siempre 32 bytes, incluso cuando no sea necesaria toda esta capacidad. Por este motivo, a veces, las variables constantes resultan más económicas que las variables inmutables.

Por ahora, no se soportan todos los tipos de datos para estas variables constantes e inmutables. Únicamente se soportan los tipos *strings* (solo para constantes) y *value types*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;
```

(continué en la próxima página)

(proviene de la página anterior)

```
contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint decimals_, address ref) {
        decimals = decimals_;
        // Las asignaciones a variables inmutables también pueden tener acceso a datos␣
↪de su entorno.
        maxBalance = ref.balance;
    }

    function isBalanceTooHigh(address other) public view returns (bool) {
        return other.balance > maxBalance;
    }
}
```

### Constantes

En el caso de las variables `constant`, el valor tiene que ser una constante en tiempo de compilación, y tiene que ser asignado en cada lugar donde las variables sean declaradas. Cualquier expresión que acceda al almacenamiento, información de la blockchain (por ejemplo, `block.timestamp`, `address(this).balance` o `block.number`) o datos de ejecución (`msg.value` o `gasleft()`) o llamadas hechas a contratos externos, son expresiones no permitidas. Las expresiones con efectos secundarios en las asignaciones de memoria están permitidas, pero las que tengan efectos secundarios sobre los objetos de la memoria no. Las funciones integradas (built-in functions) `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` y `mulmod` están permitidas (e incluso, a excepción de `keccak256`, aunque hagan llamadas a contratos externos).

El motivo por el cual se permiten efectos secundarios sobre las asignaciones de memoria, es que sea posible construir objetos complejos como, por ejemplo, lookup-tables. Aunque esta característica todavía no está totalmente disponible para ser usada.

### Inmutables

Las variables declaradas como `immutable` son un poco menos restrictivas que las declaradas como `constant`: Las variables inmutables pueden tener un valor arbitrario en el constructor del contracto o en el lugar de su declaración. Eso sí, su valor solo puede ser asignado una vez. Pero a partir de ahí, ese valor puede leerse incluso durante el proceso de construcción.

El código de creación del contrato el cual es generado por el compilador, será modificado en tiempo de ejecución, y reemplazará todas las referencias a variables inmutables por sus correspondientes valores asignados en cada caso. Esto es importante a la hora de comparar el código que se usa en tiempo de ejecución, y el cual está generado por el compilador, respecto del código que finalmente permanece alojado en la blockchain.

---

**Nota:** Las variables inmutables que sean asignadas al ser declaradas solo se considerarán inicializadas una vez sea ejecutado el constructor del contrato. Esto implica que no puedes inicializar inmutables en línea con un valor el cual dependa de otra variable inmutable. Sin embargo, sí que puedes hacerlo dentro del constructor del contrato.

Esto evita que existan diferentes interpretaciones del código, en relación al orden de inicialización de las variables de estado y la ejecución del constructor, especialmente cuando hay casos de herencia de valores.

### 3.9.5 Functions

Functions can be defined inside and outside of contracts.

Functions outside of a contract, also called «free functions», always have implicit `internal` *visibility*. Their code is included in all contracts that call them, similar to internal library functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

function sum(uint[] memory arr) pure returns (uint s) {
    for (uint i = 0; i < arr.length; i++)
        s += arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory arr) public {
        // This calls the free function internally.
        // The compiler will add its code to the contract.
        uint s = sum(arr);
        require(s >= 10);
        found = true;
    }
}
```

**Nota:** Functions defined outside a contract are still always executed in the context of a contract. They still can call other contracts, send them Ether and destroy the contract that called them, among other things. The main difference to functions defined inside a contract is that free functions do not have direct access to the variable `this`, storage variables and functions not in their scope.

#### Function Parameters and Return Variables

Functions take typed parameters as input and may, unlike in many other languages, also return an arbitrary number of values as output.

#### Function Parameters

Function parameters are declared the same way as variables, and the name of unused parameters can be omitted.

For example, if you want your contract to accept one kind of external call with two integers, you would use something like the following:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;
```

(continué en la próxima página)

(proviene de la página anterior)

```
contract Simple {
    uint sum;
    function taker(uint a, uint b) public {
        sum = a + b;
    }
}
```

Function parameters can be used as any other local variable and they can also be assigned to.

### Return Variables

Function return variables are declared with the same syntax after the `returns` keyword.

For example, suppose you want to return two results: the sum and the product of two integers passed as function parameters, then you use something like:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        sum = a + b;
        product = a * b;
    }
}
```

The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their *default value* and have that value until they are (re-)assigned.

You can either explicitly assign to return variables and then leave the function as above, or you can provide return values (either a single or *multiple ones*) directly with the `return` statement:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        return (a + b, a * b);
    }
}
```

If you use an early `return` to leave a function that has return variables, you must provide return values together with the return statement.

**Nota:** You cannot return some types from non-internal functions. This includes the types listed below and any composite types that recursively contain them:

- mappings,
- internal function types,
- reference types with location set to `storage`,
- multi-dimensional arrays (applies only to *ABI coder v1*),
- structs (applies only to *ABI coder v1*).

This restriction does not apply to library functions because of their different *internal ABI*.

### Returning Multiple Values

When a function has multiple return types, the statement `return (v0, v1, ..., vn)` can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an *implicit conversion*.

### State Mutability

### View Functions

Functions can be declared `view` in which case they promise not to modify the state.

**Nota:** If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used when `view` functions are called, which enforces the state to stay unmodified as part of the EVM execution. For library `view` functions `DELEGATECALL` is used, because there is no combined `DELEGATECALL` and `STATICCALL`. This means library `view` functions do not have run-time checks that prevent state modifications. This should not impact security negatively because library code is usually known at compile-time and the static checker performs compile-time checks.

The following statements are considered modifying the state:

1. Writing to state variables.
2. *Emitting events*.
3. *Creating other contracts*.
4. Using `selfdestruct`.
5. Sending Ether via calls.
6. Calling any function not marked `view` or `pure`.
7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + block.timestamp;
    }
}
```

**Nota:** `constant` on functions used to be an alias to `view`, but this was dropped in version 0.5.0.

**Nota:** Getter methods are automatically marked `view`.

**Nota:** Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `view` functions. This enabled state modifications in `view` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `view` functions, modifications to the state are prevented on the level of the EVM.

### Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state. In particular, it should be possible to evaluate a `pure` function at compile-time given only its inputs and `msg.data`, but without any knowledge of the current blockchain state. This means that reading from `immutable` variables can be a non-pure operation.

**Nota:** If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used, which does not guarantee that the state is not read, but at least that it is not modified.

In addition to the list of state modifying statements explained above, the following are considered reading from the state:

1. Reading from state variables.
2. Accessing `address(this).balance` or `<address>.balance`.
3. Accessing any of the members of `block`, `tx`, `msg` (with the exception of `msg.sig` and `msg.data`).
4. Calling any function not marked `pure`.
5. Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

Pure functions are able to use the `revert()` and `require()` functions to revert potential state changes when an *error occurs*.

Reverting a state change is not considered a «state modification», as only changes to the state made previously in code that did not have the `view` or `pure` restriction are reverted and that code has the option to catch the `revert` and not pass it on.

This behaviour is also in line with the `STATICCALL` opcode.

**Advertencia:** It is not possible to prevent functions from reading the state at the level of the EVM, it is only possible to prevent them from writing to the state (i.e. only `view` can be enforced at the EVM level, `pure` can not).

**Nota:** Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `pure` functions. This enabled state modifications in `pure` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `pure` functions, modifications to the state are prevented on the level of the EVM.

**Nota:** Prior to version 0.4.17 the compiler did not enforce that `pure` is not reading the state. It is a compile-time type check, which can be circumvented doing invalid explicit conversions between contract types, because the compiler can verify that the type of the contract does not do state-changing operations, but it cannot check that the contract that will be called at runtime is actually of that type.

### Special Functions

#### Receive Ether Function

A contract can have at most one `receive` function, declared using `receive() external payable { ... }` (without the `function` keyword). This function cannot have arguments, cannot return anything and must have `external` visibility and `payable` state mutability. It can be virtual, can override and can have modifiers.

The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers (e.g. via `.send()` or `.transfer()`). If no such function exists, but a payable *fallback function* exists, the fallback function will be called on a plain Ether transfer. If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through a transaction that does not represent a payable function call and throws an exception.

In the worst case, the `receive` function can only rely on 2300 gas being available (for example when `send` or `transfer` is used), leaving little room to perform other operations except basic logging. The following operations will consume more gas than the 2300 gas stipend:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

**Advertencia:** When Ether is sent directly to a contract (without a function call, i.e. sender uses `send` or `transfer`) but the receiving contract does not define a receive Ether function or a payable fallback function, an exception will be thrown, sending back the Ether (this was different before Solidity v0.4.0). If you want your contract to receive Ether, you have to implement a receive Ether function (using payable fallback functions for receiving Ether is not recommended, since the fallback is invoked and would not fail for interface confusions on the part of the sender).

**Advertencia:** A contract without a receive Ether function can receive Ether as a recipient of a *coinbase transaction* (aka *miner block reward*) or as a destination of a `selfdestruct`.

A contract cannot react to such Ether transfers and thus also cannot reject them. This is a design choice of the EVM and Solidity cannot work around it.

It also means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

Below you can see an example of a Sink contract that uses function `receive`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

### Fallback Function

A contract can have at most one `fallback` function, declared using either `fallback () external [payable]` or `fallback (bytes calldata input) external [payable] returns (bytes memory output)` (both without the `function` keyword). This function must have `external` visibility. A fallback function can be virtual, can override and can have modifiers.

The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no *receive Ether function*. The fallback function always receives data, but in order to also receive Ether it must be marked `payable`.

If the version with parameters is used, `input` will contain the full data sent to the contract (equal to `msg.data`) and can return data in `output`. The returned data will not be ABI-encoded. Instead it will be returned without modifications (not even padding).

In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available (see *receive Ether function* for a brief description of the implications of this).

Like any function, the fallback function can execute complex operations as long as there is enough gas passed on to it.

**Advertencia:** A `payable` fallback function is also executed for plain Ether transfers, if no *receive Ether function* is present. It is recommended to always define a receive Ether function as well, if you define a payable fallback function to distinguish Ether transfers from interface confusions.

**Nota:** If you want to decode the input data, you can check the first four bytes for the function selector and then you can use `abi.decode` together with the array slice syntax to decode ABI-encoded data: `(c, d) = abi.decode(input[4:], (uint256, uint256));` Note that this should only be used as a last resort and proper functions should be used instead.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Test {
    uint x;
    // This function is called for all messages sent to
    // this contract (there is no other function).
    // Sending Ether to this contract will cause an exception,
    // because the fallback function does not have the `payable`
    // modifier.
    fallback() external { x = 1; }
}

contract TestPayable {
    uint x;
    uint y;
    // This function is called for all messages sent to
    // this contract, except plain Ether transfers
    // (there is no other function except the receive function).
    // Any call with non-empty calldata to this contract will execute
    // the fallback function (even if Ether is sent along with the call).
    fallback() external payable { x = 1; y = msg.value; }

    // This function is called for plain Ether transfers, i.e.
    // for every call with empty calldata.
    receive() external payable { x = 2; y = msg.value; }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1.

        // address(test) will not allow to call ``send`` directly, since ``test`` has no␣
↪payable
        // fallback function.
        // It has to be converted to the ``address payable`` type to even allow calling␣
↪``send`` on it.
        address payable testPayable = payable(address(test));

        // If someone sends Ether to that contract,
        // the transfer will fail, i.e. this returns false here.
        return testPayable.send(2 ether);
    }

    function callTestPayable(TestPayable test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1 and test.y becoming 0.
        (success,) = address(test).call{value: 1}(abi.encodeWithSignature(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1 and test.y becoming 1.

        // If someone sends Ether to that contract, the receive function in TestPayable␣
↪will be called.
        // Since that function writes to storage, it takes more gas than is available␣
↪with a
        // simple ``send`` or ``transfer``. Because of that, we have to use a low-level␣
↪call.
        (success,) = address(test).call{value: 2 ether}("");
        require(success);
        // results in test.x becoming == 2 and test.y becoming 2 ether.

        return true;
    }
}
```

### Function Overloading

A contract can have multiple functions of the same name but with different parameter types. This process is called «overloading» and also applies to inherited functions. The following example shows overloading of the function `f` in the scope of contract `A`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint value) public pure returns (uint out) {
        out = value;
    }

    function f(uint value, bool really) public pure returns (uint out) {
        if (really)
            out = value;
    }
}
```

Overloaded functions are also present in the external interface. It is an error if two externally visible functions differ by their Solidity types but not by their external types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

// This will not compile
contract A {
    function f(B value) public pure returns (B out) {
        out = value;
    }

    function f(address value) public pure returns (address out) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        out = value;
    }
}

contract B {
}
```

Both `f` function overloads above end up accepting the address type for the ABI although they are considered different inside Solidity.

### Overload resolution and Argument matching

Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call. Functions are selected as overload candidates if all arguments can be implicitly converted to the expected types. If there is not exactly one candidate, resolution fails.

**Nota:** Return parameters are not taken into account for overload resolution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint8 val) public pure returns (uint8 out) {
        out = val;
    }

    function f(uint256 val) public pure returns (uint256 out) {
        out = val;
    }
}
```

Calling `f(50)` would create a type error since `50` can be implicitly converted both to `uint8` and `uint256` types. On another hand `f(256)` would resolve to `f(uint256)` overload as `256` cannot be implicitly converted to `uint8`.

## 3.9.6 Events

Solidity events give an abstraction on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible (forever as of now, but this might change with Serenity). The Log and its event data is not accessible from within contracts (not even from the contract that created them).

It is possible to request a Merkle proof for logs, so if an external entity supplies a contract with such a proof, it can check that the log actually exists inside the blockchain. You have to supply block headers because the contract can only see the last 256 block hashes.

You can add the attribute `indexed` to up to three parameters which adds them to a special data structure known as *«topics»* instead of the data part of the log. A topic can only hold a single word (32 bytes) so if you use a *reference type*

for an indexed argument, the Keccak-256 hash of the value is stored as a topic instead.

All parameters without the `indexed` attribute are *ABI-encoded* into the data part of the log.

Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

For example, the code below uses the web3.js `subscribe("logs")` method to filter logs that match a topic with a certain address value:

```
var options = {
    fromBlock: 0,
    address: web3.eth.defaultAccount,
    topics: ["0x0000000000000000000000000000000000000000000000000000000000000000", null,␣
↪null]
};
web3.eth.subscribe('logs', options, function (error, result) {
    if (!error)
        console.log(result);
})
    .on("data", function (log) {
        console.log(log);
    })
    .on("changed", function (log) {
});
```

The hash of the signature of the event is one of the topics, except if you declared the event with the `anonymous` specifier. This means that it is not possible to filter for specific anonymous events by name, you can only filter by the contract address. The advantage of anonymous events is that they are cheaper to deploy and call. It also allows you to declare four indexed arguments rather than three.

**Nota:** Since the transaction log only stores the event data and not the type, you have to know the type of the event, including which parameter is indexed and if the event is anonymous in order to correctly interpret the data. In particular, it is possible to «fake» the signature of another event using an anonymous event.

### Members of Events

- `event.selector`: For non-anonymous events, this is a `bytes32` value containing the `keccak256` hash of the event signature, as used in the default topic.

### Example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract ClientReceipt {
    event Deposit(
        address indexed from,
        bytes32 indexed id,
        uint value
    );
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function deposit(bytes32 id) public payable {
        // Events are emitted using `emit`, followed by
        // the name of the event and the arguments
        // (if any) in parentheses. Any such invocation
        // (even deeply nested) can be detected from
        // the JavaScript API by filtering for `Deposit`.
        emit Deposit(msg.sender, id, msg.value);
    }
}
```

The use in the JavaScript API is as follows:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var depositEvent = clientReceipt.Deposit();

// watch for changes
depositEvent.watch(function(error, result){
    // result contains non-indexed arguments and topics
    // given to the `Deposit` call.
    if (!error)
        console.log(result);
});


// Or pass a callback to start watching immediately
var depositEvent = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

The output of the above looks like the following (trimmed):

```
{
   "returnValues": {
       "from": "0x1111...FFFFCCCC",
       "id": "0x50...sd5adb20",
       "value": "0x420042"
   },
   "raw": {
       "data": "0x7f...91385",
       "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
   }
}
```

#### Additional Resources for Understanding Events

- Javascript documentation
- Example usage of events
- How to access them in js

### 3.9.7 Errores e Instrucción Revert

Los errores en Solidity proporcionan una forma conveniente y eficiente en gas de explicar al usuario por qué ha fallado una operación. Se pueden definir dentro y fuera de los contratos (incluidas las interfaces y bibliotecas).

Deben utilizarse junto con la instrucción *revert statement* que hace que se reviertan todos los cambios en la llamada actual y que los datos de error se devuelvan al llamador.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Insufficient balance for transfer. Needed `required` but only
/// `available` available.
/// @param available balance available.
/// @param required requested amount to transfer.
error InsufficientBalance(uint256 available, uint256 required);

contract TestToken {
    mapping(address => uint) balance;
    function transfer(address to, uint256 amount) public {
        if (amount > balance[msg.sender])
            revert InsufficientBalance({
                available: balance[msg.sender],
                required: amount
            });
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    // ...
}
```

Los errores no se pueden sobrecargar ni anular, pero se heredan. El mismo error se puede definir en varios lugares, siempre y cuando los ámbitos sean distintos. Las instancias de errores solo se pueden crear utilizando instrucciones `revert`.

El error crea datos que luego se pasan al llamador con la operación de reversión para volver al componente fuera de la cadena o capturarlo en una instrucción *try/catch*. Tenga en cuenta que un error solo se puede detectar cuando proviene de una llamada externa, las reversiones que ocurren en llamadas internas o dentro de la misma función no se pueden capturar.

Si no proporciona ningún parámetro, el error solo necesita cuatro bytes de datos y puede utilizar *NatSpec* como se indica anteriormente para explicar más a fondo las razones del error, que no se almacena en la cadena. Esto hace que esta sea una función de informe de errores muy barata y conveniente al mismo tiempo.

Más específicamente, una instancia de error está codificada en ABI de la misma manera que sería una llamada a una función del mismo nombre y tipos y luego utilizada como los datos devueltos en el opcode `revert`. Esto significa que los datos consisten en un selector de 4 bytes seguido por datos de *ABI-encoded*. El selector consiste en los primeros 4 bytes del hash keccak256 de la firma del tipo de error.

**Nota:** Es posible que un contrato se revierta con diferentes errores del mismo nombre o incluso con errores definidos en diferentes lugares que no son identificables por el llamante. Para el exterior, es decir, el ABI, sólo el nombre del error es relevante, no el contrato o el archivo donde está definido.

La sentencia `require(condition, "description");` sería equivalente a `if (!condition) revert Error("description")` si pudiera definir `error Error(string)`. Tenga en cuenta, sin embargo, que `Error` es un tipo integrado y no se puede definir en código proporcionado por el usuario.

De manera similar, un `assert` o condiciones similares se revertirán con un error del tipo integrado `Panic(uint256)`

**Nota:** Los datos de error sólo se deben utilizar para indicar un fallo, pero no como un medio para el control de flujo. El motivo es que los datos de reversión de llamadas internas se propaga de vuelta a través de la cadena de llamadas externas de forma predeterminada. Esto significa que una llamada interna puede "forjar" datos de reversión que parecen haber venido del contrato que lo llamó.

#### Miembros de Errores

- `error.selector`: Un valor de `bytes4` que contiene el selector de errores.

### 3.9.8 Inheritance

Solidity supports multiple inheritance including polymorphism.

Polymorphism means that a function call (internal and external) always executes the function of the same name (and parameter types) in the most derived contract in the inheritance hierarchy. This has to be explicitly enabled on each function in the hierarchy using the `virtual` and `override` keywords. See *Function Overriding* for more details.

It is possible to call functions further up in the inheritance hierarchy internally by explicitly specifying the contract using `ContractName.functionName()` or using `super.functionName()` if you want to call the function one level higher up in the flattened inheritance hierarchy (see below).

When a contract inherits from other contracts, only a single contract is created on the blockchain, and the code from all the base contracts is compiled into the created contract. This means that all internal calls to functions of base contracts also just use internal function calls (`super.f(..)` will use JUMP and not a message call).

State variable shadowing is considered as an error. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

The general inheritance system is very similar to Python's, especially concerning multiple inheritance, but there are also some *differences*.

Details are given in the following example.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// This will report a warning due to deprecated selfdestruct

contract Owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Use `is` to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract Destructible is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


// These abstract contracts are only provided to make the
// interface known to the compiler. Note the function
// without body. If a contract does not implement all
// functions it can only be used as an interface.
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}


abstract contract NameReg {
    function register(bytes32 name) public virtual;
    function unregister() public virtual;
}


// Multiple inheritance is possible. Note that `Owned` is
// also a base class of `Destructible`, yet there is only a single
// instance of `Owned` (as for virtual inheritance in C++).
contract Named is Owned, Destructible {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Functions can be overridden by another function with the same name and
    // the same number/types of inputs.  If the overriding function has different
    // types of output parameters, that causes an error.
    // Both local and message-based function calls take these overrides
    // into account.
    // If you want the function to override, you need to use the
    // `override` keyword. You need to specify the `virtual` keyword again
    // if you want this function to be overridden again.
    function destroy() public virtual override {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // It is still possible to call a specific
            // overridden function.
```

(continué en la próxima página)

(proviene de la página anterior)

```
            Destructible.destroy();
        }
    }
}


// If a constructor takes an argument, it needs to be
// provided in the header or modifier-invocation-style at
// the constructor of the derived contract (see below).
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // Here, we only specify `override` and not `virtual`.
    // This means that contracts deriving from `PriceFeed`
    // cannot change the behaviour of `destroy` anymore.
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

Note that above, we call `Destructible.destroy()` to «forward» the destruction request. The way this is done is problematic, as seen in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// This will report a warning due to deprecated selfdestruct

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

contract Destructible is owned {
    function destroy() public virtual {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ Destructible.
↪destroy(); }
}

contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ Destructible.
↪destroy(); }
}

contract Final is Base1, Base2 {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function destroy() public override(Base1, Base2) { Base2.destroy(); }
}
```

A call to `Final.destroy()` will call `Base2.destroy` because we specify it explicitly in the final override, but this function will bypass `Base1.destroy`. The way around this is to use `super`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// This will report a warning due to deprecated selfdestruct

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

contract Destructible is owned {
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ super.destroy(); }
}


contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ super.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { super.destroy(); }
}
```

If `Base2` calls a function of `super`, it does not simply call this function on one of its base contracts. Rather, it calls this function on the next base contract in the final inheritance graph, so it will call `Base1.destroy()` (note that the final inheritance sequence is – starting with the most derived contract: Final, Base2, Base1, Destructible, owned). The actual function that is called when using super is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

### Function Overriding

Base functions can be overridden by inheriting contracts to change their behavior if they are marked as `virtual`. The overriding function must then use the `override` keyword in the function header. The overriding function may only change the visibility of the overridden function from `external` to `public`. The mutability may be changed to a more strict one following the order: `nonpayable` can be overridden by `view` and `pure`. `view` can be overridden by `pure`. `payable` is an exception and cannot be changed to any other mutability.

The following example demonstrates changing mutability and visibility:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
```

(continué en la próxima página)

(proviene de la página anterior)

```
contract Base
{
    function foo() virtual external view {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() override public pure {}
}
```

For multiple inheritance, the most derived base contracts that define the same function must be specified explicitly after the `override` keyword. In other words, you have to specify all base contracts that define the same function and have not yet been overridden by another base contract (on some path through the inheritance graph). Additionally, if a contract inherits the same function from multiple (unrelated) bases, it has to explicitly override it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
    // Derives from multiple bases defining foo(), so we must explicitly
    // override it
    function foo() public override(Base1, Base2) {}
}
```

An explicit override specifier is not required if the function is defined in a common base contract or if there is a unique function in a common base contract that already overrides all other functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// No explicit override required
contract D is B, C {}
```

More formally, it is not required to override a function (directly or indirectly) inherited from multiple bases if there is a base contract that is part of all override paths for the signature, and (1) that base implements the function and no paths from the current contract to the base mentions a function with that signature or (2) that base does not implement the

function and there is at most one mention of the function in all paths from the current contract to that base.

In this sense, an override path for a signature is a path through the inheritance graph that starts at the contract under consideration and ends at a contract mentioning a function with that signature that does not override.

If you do not mark a function that overrides as `virtual`, derived contracts can no longer change the behaviour of that function.

**Nota:** Functions with the `private` visibility cannot be `virtual`.

**Nota:** Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered `virtual`.

**Nota:** Starting from Solidity 0.8.8, the `override` keyword is not required when overriding an interface function, except for the case where the function is defined in multiple bases.

Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A
{
    function f() external view virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

**Nota:** While public state variables can override external functions, they themselves cannot be overridden.

### Modifier Overriding

Function modifiers can override each other. This works in the same way as *function overriding* (except that there is no overloading for modifiers). The `virtual` keyword must be used on the overridden modifier and the `override` keyword must be used in the overriding modifier:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base
{
    modifier foo() virtual {_;}
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
contract Inherited is Base
{
    modifier foo() override {_;}
}
```

In case of multiple inheritance, all direct base contracts must be specified explicitly:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    modifier foo() virtual {_;}
}

contract Base2
{
    modifier foo() virtual {_;}
}

contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

### Constructors

A constructor is an optional function declared with the `constructor` keyword which is executed upon contract creation, and where you can run contract initialisation code.

Before the constructor code is executed, state variables are initialised to their specified value if you initialise them inline, or their *default value* if you do not.

After the constructor has run, the final code of the contract is deployed to the blockchain. The deployment of the code costs additional gas linear to the length of the code. This code includes all functions that are part of the public interface and all functions that are reachable from there through function calls. It does not include the constructor code or internal functions that are only called from the constructor.

If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() {}`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint a_) {
        a = a_;
    }
}

contract B is A(1) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    constructor() {}
}
```

You can use internal parameters in a constructor (for example storage pointers). In this case, the contract has to be marked *abstract*, because these parameters cannot be assigned valid values from outside but only through the constructors of derived contracts.

**Advertencia:** Prior to version 0.4.22, constructors were defined as functions with the same name as the contract. This syntax was deprecated and is not allowed anymore in version 0.5.0.

**Advertencia:** Prior to version 0.7.0, you had to specify the visibility of constructors as either `internal` or `public`.

### Arguments for Base Constructors

The constructors of all the base contracts will be called following the linearization rules explained below. If the base constructors have arguments, derived contracts need to specify all of them. This can be done in two ways:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base {
    uint x;
    constructor(uint x_) { x = x_; }
}

// Either directly specify in the inheritance list...
contract Derived1 is Base(7) {
    constructor() {}
}

// or through a "modifier" of the derived constructor...
contract Derived2 is Base {
    constructor(uint y) Base(y * y) {}
}

// or declare abstract...
abstract contract Derived3 is Base {
}

// and have the next concrete derived contract initialize it.
contract DerivedFromDerived is Derived3 {
    constructor() Base(10 + 10) {}
}
```

One way is directly in the inheritance list (`is Base(7)`). The other is in the way a modifier is invoked as part of the derived constructor (`Base(y * y)`). The first way to do it is more convenient if the constructor argument is a constant and defines the behaviour of the contract or describes it. The second way has to be used if the constructor arguments of the base depend on those of the derived contract. Arguments have to be given either in the inheritance list or in modifier-style in the derived constructor. Specifying arguments in both places is an error.

If a derived contract does not specify the arguments to all of its base contracts" constructors, it must be declared abstract. In that case, when another contract derives from it, that other contract's inheritance list or constructor must provide the necessary parameters for all base classes that haven't had their parameters specified (otherwise, that other contract must be declared abstract as well). For example, in the above code snippet, see `Derived3` and `DerivedFromDerived`.

### Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity is similar to Python in that it uses «C3 Linearization» to force a specific order in the directed acyclic graph (DAG) of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the `is` directive is important: You have to list the direct base contracts in the order from «most base-like» to «most derived». Note that this order is the reverse of the one used in Python.

Another simplifying way to explain this is that when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in Python) in a depth-first manner, stopping at the first match. If a base contract has already been searched, it is skipped.

In the following code, Solidity will give the error «Linearization of inheritance graph impossible».

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract X {}
contract A is X {}
// This will not compile
contract C is A, X {}
```

The reason for this is that `C` requests `X` to override `A` (by specifying `A, X` in this order), but `A` itself requests to override `X`, which is a contradiction that cannot be resolved.

Due to the fact that you have to explicitly override a function that is inherited from multiple bases without a unique override, C3 linearization is not too important in practice.

One area where inheritance linearization is especially important and perhaps not as clear is when there are multiple constructors in the inheritance hierarchy. The constructors will always be executed in the linearized order, regardless of the order in which their arguments are provided in the inheriting contract's constructor. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base1 {
    constructor() {}
}

contract Base2 {
    constructor() {}
}

// Constructors are executed in the following order:
//  1 - Base1
//  2 - Base2
//  3 - Derived1
contract Derived1 is Base1, Base2 {
    constructor() Base1() Base2() {}
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Constructors are executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() Base2() Base1() {}
}

// Constructors are still executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived3
contract Derived3 is Base2, Base1 {
    constructor() Base1() Base2() {}
}
```

#### Inheriting Different Kinds of Members of the Same Name

**It is an error when any of the following pairs in a contract have the same name due to inheritance:**

- a function and a modifier
- a function and an event
- an event and a modifier

As an exception, a state variable getter can override an external function.

### 3.9.9 Contratos Abstractos

Los contratos deben marcarse como abstractos cuando al menos una de sus funciones no está implementada o cuando no proporcionan argumentos para todos los constructores en los contratos base. Incluso si este no es el caso, un contrato aún puede marcarse como abstracto cuando no tiene la intención de que ser creado directamente. Los contratos abstractos son similares a las *Interfaces*, sin embargo, una interfaz está más limitada en lo que puede declarar.

Un contrato abstracto se declara utilizando la palabra clave `abstract`, como se muestra en el siguiente ejemplo. Se debe tener en cuenta que este contrato debe definirse como abstracto, porque se declara la función `utterance()`, pero no se proporcionó ninguna implementación (no se proporcionó ninguna implementación dentro del cuerpo de la función `{ }`).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}
```

Dichos contratos abstractos no pueden instanciarse directamente. Esto también es cierto, si un contrato abstracto en sí mismo implementa todas las funciones definidas. El uso de un contrato abstracto como clase base se muestra en el siguiente ejemplo:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public pure virtual returns (bytes32);
}

contract Cat is Feline {
    function utterance() public pure override returns (bytes32) { return "miaow"; }
}
```

Si un contrato hereda partes de un contrato abstracto y no implementa todas las funciones que no fueron implementadas mediante anulación, debe marcarse como abstracto.

Se debe tener en cuenta que una función sin implementación es diferente de una *Función Type*, aunque su sintaxis sea muy similar.

Ejemplo de función sin implementación (una declaración de función):

```
function foo(address) external returns (address);
```

Ejemplo de declaración de una variable cuyo tipo es un tipo función:

```
function(address) external returns (address) foo;
```

Los contratos abstractos desacoplan la definición de un contrato de su implementación, proporcionando una mejor extensibilidad y autodocumentación, facilitando patrones como el método de plantilla y eliminando la duplicación de código. Los contratos abstractos son útiles de la misma manera que lo es definir métodos en una interfaz. Es una forma de que el diseñador del contrato abstracto diga «cualquier hijo mío debe implementar este método».

**Nota:** Los contratos abstractos no pueden anular una función virtual implementada con una no implementada.

### 3.9.10 Interfaces

Las interfaces son similares a los contratos abstractos, pero no pueden tener funciones implementadas. Cuentan con más restricciones:

- No pueden heredar de otros contratos, pero pueden heredar de otras interfaces.
- Todas las funciones declaradas deben ser externas en la interfaz, incluso si son públicas en el contrato.
- No pueden declarar un constructor.
- No pueden declarar variables de estado.
- No pueden declarar modificadores.

Algunas de estas restricciones podrían dejar de aplicarse en un futuro.

Las interfaces se limitan básicamente a lo que puede representar el ABI del contrato. La conversión entre el ABI y una interfaz debería ser posible sin ninguna pérdida de información.

Las interfaces se indican con su propia palabra clave:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Los contratos pueden heredar interfaces como heredarían otros contratos.

Todas las funciones declaradas en las interfaces son implícitamente `virtual` y cualquier función que las invalide no necesita la palabra clave `override`. Esto no significa automáticamente que una función de anulación se pueda anular de nuevo; esto solamente es posible si la función de anulación está marcada como `virtual`.

Las interfaces pueden heredar de otras interfaces. Aplican las mismas reglas de una herencia normal.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Must redefine test in order to assert that the parent
    // meanings are compatible.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Se puede acceder a los tipos definidos dentro de las interfaces y otras estructuras similares a contratos desde otros contratos: `Token.TokenType` o `Token.Coin`.

### 3.9.11 Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the `DELEGATECALL` (`CALLCODE` until Homestead) feature of the EVM. This means that if library functions are called, their code is executed in the context of the calling contract, i.e. `this` points to the calling contract, and especially the storage from the calling contract can be accessed. As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise). Library functions can only be called directly (i.e. without the use of `DELEGATECALL`) if they do not modify the state (i.e. if they are `view` or `pure` functions), because libraries are assumed to be stateless. In particular, it is not possible to destroy a library.

---

**Nota:** Until version 0.4.20, it was possible to destroy libraries by circumventing Solidity's type system. Starting from that version, libraries contain a *mechanism* that disallows state-modifying functions to be called directly (i.e. without `DELEGATECALL`).

---

Libraries can be seen as implicit base contracts of the contracts that use them. They will not be explicitly visible in the inheritance hierarchy, but calls to library functions look just like calls to functions of explicit base contracts (using qualified access like `L.f()`). Of course, calls to internal functions use the internal calling convention, which means that all internal types can be passed and types *stored in memory* will be passed by reference and not copied. To realize this in the EVM, the code of internal library functions that are called from a contract and all functions called from therein will at compile time be included in the calling contract, and a regular `JUMP` call will be used instead of a `DELEGATECALL`.

---

**Nota:** The inheritance analogy breaks down when it comes to public functions. Calling a public library function with `L.f()` results in an external call (`DELEGATECALL` to be precise). In contrast, `A.f()` is an internal call when `A` is a base contract of the current contract.

---

The following example illustrates how to use libraries (but using a manual method, be sure to check out *using for* for a more advanced example to implement a set).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;


// We define a new struct datatype that will be used to
// hold its data in the calling contract.
struct Data {
    mapping(uint => bool) flags;
}

library Set {
    // Note that the first parameter is of type "storage
    // reference" and thus only its storage address and not
    // its contents is passed as part of the call.  This is a
    // special feature of library functions.  It is idiomatic
    // to call the first parameter `self`, if the function can
    // be seen as a method of that object.
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
        if (self.flags[value])
            return false; // already there
        self.flags[value] = true;
        return true;
    }

    function remove(Data storage self, uint value)
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false; // not there
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
```

(continué en la próxima página)

(proviene de la página anterior)

```
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}


contract C {
    Data knownValues;

    function register(uint value) public {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        require(Set.insert(knownValues, value));
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}
```

Of course, you do not have to follow this way to use libraries: they can also be used without defining struct data types. Functions also work without any storage reference parameters, and they can have multiple storage reference parameters and in any position.

The calls to `Set.contains`, `Set.insert` and `Set.remove` are all compiled as calls (`DELEGATECALL`) to an external contract/library. If you use libraries, be aware that an actual external function call is performed. `msg.sender`, `msg.value` and `this` will retain their values in this call, though (prior to Homestead, because of the use of `CALLCODE`, `msg.sender` and `msg.value` changed, though).

The following example shows how to use *types stored in memory* and internal functions in libraries in order to implement custom types without the overhead of external function calls:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

struct bigint {
    uint[] limbs;
}

library BigInt {
    function fromUint(uint x) internal pure returns (bigint memory r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint memory a, bigint memory b) internal pure returns (bigint memory␣
↪r) {
        r.limbs = new uint[](max(a.limbs.length, b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint limbA = limb(a, i);
            uint limbB = limb(b, i);
```

(continué en la próxima página)

(proviene de la página anterior)

```
            unchecked {
                r.limbs[i] = limbA + limbB + carry;

                if (limbA + limbB < limbA || (limbA + limbB == type(uint).max && carry >␣
↪0))
                    carry = 1;
                else
                    carry = 0;
            }
        }
        if (carry > 0) {
            // too bad, we have to add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            uint i;
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint memory a, uint index) internal pure returns (uint) {
        return index < a.limbs.length ? a.limbs[index] : 0;
    }

    function max(uint a, uint b) private pure returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
    using BigInt for bigint;

    function f() public pure {
        bigint memory x = BigInt.fromUint(7);
        bigint memory y = BigInt.fromUint(type(uint).max);
        bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}
```

It is possible to obtain the address of a library by converting the library type to the `address` type, i.e. using `address(LibraryName)`.

As the compiler does not know the address where the library will be deployed, the compiled hex code will contain placeholders of the form `__$30bbc0abd4d636451586595 0d3e0d10953$__`. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name, which would be for example `libraries/bigint.sol:BigInt` if the library was stored in a file called `bigint.sol` in a `libraries/` directory. Such bytecode is incomplete and should not be deployed. Placeholders need to be replaced with actual addresses. You can do that by either passing them to the compiler when the library is being compiled or by using the linker to update an already compiled binary. See *Library Linking* for information on how to use the commandline compiler for linking.

In comparison to contracts, libraries are restricted in the following ways:

- they cannot have state variables
- they cannot inherit nor be inherited
- they cannot receive Ether
- they cannot be destroyed

(These might be lifted at a later point.)

### Function Signatures and Selectors in Libraries

While external calls to public or external library functions are possible, the calling convention for such calls is considered to be internal to Solidity and not the same as specified for the regular *contract ABI*. External library functions support more argument types than external contract functions, for example recursive structs and storage pointers. For that reason, the function signatures used to compute the 4-byte selector are computed following an internal naming schema and arguments of types not supported in the contract ABI use an internal encoding.

The following identifiers are used for the types in the signatures:

- Value types, non-storage `string` and non-storage `bytes` use the same identifiers as in the contract ABI.
- Non-storage array types follow the same convention as in the contract ABI, i.e. `<type>[]` for dynamic arrays and `<type>[M]` for fixed-size arrays of `M` elements.
- Non-storage structs are referred to by their fully qualified name, i.e. `C.S` for `contract C { struct S { ... } }`.
- Storage pointer mappings use `mapping(<keyType> => <valueType>) storage` where `<keyType>` and `<valueType>` are the identifiers for the key and value types of the mapping, respectively.
- Other storage pointer types use the type identifier of their corresponding non-storage type, but append a single space followed by `storage` to it.

The argument encoding is the same as for the regular contract ABI, except for storage pointers, which are encoded as a `uint256` value referring to the storage slot to which they point.

Similarly to the contract ABI, the selector consists of the first four bytes of the Keccak256-hash of the signature. Its value can be obtained from Solidity using the `.selector` member as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

### Call Protection For Libraries

As mentioned in the introduction, if a library's code is executed using a `CALL` instead of a `DELEGATECALL` or `CALLCODE`, it will revert unless a `view` or `pure` function is called.

The EVM does not provide a direct way for a contract to detect whether it was called using `CALL` or not, but a contract can use the `ADDRESS` opcode to find out «where» it is currently running. The generated code compares this address to the address used at construction time to determine the mode of calling.

More specifically, the runtime code of a library always starts with a push instruction, which is a zero of 20 bytes at compilation time. When the deploy code runs, this constant is replaced in memory by the current address and this modified code is stored in the contract. At runtime, this causes the deploy time address to be the first constant to be pushed onto the stack and the dispatcher code compares the current address against this constant for any non-view and non-pure function.

This means that the actual code stored on chain for a library is different from the code reported by the compiler as `deployedBytecode`.

## 3.9.12 Using For

La directiva `using A for B` se puede utilizar para adjuntar funciones (`A`) como operadores a tipos de valor definidos por el usuario o como funciones miembro de cualquier tipo (`B`). Las funciones miembro reciben como primer parámetro el objeto al que se llama como primer parámetro (como la variable `self` en Python). Las funciones operador reciben operandos como parámetros.

Es válido tanto a nivel de fichero como dentro de un contrato, a nivel de contrato.

The first part, `A`, can be one of:

- Una lista de funciones, opcionalmente con un nombre de operador asignado (p. ej. `usando {f, g como +, h, L.t} para uint`). Si no se especifica ningún operador, la función puede ser una función de biblioteca o una función libre y se adjunta al tipo como función miembro. En caso contrario, debe ser una función libre y se convierte en la definición de ese operador en el tipo.
- El nombre de una biblioteca (por ejemplo, `usando L para uint`) - todas las funciones no privadas de la biblioteca se adjuntan al tipo como funciones miembro

A nivel de fichero, la segunda parte, `B`, tiene que ser un tipo explícito (sin especificador de ubicación de datos). Dentro de los contratos, también se puede utilizar `*` en lugar del tipo (por ejemplo, `usando L para *;`), que tiene el efecto de que todas las funciones de la biblioteca `L` se adjuntan a *todos* los tipos.

Si especifica una biblioteca, se adjuntan *todas* las funciones no privadas de la biblioteca, incluso aquellas en las que el tipo del primer parámetro no no coincide con el tipo del objeto. El tipo se comprueba en el en el momento en que se llama a la función y se de la función.

Si usas una lista de funciones (por ejemplo `usando {f, g, h, L.t} para uint`), entonces el tipo (`uint`) tiene que ser implícitamente convertible al primer parámetro de cada una de estas funciones. Esta comprobación se realiza realiza incluso si no se llama a ninguna de estas funciones. Tenga en cuenta que las funciones privadas de biblioteca sólo pueden especificarse cuando `using for` está dentro de una biblioteca

> Si defines un operador (por ejemplo `usando {f como +} para T`), entonces el tipo (`T`) debe ser un

*user-defined value type* y la definición debe ser una función `pure`. Las definiciones de operadores deben ser globales. Los siguientes operadores pueden definirse de esta forma:

| Category | Operator | Possible signatures |
|---|---|---|
| Bitwise | & | `function (T, T) pure returns (T)` |
| | \| | `function (T, T) pure returns (T)` |
| | ^ | `function (T, T) pure returns (T)` |
| | ~ | `function (T) pure returns (T)` |
| Aritmética | + | `function (T, T) pure returns (T)` |
| | - | `function (T, T) pure returns (T)` |
| | | `function (T) pure returns (T)` |
| | * | `function (T, T) pure returns (T)` |
| | / | `function (T, T) pure returns (T)` |
| | % | `function (T, T) pure returns (T)` |
| Comparación | == | `function (T, T) pure returns (bool)` |
| | != | `function (T, T) pure returns (bool)` |
| | < | `function (T, T) pure returns (bool)` |
| | <= | `function (T, T) pure returns (bool)` |
| | > | `function (T, T) pure returns (bool)` |
| | >= | `function (T, T) pure returns (bool)` |

Tenga en cuenta que unario y binario `-` necesitan definiciones separadas. El compilador elegirá la definición correcta en función de cómo se invoque el operador.

La directiva `usar A para B;` sólo está activa dentro del ámbito actual actual (ya sea el contrato o el módulo/unidad fuente actual), incluyendo dentro de todas sus funciones, y no tiene efecto fuera del contrato o módulo en el que se utiliza.

Cuando la directiva se utiliza a nivel de fichero y se aplica a un tipo definido por el usuario que se definió a nivel de archivo en el mismo archivo, puede añadirse al final la palabra `global`. Esto tendrá el efecto efecto que las funciones y operadores se adjuntan al tipo en todas partes el tipo esté disponible (incluyendo otros ficheros), no sólo en el ámbito de la sentencia «using».

Reescribamos el ejemplo de conjunto de la sección ref:*libraries* de esta manera, usando funciones a nivel de fichero en lugar de funciones de biblioteca.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

struct Data { mapping(uint => bool) flags; }
// Ahora adjuntamos funciones al tipo.
// Las funciones adjuntas pueden utilizarse en el resto del módulo.
// Si importa el módulo, tiene que
// repita allí la directiva using, por ejemplo como
//   import "flags.sol" as Flags;
//   using {Flags.insert, Flags.remove, Flags.contains}
//     for Flags.Data;
using {insert, remove, contains} for Data;

function insert(Data storage self, uint value)
    returns (bool)
{
    if (self.flags[value])
        return false; // ahí ya
    self.flags[value] = true;
    return true;
```

(continué en la próxima página)

(proviene de la página anterior)

```
}

function remove(Data storage self, uint value)
    returns (bool)
{
    if (!self.flags[value])
        return false; // ahí no
    self.flags[value] = false;
    return true;
}

function contains(Data storage self, uint value)
    view
    returns (bool)
{
    return self.flags[value];
}


contract C {
    Data knownValues;

    function register(uint value) public {
        // Aquí, todas las variables de tipo Datos tienen
        // funciones miembro correspondientes.
        // La siguiente llamada de función es idéntica a
        // `Set.insert(knownValues, value)`
        require(knownValues.insert(value));
    }
}
```

También es posible extender tipos incorporados de esa manera. En este ejemplo, utilizaremos una biblioteca.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return type(uint).max;
    }
}
using Search for uint[];

contract C {
    uint[] data;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function append(uint value) public {
        data.push(value);
    }

    function replace(uint from, uint to) public {
        // Esto realiza la llamada a la función de biblioteca
        uint index = data.indexOf(from);
        if (index == type(uint).max)
            data.push(to);
        else
            data[index] = to;
    }
}
```

Tenga en cuenta que todas las llamadas a bibliotecas externas son llamadas a funciones reales de EVM. Esto significa que si pasas memoria o tipos de valores, se realizará una copia, incluso en el caso de la variable `self`. La única situación en la que no se realizará ninguna copia es cuando se utilizan variables de referencia de almacenamiento o cuando se llaman funciones de la biblioteca.

Otro ejemplo muestra cómo definir un operador personalizado para un tipo definido por el usuario:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.19;

type UFixed16x2 is uint16;

using {
    add as +,
    div as /
} for UFixed16x2 global;

uint32 constant SCALE = 100;

function add(UFixed16x2 a, UFixed16x2 b) pure returns (UFixed16x2) {
    return UFixed16x2.wrap(UFixed16x2.unwrap(a) + UFixed16x2.unwrap(b));
}

function div(UFixed16x2 a, UFixed16x2 b) pure returns (UFixed16x2) {
    uint32 a32 = UFixed16x2.unwrap(a);
    uint32 b32 = UFixed16x2.unwrap(b);
    uint32 result32 = a32 * SCALE / b32;
    require(result32 <= type(uint16).max, "Divide overflow");
    return UFixed16x2.wrap(uint16(a32 * SCALE / b32));
}

contract Math {
    function avg(UFixed16x2 a, UFixed16x2 b) public pure returns (UFixed16x2) {
        return (a + b) / UFixed16x2.wrap(200);
    }
}
```

## 3.10 Ensamblado en línea

Puedes intercalar declaraciones de Solidity con ensamblado en línea en un lenguaje cercano al de la Máquina Virtual de Ethereum. Esto te brinda un control más preciso, especialmente útil cuando estás mejorando el lenguaje escribiendo librerías.

El lenguaje utilizado para el ensamblado en línea en Solidity se llama *Yul* y está documentado en su propia sección. Esta sección solo cubre cómo el código de ensamblado en línea puede interactuar con el código en Solidity que lo rodea.

**Advertencia:** El ensamblado en línea es una forma de acceder a la Máquina Virtual de Ethereum aun nivel bajo. Esto evita varias características y comprobaciones de seguridad importantes de Solidity. Solo debes usarlo para tareas que lo necesiten y solo si tienes confianza en su uso.

Un bloque de ensamblado en línea está marcado con `assembly { ... }`, donde el código dentro de las llaves es código en el lenguaje *Yul*.

El código de ensamblado en línea puede acceder variables locales de Solidity como se explica a continuación.

Los diferentes bloques de ensamblado en línea no comparten ningún espacio de nombres, por ejemplo, no es posible llamar a una función Yul o acceder a una variable Yul definida en un bloque de ensamblado en línea diferente.

### 3.10.1 Ejemplo

El siguiente ejemplo proporciona código de la librería para acceder al código de otro contrato y cargarlo en una variable `bytes`. Esto también es posible con «Solidity plano» usando `<address>.code`. Pero el punto aquí es que las librerías de ensamblado reutilizables pueden mejorar Solidity sin un cambio en el compilador.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library GetCode {
    function at(address addr) public view returns (bytes memory code) {
        assembly {
            // recuperar el tamaño del código, esto necesita ensamblaje
            let size := extcodesize(addr)
            // asignar el arreglo de bytes de salida, esto también podría hacerse sin␣
↪ensamblaje
            // utilizando code = new bytes(size)
            code := mload(0x40)
            // nuevo "fin de memoria" incluyendo relleno
            mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // almacenar longitud en memoria
            mstore(code, size)
            // recuperar realmente el código, esto necesita ensamblaje
            extcodecopy(addr, add(code, 0x20), 0, size)
        }
    }
}
```

El ensamblado en línea también es beneficioso en casos en los que el optimizador no logra producir código eficiente, por ejemplo:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;


library VectorSum {
    // Esta función es menos eficiente porque actualmente el optimizador
    // no puede eliminar las comprobaciones de límites en el acceso a arreglo.
    function sumSolidity(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i)
            sum += data[i];
    }

    // Sabemos que solo accedemos al arreglo dentro de los límites, por lo que podemos
    // evitar la comprobación.
    // 0x20 se debe añadir a un arreglo porque el primer espacio contiene la
    // longitud del arreglo.
    function sumAsm(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i) {
            assembly {
                sum := add(sum, mload(add(add(data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Al igual que el anterior, pero realizando todo el código dentro de ensamblado en␣
↪línea.
    function sumPureAsm(uint[] memory data) public pure returns (uint sum) {
        assembly {
            // Cargar la longitud (los primeros 32 bytes)
            let len := mload(data)

            // Saltar el campo de longitud.
            //
            // Mantener la variable temporal para que pueda ser incrementada en su lugar.
            //
            // NOTA: incrementar los datos resultaría en una variable de datos␣
↪inutilizable
            //       después de este bloque de ensamblado
            let dataElementLocation := add(data, 0x20)

            // Iterar hasta que no se cumpla el límite.
            for
                { let end := add(dataElementLocation, mul(len, 0x20)) }
                lt(dataElementLocation, end)
                { dataElementLocation := add(dataElementLocation, 0x20) }
            {
                sum := add(sum, mload(dataElementLocation))
            }
        }
    }
}
```

### 3.10.2 Acceso a variables, funciones y librerías externas

Puedes acceder a las variables y otros identificadores de Solidity utilizando su nombre.

Las variables locales de tipo valor son directamente utilizables en el ensamblado en línea. Ambas se pueden leer y asignar.

Las variables locales que hacen referencia a memoria evalúan la dirección de la variable en memoria, no el valor en sí. Tales variables también se pueden asignar, pero ten en cuenta que una asignación solo cambiará hacia donde apunta y no los datos, también que es tu responsabilidad respetar la gestión de memoria de Solidity. Ver *Convenciones en Solidity*.

Del mismo modo, las variables locales que hacen referencia a arreglos de calldata de tamaño estático o estructuras calldata evalúan la dirección de la variable en calldata, no el valor en sí. También se le puede asignar un nuevo offset a la variable, pero ten en cuenta que no se realiza ninguna validación para asegurar que la variable no apunte más allá de `calldatasize()`.

Para los punteros de función externos, la dirección y el selector de función pueden accederse usando `x.address` y `x.selector`. El selector consiste de cuatro bytes alineados a la derecha. Ambos valores se pueden asignar. Por ejemplo:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.10 <0.9.0;

contract C {
    // Asigna un nuevo selector y dirección a la variable de retorno @fun
    function combineToFunctionPointer(address newAddress, uint newSelector) public pure␣
↪returns (function() external fun) {
        assembly {
            fun.selector := newSelector
            fun.address  := newAddress
        }
    }
}
```

Para los arreglos dinámicos de calldata, puedes acceder a su offset de calldata (en bytes) y longitud (número de elementos) utilizando `x.offset` y `x.length`. Ambas expresiones también pueden ser asignadas, pero como en el caso estático, no se realizará ninguna validación para asegurarse que el área de datos resultante esté dentro de los límites de `calldatasize()`.

Para las variables de almacenamiento local, o variables de estado, un identificador único Yul no es suficiente ya que no necesariamente ocupan un solo espacio de almacenamiento completo. Por lo tanto, su «dirección» está compuesta por un espacio y un offset de bytes dentro del espacio. Para recuperar el espacio apuntado por la variable *x*, utiliza *x.slot* y para recuperar el offset de bytes utiliza *x.offset*. El uso de *x* en sí mismo resultará en un error.

También puedes asignar a la parte `.slot` de un puntero de variable de almacenamiento local. Para estos (estructuras, arreglos o mapeos), la parte `.offset` siempre es cero. Sin embargo, no es posible asignar a la parte `.slot` o `.offset` de una variable de estado.

Las variables locales en Solidity están disponibles para asignaciones, por ejemplo:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
```

(continué en la próxima página)

(proviene de la página anterior)

```
            // Ignoramos el desplazamiento de la ranura de almacenamiento,
            // sabemos que es cero en este caso especial.
            r := mul(x, sload(b.slot))
        }
    }
}
```

**Advertencia:** Si accedes a variables de un tipo que abarque menos de 256 bits (por ejemplo, `uint64`, `address`, o `bytes16`), no puedes hacer ninguna suposición acerca de bits que no son parte de la codificación del tipo. Especialmente, no debes suponer que son cero. Para estar seguro, siempre limpia de forma adecuada los datos antes de utilizarlos en un contexto en el que esto sea importante: `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` Para limpiar tipos firmados, puedes usar el código de operación: `assembly { signextend(<num_bytes_of_x_minus_one>, x) }`

Desde Solidity 0.6.0, puede que el nombre de una variable de ensamblado en línea no oculte ninguna declaración visible en el ámbito del bloque de ensamblado en línea (incluyendo declaraciones de variables, contratos y funciones).

Desde Solidity 0.7.0, puede que las variables y funciones declaradas dentro del bloque de ensamblado en línea no contengan `.`, pero usar `.` es válido para acceder a las variables de Solidity desde fuera del bloque de ensamblado en línea.

### 3.10.3 Cosas a evitar

El ensamblado en línea puede tener un aspecto de alto nivel, pero en realidad es bastante de bajo nivel. Las llamadas a funciones, bucles, condiciones y conmutadores se convierten mediante reglas de reescritura simples, después de eso lo único que hace el ensamblador por ti es reorganizar las instrucciones de estilo funcional de los códigos de operación, contar la altura de la pila para acceder a las variables y eliminar los espacios de la pila de las variables locales al ensamblado cuando se alcanza el final de su bloque.

### 3.10.4 Convenciones en Solidity

#### Valores de variables con tipo

En contraste con el ensamblado EVM, Solidity tiene tipos más estrechos que 256 bits, como `uint24`. Por eficiencia, la mayoría de operaciones aritméticas ignoran el hecho de que los tipos pueden ser más cortos que 256 bits y se limpian los bits de mayor orden cuando es necesario, es decir, poco antes de que sean escritas en la memoria o antes de realizar comparaciones. Esto significa que si accedes a una variable de este tipo desde el ensamblado en línea, es posible que primero tengas que limpiar manualmente los bits de mayor orden.

### Gestión de memoria

Solidity maneja la memoria de la siguiente forma. Hay un «puntero de memoria libre» en la posición `0x40` de la memoria. Si quieres asignar memoria, usa la memoria a partir de donde apunta ese puntero y actualiza el mismo. No hay garantía de que la memoria no haya sido utilizada anteriormente y por lo tanto no puedes asumir que sean bytes en cero. No hay un mecanismo incorporado para soltar o liberar memoria asignada. Aquí tienes un fragmento de ensamblado que puedes usar para asignar memoria siguiendo el proceso descrito anteriormente:

```
function allocate(length) -> pos {
  pos := mload(0x40)
  mstore(0x40, add(pos, length))
}
```

Los primeros 64 bytes de memoria pueden usarse como «espacio temporal» para asignaciones a corto plazo. Los siguientes 32 bytes tras el puntero de memoria libre (es decir, a partir de `0x60`) están destinados a ser cero permanentemente y se usan como valor inicial para los arreglos de memoria dinámica vacíos. Esto significa que la memoria asignable comienza en `0x80`, que es el valor inicial del puntero de memoria libre.

Los elementos en los arreglos de memoria de Solidity siempre ocupan múltiplos de 32 bytes (esto es cierto también para `bytes1[]`, pero no para `bytes` y `string`). Los arreglos de memoria multidimensionales son punteros a arreglos de memoria. La longitud de un arreglo dinámico se almacena en el primer espacio del arreglo que le sigue, seguido por los elementos del arreglo.

**Advertencia:** Los arreglos de memoria de tamaño estático no tienen un campo de longitud, pero puede ser que se añada más adelante para permitir una mejor conversión entre arreglos de tamaño estático y dinámico; así que no dependas de esto.

### Seguridad de la memoria

Sin el uso del ensamblado en línea, el compilador puede confiar en que la memoria permanezca en un estado bien definido en todo momento. Esto es especialmente relevante para *la nueva ruta de generación de código a través de Yul IR*: esta vía de generación de código puede mover variables locales de la pila a la memoria para evitar errores de pila demasiado profundos y realizar optimizaciones de memoria adicionales, si puede confiar en ciertas suposiciones sobre el uso de la memoria.

Aunque recomendamos siempre respetar el modelo de memoria de Solidity, el ensamblado en línea te permite usar la memoria de una manera incompatible. Por lo tanto, el traslado de variables de la pila a la memoria y las optimizaciones adicionales están deshabilitadas globalmente por defecto en la presencia de cualquier bloque de ensamblado en línea que contenga una operación de memoria o asigne variables de Solidity en la memoria.

Sin embargo, puede anotar específicamente un bloque de ensamblado para indicar que, de hecho, respeta el modelo de memoria de Solidity de la siguiente manera:

```
assembly ("memory-safe") {
    ...
}
```

En particular, un bloque de ensamblado seguro en cuanto a la memoria solo puede acceder a los siguientes intervalos de memoria:

- Memoria asignada por ti mismo usando un mecanismo como la función `allocate` descrita anteriormente.
- Memoria asignada por Solidity, por ejemplo, memoria dentro de los límites de un arreglo de memoria a la que haces referencia.

- El espacio de memoria virtual entre el offset de memoria 0 y 64 mencionados anteriormente.
- Memoria temporal que se encuentra *después* del valor del puntero de memoria libre al comienzo del bloque de ensamblado, es decir, memoria que se «asigna» al puntero de memoria libre sin actualizar el puntero de memoria libre.

Además, si el bloque de ensamblado asigna variables de Solidity en la memoria, debes asegurarte de que los accesos a las variables de Solidity solo accedan a estos intervalos de memoria.

Dado que esto se trata principalmente del optimizador, estas restricciones todavía deben seguirse, incluso si el bloque de ensamblado se revierte o termina. Como ejemplo, el siguiente fragmento de ensamblado no es seguro en cuanto a la memoria, ya que el valor de `returndatasize()` puede exceder el espacio temporal de 64 bytes:

```
assembly {
  returndatacopy(0, 0, returndatasize())
  revert(0, returndatasize())
}
```

Por el otro lado, el siguiente código *es* seguro en cuanto a la memoria, porque la memoria más allá de la ubicación apuntada por el puntero de memoria libre se puede usar con seguridad como espacio temporal de memoria virtual:

```
assembly ("memory-safe") {
  let p := mload(0x40)
  returndatacopy(p, 0, returndatasize())
  revert(p, returndatasize())
}
```

Ten en cuenta que no necesitas actualizar el puntero de memoria libre si no hay una asignación posterior, pero solo puedes usar la memoria a partir de la dirección actual dad por el puntero de memoria libre.

Si las operaciones de memoria usan una longitud cero, también es aceptable usar cualquier offset (no solo si cae en el espacio temporal):

```
assembly ("memory-safe") {
  revert(0, 0)
}
```

Ten en cuenta que no solo las operaciones de memoria en ensamblado en línea en sí pueden ser inseguras en cuanto a la memoria, pero también las asignaciones a variables de Solidity de tipo referencia en memoria. Por ejemplo, esto no es seguro para la memoria:

```
bytes memory x;
assembly {
  x := 0x40
}
x[0x20] = 0x42;
```

El ensamblado en línea que no involucra ninguna operación que acceda a la memoria ni asigna ninguna variable de Solidity en la memoria se considera automáticamente seguro para la memoria y no necesita ser anotado.

**Advertencia:** Es tu responsabilidad asegurarte de que el ensamblado realmente cumpla el modelo de memoria. Si anotas un bloque de ensamblado como seguro en cuanto a la memoria, pero viola una de las suposiciones de la memoria, esto *provocará* a un comportamiento incorrecto e indeterminado que no puede descubrirse con facilidad mediante pruebas.

En caso de que estes desarrollando una biblioteca que esté destinada a ser compatible con varias versiones de Solidity, puedes usar un comentario especial para anotar un bloque de ensamblado como seguro en cuanto a la memoria:

```
/// @solidity memory-safe-assembly
assembly {
    ...
}
```

Ten en cuenta que en una futura versión deshabilitaremos la anotación mediante comentarios. Si no te preocupa la compatibilidad con versiones anteriores del compilador, preferiblemente usa la secuencia de dialecto.

## 3.11 Apuntes para repaso

### 3.11.1 Orden de Precedencia de Operadores

The following is the order of precedence for operators, listed in order of evaluation.

| Precedence | Description | Operator |
|---|---|---|
| *1* | Postfix increment and decrement | `++`, `--` |
| | New expression | `new <typename>` |
| | Array subscripting | `<array>[<index>]` |
| | Member access | `<object>.<member>` |
| | Function-like call | `<func>(<args...>)` |
| | Parentheses | `(<statement>)` |
| *2* | Prefix increment and decrement | `++`, `--` |
| | Unary minus | `-` |
| | Unary operations | `delete` |
| | Logical NOT | `!` |
| | Bitwise NOT | `~` |
| *3* | Exponentiation | `**` |
| *4* | Multiplication, division and modulo | `*`, `/`, `%` |
| *5* | Addition and subtraction | `+`, `-` |
| *6* | Bitwise shift operators | `<<`, `>>` |
| *7* | Bitwise AND | `&` |
| *8* | Bitwise XOR | `^` |
| *9* | Bitwise OR | `\|` |
| *10* | Inequality operators | `<`, `>`, `<=`, `>=` |
| *11* | Equality operators | `==`, `!=` |
| *12* | Logical AND | `&&` |
| *13* | Logical OR | `\|\|` |
| *14* | Ternary operator | `<conditional> ? <if-true> : <if-false>` |
| | Assignment operators | `=`, `\|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=` |
| *15* | Comma operator | `,` |

### 3.11.2 Variables Globales

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: *ABI*-decodifica la data provista. Los tipos son dados en paréntesis como segundo argumento. Ejemplo: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: *ABI*-codifica los argumentos dados
- `abi.encodePacked(...) returns (bytes memory)`: Realiza *la codificación empaquetada* de los argumentos dados. ¡Note que esta codificación puede ser ambigua!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: *ABI*-codifica los argumentos dados comenzando desde el segundo y antepone el selector dado de cuatro bytes
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)`: ABI-codifica a llamada a `functionPointer` con los argumentos encontrados en la tupla. Realiza un completa verificación de tipos asegurándose que los tipos coincidan con la signatura de la función. El resultado equivale a `abi.encodeWithSelector(functionPointer.selector, (...))`
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Equivalente a `abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...)`
- `bytes.concat(...) returns (bytes memory)`: *Concatena un número variable de argumentos a un array de un byte*
- `string.concat(...) returns (string memory)`: *Concatena un número variable de argumentos a un array de un string*
- `block.basefee` (`uint`): pago base del bloque actual (EIP-3198 y EIP-1559)
- `block.chainid` (`uint`): id de la cadena actual
- `block.coinbase` (`address payable`): dirección del minero del bloque actual
- `block.difficulty` (`uint`): dificultad del bloque actual
- `block.gaslimit` (`uint`): límite de gas del bloque actual
- `block.number` (`uint`): número del bloque actual
- `block.timestamp` (`uint`): marca de fecha del bloque actual en segundos desde el tiempo Unix
- `gasleft() returns (uint256)`: gas restante
- `msg.data` (`bytes`): datos de llamada completos
- `msg.sender` (`address`): remitente del mensaje (llamada actual)
- `msg.sig` (`bytes4`): primeros cuatro bytes de los datos de llamada (i.e. identificador de función)
- `msg.value` (`uint`): cantidad de wei enviados con el mensaje
- `tx.gasprice` (`uint`): precio de gas de la transacción
- `tx.origin` (`address`): remitente de la transacción (cadena de la llamada completa)
- `assert(bool condition)`: aborta la ejecución y revierte los cambios de estado si la condición es `false` (usar para error interno)
- `require(bool condition)`: aborta la ejecución y revierte los cambios de estado si la condición es `false` (usar para entradas mal construidas o error en componente externo)
- `require(bool condition, string memory message)`: aborta la ejecución y revierte los cambios de estado si la condición es `false` (usar para entradas mal construidas o error en componente externo). También provee mensaje de error.

- `revert()`: aborta la ejecución y revierte los cambios de estado
- `revert(string memory message)`: aborta la ejecución y revierte los cambios de estado proveyendo una cadena de caracteres explicativa
- `blockhash(uint blockNumber) returns (bytes32)`: hash del bloque dado - solo funciona para los 256 bloques más recientes
- `keccak256(bytes memory) returns (bytes32)`: computa el hash Keccak-256 de la entrada
- `sha256(bytes memory) returns (bytes32)`: computa el hash SHA-256 de la entrada
- `ripemd160(bytes memory) returns (bytes20)`: computa el hash RIPEMD-160 de la entrada
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recupera la dirección asociada con la clave pública desde la signatura de curva elíptica, devuelve cero en error.
- `addmod(uint x, uint y, uint k) returns (uint)`: computa `(x + y) % k` donde la adición se lleva a cabo con precisión arbitraria y no se detiene en `2**256`. Impone que `k != 0` a partir de la versión 0.5.0
- `mulmod(uint x, uint y, uint k) returns (uint)`: computa `(x * y) % k` donde la multiplicación se lleva a cabo con precisión arbitraria y no se detiene en `2**256`. Impone que `k != 0` a partir de la versión 0.5.0.
- `this` (tipo del contrato actual): el contrato actual, explícitamente convertible a `address` o `address payable`
- `super`: un nivel más alto del contrato en la jerarquía de herencia
- `selfdestruct(address payable recipient)`: destruye el contrato actual enviando sus fondos a la dirección dada
- `<address>.balance` (`uint256`): balance de la dirección en Wei
- `<address>.code` (`bytes memory`): código en la dirección (puede estar vacío)
- `<address>.codehash` (`bytes32`): el código hash de la dirección
- `<address payable>.send(uint256 amount) returns (bool)`: envía la cantidad dada de Wei a la dirección, regresa `false` al fallar
- `<address payable>.transfer(uint256 amount)`: envía la cantidad dada de Wei a la dirección, lanza una excepción al fallar
- `type(C).name` (`string`): el nombre del contrato
- `type(C).creationCode` (`bytes memory`): creación en bytecode del contrato dado, véase *Información de Tipos*.
- `type(C).runtimeCode` (`bytes memory`): bytecode en tiempo de ejecución del contrato dado, véase *Información de Tipos*.
- `type(I).interfaceId` (`bytes4`): valor que contiene el identificador de la interface EIP-165 de la intergace dada, véase *Información de Tipos*.
- `type(T).min` (`T`): el valor mínimo representable por el tipo entero `T`, véase *Información de Tipos*.
- `type(T).max` (`T`): el valor máximo representable por el tipo entero `T`, véase *Información de Tipos*.

### 3.11.3 Especificadores de la Visibilidad de Funciones

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`: visible externamente e internamente (crea una *función getter* para variables de estado/almacenamiento)
- `private`: solamente visible en el contrato actual
- `external`: solamente visible externamente (solo para funciones) - i.e. solo se puede llamar por mensaje (a través de `this.func`)
- `internal`: solamente visible internamente

### 3.11.4 Modificadores

- `pure` para funciones: No acepta la modificación o acceso al estado.
- `view` para funciones: No acepta la modificación del estado.
- `payable` para funciones: Permite recibir Ether junto con una llamada.
- `constant` para variables de estado: No permite la asignación (excepto la inicialización), no ocupa lugar de almacenamiento.
- `immutable` para variables de estado: Permite exactamente una asignación en el tiempo de construcción y es constante después. Se almacena en el código.
- `anonymous` para eventos: No almacena la signatura del evento como tema.
- `indexed` para parámetros de eventos: Almacena el parámetro como tema.
- `virtual` para funciones y modificadores: Permite que el comportamiento de las funciones y modificadores se modifique en contratos derivados.
- `override`: Establece que esta función, modificador o variable de estado pública cambia el comportamiento de una función o modificador en un contrato de base.

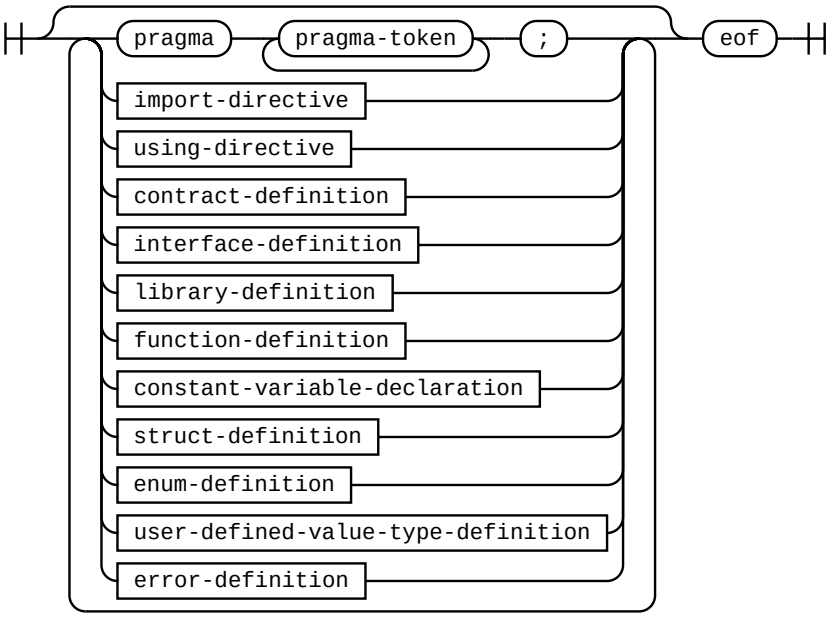
## 3.12 Gramática del Lenguaje

**parser grammar SolidityParser**

Solidity is a statically typed, contract-oriented, high-level language for implementing smart contracts on the Ethereum platform.

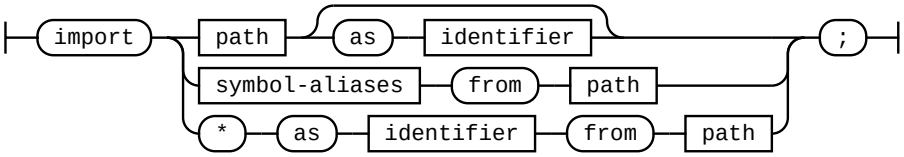
**rule source-unit**

On top level, Solidity allows pragmas, import directives, and definitions of contracts, interfaces, libraries, structs, enums and constants.



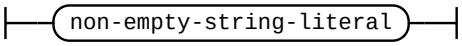

**rule import-directive**

Import directives import identifiers from different files.



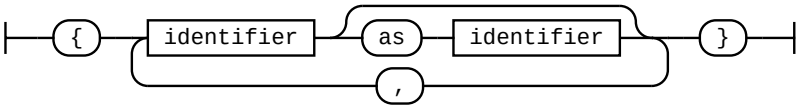

**rule path**

Path of a file to be imported.



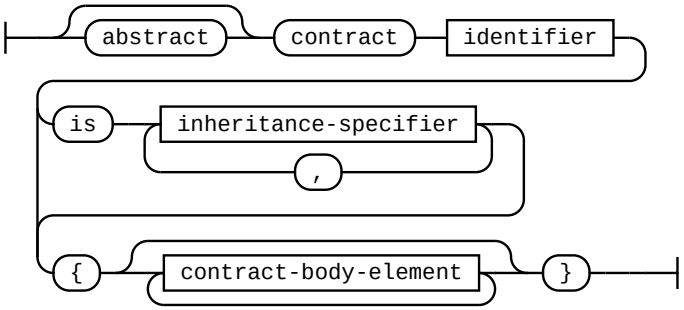

**rule symbol-aliases**

List of aliases for symbols to be imported.



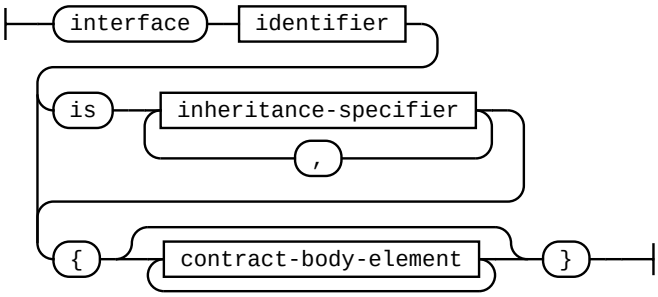

**rule contract-definition**

Top-level definition of a contract.



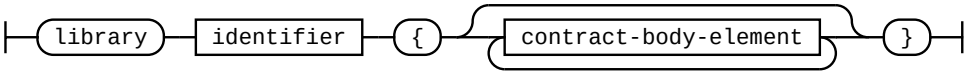
**rule interface-definition**

Top-level definition of an interface.



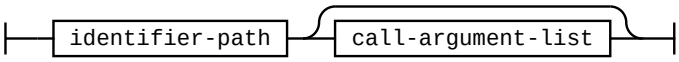

**rule library-definition**

Top-level definition of a library.




**rule inheritance-specifier**

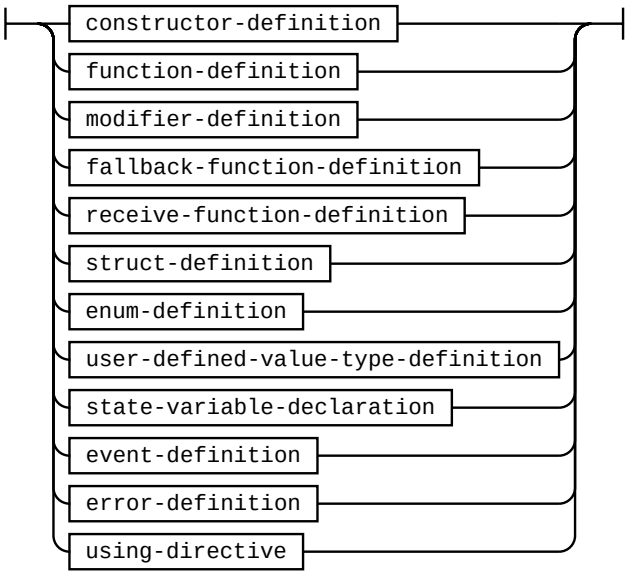
Inheritance specifier for contracts and interfaces. Can optionally supply base constructor arguments.




**rule contract-body-element**

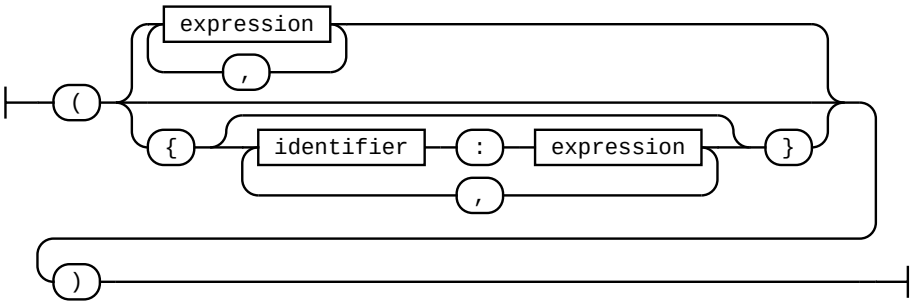
Declarations that can be used in contracts, interfaces and libraries.

Note that interfaces and libraries may not contain constructors, interfaces may not contain state variables and libraries may not contain fallback, receive functions nor non-constant state variables.



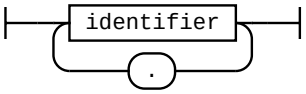

**rule call-argument-list**

Arguments when calling a function or a similar callable object. The arguments are either given as comma separated list or as map of named arguments.



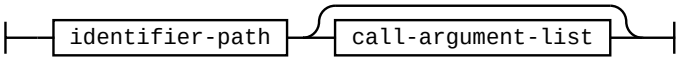

**rule identifier-path**

Qualified name.



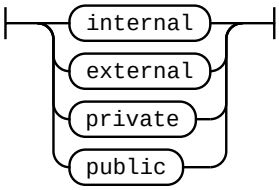

**rule modifier-invocation**

Call to a modifier. If the modifier takes no arguments, the argument list can be skipped entirely (including opening and closing parentheses).



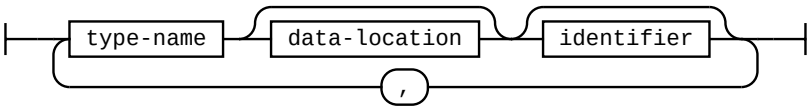

**rule visibility**

Visibility for functions and function types.



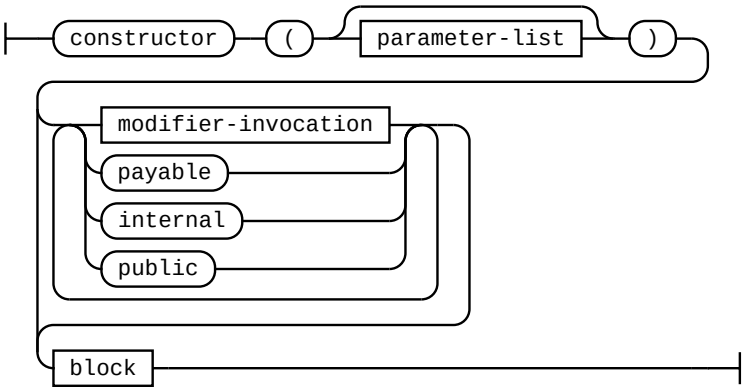

**rule parameter-list**

A list of parameters, such as function arguments or return values.



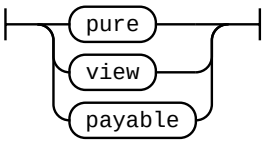

**rule constructor-definition**

Definition of a constructor. Must always supply an implementation. Note that specifying internal or public visibility is deprecated.



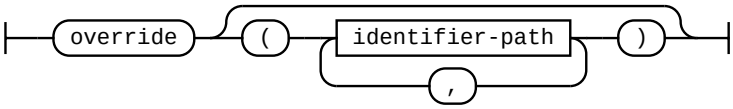

**rule state-mutability**

State mutability for function types. The default mutability "non-payable" is assumed if no mutability is specified.



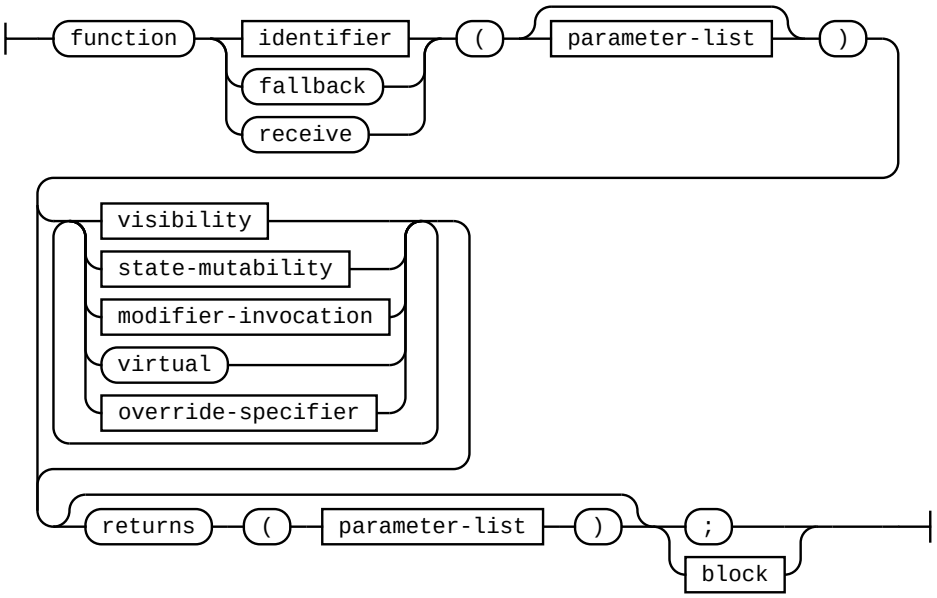

**rule override-specifier**

An override specifier used for functions, modifiers or state variables. In cases where there are ambiguous declarations in several base contracts being overridden, a complete list of base contracts has to be given.



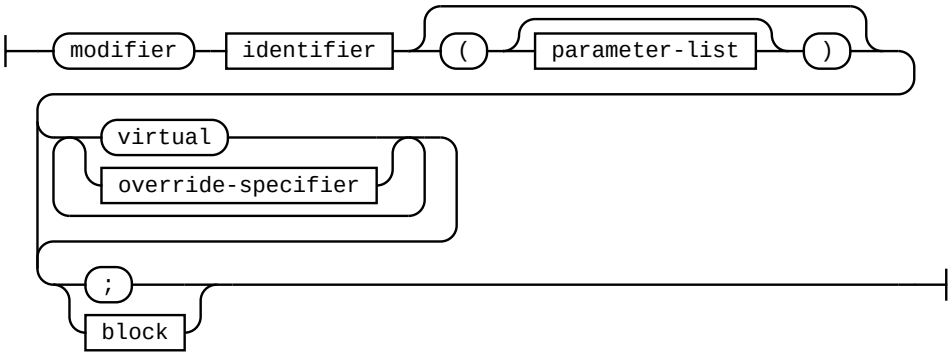

**rule function-definition**

The definition of contract, library and interface functions. Depending on the context in which the function is defined, further restrictions may apply, e.g. functions in interfaces have to be unimplemented, i.e. may not contain a body block.



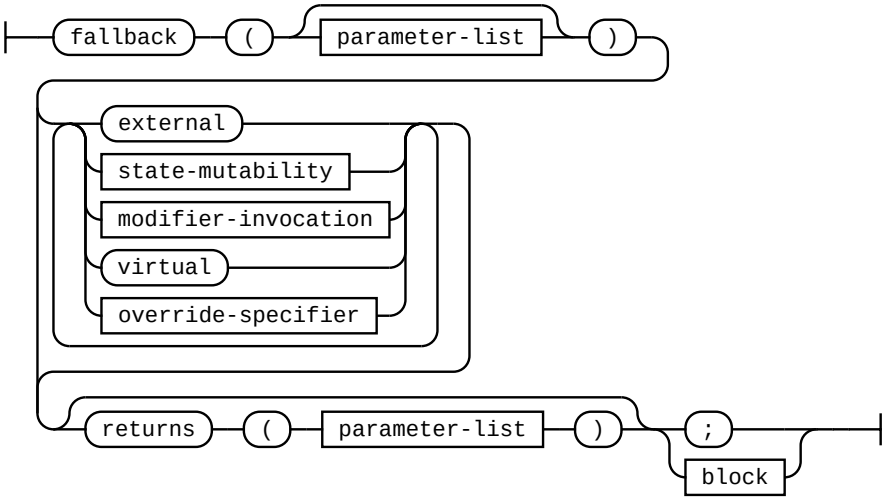
**rule modifier-definition**

The definition of a modifier. Note that within the body block of a modifier, the underscore cannot be used as identifier, but is used as placeholder statement for the body of a function to which the modifier is applied.



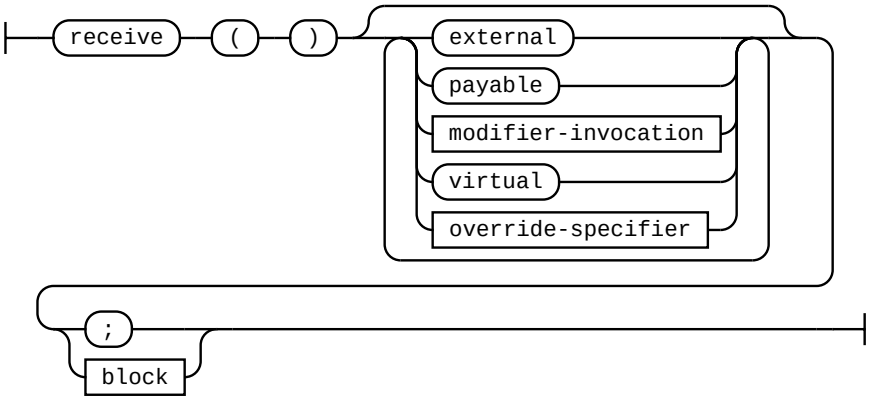

**rule fallback-function-definition**

Definition of the special fallback function.



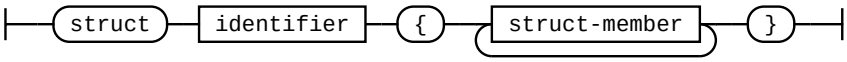

**rule receive-function-definition**

Definition of the special receive function.



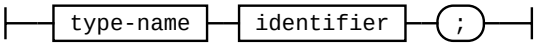

**rule struct-definition**

Definition of a struct. Can occur at top-level within a source unit or within a contract, library or interface.



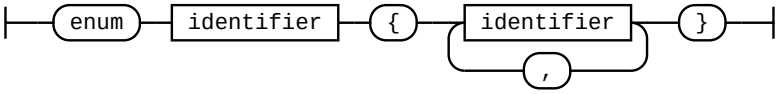

**rule struct-member**

The declaration of a named struct member.



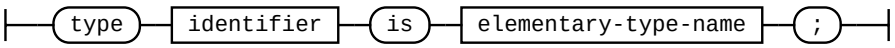

**rule enum-definition**

Definition of an enum. Can occur at top-level within a source unit or within a contract, library or interface.



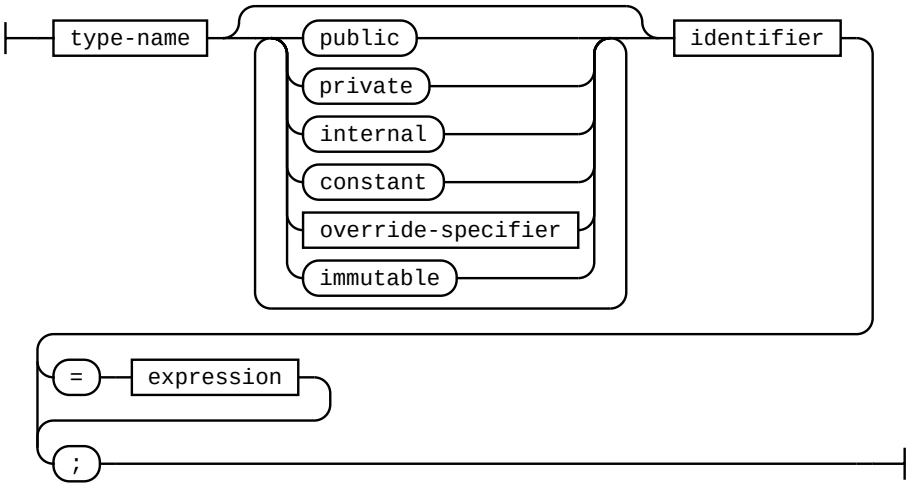

**rule user-defined-value-type-definition**

Definition of a user defined value type. Can occur at top-level within a source unit or within a contract, library or interface.



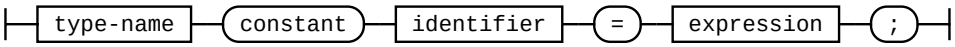

**rule state-variable-declaration**

The declaration of a state variable.



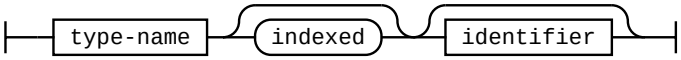

**rule constant-variable-declaration**

The declaration of a constant variable.



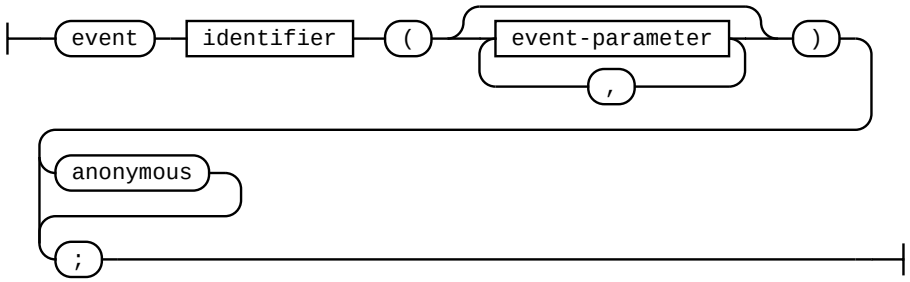

**rule event-parameter**

Parameter of an event.



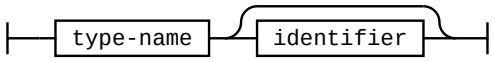

**rule event-definition**

Definition of an event. Can occur in contracts, libraries or interfaces.



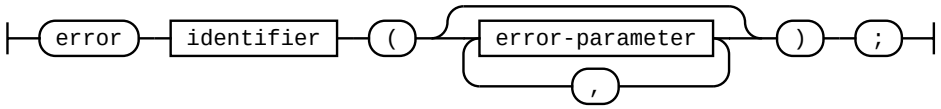

**rule error-parameter**

Parameter of an error.



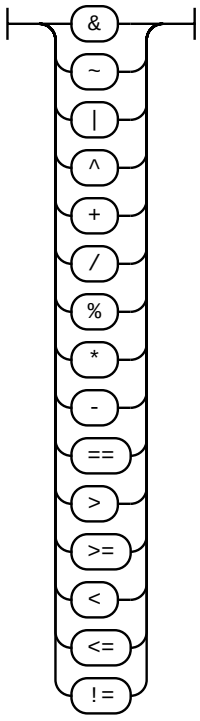

**rule error-definition**

Definition of an error.



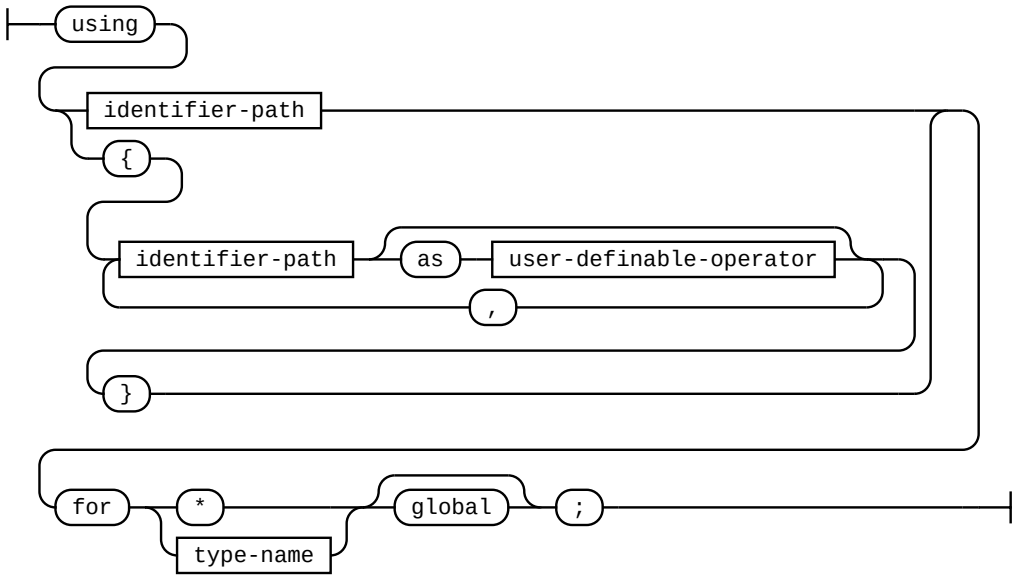

**rule user-definable-operator**

Operators that users are allowed to implement for some types with *using for*.



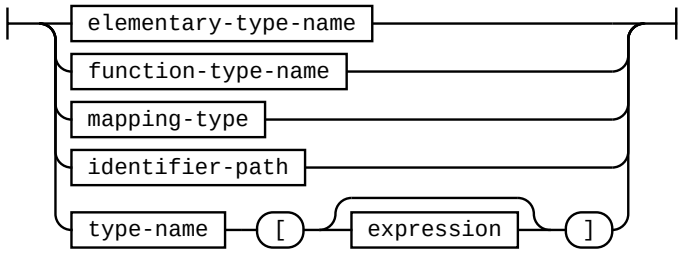

**rule using-directive**

Using directive to attach library functions and free functions to types. Can occur within contracts and libraries and at the file level.

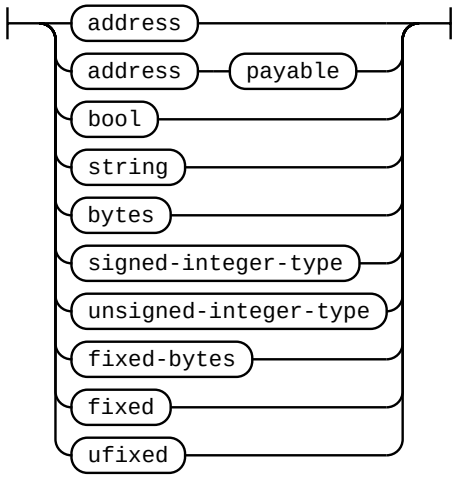



**rule type-name**

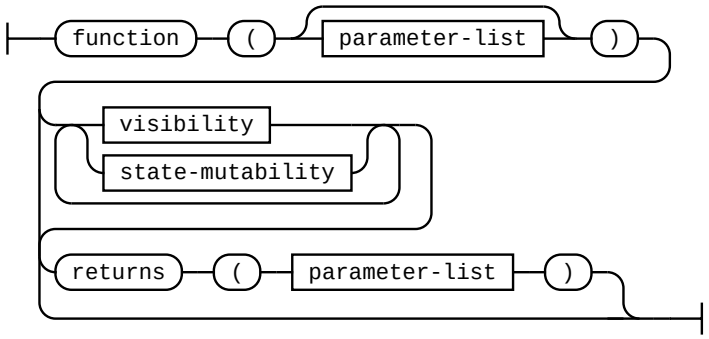
A type name can be an elementary type, a function type, a mapping type, a user-defined type (e.g. a contract or struct) or an array type.




**rule elementary-type-name**

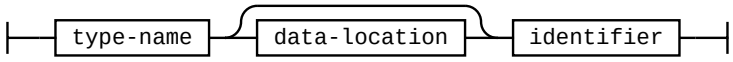



**rule function-type-name**

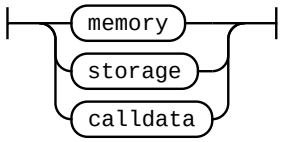



**rule variable-declaration**

The declaration of a single variable.

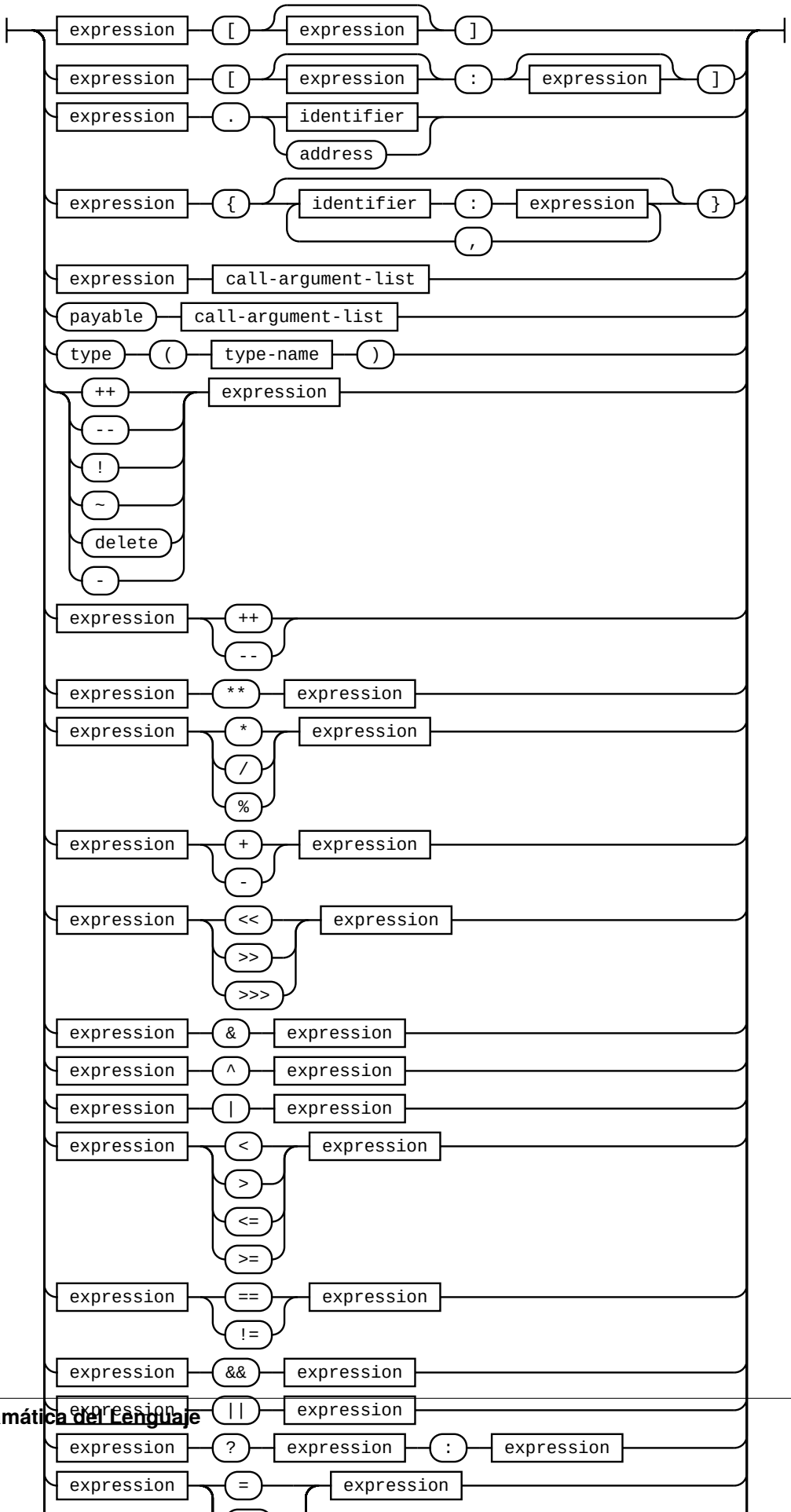



**rule data-location**

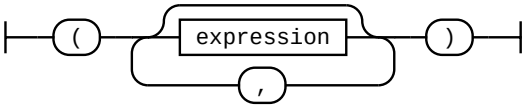



**rule expression**

Complex expression. Can be an index access, an index range access, a member access, a function call (with optional function call options), a type conversion, an unary or binary expression, a comparison or assignment, a ternary expression, a new-expression (i.e. a contract creation or the allocation of a dynamic memory array), a tuple, an inline array or a primary expression (i.e. an identifier, literal or type name).

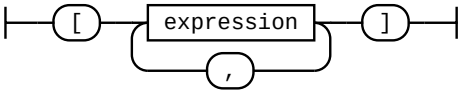

expression
[
expression
]
expression
[
expression
:
expression
]
expression
.
identifier
address
expression
{
identifier
:
expression
}
,
expression
call-argument-list
payable
call-argument-list
type
(
type-name
)
++
--
!
~
delete
-
expression
expression
++
--
expression
**
expression
expression
*
/
%
expression
expression
+
-
expression
expression
<<
>>
>>>
expression
expression
&
expression
expression
^
expression
expression
|
expression
expression
<
>
<=
>=
expression
expression
==
!=
expression
expression
&&
expression
expression
||
expression
expression
?
expression
:
expression
expression
=
expression

**rule tuple-expression**



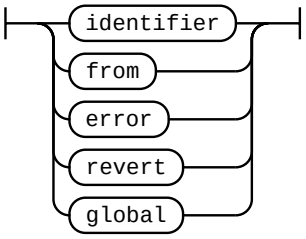

**rule inline-array-expression**

An inline array expression denotes a statically sized array of the common type of the contained expressions.

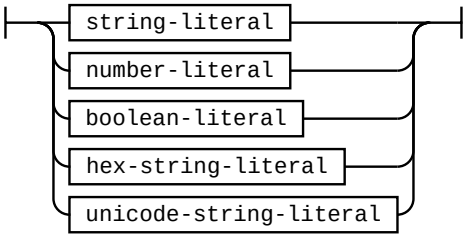



**rule identifier**

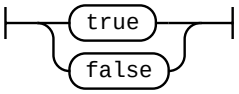
Besides regular non-keyword Identifiers, some keywords like “from” and “error” can also be used as identifiers.




**rule literal**

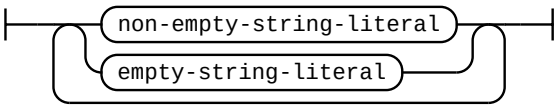



**rule boolean-literal**



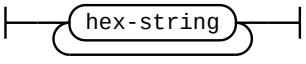

**rule string-literal**

A full string literal consists of either one or several consecutive quoted strings.



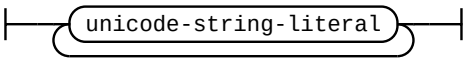

**rule hex-string-literal**

A full hex string literal that consists of either one or several consecutive hex strings.



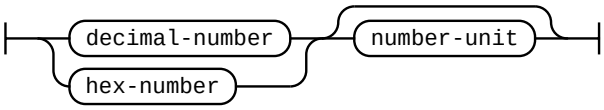

**rule unicode-string-literal**

A full unicode string literal that consists of either one or several consecutive unicode strings.



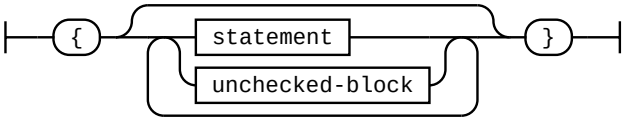

**rule number-literal**

Number literals can be decimal or hexadecimal numbers with an optional unit.

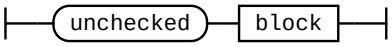



**rule block**

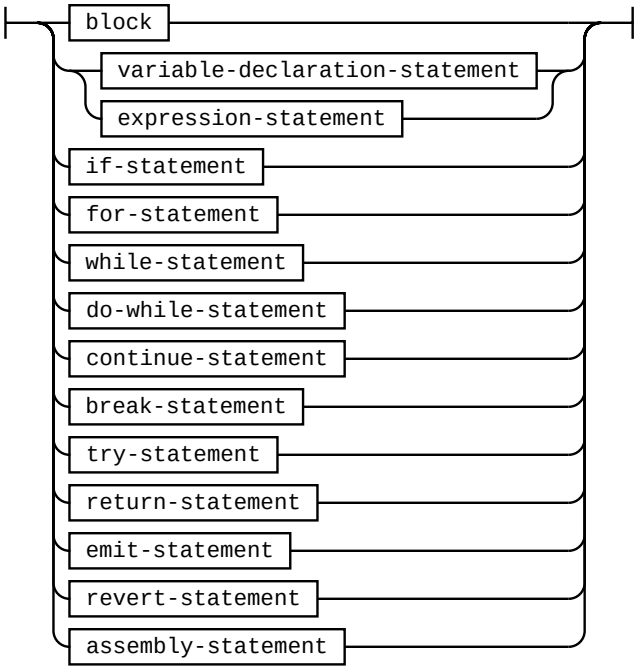
A curly-braced block of statements. Opens its own scope.




**rule unchecked-block**

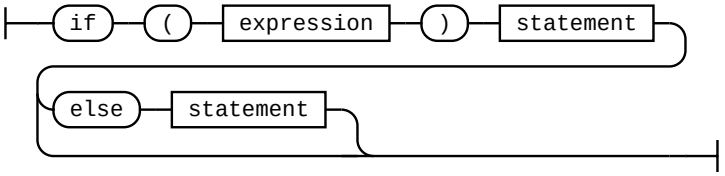



**rule statement**



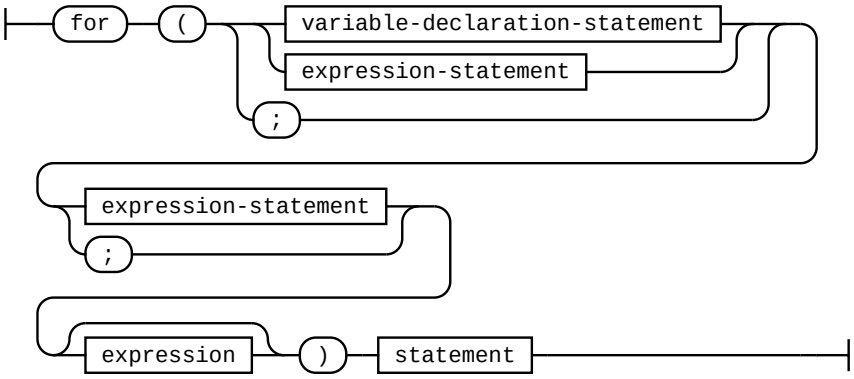

**rule if-statement**

If statement with optional else part.

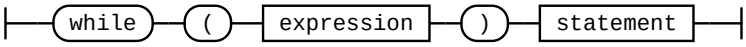


**rule for-statement**

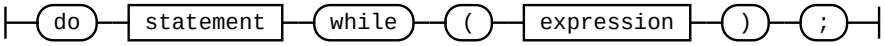
For statement with optional init, condition and post-loop part.




**rule while-statement**

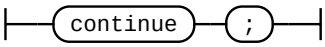



**rule do-while-statement**



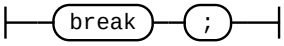

**rule continue-statement**

A continue statement. Only allowed inside for, while or do-while loops.



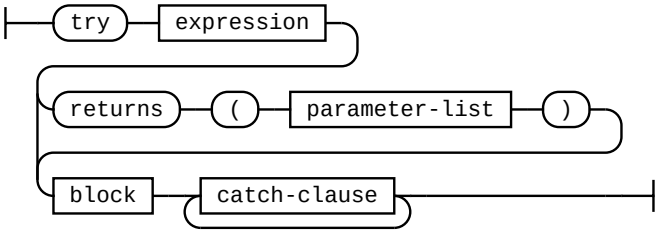

**rule break-statement**

A break statement. Only allowed inside for, while or do-while loops.



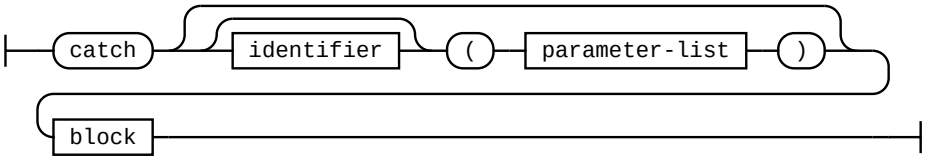

**rule try-statement**

A try statement. The contained expression needs to be an external function call or a contract creation.

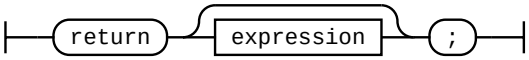



**rule catch-clause**

The catch clause of a try statement.

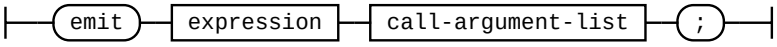


**rule return-statement**



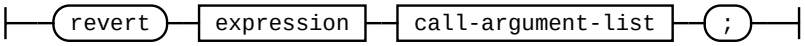

**rule emit-statement**

An emit statement. The contained expression needs to refer to an event.



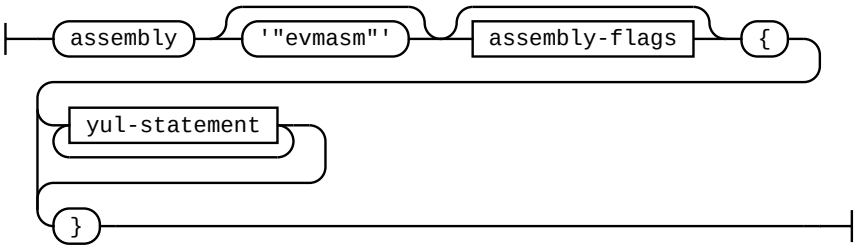

**rule revert-statement**

A revert statement. The contained expression needs to refer to an error.



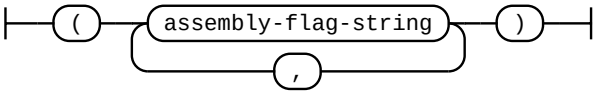

**rule assembly-statement**

An inline assembly block. The contents of an inline assembly block use a separate scanner/lexer, i.e. the set of keywords and allowed identifiers is different inside an inline assembly block.



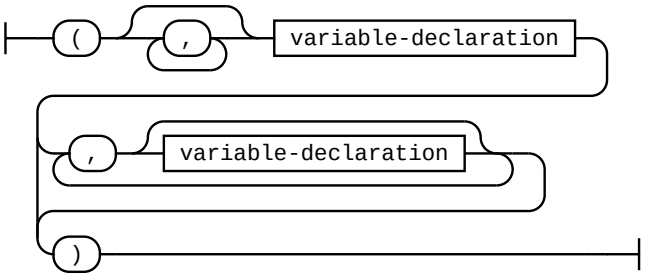

**rule assembly-flags**

Assembly flags. Comma-separated list of double-quoted strings as flags.



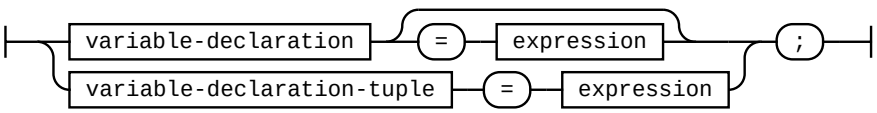

**rule variable-declaration-tuple**

A tuple of variable names to be used in variable declarations. May contain empty fields.

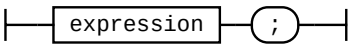



**rule variable-declaration-statement**

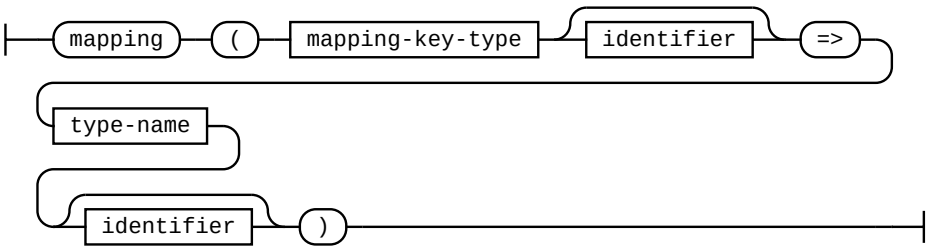
A variable declaration statement. A single variable may be declared without initial value, whereas a tuple of variables can only be declared with initial value.



**rule expression-statement**

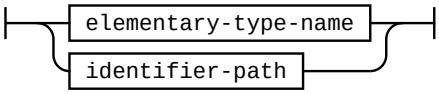



**rule mapping-type**



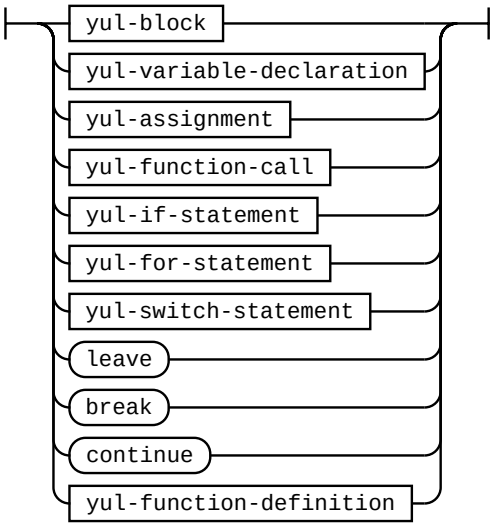

**rule mapping-key-type**

Only elementary types or user defined types are viable as mapping keys.

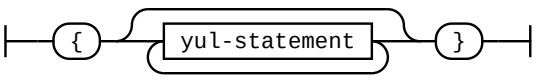



**rule yul-statement**

A Yul statement within an inline assembly block. continue and break statements are only valid within for loops. leave statements are only valid within function bodies.

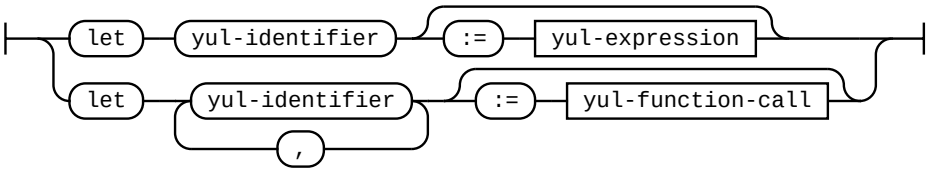



**rule yul-block**



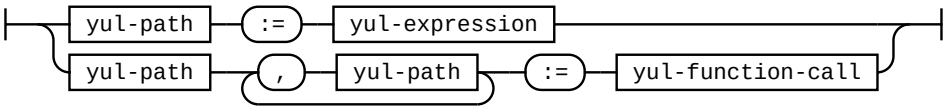

**rule yul-variable-declaration**

The declaration of one or more Yul variables with optional initial value. If multiple variables are declared, only a function call is a valid initial value.

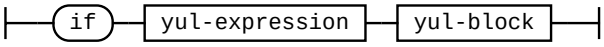



**rule yul-assignment**

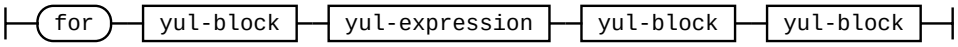
Any expression can be assigned to a single Yul variable, whereas multi-assignments require a function call on the right-hand side.




**rule yul-if-statement**

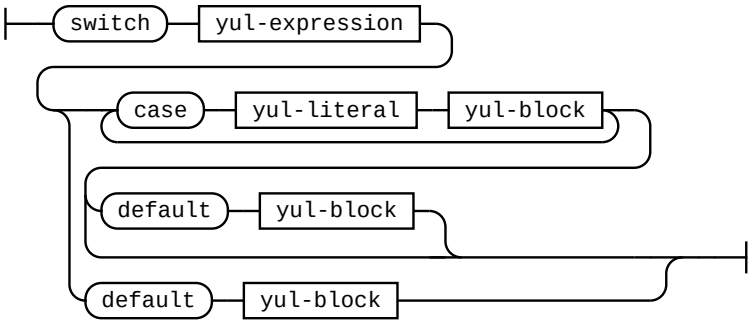



**rule yul-for-statement**

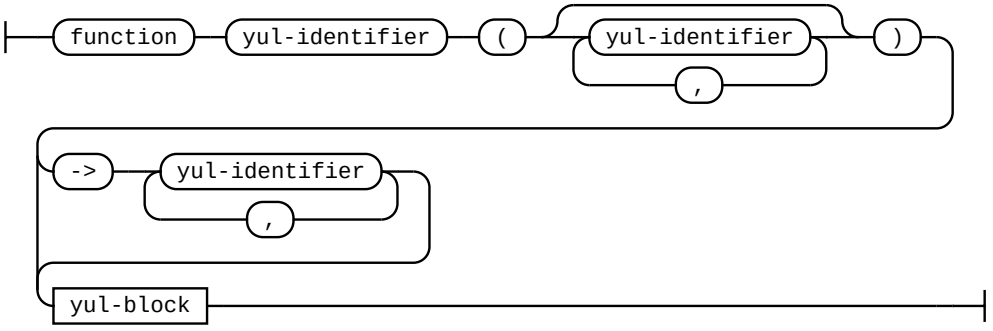



**rule yul-switch-statement**

A Yul switch statement can consist of only a default-case (deprecated) or one or more non-default cases optionally followed by a default-case.

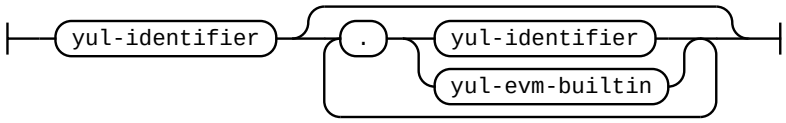



**rule yul-function-definition**



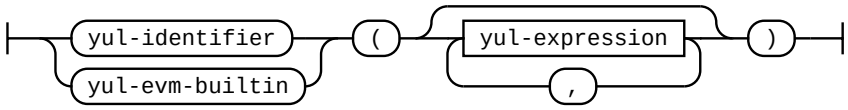

**rule yul-path**

While only identifiers without dots can be declared within inline assembly, paths containing dots can refer to declarations outside the inline assembly block.

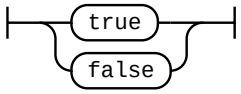



**rule yul-function-call**

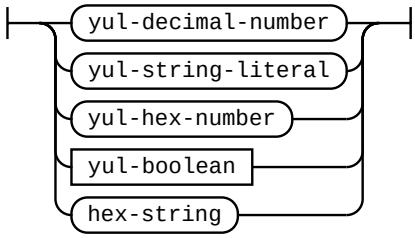
A call to a function with return values can only occur as right-hand side of an assignment or a variable declaration.



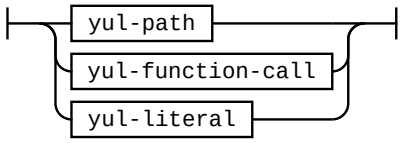

**rule yul-boolean**




**rule yul-literal**



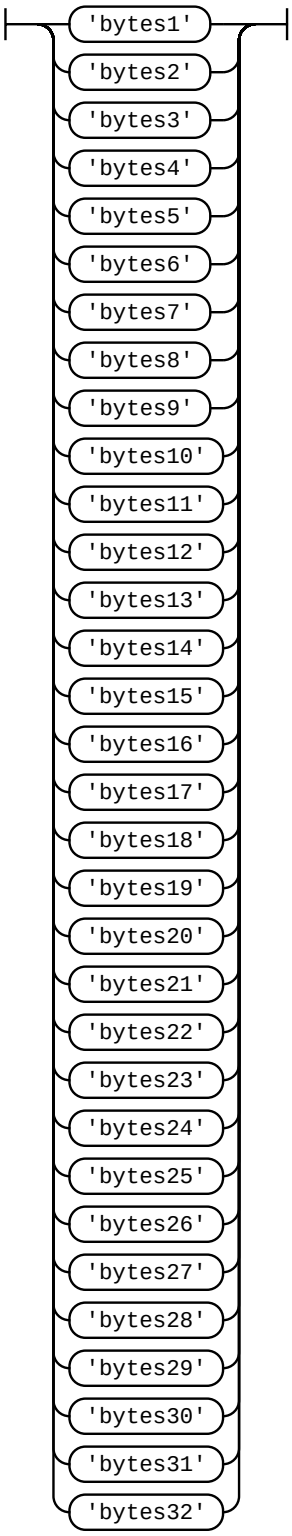

**rule yul-expression**




**lexer grammar SolidityLexer**

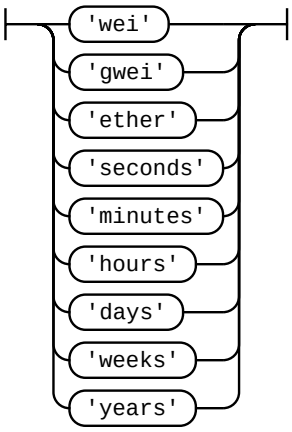
**rule fixed-bytes**

Bytes types of fixed length.



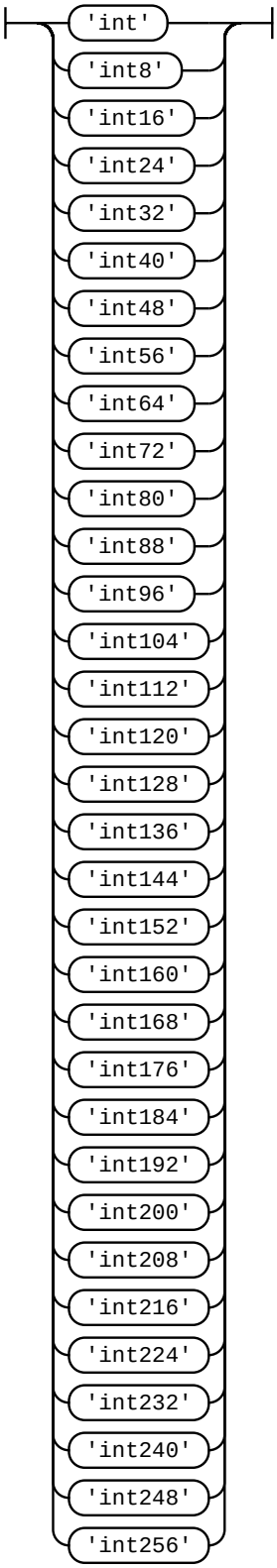

**rule number-unit**

Unit denomination for numbers.



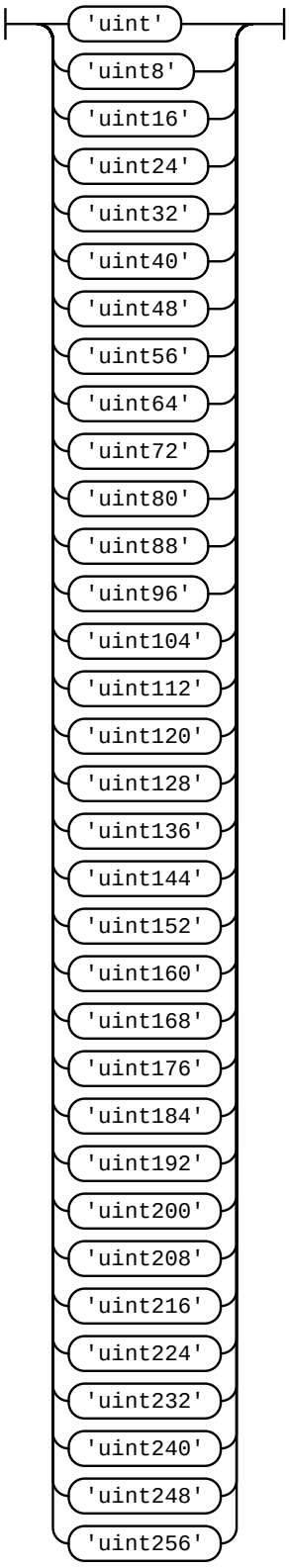

**rule signed-integer-type**

Sized signed integer types. int is an alias of int256.



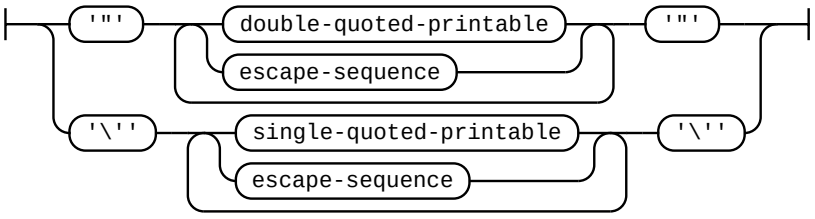

**rule unsigned-integer-type**

Sized unsigned integer types. uint is an alias of uint256.



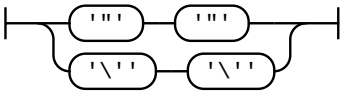

**rule non-empty-string-literal**

A non-empty quoted string literal restricted to printable characters.



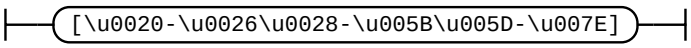

**rule empty-string-literal**

An empty string literal



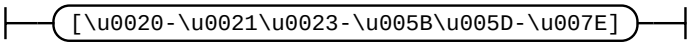

**rule single-quoted-printable**

Any printable character except single quote or back slash.



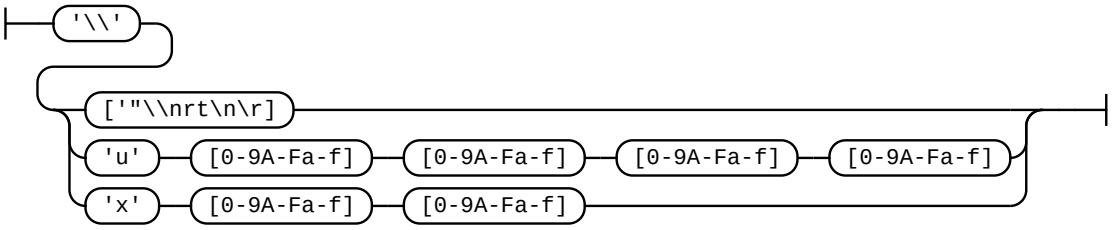

**rule double-quoted-printable**

Any printable character except double quote or back slash.



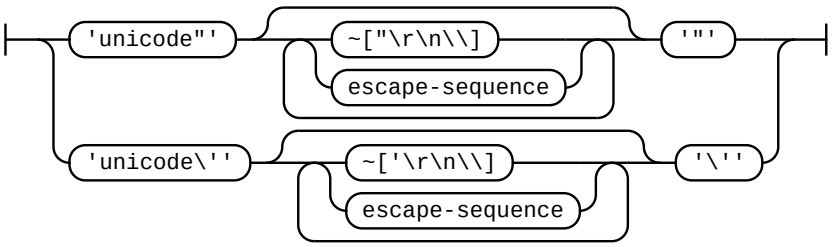

**rule escape-sequence**

Escape sequence. Apart from common single character escape sequences, line breaks can be escaped as well as four hex digit unicode escapes \uXXXX and two digit hex escape sequences \xXX are allowed.



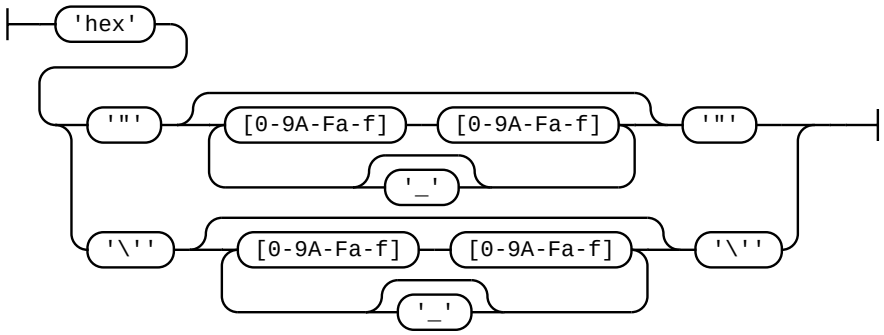

**rule unicode-string-literal**

A single quoted string literal allowing arbitrary unicode characters.



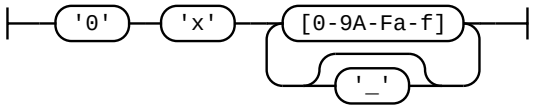

**rule hex-string**

Hex strings need to consist of an even number of hex digits that may be grouped using underscores.



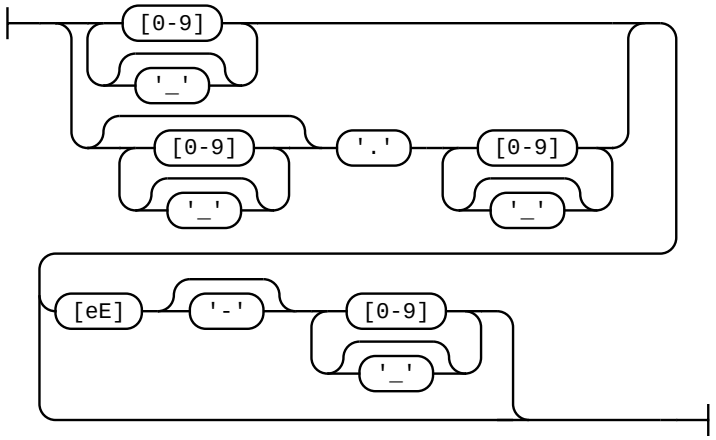

**rule hex-number**

Hex numbers consist of a prefix and an arbitrary number of hex digits that may be delimited by underscores.



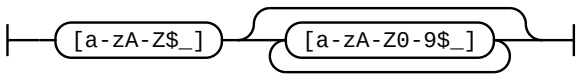

**rule decimal-number**

A decimal number literal consists of decimal digits that may be delimited by underscores and an optional positive or negative exponent. If the digits contain a decimal point, the literal has fixed point type.



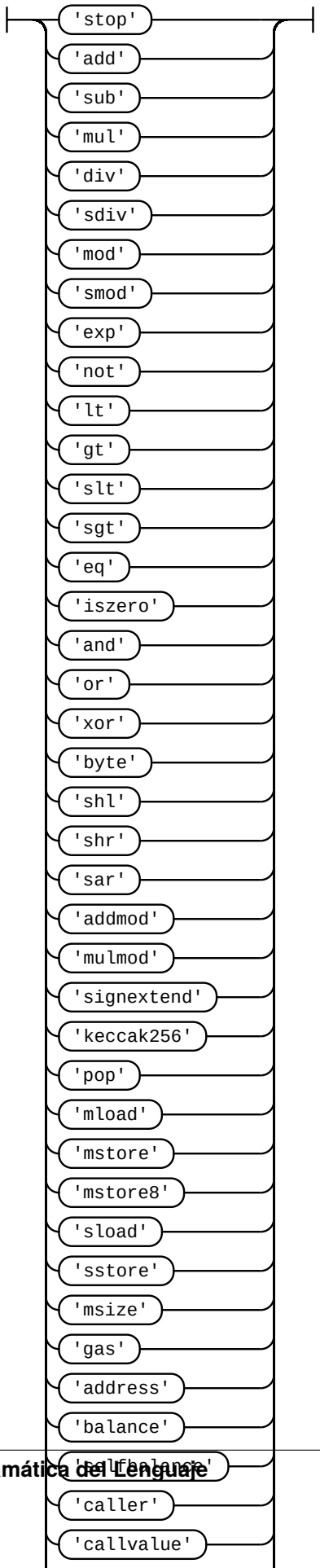

**rule identifier**

An identifier in solidity has to start with a letter, a dollar-sign or an underscore and may additionally contain numbers after the first symbol.



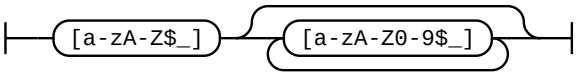

**rule yul-evm-builtin**

Builtin functions in the EVM Yul dialect.


'stop'
'add'
'sub'
'mul'
'div'
'sdiv'
'mod'
'smod'
'exp'
'not'
'lt'
'gt'
'slt'
'sgt'
'eq'
'iszero'
'and'
'or'
'xor'
'byte'
'shl'
'shr'
'sar'
'addmod'
'mulmod'
'signextend'
'keccak256'
'pop'
'mload'
'mstore'
'mstore8'
'sload'
'sstore'
'msize'
'gas'
'address'
'balance'
'selfbalance'
'caller'
'callvalue'

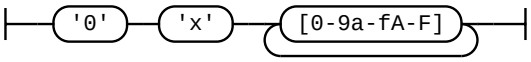
**rule yul-identifier**

Yul identifiers consist of letters, dollar signs, underscores and numbers, but may not start with a number. In inline assembly there cannot be dots in user-defined identifiers. Instead see yulPath for expressions consisting of identifiers with dots.



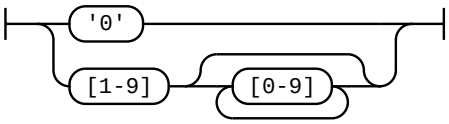

**rule yul-hex-number**

Hex literals in Yul consist of a prefix and one or more hexadecimal digits.



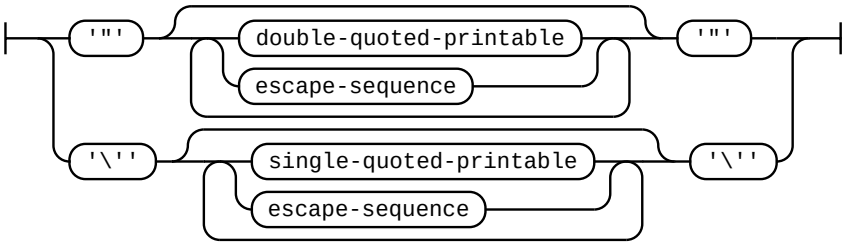

**rule yul-decimal-number**

Decimal literals in Yul may be zero or any sequence of decimal digits without leading zeroes.




**rule yul-string-literal**

String literals in Yul consist of one or more double-quoted or single-quoted strings that may contain escape sequences and printable characters except unescaped line breaks or unescaped double-quotes or single-quotes, respectively.




**rule pragma-token**

Pragma token. Can contain any kind of symbol except a semicolon. Note that currently the solidity parser only allows a subset of this.

## 3.13 Using the Compiler

### 3.13.1 Using the Commandline Compiler

**Nota:** This section does not apply to *solcjs*, not even if it is used in commandline mode.

### Basic Usage

One of the build targets of the Solidity repository is `solc`, the solidity commandline compiler. Using `solc --help` provides you with an explanation of all options. The compiler can produce various outputs, ranging from simple binaries and assembly over an abstract syntax tree (parse tree) to estimations of gas usage. If you only want to compile a single file, you run it as `solc --bin sourceFile.sol` and it will print the binary. If you want to get some of the more advanced output variants of `solc`, it is probably better to tell it to output everything to separate files using `solc -o outputDirectory --bin --ast-compact-json --asm sourceFile.sol`.

### Optimizer Options

Before you deploy your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime (more specifically, it assumes each opcode is executed around 200 times). If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--optimize-runs=1`. If you expect many transactions and do not care for higher deployment cost and output size, set `--optimize-runs` to a high number. This parameter has effects on the following (this might change in the future):

- the size of the binary search in the function dispatch routine
- the way constants like large numbers or strings are stored

### Base Path and Import Remapping

The commandline compiler will automatically read imported files from the filesystem, but it is also possible to provide *path redirects* using `prefix=path` in the following way:

```
solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ file.sol
```

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`.

When accessing the filesystem to search for imports, *paths that do not start with ./ or ../* are treated as relative to the directories specified using `--base-path` and `--include-path` options (or the current working directory if base path is not specified). Furthermore, the part of the path added via these options will not appear in the contract metadata.

For security reasons the compiler has *restrictions on what directories it can access*. Directories of source files specified on the command line and target paths of remappings are automatically allowed to be accessed by the file reader, but everything else is rejected by default. Additional paths (and their subdirectories) can be allowed via the `--allow-paths /sample/path,/another/sample/path` switch. Everything inside the path specified via `--base-path` is always allowed.

The above is only a simplification of how the compiler handles import paths. For a detailed explanation with examples and discussion of corner cases please refer to the section on *path resolution*.

### Library Linking

If your contracts use *libraries*, you will notice that the bytecode contains substrings of the form `__$53aea86b7d70b31448b230b20ae141a537$__`. These are placeholders for the actual library addresses. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name. The bytecode file will also contain lines of the form `// <placeholder> -> <fq library name>` at the end to help identify which libraries the placeholders represent. Note that the fully qualified library name is the path of its source file and the library name separated by `:`. You can use `solc` as a linker meaning that it will insert the library addresses for you at those points:

Either add `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890 file.sol:Heap=0xabCD5678901234567890123456789012345678890"` to your command to provide an address for each library (use commas or spaces as separators) or store the string in a file (one library per line) and run `solc` using `--libraries fileName`.

---

**Nota:** Starting Solidity 0.8.1 accepts `=` as separator between library and address, and `:` as a separator is deprecated. It will be removed in the future. Currently `--libraries "file.sol:Math:0x1234567890123456789012345678901234567890 file.sol:Heap:0xabCD5678901234567890123456789012345678890"` will work too.

---

If `solc` is called with the option `--standard-json`, it will expect a JSON input (as explained below) on the standard input, and return a JSON output on the standard output. This is the recommended interface for more complex and especially automated uses. The process will always terminate in a «success» state and report any errors via the JSON output. The option `--base-path` is also processed in standard-json mode.

If `solc` is called with the option `--link`, all input files are interpreted to be unlinked binaries (hex-encoded) in the `__$53aea86b7d70b31448b230b20ae141a537$__`-format given above and are linked in-place (if the input is read from stdin, it is written to stdout). All options except `--libraries` are ignored (including `-o`) in this case.

**Advertencia:** Manually linking libraries on the generated bytecode is discouraged because it does not update contract metadata. Since metadata contains a list of libraries specified at the time of compilation and bytecode contains a metadata hash, you will get different binaries, depending on when linking is performed.

You should ask the compiler to link the libraries at the time a contract is compiled by either using the `--libraries` option of `solc` or the `libraries` key if you use the standard-JSON interface to the compiler.

---

**Nota:** The library placeholder used to be the fully qualified name of the library itself instead of the hash of it. This format is still supported by `solc --link` but the compiler will no longer output it. This change was made to reduce the likelihood of a collision between libraries, since only the first 36 characters of the fully qualified library name could be used.

---

### 3.13.2 Setting the EVM Version to Target

When you compile your contract code you can specify the Ethereum virtual machine version to compile for to avoid particular features or behaviours.

**Advertencia:** Compiling for the wrong EVM version can result in wrong, strange and failing behaviour. Please ensure, especially if running a private chain, that you use matching EVM versions.

On the command line, you can select the EVM version as follows:

```
solc --evm-version <VERSION> contract.sol
```

In the *standard JSON interface*, use the `"evmVersion"` key in the `"settings"` field:

```
{
  "sources": {/* ... */},
  "settings": {
    "optimizer": {/* ... */},
    "evmVersion": "<VERSION>"
  }
}
```

#### Target Options

Below is a list of target EVM versions and the compiler-relevant changes introduced at each version. Backward compatibility is not guaranteed between each version.

- **homestead**
  - (oldest version)
- **tangerineWhistle**
  - Gas cost for access to other accounts increased, relevant for gas estimation and the optimizer.
  - All gas sent by default for external calls, previously a certain amount had to be retained.
- **spuriousDragon**
  - Gas cost for the `exp` opcode increased, relevant for gas estimation and the optimizer.
- **byzantium**
  - Opcodes `returndatacopy`, `returndatasize` and `staticcall` are available in assembly.
  - The `staticcall` opcode is used when calling non-library view or pure functions, which prevents the functions from modifying state at the EVM level, i.e., even applies when you use invalid type conversions.
  - It is possible to access dynamic data returned from function calls.
  - `revert` opcode introduced, which means that `revert()` will not waste gas.
- **constantinople**
  - Opcodes `create2`, ``extcodehash`, `shl`, `shr` and `sar` are available in assembly.
  - Shifting operators use shifting opcodes and thus need less gas.
- **petersburg**

- The compiler behaves the same way as with constantinople.

- **istanbul**
  - Opcodes `chainid` and `selfbalance` are available in assembly.
- **berlin**
  - Gas costs for `SLOAD`, `*CALL`, `BALANCE`, `EXT*` and `SELFDESTRUCT` increased. The compiler assumes cold gas costs for such operations. This is relevant for gas estimation and the optimizer.
- **london**
  - The block's base fee (EIP-3198 and EIP-1559) can be accessed via the global `block.basefee` or `basefee()` in inline assembly.
- **paris (default)**
  - Introduces `prevrandao()` and `block.prevrandao`, and changes the semantics of the now deprecated `block.difficulty`, disallowing `difficulty()` in inline assembly (see EIP-4399).

### 3.13.3 Compiler Input and Output JSON Description

The recommended way to interface with the Solidity compiler especially for more complex and automated setups is the so-called JSON-input-output interface. The same interface is provided by all distributions of the compiler.

The fields are generally subject to change, some are optional (as noted), but we try to only make backwards compatible changes.

The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output. The standard error output is not used and the process will always terminate in a «success» state, even if there were errors. Errors are always reported as part of the JSON output.

The following subsections describe the format through an example. Comments are of course not permitted and used here only for explanatory purposes.

#### Input Description

```
{
  // Required: Source code language. Currently supported are "Solidity" and "Yul".
  "language": "Solidity",
  // Required
  "sources":
  {
    // The keys here are the "global" names of the source files,
    // imports can use other files via remappings (see below).
    "myFile.sol":
    {
      // Optional: keccak256 hash of the source file
      // It is used to verify the retrieved content if imported via URLs.
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): URL(s) to the source file.
      // URL(s) should be imported in this order and the result checked against the
      // keccak256 hash (if available). If the hash doesn't match or none of the
      // URL(s) result in success, an error should be raised.
      // Using the commandline interface only filesystem paths are supported.
      // With the JavaScript interface the URL will be passed to the user-supplied
```

(continué en la próxima página)

(proviene de la página anterior)

```
      // read callback, so any URL supported by the callback can be used.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "/tmp/path/to/file.sol"
        // If files are used, their directories should be added to the command line via
        // `--allow-paths <path>`.
      ]
    },
    "destructible":
    {
      // Optional: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "urls" is used): literal contents of the source file
      "content": "contract destructible is owned { function shutdown() { if (msg.sender
↪== owner) selfdestruct(owner); } }"
    }
  },
  // Optional
  "settings":
  {
    // Optional: Stop compilation after the given stage. Currently only "parsing" is
↪valid here
    "stopAfter": "parsing",
    // Optional: Sorted list of remappings
    "remappings": [ ":g=/dir" ],
    // Optional: Optimizer settings
    "optimizer": {
      // Disabled by default.
      // NOTE: enabled=false still leaves some optimizations on. See comments below.
      // WARNING: Before version 0.8.6 omitting the 'enabled' key was not equivalent to
↪setting
      // it to false and would actually disable all the optimizations.
      "enabled": true,
      // Optimize for how many times you intend to run the code.
      // Lower values will optimize more for initial deployment cost, higher
      // values will optimize more for high-frequency usage.
      "runs": 200,
      // Switch optimizer components on or off in detail.
      // The "enabled" switch above provides two defaults which can be
      // tweaked here. If "details" is given, "enabled" can be omitted.
      "details": {
        // The peephole optimizer is always on if no details are given,
        // use details to switch it off.
        "peephole": true,
        // The inliner is always on if no details are given,
        // use details to switch it off.
        "inliner": true,
        // The unused jumpdest remover is always on if no details are given,
        // use details to switch it off.
        "jumpdestRemover": true,
```

(continué en la próxima página)

(proviene de la página anterior)

```
        // Sometimes re-orders literals in commutative operations.
        "orderLiterals": false,
        // Removes duplicate code blocks
        "deduplicate": false,
        // Common subexpression elimination, this is the most complicated step but
        // can also provide the largest gain.
        "cse": false,
        // Optimize representation of literal numbers and strings in code.
        "constantOptimizer": false,
        // The new Yul optimizer. Mostly operates on the code of ABI coder v2
        // and inline assembly.
        // It is activated together with the global optimizer setting
        // and can be deactivated here.
        // Before Solidity 0.6.0 it had to be activated through this switch.
        "yul": false,
        // Tuning options for the Yul optimizer.
        "yulDetails": {
          // Improve allocation of stack slots for variables, can free up stack slots␣
↪early.
          // Activated by default if the Yul optimizer is activated.
          "stackAllocation": true,
          // Select optimization steps to be applied. It is also possible to modify both␣
↪the
          // optimization sequence and the clean-up sequence. Instructions for each␣
↪sequence
          // are separated with the ":" delimiter and the values are provided in the␣
↪form of
          // optimization-sequence:clean-up-sequence. For more information see
          // "The Optimizer > Selecting Optimizations".
          // This field is optional, and if not provided, the default sequences for both
          // optimization and clean-up are used. If only one of the options is provivded
          // the other will not be run.
          // If only the delimiter ":" is provided then neither the optimization nor the␣
↪clean-up
          // sequence will be run.
          // If set to an empty value, only the default clean-up sequence is used and
          // no optimization steps are applied.
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    // Version of the EVM to compile for.
    // Affects type checking and code generation. Can be homestead,
    // tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg, istanbul,
↪ berlin, london or paris
    "evmVersion": "byzantium",
    // Optional: Change compilation pipeline to go through the Yul intermediate␣
↪representation.
    // This is false by default.
    "viaIR": true,
    // Optional: Debugging settings
    "debug": {
```

(continué en la próxima página)

(proviene de la página anterior)

```
      // How to treat revert (and require) reason strings. Settings are
      // "default", "strip", "debug" and "verboseDebug".
      // "default" does not inject compiler-generated revert strings and keeps user-
↪supplied ones.
      // "strip" removes all revert strings (if possible, i.e. if literals are used)␣
↪keeping side-effects
      // "debug" injects strings for compiler-generated internal reverts, implemented␣
↪for ABI encoders V1 and V2 for now.
      // "verboseDebug" even appends further information to user-supplied revert strings␣
↪(not yet implemented)
      "revertStrings": "default",
      // Optional: How much extra debug information to include in comments in the␣
↪produced EVM
      // assembly and Yul code. Available components are:
      // - `location`: Annotations of the form `@src <index>:<start>:<end>` indicating the
      //    location of the corresponding element in the original Solidity file, where:
      //     - `<index>` is the file index matching the `@use-src` annotation,
      //     - `<start>` is the index of the first byte at that location,
      //     - `<end>` is the index of the first byte after that location.
      // - `snippet`: A single-line code snippet from the location indicated by `@src`.
      //     The snippet is quoted and follows the corresponding `@src` annotation.
      // - `*`: Wildcard value that can be used to request everything.
      "debugInfo": ["location", "snippet"]
    },
    // Metadata settings (optional)
    "metadata": {
      // The CBOR metadata is appended at the end of the bytecode by default.
      // Setting this to false omits the metadata from the runtime and deploy time code.
      "appendCBOR": true,
      // Use only literal content and not URLs (false by default)
      "useLiteralContent": true,
      // Use the given hash method for the metadata hash that is appended to the␣
↪bytecode.
      // The metadata hash can be removed from the bytecode via option "none".
      // The other options are "ipfs" and "bzzr1".
      // If the option is omitted, "ipfs" is used by default.
      "bytecodeHash": "ipfs"
    },
    // Addresses of the libraries. If not all libraries are given here,
    // it can result in unlinked objects whose output data is different.
    "libraries": {
      // The top level key is the the name of the source file where the library is used.
      // If remappings are used, this source file should match the global path
      // after remappings were applied.
      // If this key is an empty string, that refers to a global level.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    },
    // The following can be used to select desired outputs based
    // on file and contract names.
    // If this field is omitted, then the compiler loads and does type checking,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // but will not generate any outputs apart from errors.
    // The first level key is the file name and the second level key is the contract␣
↪name.
    // An empty contract name is used for outputs that are not tied to a contract
    // but to the whole source file like the AST.
    // A star as contract name refers to all contracts in the file.
    // Similarly, a star as a file name matches all files.
    // To select all outputs the compiler can possibly generate, use
    // "outputSelection: { "*": { "*": [ "*" ], "": [ "*" ] } }"
    // but note that this might slow down the compilation process needlessly.
    //
    // The available output types are as follows:
    //
    // File level (needs empty string as contract name):
    //   ast - AST of all source files
    //
    // Contract level (needs the contract name or "*"):
    //   abi - ABI
    //   devdoc - Developer documentation (natspec)
    //   userdoc - User documentation (natspec)
    //   metadata - Metadata
    //   ir - Yul intermediate representation of the code before optimization
    //   irOptimized - Intermediate representation after optimization
    //   storageLayout - Slots, offsets and types of the contract's state variables.
    //   evm.assembly - New assembly format
    //   evm.legacyAssembly - Old-style assembly format in JSON
    //   evm.bytecode.functionDebugData - Debugging information at function level
    //   evm.bytecode.object - Bytecode object
    //   evm.bytecode.opcodes - Opcodes list
    //   evm.bytecode.sourceMap - Source mapping (useful for debugging)
    //   evm.bytecode.linkReferences - Link references (if unlinked object)
    //   evm.bytecode.generatedSources - Sources generated by the compiler
    //   evm.deployedBytecode* - Deployed bytecode (has all the options that evm.
↪bytecode has)
    //   evm.deployedBytecode.immutableReferences - Map from AST ids to bytecode ranges␣
↪that reference immutables
    //   evm.methodIdentifiers - The list of function hashes
    //   evm.gasEstimates - Function gas estimates
    //   ewasm.wast - Ewasm in WebAssembly S-expressions format
    //   ewasm.wasm - Ewasm in WebAssembly binary format
    //
    // Note that using a using `evm`, `evm.bytecode`, `ewasm`, etc. will select every
    // target part of that output. Additionally, `*` can be used as a wildcard to request␣
↪everything.
    //
    "outputSelection": {
      "*": {
        "*": [
          "metadata", "evm.bytecode" // Enable the metadata and bytecode outputs of␣
↪every single contract.
          , "evm.bytecode.sourceMap" // Enable the source map output of every single␣
↪contract.
```

(continué en la próxima página)

(proviene de la página anterior)

```
        ],
        "": [
          "ast" // Enable the AST output of every single file.
        ]
      },
      // Enable the abi and opcodes output of MyContract defined in file def.
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      }
    },
    // The modelChecker object is experimental and subject to changes.
    "modelChecker":
    {
      // Chose which contracts should be analyzed as the deployed one.
      "contracts":
      {
        "source1.sol": ["contract1"],
        "source2.sol": ["contract2", "contract3"]
      },
      // Choose how division and modulo operations should be encoded.
      // When using `false` they are replaced by multiplication with slack
      // variables. This is the default.
      // Using `true` here is recommended if you are using the CHC engine
      // and not using Spacer as the Horn solver (using Eldarica, for example).
      // See the Formal Verification section for a more detailed explanation of this␣
↪option.
      "divModNoSlacks": false,
      // Choose which model checker engine to use: all (default), bmc, chc, none.
      "engine": "chc",
      // Choose whether external calls should be considered trusted in case the
      // code of the called function is available at compile-time.
      // For details see the SMTChecker section.
      "extCalls": "trusted",
      // Choose which types of invariants should be reported to the user: contract,␣
↪reentrancy.
      "invariants": ["contract", "reentrancy"],
      // Choose whether to output all unproved targets. The default is `false`.
      "showUnproved": true,
      // Choose which solvers should be used, if available.
      // See the Formal Verification section for the solvers description.
      "solvers": ["cvc4", "smtlib2", "z3"],
      // Choose which targets should be checked: constantCondition,
      // underflow, overflow, divByZero, balance, assert, popEmptyArray, outOfBounds.
      // If the option is not given all targets are checked by default,
      // except underflow/overflow for Solidity >=0.8.7.
      // See the Formal Verification section for the targets description.
      "targets": ["underflow", "overflow", "assert"],
      // Timeout for each SMT query in milliseconds.
      // If this option is not given, the SMTChecker will use a deterministic
      // resource limit by default.
      // A given timeout of 0 means no resource/time restrictions for any query.
      "timeout": 20000
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }
  }
}
```

### Output Description

```
{
  // Optional: not present if no errors/warnings/infos were encountered
  "errors": [
    {
      // Optional: Location within the source file.
      "sourceLocation": {
        "file": "sourceFile.sol",
        "start": 0,
        "end": 100
      },
      // Optional: Further locations (e.g. places of conflicting declarations)
      "secondarySourceLocations": [
        {
          "file": "sourceFile.sol",
          "start": 64,
          "end": 92,
          "message": "Other declaration is here:"
        }
      ],
      // Mandatory: Error type, such as "TypeError", "InternalCompilerError", "Exception
↪", etc.
      // See below for complete list of types.
      "type": "TypeError",
      // Mandatory: Component where the error originated, such as "general", "ewasm",␣
↪etc.
      "component": "general",
      // Mandatory ("error", "warning" or "info", but please note that this may be␣
↪extended in the future)
      "severity": "error",
      // Optional: unique code for the cause of the error
      "errorCode": "3141",
      // Mandatory
      "message": "Invalid keyword",
      // Optional: the message formatted with source location
      "formattedMessage": "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // This contains the file-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "sources": {
    "sourceFile.sol": {
      // Identifier of the source (used in source maps)
      "id": 1,
      // The AST object
```

(continué en la próxima página)

(proviene de la página anterior)

```
      "ast": {}
    }
  },
  // This contains the contract-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "contracts": {
    "sourceFile.sol": {
      // If the language used has no contract names, this field should equal to an empty␣
↪string.
      "ContractName": {
        // The Ethereum Contract ABI. If empty, it is represented as an empty array.
        // See https://docs.soliditylang.org/en/develop/abi-spec.html
        "abi": [],
        // See the Metadata Output documentation (serialised JSON string)
        "metadata": "{/* ... */}",
        // User documentation (natspec)
        "userdoc": {},
        // Developer documentation (natspec)
        "devdoc": {},
        // Intermediate representation (string)
        "ir": "",
        // See the Storage Layout documentation.
        "storageLayout": {"storage": [/* ... */], "types": {/* ... */} },
        // EVM-related outputs
        "evm": {
          // Assembly (string)
          "assembly": "",
          // Old-style assembly (object)
          "legacyAssembly": {},
          // Bytecode and related details.
          "bytecode": {
            // Debugging data at the level of functions.
            "functionDebugData": {
              // Now follows a set of functions including compiler-internal and
              // user-defined function. The set does not have to be complete.
              "@mint_13": { // Internal name of the function
                "entryPoint": 128, // Byte offset into the bytecode where the function␣
↪starts (optional)
                "id": 13, // AST ID of the function definition or null for compiler-
↪internal functions (optional)
                "parameterSlots": 2, // Number of EVM stack slots for the function␣
↪parameters (optional)
                "returnSlots": 1 // Number of EVM stack slots for the return values␣
↪(optional)
              }
            },
            // The bytecode as a hex string.
            "object": "00fe",
            // Opcodes list (string)
            "opcodes": "",
            // The source mapping as a string. See the source mapping definition.
            "sourceMap": "",
```

(continué en la próxima página)

(proviene de la página anterior)

```
            // Array of sources generated by the compiler. Currently only
            // contains a single Yul file.
            "generatedSources": [{
              // Yul AST
              "ast": {/* ... */},
              // Source file in its text form (may contain comments)
              "contents":"{ function abi_decode(start, end) -> data { data :=␣
↪calldataload(start) } }",
              // Source file ID, used for source references, same "namespace" as the␣
↪Solidity source files
              "id": 2,
              "language": "Yul",
              "name": "#utility.yul"
            }],
            // If given, this is an unlinked object.
            "linkReferences": {
              "libraryFile.sol": {
                // Byte offsets into the bytecode.
                // Linking replaces the 20 bytes located there.
                "Library1": [
                  { "start": 0, "length": 20 },
                  { "start": 200, "length": 20 }
                ]
              }
            }
          },
          "deployedBytecode": {
            /* ..., */ // The same layout as above.
            "immutableReferences": {
              // There are two references to the immutable with AST ID 3, both 32 bytes␣
↪long. One is
              // at bytecode offset 42, the other at bytecode offset 80.
              "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
            }
          },
          // The list of function hashes
          "methodIdentifiers": {
            "delegate(address)": "5c19a95c"
          },
          // Function gas estimates
          "gasEstimates": {
            "creation": {
              "codeDepositCost": "420000",
              "executionCost": "infinite",
              "totalCost": "infinite"
            },
            "external": {
              "delegate(address)": "25000"
            },
            "internal": {
              "heavyLifting()": "infinite"
            }
```

(continué en la próxima página)

(proviene de la página anterior)

```
          }
        },
        // Ewasm related outputs
        "ewasm": {
          // S-expressions format
          "wast": "",
          // Binary format (hex string)
          "wasm": ""
        }
      }
    }
  }
}
```

### Error Types

1. `JSONError`: JSON input doesn't conform to the required format, e.g. input is not a JSON object, the language is not supported, etc.
2. `IOError`: IO and import processing errors, such as unresolvable URL or hash mismatch in supplied sources.
3. `ParserError`: Source code doesn't conform to the language rules.
4. `DocstringParsingError`: The NatSpec tags in the comment block cannot be parsed.
5. `SyntaxError`: Syntactical error, such as `continue` is used outside of a `for` loop.
6. `DeclarationError`: Invalid, unresolvable or clashing identifier names. e.g. `Identifier not found`
7. `TypeError`: Error within the type system, such as invalid type conversions, invalid assignments, etc.
8. `UnimplementedFeatureError`: Feature is not supported by the compiler, but is expected to be supported in future versions.
9. `InternalCompilerError`: Internal bug triggered in the compiler - this should be reported as an issue.
10. `Exception`: Unknown failure during compilation - this should be reported as an issue.
11. `CompilerError`: Invalid use of the compiler stack - this should be reported as an issue.
12. `FatalError`: Fatal error not processed correctly - this should be reported as an issue.
13. `YulException`: Error during Yul Code generation - this should be reported as an issue.
14. `Warning`: A warning, which didn't stop the compilation, but should be addressed if possible.
15. `Info`: Information that the compiler thinks the user might find useful, but is not dangerous and does not necessarily need to be addressed.

## 3.14 Analysing the Compiler Output

It is often useful to look at the assembly code generated by the compiler. The generated binary, i.e., the output of `solc --bin contract.sol`, is generally difficult to read. It is recommended to use the flag `--asm` to analyse the assembly output. Even for large contracts, looking at a visual diff of the assembly before and after a change is often very enlightening.

Consider the following contract (named, say `contract.sol`):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function one() public pure returns (uint) {
        return 1;
    }
}
```

The following would be the output of `solc --asm contract.sol`

```
======= contract.sol:C =======
EVM assembly:
    /* "contract.sol":0:86  contract C {... */
  mstore(0x40, 0x80)
  callvalue
  dup1
  iszero
  tag_1
  jumpi
  0x00
  dup1
  revert
tag_1:
  pop
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x00
  codecopy
  0x00
  return
stop

sub_0: assembly {
        /* "contract.sol":0:86  contract C {... */
      mstore(0x40, 0x80)
      callvalue
      dup1
      iszero
      tag_1
      jumpi
      0x00
      dup1
      revert
```

(continué en la próxima página)

(proviene de la página anterior)

```
    tag_1:
      pop
      jumpi(tag_2, lt(calldatasize, 0x04))
      shr(0xe0, calldataload(0x00))
      dup1
      0x901717d1
      eq
      tag_3
      jumpi
    tag_2:
      0x00
      dup1
      revert
        /* "contract.sol":17:84  function one() public pure returns (uint) {... */
    tag_3:
      tag_4
      tag_5
      jump  // in
    tag_4:
      mload(0x40)
      tag_6
      swap2
      swap1
      tag_7
      jump  // in
    tag_6:
      mload(0x40)
      dup1
      swap2
      sub
      swap1
      return
    tag_5:
        /* "contract.sol":53:57  uint */
      0x00
        /* "contract.sol":76:77  1 */
      0x01
        /* "contract.sol":69:77  return 1 */
      swap1
      pop
        /* "contract.sol":17:84  function one() public pure returns (uint) {... */
      swap1
      jump  // out
        /* "#utility.yul":7:125   */
    tag_10:
        /* "#utility.yul":94:118   */
      tag_12
        /* "#utility.yul":112:117   */
      dup2
        /* "#utility.yul":94:118   */
      tag_13
      jump  // in
```

(continué en la próxima página)

(proviene de la página anterior)

```
    tag_12:
        /* "#utility.yul":89:92   */
      dup3
        /* "#utility.yul":82:119   */
      mstore
        /* "#utility.yul":72:125   */
      pop
      pop
      jump  // out
        /* "#utility.yul":131:353   */
    tag_7:
      0x00
        /* "#utility.yul":262:264   */
      0x20
        /* "#utility.yul":251:260   */
      dup3
        /* "#utility.yul":247:265   */
      add
        /* "#utility.yul":239:265   */
      swap1
      pop
        /* "#utility.yul":275:346   */
      tag_15
        /* "#utility.yul":343:344   */
      0x00
        /* "#utility.yul":332:341   */
      dup4
        /* "#utility.yul":328:345   */
      add
        /* "#utility.yul":319:325   */
      dup5
        /* "#utility.yul":275:346   */
      tag_10
      jump  // in
    tag_15:
        /* "#utility.yul":229:353   */
      swap3
      swap2
      pop
      pop
      jump  // out
        /* "#utility.yul":359:436   */
    tag_13:
      0x00
        /* "#utility.yul":425:430   */
      dup2
        /* "#utility.yul":414:430   */
      swap1
      pop
        /* "#utility.yul":404:436   */
      swap2
      swap1
```

(continué en la próxima página)

(proviene de la página anterior)

```
      pop
      jump  // out

    auxdata:␣
↪0xa2646970667358221220a5874f19737ddd4c5d77ace1619e5160c67b3d4bedac75fce908fed32d98899864736f6c6378273
}
```

Alternatively, the above output can also be obtained from Remix, under the option «Compilation Details» after compiling a contract.

Notice that the `asm` output starts with the creation / constructor code. The deploy code is provided as part of the sub object (in the above example, it is part of the sub-object `sub_0`). The `auxdata` field corresponds to the contract *metadata*. The comments in the assembly output point to the source location. Note that `#utility.yul` is an internally generated file of utility functions that can be obtained using the flags `--combined-json generated-sources, generated-sources-runtime`.

Similarly, the optimized assembly can be obtained with the command: `solc --optimize --asm contract.sol`. Often times, it is interesting to see if two different sources in Solidity result in the same optimized code. For example, to see if the expressions `(a * b) / c`, `a * b / c` generates the same bytecode. This can be easily done by taking a `diff` of the corresponding assembly output, after potentially stripping comments that reference the source locations.

---

**Nota:** The `--asm` output is not designed to be machine readable. Therefore, there may be breaking changes on the output between minor versions of solc.

---

## 3.15 Solidity IR-based Codegen Changes

Solidity can generate EVM bytecode in two different ways: Either directly from Solidity to EVM opcodes («old codegen») or through an intermediate representation («IR») in Yul («new codegen» or «IR-based codegen»).

The IR-based code generator was introduced with an aim to not only allow code generation to be more transparent and auditable but also to enable more powerful optimization passes that span across functions.

You can enable it on the command line using `--via-ir` or with the option `{"viaIR": true}` in standard-json and we encourage everyone to try it out!

For several reasons, there are tiny semantic differences between the old and the IR-based code generator, mostly in areas where we would not expect people to rely on this behaviour anyway. This section highlights the main differences between the old and the IR-based codegen.

### 3.15.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- The order of state variable initialization has changed in case of inheritance.

  The order used to be:

  - All state variables are zero-initialized at the beginning.
  - Evaluate base constructor arguments from most derived to most base contract.
  - Initialize all state variables in the whole inheritance hierarchy from most base to most derived.
  - Run the constructor, if present, for all contracts in the linearized hierarchy from most base to most derived.

New order:

- All state variables are zero-initialized at the beginning.
- Evaluate base constructor arguments from most derived to most base contract.
- For every contract in order from most base to most derived in the linearized hierarchy:
  1. Initialize state variables.
  2. Run the constructor (if present).

This causes differences in contracts where the initial value of a state variable relies on the result of the constructor in another contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract A {
    uint x;
    constructor() {
        x = 42;
    }
    function f() public view returns(uint256) {
        return x;
    }
}
contract B is A {
    uint public y = f();
}
```

Previously, `y` would be set to 0. This is due to the fact that we would first initialize state variables: First, `x` is set to 0, and when initializing `y`, `f()` would return 0 causing `y` to be 0 as well. With the new rules, `y` will be set to 42. We first initialize `x` to 0, then call A's constructor which sets `x` to 42. Finally, when initializing `y`, `f()` returns 42 causing `y` to be 42.

- When storage structs are deleted, every storage slot that contains a member of the struct is set to zero entirely. Formerly, padding space was left untouched. Consequently, if the padding space within a struct is used to store data (e.g. in the context of a contract upgrade), you have to be aware that `delete` will now also clear the added member (while it wouldn't have been cleared in the past).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract C {
    struct S {
        uint64 y;
        uint64 z;
    }
    S s;
    function f() public {
        // ...
        delete s;
        // s occupies only first 16 bytes of the 32 bytes slot
        // delete will write zero to the full slot
    }
}
```

We have the same behavior for implicit delete, for example when array of structs is shortened.

- Function modifiers are implemented in a slightly different way regarding function parameters and return variables. This especially has an effect if the placeholder `_;` is evaluated multiple times in a modifier. In the old code generator, each function parameter and return variable has a fixed slot on the stack. If the function is run multiple times because `_;` is used multiple times or used in a loop, then a change to the function parameter's or return variable's value is visible in the next execution of the function. The new code generator implements modifiers using actual functions and passes function parameters on. This means that multiple evaluations of a function's body will get the same values for the parameters, and the effect on return variables is that they are reset to their default (zero) value for each execution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0;
contract C {
    function f(uint a) public pure mod() returns (uint r) {
        r = a++;
    }
    modifier mod() { _; _; }
}
```

If you execute `f(0)` in the old code generator, it will return 1, while it will return `0` when using the new code generator.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract C {
    bool active = true;
    modifier mod()
    {
        _;
        active = false;
        _;
    }
    function foo() external mod() returns (uint ret)
    {
        if (active)
            ret = 1; // Same as ``return 1``
    }
}
```

The function `C.foo()` returns the following values:

- Old code generator: `1` as the return variable is initialized to `0` only once before the first `_;` evaluation and then overwritten by the `return 1;`. It is not initialized again for the second `_;` evaluation and `foo()` does not explicitly assign it either (due to `active == false`), thus it keeps its first value.
- New code generator: `0` as all parameters, including return parameters, will be re-initialized before each `_;` evaluation.

- For the old code generator, the evaluation order of expressions is unspecified. For the new code generator, we try to evaluate in source order (left to right), but do not guarantee it. This can lead to semantic differences.

  For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function preincr_u8(uint8 a) public pure returns (uint8) {
        return ++a + a;
    }
}
```

The function `preincr_u8(1)` returns the following values:

- Old code generator: 3 (`1 + 2`) but the return value is unspecified in general
- New code generator: 4 (`2 + 2`) but the return value is not guaranteed

On the other hand, function argument expressions are evaluated in the same order by both code generators with the exception of the global functions `addmod` and `mulmod`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function add(uint8 a, uint8 b) public pure returns (uint8) {
        return a + b;
    }
    function g(uint8 a, uint8 b) public pure returns (uint8) {
        return add(++a + ++b, a + b);
    }
}
```

The function `g(1, 2)` returns the following values:

- Old code generator: `10` (`add(2 + 3, 2 + 3)`) but the return value is unspecified in general
- New code generator: `10` but the return value is not guaranteed

The arguments to the global functions `addmod` and `mulmod` are evaluated right-to-left by the old code generator and left-to-right by the new code generator. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f() public pure returns (uint256 aMod, uint256 mMod) {
        uint256 x = 3;
        // Old code gen: add/mulmod(5, 4, 3)
        // New code gen: add/mulmod(4, 5, 5)
        aMod = addmod(++x, ++x, x);
        mMod = mulmod(++x, ++x, x);
    }
}
```

The function `f()` returns the following values:

- Old code generator: `aMod = 0` and `mMod = 2`
- New code generator: `aMod = 4` and `mMod = 0`

- The new code generator imposes a hard limit of `type(uint64).max` (`0xffffffffffffffff`) for the free memory pointer. Allocations that would increase its value beyond this limit revert. The old code generator does not have this limit.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.8.0;
contract C {
    function f() public {
        uint[] memory arr;
        // allocation size: 576460752303423481
        // assumes freeMemPtr points to 0x80 initially
        uint solYulMaxAllocationBeforeMemPtrOverflow = (type(uint64).max - 0x80 -␣
↪31) / 32;
        // freeMemPtr overflows UINT64_MAX
        arr = new uint[](solYulMaxAllocationBeforeMemPtrOverflow);
    }
}
```

The function *f()* behaves as follows:

- Old code generator: runs out of gas while zeroing the array contents after the large memory allocation
- New code generator: reverts due to free memory pointer overflow (does not run out of gas)

## 3.15.2 Internals

### Internal function pointers

The old code generator uses code offsets or tags for values of internal function pointers. This is especially complicated since these offsets are different at construction time and after deployment and the values can cross this border via storage. Because of that, both offsets are encoded at construction time into the same value (into different bytes).

In the new code generator, function pointers use internal IDs that are allocated in sequence. Since calls via jumps are not possible, calls through function pointers always have to use an internal dispatch function that uses the `switch` statement to select the right function.

The ID `0` is reserved for uninitialized function pointers which then cause a panic in the dispatch function when called.

In the old code generator, internal function pointers are initialized with a special function that always causes a panic. This causes a storage write at construction time for internal function pointers in storage.

### Cleanup

The old code generator only performs cleanup before an operation whose result could be affected by the values of the dirty bits. The new code generator performs cleanup after any operation that can result in dirty bits. The hope is that the optimizer will be powerful enough to eliminate redundant cleanup operations.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f(uint8 a) public pure returns (uint r1, uint r2)
    {
        a = ~a;
        assembly {
            r1 := a
```

(continué en la próxima página)

(proviene de la página anterior)

```
        }
        r2 = a;
    }
}
```

The function `f(1)` returns the following values:

- Old code generator: (`ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe`, `0000000000000000000000000000000000000000000000000000000000000fe`)
- New code generator: (`00000000000000000000000000000000000000000000000000000000000000fe`, `0000000000000000000000000000000000000000000000000000000000000fe`)

Note that, unlike the new code generator, the old code generator does not perform a cleanup after the bit-not assignment (`a = ~a`). This results in different values being assigned (within the inline assembly block) to return value `r1` between the old and new code generators. However, both code generators perform a cleanup before the new value of `a` is assigned to `r2`.

## 3.16 Diseño de variables de estado en almacenamiento

Las variables de estado de loa contratos son almacen en almacenamiento en una forma compacta de modo que varios valores a veces utilizan la misma ranura de almacenamiento. Con la excepción para arrays y mapping que tiene los tamaños dinámicos (véase más adelante), datos se almacena de manera contigua elemento despues elemento que comienza con la primera variable estado, que se almacena en ranura `0`. Para cada variable, un tamaño en bytes se determina según su tipo. Varios, elementos contiguos que necesita menos que 32 bytes se empaquetan en una sola ranura de almacenamiento si es posible, de acuerdo con las siguientes reglas:

- El primer elemento en una ranura de almacenamiento se almacena alineado en orden inferior.
- Los tipos de valor se usa sólo tantos bytes como sean necesarios para almacenarlos.
- Si un tipo de valor no se ajusta a la parte restante de una ranura de almacenamiento, se almacena en el siguiente ranura de almacenamiento.
- Structs y el dato de array inician una nueva ranura y sus elementos se empaquetan firmemente de acuerdo con estas reglas.
- Items following struct or array data always start a new storage slot.
- Los elementos que siguen struct o el dato de array siempre inician una ranura de almacenamiento neuva.

Para los contratos que utilizan herencia, el orden de las variables de estado está determinado por el orden linealizado C3 de los contratos a partir del contrato más básico. Si las reglas anteriores permiten, las variables de estado de diferentes contratos comparten la misma ranura de almacenamiento.

Los elementos de structs y arrays se almacenan uno después del otro, como si se dieran como valores individuales.

**Advertencia:** Cuando se utilizan elementos de menos de 32 bytes, el uso de gas de su contrato puede ser mayor. Esto se debe a que EVM funciona con 32 bytes a la vez. Por lo tanto, si el elemento es más pequeño ademos, el EVM debe utilizar mas operaciones para reducir el tamaño del elemento de 32 bytes hasta el tamaño deseado.

Puede ser beneficioso utilizar tipos de tamaño reducido se se trata de valores de almacenamiento porque el compilador empaquetará varios elementos en una ranura de almacenamiento y, por tanto, combinará varias lecturas o escrituras en una solo operación. Si no está leyendo o escribiendo todos los valores de una ranura al mismo tiempo, este puede tener el efecto contrario, aunque: Cuando se escribe un valor en un de almacenamiento de varios valores

ranura, la ranura de almacenamiento debe leerse primero y, a continuacion, combinado con el nuevo valor de modo que no se destruyan otros datos de la misma ranura.

Cuando se trata de argumentos de función o valores de memoria, no hay ningún beneficio inherente porque el compilador no empaqueta estos valores.

Finalmente, para permitir que el EVM se optimice para esto, asegúrese de que intenta solicitar su variables de almacenamiento y miembros `struct` de tal forma que puedan ser empaquetados herméticamente. Por ejemplo, declarando las variables de almacenamiento en el orden de `uint128, uint128, uint256` en lugar de `uint128, uint256, uint128`, ya que el primero sólo ocupará dos ranuras de almacenamiento mientras que el este último ocupará tres.

**Nota:** El diseño de las variables de estado en el almacenamiento se considera parte de la interfaz external de Solidity debido al hecho de que los punteros de almacenamiento se pueden pasar a las bibliotecas. Esto significa que cualquier cambio en las reglas descritas en esta sección se considera un de cambio de ruptura de lenguaje y debido a su carácter crítico debe ser considerado muy cuidadosamente ante en ejecución. En caso de que se produjera en cambio tan importante, deseríamos publicar un modo de compatibilidad en el que el compilador generaría compatible con el diseño anterior.

### 3.16.1 Mapero y Matrices Dinámicas

Debido a sus tamaño impredecible, los mapeos y tipos de matriz de tamaño dinámico no se pueden almacenar «entre» los variables de estado que las preceden y siguen. En cambio, se considera que ocupan solo 32 bytes con respecto a *rules above* y los elementos que almacenan comenzando en una ranura de almacenamiento diferente que se calcula usando un hash Keccak-256.

Suponga que la ubicación de almacenamiento del mapeo o matrice termina siendo una ranura `p` después de aplicar *the storage layout rules*. Para matrices dinámicas, esta ranura se almenace el numero de elementos en el matrice (matrices de bytes y strings son excepciónes, consulte *below*). Para mapeos, la renura permanece vacía, pero sigue sinedo necesario para garantizar que, incluso si hay dos mapeos una al lado de la otra, su contenido termine en diferentes ubicaciones de almacenamiento.

Los datos de la matriz se ubican a partir de `keccak256(p)` y se presentan de la misma manera que lo harían los datos de matriz de tamaño estático: un elemento tras otro, potencialmente compartiendo ranuras de almacenamiento si los elementos no tienen más de 16 bytes. Las matrices dinámicas de matrices dinámicas aplican esta regla de forma recursiva. La ubicación del elemento `x[i][j]`, donde el tipo de `x` es `uint24[][]`, se calcula de la siguiente manera (de nuevo, suponiendo que `x` se almacena en la ranura `p`): La ranura es `keccak256(keccak256(p) i) floor(j / floor(256 / 24))` y el elemento se puede obtener de los datos de la ranura `v` usando `(v >> ((j % floor(256 / 24)) * 24)) & type(uint24).max`.

El valor correspondiente a una clave de mapero `k` se encuentra en `keccak256(h(k) . p)` donde `.` es concatenación y `h` es una función que se aplica a la clave dependiendo de su tipo:

- Para los tipos de valor, `h` rellena el valor a 32 bytes de la misma manera que cuando se almacena el valor en la memoria.
- Para strings y matrices de bytes, `h(k)` son solo los datos sin relleno.

Si el valor de mapero es un tipo sin valor, la ranura calculada marca el inicio de los datos. Si el valor es de tipo struct, por ejemplo, debe agregar un desplazamiento correspondiente al miembro struct para llegar al miembro.

Como ejemplo, considere el siguiente contrato:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;


contract C {
    struct S { uint16 a; uint16 b; uint256 c; }
    uint x;
    mapping(uint => mapping(uint => S)) data;
}
```

Calculemos la ubicación de almacenamiento de `data[4][9].c`. La posición de la asignación en sí es 1 (la variable `x` con 32 bytes la precede). Esto significa que `data[4]` se almacena en `keccak256(uint256(4) . uint256(1))`. El tipo de `data[4]` es de nuevo un mapeo y los datos para `data[4][9]` comienzan en la ranura `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1)))`. El desplazamiento de ranura del miembro `c` dentro de la estructura `S` es 1 porque `a` y `b` están empaquetados en una sola ranura. Esto significa que la ranura para `data[4][9].c` es `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) 1`. El tipo del valor es `uint256`, por lo que utiliza una sola ranura.

#### `bytes` y `string`

`bytes` y `string` están codificados de forma idéntica. En general, la codificación es similar a `bytes1[]`, en el sentido de que hay una ranura para la propia matriz y un área de datos que se calcula utilizando un hash `keccak256` de la posición de esa ranura. Sin embargo, para valores cortos (menos de 32 bytes) los elementos de matriz se almacenan junto con la longitud en la misma ranura.

En particular: si los datos tienen como máximo `31` bytes de longitud, los elementos se almacenan en los bytes de orden superior (alineados a la izquierda) y el byte de orden más bajo almacena el valor `length * 2`. Para las matrices de bytes que almacenan datos que tienen `32` o más bytes de longitud, la ranura principal `p` almacena `length * 2 + 1` y los datos se almacenan como de costumbre en `keccak256(p)`. Esto significa que puede distinguir una matriz corta de una matriz larga comprobando si se establece el bit más bajo: corto (no establecido) y largo (conjunto).

---

**Nota:** Actualmente no se admite el manejo de ranuras codificadas de forma no válida, pero es posible que se agreguen en el futuro. Si está compilando a través de IR, la lectura de una ranura codificada no válidamente da como resultado un error de `Panico(0x22)`.

---

### 3.16.2 Salida JSON

El diseño de almacenamiento de un contrato se puede solicitar a través de *standard JSON interface*. La salida es un objeto JSON que contiene dos claves, `storage` y `types`. El objeto `storage` es una matriz donde cada elemento tiene la siguiente forma:

```
{
    "astId": 2,
    "contract": "fileA:A",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

El ejemplo anterior es el diseño de almacenamiento de `contract A { uint x; }` de la unidad de origen `fileA` y

- `astId` es el identificador del nodo AST de la declaración de la variable de estado
- `contract` es el nombre del contrato incluyendo su ruta como prefijo
- `label` es el nombre de la variable de estado
- `offset` es el desplazamiento en bytes dentro de la ranura de almacenamiento según la codificación
- `slot` es la ranura de almacenamiento donde reside o se inicia la variable de estado. Este número puede ser muy grande y, por lo tanto, su valor JSON se representa como una cadena.
- `type` es un identificador utilizado como clave para la información de tipo de la variable (que se describe a continuación)

El `type` en esta caso `t_uint256` representa un elemento un `types`, que tiene la forma:

```
{
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32",
}
```

donde

- `encoding` cómo se codifican los datos en el almacenamiento, donde los valores posibles son:
  - `inplace`: Los datos se presentan de forma contigua en el almacenamiento (consulte *above*).
  - `mapping`: Método basado en hash Keccak-256 (consulte *above*).
  - `dynamic_array`: Método basado en hash Keccak-256 (consulte *above*).
  - `bytes`: una sola ranura o basado en hash Keccak-256 dependiendo del tamaño de los datos (consulte *above*).
- `label` es el nombre de tipo canónico.
- `numberOfBytes` es el número de bytes utilizados (como una cadena decimal). Tenga cuenta que si `numberOfBytes > 32` esto significa que se utiliza más de una ranura.

Algunos tipos tienen información adicional además de los cuatro anteriores. Los mapeos contienen sus tipos `key` y `value` (de nuevo haciendo referencia a una entrada en esta asignación de tipos), las matrices tienen su tipo `base` y las structs enumeran sus `members` en el mismo formato que el `storage` de nivel superior (consulte *above*).

---

**Nota:** El formato de salida JSON del diseño de almacenamiento de un contrato todavía se considera experimental y está sujeto a cambios en las versiones no rompedoras de Solidity.

---

El ejemplo siguiente muestra un contrato y su diseño de almacenamiento, que contiene tipos de valor y referencia, tipos codificados empaquetados y tipos anidados.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    uint x;
    uint y;
    S s;
    address addr;
    mapping(uint => mapping(address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}
```

```
{
  "storage": [
    {
      "astId": 15,
      "contract": "fileA:A",
      "label": "x",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
    },
    {
      "astId": 17,
      "contract": "fileA:A",
      "label": "y",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    },
    {
      "astId": 20,
      "contract": "fileA:A",
      "label": "s",
      "offset": 0,
      "slot": "2",
      "type": "t_struct(S)13_storage"
    },
    {
      "astId": 22,
      "contract": "fileA:A",
      "label": "addr",
      "offset": 0,
      "slot": "6",
      "type": "t_address"
    },
    {
      "astId": 28,
      "contract": "fileA:A",
      "label": "map",
      "offset": 0,
      "slot": "7",
      "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
    },
```

(continué en la próxima página)

(proviene de la página anterior)


```
    {
      "astId": 31,
      "contract": "fileA:A",
      "label": "array",
      "offset": 0,
      "slot": "8",
      "type": "t_array(t_uint256)dyn_storage"
    },
    {
      "astId": 33,
      "contract": "fileA:A",
      "label": "s1",
      "offset": 0,
      "slot": "9",
      "type": "t_string_storage"
    },
    {
      "astId": 35,
      "contract": "fileA:A",
      "label": "b1",
      "offset": 0,
      "slot": "10",
      "type": "t_bytes_storage"
    }
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    },
    "t_array(t_uint256)2_storage": {
      "base": "t_uint256",
      "encoding": "inplace",
      "label": "uint256[2]",
      "numberOfBytes": "64"
    },
    "t_array(t_uint256)dyn_storage": {
      "base": "t_uint256",
      "encoding": "dynamic_array",
      "label": "uint256[]",
      "numberOfBytes": "32"
    },
    "t_bool": {
      "encoding": "inplace",
      "label": "bool",
      "numberOfBytes": "1"
    },
    "t_bytes_storage": {
      "encoding": "bytes",
      "label": "bytes",
      "numberOfBytes": "32"
```


(continué en la próxima página)

(proviene de la página anterior)


```
    },
    "t_mapping(t_address,t_bool)": {
      "encoding": "mapping",
      "key": "t_address",
      "label": "mapping(address => bool)",
      "numberOfBytes": "32",
      "value": "t_bool"
    },
    "t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
      "encoding": "mapping",
      "key": "t_uint256",
      "label": "mapping(uint256 => mapping(address => bool))",
      "numberOfBytes": "32",
      "value": "t_mapping(t_address,t_bool)"
    },
    "t_string_storage": {
      "encoding": "bytes",
      "label": "string",
      "numberOfBytes": "32"
    },
    "t_struct(S)13_storage": {
      "encoding": "inplace",
      "label": "struct A.S",
      "members": [
        {
          "astId": 3,
          "contract": "fileA:A",
          "label": "a",
          "offset": 0,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 5,
          "contract": "fileA:A",
          "label": "b",
          "offset": 16,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 9,
          "contract": "fileA:A",
          "label": "staticArray",
          "offset": 0,
          "slot": "1",
          "type": "t_array(t_uint256)2_storage"
        },
        {
          "astId": 12,
          "contract": "fileA:A",
          "label": "dynArray",
```


(continué en la próxima página)

(proviene de la página anterior)

```
          "offset": 0,
          "slot": "3",
          "type": "t_array(t_uint256)dyn_storage"
        }
      ],
      "numberOfBytes": "128"
    },
    "t_uint128": {
      "encoding": "inplace",
      "label": "uint128",
      "numberOfBytes": "16"
    },
    "t_uint256": {
      "encoding": "inplace",
      "label": "uint256",
      "numberOfBytes": "32"
    }
  }
}
```

## 3.17 Diseño en memoria

Solidity reserva cuatro ranuras de 32 bytes, con rangos de bytes específicos (incluidos los extremos) que se utilizan de la siguiente manera:

- `0x00` - `0x3f` (64 bytes): espacio de desecho para métodos hash
- `0x40` - `0x5f` (32 bytes): tamaño de memoria asignado actualmente (aka. puntero de memoria libre)
- `0x60` - `0x7f` (32 bytes): ranura cero

El espacio de desecho se puede utilizar entre instrucciones (es decir, dentro de un assembly en línea). La ranura cero se utiliza como valor inicial para matrices de memoria dinámica y nunca debe escribirse en (el puntero de memoria libre apunta inicialmente a ``0x80`").

Solidity siempre coloca nuevos objetos en el puntero de memoria libre y la memoria nunca se libera (esto podría cambiar en el futuro).

Los elementos en matrices de memoria en Solidity siempre ocupan múltiplos de 32 bytes (esto es incluso cierto para `bytes1[]`, pero no para `bytes` y `string`). Las matrices de memoria multidimensionales son punteros a matrices de memoria. La longitud de una matriz dinámica se almacena en la primera ranura de la matriz y seguida de los elementos de la matriz.

**Advertencia:** Hay algunas operaciones en Solidity que necesitan un área de memoria temporal superior a 64 bytes y, por lo tanto, no cabe en el espacio de desecho. Se colocarán donde apunta la memoria libre, pero dado su corta duración, el puntero no se actualiza. Es posible que la memoria se ponga a cero o no. Debido a esto, uno no debería esperar que la memoria libre apunte a cero.

Aunque puede parecer una idea buena usar `msize` para llegar a un área de memoria puesta a cero definitivamente, el uso de tal puntero no-temporalmente sin actualizar el puntero de memoria libre puede tener resultados inesperados.

### 3.17.1 Diferencias con el diseño en el almacenamiento

Como se describió anteriormente, el diseño en la memoria es diferente del diseño en *storage*. A continuación hay algunos ejemplos.

#### Ejemplo de diferencia en matrices

La siguiente matriz ocupa 32 bytes (1 ranura) en almacenamiento, pero 128 bytes (4 elementos con 32 bytes cada uno) en memoria.

```
uint8[4] a;
```

#### Ejemplo de diferencia en el diseño de estructura

La siguiente estructura ocupa 96 bytes (3 ranuras de 32 bytes) en almacenamiento, pero 128 bytes (4 elementos con 32 bytes cada uno) en memoria.

```
struct S {
    uint a;
    uint b;
    uint8 c;
    uint8 d;
}
```

## 3.18 Diseño de los Datos de Llamadas

Se supone que los datos de entrada para una llamada a función están en el formato definido por *specification ABI*. Entre otros, la especificación ABI requiere que los argumentos se rellenen en múltiplos de 32 bytes. Las llamadas a funciones internas utilizan una convención diferente.

Los argumentos para el constructor de un contrato se anexan directamente al final del código del contrato, también en codificación ABI. El constructor accederá a ellos a través de un desplazamiento codificado, y no utilizando el opcode `codesize`, ya que esto cambia por supuesto al anexar datos al código.

## 3.19 Limpieza de Variables

Cuando un valor es inferior que a 256 bits, en algunos casos se deben limpiar los bits restantes. El compilador Solidity está diseñado para limpiar los bits restantes antes de cualquier operacion que pueda verse afectada negativamente por la basura potencial en los bits restantes. Por ejemplo, antes de escribir un valor en la memoria, los bits restantes deben borrarse porque el contenido de la memoria se puede usar para calcular hashes o enviarse como datos de una llamada de mensaje. Del mismo modo, antes de almacenar un valor en el almacenamiento, es necesario limpiar los bits restantes porque de lo contrario se puede observar el valor ilegible.

En última instancia, todos los valores en el EVM se almacenan en palabras de 256 bits. Por lo tanto, en algunos casos, cuando el tipo de un valor tiene menos de 256 bits, es necesario limpiar los bits restantes. El compilador Solidity está diseñado para realizar tal limpieza antes de cualquier operación que pueda verse afectada negativamente por la basura potencial en los bits restantes. Por ejemplo, antes de escribir un valor en la memoria, los bits restantes deben borrarse porque el contenido de la memoria se puede usar para calcular hashes o enviarse como datos de una llamada de mensaje. Del mismo modo, antes de almacenar un valor en el almacenamiento, los bits restantes deben limpiarse porque de lo contrario se puede observar el valor confuso.

Tenga en cuenta que el acceso a través del ensamblado en línea no se considera una operación de este tipo: Si utiliza un ensamblado en línea para acceder a variables de Solidity inferiores a 256 bits, el compilador no garantiza que el valor se limpie correctamente.

Además, no limpiamos los bits si la operación inmediatamente siguiente no se ve afectada. Por ejemplo, dado que cualquier valor distinto de cero se considera `true` por la instruccion `JUMPI`, no limpiamos los valores booleanos antes de que se usen como condición para `JUMPI`.

Además del principio de diseño anterior, el compilador Solidity limpia los datos de entrada cuando se cargan en la pila.

Los diferentes tipos tienen diferentes reglas para limpiar valores no válidos:

| Tipo | Valores Validos | Valores no Válidos Significan |
|---|---|---|
| enum of n miembros | 0 hasta n - 1 | excepción |
| bool | 0 or 1 | 1 |
| enteros con signo | firmar palabras extendidas | actualmente se envuelve silenciosamente; en el futuro se producirán excepciones |
| enteros sin signo | bits más altos puestos a cero | actualmente se envuelve silenciosamente; en el futuro se producirán excepciones |

La siguiente tabla describe las reglas de limpieza aplicadas a diferentes tipos, donde `bits superiores` se refiere a los bits restantes en caso de que el tipo tenga menos de 256 bits.

| Type | Valid Values | Cleanup of Invalid Values |
|---|---|---|
| enum de n miembros | 0 hasta n - 1 | produce una excepción |
| bool | 0 o 1 | resulta in 1 |
| enteros firmados | bits superiores establecidos en el bit con el signo | actualmente, en silencio, se extiende a un valor valido, es decir todos los bits superiores se. establacen en el bit de signo; puede producir una excepción en el futuro |
| enteros no firmados | bits superiores establecidos a cero | actualmente, en silencio, enmascara a un valorto a válido, es decir, todos los bits superiores establecen en cero; puede producir una excepción en el futuro |

Tenga en cuenta que los valores válidos y no válidos dependen de su tamaño de tipo. Considere `uint8`, el tipo de 8-bit sin signo, que tiene los siguientes valores válidos:

```
0000...0000 0000 0000
0000...0000 0000 0001
0000...0000 0000 0010
....
0000...0000 1111 1111
```

Cualquier valor no válido tendrá los bits más altos establecidos en cero:

```
0101...1101 0010 1010   invalid value
0000...0000 0010 1010   cleaned value
```

Para `int8`, el tipo de 8-bit firmado, los valores válidos son:

Negativo

```
1111...1111 1111 1111
1111...1111 1111 1110
....
1111...1111 1000 0000
```

Positivo

```
0000...0000 0000 0000
0000...0000 0000 0001
0000...0000 0000 0010
....
0000...0000 1111 1111
```

El compilador `signextend` el bit de signo, que es 1 para valores negativos y 0 para valores positivos, sobrescribiendo los bits superiores:

Negativo

```
0010...1010 1111 1111   valor inválido
1111...1111 1111 1111   valor limpiado
```

Positivo

```
1101...0101 0000 0100   valor inválido
0000...0000 0000 0100   valor limpiado
```

## 3.20 Asignaciones de origen

Como parte de la salida de AST, el compilador proporciona el rango del código fuente que está representado por el nodo respectivo en el AST. Esto se puede usar para varios propósitos, desde herramientas de análisis estático que informan errores en función del AST hasta herramientas de depuración que resaltan las variables locales y sus usos.

Además, el compilador también puede generar una asignación desde el código de bytes hasta el rango en el código fuente que generó la instrucción. Esto es de nuevo importante para las herramientas de análisis estático que operan a nivel de bytecode y para mostrar la posición actual en el código fuente dentro de un depurador o para el manejo de puntos de interrupción. Este mapeo también contiene otra información, como el tipo de salto y la profundidad del modificador (ver más abajo).

Ambos tipos de asignaciones de origen utilizan identificadores integer para referirse a los archivos de origen. El identificador de un archivo de origen se almacena en `output['sources'][sourceName]['id']` donde `output` es la salida del compilador standard-json analizada como JSON. Para algunas rutinas de utilidad, el compilador genera archivos de origen «internos» que no forman parte de la entrada original, pero a los que se hace referencia desde las asignaciones de origen. Estos archivos de origen junto con sus identificadores pueden obtenerse a través de `output['contracts'][sourceName][contractName]['evm']['bytecode']['generatedSources']`.

---

**Nota:** En el caso de instrucciones que no están asociadas a ningún archivo fuente en particular, la asignación de origen asigna un identificador integer de `-1`. Esto puede ocurrir para secciones de código de bytes que provienen de sentencias de ensamblaje en línea generadas por el compilador.

---

Las asignaciones de origen dentro del AST utilizan la siguiente notación:

`s:l:f`

Donde `s` es el byte-offset al inicio del rango en el archivo de origen, `l` es la longitud del rango de origen en bytes y `f` es el índice de origen mencionado anteriormente.

La codificación en la asignación de origen para el código de bytes es más complicada: Es una lista de `s:l:f:j:m` separados por `;`. Cada uno de estos elementos corresponde a una instrucción, es decir, no se puede utilizar el offset de bytes, sino que hay que utilizar el offset de instrucciones (las instrucciones push son más largas que un solo byte). Los campos `s`, `l` y `f` son como los anteriores. `j` puede ser `i`, `o` o `-` lo que significa que una instrucción de salto entra en una función, vuelve de una función o es un salto normal como parte de, por ejemplo, un loop. El último campo, `m`, es un integer que denota la «profundidad del modificador». Esta profundidad se incrementa cada vez que se introduce el marcador de posición (_) y disminuye cuando se vuelve a dejar. Esto permite a los debuggers rastrear casos complicados como el mismo modificador siendo utilizado dos o múltiples veces en marcadores de posición siendo usadas en un solo modificador.

Para comprimir estas asignaciones de origen, especialmente para bytecode, se utilizan las siguientes reglas:

- Si un campo está vacío, se utiliza el valor del elemento precedente.
- Si falta un `:`, todos los campos siguientes se consideran vacíos.

Esto significa que las siguientes asignaciones de origen representan la misma información:

```
1:2:1;1:9:1;2:1:2;2:1:2;2:1:2
```

```
1:2:1;:9;2:1:2;;
```

Es importante tener en cuenta que cuando se utiliza *verbatim*, las asignaciones de origen no serán válidas: El builtin se considera una única instrucción en lugar de potencialmente múltiples.

## 3.21 The Optimizer

The Solidity compiler uses two different optimizer modules: The «old» optimizer that operates at the opcode level and the «new» optimizer that operates on Yul IR code.

The opcode-based optimizer applies a set of simplification rules to opcodes. It also combines equal code sets and removes unused code.

The Yul-based optimizer is much more powerful, because it can work across function calls. For example, arbitrary jumps are not possible in Yul, so it is possible to compute the side-effects of each function. Consider two function calls, where the first does not modify storage and the second does modify storage. If their arguments and return values do not depend on each other, we can reorder the function calls. Similarly, if a function is side-effect free and its result is multiplied by zero, you can remove the function call completely.

Currently, the parameter `--optimize` activates the opcode-based optimizer for the generated bytecode and the Yul optimizer for the Yul code generated internally, for example for ABI coder v2. One can use `solc --ir-optimized --optimize` to produce an optimized Yul IR for a Solidity source. Similarly, one can use `solc --strict-assembly --optimize` for a stand-alone Yul mode.

---

**Nota:** The peephole optimizer and the inliner are always enabled by default and can only be turned off via the *Standard JSON*.

---

You can find more details on both optimizer modules and their optimization steps below.

### 3.21.1 Benefits of Optimizing Solidity Code

Overall, the optimizer tries to simplify complicated expressions, which reduces both code size and execution cost, i.e., it can reduce gas needed for contract deployment as well as for external calls made to the contract. It also specializes or inlines functions. Especially function inlining is an operation that can cause much bigger code, but it is often done because it results in opportunities for more simplifications.

### 3.21.2 Differences between Optimized and Non-Optimized Code

Generally, the most visible difference is that constant expressions are evaluated at compile time. When it comes to the ASM output, one can also notice a reduction of equivalent or duplicate code blocks (compare the output of the flags `--asm` and `--asm --optimize`). However, when it comes to the Yul/intermediate-representation, there can be significant differences, for example, functions may be inlined, combined, or rewritten to eliminate redundancies, etc. (compare the output between the flags `--ir` and `--optimize --ir-optimized`).

### 3.21.3 Optimizer Parameter Runs

The number of runs (`--optimize-runs`) specifies roughly how often each opcode of the deployed code will be executed across the life-time of the contract. This means it is a trade-off parameter between code size (deploy cost) and code execution cost (cost after deployment). A «runs» parameter of «1» will produce short but expensive code. In contrast, a larger «runs» parameter will produce longer but more gas efficient code. The maximum value of the parameter is `2**32-1`.

**Nota:** A common misconception is that this parameter specifies the number of iterations of the optimizer. This is not true: The optimizer will always run as many times as it can still improve the code.

### 3.21.4 Opcode-Based Optimizer Module

The opcode-based optimizer module operates on assembly code. It splits the sequence of instructions into basic blocks at `JUMPs` and `JUMPDESTs`. Inside these blocks, the optimizer analyzes the instructions and records every modification to the stack, memory, or storage as an expression which consists of an instruction and a list of arguments which are pointers to other expressions.

Additionally, the opcode-based optimizer uses a component called «CommonSubexpressionEliminator» that, amongst other tasks, finds expressions that are always equal (on every input) and combines them into an expression class. It first tries to find each new expression in a list of already known expressions. If no such matches are found, it simplifies the expression according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is a recursive process, we can also apply the latter rule if the second factor is a more complex expression which we know always evaluates to one.

Certain optimizer steps symbolically track the storage and memory locations. For example, this information is used to compute Keccak-256 hashes that can be evaluated during compile time. Consider the sequence:

```
PUSH 32
PUSH 0
CALLDATALOAD
PUSH 100
DUP2
MSTORE
KECCAK256
```

or the equivalent Yul

```
let x := calldataload(0)
mstore(x, 100)
let value := keccak256(x, 32)
```

In this case, the optimizer tracks the value at a memory location `calldataload(0)` and then realizes that the Keccak-256 hash can be evaluated at compile time. This only works if there is no other instruction that modifies memory between the `mstore` and `keccak256`. So if there is an instruction that writes to memory (or storage), then we need to erase the knowledge of the current memory (or storage). There is, however, an exception to this erasing, when we can easily see that the instruction doesn't write to a certain location.

For example,

```
let x := calldataload(0)
mstore(x, 100)
// Current knowledge memory location x -> 100
let y := add(x, 32)
// Does not clear the knowledge that x -> 100, since y does not write to [x, x + 32)
mstore(y, 200)
// This Keccak-256 can now be evaluated
let value := keccak256(x, 32)
```

Therefore, modifications to storage and memory locations, of say location `l`, must erase knowledge about storage or memory locations which may be equal to `l`. More specifically, for storage, the optimizer has to erase all knowledge of symbolic locations, that may be equal to `l` and for memory, the optimizer has to erase all knowledge of symbolic locations that may not be at least 32 bytes away. If `m` denotes an arbitrary location, then this decision on erasure is done by computing the value `sub(l, m)`. For storage, if this value evaluates to a literal that is non-zero, then the knowledge about `m` will be kept. For memory, if the value evaluates to a literal that is between `32` and `2**256 - 32`, then the knowledge about `m` will be kept. In all other cases, the knowledge about `m` will be erased.

After this process, we know which expressions have to be on the stack at the end, and have a list of modifications to memory and storage. This information is stored together with the basic blocks and is used to link them. Furthermore, knowledge about the stack, storage and memory configuration is forwarded to the next block(s).

If we know the targets of all `JUMP` and `JUMPI` instructions, we can build a complete control flow graph of the program. If there is only one target we do not know (this can happen as in principle, jump targets can be computed from inputs), we have to erase all knowledge about the input state of a block as it can be the target of the unknown `JUMP`. If the opcode-based optimizer module finds a `JUMPI` whose condition evaluates to a constant, it transforms it to an unconditional jump.

As the last step, the code in each block is re-generated. The optimizer creates a dependency graph from the expressions on the stack at the end of the block, and it drops every operation that is not part of this graph. It generates code that applies the modifications to memory and storage in the order they were made in the original code (dropping modifications which were found not to be needed). Finally, it generates all values that are required to be on the stack in the correct place.

These steps are applied to each basic block and the newly generated code is used as replacement if it is smaller. If a basic block is split at a `JUMPI` and during the analysis, the condition evaluates to a constant, the `JUMPI` is replaced based on the value of the constant. Thus code like

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2) // this condition is never true
  return 2;
else
  return 1;
```

simplifies to this:

```
data[7] = 9;
return 1;
```

### Simple Inlining

Since Solidity version 0.8.2, there is another optimizer step that replaces certain jumps to blocks containing «simple» instructions ending with a «jump» by a copy of these instructions. This corresponds to inlining of simple, small Solidity or Yul functions. In particular, the sequence `PUSHTAG(tag) JUMP` may be replaced, whenever the `JUMP` is marked as jump «into» a function and behind `tag` there is a basic block (as described above for the «CommonSubexpressionEliminator») that ends in another `JUMP` which is marked as a jump «out of» a function.

In particular, consider the following prototypical example of assembly generated for a call to an internal Solidity function:

```
  tag_return
  tag_f
  jump      // in
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

As long as the body of the function is a continuous basic block, the «Inliner» can replace `tag_f jump` by the block at `tag_f` resulting in:

```
  tag_return
  ...body of function f...
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

Now ideally, the other optimizer steps described above will result in the return tag push being moved towards the remaining jump resulting in:

```
  ...body of function f...
  tag_return
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

In this situation the «PeepholeOptimizer» will remove the return jump. Ideally, all of this can be done for all references to `tag_f` leaving it unused, s.t. it can be removed, yielding:

```
...body of function f...
...opcodes after call to f...
```

So the call to function `f` is inlined and the original definition of `f` can be removed.

Inlining like this is attempted, whenever a heuristics suggests that inlining is cheaper over the lifetime of a contract than not inlining. This heuristics depends on the size of the function body, the number of other references to its tag (approximating the number of calls to the function) and the expected number of executions of the contract (the global optimizer parameter «runs»).

### 3.21.5 Yul-Based Optimizer Module

The Yul-based optimizer consists of several stages and components that all transform the AST in a semantically equivalent way. The goal is to end up either with code that is shorter or at least only marginally longer but will allow further optimization steps.

**Advertencia:** Since the optimizer is under heavy development, the information here might be outdated. If you rely on a certain functionality, please reach out to the team directly.

The optimizer currently follows a purely greedy strategy and does not do any backtracking.

All components of the Yul-based optimizer module are explained below. The following transformation steps are the main components:

- SSA Transform
- Common Subexpression Eliminator
- Expression Simplifier
- Redundant Assign Eliminator
- Full Inliner

#### Optimizer Steps

This is a list of all steps the Yul-based optimizer sorted alphabetically. You can find more information on the individual steps and their sequence below.

| Abbreviation | Full name |
|---|---|
| `f` | *BlockFlattener* |
| `l` | *CircularReferencesPruner* |
| `c` | *CommonSubexpressionEliminator* |
| `C` | *ConditionalSimplifier* |
| `U` | *ConditionalUnsimplifier* |
| `n` | *ControlFlowSimplifier* |
| `D` | *DeadCodeEliminator* |
| `E` | *EqualStoreEliminator* |
| `v` | *EquivalentFunctionCombiner* |
| `e` | *ExpressionInliner* |
| `j` | *ExpressionJoiner* |
| `s` | *Expression Simplifier* |

continué en la próxima página

Tabla 1 – proviene de la página anterior

| Abbreviation | Full name |
|---|---|
| `x` | *ExpressionSplitter* |
| `I` | *ForLoopConditionIntoBody* |
| `O` | *ForLoopConditionOutOfBody* |
| `o` | *ForLoopInitRewriter* |
| `i` | *FullInliner* |
| `g` | *FunctionGrouper* |
| `h` | *FunctionHoister* |
| `F` | *FunctionSpecializer* |
| `T` | *LiteralRematerialiser* |
| `L` | *LoadResolver* |
| `M` | *LoopInvariantCodeMotion* |
| `r` | *RedundantAssignEliminator* |
| `R` | *ReasoningBasedSimplifier* - highly experimental |
| `m` | *Rematerialiser* |
| `V` | *SSAReverser* |
| `a` | *SSATransform* |
| `t` | *StructuralSimplifier* |
| `p` | *UnusedFunctionParameterPruner* |
| `S` | *UnusedStoreEliminator* |
| `u` | *UnusedPruner* |
| `d` | *VarDeclInitializer* |

Some steps depend on properties ensured by `BlockFlattener`, `FunctionGrouper`, `ForLoopInitRewriter`. For this reason the Yul optimizer always applies them before applying any steps supplied by the user.

The ReasoningBasedSimplifier is an optimizer step that is currently not enabled in the default set of steps. It uses an SMT solver to simplify arithmetic expressions and boolean conditions. It has not received thorough testing or validation yet and can produce non-reproducible results, so please use with care!

### Selecting Optimizations

By default the optimizer applies its predefined sequence of optimization steps to the generated assembly. You can override this sequence and supply your own using the `--yul-optimizations` option:

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul:fDnTOc'
```

The order of steps is significant and affects the quality of the output. Moreover, applying a step may uncover new optimization opportunities for others that were already applied, so repeating steps is often beneficial.

The sequence inside `[...]` will be applied multiple times in a loop until the Yul code remains unchanged or until the maximum number of rounds (currently 12) has been reached. Brackets (`[]`) may be used multiple times in a sequence, but can not be nested.

An important thing to note, is that there are some hardcoded steps that are always run before and after the user-supplied sequence, or the default sequence if one was not supplied by the user.

The cleanup sequence delimiter `:` is optional, and is used to supply a custom cleanup sequence in order to replace the default one. If omitted, the optimizer will simply apply the default cleanup sequence. In addition, the delimiter may be placed at the beginning of the user-supplied sequence, which will result in the optimization sequence being empty, whereas conversely, if placed at the end of the sequence, will be treated as an empty cleanup sequence.

### Preprocessing

The preprocessing components perform transformations to get the program into a certain normal form that is easier to work with. This normal form is kept during the rest of the optimization process.

### Disambiguator

The disambiguator takes an AST and returns a fresh copy where all identifiers have unique names in the input AST. This is a prerequisite for all other optimizer stages. One of the benefits is that identifier lookup does not need to take scopes into account which simplifies the analysis needed for other steps.

All subsequent stages have the property that all names stay unique. This means if a new identifier needs to be introduced, a new unique name is generated.

### FunctionHoister

The function hoister moves all function definitions to the end of the topmost block. This is a semantically equivalent transformation as long as it is performed after the disambiguation stage. The reason is that moving a definition to a higher-level block cannot decrease its visibility and it is impossible to reference variables defined in a different function.

The benefit of this stage is that function definitions can be looked up more easily and functions can be optimized in isolation without having to traverse the AST completely.

### FunctionGrouper

The function grouper has to be applied after the disambiguator and the function hoister. Its effect is that all topmost elements that are not function definitions are moved into a single block which is the first statement of the root block.

After this step, a program has the following normal form:

```
{ I F... }
```

Where `I` is a (potentially empty) block that does not contain any function definitions (not even recursively) and `F` is a list of function definitions such that no function contains a function definition.

The benefit of this stage is that we always know where the list of function begins.

### ForLoopConditionIntoBody

This transformation moves the loop-iteration condition of a for-loop into loop body. We need this transformation because *ExpressionSplitter* will not apply to iteration condition expressions (the `C` in the following example).

```
for { Init... } C { Post... } {
    Body...
}
```

is transformed to

```
for { Init... } 1 { Post... } {
    if iszero(C) { break }
    Body...
}
```

This transformation can also be useful when paired with `LoopInvariantCodeMotion`, since invariants in the loop-invariant conditions can then be taken outside the loop.

### ForLoopInitRewriter

This transformation moves the initialization part of a for-loop to before the loop:

```
for { Init... } C { Post... } {
    Body...
}
```

is transformed to

```
Init...
for {} C { Post... } {
    Body...
}
```

This eases the rest of the optimization process because we can ignore the complicated scoping rules of the for loop initialisation block.

### VarDeclInitializer

This step rewrites variable declarations so that all of them are initialized. Declarations like `let x, y` are split into multiple declaration statements.

Only supports initializing with the zero literal for now.

### Pseudo-SSA Transformation

The purpose of this components is to get the program into a longer form, so that other components can more easily work with it. The final representation will be similar to a static-single-assignment (SSA) form, with the difference that it does not make use of explicit «phi» functions which combines the values from different branches of control flow because such a feature does not exist in the Yul language. Instead, when control flow merges, if a variable is re-assigned in one of the branches, a new SSA variable is declared to hold its current value, so that the following expressions still only need to reference SSA variables.

An example transformation is the following:

```
{
    let a := calldataload(0)
    let b := calldataload(0x20)
    if gt(a, 0) {
        b := mul(b, 0x20)
    }
    a := add(a, 1)
    sstore(a, add(b, 0x20))
}
```

When all the following transformation steps are applied, the program will look as follows:

```
{
    let _1 := 0
    let a_9 := calldataload(_1)
    let a := a_9
    let _2 := 0x20
    let b_10 := calldataload(_2)
    let b := b_10
    let _3 := 0
    let _4 := gt(a_9, _3)
    if _4
    {
        let _5 := 0x20
        let b_11 := mul(b_10, _5)
        b := b_11
    }
    let b_12 := b
    let _6 := 1
    let a_13 := add(a_9, _6)
    let _7 := 0x20
    let _8 := add(b_12, _7)
    sstore(a_13, _8)
}
```

Note that the only variable that is re-assigned in this snippet is `b`. This re-assignment cannot be avoided because `b` has different values depending on the control flow. All other variables never change their value once they are defined. The advantage of this property is that variables can be freely moved around and references to them can be exchanged by their initial value (and vice-versa), as long as these values are still valid in the new context.

Of course, the code here is far from being optimized. To the contrary, it is much longer. The hope is that this code will be easier to work with and furthermore, there are optimizer steps that undo these changes and make the code more compact again at the end.

### ExpressionSplitter

The expression splitter turns expressions like `add(mload(0x123), mul(mload(0x456), 0x20))` into a sequence of declarations of unique variables that are assigned sub-expressions of that expression so that each function call has only variables as arguments.

The above would be transformed into

```
{
    let _1 := 0x20
    let _2 := 0x456
    let _3 := mload(_2)
    let _4 := mul(_3, _1)
    let _5 := 0x123
    let _6 := mload(_5)
    let z := add(_6, _4)
}
```

Note that this transformation does not change the order of opcodes or function calls.

It is not applied to loop iteration-condition, because the loop control flow does not allow this «outlining» of the inner expressions in all cases. We can sidestep this limitation by applying *ForLoopConditionIntoBody* to move the iteration

condition into loop body.

The final program should be in a form such that (with the exception of loop conditions) function calls cannot appear nested inside expressions and all function call arguments have to be variables.

The benefits of this form are that it is much easier to re-order the sequence of opcodes and it is also easier to perform function call inlining. Furthermore, it is simpler to replace individual parts of expressions or re-organize the «expression tree». The drawback is that such code is much harder to read for humans.

### SSATransform

This stage tries to replace repeated assignments to existing variables by declarations of new variables as much as possible. The reassignments are still there, but all references to the reassigned variables are replaced by the newly declared variables.

Example:

```
{
    let a := 1
    mstore(a, 2)
    a := 3
}
```

is transformed to

```
{
    let a_1 := 1
    let a := a_1
    mstore(a_1, 2)
    let a_3 := 3
    a := a_3
}
```

Exact semantics:

For any variable `a` that is assigned to somewhere in the code (variables that are declared with value and never re-assigned are not modified) perform the following transforms:

- replace `let a := v` by `let a_i := v let a := a_i`
- replace `a := v` by `let a_i := v a := a_i` where `i` is a number such that `a_i` is yet unused.

Furthermore, always record the current value of `i` used for `a` and replace each reference to `a` by `a_i`. The current value mapping is cleared for a variable `a` at the end of each block in which it was assigned to and at the end of the for loop init block if it is assigned inside the for loop body or post block. If a variable's value is cleared according to the rule above and the variable is declared outside the block, a new SSA variable will be created at the location where control flow joins, this includes the beginning of loop post/body block and the location right after If/Switch/ForLoop/Block statement.

After this stage, the Redundant Assign Eliminator is recommended to remove the unnecessary intermediate assignments.

This stage provides best results if the Expression Splitter and the Common Subexpression Eliminator are run right before it, because then it does not generate excessive amounts of variables. On the other hand, the Common Subexpression Eliminator could be more efficient if run after the SSA transform.

### RedundantAssignEliminator

The SSA transform always generates an assignment of the form `a := a_i`, even though these might be unnecessary in many cases, like the following example:

```
{
    let a := 1
    a := mload(a)
    a := sload(a)
    sstore(a, 1)
}
```

The SSA transform converts this snippet to the following:

```
{
    let a_1 := 1
    let a := a_1
    let a_2 := mload(a_1)
    a := a_2
    let a_3 := sload(a_2)
    a := a_3
    sstore(a_3, 1)
}
```

The Redundant Assign Eliminator removes all the three assignments to `a`, because the value of `a` is not used and thus turn this snippet into strict SSA form:

```
{
    let a_1 := 1
    let a_2 := mload(a_1)
    let a_3 := sload(a_2)
    sstore(a_3, 1)
}
```

Of course the intricate parts of determining whether an assignment is redundant or not are connected to joining control flow.

The component works as follows in detail:

The AST is traversed twice: in an information gathering step and in the actual removal step. During information gathering, we maintain a mapping from assignment statements to the three states «unused», «undecided» and «used» which signifies whether the assigned value will be used later by a reference to the variable.

When an assignment is visited, it is added to the mapping in the «undecided» state (see remark about for loops below) and every other assignment to the same variable that is still in the «undecided» state is changed to «unused». When a variable is referenced, the state of any assignment to that variable still in the «undecided» state is changed to «used».

At points where control flow splits, a copy of the mapping is handed over to each branch. At points where control flow joins, the two mappings coming from the two branches are combined in the following way: Statements that are only in one mapping or have the same state are used unchanged. Conflicting values are resolved in the following way:

- «unused», «undecided» -> «undecided»
- «unused», «used» -> «used»
- «undecided», «used» -> «used»

For for-loops, the condition, body and post-part are visited twice, taking the joining control-flow at the condition into account. In other words, we create three control flow paths: Zero runs of the loop, one run and two runs and then combine them at the end.

Simulating a third run or even more is unnecessary, which can be seen as follows:

A state of an assignment at the beginning of the iteration will deterministically result in a state of that assignment at the end of the iteration. Let this state mapping function be called `f`. The combination of the three different states `unused`, `undecided` and `used` as explained above is the `max` operation where `unused = 0`, `undecided = 1` and `used = 2`.

The proper way would be to compute

```
max(s, f(s), f(f(s)), f(f(f(s))), ...)
```

as state after the loop. Since `f` just has a range of three different values, iterating it has to reach a cycle after at most three iterations, and thus `f(f(f(s)))` has to equal one of `s`, `f(s)`, or `f(f(s))` and thus

```
max(s, f(s), f(f(s))) = max(s, f(s), f(f(s)), f(f(f(s))), ...).
```

In summary, running the loop at most twice is enough because there are only three different states.

For switch statements that have a «default»-case, there is no control-flow part that skips the switch.

When a variable goes out of scope, all statements still in the «undecided» state are changed to «unused», unless the variable is the return parameter of a function - there, the state changes to «used».

In the second traversal, all assignments that are in the «unused» state are removed.

This step is usually run right after the SSA transform to complete the generation of the pseudo-SSA.

### Tools

#### Movability

Movability is a property of an expression. It roughly means that the expression is side-effect free and its evaluation only depends on the values of variables and the call-constant state of the environment. Most expressions are movable. The following parts make an expression non-movable:

- function calls (might be relaxed in the future if all statements in the function are movable)
- opcodes that (can) have side-effects (like `call` or `selfdestruct`)
- opcodes that read or write memory, storage or external state information
- opcodes that depend on the current PC, memory size or returndata size

#### DataflowAnalyzer

The Dataflow Analyzer is not an optimizer step itself but is used as a tool by other components. While traversing the AST, it tracks the current value of each variable, as long as that value is a movable expression. It records the variables that are part of the expression that is currently assigned to each other variable. Upon each assignment to a variable `a`, the current stored value of `a` is updated and all stored values of all variables `b` are cleared whenever `a` is part of the currently stored expression for `b`.

At control-flow joins, knowledge about variables is cleared if they have or would be assigned in any of the control-flow paths. For instance, upon entering a for loop, all variables are cleared that will be assigned during the body or the post block.

### Expression-Scale Simplifications

These simplification passes change expressions and replace them by equivalent and hopefully simpler expressions.

### CommonSubexpressionEliminator

This step uses the Dataflow Analyzer and replaces subexpressions that syntactically match the current value of a variable by a reference to that variable. This is an equivalence transform because such subexpressions have to be movable.

All subexpressions that are identifiers themselves are replaced by their current value if the value is an identifier.

The combination of the two rules above allow to compute a local value numbering, which means that if two variables have the same value, one of them will always be unused. The Unused Pruner or the Redundant Assign Eliminator will then be able to fully eliminate such variables.

This step is especially efficient if the expression splitter is run before. If the code is in pseudo-SSA form, the values of variables are available for a longer time and thus we have a higher chance of expressions to be replaceable.

The expression simplifier will be able to perform better replacements if the common subexpression eliminator was run right before it.

### Expression Simplifier

The Expression Simplifier uses the Dataflow Analyzer and makes use of a list of equivalence transforms on expressions like `X + 0 -> X` to simplify the code.

It tries to match patterns like `X + 0` on each subexpression. During the matching procedure, it resolves variables to their currently assigned expressions to be able to match more deeply nested patterns even when the code is in pseudo-SSA form.

Some of the patterns like `X - X -> 0` can only be applied as long as the expression `X` is movable, because otherwise it would remove its potential side-effects. Since variable references are always movable, even if their current value might not be, the Expression Simplifier is again more powerful in split or pseudo-SSA form.

### LiteralRematerialiser

To be documented.

### LoadResolver

Optimisation stage that replaces expressions of type `sload(x)` and `mload(x)` by the value currently stored in storage resp. memory, if known.

Works best if the code is in SSA form.

Prerequisite: Disambiguator, ForLoopInitRewriter.

### ReasoningBasedSimplifier

This optimizer uses SMT solvers to check whether `if` conditions are constant.

- If `constraints AND condition` is UNSAT, the condition is never true and the whole body can be removed.
- If `constraints AND NOT condition` is UNSAT, the condition is always true and can be replaced by `1`.

The simplifications above can only be applied if the condition is movable.

It is only effective on the EVM dialect, but safe to use on other dialects.

Prerequisite: Disambiguator, SSATransform.

### Statement-Scale Simplifications

### CircularReferencesPruner

This stage removes functions that call each other but are neither externally referenced nor referenced from the outermost context.

### ConditionalSimplifier

The Conditional Simplifier inserts assignments to condition variables if the value can be determined from the control-flow.

Destroys SSA form.

Currently, this tool is very limited, mostly because we do not yet have support for boolean types. Since conditions only check for expressions being nonzero, we cannot assign a specific value.

Current features:

- switch cases: insert «<condition> := <caseLabel>»
- after if statement with terminating control-flow, insert «<condition> := 0»

Future features:

- allow replacements by «1»
- take termination of user-defined functions into account

Works best with SSA form and if dead code removal has run before.

Prerequisite: Disambiguator.

### ConditionalUnsimplifier

Reverse of Conditional Simplifier.

### ControlFlowSimplifier

Simplifies several control-flow structures:

- replace if with empty body with pop(condition)
- remove empty default switch case
- remove empty switch case if no default case exists
- replace switch with no cases with pop(expression)
- turn switch with single case into if
- replace switch with only default case with pop(expression) and body
- replace switch with const expr with matching case body
- replace `for` with terminating control flow and without other break/continue by `if`
- remove `leave` at the end of a function.

None of these operations depend on the data flow. The StructuralSimplifier performs similar tasks that do depend on data flow.

The ControlFlowSimplifier does record the presence or absence of `break` and `continue` statements during its traversal.

Prerequisite: Disambiguator, FunctionHoister, ForLoopInitRewriter. Important: Introduces EVM opcodes and thus can only be used on EVM code for now.

### DeadCodeEliminator

This optimization stage removes unreachable code.

Unreachable code is any code within a block which is preceded by a leave, return, invalid, break, continue, selfdestruct, revert or by a call to a user-defined function that recurses infinitely.

Function definitions are retained as they might be called by earlier code and thus are considered reachable.

Because variables declared in a for loop's init block have their scope extended to the loop body, we require ForLoopInitRewriter to run before this step.

Prerequisite: ForLoopInitRewriter, Function Hoister, Function Grouper

### EqualStoreEliminator

This steps removes `mstore(k, v)` and `sstore(k, v)` calls if there was a previous call to `mstore(k, v)` / `sstore(k, v)`, no other store in between and the values of `k` and `v` did not change.

This simple step is effective if run after the SSA transform and the Common Subexpression Eliminator, because SSA will make sure that the variables will not change and the Common Subexpression Eliminator re-uses exactly the same variable if the value is known to be the same.

Prerequisites: Disambiguator, ForLoopInitRewriter

### UnusedPruner

This step removes the definitions of all functions that are never referenced.

It also removes the declaration of variables that are never referenced. If the declaration assigns a value that is not movable, the expression is retained, but its value is discarded.

All movable expression statements (expressions that are not assigned) are removed.

### StructuralSimplifier

This is a general step that performs various kinds of simplifications on a structural level:

- replace if statement with empty body by `pop(condition)`
- replace if statement with true condition by its body
- remove if statement with false condition
- turn switch with single case into if
- replace switch with only default case by `pop(expression)` and body
- replace switch with literal expression by matching case body
- replace for loop with false condition by its initialization part

This component uses the Dataflow Analyzer.

### BlockFlattener

This stage eliminates nested blocks by inserting the statement in the inner block at the appropriate place in the outer block. It depends on the FunctionGrouper and does not flatten the outermost block to keep the form produced by the FunctionGrouper.

```
{
    {
        let x := 2
        {
            let y := 3
            mstore(x, y)
        }
    }
}
```

is transformed to

```
{
    {
        let x := 2
        let y := 3
        mstore(x, y)
    }
}
```

As long as the code is disambiguated, this does not cause a problem because the scopes of variables can only grow.

### LoopInvariantCodeMotion

This optimization moves movable SSA variable declarations outside the loop.

Only statements at the top level in a loop's body or post block are considered, i.e variable declarations inside conditional branches will not be moved out of the loop.

Requirements:

- The Disambiguator, ForLoopInitRewriter and FunctionHoister must be run upfront.
- Expression splitter and SSA transform should be run upfront to obtain better result.

## Function-Level Optimizations

### FunctionSpecializer

This step specializes the function with its literal arguments.

If a function, say, `function f(a, b) { sstore (a, b) }`, is called with literal arguments, for example, `f(x, 5)`, where `x` is an identifier, it could be specialized by creating a new function `f_1` that takes only one argument, i.e.,

```
function f_1(a_1) {
    let b_1 := 5
    sstore(a_1, b_1)
}
```

Other optimization steps will be able to make more simplifications to the function. The optimization step is mainly useful for functions that would not be inlined.

Prerequisites: Disambiguator, FunctionHoister

LiteralRematerialiser is recommended as a prerequisite, even though it's not required for correctness.

### UnusedFunctionParameterPruner

This step removes unused parameters in a function.

If a parameter is unused, like `c` and `y` in, `function f(a,b,c) -> x, y { x := div(a,b) }`, we remove the parameter and create a new «linking» function as follows:

```
function f(a,b) -> x { x := div(a,b) }
function f2(a,b,c) -> x, y { x := f(a,b) }
```

and replace all references to `f` by `f2`. The inliner should be run afterwards to make sure that all references to `f2` are replaced by `f`.

Prerequisites: Disambiguator, FunctionHoister, LiteralRematerialiser.

The step LiteralRematerialiser is not required for correctness. It helps deal with cases such as: `function f(x) -> y { revert(y, y} }` where the literal `y` will be replaced by its value `0`, allowing us to rewrite the function.

### UnusedStoreEliminator

Optimizer component that removes redundant `sstore` and memory store statements. In case of an `sstore`, if all outgoing code paths revert (due to an explicit `revert()`, `invalid()`, or infinite recursion) or lead to another `sstore` for which the optimizer can tell that it will overwrite the first store, the statement will be removed. However, if there is a read operation between the initial `sstore` and the revert, or the overwriting `sstore`, the statement will not be removed. Such read operations include: external calls, user-defined functions with any storage access, and `sload` of a slot that cannot be proven to differ from the slot written by the initial `sstore`.

For example, the following code

```
{
    let c := calldataload(0)
    sstore(c, 1)
    if c {
        sstore(c, 2)
    }
    sstore(c, 3)
}
```

will be transformed into the code below after the Unused Store Eliminator step is run

```
{
    let c := calldataload(0)
    if c { }
    sstore(c, 3)
}
```

For memory store operations, things are generally simpler, at least in the outermost yul block as all such statements will be removed if they are never read from in any code path. At function analysis level, however, the approach is similar to `sstore`, as we do not know whether the memory location will be read once we leave the function's scope, so the statement will be removed only if all code paths lead to a memory overwrite.

Best run in SSA form.

Prerequisites: Disambiguator, ForLoopInitRewriter.

### EquivalentFunctionCombiner

If two functions are syntactically equivalent, while allowing variable renaming but not any re-ordering, then any reference to one of the functions is replaced by the other.

The actual removal of the function is performed by the Unused Pruner.

## Function Inlining

### ExpressionInliner

This component of the optimizer performs restricted function inlining by inlining functions that can be inlined inside functional expressions, i.e. functions that:

- return a single value.
- have a body like `r := <functional expression>`.

- neither reference themselves nor `r` in the right hand side.

Furthermore, for all parameters, all of the following need to be true:

- The argument is movable.
- The parameter is either referenced less than twice in the function body, or the argument is rather cheap («cost» of at most 1, like a constant up to 0xff).

Example: The function to be inlined has the form of `function f(...) -> r { r := E }` where `E` is an expression that does not reference `r` and all arguments in the function call are movable expressions.

The result of this inlining is always a single expression.

This component can only be used on sources with unique names.

### FullInliner

The Full Inliner replaces certain calls of certain functions by the function's body. This is not very helpful in most cases, because it just increases the code size but does not have a benefit. Furthermore, code is usually very expensive and we would often rather have shorter code than more efficient code. In same cases, though, inlining a function can have positive effects on subsequent optimizer steps. This is the case if one of the function arguments is a constant, for example.

During inlining, a heuristic is used to tell if the function call should be inlined or not. The current heuristic does not inline into «large» functions unless the called function is tiny. Functions that are only used once are inlined, as well as medium-sized functions, while function calls with constant arguments allow slightly larger functions.

In the future, we may include a backtracking component that, instead of inlining a function right away, only specializes it, which means that a copy of the function is generated where a certain parameter is always replaced by a constant. After that, we can run the optimizer on this specialized function. If it results in heavy gains, the specialized function is kept, otherwise the original function is used instead.

### Cleanup

The cleanup is performed at the end of the optimizer run. It tries to combine split expressions into deeply nested ones again and also improves the «compilability» for stack machines by eliminating variables as much as possible.

### ExpressionJoiner

This is the opposite operation of the expression splitter. It turns a sequence of variable declarations that have exactly one reference into a complex expression. This stage fully preserves the order of function calls and opcode executions. It does not make use of any information concerning the commutativity of the opcodes; if moving the value of a variable to its place of use would change the order of any function call or opcode execution, the transformation is not performed.

Note that the component will not move the assigned value of a variable assignment or a variable that is referenced more than once.

The snippet `let x := add(0, 2) let y := mul(x, mload(2))` is not transformed, because it would cause the order of the call to the opcodes `add` and `mload` to be swapped - even though this would not make a difference because `add` is movable.

When reordering opcodes like that, variable references and literals are ignored. Because of that, the snippet `let x := add(0, 2) let y := mul(x, 3)` is transformed to `let y := mul(add(0, 2), 3)`, even though the `add` opcode would be executed after the evaluation of the literal `3`.

### SSAReverser

This is a tiny step that helps in reversing the effects of the SSA transform if it is combined with the Common Subexpression Eliminator and the Unused Pruner.

The SSA form we generate is detrimental to code generation on the EVM and WebAssembly alike because it generates many local variables. It would be better to just re-use existing variables with assignments instead of fresh variable declarations.

The SSA transform rewrites

```
let a := calldataload(0)
mstore(a, 1)
```

to

```
let a_1 := calldataload(0)
let a := a_1
mstore(a_1, 1)
let a_2 := calldataload(0x20)
a := a_2
```

The problem is that instead of `a`, the variable `a_1` is used whenever `a` was referenced. The SSA transform changes statements of this form by just swapping out the declaration and the assignment. The above snippet is turned into

```
let a := calldataload(0)
let a_1 := a
mstore(a_1, 1)
a := calldataload(0x20)
let a_2 := a
```

This is a very simple equivalence transform, but when we now run the Common Subexpression Eliminator, it will replace all occurrences of `a_1` by `a` (until `a` is re-assigned). The Unused Pruner will then eliminate the variable `a_1` altogether and thus fully reverse the SSA transform.

### StackCompressor

One problem that makes code generation for the Ethereum Virtual Machine hard is the fact that there is a hard limit of 16 slots for reaching down the expression stack. This more or less translates to a limit of 16 local variables. The stack compressor takes Yul code and compiles it to EVM bytecode. Whenever the stack difference is too large, it records the function this happened in.

For each function that caused such a problem, the Rematerialiser is called with a special request to aggressively eliminate specific variables sorted by the cost of their values.

On failure, this procedure is repeated multiple times.

### Rematerialiser

The rematerialisation stage tries to replace variable references by the expression that was last assigned to the variable. This is of course only beneficial if this expression is comparatively cheap to evaluate. Furthermore, it is only semantically equivalent if the value of the expression did not change between the point of assignment and the point of use. The main benefit of this stage is that it can save stack slots if it leads to a variable being eliminated completely (see below), but it can also save a DUP opcode on the EVM if the expression is very cheap.

The Rematerialiser uses the Dataflow Analyzer to track the current values of variables, which are always movable. If the value is very cheap or the variable was explicitly requested to be eliminated, the variable reference is replaced by its current value.

### ForLoopConditionOutOfBody

Reverses the transformation of ForLoopConditionIntoBody.

For any movable `c`, it turns

```
for { ... } 1 { ... } {
if iszero(c) { break }
...
}
```

into

```
for { ... } c { ... } {
...
}
```

and it turns

```
for { ... } 1 { ... } {
if c { break }
...
}
```

into

```
for { ... } iszero(c) { ... } {
...
}
```

The LiteralRematerialiser should be run before this step.

### WebAssembly specific

### MainFunction

Changes the topmost block to be a function with a specific name («main») which has no inputs nor outputs.

Depends on the Function Grouper.

## 3.22 Contract Metadata

The Solidity compiler automatically generates a JSON file, the contract metadata, that contains information about the compiled contract. You can use this file to query the compiler version, the sources used, the ABI and NatSpec documentation to more safely interact with the contract and verify its source code.

The compiler appends by default the IPFS hash of the metadata file to the end of the bytecode (for details, see below) of each contract, so that you can retrieve the file in an authenticated way without having to resort to a centralized data provider. The other available options are the Swarm hash and not appending the metadata hash to the bytecode. These can be configured via the *Standard JSON Interface*.

You have to publish the metadata file to IPFS, Swarm, or another service so that others can access it. You create the file by using the `solc --metadata` command together with the `--output-dir` parameter. Without the parameter, the metadata will be written to standard output. The metadata contains IPFS and Swarm references to the source code, so you have to upload all source files in addition to the metadata file. For IPFS, the hash contained in the CID returned by `ipfs add` (not the direct sha2-256 hash of the file) shall match with the one contained in the bytecode.

The metadata file has the following format. The example below is presented in a human-readable way. Properly formatted metadata should use quotes correctly, reduce whitespace to a minimum and sort the keys of all objects to arrive at a unique formatting. Comments are not permitted and used here only for explanatory purposes.

```
{
  // Required: The version of the metadata format
  "version": "1",
  // Required: Source code language, basically selects a "sub-version"
  // of the specification
  "language": "Solidity",
  // Required: Details about the compiler, contents are specific
  // to the language.
  "compiler": {
    // Required for Solidity: Version of the compiler
    "version": "0.8.2+commit.661d1103",
    // Optional: Hash of the compiler binary which produced this output
    "keccak256": "0x123..."
  },
  // Required: Compilation source files/source units, keys are file paths
  "sources":
  {
    "myDirectory/myFile.sol": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): Sorted URL(s)
      // to the source file, protocol is more or less arbitrary, but an
      // IPFS URL is recommended
      "urls": [ "bzz-raw://7d7a...", "dweb:/ipfs/QmN..." ],
      // Optional: SPDX license identifier as given in the source file
      "license": "MIT"
    },
    "destructible": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "url" is used): literal contents of the source file
      "content": "contract destructible is owned { function destroy() { if (msg.sender
↪== owner) selfdestruct(owner); } }"
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }
  },
  // Required: Compiler settings
  "settings":
  {
    // Required for Solidity: Sorted list of import remappings
    "remappings": [ ":g=/dir" ],
    // Optional: Optimizer settings. The fields "enabled" and "runs" are deprecated
    // and are only given for backwards-compatibility.
    "optimizer": {
      "enabled": true,
      "runs": 500,
      "details": {
        // peephole defaults to "true"
        "peephole": true,
        // inliner defaults to "true"
        "inliner": true,
        // jumpdestRemover defaults to "true"
        "jumpdestRemover": true,
        "orderLiterals": false,
        "deduplicate": false,
        "cse": false,
        "constantOptimizer": false,
        "yul": true,
        // Optional: Only present if "yul" is "true"
        "yulDetails": {
          "stackAllocation": false,
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    "metadata": {
      // Reflects the setting used in the input json, defaults to "true"
      "appendCBOR": true,
      // Reflects the setting used in the input json, defaults to "false"
      "useLiteralContent": true,
      // Reflects the setting used in the input json, defaults to "ipfs"
      "bytecodeHash": "ipfs"
    },
    // Required for Solidity: File path and the name of the contract or library this
    // metadata is created for.
    "compilationTarget": {
      "myDirectory/myFile.sol": "MyContract"
    },
    // Required for Solidity: Addresses for libraries used
    "libraries": {
      "MyLib": "0x123123..."
    }
  },
  // Required: Generated information about the contract.
  "output":
  {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // Required: ABI definition of the contract. See "Contract ABI Specification"
    "abi": [/* ... */],
    // Required: NatSpec developer documentation of the contract.
    "devdoc": {
      "version": 1 // NatSpec version
      "kind": "dev",
      // Contents of the @author NatSpec field of the contract
      "author": "John Doe",
      // Contents of the @title NatSpec field of the contract
      "title": "MyERC20: an example ERC20"
      // Contents of the @dev NatSpec field of the contract
      "details": "Interface of the ERC20 standard as defined in the EIP. See https://
↪eips.ethereum.org/EIPS/eip-20 for details",
      "methods": {
        "transfer(address,uint256)": {
          // Contents of the @dev NatSpec field of the method
          "details": "Returns a boolean value indicating whether the operation succeeded.
↪ Must be called by the token holder address",
          // Contents of the @param NatSpec fields of the method
          "params": {
            "_value": "The amount tokens to be transferred",
            "_to": "The receiver address"
          }
          // Contents of the @return NatSpec field.
          "returns": {
            // Return var name (here "success") if exists. "_0" as key if return var is␣
↪unnamed
            "success": "a boolean value indicating whether the operation succeeded"
          }
        }
      },
      "stateVariables": {
        "owner": {
          // Contents of the @dev NatSpec field of the state variable
          "details": "Must be set during contract creation. Can then only be changed by␣
↪the owner"
        }
      }
      "events": {
         "Transfer(address,address,uint256)": {
           "details": "Emitted when `value` tokens are moved from one account (`from`)␣
↪toanother (`to`)."
           "params": {
             "from": "The sender address"
             "to": "The receiver address"
             "value": "The token amount"
           }
         }
      }
    },
    // Required: NatSpec user documentation of the contract
    "userdoc": {
```

(continué en la próxima página)

(proviene de la página anterior)

```
      "version": 1 // NatSpec version
      "kind": "user",
      "methods": {
        "transfer(address,uint256)": {
          "notice": "Transfers `_value` tokens to address `_to`"
        }
      },
      "events": {
        "Transfer(address,address,uint256)": {
          "notice": "`_value` tokens have been moved from `from` to `to`"
        }
      }
    }
  }
}
```

**Advertencia:** Since the bytecode of the resulting contract contains the metadata hash by default, any change to the metadata might result in a change of the bytecode. This includes changes to a filename or path, and since the metadata includes a hash of all the sources used, a single whitespace change results in different metadata, and different bytecode.

**Nota:** The ABI definition above has no fixed order. It can change with compiler versions. Starting from Solidity version 0.5.12, though, the array maintains a certain order.

### 3.22.1 Encoding of the Metadata Hash in the Bytecode

Because we might support other ways to retrieve the metadata file in the future, the mapping `{"ipfs": <IPFS hash>, "solc": <compiler version>}` is stored CBOR-encoded. Since the mapping might contain more keys (see below) and the beginning of that encoding is not easy to find, its length is added in a two-byte big-endian encoding. The current version of the Solidity compiler usually adds the following to the end of the deployed bytecode

```
0xa2
0x64 'i' 'p' 'f' 's' 0x58 0x22 <34 bytes IPFS hash>
0x64 's' 'o' 'l' 'c' 0x43 <3 byte version encoding>
0x00 0x33
```

So in order to retrieve the data, the end of the deployed bytecode can be checked to match that pattern and the IPFS hash can be used to retrieve the file (if pinned/published).

Whereas release builds of solc use a 3 byte encoding of the version as shown above (one byte each for major, minor and patch version number), prerelease builds will instead use a complete version string including commit hash and build date.

The commandline flag `--no-cbor-metadata` can be used to skip metadata from getting appended at the end of the deployed bytecode. Equivalently, the boolean field `settings.metadata.appendCBOR` in Standard JSON input can be set to false.

**Nota:** The CBOR mapping can also contain other keys, so it is better to fully decode the data instead of relying on it starting with `0xa264`. For example, if any experimental features that affect code generation are used, the mapping will

also contain `"experimental": true`.

**Nota:** The compiler currently uses the IPFS hash of the metadata by default, but it may also use the bzzr1 hash or some other hash in the future, so do not rely on this sequence to start with `0xa2 0x64 'i' 'p' 'f' 's'`. We might also add additional data to this CBOR structure, so the best option is to use a proper CBOR parser.

### 3.22.2 Usage for Automatic Interface Generation and NatSpec

The metadata is used in the following way: A component that wants to interact with a contract (e.g. a wallet) retrieves the code of the contract. It decodes the CBOR encoded section containing the IPFS/Swarm hash of the metadata file. With that hash, the metadata file is retrieved. That file is JSON-decoded into a structure like above.

The component can then use the ABI to automatically generate a rudimentary user interface for the contract.

Furthermore, the wallet can use the NatSpec user documentation to display a human-readable confirmation message to the user whenever they interact with the contract, together with requesting authorization for the transaction signature.

For additional information, read *Ethereum Natural Language Specification (NatSpec) format*.

### 3.22.3 Usage for Source Code Verification

In order to verify the compilation, sources can be retrieved from IPFS/Swarm via the link in the metadata file. The compiler of the correct version (which is checked to be part of the «official» compilers) is invoked on that input with the specified settings. The resulting bytecode is compared to the data of the creation transaction or `CREATE` opcode data. This automatically verifies the metadata since its hash is part of the bytecode. Excess data corresponds to the constructor input data, which should be decoded according to the interface and presented to the user.

In the repository sourcify (npm package) you can see example code that shows how to use this feature.

## 3.23 Contract ABI Specification

### 3.23.1 Basic Design

The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data is encoded according to its type, as described in this specification. The encoding is not self describing and thus requires a schema in order to decode.

We assume that the interface functions of a contract are strongly typed, known at compilation time and static. We assume that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time.

### 3.23.2 Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma — no spaces are used.

---

**Nota:** The return type of a function is not part of this signature. In *Solidity's function overloading* return types are not considered. The reason is to keep function call resolution context-independent. The *JSON description of the ABI* however contains both inputs and outputs.

---

### 3.23.3 Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

### 3.23.4 Types

The following elementary types exist:

- `uint<M>`: unsigned integer type of `M` bits, `0 < M <= 256`, `M% 8 == 0`. e.g. `uint32`, `uint8`, `uint256`.
- `int<M>`: two's complement signed integer type of `M` bits, `0 < M <= 256`, `M% 8 == 0`.
- `address`: equivalent to `uint160`, except for the assumed interpretation and language typing. For computing the function selector, `address` is used.
- `uint`, `int`: synonyms for `uint256`, `int256` respectively. For computing the function selector, `uint256` and `int256` have to be used.
- `bool`: equivalent to `uint8` restricted to the values 0 and 1. For computing the function selector, `bool` is used.
- `fixed<M>x<N>`: signed fixed-point decimal number of `M` bits, `8 <= M <= 256`, `M% 8 == 0`, and `0 < N <= 80`, which denotes the value `v` as `v / (10 ** N)`.
- `ufixed<M>x<N>`: unsigned variant of `fixed<M>x<N>`.
- `fixed`, `ufixed`: synonyms for `fixed128x18`, `ufixed128x18` respectively. For computing the function selector, `fixed128x18` and `ufixed128x18` have to be used.
- `bytes<M>`: binary type of `M` bytes, `0 < M <= 32`.
- `function`: an address (20 bytes) followed by a function selector (4 bytes). Encoded identical to `bytes24`.

The following (fixed-size) array type exists:

- `<type>[M]`: a fixed-length array of `M` elements, `M >= 0`, of the given type.

  ---

  **Nota:** While this ABI specification can express fixed-length arrays with zero elements, they're not supported by the compiler.

  ---

The following non-fixed-size types exist:

- `bytes`: dynamic sized byte sequence.
- `string`: dynamic sized unicode string assumed to be UTF-8 encoded.

- `<type>[]`: a variable-length array of elements of the given type.

Types can be combined to a tuple by enclosing them inside parentheses, separated by commas:

- `(T1,T2,...,Tn)`: tuple consisting of the types `T1`, ..., `Tn`, `n >= 0`

It is possible to form tuples of tuples, arrays of tuples and so on. It is also possible to form zero-tuples (where `n == 0`).

#### Mapping Solidity to ABI types

Solidity supports all the types presented above with the same names with the exception of tuples. On the other hand, some Solidity types are not supported by the ABI. The following table shows on the left column Solidity types that are not part of the ABI, and on the right column the ABI types that represent them.

| Solidity | ABI |
|---|---|
| *address payable* | `address` |
| *contract* | `address` |
| *enum* | `uint8` |
| *user defined value types* | its underlying value type |
| *struct* | `tuple` |

**Advertencia:** Before version `0.8.0` enums could have more than 256 members and were represented by the smallest integer type just big enough to hold the value of any member.

### 3.23.5 Design Criteria for the Encoding

The encoding is designed to have the following properties, which are especially useful if some arguments are nested arrays:

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]`. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
2. The data of a variable or an array element is not interleaved with other data and it is relocatable, i.e. it only uses relative «addresses».

### 3.23.6 Formal Specification of the Encoding

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

**Definition:** The following types are called «dynamic»:

- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k >= 0`
- `(T1,...,Tk)` if `Ti` is dynamic for some `1 <= i <= k`

All other types are called «static».

**Definition:** `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

**Definition:** For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being

- `(T1,...,Tk)` for `k >= 0` and any types `T1`, …, `Tk`

  `enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`

  where `X = (X(1), ..., X(k))` and `head` and `tail` are defined for `Ti` as follows:

  if `Ti` is static:

  `head(X(i)) = enc(X(i))` and `tail(X(i)) = ""` (the empty string)

  otherwise, i.e. if `Ti` is dynamic:

  `head(X(i)) = enc(len( head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1)) )) tail(X(i)) = enc(X(i))`

  Note that in the dynamic case, `head(X(i))` is well-defined since the lengths of the head parts only depend on the types and not the values. The value of `head(X(i))` is the offset of the beginning of `tail(X(i))` relative to the start of `enc(X)`.

- `T[k]` for any `T` and `k`:

  `enc(X) = enc((X[0], ..., X[k-1]))`

  i.e. it is encoded as if it were a tuple with `k` elements of the same type.

- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):

  `enc(X) = enc(k) enc((X[0], ..., X[k-1]))`

  i.e. it is encoded as if it were a tuple with `k` elements of the same type (resp. an array of static size `k`), prefixed with the number of elements.

- `bytes`, of length `k` (which is assumed to be of type `uint256`):

  `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.

- `string`:

  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is UTF-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the UTF-8 encoded string, not its number of characters.

- `uint<M>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is 32 bytes.

- `address`: as in the `uint160` case

- `int<M>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-order (left) side with `0xff` bytes for negative `X` and with zero-bytes for non-negative `X` such that the length is 32 bytes.

- `bool`: as in the `uint8` case, where 1 is used for `true` and `0` for `false`

- `fixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `int256`.

- `fixed`: as in the `fixed128x18` case

- `ufixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `uint256`.

- `ufixed`: as in the `ufixed128x18` case
- `bytes<M>`: `enc(X)` is the sequence of bytes in `X` padded with trailing zero-bytes to a length of 32 bytes.

Note that for any `X`, `len(enc(X))` is a multiple of 32.

### 3.23.7 Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

```
function_selector(f) enc((a_1, ..., a_n))
```

and the return values `v_1, ..., v_k` of `f` are encoded as

```
enc((v_1, ..., v_k))
```

i.e. the values are combined into a tuple and encoded.

### 3.23.8 Examples

Given the contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

Thus, for our `Foo` example if we wanted to call `baz` with the parameters `69` and `true`, we would pass 68 bytes total, which can be broken down into:

- `0xcdcd77c0`: the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature `baz(uint32,bool)`.
- `0x0000000000000000000000000000000000000000000000000000000000000045`: the first parameter, a uint32 value `69` padded to 32 bytes
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the second parameter - boolean `true`, padded to 32 bytes

In total:

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000
```

It returns a single `bool`. If, for example, it were to return `false`, its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000`, a single bool.

If we wanted to call `bar` with the argument `["abc", "def"]`, we would pass 68 bytes total, broken down into:

- `0xfce353f6`: the Method ID. This is derived from the signature `bar(bytes3[2])`.
- `0x6162630000000000000000000000000000000000000000000000000000000000`: the first part of the first parameter, a `bytes3` value `"abc"` (left-aligned).
- `0x6465660000000000000000000000000000000000000000000000000000000000`: the second part of the first parameter, a `bytes3` value `"def"` (left-aligned).

In total:

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000000000000000000000000000000000006465660000000000000000000000000
```

If we wanted to call `sam` with the arguments `"dave"`, `true` and `[1,2,3]`, we would pass 292 bytes total, broken down into:

- `0xa5643bf2`: the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])`. Note that `uint` is replaced with its canonical representation `uint256`.
- `0x0000000000000000000000000000000000000000000000000000000000000060`: the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60`.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the second parameter: boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0`: the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xa0`.
- `0x0000000000000000000000000000000000000000000000000000000000000004`: the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.
- `0x6461766500000000000000000000000000000000000000000000000000000000`: the contents of the first argument: the UTF-8 (equal to ASCII in this case) encoding of `"dave"`, padded on the right to 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the first entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000002`: the second entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the third entry of the third parameter.

In total:

```
0xa5643bf2000000000000000000000000000000000000000000000000000000000000006000000000000000000000000000000000000000000000000000000000000000a0000000000000000000000000000000000000000000000000000000000000046461766500000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000300000000000000000000000000000000000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000000000000000300000000000000000000000000000000000000000000000000000000000000
```

### 3.23.9 Use of Dynamic Types

A call to a function with the signature `f(uint256,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way:

We take the first four bytes of `keccak("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)

- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32 + 3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"`:

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case): 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (newline after function selector and each 32-bytes for clarity):

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Let us apply the same principle to encode the data for a function with a signature `g(uint256[][],string[])` with values `([[1, 2], [3]], ["one", "two", "three"])` but start from the most atomic parts of the encoding:

First we encode the length and data of the first embedded dynamic array `[1, 2]` of the first root array `[[1, 2], [3]]`:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first array, 2; the elements themselves are 1 and 2)
- `0x0000000000000000000000000000000000000000000000000000000000000001` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000002` (second element)

Then we encode the length and data of the second embedded dynamic array `[3]` of the first root array `[[1, 2], [3]]`:

- `0x0000000000000000000000000000000000000000000000000000000000000001` (number of elements in the second array, 1; the element is 3)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (first element)

Then we need to find the offsets `a` and `b` for their respective dynamic arrays `[1, 2]` and `[3]`. To calculate the offsets we can take a look at the encoded data of the first root array `[[1, 2], [3]]` enumerating each line in the encoding:

```
0 - a                                                               - offset of [1, 2]
1 - b                                                               - offset of [3]
2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1, 2]
3 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
4 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
```

(continué en la próxima página)

(proviene de la página anterior)

```
5 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
6 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
```

Offset `a` points to the start of the content of the array `[1, 2]` which is line 2 (64 bytes); thus `a = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `b` points to the start of the content of the array `[3]` which is line 5 (160 bytes); thus `b = 0x00000000000000000000000000000000000000000000000000000000000000a0`.

Then we encode the embedded strings of the second root array:

- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of characters in word `"one"`)
- `0x6f6e650000000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"one"`)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of characters in word `"two"`)
- `0x74776f0000000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"two"`)
- `0x0000000000000000000000000000000000000000000000000000000000000005` (number of characters in word `"three"`)
- `0x7468726565000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"three"`)

In parallel to the first root array, since strings are dynamic elements we need to find their offsets `c`, `d` and `e`:

```
0 - c                                                                - offset for "one"
1 - d                                                                - offset for "two"
2 - e                                                                - offset for "three"
3 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of "one"
5 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of "two"
7 - 0000000000000000000000000000000000000000000000000000000000000005 - count for "three"
8 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of "three
↪"
```

Offset `c` points to the start of the content of the string `"one"` which is line 3 (96 bytes); thus `c = 0x0000000000000000000000000000000000000000000000000000000000000060`.

Offset `d` points to the start of the content of the string `"two"` which is line 5 (160 bytes); thus `d = 0x00000000000000000000000000000000000000000000000000000000000000a0`.

Offset `e` points to the start of the content of the string `"three"` which is line 7 (224 bytes); thus `e = 0x00000000000000000000000000000000000000000000000000000000000000e0`.

Note that the encodings of the embedded elements of the root arrays are not dependent on each other and have the same encodings for a function with a signature `g(string[],uint256[][])`.

Then we encode the length of the first root array:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first root array, 2; the elements themselves are `[1, 2]` and `[3]`)

Then we encode the length of the second root array:

- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of strings in the second root array, 3; the strings themselves are `"one"`, `"two"` and `"three"`)

Finally we find the offsets `f` and `g` for their respective root dynamic arrays `[[1, 2], [3]]` and `["one", "two", "three"]`, and assemble parts in the correct order:

```
0x2289b18c                                                         - function␣
↪signature
 0 - f                                                             - offset of [[1,␣
↪2], [3]]
 1 - g                                                             - offset of ["one",
↪ "two", "three"]
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [[1,␣
↪2], [3]]
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - offset of [1, 2]
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset of [3]
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1, 2]
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
10 - 0000000000000000000000000000000000000000000000000000000000000003 - count for ["one",
↪ "two", "three"]
11 - 0000000000000000000000000000000000000000000000000000000000000060 - offset for "one"
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset for "two"
13 - 00000000000000000000000000000000000000000000000000000000000000e0 - offset for "three
↪"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of "one"
16 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of "two"
18 - 0000000000000000000000000000000000000000000000000000000000000005 - count for "three"
19 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of
↪"three"
```

Offset `f` points to the start of the content of the array `[[1, 2], [3]]` which is line 2 (64 bytes); thus `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `g` points to the start of the content of the array `["one", "two", "three"]` which is line 10 (320 bytes); thus `g = 0x0000000000000000000000000000000000000000000000000000000000000140`.

### 3.23.10 Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3 (for non-anonymous events) or 4 (for anonymous ones), are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as:

- `address`: the address of the contract (intrinsically provided by Ethereum);

- `topics[0]`: `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo`, it would return `uint256`). This value is only present in `topics[0]` if the event is not declared as `anonymous`;
- `topics[n]`: `abi_encode(EVENT_INDEXED_ARGS[n - 1])` if the event is not declared as `anonymous` or `abi_encode(EVENT_INDEXED_ARGS[n])` if it is (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
- `data`: ABI encoding of `EVENT_NON_INDEXED_ARGS` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_encode` is the ABI encoding function used for returning a series of typed values from a function, as described above).

For all types of length at most 32 bytes, the `EVENT_INDEXED_ARGS` array contains the value directly, padded or sign-extended (for signed integers) to 32 bytes, just as for regular ABI encoding. However, for all «complex» types or types of dynamic length, including all arrays, `string`, `bytes` and structs, `EVENT_INDEXED_ARGS` will contain the *Keccak hash* of a special in-place encoded value (see *Encoding of Indexed Event Parameters*), rather than the encoded value directly. This allows applications to efficiently query for values of dynamic-length types (by setting the hash of the encoded value as the topic), but leaves applications unable to decode indexed values they have not queried for. For dynamic-length types, application developers face a trade-off between fast search for predetermined values (if the argument is indexed) and legibility of arbitrary values (which requires that the arguments not be indexed). Developers may overcome this tradeoff and achieve both efficient search and arbitrary legibility by defining events with two arguments — one indexed, one not — intended to hold the same value.

### 3.23.11 Errors

In case of a failure inside a contract, the contract can use a special opcode to abort execution and revert all state changes. In addition to these effects, descriptive data can be returned to the caller. This descriptive data is the encoding of an error and its arguments in the same way as data for a function call.

As an example, let us consider the following contract whose `transfer` function always reverts with a custom error of «insufficient balance»:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract TestToken {
    error InsufficientBalance(uint256 available, uint256 required);
    function transfer(address /*to*/, uint amount) public pure {
        revert InsufficientBalance(0, amount);
    }
}
```

The return data would be encoded in the same way as the function call `InsufficientBalance(0, amount)` to the function `InsufficientBalance(uint256,uint256)`, i.e. `0xcf479181`, `uint256(0)`, `uint256(amount)`.

The error selectors `0x00000000` and `0xffffffff` are reserved for future use.

**Advertencia:** Never trust error data. The error data by default bubbles up through the chain of external calls, which means that a contract may receive an error not defined in any of the contracts it calls directly. Furthermore, any contract can fake any error by returning data that matches an error signature, even if the error is not defined anywhere.

### 3.23.12 JSON

The JSON format for a contract's interface is given by an array of function, event and error descriptions. A function description is a JSON object with the fields:

- `type`: `"function"`, `"constructor"`, `"receive"` (the *«receive Ether» function*) or `"fallback"` (the *«default» function*);
- `name`: the name of the function;
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
- `outputs`: an array of objects similar to `inputs`.
- `stateMutability`: a string with one of the following values: `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` (function does not accept Ether - the default) and `payable` (function accepts Ether).

Constructor, receive, and fallback never have `name` or `outputs`. Receive and fallback don't have `inputs` either.

**Nota:** Sending non-zero Ether to non-payable function will revert the transaction.

**Nota:** The state mutability `nonpayable` is reflected in Solidity by not specifying a state mutability modifier at all.

An event description is a JSON object with fairly similar fields:

- `type`: always `"event"`
- `name`: the name of the event.
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
  - `indexed`: `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous`: `true` if the event was declared as `anonymous`.

Errors look as follows:

- `type`: always `"error"`
- `name`: the name of the error.
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).

**Nota:** There can be multiple errors with the same name and even with identical signature in the JSON array; for example, if the errors originate from different files in the smart contract or are referenced from another smart contract. For the ABI, only the name of the error itself is relevant and not where it is defined.

For example,

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;


contract Test {
    constructor() { b = hex"12345678901234567890123456789012"; }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    error InsufficientBalance(uint256 available, uint256 required);
    function foo(uint a) public { emit Event(a, b); }
    bytes32 b;
}
```

would result in the JSON:

```
[{
"type":"error",
"inputs": [{"name":"available","type":"uint256"},{"name":"required","type":"uint256"}],
"name":"InsufficientBalance"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

### Handling tuple types

Despite the fact that names are intentionally not part of the ABI encoding, they do make a lot of sense to be included in the JSON to enable displaying it to the end user. The structure is nested in the following way:

An object with members `name`, `type` and potentially `components` describes a typed variable. The canonical type is determined until a tuple type is reached and the string description up to that point is stored in `type` prefix with the word `tuple`, i.e. it will be `tuple` followed by a sequence of `[]` and `[k]` with integers `k`. The components of the tuple are then stored in the member `components`, which is of an array type and has the same structure as the top-level object except that `indexed` is not allowed there.

As an example, the code

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.5 <0.9.0;
pragma abicoder v2;

contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory, T memory, uint) public pure {}
    function g() public pure returns (S memory, T memory, uint) {}
}
```

would result in the JSON:

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
              }
```

(continué en la próxima página)

(proviene de la página anterior)

```
            ]
          }
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

### 3.23.13 Strict Encoding Mode

Strict encoding mode is the mode that leads to exactly the same encoding as defined in the formal specification above. This means that offsets have to be as small as possible while still not creating overlaps in the data areas, and thus no gaps are allowed.

Usually, ABI decoders are written in a straightforward way by just following offset pointers, but some decoders might enforce strict mode. The Solidity ABI decoder currently does not enforce strict mode, but the encoder always creates data in strict mode.

### 3.23.14 Non-standard Packed Mode

Through `abi.encodePacked()`, Solidity supports a non-standard packed mode where:

- types shorter than 32 bytes are concatenated directly, without padding or sign extension
- dynamic types are encoded in-place and without the length.
- array elements are padded, but still encoded in-place

Furthermore, structs as well as nested arrays are not supported.

As an example, the encoding of `int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world!")` results in:

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                                int16(-1)
      ^^                              bytes1(0x42)
        ^^^^                          uint16(0x03)
            ^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") without a length field
```

More specifically:

- During the encoding, everything is encoded in-place. This means that there is no distinction between head and tail, as in the ABI encoding, and the length of an array is not encoded.
- The direct arguments of `abi.encodePacked` are encoded without padding, as long as they are not arrays (or `string` or `bytes`).
- The encoding of an array is the concatenation of the encoding of its elements **with** padding.
- Dynamically-sized types like `string`, `bytes` or `uint[]` are encoded without their length field.
- The encoding of `string` or `bytes` does not apply padding at the end, unless it is part of an array or struct (then it is padded to a multiple of 32 bytes).

In general, the encoding is ambiguous as soon as there are two dynamically-sized elements, because of the missing length field.

If padding is needed, explicit type conversions can be used: `abi.encodePacked(uint16(0x12)) == hex"0012"`.

Since packed encoding is not used when calling functions, there is no special support for prepending a function selector. Since the encoding is ambiguous, there is no decoding function.

**Advertencia:** If you use `keccak256(abi.encodePacked(a, b))` and both `a` and `b` are dynamic types, it is easy to craft collisions in the hash value by moving parts of `a` into `b` and vice-versa. More specifically, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. If you use `abi.encodePacked` for signatures, authentication or data integrity, make sure to always use the same types and check that at most one of them is dynamic. Unless there is a compelling reason, `abi.encode` should be preferred.

### 3.23.15 Encoding of Indexed Event Parameters

Indexed event parameters that are not value types, i.e. arrays and structs are not stored directly but instead a Keccak-256 hash of an encoding is stored. This encoding is defined as follows:

- the encoding of a `bytes` and `string` value is just the string contents without any padding or length prefix.
- the encoding of a struct is the concatenation of the encoding of its members, always padded to a multiple of 32 bytes (even `bytes` and `string`).
- the encoding of an array (both dynamically- and statically-sized) is the concatenation of the encoding of its elements, always padded to a multiple of 32 bytes (even `bytes` and `string`) and without any length prefix

In the above, as usual, a negative number is padded by sign extension and not zero padded. `bytesNN` types are padded on the right while `uintNN` / `intNN` are padded on the left.

**Advertencia:** The encoding of a struct is ambiguous if it contains more than one dynamically-sized array. Because of that, always re-check the event data and do not rely on the search result based on the indexed parameters alone.

## 3.24 Solidity v0.5.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.5.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

**Nota:** Contracts compiled with Solidity v0.5.0 can still interface with contracts and even libraries compiled with older versions without recompiling or redeploying them. Changing the interfaces to include data locations and visibility and mutability specifiers suffices. See the *Interoperability With Older Contracts* section below.

### 3.24.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- Signed right shift now uses proper arithmetic shift, i.e. rounding towards negative infinity, instead of rounding towards zero. Signed and unsigned shift will have dedicated opcodes in Constantinople, and are emulated by Solidity for the moment.
- The `continue` statement in a `do...while` loop now jumps to the condition, which is the common behavior in such cases. It used to jump to the loop body. Thus, if the condition is false, the loop terminates.
- The functions `.call()`, `.delegatecall()` and `.staticcall()` do not pad anymore when given a single `bytes` parameter.
- Pure and view functions are now called using the opcode `STATICCALL` instead of `CALL` if the EVM version is Byzantium or later. This disallows state changes on the EVM level.
- The ABI encoder now properly pads byte arrays and strings from calldata (`msg.data` and external function parameters) when used in external function calls and in `abi.encode`. For unpadded encoding, use `abi.encodePacked`.
- The ABI decoder reverts in the beginning of functions and in `abi.decode()` if passed calldata is too short or points out of bounds. Note that dirty higher order bits are still simply ignored.
- Forward all available gas with external function calls starting from Tangerine Whistle.

### 3.24.2 Semantic and Syntactic Changes

This section highlights changes that affect syntax and semantics.

- The functions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` and `ripemd160()` now accept only a single `bytes` argument. Moreover, the argument is not padded. This was changed to make more explicit and clear how the arguments are concatenated. Change every `.call()` (and family) to a `.call("")` and every `.call(signature, a, b, c)` to use `.call(abi.encodeWithSignature(signature, a, b, c))` (the last one only works for value types). Change every `keccak256(a, b, c)` to `keccak256(abi.encodePacked(a, b, c))`. Even though it is not a breaking change, it is suggested that developers change `x.call(bytes4(keccak256("f(uint256)")), a, b)` to `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.
- Functions `.call()`, `.delegatecall()` and `.staticcall()` now return `(bool, bytes memory)` to provide access to the return data. Change `bool success = otherContract.call("f")` to `(bool success, bytes memory data) = otherContract.call("f")`.
- Solidity now implements C99-style scoping rules for function local variables, that is, variables can only be used after they have been declared and only in the same or nested scopes. Variables declared in the initialization block of a `for` loop are valid at any point inside the loop.

### 3.24.3 Explicitness Requirements

This section lists changes where the code now needs to be more explicit. For most of the topics the compiler will provide suggestions.

- Explicit function visibility is now mandatory. Add `public` to every function and constructor, and `external` to every fallback or interface function that does not specify its visibility already.
- Explicit data location for all variables of struct, array or mapping types is now mandatory. This is also applied to function parameters and return variables. For example, change `uint[] x = z` to `uint[] storage x = z`, and `function f(uint[][] x)` to `function f(uint[][] memory x)` where `memory` is the data location and might be replaced by `storage` or `calldata` accordingly. Note that `external` functions require parameters with a data location of `calldata`.
- Contract types do not include `address` members anymore in order to separate the namespaces. Therefore, it is now necessary to explicitly convert values of contract type to addresses before using an `address` member. Example: if `c` is a contract, change `c.transfer(...)` to `address(c).transfer(...)`, and `c.balance` to `address(c).balance`.
- Explicit conversions between unrelated contract types are now disallowed. You can only convert from a contract type to one of its base or ancestor types. If you are sure that a contract is compatible with the contract type you want to convert to, although it does not inherit from it, you can work around this by converting to `address` first. Example: if `A` and `B` are contract types, `B` does not inherit from `A` and `b` is a contract of type `B`, you can still convert `b` to type `A` using `A(address(b))`. Note that you still need to watch out for matching payable fallback functions, as explained below.
- The `address` type was split into `address` and `address payable`, where only `address payable` provides the `transfer` function. An `address payable` can be directly converted to an `address`, but the other way around is not allowed. Converting `address` to `address payable` is possible via conversion through `uint160`. If `c` is a contract, `address(c)` results in `address payable` only if `c` has a payable fallback function. If you use the *withdraw pattern*, you most likely do not have to change your code because `transfer` is only used on `msg.sender` instead of stored addresses and `msg.sender` is an `address payable`.
- Conversions between `bytesX` and `uintY` of different size are now disallowed due to `bytesX` padding on the right and `uintY` padding on the left which may cause unexpected conversion results. The size must now be adjusted within the type before the conversion. For example, you can convert a `bytes4` (4 bytes) to a `uint64` (8 bytes) by first converting the `bytes4` variable to `bytes8` and then to `uint64`. You get the opposite padding when converting through `uint32`. Before v0.5.0 any conversion between `bytesX` and `uintY` would go through `uint8X`. For example `uint8(bytes3(0x291807))` would be converted to `uint8(uint24(bytes3(0x291807)))` (the result is `0x07`).
- Using `msg.value` in non-payable functions (or introducing it via a modifier) is disallowed as a security feature. Turn the function into `payable` or create a new internal function for the program logic that uses `msg.value`.
- For clarity reasons, the command line interface now requires `-` if the standard input is used as source.

### 3.24.4 Deprecated Elements

This section lists changes that deprecate prior features or syntax. Note that many of these changes were already enabled in the experimental mode `v0.5.0`.

### Command Line and JSON Interfaces

- The command line option `--formal` (used to generate Why3 output for further formal verification) was deprecated and is now removed. A new formal verification module, the SMTChecker, is enabled via `pragma experimental SMTChecker;`.
- The command line option `--julia` was renamed to `--yul` due to the renaming of the intermediate language `Julia` to `Yul`.
- The `--clone-bin` and `--combined-json clone-bin` command line options were removed.
- Remappings with empty prefix are disallowed.
- The JSON AST fields `constant` and `payable` were removed. The information is now present in the `stateMutability` field.
- The JSON AST field `isConstructor` of the `FunctionDefinition` node was replaced by a field called `kind` which can have the value `"constructor"`, `"fallback"` or `"function"`.
- In unlinked binary hex files, library address placeholders are now the first 36 hex characters of the keccak256 hash of the fully qualified library name, surrounded by `$...$`. Previously, just the fully qualified library name was used. This reduces the chances of collisions, especially when long paths are used. Binary files now also contain a list of mappings from these placeholders to the fully qualified names.

### Constructors

- Constructors must now be defined using the `constructor` keyword.
- Calling base constructors without parentheses is now disallowed.
- Specifying base constructor arguments multiple times in the same inheritance hierarchy is now disallowed.
- Calling a constructor with arguments but with wrong argument count is now disallowed. If you only want to specify an inheritance relation without giving arguments, do not provide parentheses at all.

### Functions

- Function `callcode` is now disallowed (in favor of `delegatecall`). It is still possible to use it via inline assembly.
- `suicide` is now disallowed (in favor of `selfdestruct`).
- `sha3` is now disallowed (in favor of `keccak256`).
- `throw` is now disallowed (in favor of `revert`, `require` and `assert`).

### Conversions

- Explicit and implicit conversions from decimal literals to `bytesXX` types is now disallowed.
- Explicit and implicit conversions from hex literals to `bytesXX` types of different size is now disallowed.

### Literals and Suffixes

- The unit denomination `years` is now disallowed due to complications and confusions about leap years.
- Trailing dots that are not followed by a number are now disallowed.
- Combining hex numbers with unit denominations (e.g. `0x1e wei`) is now disallowed.
- The prefix `0X` for hex numbers is disallowed, only `0x` is possible.

### Variables

- Declaring empty structs is now disallowed for clarity.
- The `var` keyword is now disallowed to favor explicitness.
- Assignments between tuples with different number of components is now disallowed.
- Values for constants that are not compile-time constants are disallowed.
- Multi-variable declarations with mismatching number of values are now disallowed.
- Uninitialized storage variables are now disallowed.
- Empty tuple components are now disallowed.
- Detecting cyclic dependencies in variables and structs is limited in recursion to 256.
- Fixed-size arrays with a length of zero are now disallowed.

### Syntax

- Using `constant` as function state mutability modifier is now disallowed.
- Boolean expressions cannot use arithmetic operations.
- The unary + operator is now disallowed.
- Literals cannot anymore be used with `abi.encodePacked` without prior conversion to an explicit type.
- Empty return statements for functions with one or more return values are now disallowed.
- The «loose assembly» syntax is now disallowed entirely, that is, jump labels, jumps and non-functional instructions cannot be used anymore. Use the new `while`, `switch` and `if` constructs instead.
- Functions without implementation cannot use modifiers anymore.
- Function types with named return values are now disallowed.
- Single statement variable declarations inside if/while/for bodies that are not blocks are now disallowed.
- New keywords: `calldata` and `constructor`.
- New reserved keywords: `alias`, `apply`, `auto`, `copyof`, `define`, `immutable`, `implements`, `macro`, `mutable`, `override`, `partial`, `promise`, `reference`, `sealed`, `sizeof`, `supports`, `typedef` and `unchecked`.

### 3.24.5 Interoperability With Older Contracts

It is still possible to interface with contracts written for Solidity versions prior to v0.5.0 (or the other way around) by defining interfaces for them. Consider you have the following pre-0.5.0 contract already deployed:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will report a warning until version 0.4.25 of the compiler
// This will not compile after 0.5.0
contract OldContract {
    function someOldFunction(uint8 a) {
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

This will no longer compile with Solidity v0.5.0. However, you can define a compatible interface for it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

Note that we did not declare `anotherOldFunction` to be `view`, despite it being declared `constant` in the original contract. This is due to the fact that starting with Solidity v0.5.0 `staticcall` is used to call `view` functions. Prior to v0.5.0 the `constant` keyword was not enforced, so calling a function declared `constant` with `staticcall` may still revert, since the `constant` function may still attempt to modify storage. Consequently, when defining an interface for older contracts, you should only use `view` in place of `constant` in case you are absolutely sure that the function will work with `staticcall`.

Given the interface defined above, you can now easily use the already deployed pre-0.5.0 contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}

contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

Similarly, pre-0.5.0 libraries can be used by defining the functions of the library without implementation and supplying the address of the pre-0.5.0 library during linking (see *Using the Commandline Compiler* for how to use the command-line compiler for linking):

```
// This will not compile after 0.6.0
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

### 3.24.6 Example

The following example shows a contract and its updated version for Solidity v0.5.0 with some of the changes listed in this section.

Old version:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will not compile after 0.5.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // Function mutability not provided, not an error.
    function someInteger() internal returns (uint) { return 2; }

    // Function visibility not provided, not an error.
    // Function mutability not provided, not an error.
    function f(uint x) returns (bytes) {
        // Var is fine in this version.
        var z = someInteger();
        x += z;
        // Throw is fine in this version.
        if (x > 100)
            throw;
        bytes memory b = new bytes(x);
        y = -3 >> 1;
        // y == -1 (wrong, should be -2)
```

(continué en la próxima página)

(proviene de la página anterior)

```
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' causes an infinite loop.
        } while (x < 11);
        // Call returns only a Bool.
        bool success = address(other).call("f");
        if (!success)
            revert();
        else {
            // Local variables could be declared after their use.
            int y;
        }
        return b;
    }

    // No need for an explicit data location for 'arr'
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
        otherContract.transfer(1 ether);

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the first 4 bytes of x will be lost. This might lead to
        // unexpected behavior since bytesX are right padded.
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

New version:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;
// This will not compile after 0.6.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Function mutability must be specified.
    function someInteger() internal pure returns (uint) { return 2; }

    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        // The type must now be explicitly given.
        uint z = someInteger();
        x += z;
        // Throw is now disallowed.
        require(x <= 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' jumps to the condition below.
        } while (x < 11);

        // Call returns (bool, bytes).
        // Data location must be specified.
        (bool success, bytes memory data) = address(other).call("f");
        if (!success)
            revert();
        return data;
    }

    using AddressMakePayable for address;
    // Data location for 'arr' must be specified
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract, address␣
↪unknownContract) public payable {
        // 'otherContract.transfer' is not provided.
        // Since the code of 'OtherContract' is known and has the fallback
        // function, address(otherContract) has type 'address payable'.
        address(otherContract).transfer(1 ether);

        // 'unknownContract.transfer' is not provided.
        // 'address(unknownContract).transfer' is not provided
        // since 'address(unknownContract)' is not 'address payable'.
        // If the function takes an 'address' which you want to send
        // funds to, you can convert it to 'address payable' via 'uint160'.
        // Note: This is not recommended and the explicit type
        // 'address payable' should be used whenever possible.
        // To increase clarity, we suggest the use of a library for
        // the conversion (provided after the contract in this example).
        address payable addr = unknownContract.makePayable();
        require(addr.send(1 ether));

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the conversion is not allowed.
        // We need to convert to a common size first:
        bytes4 x4 = bytes4(x); // Padding happens on the right
        uint32 y = uint32(x4); // Conversion is consistent
        // 'msg.value' cannot be used in a 'non-payable' function.
        // We need to make the function payable
        myNumber += y + msg.value;
    }
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
// We can define a library for explicitly converting ``address``
// to ``address payable`` as a workaround.
library AddressMakePayable {
    function makePayable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

## 3.25 Solidity v0.6.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.6.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

### 3.25.1 Changes the Compiler Might not Warn About

This section lists changes where the behaviour of your code might change without the compiler telling you about it.

- The resulting type of an exponentiation is the type of the base. It used to be the smallest type that can hold both the type of the base and the type of the exponent, as with symmetric operations. Additionally, signed types are allowed for the base of the exponentiation.

### 3.25.2 Explicitness Requirements

This section lists changes where the code now needs to be more explicit, but the semantics do not change. For most of the topics the compiler will provide suggestions.

- Functions can now only be overridden when they are either marked with the `virtual` keyword or defined in an interface. Functions without implementation outside an interface have to be marked `virtual`. When overriding a function or modifier, the new keyword `override` must be used. When overriding a function or modifier defined in multiple parallel bases, all bases must be listed in parentheses after the keyword like so: `override(Base1, Base2)`.
- Member-access to `length` of arrays is now always read-only, even for storage arrays. It is no longer possible to resize storage arrays by assigning a new value to their length. Use `push()`, `push(value)` or `pop()` instead, or assign a full array, which will of course overwrite the existing content. The reason behind this is to prevent storage collisions of gigantic storage arrays.
- The new keyword `abstract` can be used to mark contracts as abstract. It has to be used if a contract does not implement all its functions. Abstract contracts cannot be created using the `new` operator, and it is not possible to generate bytecode for them during compilation.
- Libraries have to implement all their functions, not only the internal ones.
- The names of variables declared in inline assembly may no longer end in `_slot` or `_offset`.
- Variable declarations in inline assembly may no longer shadow any declaration outside the inline assembly block. If the name contains a dot, its prefix up to the dot may not conflict with any declaration outside the inline assembly block.
- In inline assembly, opcodes that do not take arguments are now represented as «built-in functions» instead of standalone identifiers. So `gas` is now `gas()`.

- State variable shadowing is now disallowed. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

### 3.25.3 Semantic and Syntactic Changes

This section lists changes where you have to modify your code and it does something else afterwards.

- Conversions from external function types to `address` are now disallowed. Instead external function types have a member called `address`, similar to the existing `selector` member.
- The function `push(value)` for dynamic storage arrays does not return the new length anymore (it returns nothing).
- The unnamed function commonly referred to as «fallback function» was split up into a new fallback function that is defined using the `fallback` keyword and a receive ether function defined using the `receive` keyword.
  - If present, the receive ether function is called whenever the call data is empty (whether or not ether is received). This function is implicitly `payable`.
  - The new fallback function is called when no other function matches (if the receive ether function does not exist then this includes calls with empty call data). You can make this function `payable` or not. If it is not `payable` then transactions not matching any other function which send value will revert. You should only need to implement the new fallback function if you are following an upgrade or proxy pattern.

### 3.25.4 New Features

This section lists things that were not possible prior to Solidity 0.6.0 or were more difficult to achieve.

- The *try/catch statement* allows you to react on failed external calls.
- `struct` and `enum` types can be declared at file level.
- Array slices can be used for calldata arrays, for example `abi.decode(msg.data[4:], (uint, uint))` is a low-level way to decode the function call payload.
- Natspec supports multiple return parameters in developer documentation, enforcing the same naming check as `@param`.
- Yul and Inline Assembly have a new statement called `leave` that exits the current function.
- Conversions from `address` to `address payable` are now possible via `payable(x)`, where `x` must be of type `address`.

### 3.25.5 Interface Changes

This section lists changes that are unrelated to the language itself, but that have an effect on the interfaces of the compiler. These may change the way how you use the compiler on the command line, how you use its programmable interface, or how you analyze the output produced by it.

#### New Error Reporter

A new error reporter was introduced, which aims at producing more accessible error messages on the command line. It is enabled by default, but passing `--old-reporter` falls back to the the deprecated old error reporter.

#### Metadata Hash Options

The compiler now appends the IPFS hash of the metadata file to the end of the bytecode by default (for details, see documentation on *contract metadata*). Before 0.6.0, the compiler appended the Swarm hash by default, and in order to still support this behaviour, the new command line option `--metadata-hash` was introduced. It allows you to select the hash to be produced and appended, by passing either `ipfs` or `swarm` as value to the `--metadata-hash` command line option. Passing the value `none` completely removes the hash.

These changes can also be used via the *Standard JSON Interface* and effect the metadata JSON generated by the compiler.

The recommended way to read the metadata is to read the last two bytes to determine the length of the CBOR encoding and perform a proper decoding on that data block as explained in the *metadata section*.

#### Yul Optimizer

Together with the legacy bytecode optimizer, the *Yul* optimizer is now enabled by default when you call the compiler with `--optimize`. It can be disabled by calling the compiler with `--no-optimize-yul`. This mostly affects code that uses ABI coder v2.

#### C API Changes

The client code that uses the C API of `libsolc` is now in control of the memory used by the compiler. To make this change consistent, `solidity_free` was renamed to `solidity_reset`, the functions `solidity_alloc` and `solidity_free` were added and `solidity_compile` now returns a string that must be explicitly freed via `solidity_free()`.

### 3.25.6 How to update your code

This section gives detailed instructions on how to update prior code for every breaking change.

- Change `address(f)` to `f.address` for `f` being of external function type.
- Replace `function () external [payable] { ... }` by either `receive() external payable { ... }`, `fallback() external [payable] { ... }` or both. Prefer using a `receive` function only, whenever possible.
- Change `uint length = array.push(value)` to `array.push(value);`. The new length can be accessed via `array.length`.
- Change `array.length++` to `array.push()` to increase, and use `pop()` to decrease the length of a storage array.
- For every named return parameter in a function's `@dev` documentation define a `@return` entry which contains the parameter's name as the first word. E.g. if you have function `f()` defined like `function f() public returns (uint value)` and a `@dev` annotating it, document its return parameters like so: `@return value The return value.`. You can mix named and un-named return parameters documentation so long as the notices are in the order they appear in the tuple return type.
- Choose unique identifiers for variable declarations in inline assembly that do not conflict with declarations outside the inline assembly block.

- Add `virtual` to every non-interface function you intend to override. Add `virtual` to all functions without implementation outside interfaces. For single inheritance, add `override` to every overriding function. For multiple inheritance, add `override(A, B, ..)`, where you list all contracts that define the overridden function in the parentheses. When multiple bases define the same function, the inheriting contract must override all conflicting functions.
- In inline assembly, add `()` to all opcodes that do not otherwise accept an argument. For example, change `pc` to `pc()`, and `gas` to `gas()`.

## 3.26 Solidity v0.7.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.7.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

### 3.26.1 Silent Changes of the Semantics

- Exponentiation and shifts of literals by non-literals (e.g. `1 << x` or `2 ** x`) will always use either the type `uint256` (for non-negative literals) or `int256` (for negative literals) to perform the operation. Previously, the operation was performed in the type of the shift amount / the exponent which can be misleading.

### 3.26.2 Changes to the Syntax

- In external function and contract creation calls, Ether and gas is now specified using a new syntax: `x.f{gas: 10000, value: 2 ether}(arg1, arg2)`. The old syntax – `x.f.gas(10000).value(2 ether)(arg1, arg2)` – will cause an error.
- The global variable `now` is deprecated, `block.timestamp` should be used instead. The single identifier `now` is too generic for a global variable and could give the impression that it changes during transaction processing, whereas `block.timestamp` correctly reflects the fact that it is just a property of the block.
- NatSpec comments on variables are only allowed for public state variables and not for local or internal variables.
- The token `gwei` is a keyword now (used to specify, e.g. `2 gwei` as a number) and cannot be used as an identifier.
- String literals now can only contain printable ASCII characters and this also includes a variety of escape sequences, such as hexadecimal (`\xff`) and unicode escapes (`\u20ac`).
- Unicode string literals are supported now to accommodate valid UTF-8 sequences. They are identified with the `unicode` prefix: `unicode"Hello "`.
- State Mutability: The state mutability of functions can now be restricted during inheritance: Functions with default state mutability can be overridden by `pure` and `view` functions while `view` functions can be overridden by `pure` functions. At the same time, public state variables are considered `view` and even `pure` if they are constants.

#### Inline Assembly

- Disallow `.` in user-defined function and variable names in inline assembly. It is still valid if you use Solidity in Yul-only mode.
- Slot and offset of storage pointer variable `x` are accessed via `x.slot` and `x.offset` instead of `x_slot` and `x_offset`.

### 3.26.3 Removal of Unused or Unsafe Features

#### Mappings outside Storage

- If a struct or array contains a mapping, it can only be used in storage. Previously, mapping members were silently skipped in memory, which is confusing and error-prone.
- Assignments to structs or arrays in storage does not work if they contain mappings. Previously, mappings were silently skipped during the copy operation, which is misleading and error-prone.

#### Functions and Events

- Visibility (`public` / `internal`) is not needed for constructors anymore: To prevent a contract from being created, it can be marked `abstract`. This makes the visibility concept for constructors obsolete.
- Type Checker: Disallow `virtual` for library functions: Since libraries cannot be inherited from, library functions should not be virtual.
- Multiple events with the same name and parameter types in the same inheritance hierarchy are disallowed.
- `using A for B` only affects the contract it is mentioned in. Previously, the effect was inherited. Now, you have to repeat the `using` statement in all derived contracts that make use of the feature.

#### Expressions

- Shifts by signed types are disallowed. Previously, shifts by negative amounts were allowed, but reverted at runtime.
- The `finney` and `szabo` denominations are removed. They are rarely used and do not make the actual amount readily visible. Instead, explicit values like `1e20` or the very common `gwei` can be used.

#### Declarations

- The keyword `var` cannot be used anymore. Previously, this keyword would parse but result in a type error and a suggestion about which type to use. Now, it results in a parser error.

### 3.26.4 Interface Changes

- JSON AST: Mark hex string literals with `kind: "hexString"`.
- JSON AST: Members with value `null` are removed from JSON output.
- NatSpec: Constructors and functions have consistent userdoc output.

### 3.26.5 How to update your code

This section gives detailed instructions on how to update prior code for every breaking change.

- Change `x.f.value(...)()` to `x.f{value: ...}()`. Similarly `(new C).value(...)()` to `new C{value: ...}()` and `x.f.gas(...).value(...)()` to `x.f{gas: ..., value: ...}()`.
- Change `now` to `block.timestamp`.
- Change types of right operand in shift operators to unsigned types. For example change `x >> (256 - y)` to `x >> uint(256 - y)`.
- Repeat the `using A for B` statements in all derived contracts if needed.
- Remove the `public` keyword from every constructor.
- Remove the `internal` keyword from every constructor and add `abstract` to the contract (if not already present).
- Change `_slot` and `_offset` suffixes in inline assembly to `.slot` and `.offset`, respectively.

## 3.27 Cambios introducidos en Solidity v0.8.0

Esta sección versa sobre los principales cambios introducidos en la versión 0.8.0 de Solidity. Para ver la lista completa registro de cambios de lanzamiento.

### 3.27.1 Cambios silenciosos en la semántica

Esta sección enumera los cambios en los que el código existente cambia su comportamiento sin que el compilador le notifique nada al respecto.

- Las operaciones aritmeticas se revertirán en underflow y overflow. Puede escribir `unchecked { ... }` para usar el comportamiento anterior del adaptador.

  Comprobar posibles problemas por overflow es bastante común, por lo que los hicimos predeterminados para aumentar la legibilidad del código, aunque eso signifique un ligero aumento del gas.

- ABI coder v2 está activado por defecto.

  Puede usar el comportamiento anterior con `pragma abicoder v1;`. La directiva `pragma experimental ABIEncoderV2;` aún es válida, pero está obsoleta y no tiene ningún efecto. Si quiere ser explícitco, use `pragma abicoder v2;`.

  Tenga en cuenta que el ABI coder v2 admite más tipos que v1 y realiza más controles sanitarios en los inputs. ABI coder v2 encarece algunas llamadas de función y también puede hacer que algunas llamadas al contrato sean revertidas y que, sin embargo, no se revertirían con ABI coder v1, cuando contienen datos que no se ajustan a los tipos de parámetros.

- La exponienciación es asociativa a derechas, por ejemplo, la expresión `a**b**c` es analizada como `a**(b**c)`. Antes de la versión 0.8.0, era analizada como `(a**b)**c`.

  Esta es la forma común de analizar el operador de exponenciación.

- Los asserts y otras verificaciones inernas como la división por cero o el desbordamiento aritmético no usan el opcode invalid, en su lugar usan el opcode revert. Más específicamente, usarán datos de erro equivalentes a una llamada a la función `Panic(uint256)` con un código de error acorde a las circunstancias.

  Esto ahorrará gas en errores mientras que permite a las herramientas de análisis estático distinguir estas situaciones entre un revert y un input inválido, como por ejemplo un fallo en `require`.

- Si se accede a un byte array en el almacenamiento cuya longitud está codificada incorrectamente, se generará un panic. Un contrato no puede entrar en esta situación a menos que se use inline assembly para modificar la representación raw de los byte arrays del almacenamiento (storage)
- Si se usan constantes en expresiones de longitud de array, las versiones previas de Solidity usarían precisión arbitraria en todas las ramas del arbol de decisiones. Ahora, is las variables constantes se usan como expresiones intermedias, sus valores serán propiamente redondeados de la misma forma que las expresiones en tiempo de ejecución.
- El tipo `byte` se ha eliminado. Era un alias de `bytes1`.

### 3.27.2 Nuevas restriciones

Esta sección enumera los cambios que pueden ocasionar que los contratos existentes no vuelvan a compilar nunca.

- Hay nuevas restricciones relacionadas con conversiones explícitas de literales. El comportamiento anterior en los siguientes casos probablemente era ambiguo:

  1. Conversiones explícitas de literales negativos y literales mayores que `type(uint160).max` a `address` no están permitidas.
  2. Las conversiones explícitas entre literales y un tipo entero `T` solo se permiten si el literal se encuentra entre `tipo(T).min` y `tipo(T).max`. En particular, reemplace los usos de `uint(-1)` con `tipo(uint).max`.
  3. **Las conversiones explícitas entre literales y enumeraciones solo se permiten si el literal puede** representan un valor del enumerado.
  4. Las conversiones explícitas entre literales y el tipo `address` (por ejemplo, `dirección(literal)`) tienen el

     tipo `address` en lugar de `address payable`. Se puede obtener un tipo de payable address utilizando una conversión explícita, es decir, `payable(literal)`.

- *Address literals* tienen el tipo `address` en lugar de `address payable`. Se pueden convertir a `address payable` usando una conversión explícita, por ejemplo `payable(0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF)`.
- Hay nuevas restricciones en las conversiones de tipos explícitos. La conversión sólo se permite cuando hay como máximo un cambio de signo, tamaño o la categoría de tipo (`int`, `address`, `bytesNN`, etc.). Para realizar varios cambios, utilice varias conversiones.

  Use la notación `T(S)` para denotar la conversión explícita `T(x)`, donde, `T` y `S` son tipos, y `x` es cualquier variable arbitraria de tipo `S`. Un ejemplo de tal conversión no permitida sería `uint16(int8)` ya que cambia el tamaño (8 bits a 16 bits) y signo (entero con signo a entero sin signo). Para hacer la conversión, tiene que ir a través de un tipo intermedio. En el ejemplo anterior, sería `uint16(uint8(int8))` o `uint16(int16(int8))`. Tenga en cuenta que las dos formas de convertir producirán resultados diferentes, por ejemplo, para `-1`. Los siguientes son algunos ejemplos de conversiones que no están permitidas por esta regla.

  - `address(uint)` y `uint(address)`: conversión simultanea de la categoría del tipo y tamaño. Reemplace esto por `address(uint160(uint))` y `uint(uint160(address))` respectivamente..
  - `payable(uint160)`, `payable(bytes20)` y `payable(integer-literal)`: conversión simultanea de la categoría del tipo and estado de la mutabilidad. Reemplace esto por `payable(address(uint160))`,

`payable(address(bytes20))` y `payable(address(integer-literal))` respectivamente. Tenga en cuenta que `payable(0)` es válido y no es una excepción de la regla.

- `int80(bytes10)` y `bytes10(int80)`: conversión simultanea de la categoría del tipo y signo. Reemplace esto por `int80(uint80(bytes10))` y `bytes10(uint80(int80)` respectivamente.
- `Contract(uint)`: conversión simultanea de la categoría del tipo y el tamaño. Reemplace esto por `Contract(address(uint160(uint)))`.

Estas conversiones fueron deshabilitadas para evitar ambiguedades. Por ejemplo, en la expresión `uint16 x = uint16(int8(-1))`, el valor de `x` dependería de que conversión, el signo ó el ancho, se aplicará antes.

- Las opciones de llamada de función solo se pueden dar una vez, es decir, `c.f{gas: 10000}{value: 1}()` no es válido y debe cambiarse a ``c.f{gas: 10000, value: 1}()``.
- Las funciones globales `log0`, `log1`, `log2`, `log3` y `log4` han sido eliminadas.

  Estas son funciones de bajo nivel que en gran parte se dejaron de utilizar. Se puede acceder a su comportamiento desde el inline assembly.
- Las definiciones de `enum` no pueden contener más de 256 miembros.

  Esto hará que sea seguro asumir que el tipo subyacente en la ABI siempre sea `uint8`.
- Las declaraciones con el nombre `this`, `super` y `_` no están permitidas, con la excepción de funciones y eventos públicos. La excepción es hacer posible declarar interfaces de contratos implementados en lenguajes distintos a Solidity que permiten tales nombres de funciones.
- Eliminada la compatibilidad con las secuencias de escape `\b`, `\f` y `\v` en el código. Todavía se pueden insertar a través de carácteres de escapes hexadecimales, p. `\x08`, `\x0c` y `\x0b`, respectivamente.
- Las variables globales `tx.origin` y `msg.sender` tienen el tipo `address` en lugar de `address payable`. Se pueden convertir en `address payable` usando una conversión explícita, por ejemplo, `payable(tx.origin)` o `payable(msg.sender)`.

  This change was done since the compiler cannot determine whether or not these addresses are payable or not, so it now requires an explicit conversion to make this requirement visible.

  Este cambio se realizó ya que el compilador no puede determinar si estas direcciones son payable ó no, por lo que ahora requiere una conversión explícita para cumplir este requisito.
- La conversión explícita al tipo `address` siempre devuelve un tipo not-payable `address`. En particular, Las siguientes conversiones explícitas tienen el tipo `address` en lugar de `address payable`:
  - `address(u)` donde `u` es una variable de tipo `uint160`. Se puede convertior `u` en el tipo `address payable` usando dos conversiones explícitas, por ejemplo, `payable(address(u))`.
  - `address(b)` donde `b` es una variable de tipo `bytes20`. Se puede convertir `b` en el tipo `address payable` usando dos conversiones explícitas, por ejemplo, `payable(address(b))`.
  - `address(c)` donde `c` es un contrato. Previamente, el tipo de retorno de esta conversión dependía si el contrato podía recibir Ether (bien por que tenía una función receive o bien por una

    función payable fallback). La conversión `payable(c)` tiene el tipo ``address

    payable`` y solamente está permitida cuando el contrato `c` puede recibir Ether. En general, siempre se puede convertir `c` en el tipo `address payable` usando la siguiente conversión explícita:

    `payable(address(c))`. Tenga en cuenta que `address(this)` pertenece a la misma categoría que `address(c)` y se aplican las misma reglas.
- El `chainid` incorporado en el inline assembly ahora se considera `view` en lugar de `pure`.
- La negación unaria ya no se puede usar, solo para enteros con signo.

### 3.27.3 Cambios en el interface

- La salida de `--combined-json` ha cambiado: Los campos JSON `abi`, `devdoc`, `userdoc` y `storage-layout` ahora son subobjetos. Antes de la versión 0.8.0 se usaban serializados como strings.
- El «legacy AST» ha sido eliminado (`--ast-json` en el interfaz de linea de comandos `legacyAST` para el standard JSON). Use «compact AST» (`--ast-compact--json` para `AST`) en su lugar.
- El antigüo informador (`--old-reporter`) ha sido eliminado.

### 3.27.4 Como actualizar su código

- Si confía en la aritmética subyacente, encuelva cada operación con `unchecked { ... }`.
- Opcional: Si usa SafeMath o una librería similar, cambie `x.add(y)` a `x + y`, `x.mul(y)` a `x * y` etc.
- Añada `pragma abicoder v1;` si quiere mantener el antiggüo codificador de ABI.
- Opcionalmente elimine `pragma experimental ABIEncoderV2` o `pragma abicoder v2` ya que es redundante.
- Cambie `byte` a `bytes1`.
- Agregue conversiones de tipo explícitas intermedias si es necesario.
- Combine `c.f{gas: 10000}{value: 1}()` a `c.f{gas: 10000, value: 1}()`.
- Cambie `msg.sender.transfer(x)` a `payable(msg.sender).transfer(x)` ó use una variable de almacenamiento de tipo `address payable`.
- Cambie `x**y**z` a `(x**y)**z`.
- Use inline assembly reemplazando `log0`, ..., `log4`.
- Niegue los enteros sin signo restándolos del valor máximo del tipo y sumando 1 (por ejemplo `type(uint256).max - x + 1`, mientras se asegura que $x$ no es cero)

## 3.28 Formato NatSpec

Los contratos de Solidity pueden usar una forma especial de comentarios para proveer documentación valiosa para funciones, variables de retorno y más. Esta forma especial se denomina Ethereum Natural Language Specification Format (NatSpec).

---

**Nota:** NatSpec se inspiró en Doxygen. Aunque usa comentarios y etiquetas al estilo Doxygen, no hay intención de mantener compatibilidad estricta con Doxygen. Por favor, examine cuidadosamente las etiquetas admitidas listadas abajo.

---

Esta documentación se segmenta en mensajes enfocados al desarrollador y mensajes al usuario final. Es posible que estos mensajes se muestren al usuario final (el humano) al momento que interactue con el contrato (i.e. firma de una transacción).

Se recomienda que los contratos de Solidity estén completamente comentados usando NatSpec para todas las interfaces públicas (todo en el ABI).

NatSpec incluye el formateo para comentarios que el autor del contrato inteligente usará, y los cuales son entendidos por el compilador de Solidity. También se detalla debajo la salida del compilador de Solidity el cual extrae estos comentarios a un formato de datos legibles mediante máquina.

NatSpec también pudiese incluir anotaciones usadas por herramientas de terceros. Estas se logran muy probablemente por medio de la etiqueta `@custom:<name>`, y un buen caso de uso es el análisis y las herramientas de verificación.

### 3.28.1 Ejemplo de Documentación

La documentación se inserta encima de cada `contrato`, `interfaz`, `biblioteca`, `función` y `evento` usando el formato de notación Doxygen. Una variable de estado `pública` es equivalente a una `función` para los propósitos de NatSpec.

- Para Solidity puede optar `///` para comentarios de una línea o múltiples líneas, o `/**` y finalizar con `*/`.

**Para Vyper, utilice `"""` sangrado al contenido interior con desnudo**
comentarios. Consulte la documentación de Vyper documentación.

El siguiente ejemplo muestra un contrato y una función usando todas las etiquetas disponibles.

---

**Nota:** El compilador de Solidity solo interpreta etiquetas si son externas o públicas. Puede usar comentarios similares para sus funciones privadas e internas, pero no serán interpretados.

Esto podría cambiar en el futuro.

---

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title Un simulador para árboles
/// @author Larry A. Gardner
/// @notice Puede usar este contrato solo para la simulación más básica
/// @dev Todas las llamadas a funciones están actualmente implementadas sin␣
↪efectos secundarios
/// @custom:experimental Este es un contrato experimental.
contract Tree {
    /// @notice Calcula la edad del árbol en años, redondeado hacia arriba, para␣
↪los árboles vivos.
    /// @dev El algoritmo de Alexandr N. Tetearing podría incrementar la precisión
    /// @param rings El número de anillos de una muestra dendrocronológica
    /// @return Edad en años, redondeado hacia arriba para años parciales
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice Retorna la cantidad de hojas que tiene el árbol.
    /// @dev Retorna solamente un número fijo.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

contract Plant {
    function leaves() external virtual pure returns(uint256) {
        return 3;
    }
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
contract KumquatTree is Tree, Plant {
    function age(uint256 rings) external override pure returns (uint256) {
        return rings + 2;
    }

    /// Retorna la cantidad de hojas que tiene este tipo específico de árbol
    /// @inheritdoc Tree
    function leaves() external override(Tree, Plant) pure returns(uint256) {
        return 3;
    }
}
```

### 3.28.2 Etiquetas

Todas las etiquetas son opcionales. La siguiente tabla explica el propósito de cada etiqueta NatSpec y donde puede ser usada. Como caso especial, si ninguna etiqueta se usa, entonces el compilador de Solidity interpretará un comentario con `///` o `/**` de la misma manera como si fuese etiquetado con `@notice`.

| Etiqueta | | Contexto |
|---|---|---|
| `@title` | Un título que debería describir el contrato o la interfaz | contrato, biblioteca, interfaz |
| `@author` | El nombre del autor | contrato, biblioteca, interfaz |
| `@notice` | Explica a un usuario final lo que esto hace | contrato, biblioteca, interfaz, función, variable de estado pública, evento |
| `@dev` | Explica a un desarrollador cualquier detalle extra | contrato, biblioteca, interfaz, función, variable de estado pública, evento |
| `@param` | Documenta un parámetro como en Doxygen (debe ser seguida por un nombre de parámetro) | función, evento |
| `@return` | Documenta las variables de retorno de la función de un contrato | función, variable de estado pública |
| `@inheritdoc` | Copia todas las etiquetas faltantes de la función base (debe ser seguida por el nombre del contrato) | función, variable de estado pública |
| `@custom:...` | Etiqueta personalizada, la semántica se define por la aplicación | en todas partes |

Si su función retorna múltiples valores, como `(int quotient, int remainder)`, entonces use múltiples declaraciones `@return` en el mismo formato como las declaraciones `@param`.

Las etiquetas personalizadas comienzan con `@custom:` y deben ser seguidas por una o más letras minúsculas o guiones. Sin embargo, no pueden comenzar con un guion. Se pueden usar en todas partes y forman parte de la documentación del desarrollador.

#### Expresiones dinámicas

El compilador de Solidity pasará a través de la documentación de NatSpec desde su código fuente Solidity hasta la salida JSON como se describe en esta guía. El consumidor de esta salida JSON, por ejemplo el software cliente del usuario final, puede presentar esto al usuario final directamente o podría aplicar algún preprocesamiento.

Por ejemplo, algún software cliente presentará:

```
/// @notice Esta función multiplicará `a` por 7
```

al usuario final como:

```
Esta función multiplicará 10 por 7
```

si una función es invocada y la entrada `a` se le asigna un valor de 10.

<<<<<<< HEAD Específicamente estas expresiones dinámicas están fuera del alcance de la documentación de Solidity y puede leer más en el proyecto radspec.

#### Notas de Herencia

Las funciones sin NatSpec automáticamente heredarán la documentación de su función base. Excepciones a esto son:

- Cuando los nombres de parámetros son diferentes.
- Cuando hay más de una función base.
- Cuando hay una etiqueta explícita `@inheritdoc` la cual especifica cuál contrato debería ser usado para heredar.

### 3.28.3 Salida de la Documentación

Cuando se analiza por el compilador, la documentación como la del ejemplo de arriba producirá dos archivos JSON diferentes. Se pretende que uno sea consumido por el usuario final como un aviso cuando una función se ejecuta y el otro para ser usado por el desarrollador.

Si el contrato de arriba se guarda como `ex1.sol`, entonces puede generar la documentación usando:

```
solc --userdoc --devdoc ex1.sol
```

Y la salida está abajo.

---

**Nota:** A partir de la versión de Solidity 0.6.11, la salida NatSpec también contiene un campo `version` y un `kind`. Actualmente `version` está establecido en `1` y `kind` debe ser o `user` o `dev`. En el futuro es posible que nuevas versiones sean introducidas, discontinuando viejas versiones.

---

### Documentación del Usuario

La documentación de arriba producirá el siguiente archivo JSON de la documentación del usuario como salida:

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calcula la edad del árbol en años, redondeado hacia arriba, para los␣
↪árboles vivos"
    }
  },
  "notice" : "Puede usar este contrato solo para la simulación más básica"
}
```

Note que la clave por la cual se encuentran los métodos es la signatura canónica de la función como se define en el *Contract ABI* y no simplemente el nombre de la función

### Documentación del Desarrollador

Aparte del archivo de documentación del usuario, un archivo JSON de la documentación del desarrollador también se debería producir y debería asemejarse a esto:

```
{
  "version" : 1,
  "kind" : "dev",
  "author" : "Larry A. Gardner",
  "details" : "Todas las llamadas a funciones están actualmente implementadas sin␣
↪efectos secundarios",
  "custom:experimental" : "Este es un contrato experimental.",
  "methods" :
  {
    "age(uint256)" :
    {
      "details" : "El algoritmo de Alexandr N. Tetearing podría incrementar la precisión
↪",
      "params" :
      {
        "rings" : "El número de anillos de una muestra dendrocronológica"
      },
      "return" : "Edad en años, redondeado hacia arriba para años parciales"
    }
  },
  "title" : "Un simulador para árboles"
}
```

## 3.29 Consideraciones de Seguridad

Aunque normalmente es bastante fácil construir software que funcione como se espera, es mucho más difícil controlar que nadie pueda usarlo de un manera **no** anticipada.

En Solidity, esto es incluso más importante porque puede usar contratos inteligentes para manejar tokens o, posiblemente, incluso cosas más valiosas. Además, cada ejecución de un contrato inteligente sucede en público y, además de ello, a menudo el código fuente está disponible.

Por supuesto siempre tiene que considerar cuánto está en riesgo: Puede comparar un contrato inteligente con un servicio web que está abierto al público (y por eso, también a actores maliciosos) y quizá incluso de código abierto. Si solo almacena su lista de compras en ese servicio web, es posible que no tenga mucho cuidado, pero si administra su cuenta bancaria usando ese servicio web, debería ser más cuidadoso.

Esta sección listará algunos peligros y recomendaciones de seguridad generales pero, por supuesto, nunca puede estar completo. También, tenga presente que incluso si su contrato inteligente está libre de bugs, el compilador o la plataforma misma pudiera tener un bug. Una lista de algunos bugs públicamente conocidos relevantes para la seguridad del compilador se pueden encontrar en la *lista de bugs conocidos*, la cual también es legible por máquina. Note que hay un programa de bug bounty que cubre el generador de código del compilador de Solidity.

Como siempre. con documentación de código abierto, por favor ayúdenos a extender esta sección (especialmente, algunos ejemplos no harían daño)!

NOTA: Además de la lista de abajo, puede encontrar más recomendaciones de seguridad y buenas prácticas en la lista de Guy Lando y the Consensys GitHub repo.

### 3.29.1 Peligros

#### Información privada y Aleatoriedad

Todo lo que use en un contrato inteligente es públicamente visible, incluso variables locales y variables de estado marcados como `private`.

El uso de números aleatorios en contratos inteligentes es bastante complicado si no quiere que los mineros sean capaces de hacer trampa.

#### Re-Entrancy

Cualquier interacción de un contrato (A) con otro contrato (B) y cualquier transferencia de Ether pasa el control a ese contrato (B). Esto permite que (B) vuelva a A antes de que esta interacción finalice. Para dar un ejemplo, el siguiente código contiene un bug (es solo un fragmento y no un contrato completo):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// ESTE CONTRATO CONTIENE UN BUG - NO USAR
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }
}
```

El problema aquí no es demasiado serio debido al gas limitado como parte de `send`, pero aun así expone una debilidad: La transferencia de Ether siempre puede incluir ejecución de código, así que el destinatario podría ser un contrato que vuelve a `withdraw`. Esto le permitiría obtener múltiples reembolsos y básicamente recuperar todo el Ether del contrato. En particular, el siguiente contrato permitiría a un atacante recuperar múltiples veces al usar `call` el cual envía todo el gas sobrante por defecto:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

// ESTE CONTRATO CONTIENE UN BUG - NO USAR
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

Para evitar un ataque de re-entrancy, puede usar el patrón Verificaciónes-Efectos-Interacción como se esboza más abajo:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        payable(msg.sender).transfer(share);
    }
}
```

El patrón Verificaciónes-Efectos-Interaccion garantiza que todas las rutas de código a través de un contrato completen todas las comprobaciones necesarias de los parámetros suministrados antes de modificar el estado del contrato (Verificación); solo entonces realiza cambios en el estado (Efectos); puede hacer llamadas a funciones en otros contratos *después* de que todos los cambios de estado planificados se hayan escrito en almacenamiento (interacciones). Esta es una forma común e infalible de prevenir *ataques de reingreso*, donde una llamada externa contrato malicioso es capaz de gastar dos veces una asignación, retirar dos veces un saldo, entre otras cosas, mediante el uso de la lógica que vuelve a llamar a la contrato original antes de que haya finalizado su transacción.

Note que el ataque por re-entrancy no solo es un efecto de la transferencia de Ether sino de cualquier llamada de función sobre otro contrato. Además, también tiene que tener en cuenta situaciones de contratos múltiples. Una llamada a un contrato podría modificar el estado de otro contrato del cual depende.

### Límite de Gas y Bucles

Los bucles que no tienen un número fijo de iteraciones, por ejemplo, bucles que dependen de valores almacenados, tienen que ser usados cuidadosamente: Debido al límite de gas de bloque, las transacciones solo pueden consumir una cierta cantidad de gas. O explícitamente o solo debido a una operación normal, el número de iteraciones en un bucle puede crecer más allá del límite de gas de bloque, el cual puede causar que el contrato completo se pare en cierto punto. Esto no puede aplicar a las funciones `view` que solo son ejecutadas para leer datos de la cadena de bloques. Aun así, tales funciones podrían ser invocadas por otros contratos como parte de operaciones en cadena y parar aquellas. Por favor, sea explícito sobre tales casos en la documentación de sus contratos.

### Envío y Recepción de Ether

- Ni los contratos ni «cuentas externas» son actualmente capaces de prevenir que alguien les envía Ether. Los contratos pueden reaccionar y rechazar una transferencia regular, pero hay maneras de mover Ether sin crear un message call. Una manera es simplemente minar a la dirección de contrato y la segunda manera es usar `selfdestruct(x)`.
- Si un contrato recibe Ether (sin una función siendo llamada), o la función *receive Ether* o la función *fallback* se jecutan. Si no tiene una función receive ni fallback, el Ether será rechazado (al lanzar una excepción). Durante la ejecución de una de estas funciones, el contrato solo puede depender del «estipendio de gas» que se le pase (2300 gas) estando disponible en ese momento. Este estipendio no es suficiente para modificar el almacenamiento (aunque no dé por sentado esto, el estipendio podría cambiar con futuros hard forks). Para asegurarse de que su contrato puede recibir Ether de esa manera, compruebe los requerimientos de gas de las funciones receive y fallback (por ejemplo, en la sección de «details» de Remix).
- Hay una manera de enviar más gas al contrato receptor usando `addr.call{value: x}("")`. Esto es esencialmente lo mismo como `addr.transfer(x)`, solo que envía todo el gas restante y facilita al receptor realizar acciones más caras (y retorna un código de fallo en lugar de propagar automáticamente el error). Esto podría incluir el llamado de vuelta al contrato de envío u otros cambios de estado los cuales usted no podría pensar. Así que permite gran flexibilidad para los usuarios honestos pero también para actores maliciosos.
- Use las unidades más precisas como sea posible para representar la cantidad de wei, puesto que pierde todo lo que esté redondeado debido a una falta de precisión.
- Si usted quiere enviar Ether usando `address.transfer`, hay ciertos detalles de los cuales debe estar al tanto:
  1. Si el receptor es un contrato, causa que la función receive o fallback sea ejecutada lo cual puede, después, llamar de vuelta al contrato emisor.
  2. El envío de Ether puede fallar debido a la profundidad de la llamada al superar 1024. Ya que el que llama está en total control de la profundidad de la llamada, ellos pueden forzar que la transferencia falle; tome esta posibilidad en cuenta o use `send` y asegúrese de siempre corroborar el valor de retorno. Mejor aún, escriba su contrato usando un patrón en donde el receptor pueda retirar Ether.
  3. El envío de Ether también puede fallar debido a que la ejecución del contrato receptor requiere más de la cantidad de gas asignada (explícitamente al usar *require*, *assert*, *revert* o porque la operación es demasiado costosa) - it «runs out of gas» (OOG) se consumió todo el gas. Si usa `transfer` o `send` con una comprobación del valor de retorno, esto podría proveer un medio para que el receptor bloquee el progreso en el contrato remitente. Una vez más, la mejor práctica aquí es usar un patrón *«withdraw» en lugar de un patrón «send»*.

### Profundidad de la Pila de Llamadas

Las llamadas a funciones externas pueden fallar en cualquier momento debido a que exceden el límite de tamaño máximo de la pila de llamadas de 1024. En tales situaciones, Solidity lanza una excepción. Actores maliciosos podrían ser capaces de forzar la pila de llamadas a un valor alto antes de que ellos interactúen con su contrato. Note que, desde el hardfork Tangerine Whistle, la regla 63/64 hace impráctico el ataque a la profundidad de la pila de llamadas. También note que la pila de llamadas y la pila de expresiones no están relacionadas, a pesar de que ambas tienen un límite de tamaño de 1024 ranuras de pilas.

Note que `.send()` **no** lanza una excepción si la pila de llamadas está agotada sino que retorna `false` en ese caso. Las funciones de bajo nivel `.call()`, `.delegatecall()` y `.staticcall()` se comportan de la misma manera.

### Proxies Autorizados

Si su contrato puede actuar como un proxy, i. e. si puede llamar contratos arbitrarios con datos suministrados por el usuario, entonces el usuario puede esencialmente asumir la identidad del contrato proxy. Incluso si tiene otras medidas protectivas en lugar, es mejor construir su sistema de contratos de tal manera que el proxy no tenga cualquier permiso (ni siquiera para sí mismo). De ser necesario, puede lograr eso usando un segundo proxy:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ProxyWithMoreFunctionality {
    PermissionlessProxy proxy;

    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return proxy.callOther(addr, payload);
    }
    // Otras funciones y otra funcionalidad
}

// Este es el contrato entero, no tiene otra funcionalidad y
// no requiere privilegios para funcionar.
contract PermissionlessProxy {
    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return addr.call(payload);
    }
}
```

### tx.origin

Nunca use tx.origin para autorización. Digamos que tiene un contrato de una billetera como este:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// ESTE CONTRATO CONTIENE UN BUG - NO USAR
contract TxUserWallet {
    address owner;

    constructor() {
        owner = msg.sender;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }

    function transferTo(address payable dest, uint amount) public {
        // EL BUS ESTA JUSTO AQUÍ, usted debe usar msg.sender en lugar de tx.origin
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Ahora alguien puede engañarlo al enviar Ether a la dirección de esta billetera de ataque:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Si su billetera ha corroborado `msg.sender` para autorización, tendría la dirección de la billetera de ataque, en lugar de la dirección del propietario. Pero al validar `tx.origin`, obtiene la dirección original que empezó la transacción, el cual es aún la dirección del propietario. La billetera de ataque instantáneamente vacía todos sus fondos.

### Complemento a Dos / Underflows / Overflows

Como en muchos lenguajes de programación, los tipo enteros de Solidity no son de hecho enteros. Ellos se parecen a enteros cuando los valores son pequeños, pero no pueden representar arbitrariamente números grandes.

El siguiente código causa overflow porque el resultado de la adición es demasiado grande para ser almacenado en el tipo `uint8`:

```
uint8 x = 255;
uint8 y = 1;
return x + y;
```

Solidity tiene dos modos por medio de los cuales trata overflows: modo verificado y no verificado o «envolvente».

El modo por defecto verificado detectará overflows y causará una aserción que falla. Usted pude deshabilitar esta verificación usando `unchecked { ... }`, lo que causa que el overflow sea ignorado silenciosamente. El código de arriba retornaría `0` si estuviese envuelto con `unchecked { ... }`.

Incluso en modo verificado, no asuma que está protegido de bugs de overflow. En este modo, overflows siempre revertirán. Si no es posible evitar el overflow, esto puede llevar a que un contrato inteligente se quede atascado en cierto estado.

En general, lea sobre los límites de la representación del complemento a dos, la cual tiene otros casos especiales para números con signos.

Trate de usar `require` para limitar el tamaño de entradas a un rango rasonable y use *SMT checker* para encontrar potenciales overflows.

### Limpieza de Mappings

El tipo `mapping` (véase *Tipos Mapping*) de Solidity es una estructura de datos clave-valor de solo almacenamiento que no mantiene un registro de las claves que fueron asignadas un valor no nulo. Debido a ello, limpiar un mapping sin información extra sobre las claves escritas no es posible. Si `mapping` se usa como el tipo base de un array de almacenamiento dinámico, borrar o quitar el último elemento del array no tendrá efecto sobre los elemetos del `mapping`. Lo mismo sucede, por ejemplo, si un `mapping` se usa como el tipo de un campo miembro de un `struct` que es el tipo base de un array de almacenamiento dinámico. `mapping` también es ignorado in asignaciones de structs o arrays que contienen un `mapping`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Map {
    mapping(uint => uint)[] array;

    function allocate(uint newMaps) public {
        for (uint i = 0; i < newMaps; i++)
            array.push();
    }

    function writeMap(uint map, uint key, uint value) public {
        array[map][key] = value;
    }

    function readMap(uint map, uint key) public view returns (uint) {
        return array[map][key];
    }

    function eraseMaps() public {
        delete array;
    }
}
```

Considere el ejemplo de arriba y la siguiente secuencia de llamadas: `allocate(10)`, `writeMap(4, 128, 256)`. En este punto, llamar `readMap(4, 128)` retorna 256. Si llamamos `eraseMaps`, la longitud de la variable de estado `array` es reestablecida a cero, pero ya que los elementos de `mapping` no pueden ser ceros, su información permanece viva en el almacenamiento del contrato. Luego de borrar `array`, invocar `allocate(5)` nos permite acceder a `array[4]` otra vez, y llamar `readMap(4, 128)` retorna 256 incluso sin llarmar a `writeMap`.

SI su información de `mapping` debe ser eliminada, considere usar una biblioteca similar a iterable mapping, que le permite atravesar las claves y eliminar sus valores en el `mapping` apropiado.

#### Detalles Menores

- Tipos que no ocupan los 32 bytes completos podrían contener «bits sucios de orden mayor». Esto es especialmente importante si usted accede a `msg.data` - representa un riesgo de maleabilidad: Usted puede hacer transacciones que invoquen una función `f(uint8 x)` con un argumento de `0xff000001` y con `0x00000001`. Ambos se suministran al contrato y ambos parecerán iguales al número 1 en lo que respecta a `x`, pero `msg.data` será diferente, así que si usa `keccak256(msg.data)` para algo , obtendrá resultados diferentes.

### 3.29.2 Recomendaciones

#### Tome las advertencias seriamente

Si el compilador le advierte sobre algo, usted debería cambiarlo. Incluso si usted no cree que esta advertencia particular tiene implicaciones de seguridad, podría haber otro asunto escondido debajo de ello. Cualquiera advertencia del compilador que emitimos puede ser silenciada por cambios ligeros al código.

Siempre use la última versión del compilador para ser notificado sobre todas las advertencias introducidas recientemente.

Mensajes de tipo `info` emitidos por el compilador no son peligrosos y simplemente representan sugerencias extras e información opcional que el compilador cree que podría ser útil al usuario.

#### Limite la cantidad de Ether

Limite la cantidad de Ether (u otros tokens) que pueden ser almacenados en un contrato inteligente. Si su código fuente, el compilador o la plataforma tienen un bug, estos fondos podrían perderse. Si quiere limitar su perdida, limite la cantidad de Ether.

#### Manténgalo modular y pequeño

Mantenga sus contratos pequeños y fácilmente entendibles. Eliga funcionalidad no relacionada de otros contratos o bibliotecas. Recomendaciones generales sobre la calidad del código fuente por supuesto aplican: Limite la cantidad de variables locales, la longitud de funciones, etcétera. Documente sus funciones, de modo que otros puedan ver cuál era su intención y si es diferente de lo que hace el código.

#### Use el patrón Checks-Effects-Interactions

La mayoría de las funciones llevarán a cabo primero algunas verificaciones (quién llamó la función, están los argumentos en rango, se envió suficiente Ether, la persona tiene tokens, etc.). Estas verificaciones deberían ser hechas primero.

Como segundo paso, si todas las verificaciones pasaron, los efectos a las variables de estado del contrato actual deberían tener lugar. La interacción con otros contratos deberían ser el último paso en cualquier función.

Los primeros contratos retrasaban algunos efectos y esperaban por invocaciones a funciones externas para retornar un estado de no error. A menudo esto es un error serio debido a el problema de re-entrancy explicado arriba.

Note que, también, llamadas a contratos conocidos podrían, después, causar llamadas a contratos desconocidos, así que es probablemente mejor aplicar siempre este patrón.

### Incluye un modo de fallo seguro

Aunque al hacer su sistema completamente descentralizado removerá cualquier intermediario, podría ser una buena idea, especialmente para código nuevo, incluir algún tipo de mecanismo de fallo seguro:

Puede agregar una función en su contrato inetligente que lleve a cabo algunas auto-verificaciones como «¿Se ha perdido Ether?», «¿La suma de los tokens es igual al balance del contrato?» o cosas similares. No olvide que no puede usar demasiado gas para eso, así que ayuda por medio de computación fuera de la cadena podría ser necesario.

Si la auto-verificación falla, el contrato automáticamente cambia a un modo de tipo «fallo seguro», el cual, por ejemplo, deshabilita la mayoría de las características, pasa control a un tercero fijo y confiable o solo convierte el contrato a un simple contrato de tipo «devuélvame mi dinero».

### Solicite revisión por pares

Mientras más personas examinan una pieza de código, más problemas se encuentran. Pedir a las personas que revisen su código también ayuda como un verificación para averiguar si su código es fácil de entender - un criterio muy importante para buenos contratos inteligentes.

## 3.30 SMTChecker and Formal Verification

Using formal verification it is possible to perform an automated mathematical proof that your source code fulfills a certain formal specification. The specification is still formal (just as the source code), but usually much simpler.

Note that formal verification itself can only help you understand the difference between what you did (the specification) and how you did it (the actual implementation). You still need to check whether the specification is what you wanted and that you did not miss any unintended effects of it.

Solidity implements a formal verification approach based on SMT (Satisfiability Modulo Theories) and Horn solving. The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements. That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true. If an assertion failure is found, a counterexample may be given to the user showing how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe.

The other verification targets that the SMTChecker checks at compile time are:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.
- Out of bounds index access.
- Insufficient funds for a transfer.

All the targets above are automatically checked by default if all engines are enabled, except underflow and overflow for Solidity >=0.8.7.

The potential warnings that the SMTChecker reports are:

- `<failing property> happens here.`. This means that the SMTChecker proved that a certain property fails. A counterexample may be given, however in complex situations it may also not show a counterexample. This result may also be a false positive in certain cases, when the SMT encoding adds abstractions for Solidity code that is either hard or impossible to express.

- `<failing property> might happen here`. This means that the solver could not prove either case within the given timeout. Since the result is unknown, the SMTChecker reports the potential failure for soundness. This may be solved by increasing the query timeout, but the problem might also simply be too hard for the engine to solve.

To enable the SMTChecker, you must select *which engine should run*, where the default is no engine. Selecting the engine enables the SMTChecker on all files.

**Nota:** Prior to Solidity 0.8.4, the default way to enable the SMTChecker was via `pragma experimental SMTChecker;` and only the contracts containing the pragma would be analyzed. That pragma has been deprecated, and although it still enables the SMTChecker for backwards compatibility, it will be removed in Solidity 0.9.0. Note also that now using the pragma even in a single file enables the SMTChecker for all files.

**Nota:** The lack of warnings for a verification target represents an undisputed mathematical proof of correctness, assuming no bugs in the SMTChecker and the underlying solver. Keep in mind that these problems are *very hard* and sometimes *impossible* to solve automatically in the general case. Therefore, several properties might not be solved or might lead to false positives for large contracts. Every proven property should be seen as an important achievement. For advanced users, see *SMTChecker Tuning* to learn a few options that might help proving more complex properties.

### 3.30.1 Tutorial

#### Overflow

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        return add(x, y);
    }
}
```

The contract above shows an overflow check example. The SMTChecker does not check underflow and overflow by default for Solidity >=0.8.7, so we need to use the command line option `--model-checker-targets "underflow, overflow"` or the JSON option `settings.modelChecker.targets = ["underflow", "overflow"]`. See *this section for targets configuration*. Here, it reports the following:

```
Warning: CHC: Overflow (resulting value larger than 2**256 - 1) happens here.
Counterexample:
```

(continué en la próxima página)

(proviene de la página anterior)

```
x = 1, y = 115792089237316195423570985008687907853269984665640564039457584007913129639935
 = 0

Transaction trace:
Overflow.constructor(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935)
State: x = 1, y =␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935
Overflow.stateAdd()
    Overflow.add(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935) --␣
↪internal call
 --> o.sol:9:20:
  |
9 |             return x_ + y_;
  |                    ^^^^^^^
```

If we add `require` statements that filter out overflow cases, the SMTChecker proves that no overflow is reachable (by not reporting warnings):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        require(x < type(uint128).max);
        require(y < type(uint128).max);
        return add(x, y);
    }
}
```

### Assert

An assertion represents an invariant in your code: a property that must be true *for all transactions, including all input and storage values*, otherwise there is a bug.

The code below defines a function `f` that guarantees no overflow. Function `inv` defines the specification that `f` is monotonically increasing: for every possible pair `(a, b)`, if `b > a` then `f(b) > f(a)`. Since `f` is indeed monotonically increasing, the SMTChecker proves that our property is correct. You are encouraged to play with the property and the function definition to see what results come out!

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Monotonic {
    function f(uint x) internal pure returns (uint) {
        require(x < type(uint128).max);
        return x * 42;
    }

    function inv(uint a, uint b) public pure {
        require(b > a);
        assert(f(b) > f(a));
    }
}
```

We can also add assertions inside loops to verify more complicated properties. The following code searches for the maximum element of an unrestricted array of numbers, and asserts the property that the found element must be greater or equal every element in the array.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m >= a[i]);

        return m;
    }
}
```

Note that in this example the SMTChecker will automatically try to prove three properties:

1. `++i` in the first loop does not overflow.
2. `++i` in the second loop does not overflow.
3. The assertion is always true.

**Nota:** The properties involve loops, which makes it *much much* harder than the previous examples, so beware of loops!

All the properties are correctly proven safe. Feel free to change the properties and/or add restrictions on the array to see different results. For example, changing the code to

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        require(a.length >= 5);
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m > a[i]);

        return m;
    }
}
```

gives us:

```
Warning: CHC: Assertion violation happens here.
Counterexample:

a = [0, 0, 0, 0, 0]
 = 0

Transaction trace:
Test.constructor()
Test.max([0, 0, 0, 0, 0])
  --> max.sol:14:4:
   |
14 |            assert(m > a[i]);
```

### State Properties

So far the examples only demonstrated the use of the SMTChecker over pure code, proving properties about specific operations or algorithms. A common type of properties in smart contracts are properties that involve the state of the contract. Multiple transactions might be needed to make an assertion fail for such a property.

As an example, consider a 2D grid where both axis have coordinates in the range (-2^128, 2^128 - 1). Let us place a robot at position (0, 0). The robot can only move diagonally, one step at a time, and cannot move outside the grid. The robot's state machine can be represented by the smart contract below.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Robot {
    int x = 0;
    int y = 0;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    modifier wall {
        require(x > type(int128).min && x < type(int128).max);
        require(y > type(int128).min && y < type(int128).max);
        _;
    }

    function moveLeftUp() wall public {
        --x;
        ++y;
    }

    function moveLeftDown() wall public {
        --x;
        --y;
    }

    function moveRightUp() wall public {
        ++x;
        ++y;
    }

    function moveRightDown() wall public {
        ++x;
        --y;
    }

    function inv() public view {
        assert((x + y) % 2 == 0);
    }
}
```

Function `inv` represents an invariant of the state machine that `x + y` must be even. The SMTChecker manages to prove that regardless how many commands we give the robot, even if infinitely many, the invariant can *never* fail. The interested reader may want to prove that fact manually as well. Hint: this invariant is inductive.

We can also trick the SMTChecker into giving us a path to a certain position we think might be reachable. We can add the property that (2, 4) is *not* reachable, by adding the following function.

```
function reach_2_4() public view {
    assert(!(x == 2 && y == 4));
}
```

This property is false, and while proving that the property is false, the SMTChecker tells us exactly *how* to reach (2, 4):

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 2, y = 4

Transaction trace:
Robot.constructor()
State: x = 0, y = 0
Robot.moveLeftUp()
State: x = (- 1), y = 1
```

(continué en la próxima página)

(proviene de la página anterior)

```
Robot.moveRightUp()
State: x = 0, y = 2
Robot.moveRightUp()
State: x = 1, y = 3
Robot.moveRightUp()
State: x = 2, y = 4
Robot.reach_2_4()
  --> r.sol:35:4:
   |
35 |            assert(!(x == 2 && y == 4));
   |            ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Note that the path above is not necessarily deterministic, as there are other paths that could reach (2, 4). The choice of which path is shown might change depending on the used solver, its version, or just randomly.

### External Calls and Reentrancy

Every external call is treated as a call to unknown code by the SMTChecker. The reasoning behind that is that even if the code of the called contract is available at compile time, there is no guarantee that the deployed contract will indeed be the same as the contract where the interface came from at compile time.

In some cases, it is possible to automatically infer properties over state variables that are still true even if the externally called code can do anything, including reenter the caller contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

interface Unknown {
    function run() external;
}

contract Mutex {
    uint x;
    bool lock;

    Unknown immutable unknown;

    constructor(Unknown u) {
        require(address(u) != address(0));
        unknown = u;
    }

    modifier mutex {
        require(!lock);
        lock = true;
        _;
        lock = false;
    }

    function set(uint x_) mutex public {
        x = x_;
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function run() mutex public {
        uint xPre = x;
        unknown.run();
        assert(xPre == x);
    }
}
```

The example above shows a contract that uses a mutex flag to forbid reentrancy. The solver is able to infer that when `unknown.run()` is called, the contract is already «locked», so it would not be possible to change the value of `x`, regardless of what the unknown called code does.

If we «forget» to use the `mutex` modifier on function `set`, the SMTChecker is able to synthesize the behaviour of the externally called code so that the assertion fails:

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 1, lock = true, unknown = 1

Transaction trace:
Mutex.constructor(1)
State: x = 0, lock = false, unknown = 1
Mutex.run()
    unknown.run() -- untrusted external call, synthesized as:
        Mutex.set(1) -- reentrant call
  --> m.sol:32:3:
   |
32 |                assert(xPre == x);
   |                ^^^^^^^^^^^^^^^^
```

### 3.30.2 SMTChecker Options and Tuning

#### Timeout

The SMTChecker uses a hardcoded resource limit (`rlimit`) chosen per solver, which is not precisely related to time. We chose the `rlimit` option as the default because it gives more determinism guarantees than time inside the solver.

This options translates roughly to «a few seconds timeout» per query. Of course many properties are very complex and need a lot of time to be solved, where determinism does not matter. If the SMTChecker does not manage to solve the contract properties with the default `rlimit`, a timeout can be given in milliseconds via the CLI option `--model-checker-timeout <time>` or the JSON option `settings.modelChecker.timeout=<time>`, where 0 means no timeout.

### Verification Targets

The types of verification targets created by the SMTChecker can also be customized via the CLI option `--model-checker-target <targets>` or the JSON option `settings.modelChecker.targets=<targets>`. In the CLI case, `<targets>` is a no-space-comma-separated list of one or more verification targets, and an array of one or more targets as strings in the JSON input. The keywords that represent the targets are:

- Assertions: `assert`.
- Arithmetic underflow: `underflow`.
- Arithmetic overflow: `overflow`.
- Division by zero: `divByZero`.
- Trivial conditions and unreachable code: `constantCondition`.
- Popping an empty array: `popEmptyArray`.
- Out of bounds array/fixed bytes index access: `outOfBounds`.
- Insufficient funds for a transfer: `balance`.
- All of the above: `default` (CLI only).

A common subset of targets might be, for example: `--model-checker-targets assert,overflow`.

All targets are checked by default, except underflow and overflow for Solidity >=0.8.7.

There is no precise heuristic on how and when to split verification targets, but it can be useful especially when dealing with large contracts.

### Unproved Targets

If there are any unproved targets, the SMTChecker issues one warning stating how many unproved targets there are. If the user wishes to see all the specific unproved targets, the CLI option `--model-checker-show-unproved` and the JSON option `settings.modelChecker.showUnproved = true` can be used.

### Verified Contracts

By default all the deployable contracts in the given sources are analyzed separately as the one that will be deployed. This means that if a contract has many direct and indirect inheritance parents, all of them will be analyzed on their own, even though only the most derived will be accessed directly on the blockchain. This causes an unnecessary burden on the SMTChecker and the solver. To aid cases like this, users can specify which contracts should be analyzed as the deployed one. The parent contracts are of course still analyzed, but only in the context of the most derived contract, reducing the complexity of the encoding and generated queries. Note that abstract contracts are by default not analyzed as the most derived by the SMTChecker.

The chosen contracts can be given via a comma-separated list (whitespace is not allowed) of <source>:<contract> pairs in the CLI: `--model-checker-contracts "<source1.sol:contract1>,<source2.sol:contract2>, <source2.sol:contract3>"`, and via the object `settings.modelChecker.contracts` in the *JSON input*, which has the following form:

```
"contracts": {
    "source1.sol": ["contract1"],
    "source2.sol": ["contract2", "contract3"]
}
```

### Trusted External Calls

By default, the SMTChecker does not assume that compile-time available code is the same as the runtime code for external calls. Take the following contracts as an example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Ext {
    uint public x;
    function setX(uint _x) public { x = _x; }
}
contract MyContract {
    function callExt(Ext _e) public {
        _e.setX(42);
        assert(_e.x() == 42);
    }
}
```

When `MyContract.callExt` is called, an address is given as the argument. At deployment time, we cannot know for sure that address `_e` actually contains a deployment of contract `Ext`. Therefore, the SMTChecker will warn that the assertion above can be violated, which is true, if `_e` contains another contract than `Ext`.

However, it can be useful to treat these external calls as trusted, for example, to test that different implementations of an interface conform to the same property. This means assuming that address `_e` indeed was deployed as contract `Ext`. This mode can be enabled via the CLI option `--model-checker-ext-calls=trusted` or the JSON field `settings.modelChecker.extCalls: "trusted"`.

Please be aware that enabling this mode can make the SMTChecker analysis much more computationally costly.

An important part of this mode is that it is applied to contract types and high level external calls to contracts, and not low level calls such as `call` and `delegatecall`. The storage of an address is stored per contract type, and the SMTChecker assumes that an externally called contract has the type of the caller expression. Therefore, casting an `address` or a contract to different contract types will yield different storage values and can give unsound results if the assumptions are inconsistent, such as the example below:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract D {
    constructor(uint _x) { x = _x; }
    uint public x;
    function setX(uint _x) public { x = _x; }
}

contract E {
    constructor() { x = 2; }
    uint public x;
    function setX(uint _x) public { x = _x; }
}

contract C {
    function f() public {
        address d = address(new D(42));
```

(continué en la próxima página)

(proviene de la página anterior)

```
        // `d` was deployed as `D`, so its `x` should be 42 now.
        assert(D(d).x() == 42); // should hold
        assert(D(d).x() == 43); // should fail

        // E and D have the same interface, so the following
        // call would also work at runtime.
        // However, the change to `E(d)` is not reflected in `D(d)`.
        E(d).setX(1024);

        // Reading from `D(d)` now will show old values.
        // The assertion below should fail at runtime,
        // but succeeds in this mode's analysis (unsound).
        assert(D(d).x() == 42);
        // The assertion below should succeed at runtime,
        // but fails in this mode's analysis (false positive).
        assert(D(d).x() == 1024);
    }
}
```

Due to the above, make sure that the trusted external calls to a certain variable of `address` or `contract` type always have the same caller expression type.

It is also helpful to cast the called contract's variable as the type of the most derived type in case of inheritance.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

interface Token {
    function balanceOf(address _a) external view returns (uint);
    function transfer(address _to, uint _amt) external;
}

contract TokenCorrect is Token {
    mapping (address => uint) balance;
    constructor(address _a, uint _b) {
        balance[_a] = _b;
    }
    function balanceOf(address _a) public view override returns (uint) {
        return balance[_a];
    }
    function transfer(address _to, uint _amt) public override {
        require(balance[msg.sender] >= _amt);
        balance[msg.sender] -= _amt;
        balance[_to] += _amt;
    }
}

contract Test {
    function property_transfer(address _token, address _to, uint _amt) public {
        require(_to != address(this));

        TokenCorrect t = TokenCorrect(_token);
```

(continué en la próxima página)

(proviene de la página anterior)

```
        uint xPre = t.balanceOf(address(this));
        require(xPre >= _amt);
        uint yPre = t.balanceOf(_to);

        t.transfer(_to, _amt);
        uint xPost = t.balanceOf(address(this));
        uint yPost = t.balanceOf(_to);

        assert(xPost == xPre - _amt);
        assert(yPost == yPre + _amt);
    }
}
```

Note that in function `property_transfer`, the external calls are performed on variable `t`

Another caveat of this mode are calls to state variables of contract type outside the analyzed contract. In the code below, even though `B` deploys `A`, it is also possible for the address stored in `B.a` to be called by anyone outside of `B` in between transactions to `B` itself. To reflect the possible changes to `B.a`, the encoding allows an unbounded number of calls to be made to `B.a` externally. The encoding will keep track of `B.a`'s storage, therefore assertion (2) should hold. However, currently the encoding allows such calls to be made from `B` conceptually, therefore assertion (3) fails. Making the encoding stronger logically is an extension of the trusted mode and is under development. Note that the encoding does not keep track of storage for `address` variables, therefore if `B.a` had type `address` the encoding would assume that its storage does not change in between transactions to `B`.

```
pragma solidity >=0.8.0;

contract A {
    uint public x;
    address immutable public owner;
    constructor() {
        owner = msg.sender;
    }
    function setX(uint _x) public {
        require(msg.sender == owner);
        x = _x;
    }
}

contract B {
    A a;
    constructor() {
        a = new A();
        assert(a.x() == 0); // (1) should hold
    }
    function g() public view {
        assert(a.owner() == address(this)); // (2) should hold
        assert(a.x() == 0); // (3) should hold, but fails due to a false␣
↪positive
    }
}
```

### Reported Inferred Inductive Invariants

For properties that were proved safe with the CHC engine, the SMTChecker can retrieve inductive invariants that were inferred by the Horn solver as part of the proof. Currently only two types of invariants can be reported to the user:

- Contract Invariants: these are properties over the contract's state variables that are true before and after every possible transaction that the contract may ever run. For example, `x >= y`, where `x` and `y` are a contract's state variables.
- Reentrancy Properties: they represent the behavior of the contract in the presence of external calls to unknown code. These properties can express a relation between the value of the state variables before and after the external call, where the external call is free to do anything, including making reentrant calls to the analyzed contract. Primed variables represent the state variables" values after said external call. Example: `lock -> x = x'`.

The user can choose the type of invariants to be reported using the CLI option `--model-checker-invariants "contract,reentrancy"` or as an array in the field `settings.modelChecker.invariants` in the *JSON input*. By default the SMTChecker does not report invariants.

### Division and Modulo With Slack Variables

Spacer, the default Horn solver used by the SMTChecker, often dislikes division and modulo operations inside Horn rules. Because of that, by default the Solidity division and modulo operations are encoded using the constraint `a = b * d + m` where `d = a / b` and `m = a% b`. However, other solvers, such as Eldarica, prefer the syntactically precise operations. The command line flag `--model-checker-div-mod-no-slacks` and the JSON option `settings.modelChecker.divModNoSlacks` can be used to toggle the encoding depending on the used solver preferences.

### Natspec Function Abstraction

Certain functions including common math methods such as `pow` and `sqrt` may be too complex to be analyzed in a fully automated way. These functions can be annotated with Natspec tags that indicate to the SMTChecker that these functions should be abstracted. This means that the body of the function is not used, and when called, the function will:

- Return a nondeterministic value, and either keep the state variables unchanged if the abstracted function is view/pure, or also set the state variables to nondeterministic values otherwise. This can be used via the annotation `/// @custom:smtchecker abstract-function-nondet`.
- Act as an uninterpreted function. This means that the semantics of the function (given by the body) are ignored, and the only property this function has is that given the same input it guarantees the same output. This is currently under development and will be available via the annotation `/// @custom:smtchecker abstract-function-uf`.

### Model Checking Engines

The SMTChecker module implements two different reasoning engines, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have different characteristics. The engines are independent and every property warning states from which engine it came. Note that all the examples above with counterexamples were reported by CHC, the more powerful engine.

By default both engines are used, where CHC runs first, and every property that was not proven is passed over to BMC. You can choose a specific engine via the CLI option `--model-checker-engine {all,bmc,chc,none}` or the JSON option `settings.modelChecker.engine={all,bmc,chc,none}`.

### Bounded Model Checker (BMC)

The BMC engine analyzes functions in isolation, that is, it does not take the overall behavior of the contract over multiple transactions into account when analyzing each function. Loops are also ignored in this engine at the moment. Internal function calls are inlined as long as they are not recursive, directly or indirectly. External function calls are inlined if possible. Knowledge that is potentially affected by reentrancy is erased.

The characteristics above make BMC prone to reporting false positives, but it is also lightweight and should be able to quickly find small local bugs.

### Constrained Horn Clauses (CHC)

A contract's Control Flow Graph (CFG) is modelled as a system of Horn clauses, where the life cycle of the contract is represented by a loop that can visit every public/external function non-deterministically. This way, the behavior of the entire contract over an unbounded number of transactions is taken into account when analyzing any function. Loops are fully supported by this engine. Internal function calls are supported, and external function calls assume the called code is unknown and can do anything.

The CHC engine is much more powerful than BMC in terms of what it can prove, and might require more computing resources.

### SMT and Horn solvers

The two engines detailed above use automated theorem provers as their logical backends. BMC uses an SMT solver, whereas CHC uses a Horn solver. Often the same tool can act as both, as seen in z3, which is primarily an SMT solver and makes Spacer available as a Horn solver, and Eldarica which does both.

The user can choose which solvers should be used, if available, via the CLI option `--model-checker-solvers {all,cvc4,eld,smtlib2,z3}` or the JSON option `settings.modelChecker.solvers=[smtlib2,z3]`, where:

- `cvc4` is only available if the `solc` binary is compiled with it. Only BMC uses `cvc4`.
- `eld` is used via its binary which must be installed in the system. Only CHC uses `eld`, and only if `z3` is not enabled.
- `smtlib2` outputs SMT/Horn queries in the smtlib2 format. These can be used together with the compiler's callback mechanism so that any solver binary from the system can be employed to synchronously return the results of the queries to the compiler. This can be used by both BMC and CHC depending on which solvers are called.
- `z3` is available
  - if `solc` is compiled with it;
  - if a dynamic `z3` library of version >=4.8.x is installed in a Linux system (from Solidity 0.7.6);
  - statically in `soljson.js` (from Solidity 0.6.9), that is, the Javascript binary of the compiler.

---

**Nota:** z3 version 4.8.16 broke ABI compatibility with previous versions and cannot be used with solc <=0.8.13. If you are using z3 >=4.8.16 please use solc >=0.8.14, and conversely, only use older z3 with older solc releases. We also recommend using the latest z3 release which is what SMTChecker also does.

---

Since both BMC and CHC use `z3`, and `z3` is available in a greater variety of environments, including in the browser, most users will almost never need to be concerned about this option. More advanced users might apply this option to try alternative solvers on more complex problems.

Please note that certain combinations of chosen engine and solver will lead to the SMTChecker doing nothing, for example choosing CHC and `cvc4`.

### 3.30.3 Abstraction and False Positives

The SMTChecker implements abstractions in an incomplete and sound way: If a bug is reported, it might be a false positive introduced by abstractions (due to erasing knowledge or using a non-precise type). If it determines that a verification target is safe, it is indeed safe, that is, there are no false negatives (unless there is a bug in the SMTChecker).

If a target cannot be proven you can try to help the solver by using the tuning options in the previous section. If you are sure of a false positive, adding `require` statements in the code with more information may also give some more power to the solver.

#### SMT Encoding and Types

The SMTChecker encoding tries to be as precise as possible, mapping Solidity types and expressions to their closest SMT-LIB representation, as shown in the table below.

| Solidity type | SMT sort | Theories |
|---|---|---|
| Boolean | Bool | Bool |
| intN, uintN, address, bytesN, enum, contract | Integer | LIA, NIA |
| array, mapping, bytes, string | Tuple (Array elements, Integer length) | Datatypes, Arrays, LIA |
| struct | Tuple | Datatypes |
| other types | Integer | LIA |

Types that are not yet supported are abstracted by a single 256-bit unsigned integer, where their unsupported operations are ignored.

For more details on how the SMT encoding works internally, see the paper SMT-based Verification of Solidity Smart Contracts.

#### Function Calls

In the BMC engine, function calls to the same contract (or base contracts) are inlined when possible, that is, when their implementation is available. Calls to functions in other contracts are not inlined even if their code is available, since we cannot guarantee that the actual deployed code is the same.

The CHC engine creates nonlinear Horn clauses that use summaries of the called functions to support internal function calls. External function calls are treated as calls to unknown code, including potential reentrant calls.

Complex pure functions are abstracted by an uninterpreted function (UF) over the arguments.

| Functions | BMC/CHC behavior |
|---|---|
| `assert` | Verification target. |
| `require` | Assumption. |
| internal call | BMC: Inline function call. CHC: Function summaries. |
| external call to known code | BMC: Inline function call or erase knowledge about state variables and local storage references. CHC: Assume called code is unknown. Try to infer invariants that hold after the call returns. |
| Storage array push/pop | Supported precisely. Checks whether it is popping an empty array. |
| ABI functions | Abstracted with UF. |
| `addmod`, `mulmod` | Supported precisely. |
| `gasleft`, `blockhash`, `keccak256`, `ecrecover` `ripemd160` | Abstracted with UF. |
| pure functions without implementation (external or complex) | Abstracted with UF |
| external functions without implementation | BMC: Erase state knowledge and assume result is nondeterminisc. CHC: Nondeterministic summary. Try to infer invariants that hold after the call returns. |
| transfer | BMC: Checks whether the contract's balance is sufficient. CHC: does not yet perform the check. |
| others | Currently unsupported |

Using abstraction means loss of precise knowledge, but in many cases it does not mean loss of proving power.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Recover
{
    function f(
        bytes32 hash,
        uint8 v1, uint8 v2,
        bytes32 r1, bytes32 r2,
        bytes32 s1, bytes32 s2
    ) public pure returns (address) {
        address a1 = ecrecover(hash, v1, r1, s1);
        require(v1 == v2);
        require(r1 == r2);
        require(s1 == s2);
        address a2 = ecrecover(hash, v2, r2, s2);
        assert(a1 == a2);
        return a1;
    }
}
```

In the example above, the SMTChecker is not expressive enough to actually compute `ecrecover`, but by modelling the function calls as uninterpreted functions we know that the return value is the same when called on equivalent parameters. This is enough to prove that the assertion above is always true.

Abstracting a function call with an UF can be done for functions known to be deterministic, and can be easily done for pure functions. It is however difficult to do this with general external functions, since they might depend on state variables.

### Reference Types and Aliasing

Solidity implements aliasing for reference types with the same *data location*. That means one variable may be modified through a reference to the same data area. The SMTChecker does not keep track of which references refer to the same data. This implies that whenever a local reference or state variable of reference type is assigned, all knowledge regarding variables of the same type and data location is erased. If the type is nested, the knowledge removal also includes all the prefix base types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Aliasing
{
    uint[] array1;
    uint[][] array2;
    function f(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint[] storage d
    ) internal {
        array1[0] = 42;
        a[0] = 2;
        c[0][0] = 2;
        b[0] = 1;
        // Erasing knowledge about memory references should not
        // erase knowledge about state variables.
        assert(array1[0] == 42);
        // However, an assignment to a storage reference will erase
        // storage knowledge accordingly.
        d[0] = 2;
        // Fails as false positive because of the assignment above.
        assert(array1[0] == 42);
        // Fails because `a == b` is possible.
        assert(a[0] == 2);
        // Fails because `c[i] == b` is possible.
        assert(c[0][0] == 2);
        assert(d[0] == 2);
        assert(b[0] == 1);
    }
    function g(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint x
    ) public {
        f(a, b, c, array2[x]);
    }
}
```

After the assignment to `b[0]`, we need to clear knowledge about `a` since it has the same type (`uint[]`) and data location (memory). We also need to clear knowledge about `c`, since its base type is also a `uint[]` located in memory. This implies that some `c[i]` could refer to the same data as `b` or `a`.

Notice that we do not clear knowledge about `array` and `d` because they are located in storage, even though they also

have type `uint[]`. However, if `d` was assigned, we would need to clear knowledge about `array` and vice-versa.

#### Contract Balance

A contract may be deployed with funds sent to it, if `msg.value` > 0 in the deployment transaction. However, the contract's address may already have funds before deployment, which are kept by the contract. Therefore, the SMTChecker assumes that `address(this).balance >= msg.value` in the constructor in order to be consistent with the EVM rules. The contract's balance may also increase without triggering any calls to the contract, if

- `selfdestruct` is executed by another contract with the analyzed contract as the target of the remaining funds,
- the contract is the coinbase (i.e., `block.coinbase`) of some block.

To model this properly, the SMTChecker assumes that at every new transaction the contract's balance may grow by at least `msg.value`.

### 3.30.4 Real World Assumptions

Some scenarios can be expressed in Solidity and the EVM, but are expected to never occur in practice. One of such cases is the length of a dynamic storage array overflowing during a push: If the `push` operation is applied to an array of length 2^256 - 1, its length silently overflows. However, this is unlikely to happen in practice, since the operations required to grow the array to that point would take billions of years to execute. Another similar assumption taken by the SMTChecker is that an address" balance can never overflow.

A similar idea was presented in EIP-1985.

## 3.31 Recursos

### 3.31.1 Recursos Generales

- Portal del Desarrollador Ethereum.org
- Ethereum StackExchange
- Portal de Solidity
- Solidity Changelog
- Código fuente en GitHub de Solidity
- Chat de usuarios del leguaje Solidity
- Chat de desarrolladores del compilador Solidity
- Awesome Solidity
- Solidity by Example
- Traducciones de la documentación por la comunidad de Solidity

### 3.31.2 Ambientes de Desarrollo Integrado (Ethereum)

- **Brownie**
  Framework de desarrollo y testing basado en Python para contratos inteligentes dirigido a la Máquina Virtual de Ethereum.
- **Dapp**
  Herramienta para la construcción, testeo y deployment de contratos inteligentes desde la línea de comandos.
- **Embark**
  Plataforma para la construcción y deployment de aplicaciones descentralizadas.
- **Foundry**
  Juego de herramientas rápido, portable y modular para el desarrollo de aplicaciones de Ethereum escritas en Rust.
- **Hardhat**
  Ambiente de desarrollo de Ethereum con red local Ethereum, características para debugging y ecosistema de programas adicionales.
- **Remix**
  IDE basado en el navegador con compilador integrado y ambiente de ejecución de Solidity sin componentes del lado del servidor.
- **Truffle**
  Framework para el desarollo en Ethereum.

### 3.31.3 Integraciones de Editores

- Emacs
  - **Emacs Solidity**
    Plugin para el editor Emacs que provee resaltado de sintaxis y reporte de errores de compilación.
- IntelliJ
  - **IntelliJ IDEA plugin**
    Plugin de Solidity para IntelliJ IDEA (y todos los otros JetBrains IDEs)
- Sublime Text
  - **Package for SublimeText - Solidity language syntax**
    Resaltado de sintaxis de Solidity para el editor de SublimeText.
- Vim
  - Vim Solidity by Thesis

    Resaltado de sintaxis para Solidity en Vim.
  - **Vim Solidity by TovarishFin**
    Sintaxis Vim para Solidity.
  - **Vim Syntastic**
    Plugin para el editor de Vim que provee comprobación de compilación.
- Visual Studio Code (VS Code)
  - **Ethereum Remix Visual Studio Code extension**
    Paquete de extensión de Ethereum Remix para VS Code.

- **Solidity Visual Studio Code extension, by Juan Blanco**
  Plugin de Solidity para Microsoft Visual Studio Code que incluye resaltado de sintaxis y el compilador de Solidity.
- **Solidity Visual Studio Code extension, by Nomic Foundation**
  Soporte para Solidity y Hardhat por el equipo de Hardhat, incluye: resaltado de sintaxis, salto a definición, renombrar, modificaciones rápidas y errores y advertencias en línea.
- **Solidity Visual Auditor extension**
  Agrega sintaxis céntrica de seguridad y resaltado semántico para Visual Studio Code.
- **Truffle for VS Code**
  Desarrollo, debug y deploy de contratos inteligentes en Ethereum y cadenas de bloques compatibles con EVM.

### 3.31.4 Herramientas para Solidity

- **ABI to Solidity interface converter**
  Un script para generar interfaces de contratos desde el ABI de un contrato inteligente.
- **abi-to-sol**
  Herramienta para generar la fuente de interfaz de Solidity desde un ABI JSON dado.
- **Doxity**
  Generador de documentación para Solidity.
- **Ethlint**
  Linter para identificar y corregir asuntos de estilo y seguridad en Solidity.
- **evmdis**
  Desensamblador EVM que lleva a cabo análisis estático en el bytecode para proveer un nivel alto de abstracción que las operaciones EVM sin procesar.
- **EVM Lab**
  Paquete abundante de herramientas para interactuar con el EVM. Incluye un VM, Etherchain API, y un mostrador con muestra de costo de gas.
- **hevm**
  Debugger EVM y motor de ejecución simbólica.
- **leafleth**
  Un generador de documentación para contratos inteligentes de Solidity.
- **PIET**
  Una herramienta para desarrollar, auditar y usar contratos inteligentes de Solidity a través de una interfaz gráfica simple.
- **Scaffold-ETH**
  Stack de desarollo en Ethereum forkeable enfocado en iteraciones rápidas de productos.
- **sol2uml**
  Generador de diagrama de clase Unified Modeling Language (UML) para contratos en Solidity.
- **solc-select**
  Un script para cambiar rápidamente entre versiones de compilador de Solidity.
- **Solidity prettier plugin**
  Un plugin prettier para solidity.
- **Solidity REPL**
  Prueba instantáneamente Solidity con una consola de línea de comandos para Solidity.

- **solgraph**
  Visualice el flujo de control de Solidity y destaque las vulnerabilidades potenciales de seguridad.
- **Solhint**
  Linter para Solidity que provee seguridad, guía de estilo y reglas de buenas prácticas para validación de contratos inteligentes.
- **Sourcify**
  Servicio de verificación de contratos automatizados descentralizados y repositorio pñublico de metadatos de contratos.
- **Sūrya**
  Herramienta de utilidad para sistemas de contratos inteligentes que ofrece varias salidas visuales e información sobre la estructura de los contratos. También soporta búsqueda en el gráfico de la llamada a función.
- **Universal Mutator**
  Una herramienta para generación de mutaciones, con reglas configurables y soporte para Solidity y Vyper.

### 3.31.5 Parsers y Grammars de Terceros

- **Solidity Parser for JavaScript**
  Un parser de Solidity para JS construido sobre una robusta gramática ANTLR4.

## 3.32 Import Path Resolution

In order to be able to support reproducible builds on all platforms, the Solidity compiler has to abstract away the details of the filesystem where source files are stored. Paths used in imports must work the same way everywhere while the command-line interface must be able to work with platform-specific paths to provide good user experience. This section aims to explain in detail how Solidity reconciles these requirements.

### 3.32.1 Virtual Filesystem

The compiler maintains an internal database (*virtual filesystem* or *VFS* for short) where each source unit is assigned a unique *source unit name* which is an opaque and unstructured identifier. When you use the *import statement*, you specify an *import path* that references a source unit name.

#### Import Callback

The VFS is initially populated only with files the compiler has received as input. Additional files can be loaded during compilation using an *import callback*, which is different depending on the type of compiler you use (see below). If the compiler does not find any source unit name matching the import path in the VFS, it invokes the callback, which is responsible for obtaining the source code to be placed under that name. An import callback is free to interpret source unit names in an arbitrary way, not just as paths. If there is no callback available when one is needed or if it fails to locate the source code, compilation fails.

The command-line compiler provides the *Host Filesystem Loader* - a rudimentary callback that interprets a source unit name as a path in the local filesystem. The JavaScript interface does not provide any by default, but one can be provided by the user. This mechanism can be used to obtain source code from locations other then the local filesystem (which may not even be accessible, e.g. when the compiler is running in a browser). For example the Remix IDE provides a versatile callback that lets you import files from HTTP, IPFS and Swarm URLs or refer directly to packages in NPM registry.

**Nota:** Host Filesystem Loader's file lookup is platform-dependent. For example backslashes in a source unit name can be interpreted as directory separators or not and the lookup can be case-sensitive or not, depending on the underlying platform.

For portability it is recommended to avoid using import paths that will work correctly only with a specific import callback or only on one platform. For example you should always use forward slashes since they work as path separators also on platforms that support backslashes.

### Initial Content of the Virtual Filesystem

The initial content of the VFS depends on how you invoke the compiler:

1. **solc / command-line interface**

   When you compile a file using the command-line interface of the compiler, you provide one or more paths to files containing Solidity code:

   ```
   solc contract.sol /usr/local/dapp-bin/token.sol
   ```

   The source unit name of a file loaded this way is constructed by converting its path to a canonical form and, if possible, making it relative to either the base path or one of the include paths. See *CLI Path Normalization and Stripping* for a detailed description of this process.

2. **Standard JSON**

   When using the *Standard JSON* API (via either the JavaScript interface or the `--standard-json` command-line option) you provide input in JSON format, containing, among other things, the content of all your source files:

   ```
   {
       "language": "Solidity",
       "sources": {
           "contract.sol": {
               "content": "import \"./util.sol\";\ncontract C {}"
           },
           "util.sol": {
               "content": "library Util {}"
           },
           "/usr/local/dapp-bin/token.sol": {
               "content": "contract Token {}"
           }
       },
       "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
   }
   ```

   The `sources` dictionary becomes the initial content of the virtual filesystem and its keys are used as source unit names.

3. **Standard JSON (via import callback)**

   With Standard JSON it is also possible to tell the compiler to use the import callback to obtain the source code:

   ```
   {
       "language": "Solidity",
       "sources": {
           "/usr/local/dapp-bin/token.sol": {
   ```

(continué en la próxima página)

(proviene de la página anterior)

```
            "urls": [
                "/projects/mytoken.sol",
                "https://example.com/projects/mytoken.sol"
            ]
        }
    },
    "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
}
```

If an import callback is available, the compiler will give it the strings specified in `urls` one by one, until one is loaded successfully or the end of the list is reached.

The source unit names are determined the same way as when using `content` - they are keys of the `sources` dictionary and the content of `urls` does not affect them in any way.

4. **Standard input**

   On the command line it is also possible to provide the source by sending it to compiler's standard input:

```
echo 'import "./util.sol"; contract C {}' | solc -
```

   `-` used as one of the arguments instructs the compiler to place the content of the standard input in the virtual filesystem under a special source unit name: `<stdin>`.

Once the VFS is initialized, additional files can still be added to it only through the import callback.

### 3.32.2 Imports

The import statement specifies an *import path*. Based on how the import path is specified, we can divide imports into two categories:

- *Direct imports*, where you specify the full source unit name directly.
- *Relative imports*, where you specify a path starting with `./` or `../` to be combined with the source unit name of the importing file.

Lista 1: contracts/contract.sol

```
import "./math/math.sol";
import "contracts/tokens/token.sol";
```

In the above `./math/math.sol` and `contracts/tokens/token.sol` are import paths while the source unit names they translate to are `contracts/math/math.sol` and `contracts/tokens/token.sol` respectively.

#### Direct Imports

An import that does not start with `./` or `../` is a *direct import*.

```
import "/project/lib/util.sol";         // source unit name: /project/lib/util.sol
import "lib/util.sol";                  // source unit name: lib/util.sol
import "@openzeppelin/address.sol";     // source unit name: @openzeppelin/address.sol
import "https://example.com/token.sol"; // source unit name: https://example.com/token.
↪sol
```

After applying any *import remappings* the import path simply becomes the source unit name.

---

**Nota:** A source unit name is just an identifier and even if its value happens to look like a path, it is not subject to the normalization rules you would typically expect in a shell. Any `/./` or `/../` segments or sequences of multiple slashes remain a part of it. When the source is provided via Standard JSON interface it is entirely possible to associate different content with source unit names that would refer to the same file on disk.

---

When the source is not available in the virtual filesystem, the compiler passes the source unit name to the import callback. The Host Filesystem Loader will attempt to use it as a path and look up the file on disk. At this point the platform-specific normalization rules kick in and names that were considered different in the VFS may actually result in the same file being loaded. For example `/project/lib/math.sol` and `/project/lib/../lib///math.sol` are considered completely different in the VFS even though they refer to the same file on disk.

---

**Nota:** Even if an import callback ends up loading source code for two different source unit names from the same file on disk, the compiler will still see them as separate source units. It is the source unit name that matters, not the physical location of the code.

---

### Relative Imports

An import starting with `./` or `../` is a *relative import*. Such imports specify a path relative to the source unit name of the importing source unit:

Lista 2: /project/lib/math.sol

```
import "./util.sol" as util;    // source unit name: /project/lib/util.sol
import "../token.sol" as token; // source unit name: /project/token.sol
```

Lista 3: lib/math.sol

```
import "./util.sol" as util;    // source unit name: lib/util.sol
import "../token.sol" as token; // source unit name: token.sol
```

---

**Nota:** Relative imports **always** start with `./` or `../` so `import "util.sol"`, unlike `import "./util.sol"`, is a direct import. While both paths would be considered relative in the host filesystem, `util.sol` is actually absolute in the VFS.

---

Let us define a *path segment* as any non-empty part of the path that does not contain a separator and is bounded by two path separators. A separator is a forward slash or the beginning/end of the string. For example in `./abc/..//` there are three path segments: `.`, `abc` and `..`.

The compiler resolves the import into a source unit name based on the import path, in the following way:

1. We start with the source unit name of the importing source unit.
2. The last path segment with preceding slashes is removed from the resolved name.
3. **Then, for every segment in the import path, starting from the leftmost one:**
    - If the segment is `.`, it is skipped.
    - If the segment is `..`, the last path segment with preceding slashes is removed from the resolved name.

- Otherwise, the segment (preceded by a single slash if the resolved name is not empty), is appended to the resolved name.

The removal of the last path segment with preceding slashes is understood to work as follows:

1. Everything past the last slash is removed (i.e. `a/b//c.sol` becomes `a/b//`).
2. All trailing slashes are removed (i.e. `a/b//` becomes `a/b`).

Note that the process normalizes the part of the resolved source unit name that comes from the import path according to the usual rules for UNIX paths, i.e. all `.` and `..` are removed and multiple slashes are squashed into a single one. On the other hand, the part that comes from the source unit name of the importing module remains unnormalized. This ensures that the `protocol://` part does not turn into `protocol:/` if the importing file is identified with a URL.

If your import paths are already normalized, you can expect the above algorithm to produce very intuitive results. Here are some examples of what you can expect if they are not:

Lista 4: lib/src/../contract.sol

```
import "./util/./util.sol";         // source unit name: lib/src/../util/util.sol
import "./util//util.sol";          // source unit name: lib/src/../util/util.sol
import "../util/../array/util.sol"; // source unit name: lib/src/array/util.sol
import "../../../../util.sol";      // source unit name: util.sol
import "../../../../../util.sol";   // source unit name: util.sol
```

**Nota:** The use of relative imports containing leading `..` segments is not recommended. The same effect can be achieved in a more reliable way by using direct imports with *base path and include paths*.

### 3.32.3 Base Path and Include Paths

The base path and include paths represent directories that the Host Filesystem Loader will load files from. When a source unit name is passed to the loader, it prepends the base path to it and performs a filesystem lookup. If the lookup does not succeed, the same is done with all directories on the include path list.

It is recommended to set the base path to the root directory of your project and use include paths to specify additional locations that may contain libraries your project depends on. This lets you import from these libraries in a uniform way, no matter where they are located in the filesystem relative to your project. For example, if you use npm to install packages and your contract imports `@openzeppelin/contracts/utils/Strings.sol`, you can use these options to tell the compiler that the library can be found in one of the npm package directories:

```
solc contract.sol \
    --base-path . \
    --include-path node_modules/ \
    --include-path /usr/local/lib/node_modules/
```

Your contract will compile (with the same exact metadata) no matter whether you install the library in the local or global package directory or even directly under your project root.

By default the base path is empty, which leaves the source unit name unchanged. When the source unit name is a relative path, this results in the file being looked up in the directory the compiler has been invoked from. It is also the only value that results in absolute paths in source unit names being actually interpreted as absolute paths on disk. If the base path itself is relative, it is interpreted as relative to the current working directory of the compiler.

**Nota:** Include paths cannot have empty values and must be used together with a non-empty base path.

**Nota:** Include paths and base path can overlap as long as it does not make import resolution ambiguous. For example, you can specify a directory inside base path as an include directory or have an include directory that is a subdirectory of another include directory. The compiler will only issue an error if the source unit name passed to the Host Filesystem Loader represents an existing path when combined with multiple include paths or an include path and base path.

### CLI Path Normalization and Stripping

On the command line the compiler behaves just as you would expect from any other program: it accepts paths in a format native to the platform and relative paths are relative to the current working directory. The source unit names assigned to files whose paths are specified on the command line, however, should not change just because the project is being compiled on a different platform or because the compiler happens to have been invoked from a different directory. To achieve this, paths to source files coming from the command line must be converted to a canonical form, and, if possible, made relative to the base path or one of the include paths.

The normalization rules are as follows:

- If a path is relative, it is made absolute by prepending the current working directory to it.
- Internal `.` and `..` segments are collapsed.
- Platform-specific path separators are replaced with forward slashes.
- Sequences of multiple consecutive path separators are squashed into a single separator (unless they are the leading slashes of an UNC path).
- If the path includes a root name (e.g. a drive letter on Windows) and the root is the same as the root of the current working directory, the root is replaced with `/`.
- Symbolic links in the path are **not** resolved.
    - The only exception is the path to the current working directory prepended to relative paths in the process of making them absolute. On some platforms the working directory is reported always with symbolic links resolved so for consistency the compiler resolves them everywhere.
- The original case of the path is preserved even if the filesystem is case-insensitive but case-preserving and the actual case on disk is different.

**Nota:** There are situations where paths cannot be made platform-independent. For example on Windows the compiler can avoid using drive letters by referring to the root directory of the current drive as `/` but drive letters are still necessary for paths leading to other drives. You can avoid such situations by ensuring that all the files are available within a single directory tree on the same drive.

After normalization the compiler attempts to make the source file path relative. It tries the base path first and then the include paths in the order they were given. If the base path is empty or not specified, it is treated as if it was equal to the path to the current working directory (with all symbolic links resolved). The result is accepted only if the normalized directory path is the exact prefix of the normalized file path. Otherwise the file path remains absolute. This makes the conversion unambiguous and ensures that the relative path does not start with `../`. The resulting file path becomes the source unit name.

**Nota:** The relative path produced by stripping must remain unique within the base path and include paths. For example the compiler will issue an error for the following command if both `/project/contract.sol` and `/lib/contract.sol` exist:

```
solc /project/contract.sol --base-path /project --include-path /lib
```

**Nota:** Prior to version 0.8.8, CLI path stripping was not performed and the only normalization applied was the conversion of path separators. When working with older versions of the compiler it is recommended to invoke the compiler from the base path and to only use relative paths on the command line.

### 3.32.4 Allowed Paths

As a security measure, the Host Filesystem Loader will refuse to load files from outside of a few locations that are considered safe by default:

- Outside of Standard JSON mode:
  - The directories containing input files listed on the command line.
  - The directories used as *remapping* targets. If the target is not a directory (i.e does not end with `/`, `/.` or `/..`) the directory containing the target is used instead.
  - Base path and include paths.
- In Standard JSON mode:
  - Base path and include paths.

Additional directories can be whitelisted using the `--allow-paths` option. The option accepts a comma-separated list of paths:

```
cd /home/user/project/
solc token/contract.sol \
    lib/util.sol=libs/util.sol \
    --base-path=token/ \
    --include-path=/lib/ \
    --allow-paths=../utils/,/tmp/libraries
```

When the compiler is invoked with the command shown above, the Host Filesystem Loader will allow importing files from the following directories:

- `/home/user/project/token/` (because `token/` contains the input file and also because it is the base path),
- `/lib/` (because `/lib/` is one of the include paths),
- `/home/user/project/libs/` (because `libs/` is a directory containing a remapping target),
- `/home/user/utils/` (because of `../utils/` passed to `--allow-paths`),
- `/tmp/libraries/` (because of `/tmp/libraries` passed to `--allow-paths`),

**Nota:** The working directory of the compiler is one of the paths allowed by default only if it happens to be the base path (or the base path is not specified or has an empty value).

**Nota:** The compiler does not check if allowed paths actually exist and whether they are directories. Non-existent or empty paths are simply ignored. If an allowed path matches a file rather than a directory, the file is considered whitelisted, too.

**Nota:** Allowed paths are case-sensitive even if the filesystem is not. The case must exactly match the one used in your imports. For example `--allow-paths tokens` will not match `import "Tokens/IERC20.sol"`.

**Advertencia:** Files and directories only reachable through symbolic links from allowed directories are not automatically whitelisted. For example if `token/contract.sol` in the example above was actually a symlink pointing at `/etc/passwd` the compiler would refuse to load it unless `/etc/` was one of the allowed paths too.

### 3.32.5 Import Remapping

Import remapping allows you to redirect imports to a different location in the virtual filesystem. The mechanism works by changing the translation between import paths and source unit names. For example you can set up a remapping so that any import from the virtual directory `github.com/ethereum/dapp-bin/library/` would be seen as an import from `dapp-bin/library/` instead.

You can limit the scope of a remapping by specifying a *context*. This allows creating remappings that apply only to imports located in a specific library or a specific file. Without a context a remapping is applied to every matching import in all the files in the virtual filesystem.

Import remappings have the form of `context:prefix=target`:

- `context` must match the beginning of the source unit name of the file containing the import.
- `prefix` must match the beginning of the source unit name resulting from the import.
- `target` is the value the prefix is replaced with.

For example, if you clone https://github.com/ethereum/dapp-bin/ locally to `/project/dapp-bin` and run the compiler with:

```
solc github.com/ethereum/dapp-bin/=dapp-bin/ --base-path /project source.sol
```

you can use the following in your source file:

```
import "github.com/ethereum/dapp-bin/library/math.sol"; // source unit name: dapp-bin/
↪library/math.sol
```

The compiler will look for the file in the VFS under `dapp-bin/library/math.sol`. If the file is not available there, the source unit name will be passed to the Host Filesystem Loader, which will then look in `/project/dapp-bin/library/iterable_mapping.sol`.

**Advertencia:** Information about remappings is stored in contract metadata. Since the binary produced by the compiler has a hash of the metadata embedded in it, any modification to the remappings will result in different bytecode.

For this reason you should be careful not to include any local information in remapping targets. For example if your library is located in `/home/user/packages/mymath/math.sol`, a remapping like `@math/=/home/user/packages/mymath/` would result in your home directory being included in the metadata. To be able to reproduce the same bytecode with such a remapping on a different machine, you would need to recreate parts of your local directory structure in the VFS and (if you rely on Host Filesystem Loader) also in the host filesystem.

To avoid having your local directory structure embedded in the metadata, it is recommended to designate the directories containing libraries as *include paths* instead. For example, in the example above `--include-path /home/`

`user/packages/` would let you use imports starting with `mymath/`. Unlike remapping, the option on its own will not make `mymath` appear as `@math` but this can be achieved by creating a symbolic link or renaming the package subdirectory.

As a more complex example, suppose you rely on a module that uses an old version of dapp-bin that you checked out to `/project/dapp-bin_old`, then you can run:

```
solc module1:github.com/ethereum/dapp-bin/=dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=dapp-bin_old/ \
     --base-path /project \
     source.sol
```

This means that all imports in `module2` point to the old version but imports in `module1` point to the new version.

Here are the detailed rules governing the behaviour of remappings:

1. **Remappings only affect the translation between import paths and source unit names.**

   Source unit names added to the VFS in any other way cannot be remapped. For example the paths you specify on the command-line and the ones in `sources.urls` in Standard JSON are not affected.

   ```
   solc /project/=/contracts/ /project/contract.sol # source unit name: /project/
   ↪contract.sol
   ```

   In the example above the compiler will load the source code from `/project/contract.sol` and place it under that exact source unit name in the VFS, not under `/contract/contract.sol`.

2. **Context and prefix must match source unit names, not import paths.**

   - This means that you cannot remap `./` or `../` directly since they are replaced during the translation to source unit name but you can remap the part of the name they are replaced with:

     ```
     solc ./=a/ /project/=b/ /project/contract.sol # source unit name: /project/
     ↪contract.sol
     ```

     Lista 5: /project/contract.sol

     ```
     import "./util.sol" as util; // source unit name: b/util.sol
     ```

   - You cannot remap base path or any other part of the path that is only added internally by an import callback:

     ```
     solc /project/=/contracts/ /project/contract.sol --base-path /project # source␣
     ↪unit name: contract.sol
     ```

     Lista 6: /project/contract.sol

     ```
     import "util.sol" as util; // source unit name: util.sol
     ```

3. **Target is inserted directly into the source unit name and does not necessarily have to be a valid path.**

   - It can be anything as long as the import callback can handle it. In case of the Host Filesystem Loader this includes also relative paths. When using the JavaScript interface you can even use URLs and abstract identifiers if your callback can handle them.
   - Remapping happens after relative imports have already been resolved into source unit names. This means that targets starting with `./` and `../` have no special meaning and are relative to the base path rather than to the location of the source file.

- Remapping targets are not normalized so `@root/=./a/b//` will remap `@root/contract.sol` to `./a/b//contract.sol` and not `a/b/contract.sol`.
- If the target does not end with a slash, the compiler will not add one automatically:

```
solc /project/=/contracts /project/contract.sol # source unit name: /project/
↪contract.sol
```

Lista 7: /project/contract.sol

```
import "/project/util.sol" as util; // source unit name: /contractsutil.sol
```

4. **Context and prefix are patterns and matches must be exact.**
   - `a//b=c` will not match `a/b`.
   - source unit names are not normalized so `a/b=c` will not match `a//b` either.
   - Parts of file and directory names can match as well. `/newProject/con:/new=old` will match `/newProject/contract.sol` and remap it to `oldProject/contract.sol`.
5. **At most one remapping is applied to a single import.**
   - If multiple remappings match the same source unit name, the one with the longest matching prefix is chosen.
   - If prefixes are identical, the one specified last wins.
   - Remappings do not work on other remappings. For example `a=b b=c c=d` will not result in `a` being remapped to `d`.
6. **Prefix cannot be empty but context and target are optional.**
   - If `target` is the empty string, `prefix` is simply removed from import paths.
   - Empty `context` means that the remapping applies to all imports in all source units.

### 3.32.6 Using URLs in imports

Most URL prefixes such as `https://` or `data://` have no special meaning in import paths. The only exception is `file://` which is stripped from source unit names by the Host Filesystem Loader.

When compiling locally you can use import remapping to replace the protocol and domain part with a local path:

```
solc :https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin contract.sol
```

Note the leading `:`, which is necessary when the remapping context is empty. Otherwise the `https:` part would be interpreted by the compiler as the context.

## 3.33 Yul

Yul (previously also called JULIA or IULIA) is an intermediate language that can be compiled to bytecode for different backends.

It can be used in stand-alone mode and for «inline assembly» inside Solidity. The compiler uses Yul as an intermediate language in the IR-based code generator («new codegen» or «IR-based codegen»). Yul is a good target for high-level optimisation stages that can benefit all target platforms equally.

### 3.33.1 Motivation and High-level Description

The design of Yul tries to achieve several goals:

1. Programs written in Yul should be readable, even if the code is generated by a compiler from Solidity or another high-level language.
2. Control flow should be easy to understand to help in manual inspection, formal verification and optimization.
3. The translation from Yul to bytecode should be as straightforward as possible.
4. Yul should be suitable for whole-program optimization.

In order to achieve the first and second goal, Yul provides high-level constructs like `for` loops, `if` and `switch` statements and function calls. These should be sufficient for adequately representing the control flow for assembly programs. Therefore, no explicit statements for `SWAP`, `DUP`, `JUMPDEST`, `JUMP` and `JUMPI` are provided, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form `mul(add(x, y), 7)` are preferred over pure opcode statements like `7 y x add mul` because in the first form, it is much easier to see which operand is used for which opcode.

Even though it was designed for stack machines, Yul does not expose the complexity of the stack itself. The programmer or auditor should not have to worry about the stack.

The third goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, ...) and cleanup of local variables from the stack.

To avoid confusions between concepts like values and references, Yul is statically typed. At the same time, there is a default type (usually the integer word of the target machine) that can always be omitted to help readability.

To keep the language simple and flexible, Yul does not have any built-in operations, functions or types in its pure form. These are added together with their semantics when specifying a dialect of Yul, which allows specializing Yul to the requirements of different target platforms and feature sets.

Currently, there is only one specified dialect of Yul. This dialect uses the EVM opcodes as builtin functions (see below) and defines only the type `u256`, which is the native 256-bit type of the EVM. Because of that, we will not provide types in the examples below.

### 3.33.2 Simple Example

The following example program is written in the EVM dialect and computes exponentiation. It can be compiled using `solc --strict-assembly`. The builtin functions `mul` and `div` compute product and division, respectively.

```
{
    function power(base, exponent) -> result
    {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default
        {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

It is also possible to implement the same function using a for-loop instead of with recursion. Here, `lt(a, b)` computes whether `a` is less than `b`.

```
{
    function power(base, exponent) -> result
    {
        result := 1
        for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
        {
            result := mul(result, base)
        }
    }
}
```

At the *end of the section*, a complete implementation of the ERC-20 standard can be found.

### 3.33.3 Stand-Alone Usage

You can use Yul in its stand-alone form in the EVM dialect using the Solidity compiler. This will use the *Yul object notation* so that it is possible to refer to code as data to deploy contracts. This Yul mode is available for the commandline compiler (use `--strict-assembly`) and for the *standard-json interface*:

```
{
    "language": "Yul",
    "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
    "settings": {
        "outputSelection": { "*": { "*": ["*"], "": [ "*" ] } },
        "optimizer": { "enabled": true, "details": { "yul": true } }
    }
}
```

**Advertencia:** Yul is in active development and bytecode generation is only fully implemented for the EVM dialect of Yul with EVM 1.0 as target.

### 3.33.4 Informal Description of Yul

In the following, we will talk about each individual aspect of the Yul language. In examples, we will use the default EVM dialect.

#### Syntax

Yul parses comments, literals and identifiers in the same way as Solidity, so you can e.g. use `//` and `/* */` to denote comments. There is one exception: Identifiers in Yul can contain dots: `.`.

Yul can specify «objects» that consist of code, data and sub-objects. Please see *Yul Objects* below for details on that. In this section, we are only concerned with the code part of such an object. This code part always consists of a curly-braces delimited block. Most tools support specifying just a code block where an object is expected.

Inside a code block, the following elements can be used (see the later sections for more details):

- literals, i.e. `0x123`, `42` or `"abc"` (strings up to 32 characters)

- calls to builtin functions, e.g. `add(1, mload(0))`
- variable declarations, e.g. `let x := 7`, `let x := add(y, 3)` or `let x` (initial value of 0 is assigned)
- identifiers (variables), e.g. `add(3, x)`
- assignments, e.g. `x := add(y, 3)`
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`
- if statements, e.g. `if lt(a, b) { sstore(0, 1) }`
- switch statements, e.g. `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- for loops, e.g. `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- function definitions, e.g. `function f(a, b) -> c { c := add(a, b) }`

Multiple syntactical elements can follow each other simply separated by whitespace, i.e. there is no terminating `;` or newline required.

### Literals

As literals, you can use:

- Integer constants in decimal or hexadecimal notation.
- ASCII strings (e.g. `"abc"`), which may contain hex escapes `\xNN` and Unicode escapes `\uNNNN` where `N` are hexadecimal digits.
- Hex strings (e.g. `hex"616263"`).

In the EVM dialect of Yul, literals represent 256-bit words as follows:

- Decimal or hexadecimal constants must be less than `2**256`. They represent the 256-bit word with that value as an unsigned integer in big endian encoding.
- An ASCII string is first viewed as a byte sequence, by viewing a non-escape ASCII character as a single byte whose value is the ASCII code, an escape `\xNN` as single byte with that value, and an escape `\uNNNN` as the UTF-8 sequence of bytes for that code point. The byte sequence must not exceed 32 bytes. The byte sequence is padded with zeros on the right to reach 32 bytes in length; in other words, the string is stored left-aligned. The padded byte sequence represents a 256-bit word whose most significant 8 bits are the ones from the first byte, i.e. the bytes are interpreted in big endian form.
- A hex string is first viewed as a byte sequence, by viewing each pair of contiguous hex digits as a byte. The byte sequence must not exceed 32 bytes (i.e. 64 hex digits), and is treated as above.

When compiling for the EVM, this will be translated into an appropriate `PUSHi` instruction. In the following example, `3` and `2` are added resulting in 5 and then the bitwise `and` with the string «abc» is computed. The final value is assigned to a local variable called `x`.

The 32-byte limit above does not apply to string literals passed to builtin functions that require literal arguments (e.g. `setimmutable` or `loadimmutable`). Those strings never end up in the generated bytecode.

```
let x := and("abc", add(3, 2))
```

Unless it is the default type, the type of a literal has to be specified after a colon:

```
// This will not compile (u32 and u256 type not implemented yet)
let x := and("abc":u32, add(3:u256, 2:u256))
```

### Function Calls

Both built-in and user-defined functions (see below) can be called in the same way as shown in the previous example. If the function returns a single value, it can be directly used inside an expression again. If it returns multiple values, they have to be assigned to local variables.

```
function f(x, y) -> a, b { /* ... */ }
mstore(0x80, add(mload(0x80), 3))
// Here, the user-defined function `f` returns two values.
let x, y := f(1, mload(0))
```

For built-in functions of the EVM, functional expressions can be directly translated to a stream of opcodes: You just read the expression from right to left to obtain the opcodes. In the case of the first line in the example, this is `PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE`.

For calls to user-defined functions, the arguments are also put on the stack from right to left and this is the order in which argument lists are evaluated. The return values, though, are expected on the stack from left to right, i.e. in this example, `y` is on top of the stack and `x` is below it.

### Variable Declarations

You can use the `let` keyword to declare variables. A variable is only visible inside the `{...}`-block it was defined in. When compiling to the EVM, a new stack slot is created that is reserved for the variable and automatically removed again when the end of the block is reached. You can provide an initial value for the variable. If you do not provide a value, the variable will be initialized to zero.

Since variables are stored on the stack, they do not directly influence memory or storage, but they can be used as pointers to memory or storage locations in the built-in functions `mstore`, `mload`, `sstore` and `sload`. Future dialects might introduce specific types for such pointers.

When a variable is referenced, its current value is copied. For the EVM, this translates to a `DUP` instruction.

```
{
    let zero := 0
    let v := calldataload(zero)
    {
        let y := add(sload(v), 1)
        v := y
    } // y is "deallocated" here
    sstore(v, zero)
} // v and zero are "deallocated" here
```

If the declared variable should have a type different from the default type, you denote that following a colon. You can also declare multiple variables in one statement when you assign from a function call that returns multiple values.

```
// This will not compile (u32 and u256 type not implemented yet)
{
    let zero:u32 := 0:u32
    let v:u256, t:u32 := f()
    let x, y := g()
}
```

Depending on the optimiser settings, the compiler can free the stack slots already after the variable has been used for the last time, even though it is still in scope.

### Assignments

Variables can be assigned to after their definition using the `:=` operator. It is possible to assign multiple variables at the same time. For this, the number and types of the values have to match. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables. The same variable may not occur multiple times on the left-hand side of an assignment, e.g. `x, x := f()` is invalid.

```
let v := 0
// re-assign v
v := 2
let t := add(v, 2)
function f() -> a, b { }
// assign multiple values
v, t := f()
```

### If

The if statement can be used for conditionally executing code. No «else» block can be defined. Consider using «switch» instead (see below) if you need multiple alternatives.

```
if lt(calldatasize(), 4) { revert(0, 0) }
```

The curly braces for the body are required.

### Switch

You can use a switch statement as an extended version of the if statement. It takes the value of an expression and compares it to several literal constants. The branch corresponding to the matching constant is taken. Contrary to other programming languages, for safety reasons, control flow does not continue from one case to the next. There can be a fallback or default case called `default` which is taken if none of the literal constants matches.

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

The list of cases is not enclosed by curly braces, but the body of a case does require them.

### Loops

Yul supports for-loops which consist of a header containing an initializing part, a condition, a post-iteration part and a body. The condition has to be an expression, while the other three are blocks. If the initializing part declares any variables at the top level, the scope of these variables extends to all other parts of the loop.

The `break` and `continue` statements can be used in the body to exit the loop or skip to the post-part, respectively.

The following example computes the sum of an area in memory.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

For loops can also be used as a replacement for while loops: Simply leave the initialization and post-iteration parts empty.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {     // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

### Function Declarations

Yul allows the definition of functions. These should not be confused with functions in Solidity since they are never part of an external interface of a contract and are part of a namespace separate from the one for Solidity functions.

For the EVM, Yul functions take their arguments (and a return PC) from the stack and also put the results onto the stack. User-defined functions and built-in functions are called in exactly the same way.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function.

Functions declare parameters and return variables, similar to Solidity. To return a value, you assign it to the return variable(s).

If you call a function that returns multiple values, you have to assign them to multiple variables using `a, b := f(x)` or `let a, b := f(x)`.

The `leave` statement can be used to exit the current function. It works like the `return` statement in other languages just that it does not take a value to return, it just exits the functions and the function will return whatever values are currently assigned to the return variable(s).

Note that the EVM dialect has a built-in function called `return` that quits the full execution context (internal message call) and not just the current yul function.

The following example implements the power function by square-and-multiply.

```
{
    function power(base, exponent) -> result {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

### 3.33.5 Specification of Yul

This chapter describes Yul code formally. Yul code is usually placed inside Yul objects, which are explained in their own chapter.

```
Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':=' Expression )?
Assignment =
    IdentifierList ':=' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
Leave = 'leave'
FunctionCall =
```

(continué en la próxima página)

(proviene de la página anterior)

```
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9.]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )*
Literal =
    (NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?
NumberLiteral = HexNumber | DecimalNumber
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

### Restrictions on the Grammar

Apart from those directly imposed by the grammar, the following restrictions apply:

Switches must have at least one case (including the default case). All case values need to have the same type and distinct values. If all possible values of the expression type are covered, a default case is not allowed (i.e. a switch with a `bool` expression that has both a true and a false case do not allow a default case).

Every expression evaluates to zero or more values. Identifiers and Literals evaluate to exactly one value and function calls evaluate to a number of values equal to the number of return variables of the function called.

In variable declarations and assignments, the right-hand-side expression (if present) has to evaluate to a number of values equal to the number of variables on the left-hand-side. This is the only situation where an expression evaluating to more than one value is allowed. The same variable name cannot occur more than once in the left-hand-side of an assignment or variable declaration.

Expressions that are also statements (i.e. at the block level) have to evaluate to zero values.

In all other situations, expressions have to evaluate to exactly one value.

A `continue` or `break` statement can only be used inside the body of a for-loop, as follows. Consider the innermost loop that contains the statement. The loop and the statement must be in the same function, or both must be at the top level. The statement must be in the loop's body block; it cannot be in the loop's initialization block or update block. It is worth emphasizing that this restriction applies just to the innermost loop that contains the `continue` or `break` statement: this innermost loop, and therefore the `continue` or `break` statement, may appear anywhere in an outer loop, possibly in an outer loop's initialization block or update block. For example, the following is legal, because the `break` occurs in the body block of the inner loop, despite also occurring in the update block of the outer loop:

```
for {} true { for {} true {} { break } }
{
}
```

The condition part of the for-loop has to evaluate to exactly one value.

The `leave` statement can only be used inside a function.

Functions cannot be defined anywhere inside for loop init blocks.

Literals cannot be larger than their type. The largest type defined is 256-bit wide.

During assignments and function calls, the types of the respective values have to match. There is no implicit type conversion. Type conversion in general can only be achieved if the dialect provides an appropriate built-in function that takes a value of one type and returns a value of a different type.

### Scoping Rules

Scopes in Yul are tied to Blocks (exceptions are functions and the for loop as explained below) and all declarations (`FunctionDefinition`, `VariableDeclaration`) introduce new identifiers into these scopes.

Identifiers are visible in the block they are defined in (including all sub-nodes and sub-blocks): Functions are visible in the whole block (even before their definitions) while variables are only visible starting from the statement after the `VariableDeclaration`.

In particular, variables cannot be referenced in the right hand side of their own variable declaration. Functions can be referenced already before their declaration (if they are visible).

As an exception to the general scoping rule, the scope of the «init» part of the for-loop (the first block) extends across all other parts of the for loop. This means that variables (and functions) declared in the init part (but not inside a block inside the init part) are visible in all other parts of the for-loop.

Identifiers declared in the other parts of the for loop respect the regular syntactical scoping rules.

This means a for-loop of the form `for { I... } C { P... } { B... }` is equivalent to `{ I... for {} C { P... } { B... } }`.

The parameters and return parameters of functions are visible in the function body and their names have to be distinct.

Inside functions, it is not possible to reference a variable that was declared outside of that function.

Shadowing is disallowed, i.e. you cannot declare an identifier at a point where another identifier with the same name is also visible, even if it is not possible to reference it because it was declared outside the current function.

### Formal Specification

We formally specify Yul by providing an evaluation function E overloaded on the various nodes of the AST. As builtin functions can have side effects, E takes two state objects and the AST node and returns two new state objects and a variable number of other values. The two state objects are the global state object (which in the context of the EVM is the memory, storage and state of the blockchain) and the local state object (the state of local variables, i.e. a segment of the stack in the EVM).

If the AST node is a statement, E returns the two state objects and a «mode», which is used for the `break`, `continue` and `leave` statements. If the AST node is an expression, E returns the two state objects and as many values as the expression evaluates to.

The exact nature of the global state is unspecified for this high level description. The local state `L` is a mapping of identifiers `i` to values `v`, denoted as `L[i] = v`.

For an identifier `v`, let `$v` be the name of the identifier.

We will use a destructuring notation for the AST nodes.

```
E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    let L2 be a restriction of L1 to the identifiers of L
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
            E(G1, L1, St2, ..., Stn)
        otherwise
```

(continué en la próxima página)

(proviene de la página anterior)

```
            G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
    E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
    let L1 be a copy of L where L1[$var_i] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$var_i] = vi for i = 1, ..., n
    G1, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L1, mode = E(G, L, i1, ..., in)
        // mode has to be regular or leave due to the syntactic restrictions
        if mode is leave then
            G1, L1 restricted to variables of L, leave
        otherwise
            let G2, L2, mode = E(G1, L1, for {} condition post body)
            G2, L2 restricted to variables of L, mode
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            otherwise if mode is leave:
                G2, L2, leave
            else:
                G3, L3, mode = E(G2, L2, post)
                if mode is leave:
                    G3, L3, leave
                otherwise
                    E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
    G, L, continue
E(G, L, leave: Leave) =
    G, L, leave
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
```

(continué en la próxima página)

(proviene de la página anterior)

```
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // Evaluate literals, context doesn't matter
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    if there exists smallest i such that vi = v:
        E(G0, L0, sti)
    else:
        E(G0, L0, st')

E(G, L, <name>: Identifier) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
    ...
    G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
    Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
    Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
    be the function of name $fname visible at the point of the call.
    Let L' be a new local state such that
    L'[$parami] = vi and L'[$reti] = 0 for all i.
    Let G'', L'', mode = E(Gn, L', block)
    G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: StringLiteral) = G, L, str(l),
    where str is the string evaluation function,
    which for the EVM dialect is defined in the section 'Literals' above
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal evaluation function,
    which turns a sequence of hexadecimal digits into their big endian value
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal evaluation function,
    which turns a sequence of decimal digits into their big endian value
```

### EVM Dialect

The default dialect of Yul currently is the EVM dialect for the currently selected version of the EVM. with a version of the EVM. The only type available in this dialect is `u256`, the 256-bit native type of the Ethereum Virtual Machine. Since it is the default type of this dialect, it can be omitted.

The following table lists all builtin functions (depending on the EVM version) and provides a short description of the semantics of the function / opcode. This document does not want to be a full description of the Ethereum virtual machine. Please refer to a different document if you are interested in the precise semantics.

Opcodes marked with `-` do not return a result and all others return exactly one value. Opcodes marked with `F`, `H`, `B`, `C`, `I`, `L` and `P` are present since Frontier, Homestead, Byzantium, Constantinople, Istanbul, London or Paris respectively.

In the following, `mem[a...b)` signifies the bytes of memory starting at position `a` up to but not including position `b` and `storage[p]` signifies the storage contents at slot `p`.

Since Yul manages local variables and control-flow, opcodes that interfere with these features are not available. This includes the `dup` and `swap` instructions as well as `jump` instructions, labels and the `push` instructions.

| Instruction | | | Explanation |
|---|---|---|---|
| stop() | - | F | stop execution, identical to return(0, 0) |
| add(x, y) | | F | x + y |
| sub(x, y) | | F | x - y |
| mul(x, y) | | F | x * y |
| div(x, y) | | F | x / y or 0 if y == 0 |
| sdiv(x, y) | | F | x / y, for signed numbers in two's complement, 0 if y == 0 |
| mod(x, y) | | F | x % y, 0 if y == 0 |
| smod(x, y) | | F | x % y, for signed numbers in two's complement, 0 if y == 0 |
| exp(x, y) | | F | x to the power of y |
| not(x) | | F | bitwise «not» of x (every bit of x is negated) |
| lt(x, y) | | F | 1 if x < y, 0 otherwise |
| gt(x, y) | | F | 1 if x > y, 0 otherwise |
| slt(x, y) | | F | 1 if x < y, 0 otherwise, for signed numbers in two's complement |
| sgt(x, y) | | F | 1 if x > y, 0 otherwise, for signed numbers in two's complement |
| eq(x, y) | | F | 1 if x == y, 0 otherwise |
| iszero(x) | | F | 1 if x == 0, 0 otherwise |
| and(x, y) | | F | bitwise «and» of x and y |
| or(x, y) | | F | bitwise «or» of x and y |
| xor(x, y) | | F | bitwise «xor» of x and y |
| byte(n, x) | | F | nth byte of x, where the most significant byte is the 0th byte |
| shl(x, y) | | C | logical shift left y by x bits |
| shr(x, y) | | C | logical shift right y by x bits |
| sar(x, y) | | C | signed arithmetic shift right y by x bits |
| addmod(x, y, m) | | F | (x + y) % m with arbitrary precision arithmetic, 0 if m == 0 |
| mulmod(x, y, m) | | F | (x * y) % m with arbitrary precision arithmetic, 0 if m == 0 |
| signextend(i, x) | | F | sign extend from (i*8+7)th bit counting from least significant |
| keccak256(p, n) | | F | keccak(mem[p…(p+n))) |
| pc() | | F | current position in code |
| pop(x) | - | F | discard value x |
| mload(p) | | F | mem[p…(p+32)) |
| mstore(p, v) | - | F | mem[p…(p+32)) := v |
| mstore8(p, v) | - | F | mem[p] := v & 0xff (only modifies a single byte) |
| sload(p) | | F | storage[p] |
| sstore(p, v) | - | F | storage[p] := v |
| msize() | | F | size of memory, i.e. largest accessed memory index |
| gas() | | F | gas still available to execution |
| address() | | F | address of the current contract / execution context |
| balance(a) | | F | wei balance at address a |
| selfbalance() | | I | equivalent to balance(address()), but cheaper |
| caller() | | F | call sender (excluding `delegatecall`) |
| callvalue() | | F | wei sent together with the current call |
| calldataload(p) | | F | call data starting from position p (32 bytes) |
| calldatasize() | | F | size of call data in bytes |
| calldatacopy(t, f, s) | - | F | copy s bytes from calldata at position f to mem at position t |
| codesize() | | F | size of the code of the current contract / execution context |
| codecopy(t, f, s) | - | F | copy s bytes from code at position f to mem at position t |
| extcodesize(a) | | F | size of the code at address a |
| extcodecopy(a, t, f, s) | - | F | like codecopy(t, f, s) but take code at address a |
| returndatasize() | | B | size of the last returndata |
| returndatacopy(t, f, s) | - | B | copy s bytes from returndata at position f to mem at position t |

| Instruction | | | Explanation |
|---|---|---|---|
| extcodehash(a) | | C | code hash of address a |
| create(v, p, n) | | F | create new contract with code mem[p…(p+n)) and send v wei and return the new |
| create2(v, p, n, s) | | C | create new contract with code mem[p…(p+n)) at address keccak256(0xff . this . s |
| call(g, a, v, in, insize, out, outsize) | | F | call contract at address a with input mem[in…(in+insize)) providing g gas and v v |
| callcode(g, a, v, in, insize, out, outsize) | | F | identical to `call` but only use the code from a and stay in the context of the curren |
| delegatecall(g, a, in, insize, out, outsize) | | H | identical to `callcode` but also keep `caller` and `callvalue` *See more* |
| staticcall(g, a, in, insize, out, outsize) | | B | identical to `call(g, a, 0, in, insize, out, outsize)` but do not allow st |
| return(p, s) | - | F | end execution, return data mem[p…(p+s)) |
| revert(p, s) | - | B | end execution, revert state changes, return data mem[p…(p+s)) |
| selfdestruct(a) | - | F | end execution, destroy current contract and send funds to a (deprecated) |
| invalid() | - | F | end execution with invalid instruction |
| log0(p, s) | - | F | log without topics and data mem[p…(p+s)) |
| log1(p, s, t1) | - | F | log with topic t1 and data mem[p…(p+s)) |
| log2(p, s, t1, t2) | - | F | log with topics t1, t2 and data mem[p…(p+s)) |
| log3(p, s, t1, t2, t3) | - | F | log with topics t1, t2, t3 and data mem[p…(p+s)) |
| log4(p, s, t1, t2, t3, t4) | - | F | log with topics t1, t2, t3, t4 and data mem[p…(p+s)) |
| chainid() | | I | ID of the executing chain (EIP-1344) |
| basefee() | | L | current block's base fee (EIP-3198 and EIP-1559) |
| origin() | | F | transaction sender |
| gasprice() | | F | gas price of the transaction |
| blockhash(b) | | F | hash of block nr b - only for last 256 blocks excluding current |
| coinbase() | | F | current mining beneficiary |
| timestamp() | | F | timestamp of the current block in seconds since the epoch |
| number() | | F | current block number |
| difficulty() | | F | difficulty of the current block (see note below) |
| prevrandao() | | P | randomness provided by the beacon chain (see note below) |
| gaslimit() | | F | block gas limit of the current block |

**Nota:** The `call*` instructions use the `out` and `outsize` parameters to define an area in memory where the return or failure data is placed. This area is written to depending on how many bytes the called contract returns. If it returns more data, only the first `outsize` bytes are written. You can access the rest of the data using the `returndatacopy` opcode. If it returns less data, then the remaining bytes are not touched at all. You need to use the `returndatasize` opcode to check which part of this memory area contains the return data. The remaining bytes will retain their values as of before the call.

**Nota:** The *difficulty()* instruction is disallowed in EVM version >= Paris. With the Paris network upgrade the semantics of the instruction that was previously called `difficulty` have been changed and the instruction was renamed to `prevrandao`. It can now return arbitrary values in the full 256-bit range, whereas the highest recorded difficulty value within Ethash was ~54 bits. This change is described in EIP-4399. Please note that irrelevant to which EVM version is selected in the compiler, the semantics of instructions depend on the final chain of deployment.

**Advertencia:** From version 0.8.18 and up, the use of `selfdestruct` in both Solidity and Yul will trigger a deprecation warning, since the `SELFDESTRUCT` opcode will eventually undergo breaking changes in behaviour as stated in EIP-6049.

In some internal dialects, there are additional functions:

### datasize, dataoffset, datacopy

The functions `datasize(x)`, `dataoffset(x)` and `datacopy(t, f, l)` are used to access other parts of a Yul object.

`datasize` and `dataoffset` can only take string literals (the names of other objects) as arguments and return the size and offset in the data area, respectively. For the EVM, the `datacopy` function is equivalent to `codecopy`.

### setimmutable, loadimmutable

The functions `setimmutable(offset, "name", value)` and `loadimmutable("name")` are used for the immutable mechanism in Solidity and do not nicely map to pure Yul. The call to `setimmutable(offset, "name", value)` assumes that the runtime code of the contract containing the given named immutable was copied to memory at offset `offset` and will write `value` to all positions in memory (relative to `offset`) that contain the placeholder that was generated for calls to `loadimmutable("name")` in the runtime code.

### linkersymbol

The function `linkersymbol("library_id")` is a placeholder for an address literal to be substituted by the linker. Its first and only argument must be a string literal and uniquely represents the address to be inserted. Identifiers can be arbitrary but when the compiler produces Yul code from Solidity sources, it uses a library name qualified with the name of the source unit that defines that library. To link the code with a particular library address, the same identifier must be provided to the `--libraries` option on the command line.

For example this code

```
let a := linkersymbol("file.sol:Math")
```

is equivalent to

```
let a := 0x1234567890123456789012345678901234567890
```

when the linker is invoked with `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890` option.

See *Using the Commandline Compiler* for details about the Solidity linker.

### memoryguard

This function is available in the EVM dialect with objects. The caller of `let ptr := memoryguard(size)` (where `size` has to be a literal number) promises that they only use memory in either the range `[0, size)` or the unbounded range starting at `ptr`.

Since the presence of a `memoryguard` call indicates that all memory access adheres to this restriction, it allows the optimizer to perform additional optimization steps, for example the stack limit evader, which attempts to move stack variables that would otherwise be unreachable to memory.

The Yul optimizer promises to only use the memory range `[size, ptr)` for its purposes. If the optimizer does not need to reserve any memory, it holds that `ptr == size`.

`memoryguard` can be called multiple times, but needs to have the same literal as argument within one Yul subobject. If at least one `memoryguard` call is found in a subobject, the additional optimiser steps will be run on it.

### verbatim

The set of `verbatim...` builtin functions lets you create bytecode for opcodes that are not known to the Yul compiler. It also allows you to create bytecode sequences that will not be modified by the optimizer.

The functions are `verbatim_<n>i_<m>o("<data>", ...)`, where

- `n` is a decimal between 0 and 99 that specifies the number of input stack slots / variables
- `m` is a decimal between 0 and 99 that specifies the number of output stack slots / variables
- `data` is a string literal that contains the sequence of bytes

If you for example want to define a function that multiplies the input by two, without the optimizer touching the constant two, you can use

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

This code will result in a `dup1` opcode to retrieve `x` (the optimizer might directly re-use result of the `calldataload` opcode, though) directly followed by `600202`. The code is assumed to consume the copied value of `x` and produce the result on the top of the stack. The compiler then generates code to allocate a stack slot for `double` and store the result there.

As with all opcodes, the arguments are arranged on the stack with the leftmost argument on the top, while the return values are assumed to be laid out such that the rightmost variable is at the top of the stack.

Since `verbatim` can be used to generate arbitrary opcodes or even opcodes unknown to the Solidity compiler, care has to be taken when using `verbatim` together with the optimizer. Even when the optimizer is switched off, the code generator has to determine the stack layout, which means that e.g. using `verbatim` to modify the stack height can lead to undefined behaviour.

The following is a non-exhaustive list of restrictions on verbatim bytecode that are not checked by the compiler. Violations of these restrictions can result in undefined behaviour.

- Control-flow should not jump into or out of verbatim blocks, but it can jump within the same verbatim block.
- Stack contents apart from the input and output parameters should not be accessed.
- The stack height difference should be exactly `m - n` (output slots minus input slots).
- Verbatim bytecode cannot make any assumptions about the surrounding bytecode. All required parameters have to be passed in as stack variables.

The optimizer does not analyze verbatim bytecode and always assumes that it modifies all aspects of state and thus can only do very few optimizations across `verbatim` function calls.

The optimizer treats verbatim bytecode as an opaque block of code. It will not split it but might move, duplicate or combine it with identical verbatim bytecode blocks. If a verbatim bytecode block is unreachable by the control-flow, it can be removed.

**Advertencia:** During discussions about whether or not EVM improvements might break existing smart contracts, features inside `verbatim` cannot receive the same consideration as those used by the Solidity compiler itself.

**Nota:** To avoid confusion, all identifiers starting with the string `verbatim` are reserved and cannot be used for user-defined identifiers.

### 3.33.6 Specification of Yul Object

Yul objects are used to group named code and data sections. The functions `datasize`, `dataoffset` and `datacopy` can be used to access these sections from within code. Hex strings can be used to specify data in hex encoding, regular strings in native encoding. For code, `datacopy` will access its assembled binary representation.

```
Object = 'object' StringLiteral '{' Code ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral ( HexLiteral | StringLiteral )
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
```

Above, `Block` refers to `Block` in the Yul code grammar explained in the previous chapter.

**Nota:** An object with a name that ends in `_deployed` is treated as deployed code by the Yul optimizer. The only consequence of this is a different gas cost heuristic in the optimizer.

**Nota:** Data objects or sub-objects whose names contain a `.` can be defined but it is not possible to access them through `datasize`, `dataoffset` or `datacopy` because `.` is used as a separator to access objects inside another object.

**Nota:** The data object called `".metadata"` has a special meaning: It cannot be accessed from code and is always appended to the very end of the bytecode, regardless of its position in the object.

Other data objects with special significance might be added in the future, but their names will always start with a `.`.

An example Yul Object is shown below:

```
// A contract consists of a single object with sub-objects representing
// the code to be deployed or other contracts it can create.
// The single "code" node is the executable code of the object.
// Every (other) named object or data section is serialized and
// made accessible to the special built-in functions datacopy / dataoffset /␣
↪datasize
// The current object, sub-objects and data items inside the current object
// are in scope.
object "Contract1" {
    // This is the constructor code of the contract.
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            // Note that Solidity generated IR code reserves memory offset ``0x60``␣
↪as well, but a pure Yul object is free to use memory as it chooses.
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // first create "Contract2"
        let size := datasize("Contract2")
        let offset := allocate(size)
        // This will turn into codecopy for EVM
```

(continué en la próxima página)

(proviene de la página anterior)

```
        datacopy(offset, dataoffset("Contract2"), size)
        // constructor parameter is a single number 0x1234
        mstore(add(offset, size), 0x1234)
        pop(create(0, offset, add(size, 32)))

        // now return the runtime object (the currently
        // executing code is the constructor code)
        size := datasize("Contract1_deployed")
        offset := allocate(size)
        // This will turn into a memory->memory copy for Ewasm and
        // a codecopy for EVM
        datacopy(offset, dataoffset("Contract1_deployed"), size)
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "Contract1_deployed" {
        code {
            function allocate(size) -> ptr {
                ptr := mload(0x40)
                // Note that Solidity generated IR code reserves memory offset␣
↪``0x60`` as well, but a pure Yul object is free to use memory as it chooses.
                if iszero(ptr) { ptr := 0x60 }
                mstore(0x40, add(ptr, size))
            }

            // runtime code

            mstore(0, "Hello, World!")
            return(0, 0x20)
        }
    }

    // Embedded object. Use case is that the outside is a factory contract,
    // and Contract2 is the code to be created by the factory
    object "Contract2" {
        code {
            // code here ...
        }

        object "Contract2_deployed" {
            code {
                // code here ...
            }
        }

        data "Table1" hex"4123"
    }
}
```

### 3.33.7 Yul Optimizer

The Yul optimizer operates on Yul code and uses the same language for input, output and intermediate states. This allows for easy debugging and verification of the optimizer.

Please refer to the general *optimizer documentation* for more details about the different optimization stages and how to use the optimizer.

If you want to use Solidity in stand-alone Yul mode, you activate the optimizer using `--optimize` and optionally specify the *expected number of contract executions* with `--optimize-runs`:

```
solc --strict-assembly --optimize --optimize-runs 200
```

In Solidity mode, the Yul optimizer is activated together with the regular optimizer.

#### Optimization Step Sequence

Detailed information regrading the optimization sequence as well a list of abbreviations is available in the *optimizer docs*.

### 3.33.8 Complete ERC20 Example

```
object "Token" {
    code {
        // Store the creator in slot zero.
        sstore(0, caller())

        // Deploy the contract
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            // Protection against sending Ether
            require(iszero(callvalue()))

            // Dispatcher
            switch selector()
            case 0x70a08231 /* "balanceOf(address)" */ {
                returnUint(balanceOf(decodeAsAddress(0)))
            }
            case 0x18160ddd /* "totalSupply()" */ {
                returnUint(totalSupply())
            }
            case 0xa9059cbb /* "transfer(address,uint256)" */ {
                transfer(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            case 0x23b872dd /* "transferFrom(address,address,uint256)" */ {
                transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))
                returnTrue()
            }
```

(continué en la próxima página)

(proviene de la página anterior)

```
            case 0x095ea7b3 /* "approve(address,uint256)" */ {
                approve(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            case 0xdd62ed3e /* "allowance(address,address)" */ {
                returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))
            }
            case 0x40c10f19 /* "mint(address,uint256)" */ {
                mint(decodeAsAddress(0), decodeAsUint(1))
                returnTrue()
            }
            default {
                revert(0, 0)
            }

            function mint(account, amount) {
                require(calledByOwner())

                mintTokens(amount)
                addToBalance(account, amount)
                emitTransfer(0, account, amount)
            }
            function transfer(to, amount) {
                executeTransfer(caller(), to, amount)
            }
            function approve(spender, amount) {
                revertIfZeroAddress(spender)
                setAllowance(caller(), spender, amount)
                emitApproval(caller(), spender, amount)
            }
            function transferFrom(from, to, amount) {
                decreaseAllowanceBy(from, caller(), amount)
                executeTransfer(from, to, amount)
            }

            function executeTransfer(from, to, amount) {
                revertIfZeroAddress(to)
                deductFromBalance(from, amount)
                addToBalance(to, amount)
                emitTransfer(from, to, amount)
            }


            /* ---------- calldata decoding functions ----------- */
            function selector() -> s {
                s := div(calldataload(0),␣
↪0x100000000000000000000000000000000000000000000000000000000)
            }

            function decodeAsAddress(offset) -> v {
                v := decodeAsUint(offset)
                if iszero(iszero(and(v,␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪not(0xffffffffffffffffffffffffffffffffffffffff)))) {
                    revert(0, 0)
                }
            }
            function decodeAsUint(offset) -> v {
                let pos := add(4, mul(offset, 0x20))
                if lt(calldatasize(), add(pos, 0x20)) {
                    revert(0, 0)
                }
                v := calldataload(pos)
            }
            /* ---------- calldata encoding functions ---------- */
            function returnUint(v) {
                mstore(0, v)
                return(0, 0x20)
            }
            function returnTrue() {
                returnUint(1)
            }

            /* -------- events ---------- */
            function emitTransfer(from, to, amount) {
                let signatureHash :=␣
↪0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
                emitEvent(signatureHash, from, to, amount)
            }
            function emitApproval(from, spender, amount) {
                let signatureHash :=␣
↪0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
                emitEvent(signatureHash, from, spender, amount)
            }
            function emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {
                mstore(0, nonIndexed)
                log3(0, 0x20, signatureHash, indexed1, indexed2)
            }

            /* -------- storage layout ---------- */
            function ownerPos() -> p { p := 0 }
            function totalSupplyPos() -> p { p := 1 }
            function accountToStorageOffset(account) -> offset {
                offset := add(0x1000, account)
            }
            function allowanceStorageOffset(account, spender) -> offset {
                offset := accountToStorageOffset(account)
                mstore(0, offset)
                mstore(0x20, spender)
                offset := keccak256(0, 0x40)
            }

            /* -------- storage access ---------- */
            function owner() -> o {
                o := sload(ownerPos())
```

(continué en la próxima página)

(proviene de la página anterior)

```
            }
            function totalSupply() -> supply {
                supply := sload(totalSupplyPos())
            }
            function mintTokens(amount) {
                sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))
            }
            function balanceOf(account) -> bal {
                bal := sload(accountToStorageOffset(account))
            }
            function addToBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                sstore(offset, safeAdd(sload(offset), amount))
            }
            function deductFromBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                let bal := sload(offset)
                require(lte(amount, bal))
                sstore(offset, sub(bal, amount))
            }
            function allowance(account, spender) -> amount {
                amount := sload(allowanceStorageOffset(account, spender))
            }
            function setAllowance(account, spender, amount) {
                sstore(allowanceStorageOffset(account, spender), amount)
            }
            function decreaseAllowanceBy(account, spender, amount) {
                let offset := allowanceStorageOffset(account, spender)
                let currentAllowance := sload(offset)
                require(lte(amount, currentAllowance))
                sstore(offset, sub(currentAllowance, amount))
            }

            /* ---------- utility functions ---------- */
            function lte(a, b) -> r {
                r := iszero(gt(a, b))
            }
            function gte(a, b) -> r {
                r := iszero(lt(a, b))
            }
            function safeAdd(a, b) -> r {
                r := add(a, b)
                if or(lt(r, a), lt(r, b)) { revert(0, 0) }
            }
            function calledByOwner() -> cbo {
                cbo := eq(owner(), caller())
            }
            function revertIfZeroAddress(addr) {
                require(addr)
            }
            function require(condition) {
                if iszero(condition) { revert(0, 0) }
```

(continué en la próxima página)

(proviene de la página anterior)

```
            }
        }
    }
}
```

## 3.34 Style Guide

### 3.34.1 Introduction

This guide is intended to provide coding conventions for writing Solidity code. This guide should be thought of as an evolving document that will change over time as useful conventions are found and old conventions are rendered obsolete.

Many projects will implement their own style guides. In the event of conflicts, project specific style guides take precedence.

The structure and many of the recommendations within this style guide were taken from python's pep8 style guide.

The goal of this guide is *not* to be the right way or the best way to write Solidity code. The goal of this guide is *consistency*. A quote from python's pep8 captures this concept well.

---

**Nota:** A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: **know when to be inconsistent** – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

---

### 3.34.2 Code Layout

#### Indentation

Use 4 spaces per indentation level.

#### Tabs or Spaces

Spaces are the preferred indentation method.

Mixing tabs and spaces should be avoided.

#### Blank Lines

Surround top level declarations in Solidity source with two blank lines.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // ...
}


contract B {
    // ...
}


contract C {
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Within a contract surround function declarations with a single blank line.

Blank lines may be omitted between groups of related one-liners (such as stub functions for an abstract contract)

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() public virtual pure;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() virtual pure public;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}
```

### Maximum Line Length

Maximum suggested line length is 120 characters.

Wrapped lines should conform to the following guidelines.

1. The first argument should not be attached to the opening parenthesis.
2. One, and only one, indent should be used.
3. Each argument should fall on its own line.
4. The terminating element, `);`, should be placed on the final line by itself.

Function Calls

Yes:

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

No:

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);
```

(continué en la próxima página)

(proviene de la página anterior)

```
thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Assignment Statements

Yes:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

No:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(argument1,
                                                                 argument2,
                                                                 argument3,
                                                                 argument4);
```

Event Definitions and Event Emitters

Yes:

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

No:

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

### Source File Encoding

UTF-8 or ASCII encoding is preferred.

### Imports

Import statements should always be placed at the top of the file.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";

contract A {
    // ...
}


contract B is Owned {
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}


import "./Owned.sol";


contract B is Owned {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // ...
}
```

### Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the `view` and `pure` functions last.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {
    constructor() {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
    }

    // External functions
    // ...

    // External functions that are view
    // ...

    // External functions that are pure
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // Private functions
    // ...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {

    // External functions
    // ...

    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // Private functions
    // ...

    // Public functions
    // ...

    constructor() {
        // ...
    }

    // Internal functions
    // ...
}
```

### Whitespace in Expressions

Avoid extraneous whitespace in the following situations:

Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.

Yes:

```
spam(ham[1], Coin({name: "ham"}));
```

No:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception:

```
function singleLine() public { spam(); }
```

Immediately before a comma, semicolon:

Yes:

```
function spam(uint i, Coin coin) public;
```

No:

```
function spam(uint i , Coin coin) public ;
```

More than one space around an assignment or other operator to align with another:

Yes:

```
x = 1;
y = 2;
longVariable = 3;
```

No:

```
x            = 1;
y            = 2;
longVariable = 3;
```

Don't include a whitespace in the receive and fallback functions:

Yes:

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

No:

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

### Control Structures

The braces denoting the body of a contract, library, functions and structs should:

- open on the same line as the declaration
- close on their own line at the same indentation level as the beginning of the declaration.
- The opening brace should be preceded by a single space.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

The same recommendations apply to the control structures `if`, `else`, `while`, and `for`.

Additionally there should be a single space between the control structures `if`, `while`, and `for` and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.

Yes:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

For control structures whose body contains a single statement, omitting the braces is ok *if* the statement is contained on a single line.

Yes:

```
if (x < 10)
    x += 1;
```

No:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

For `if` blocks which have an `else` or `else if` clause, the `else` should be placed on the same line as the `if`'s closing brace. This is an exception compared to the rules of other block-like structures.

Yes:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

No:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

### Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.

The closing brace should be at the same indentation level as the function declaration.

The opening brace should be preceded by a single space.

Yes:

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
    }

function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

The modifier order for a function should be:

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

Yes:

```
function balance(uint from) public view override returns (uint)  {
    return balanceOf[from];
}

function shutdown() public onlyOwner {
    selfdestruct(owner);
}
```

No:

```
function balance(uint from) public override view returns (uint)  {
    return balanceOf[from];
}

function shutdown() onlyOwner public {
    selfdestruct(owner);
}
```

For long function declarations, it is recommended to drop each argument onto its own line at the same indentation level as the function body. The closing parenthesis and opening bracket should be placed on their own line as well at the

same indentation level as the function declaration.

Yes:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
{
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to its own line.

Yes:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyOwner
    priced
    returns (address)
{
    doSomething();
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z
)
    public
    onlyOwner
    priced
    returns (address)
{
    doSomething();
}
```

No:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyOwner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyOwner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyOwner
    priced
    returns (address) {
    doSomething();
}
```

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the *Maximum Line Length* section.

Yes:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
```

(continué en la próxima página)

(proviene de la página anterior)

```
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

No:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
             uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
        x = param5;
    }
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4) {
        x = param5;
    }
}
```

(continué en la próxima página)

(proviene de la página anterior)

```
contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
            x = param5;
        }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible:

```
function shortFunction() public { doSomething(); }
```

These guidelines for function declarations are intended to improve readability. Authors should use their best judgment as this guide does not try to cover all possible permutations for function declarations.

### Mappings

In variable declarations, do not separate the keyword `mapping` from its type by a space. Do not separate any nested `mapping` keyword from its type by whitespace.

Yes:

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

No:

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

### Variable Declarations

Declarations of array variables should not have a space between the type and the brackets.

Yes:

```
uint[] x;
```

No:

```
uint [] x;
```

### Other Recommendations

- Strings should be quoted with double-quotes instead of single-quotes.

Yes:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- Surround operators with a single space on either side.

Yes:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence. This is meant to allow for improved readability for complex statements. You should always use the same amount of whitespace on either side of an operator:

Yes:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

### 3.34.3 Order of Layout

Layout contract elements in the following order:

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Errors
5. Modifiers
6. Functions

---

**Nota:** It might be clearer to declare types close to their use in events or state variables.

---

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.4 <0.9.0;

abstract contract Math {
    error DivideByZero();
    function divide(int256 numerator, int256 denominator) public virtual returns␣
↪(uint256);
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.4 <0.9.0;

abstract contract Math {
    function divide(int256 numerator, int256 denominator) public virtual returns␣
↪(uint256);
    error DivideByZero();
}
```

### 3.34.4 Naming Conventions

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant *meta* information that would otherwise not be immediately available.

The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Lastly, consistency within a codebase should always supersede any conventions outlined in this document.

#### Naming Styles

To avoid confusion, the following names will be used to refer to different naming styles.

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords)
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)

---

**Nota:** When using initialisms in CapWords, capitalize all the letters of the initialisms. Thus HTTPServerError is better than HttpServerError. When using initialisms in mixedCase, capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name. Thus xmlHTTPRequest is better than XMLHTTPRequest.

---

#### Names to Avoid

- `l` - Lowercase letter el
- `O` - Uppercase letter oh
- `I` - Uppercase letter eye

Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.

#### Contract and Library Names

- Contracts and libraries should be named using the CapWords style. Examples: `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
- Contract and library names should also match their filenames.
- If a contract file includes multiple contracts and/or libraries, then the filename should match the *core contract*. This is not recommended however if it can be avoided.

As shown in the example below, if the contract name is `Congress` and the library name is `Owned`, then their associated filenames should be `Congress.sol` and `Owned.sol`.

Yes:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Owned.sol
contract Owned {
    address public owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";


contract Congress is Owned, TokenRecipient {
    //...
}
```

No:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// owned.sol
contract owned {
    address public owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }
}
```

and in `Congress.sol`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;


import "./owned.sol";


contract Congress is owned, tokenRecipient {
    //...
}
```

### Struct Names

Structs should be named using the CapWords style. Examples: `MyCoin`, `Position`, `PositionXY`.

### Event Names

Events should be named using the CapWords style. Examples: `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

### Function Names

Functions should use mixedCase. Examples: `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

### Function Argument Names

Function arguments should use mixedCase. Examples: `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

When writing library functions that operate on a custom struct, the struct should be the first argument and should always be named `self`.

### Local and State Variable Names

Use mixedCase. Examples: `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

#### Constants

Constants should be named with all capital letters with underscores separating words. Examples: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

#### Modifier Names

Use mixedCase. Examples: `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

#### Enums

Enums, in the style of simple type declarations, should be named using the CapWords style. Examples: `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

#### Avoiding Naming Collisions

- `singleTrailingUnderscore_`

This convention is suggested when the desired name collides with that of an existing state variable, function, built-in or otherwise reserved name.

#### Underscore Prefix for Non-external Functions and Variables

- `_singleLeadingUnderscore`

This convention is suggested for non-external functions and state variables (`private` or `internal`). State variables without a specified visibility are `internal` by default.

When designing a smart contract, the public-facing API (functions that can be called by any account) is an important consideration. Leading underscores allow you to immediately recognize the intent of such functions, but more importantly, if you change a function from non-external to external (including `public`) and rename it accordingly, this forces you to review every call site while renaming. This can be an important manual check against unintended external functions and a common source of security vulnerabilities (avoid find-replace-all tooling for this change).

### 3.34.5 NatSpec

Solidity contracts can also contain NatSpec comments. They are written with a triple slash (`///`) or a double asterisk block (`/** ... */`) and they should be used directly above function declarations or statements.

For example, the contract from *a simple smart contract* with the comments added looks like the one below:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

/// @author The Solidity Team
/// @title A simple storage example
contract SimpleStorage {
    uint storedData;

    /// Store `x`.
    /// @param x the new value to store
    /// @dev stores the number in the state variable `storedData`
```

(continué en la próxima página)

(proviene de la página anterior)

```
    function set(uint x) public {
        storedData = x;
    }

    /// Return the stored value.
    /// @dev retrieves the value of the state variable `storedData`
    /// @return the stored value
    function get() public view returns (uint) {
        return storedData;
    }
}
```

It is recommended that Solidity contracts are fully annotated using *NatSpec* for all public interfaces (everything in the ABI).

Please see the section about *NatSpec* for a detailed explanation.

## 3.35 Patrones comunes

### 3.35.1 Retiros desde Contratos

El método recomendado al enviar fondos luego de un efecto es usar el patrón de retiros. Aunque el método más intuitivo de enviar Ether, como resultado de un efecto, es una llamada directa a `transfer`, esto no se recomienda ya que introduce un riesgo potencial de seguridad. Podría leer más sobre esto en la página consideraciones_de_seguridad.

El siguiente ejemplo es un patrón de retiros en práctica en un contrato donde el objetivo es enviar la mayor cantidad de dinero al contrato a fin de llegar a ser «el más rico», inspirado por King of the Ether.

En el siguiente contrato, si usted ya no es el más rico, recibe los fondos de la persona que ahora es la más rica.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping(address => uint) pendingWithdrawals;

    /// La cantidad enviada de Ether no fue más alta que
    /// la cantidad actualmente más alta.
    error NotEnoughEther();

    constructor() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
```

(continué en la próxima página)

(proviene de la página anterior)

```
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Recuerde poner a cero la devolución pendiente antes de
        // enviar para prevenir ataques de reentrada.
        pendingWithdrawals[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

Patrón de envío:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    /// La cantidad enviada de Ether no fue más alta que
    /// la cantidad actualmente más alta.
    error NotEnoughEther();

    constructor() payable {
        richest = payable(msg.sender);
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        // Esta línea puede causar problemas (explicado más abajo).
        richest.transfer(msg.value);
        richest = payable(msg.sender);
        mostSent = msg.value;
    }
}
```

Nótese que, en este ejemplo, un atacante podría atrapar al contrato hacia un estado inservible al causar que `richest` sea la dirección de un contrato que tiene una función receive o fallback el cual falla (e.g. al usar `revert()` o por solo consumir más de 2300 estipendio de gas transferido a ello). De esa manera, cuando se llame `transfer` para entregar los fondos al contrato «envenenado», fallará y así también fallará `becomeRichest`, con el contrato quedando estancado para siempre.

En contraste, si usted usa el patrón de retiros del primer ejemplo, el atacante solo puede causar que falle su propio retiro y no el resto del funcionamiento del contrato.

### 3.35.2 Restricción de Acceso

Restricción de acceso es un patrón muy común en contratos. Note que usted no puede nunca restringir a ningún humano o computadora de leer el contenido de sus transacciones o el estado del contrato. Lo puede hacer un poco más complicado al usar encriptación, pero si se supone que su contrato lea datos, entonces todos podrán también hacerlo.

usted puede restringir el acceso a lectura del estado de su contrato por **otros contratos**. De hecho, ese es el valor por defecto a menos que usted declare sus variables de estado `public`.

Además, puede restringir quién puede hacer modificaciones al estado de sus contratos o llamar a las funciones de su contrato y esto es de lo que se trata esta sección.

El uso de **modificadores de funciones** hacen a estas restricciones sumamente legibles.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract AccessRestriction {
    // Estos serán asignados en la fase de
    // construcción, donde `msg.sender` es la cuenta
    // que crea este contrato.
    address public owner = msg.sender;
    uint public creationTime = block.timestamp;

    // Ahora sigue una lista de errores que
    // este contrato puede generar junto a
    // una explicación textual en
    // comentarios especiales.

    /// Remitente no autorizado para esta
    /// operación.
    error Unauthorized();

    /// Función invocada muy temprano.
    error TooEarly();

    /// No se envió suficiente Ether con la llamada a la función.
    error NotEnoughEther();

    // Modificadores pueden usarse para cambiar
    // el cuerpo de una función.
    // Si este modificador es usado,
    // antepondrá una verificación que solo pasa
    // si la función se invoca desde
    // una cierta dirección.
    modifier onlyBy(address account)
    {
        if (msg.sender != account)
            revert Unauthorized();
        // ¡No olvide el "_;"!
        // Se reemplazará por el cuerpo de la función actual
        // cuando se use el modificador.
        _;
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    /// Convierte a `newOwner` al nuevo  dueño
    /// de este contrato.
    function changeOwner(address newOwner)
        public
        onlyBy(owner)
    {
        owner = newOwner;
    }

    modifier onlyAfter(uint time) {
        if (block.timestamp < time)
            revert TooEarly();
        _;
    }

    /// Borra información de propiedad.
    /// Solo puede ser llamado 6 semanas después
    /// de que se creó el contrato.
    function disown()
        public
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // Este modificador requiere un cierto pago
    // asociado con la llamada a la función.
    // Si quien invoca la función envió demasiado, se le devuelve el dinero,
    // pero solo después del cuerpo de la función.
    // Esto era peligroso antes de Solidity 0.4.0,
    //  donde era posible saltarse la parte después de `_;`.
    modifier costs(uint amount) {
        if (msg.value < amount)
            revert NotEnoughEther();

        _;
        if (msg.value > amount)
            payable(msg.sender).transfer(msg.value - amount);
    }

    function forceOwnerChange(address newOwner)
        public
        payable
        costs(200 ether)
    {
        owner = newOwner;
        // solo un ejemplo de condición
        if (uint160(owner) & 0 == 1)
            // Esto no devolvía el dinero en Solidity
            // antes de la versiñon 0.4.0.
            return;
```

(continué en la próxima página)

(proviene de la página anterior)

```
        // pago de más devuelto
    }
}
```

Una manera más especializada en la cual se puede restringir el acceso a llamadas de funciones será discutida en el siguiente ejemplo.

### 3.35.3 Máquina de Estado

Los contratos a menudo actúan como una máquina de estado, lo cual significa que tienen ciertas **etapas** en las cuales se comportan diferentes o en la cual funciones diferentes pueden ser llamadas. Una llamada a una función a menudo finaliza una etapa y transiciona el contrato a una nueva etapa (especialmente si el contrato modela **interacción**). También es común que algunas etapas se alcancen automáticamente en un cierto punto en el **tiempo**.

Un ejemplo de esto es un contrato de subasta a ciegas el cual comienza en la etapa de «aceptación de ofertas a ciegas», luego transiciona a «revelación de ofertas» lo cual termina con la «determinación del resultado de la subasta».

Los modificadores de funciones se pueden usar en esta situación para modelar los estados y proteger de un uso incorrecto del contrato.

#### Ejemplo

En el siguiente ejemplo, el modificador `atStage` asegura que la función solo puede ser invocada en un escenario en particular.

Las transiciones automáticas planeadas son manejadas por el modificador `timedTransitions`, el cual debería ser usado para todas las funciones.

---

**Nota: El orden del modificador importa**. Si se combina atStage con timedTransitions, asegúrese que lo menciona luego del último, de modo que se tome en cuenta el nuevo escenario.

---

Finalmente, el modificador `transitionNext` se puede usar para ir automáticamente al próximo escenario cuando termina la función.

---

**Nota: El modificador puede ser saltado**. Esto solo aplica a Solidity antes de la versión 0.4.0: Ya que los modificadores se aplican al simplemente reemplazar código y noal usar una llamada a una función, el código en el modificador transitionNext puede ser saltado si la misma función usa return. Si quiere hacer esto, asegúrese llamar a nextStage manualmente desde esas funciones. A partir de la versión 0.4.0, el código del modificador se jecutará incluso si la función retorna explícitamente.

---

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }
    /// La función no puede ser llamada en este momento.
    error FunctionInvalidAtThisStage();

    // Este es el escenario actual.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = block.timestamp;

    modifier atStage(Stages stage_) {
        if (stage != stage_)
            revert FunctionInvalidAtThisStage();
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Lleva a cabo transiciones programadas. Asegúrese de mencionar
    // primero este modificador, de lo contrario los guardas
    // no tomarán en cuenta el nuevo escenario.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    block.timestamp >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                block.timestamp >= creationTime + 12 days)
            nextStage();
        // Los otros escenarios transicionan por transacción
        _;
    }

    // ¡El orden de los modificadores import aquí!
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // No implementaremos eso aquí
    }

    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // Este modificador va al nuevo escenario
    // luego de que termine la función.
```

(continué en la próxima página)

(proviene de la página anterior)

```
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

## 3.36 Lista de Bugs Conocidos

A continuación, puede encontrar una lista, en formato JSON, de algunos de los bugs más conocidos, relacionados con la seguridad del compilador de Solidity. El archivo se encuentra en el repositorio de GitHub. La lista se remonta a la versión 0.3.0, los errores que se conocen, que están presentes solamente en versiones anteriores, no se enumeran.

Hay otro archivo llamado bugs_by_version.json, que se puede utilizar para comprobar qué errores afectan a una versión específica del compilador.

Herramientas para la verificación del código fuente de un contrato, y otras herramientas que interactúen con contratos, deben consultar esta lista de acuerdo a los siguientes criterios:

- Se puede sospechar levemente si un contrato fue compilado con una versión nightly del compilador, en vez de un versión publicada. Esta lista no realiza un seguimiento de las versiones que no han sido publicadas o versiones nightly.
- También se puede sospechar levemente si un contrato se compiló con una versión que no era la más reciente en el momento de la creación del contrato. Para los contratos creados a partir de otros contratos, se debe seguir la cadena de creación hasta una transacción, y usar la fecha de esa transacción como fecha de creación.
- Se puede sospechar altamente si un contrato se compiló con un compilador que contiene un bug conocido y el contrato se creó en un momento en el que ya se había lanzado una versión más reciente del compilador que ya contenía una corrección.

El archivo JSON de bugs conocidos, que se encuentra a continuación, es una matriz de objetos, uno por cada error, con las siguientes claves:

**uid**
: Identificador único dado al error en forma de `SOL-<año>-<número>`. Es posible que existan varias entradas con el mismo uid. Esto significa que varios rangos de versiones se ven afectados por el mismo error.

**name**
: Nombre único que se le da al bug.

**summary**
: Una breve descripción del bug.

**description**
: Una descripción detallada del bug.

**link**
: URL del sitio web con información detallada. Opcional.

**introduced**
: La primera versión del compilador publicada que contenía el bug. Opcional.

**fixed**
: La primera versión del compilador publicada que ya no contenía el bug.

**publish**
: La fecha en la que el bug se hizo público. Opcional.

**severity**
: Gravedad del bug: muy baja, baja, media, alta. Se toma en cuenta la detectabilidad en los tests del contrato, probabilidad de ocurrencia y daños potenciales por exploits.

**conditions**
: Condiciones que deben cumplirse para desencadenar el bug. Se pueden utilizar las siguientes claves: `optimizer`, Valor de tipo boolean, lo que significa que el optimizador debe ser encendido para habilitar el bug. `evmVersion`, un string que indica qué tipo de configuración del compilador de la versión de EVM desencadena el bug. El string puede contener operadores de comparación. Por ejemplo, `">=constantinople"` significa que el bug está presente cuando la versión de EVM se establece en «constantinople» o posterior. Si no hay condiciones, se debe suponer que el bug aún siga presente.

**check**
: Este campo contiene diferentes comprobaciones que informan si el smart contract contiene el bug o no. El primer tipo de verificación son las expresiones regulares de Javascript que deben compararse con el código fuente («source-regex») si el bug está presente. Si no hay compatibilidad, es muy probable que el bug no esté presente. Si hay compatibilidad, es probable que el bug aún siga presente. Para mayor precisión, las comprobaciones deben aplicarse al código fuente después de eliminar los comentarios. El segundo tipo de verificación son patrones que se verificarán en el AST compacto del programa Solidity («ast-compact-json-path»). La consulta de búsqueda especificada es una expresión JsonPath. Si al menos una ruta de Solidity AST coincide con la consulta de búsqueda, es probable que el bug aún siga presente.

```
[
    {
        "uid": "SOL-2022-7",
        "name": "StorageWriteRemovalBeforeConditionalTermination",
        "summary": "Calling functions that conditionally terminate the external EVM call␣
↪using the assembly statements ``return(...)`` or ``stop()`` may result in incorrect␣
↪removals of prior storage writes.",
        "description": "A call to a Yul function that conditionally terminates the␣
↪external EVM call could result in prior storage writes being incorrectly removed by␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪the Yul optimizer. This used to happen in cases in which it would have been valid to␣
↪remove the store, if the Yul function in question never actually terminated the␣
↪external call, and the control flow always returned back to the caller instead.␣
↪Conditional termination within the same Yul block instead of within a called function␣
↪was not affected. In Solidity with optimized via-IR code generation, any storage write␣
↪before a function conditionally calling ``return(...)`` or ``stop()`` in inline␣
↪assembly, may have been incorrectly removed, whenever it would have been valid to␣
↪remove the write without the ``return(...)`` or ``stop()``. In optimized legacy code␣
↪generation, only inline assembly that did not refer to any Solidity variables and that␣
↪involved conditionally-terminating user-defined assembly functions could be affected.",
        "link": "https://blog.soliditylang.org/2022/09/08/storage-write-removal-before-
↪conditional-termination/",
        "introduced": "0.8.13",
        "fixed": "0.8.17",
        "severity": "medium/high",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2022-6",
        "name": "AbiReencodingHeadOverflowWithStaticArrayCleanup",
        "summary": "ABI-encoding a tuple with a statically-sized calldata array in the␣
↪last component would corrupt 32 leading bytes of its first dynamically encoded␣
↪component.",
        "description": "When ABI-encoding a statically-sized calldata array, the␣
↪compiler always pads the data area to a multiple of 32-bytes and ensures that the␣
↪padding bytes are zeroed. In some cases, this cleanup used to be performed by always␣
↪writing exactly 32 bytes, regardless of how many needed to be zeroed. This was done␣
↪with the assumption that the data that would eventually occupy the area past the end␣
↪of the array had not yet been written, because the encoder processes tuple components␣
↪in the order they were given. While this assumption is mostly true, there is an␣
↪important corner case: dynamically encoded tuple components are stored separately from␣
↪the statically-sized ones in an area called the *tail* of the encoding and the tail␣
↪immediately follows the *head*, which is where the statically-sized components are␣
↪placed. The aforementioned cleanup, if performed for the last component of the head␣
↪would cross into the tail and overwrite up to 32 bytes of the first component stored␣
↪there with zeros. The only array type for which the cleanup could actually result in␣
↪an overwrite were arrays with ``uint256`` or ``bytes32`` as the base element type and␣
↪in this case the size of the corrupted area was always exactly 32 bytes. The problem␣
↪affected tuples at any nesting level. This included also structs, which are encoded as␣
↪tuples in the ABI. Note also that lists of parameters and return values of functions,␣
↪events and errors are encoded as tuples.",
        "link": "https://blog.soliditylang.org/2022/08/08/calldata-tuple-reencoding-head-
↪overflow-bug/",
        "introduced": "0.5.8",
        "fixed": "0.8.16",
        "severity": "medium",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
```

(continué en la próxima página)

(proviene de la página anterior)

```
    {
        "uid": "SOL-2022-5",
        "name": "DirtyBytesArrayToStorage",
        "summary": "Copying ``bytes`` arrays from memory or calldata to storage may␣
↪result in dirty storage values.",
        "description": "Copying ``bytes`` arrays from memory or calldata to storage is␣
↪done in chunks of 32 bytes even if the length is not a multiple of 32. Thereby, extra␣
↪bytes past the end of the array may be copied from calldata or memory to storage.␣
↪These dirty bytes may then become observable after a ``.push()`` without arguments to␣
↪the bytes array in storage, i.e. such a push will not result in a zero value at the␣
↪end of the array as expected. This bug only affects the legacy code generation␣
↪pipeline, the new code generation pipeline via IR is not affected.",
        "link": "https://blog.soliditylang.org/2022/06/15/dirty-bytes-array-to-storage-
↪bug/",
        "introduced": "0.0.1",
        "fixed": "0.8.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2022-4",
        "name": "InlineAssemblyMemorySideEffects",
        "summary": "The Yul optimizer may incorrectly remove memory writes from inline␣
↪assembly blocks, that do not access solidity variables.",
        "description": "The Yul optimizer considers all memory writes in the outermost␣
↪Yul block that are never read from as unused and removes them. This is valid when that␣
↪Yul block is the entire Yul program, which is always the case for the Yul code␣
↪generated by the new via-IR pipeline. Inline assembly blocks are never optimized in␣
↪isolation when using that pipeline. Instead they are optimized as a part of the whole␣
↪Yul input. However, the legacy code generation pipeline (which is still the default)␣
↪runs the Yul optimizer individually on an inline assembly block if the block does not␣
↪refer to any local variables defined in the surrounding Solidity code. Consequently,␣
↪memory writes in such inline assembly blocks are removed as well, if the written␣
↪memory is never read from in the same assembly block, even if the written memory is␣
↪accessed later, for example by a subsequent inline assembly block.",
        "link": "https://blog.soliditylang.org/2022/06/15/inline-assembly-memory-side-
↪effects-bug/",
        "introduced": "0.8.13",
        "fixed": "0.8.15",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2022-3",
        "name": "DataLocationChangeInInternalOverride",
        "summary": "It was possible to change the data location of the parameters or␣
↪return variables from ``calldata`` to ``memory`` and vice-versa while overriding␣
↪internal and public functions. This caused invalid code to be generated when calling␣
↪such a function internally through virtual function calls.",
        "description": "When calling external functions, it is irrelevant if the data␣
↪location of the parameters is ``calldata`` or ``memory``, the encoding of the data␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪does not change. Because of that, changing the data location when overriding external␣
↪functions is allowed. The compiler incorrectly also allowed a change in the data␣
↪location for overriding public and internal functions. Since public functions can be␣
↪called internally as well as externally, this causes invalid code to be generated when␣
↪such an incorrectly overridden function is called internally through the base contract.
↪ The caller provides a memory pointer, but the called function interprets it as a␣
↪calldata pointer or vice-versa.",
        "link": "https://blog.soliditylang.org/2022/05/17/data-location-inheritance-bug/
↪",
        "introduced": "0.6.9",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-2",
        "name": "NestedCalldataArrayAbiReencodingSizeValidation",
        "summary": "ABI-reencoding of nested dynamic calldata arrays did not always␣
↪perform proper size checks against the size of calldata and could read beyond␣
↪``calldatasize()``.",
        "description": "Calldata validation for nested dynamic types is deferred until␣
↪the first access to the nested values. Such an access may for example be a copy to␣
↪memory or an index or member access to the outer type. While in most such accesses␣
↪calldata validation correctly checks that the data area of the nested array is␣
↪completely contained in the passed calldata (i.e. in the range [0, calldatasize()]),␣
↪this check may not be performed, when ABI encoding such nested types again directly␣
↪from calldata. For instance, this can happen, if a value in calldata with a nested␣
↪dynamic array is passed to an external call, used in ``abi.encode`` or emitted as␣
↪event. In such cases, if the data area of the nested array extends beyond␣
↪``calldatasize()``, ABI encoding it did not revert, but continued reading values from␣
↪beyond ``calldatasize()`` (i.e. zero values).",
        "link": "https://blog.soliditylang.org/2022/05/17/calldata-reencode-size-check-
↪bug/",
        "introduced": "0.5.8",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-1",
        "name": "AbiEncodeCallLiteralAsFixedBytesBug",
        "summary": "Literals used for a fixed length bytes parameter in ``abi.
↪encodeCall`` were encoded incorrectly.",
        "description": "For the encoding, the compiler only considered the types of the␣
↪expressions in the second argument of ``abi.encodeCall`` itself, but not the parameter␣
↪types of the function given as first argument. In almost all cases the abi encoding of␣
↪the type of the expression matches the abi encoding of the parameter type of the given␣
↪function. This is because the type checker ensures the expression is implicitly␣
↪convertible to the respective parameter type. However this is not true for number␣
↪literals used for fixed bytes types shorter than 32 bytes, nor for string literals␣
↪used for any fixed bytes type. Number literals were encoded as numbers instead of␣
↪being shifted to become left-aligned. String literals were encoded as dynamically␣
↪sized memory strings instead of being converted to a left-aligned bytes value.",
        "link": "https://blog.soliditylang.org/2022/03/16/encodecall-bug/",
```

(continué en la próxima página)

(proviene de la página anterior)

```
        "introduced": "0.8.11",
        "fixed": "0.8.13",
        "severity": "very low"

    },
    {
        "uid": "SOL-2021-4",
        "name": "UserDefinedValueTypesBug",
        "summary": "User defined value types with underlying type shorter than 32 bytes␣
↪used incorrect storage layout and wasted storage",
        "description": "The compiler did not correctly compute the storage layout of␣
↪user defined value types based on types that are shorter than 32 bytes. It would␣
↪always use a full storage slot for these types, even if the underlying type was␣
↪shorter. This was wasteful and might have problems with tooling or contract upgrades.",
        "link": "https://blog.soliditylang.org/2021/09/29/user-defined-value-types-bug/",
        "introduced": "0.8.8",
        "fixed": "0.8.9",
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-3",
        "name": "SignedImmutables",
        "summary": "Immutable variables of signed integer type shorter than 256 bits can␣
↪lead to values with invalid higher order bits if inline assembly is used.",
        "description": "When immutable variables of signed integer type shorter than 256␣
↪bits are read, their higher order bits were unconditionally set to zero. The correct␣
↪operation would be to sign-extend the value, i.e. set the higher order bits to one if␣
↪the sign bit is one. This sign-extension is performed by Solidity just prior to when␣
↪it matters, i.e. when a value is stored in memory, when it is compared or when a␣
↪division is performed. Because of that, to our knowledge, the only way to access the␣
↪value in its unclean state is by reading it through inline assembly.",
        "link": "https://blog.soliditylang.org/2021/09/29/signed-immutables-bug/",
        "introduced": "0.6.5",
        "fixed": "0.8.9",
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-2",
        "name": "ABIDecodeTwoDimensionalArrayMemory",
        "summary": "If used on memory byte arrays, result of the function ``abi.decode``␣
↪can depend on the contents of memory outside of the actual byte array that is decoded.
↪",
        "description": "The ABI specification uses pointers to data areas for everything␣
↪that is dynamically-sized. When decoding data from memory (instead of calldata), the␣
↪ABI decoder did not properly validate some of these pointers. More specifically, it␣
↪was possible to use large values for the pointers inside arrays such that computing␣
↪the offset resulted in an undetected overflow. This could lead to these pointers␣
↪targeting areas in memory outside of the actual area to be decoded. This way, it was␣
↪possible for ``abi.decode`` to return different values for the same encoded byte array.
↪",
        "link": "https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/",
        "introduced": "0.4.16",
```

(continué en la próxima página)

(proviene de la página anterior)

```
        "fixed": "0.8.4",
        "conditions": {
            "ABIEncoderV2": true
        },
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-1",
        "name": "KeccakCaching",
        "summary": "The bytecode optimizer incorrectly re-used previously evaluated␣
↪Keccak-256 hashes. You are unlikely to be affected if you do not compute Keccak-256␣
↪hashes in inline assembly.",
        "description": "Solidity's bytecode optimizer has a step that can compute Keccak-
↪256 hashes, if the contents of the memory are known during compilation time. This step␣
↪also has a mechanism to determine that two Keccak-256 hashes are equal even if the␣
↪values in memory are not known during compile time. This mechanism had a bug where␣
↪Keccak-256 of the same memory content, but different sizes were considered equal. More␣
↪specifically, ``keccak256(mpos1, length1)`` and ``keccak256(mpos2, length2)`` in some␣
↪cases were considered equal if ``length1`` and ``length2``, when rounded up to nearest␣
↪multiple of 32 were the same, and when the memory contents at ``mpos1`` and ``mpos2``␣
↪can be deduced to be equal. You maybe affected if you compute multiple Keccak-256␣
↪hashes of the same content, but with different lengths inside inline assembly. You are␣
↪unaffected if your code uses ``keccak256`` with a length that is not a compile-time␣
↪constant or if it is always a multiple of 32.",
        "link": "https://blog.soliditylang.org/2021/03/23/keccak-optimizer-bug/",
        "fixed": "0.8.3",
        "conditions": {
            "optimizer": true
        },
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-11",
        "name": "EmptyByteArrayCopy",
        "summary": "Copying an empty byte array (or string) from memory or calldata to␣
↪storage can result in data corruption if the target array's length is increased␣
↪subsequently without storing new data.",
        "description": "The routine that copies byte arrays from memory or calldata to␣
↪storage stores unrelated data from after the source array in the storage slot if the␣
↪source array is empty. If the storage array's length is subsequently increased either␣
↪by using ``.push()`` or by assigning to its ``.length`` attribute (only before 0.6.0),␣
↪the newly created byte array elements will not be zero-initialized, but contain the␣
↪unrelated data. You are not affected if you do not assign to ``.length`` and do not␣
↪use ``.push()`` on byte arrays, or only use ``.push(<arg>)`` or manually initialize␣
↪the new elements.",
        "link": "https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/",
        "fixed": "0.7.4",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-10",
        "name": "DynamicArrayCleanup",
```

(continué en la próxima página)

(proviene de la página anterior)

```
        "summary": "When assigning a dynamically-sized array with types of size at most␣
↪16 bytes in storage causing the assigned array to shrink, some parts of deleted slots␣
↪were not zeroed out.",
        "description": "Consider a dynamically-sized array in storage whose base-type is␣
↪small enough such that multiple values can be packed into a single slot, such as␣
↪`uint128[]`. Let us define its length to be `l`. When this array gets assigned from␣
↪another array with a smaller length, say `m`, the slots between elements `m` and `l`␣
↪have to be cleaned by zeroing them out. However, this cleaning was not performed␣
↪properly. Specifically, after the slot corresponding to `m`, only the first packed␣
↪value was cleaned up. If this array gets resized to a length larger than `m`, the␣
↪indices corresponding to the unclean parts of the slot contained the original value,␣
↪instead of 0. The resizing here is performed by assigning to the array `length`, by a␣
↪`push()` or via inline assembly. You are not affected if you are only using `.push(
↪<arg>)` or if you assign a value (even zero) to the new elements after increasing the␣
↪length of the array.",
        "link": "https://blog.soliditylang.org/2020/10/07/solidity-dynamic-array-cleanup-
↪bug/",
        "fixed": "0.7.3",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-9",
        "name": "FreeFunctionRedefinition",
        "summary": "The compiler does not flag an error when two or more free functions␣
↪with the same name and parameter types are defined in a source unit or when an␣
↪imported free function alias shadows another free function with a different name but␣
↪identical parameter types.",
        "description": "In contrast to functions defined inside contracts, free␣
↪functions with identical names and parameter types did not create an error. Both␣
↪definition of free functions with identical name and parameter types and an imported␣
↪free function with an alias that shadows another function with a different name but␣
↪identical parameter types were permitted due to which a call to either the multiply␣
↪defined free function or the imported free function alias within a contract led to the␣
↪execution of that free function which was defined first within the source unit.␣
↪Subsequently defined identical free function definitions were silently ignored and␣
↪their code generation was skipped.",
        "introduced": "0.7.1",
        "fixed": "0.7.2",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-8",
        "name": "UsingForCalldata",
        "summary": "Function calls to internal library functions with calldata␣
↪parameters called via ``using for`` can result in invalid data being read.",
        "description": "Function calls to internal library functions using the ``using␣
↪for`` mechanism copied all calldata parameters to memory first and passed them on like␣
↪that, regardless of whether it was an internal or an external call. Due to that, the␣
↪called function would receive a memory pointer that is interpreted as a calldata␣
↪pointer. Since dynamically sized arrays are passed using two stack slots for calldata,␣
↪but only one for memory, this can lead to stack corruption. An affected library call␣
↪will consider the JUMPDEST to which it is supposed to return as part of its arguments␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪and will instead jump out to whatever was on the stack before the call.",
        "introduced": "0.6.9",
        "fixed": "0.6.10",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-7",
        "name": "MissingEscapingInFormatting",
        "summary": "String literals containing double backslash characters passed␣
↪directly to external or encoding function calls can lead to a different string being␣
↪used when ABIEncoderV2 is enabled.",
        "description": "When ABIEncoderV2 is enabled, string literals passed directly to␣
↪encoding functions or external function calls are stored as strings in the intemediate␣
↪code. Characters outside the printable range are handled correctly, but backslashes␣
↪are not escaped in this procedure. This leads to double backslashes being reduced to␣
↪single backslashes and consequently re-interpreted as escapes potentially resulting in␣
↪a different string being encoded.",
        "introduced": "0.5.14",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-6",
        "name": "ArraySliceDynamicallyEncodedBaseType",
        "summary": "Accessing array slices of arrays with dynamically encoded base types␣
↪(e.g. multi-dimensional arrays) can result in invalid data being read.",
        "description": "For arrays with dynamically sized base types, index range␣
↪accesses that use a start expression that is non-zero will result in invalid array␣
↪slices. Any index access to such array slices will result in data being read from␣
↪incorrect calldata offsets. Array slices are only supported for dynamic calldata types␣
↪and all problematic type require ABIEncoderV2 to be enabled.",
        "introduced": "0.6.0",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-5",
        "name": "ImplicitConstructorCallvalueCheck",
        "summary": "The creation code of a contract that does not define a constructor␣
↪but has a base that does define a constructor did not revert for calls with non-zero␣
↪value.",
        "description": "Starting from Solidity 0.4.5 the creation code of contracts␣
↪without explicit payable constructor is supposed to contain a callvalue check that␣
↪results in contract creation reverting, if non-zero value is passed. However, this␣
↪check was missing in case no explicit constructor was defined in a contract at all,␣
↪but the contract has a base that does define a constructor. In these cases it is␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪possible to send value in a contract creation transaction or using inline assembly␣
↪without revert, even though the creation code is supposed to be non-payable.",
        "introduced": "0.4.5",
        "fixed": "0.6.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-4",
        "name": "TupleAssignmentMultiStackSlotComponents",
        "summary": "Tuple assignments with components that occupy several stack slots, i.
↪e. nested tuples, pointers to external functions or references to dynamically sized␣
↪calldata arrays, can result in invalid values.",
        "description": "Tuple assignments did not correctly account for tuple components␣
↪that occupy multiple stack slots in case the number of stack slots differs between␣
↪left-hand-side and right-hand-side. This can either happen in the presence of nested␣
↪tuples or if the right-hand-side contains external function pointers or references to␣
↪dynamic calldata arrays, while the left-hand-side contains an omission.",
        "introduced": "0.1.6",
        "fixed": "0.6.6",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-3",
        "name": "MemoryArrayCreationOverflow",
        "summary": "The creation of very large memory arrays can result in overlapping␣
↪memory regions and thus memory corruption.",
        "description": "No runtime overflow checks were performed for the length of␣
↪memory arrays during creation. In cases for which the memory size of an array in bytes,
↪ i.e. the array length times 32, is larger than 2^256-1, the memory allocation will␣
↪overflow, potentially resulting in overlapping memory areas. The length of the array␣
↪is still stored correctly, so copying or iterating over such an array will result in␣
↪out-of-gas.",
        "link": "https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-bug/",
        "introduced": "0.2.0",
        "fixed": "0.6.5",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and continue␣
↪and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared inside␣
↪a for loop if they can be removed in a control-flow branch that ends with ``break`` or␣
↪``continue`` even though they cannot be removed in other control-flow branches.␣
↪Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.6.0",
        "fixed": "0.6.1",
```

(continué en la próxima página)

(proviene de la página anterior)

```
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2020-2",
        "name": "privateCanBeOverridden",
        "summary": "Private methods can be overridden by inheriting contracts.",
        "description": "While private methods of base contracts are not visible and␣
↪cannot be called directly from the derived contract, it is still possible to declare a␣
↪function of the same name and type and thus change the behaviour of the base contract
↪'s function.",
        "introduced": "0.3.0",
        "fixed": "0.5.17",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue0.5",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and continue␣
↪and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared inside␣
↪a for loop if they can be removed in a control-flow branch that ends with ``break`` or␣
↪``continue`` even though they cannot be removed in other control-flow branches.␣
↪Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.5.8",
        "fixed": "0.5.16",
        "severity": "low",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2019-10",
        "name": "ABIEncoderV2LoopYulOptimizer",
        "summary": "If both the experimental ABIEncoderV2 and the experimental Yul␣
↪optimizer are activated, one component of the Yul optimizer may reuse data in memory␣
↪that has been changed in the meantime.",
        "description": "The Yul optimizer incorrectly replaces ``mload`` and ``sload``␣
↪calls with values that have been previously written to the load location (and␣
↪potentially changed in the meantime) if all of the following conditions are met: (1)␣
↪there is a matching ``mstore`` or ``sstore`` call before; (2) the contents of memory␣
↪or storage is only changed in a function that is called (directly or indirectly) in␣
↪between the first store and the load call; (3) called function contains a for loop␣
↪where the same memory location is changed in the condition or the post or body block.␣
↪When used in Solidity mode, this can only happen if the experimental ABIEncoderV2 is␣
↪activated and the experimental Yul optimizer has been activated manually in addition␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪to the regular optimizer in the compiler settings.",
        "introduced": "0.5.14",
        "fixed": "0.5.15",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true,
            "optimizer": true,
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2019-9",
        "name":
↪"ABIEncoderV2CalldataStructsWithStaticallySizedAndDynamicallyEncodedMembers",
        "summary": "Reading from calldata structs that contain dynamically encoded, but␣
↪statically-sized members can result in incorrect values.",
        "description": "When a calldata struct contains a dynamically encoded, but␣
↪statically-sized member, the offsets for all subsequent struct members are calculated␣
↪incorrectly. All reads from such members will result in invalid values. Only calldata␣
↪structs are affected, i.e. this occurs in external functions with such structs as␣
↪argument. Using affected structs in storage or memory or as arguments to public␣
↪functions on the other hand works correctly.",
        "introduced": "0.5.6",
        "fixed": "0.5.11",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-8",
        "name": "SignedArrayStorageCopy",
        "summary": "Assigning an array of signed integers to a storage array of␣
↪different type can lead to data corruption in that array.",
        "description": "In two's complement, negative integers have their higher order␣
↪bits set. In order to fit into a shared storage slot, these have to be set to zero.␣
↪When a conversion is done at the same time, the bits to set to zero were incorrectly␣
↪determined from the source and not the target type. This means that such copy␣
↪operations can lead to incorrect values being stored.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.7",
        "fixed": "0.5.10",
        "severity": "low/medium"
    },
    {
        "uid": "SOL-2019-7",
        "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
        "summary": "Storage arrays containing structs or other statically-sized arrays␣
↪are not read properly when directly encoded in external function calls or in abi.
↪encode*.",
        "description": "When storage arrays whose elements occupy more than a single␣
↪storage slot are directly encoded in external function calls or using abi.encode*,␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪their elements are read in an overlapping manner, i.e. the element pointer is not␣
↪properly advanced between reads. This is not a problem when the storage data is first␣
↪copied to a memory variable or if the storage array only contains value types or␣
↪dynamically-sized arrays.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.16",
        "fixed": "0.5.10",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-6",
        "name": "DynamicConstructorArgumentsClippedABIV2",
        "summary": "A contract's constructor that takes structs or arrays that contain␣
↪dynamically-sized arrays reverts or decodes to invalid data.",
        "description": "During construction of a contract, constructor parameters are␣
↪copied from the code section to memory for decoding. The amount of bytes to copy was␣
↪calculated incorrectly in case all parameters are statically-sized but contain␣
↪dynamically-sized arrays as struct members or inner arrays. Such types are only␣
↪available if ABIEncoderV2 is activated.",
        "introduced": "0.4.16",
        "fixed": "0.5.9",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special jump␣
↪target only considered the situation after deployment.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor_0.4.x",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special jump␣
↪target only considered the situation after deployment.",
        "introduced": "0.4.5",
```

(continué en la próxima página)

(proviene de la página anterior)

```
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries",
        "summary": "Contract types used in events in libraries cause an incorrect event␣
↪signature hash",
        "description": "Instead of using the type `address` in the hashed signature, the␣
↪actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries_0.4.x",
        "summary": "Contract types used in events in libraries cause an incorrect event␣
↪signature hash",
        "description": "Instead of using the type `address` in the hashed signature, the␣
↪actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.3.0",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but can␣
↪also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.0",
        "fixed": "0.5.7",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage_0.4.x",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but can␣
↪also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.4.19",
        "fixed": "0.4.26",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-2",
        "name": "IncorrectByteInstructionOptimization",
        "summary": "The optimizer incorrectly handles byte opcodes whose second argument␣
↪is 31 or a constant expression that evaluates to 31. This can result in unexpected␣
↪values.",
        "description": "The optimizer incorrectly handles byte opcodes that use the␣
↪constant 31 as second argument. This can happen when performing index access on␣
↪bytesNN types with a compile-time constant value (not index) of 31 or when using the␣
↪byte opcode in inline assembly.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.7",
        "severity": "very low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2019-1",
        "name": "DoubleShiftSizeOverflow",
        "summary": "Double bitwise shifts by large constants whose sum overflows 256␣
↪bits can result in unexpected values.",
        "description": "Nested logical shift operations whose total shift size is 2**256␣
↪or more are incorrectly optimized. This only applies to shifts by numbers of bits that␣
↪are compile-time constant expressions.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.6",
        "severity": "low",
        "conditions": {
            "optimizer": true,
            "evmVersion": ">=constantinople"
        }
    },
    {
        "uid": "SOL-2018-4",
        "name": "ExpExponentCleanup",
        "summary": "Using the ** operator with an exponent of type shorter than 256 bits␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪can result in unexpected values.",
        "description": "Higher order bits in the exponent are not properly cleaned␣
↪before the EXP opcode is applied if the type of the exponent expression is smaller␣
↪than 256 bits and not smaller than the type of the base. In that case, the result␣
↪might be larger than expected if the exponent is assumed to lie within the value range␣
↪of the type. Literal numbers as exponents are unaffected as are exponents or bases of␣
↪type uint256.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "fixed": "0.4.25",
        "severity": "medium/high",
        "check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
    },
    {
        "uid": "SOL-2018-3",
        "name": "EventStructWrongData",
        "summary": "Using structs in events logged wrong data.",
        "description": "If a struct is used in an event, the address of the struct is␣
↪logged instead of the actual data.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.4.17",
        "fixed": "0.4.25",
        "severity": "very low",
        "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')]..[?
↪(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
↪'struct'))]"}
    },
    {
        "uid": "SOL-2018-2",
        "name": "NestedArrayFunctionCallDecoder",
        "summary": "Calling functions that return multi-dimensional fixed-size arrays␣
↪can result in memory corruption.",
        "description": "If Solidity code calls a function that returns a multi-
↪dimensional fixed-size array, array elements are incorrectly interpreted as memory␣
↪pointers and thus can cause memory corruption if the return values are accessed.␣
↪Calling functions with multi-dimensional fixed-size arrays is unaffected as is␣
↪returning fixed-size arrays from function calls. The regular expression only checks if␣
↪such functions are present, not if they are called, which is required for the contract␣
↪to be affected.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.1.4",
        "fixed": "0.4.22",
        "severity": "medium",
        "check": {"regex-source": "returns[^;{]*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\]\\
↪s*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\][^{;]*[;{]"}
    },
    {
        "uid": "SOL-2018-1",
        "name": "OneOfTwoConstructorsSkipped",
        "summary": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored.",
        "description": "If a contract has both a new-style constructor (using the␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored. There will be a compiler␣
↪warning about the old-style constructor, so contracts only using new-style␣
↪constructors are fine.",
        "introduced": "0.4.22",
        "fixed": "0.4.23",
        "severity": "very low"
    },
    {
        "uid": "SOL-2017-5",
        "name": "ZeroFunctionSelector",
        "summary": "It is possible to craft the name of a function such that it is␣
↪executed instead of the fallback function in very specific circumstances.",
        "description": "If a function has a selector consisting only of zeros, is␣
↪payable and part of a contract that does not have a fallback function and at most five␣
↪external functions in total, this function is called instead of the fallback function␣
↪if Ether is sent to the contract without data.",
        "fixed": "0.4.18",
        "severity": "very low"
    },
    {
        "uid": "SOL-2017-4",
        "name": "DelegateCallReturnValue",
        "summary": "The low-level .delegatecall() does not return the execution outcome,␣
↪but converts the value returned by the functioned called to a boolean instead.",
        "description": "The return value of the low-level .delegatecall() function is␣
↪taken from a position in memory, where the call data or the return data resides. This␣
↪value is interpreted as a boolean and put onto the stack. This means if the called␣
↪function returns at least 32 zero bytes, .delegatecall() returns false even if the␣
↪call was successful.",
        "introduced": "0.3.0",
        "fixed": "0.4.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2017-3",
        "name": "ECRecoverMalformedInput",
        "summary": "The ecrecover() builtin can return garbage for malformed input.",
        "description": "The ecrecover precompile does not properly signal failure for␣
↪malformed input (especially in the 'v' argument) and thus the Solidity function can␣
↪return data that was previously present in the return area in memory.",
        "fixed": "0.4.14",
        "severity": "medium"
    },
    {
        "uid": "SOL-2017-2",
        "name": "SkipEmptyStringLiteral",
        "summary": "If \"\" is used in a function call, the following function arguments␣
↪will not be correctly passed to the function.",
        "description": "If the empty string literal \"\" is used as an argument in a␣
↪function call, it is skipped by the encoder. This has the effect that the encoding of␣
↪all arguments following this is shifted left by 32 bytes and thus the function call␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪data is corrupted.",
        "fixed": "0.4.12",
        "severity": "low"
    },
    {
        "uid": "SOL-2017-1",
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode by␣
↪routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.soliditylang.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-11",
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying memory,
↪ ignored its return value. On the public chain, calls to the identity precompile can␣
↪be made in a way that they never fail, but this might be different on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "uid": "SOL-2016-10",
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At␣
↪jump destinations, multiple code paths join and thus it has to compute a common state␣
↪from the incoming edges. Computing this common state was simplified to just use the␣
↪empty state, but this implementation was not done properly. This bug can cause data␣
↪corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-9",
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly and␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same 32␣
↪byte storage slot, but storage writes always write 32 bytes. For some types, the␣
↪higher order bytes were not cleaned properly, which made it sometimes possible to␣
↪overwrite a variable in storage when writing to another one.",
        "link": "https://blog.soliditylang.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "uid": "SOL-2016-8",
        "name": "OptimizerStaleKnowledgeAboutSHA3",
        "summary": "The optimizer did not properly reset its knowledge about SHA3␣
↪operations resulting in some hashes (also used for storage variable positions) not␣
↪being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-
↪evaluating expressions whose value is already known. This knowledge was not properly␣
↪reset across control flow paths and thus the optimizer sometimes thought that the␣
↪result of a SHA3 operation is already present on the stack. This could result in data␣
↪corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-7",
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that␣
↪received Ether.",
        "description": "Library functions are protected against sending them Ether␣
↪through a call. Since the DELEGATECALL opcode forwards the information about how much␣
↪Ether was sent with a call, the library function incorrectly assumed that Ether was␣
↪sent to the library and threw an exception.",
        "severity": "low",
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "uid": "SOL-2016-6",
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if no␣
↪Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a␣
↪certain amount of gas from the EVM to handle the transfer. In the case of a zero-
↪transfer, this gas is not provided which causes the recipient to throw an exception.",
        "severity": "low",
        "fixed": "0.4.0"
    },
```

(continué en la próxima página)

(proviene de la página anterior)

```
    {
        "uid": "SOL-2016-5",
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite loop␣
↪and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this length␣
↪is zero, code was generated that did not terminate and thus consumed all gas.",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "uid": "SOL-2016-4",
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At␣
↪jump destinations, multiple code paths join and thus it has to compute a common state␣
↪from the incoming edges. Computing this common state was not done correctly. This bug␣
↪can cause data corruption, but it is probably quite hard to use for targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-3",
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned before␣
↪comparison.",
        "description": "Two variables of type bytesNN were considered different if their␣
↪higher order bits, which are not part of the actual value, were different. An attacker␣
↪might use this to reach seemingly unreachable code paths by providing incorrectly␣
↪formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "uid": "SOL-2016-2",
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32 bytes␣
↪did not correctly clean the higher order bits, causing corruption in other array␣
↪elements.",
        "description": "Multiple elements of an array of values that are shorter than 17␣
↪bytes are packed into the same storage slot. Writing to a single element of such an␣
↪array did not properly clean the higher order bytes and thus could lead to data␣
↪corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "uid": "SOL-2016-1",
```

(continué en la próxima página)

(proviene de la página anterior)

```
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several␣
↪undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting␣
↪from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
]
```

## 3.37 Contributing

Help is always welcome and there are plenty of options to contribute to Solidity.

In particular, we appreciate support in the following areas:

- Reporting issues.
- Fixing and responding to Solidity's GitHub issues, especially those tagged as «good first issue» which are meant as introductory issues for external contributors.
- Improving the documentation.
- Translating the documentation into more languages.
- Responding to questions from other users on StackExchange and the Solidity Gitter Chat.
- Getting involved in the language design process by proposing language changes or new features in the Solidity forum and providing feedback.

To get started, you can try building-from-source in order to familiarize yourself with the components of Solidity and the build process. Also, it may be useful to become well-versed at writing smart-contracts in Solidity.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project — in the issues, pull requests, or Gitter channels — you agree to abide by its terms.

### 3.37.1 Team Calls

If you have issues or pull requests to discuss, or are interested in hearing what the team and contributors are working on, you can join our public team calls:

- Mondays and Wednesdays at 3PM CET/CEST.

Both calls take place on Jitsi.

### 3.37.2 How to Report Issues

To report an issue, please use the GitHub issues tracker. When reporting issues, please mention the following details:

- Solidity version.
- Source code (if applicable).
- Operating system.
- Steps to reproduce the issue.
- Actual vs. expected behaviour.

Reducing the source code that caused the issue to a bare minimum is always very helpful, and sometimes even clarifies a misunderstanding.

For technical discussions about language design, a post in the Solidity forum is the correct place (see *Solidity Language Design*).

### 3.37.3 Workflow for Pull Requests

In order to contribute, please fork off of the `develop` branch and make your changes there. Your commit messages should detail *why* you made your change in addition to *what* you did (unless it is a tiny change).

If you need to pull in any changes from `develop` after making your fork (for example, to resolve potential merge conflicts), please avoid using `git merge` and instead, `git rebase` your branch. This will help us review your change more easily.

Additionally, if you are writing a new feature, please ensure you add appropriate test cases under `test/` (see below).

However, if you are making a larger change, please consult with the Solidity Development Gitter channel (different from the one mentioned above — this one is focused on compiler and language development instead of language usage) first.

New features and bugfixes should be added to the `Changelog.md` file: please follow the style of previous entries, when applicable.

Finally, please make sure you respect the coding style for this project. Also, even though we do CI testing, please test your code and ensure that it builds locally before submitting a pull request.

We highly recommend going through our review checklist before submitting the pull request. We thoroughly review every PR and will help you get it right, but there are many common problems that can be easily avoided, making the review much smoother.

Thank you for your help!

### 3.37.4 Running the Compiler Tests

#### Prerequisites

For running all compiler tests you may want to optionally install a few dependencies (evmone, libz3, and libhera).

On macOS systems, some of the testing scripts expect GNU coreutils to be installed. This can be easiest accomplished using Homebrew: `brew install coreutils`.

On Windows systems, make sure that you have a privilege to create symlinks, otherwise several tests may fail. Administrators should have that privilege, but you may also grant it to other users or enable Developer Mode.

### Running the Tests

Solidity includes different types of tests, most of them bundled into the Boost C++ Test Framework application `soltest`. Running `build/test/soltest` or its wrapper `scripts/soltest.sh` is sufficient for most changes.

The `./scripts/tests.sh` script executes most Solidity tests automatically, including those bundled into the Boost C++ Test Framework application `soltest` (or its wrapper `scripts/soltest.sh`), as well as command line tests and compilation tests.

The test system automatically tries to discover the location of the evmone for running the semantic tests.

The `evmone` library must be located in the `deps` or `deps/lib` directory relative to the current working directory, to its parent or its parent's parent. Alternatively, an explicit location for the `evmone` shared object can be specified via the `ETH_EVMONE` environment variable.

`evmone` is needed mainly for running semantic and gas tests. If you do not have it installed, you can skip these tests by passing the `--no-semantic-tests` flag to `scripts/soltest.sh`.

Running Ewasm tests is disabled by default and can be explicitly enabled via `./scripts/soltest.sh --ewasm` and requires hera to be found by `soltest`. The mechanism for locating the `hera` library is the same as for `evmone`, except that the variable for specifying an explicit location is called `ETH_HERA`.

The `evmone` and `hera` libraries should both end with the file name extension `.so` on Linux, `.dll` on Windows systems and `.dylib` on macOS.

For running SMT tests, the `libz3` library must be installed and locatable by `cmake` during compiler configure stage.

If the `libz3` library is not installed on your system, you should disable the SMT tests by exporting `SMT_FLAGS=--no-smt` before running `./scripts/tests.sh` or running `./scripts/soltest.sh --no-smt`. These tests are `libsolidity/smtCheckerTests` and `libsolidity/smtCheckerTestsJSON`.

---

**Nota:** To get a list of all unit tests run by Soltest, run `./build/test/soltest --list_content=HRF`.

---

For quicker results you can run a subset of, or specific tests.

To run a subset of tests, you can use filters: `./scripts/soltest.sh -t TestSuite/TestName`, where `TestName` can be a wildcard `*`.

Or, for example, to run all the tests for the yul disambiguator: `./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`.

`./build/test/soltest --help` has extensive help on all of the options available.

See especially:

- show_progress (-p) to show test completion,
- run_test (-t) to run specific tests cases, and
- report-level (-r) give a more detailed report.

---

**Nota:** Those working in a Windows environment wanting to run the above basic sets without libz3. Using Git Bash, you use: `./build/test/Release/soltest.exe -- --no-smt`. If you are running this in plain Command Prompt, use `.\build\test\Release\soltest.exe -- --no-smt`.

---

If you want to debug using GDB, make sure you build differently than the «usual». For example, you could run the following command in your `build` folder: .. code-block:: bash

    cmake -DCMAKE_BUILD_TYPE=Debug .. make

This creates symbols so that when you debug a test using the `--debug` flag, you have access to functions and variables in which you can break or print with.

The CI runs additional tests (including `solc-js` and testing third party Solidity frameworks) that require compiling the Emscripten target.

### Writing and Running Syntax Tests

Syntax tests check that the compiler generates the correct error messages for invalid code and properly accepts valid code. They are stored in individual files inside the `tests/libsolidity/syntaxTests` folder. These files must contain annotations, stating the expected result(s) of the respective test. The test suite compiles and checks them against the given expectations.

For example: `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

A syntax test must contain at least the contract under test itself, followed by the separator `// ----`. The comments that follow the separator are used to describe the expected compiler errors or warnings. The number range denotes the location in the source where the error occurred. If you want the contract to compile without any errors or warning you can leave out the separator and the comments that follow it.

In the above example, the state variable `variable` was declared twice, which is not allowed. This results in a `DeclarationError` stating that the identifier was already declared.

The `isoltest` tool is used for these tests and you can find it under `./build/test/tools/`. It is an interactive tool which allows editing of failing contracts using your preferred text editor. Let's try to break this test by removing the second declaration of `variable`:

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

Running `./build/test/tools/isoltest` again results in a test failure:

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
        contract test {
            uint256 variable;
        }

    Expected result:
        DeclarationError: (36-52): Identifier already declared.
    Obtained result:
        Success
```

`isoltest` prints the expected result next to the obtained result, and also provides a way to edit, update or skip the current contract file, or quit the application.

It offers several options for failing tests:

- `edit`: `isoltest` tries to open the contract in an editor so you can adjust it. It either uses the editor given on the command line (as `isoltest --editor /path/to/editor`), in the environment variable `EDITOR` or just `/usr/bin/editor` (in that order).
- `update`: Updates the expectations for contract under test. This updates the annotations by removing unmet expectations and adding missing expectations. The test is then run again.
- `skip`: Skips the execution of this particular test.
- `quit`: Quits `isoltest`.

All of these options apply to the current contract, except `quit` which stops the entire testing process.

Automatically updating the test above changes it to

```
contract test {
    uint256 variable;
}
// ----
```

and re-run the test. It now passes again:

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

**Nota:** Choose a name for the contract file that explains what it tests, e.g. `double_variable_declaration.sol`. Do not put more than one contract into a single file, unless you are testing inheritance or cross-contract calls. Each file should test one aspect of your new feature.

### 3.37.5 Running the Fuzzer via AFL

Fuzzing is a technique that runs programs on more or less random inputs to find exceptional execution states (segmentation faults, exceptions, etc). Modern fuzzers are clever and run a directed search inside the input. We have a specialized binary called `solfuzzer` which takes source code as input and fails whenever it encounters an internal compiler error, segmentation fault or similar, but does not fail if e.g., the code contains an error. This way, fuzzing tools can find internal problems in the compiler.

We mainly use AFL for fuzzing. You need to download and install the AFL packages from your repositories (afl, afl-clang) or build them manually. Next, build Solidity (or just the `solfuzzer` binary) with AFL as your compiler:

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

At this stage, you should be able to see a message similar to the following:

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
```

(continué en la próxima página)

(proviene de la página anterior)

```
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

If the instrumentation messages did not appear, try switching the cmake flags pointing to AFL's clang binaries:

```
# if previously failed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Otherwise, upon execution the fuzzer halts with an error saying binary is not instrumented:

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6920
```

Next, you need some example source files. This makes it much easier for the fuzzer to find errors. You can either copy some files from the syntax tests or extract test files from the documentation or the other tests:

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↪SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs
```

The AFL documentation states that the corpus (the initial input files) should not be too large. The files themselves should not be larger than 1 kB and there should be at most one input file per functionality, so better start with a small number of. There is also a tool called `afl-cmin` that can trim input files that result in similar behaviour of the binary.

Now run the fuzzer (the `-m` extends the size of memory to 60 MB):

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

The fuzzer creates source files that lead to failures in `/tmp/fuzzer_reports`. Often it finds many similar source files that produce the same error. You can use the tool `scripts/uniqueErrors.sh` to filter out the unique errors.

### 3.37.6 Whiskers

*Whiskers* is a string templating system similar to Mustache. It is used by the compiler in various places to aid readability, and thus maintainability and verifiability, of the code.

The syntax comes with a substantial difference to Mustache. The template markers {{ and }} are replaced by < and > in order to aid parsing and avoid conflicts with *Yul* (The symbols < and > are invalid in inline assembly, while { and } are used to delimit blocks). Another limitation is that lists are only resolved one depth and they do not recurse. This may change in the future.

A rough specification is the following:

Any occurrence of `<name>` is replaced by the string-value of the supplied variable `name` without any escaping and without iterated replacements. An area can be delimited by `<#name>...</name>`. It is replaced by as many concatenations of its contents as there were sets of variables supplied to the template system, each time replacing any `<inner>` items by their respective value. Top-level variables can also be used inside such areas.

There are also conditionals of the form `<?name>...<!name>...</name>`, where template replacements continue recursively either in the first or the second segment depending on the value of the boolean parameter `name`. If `<?+name>...<!+name>...</+name>` is used, then the check is whether the string parameter `name` is non-empty.

### 3.37.7 Documentation Style Guide

In the following section you find style recommendations specifically focusing on documentation contributions to Solidity.

#### English Language

Use English, with British English spelling preferred, unless using project or brand names. Try to reduce the usage of local slang and references, making your language as clear to all readers as possible. Below are some references to help:

- Simplified technical English
- International English
- British English spelling

---

**Nota:** While the official Solidity documentation is written in English, there are community contributed *Traducciones* in other languages available. Please refer to the translation guide for information on how to contribute to the community translations.

---

#### Title Case for Headings

Use title case for headings. This means capitalise all principal words in titles, but not articles, conjunctions, and prepositions unless they start the title.

For example, the following are all correct:

- Title Case for Headings.
- For Headings Use Title Case.
- Local and State Variable Names.
- Order of Layout.

### Expand Contractions

Use expanded contractions for words, for example:

- «Do not» instead of «Don't».
- «Can not» instead of «Can't».

### Active and Passive Voice

Active voice is typically recommended for tutorial style documentation as it helps the reader understand who or what is performing a task. However, as the Solidity documentation is a mixture of tutorials and reference content, passive voice is sometimes more applicable.

As a summary:

- Use passive voice for technical reference, for example language definition and internals of the Ethereum VM.
- Use active voice when describing recommendations on how to apply an aspect of Solidity.

For example, the below is in passive voice as it specifies an aspect of Solidity:

> Functions can be declared `pure` in which case they promise not to read from or modify the state.

For example, the below is in active voice as it discusses an application of Solidity:

> When invoking the compiler, you can specify how to discover the first element of a path, and also path prefix remappings.

### Common Terms

- «Function parameters» and «return variables», not input and output parameters.

### Code Examples

A CI process tests all code block formatted code examples that begin with `pragma solidity`, `contract`, `library` or `interface` using the `./test/cmdlineTests.sh` script when you create a PR. If you are adding new code examples, ensure they work and pass tests before creating the PR.

Ensure that all code examples begin with a `pragma` version that spans the largest where the contract code is valid. For example `pragma solidity >=0.4.0 <0.9.0;`.

### Running Documentation Tests

Make sure your contributions pass our documentation tests by running `./docs/docs.sh` that installs dependencies needed for documentation and checks for any problems such as broken links or syntax issues.

### 3.37.8 Solidity Language Design

To actively get involved in the language design process and to share your ideas concerning the future of Solidity, please join the Solidity forum.

The Solidity forum serves as the place to propose and discuss new language features and their implementation in the early stages of ideation or modifications of existing features.

As soon as proposals get more tangible, their implementation will also be discussed in the Solidity GitHub repository in the form of issues.

In addition to the forum and issue discussions, we regularly host language design discussion calls in which selected topics, issues or feature implementations are debated in detail. The invitation to those calls is shared via the forum.

We are also sharing feedback surveys and other content that is relevant to language design in the forum.

If you want to know where the team is standing in terms or implementing new features, you can follow the implementation status in the Solidity Github project. Issues in the design backlog need further specification and will either be discussed in a language design call or in a regular team call. You can see the upcoming changes for the next breaking release by changing from the default branch (*develop*) to the breaking branch.

For ad-hoc cases and questions, you can reach out to us via the Solidity-dev Gitter channel — a dedicated chatroom for conversations around the Solidity compiler and language development.

We are happy to hear your thoughts on how we can improve the language design process to be even more collaborative and transparent.

## 3.38 Guía de Marca de Solidity

Esta guía de marca presenta información sobre la política de marca de Solidity y pautas de uso del logotipo.

### 3.38.1 Marca de Solidity

El lenguaje de programación Solidity es un proyecto comunitario de código abierto gobernado por un equipo central. El equipo central está patrocinado por la Fundación Ethereum.

Este documento tiene como objetivo proporcionar información sobre cómo utilizar de la mejor manera la marca y el logotipo de Solidity.

Le invitamos a leer este documento detenidamente antes de utilizar la marca o el logotipo. ¡Su cooperación es altamente apreciada!

### 3.38.2 Nombre de la Marca de Solidity

«Solidity» debe usarse para referirse únicamente al lenguaje de programación Solidity.

Por favor, no use «Solidity»:

- Para referirse a cualquier otro lenguaje de programación.
- De una manera que sea engañosa o pueda implicar la asociación de módulos, herramientas, documentación u otros recursos no relacionados con el lenguaje de programación Solidity.
- De maneras que confundan a la comunidad, en cuanto a si el lenguaje de programación Solidity es de código abierto y de uso gratuito.

### 3.38.3 Licencia del Logo de Solidity

El logotipo de Solidity se distribuye y cuenta con Creative Commons Attribution 4.0 International License.

Esta es la licencia Creative Commons más permisiva, que permite la reutilización y modificaciones para cualquier propósito.

Usted es libre de:

- **Compatir** — Copiar y redistribuir el material en cualquier medio o formato.
- **Adaptar** — Remezclar, transformar y construir sobre el material para cualquier propósito, incluso comercial.

Bajo los siguientes términos:

- **Atribuir** — Debe otorgar el crédito apropiado, proporcionar un enlace a la licencia e indicar si se realizaron cambios.

Puede hacerlo de cualquier manera razonable, sin embargo, no de alguna manera que sugiera que el equipo central de Solidity lo respalda a usted o su uso.

Cuando utilice el logotipo de Solidity, respete las pautas del logotipo de Solidity.

### 3.38.4 Guías para el Logo de Solidity

*(Haga click derecho en el logo para descargarlo.)*

No se debe:

- Cambiar el tamaño y la forma del logo (no se debe estirar, ni cortar).
- Cambiar los colores del logotipo, a menos que sea absolutamente necesario.

### 3.38.5 Créditos

Este documento se derivó, en parte, de Python Software Foundation Trademark Usage Policy y Rust Media Guide.

## 3.39 Influencias del Lenguaje

Solidity es un lenguaje de llaves, que ha sido influenciado e inspirado por varios lenguajes de programación conocidos.

Solidity está altamente influenciado por C++, sin embargo, también tomó prestados conceptos de lenguajes como Python, JavaScript y otros.

La influencia de C++ se puede ver en la sintaxis de las declaraciones de variables, en bucles «for», el concepto de sobrecarga de funciones, conversiones de tipo implícitas y explícitas, entre otros detalles.

En la etapa más prematura del lenguaje, Solidity solía estar fuertemente influenciado por JavaScript, esto debido al alcance de las variables a nivel de función y al uso de la palabra clave `var`. La influencia de JavaScript se redujo a partir del versión 0.4.0. Ahora, la principal similitud restante con JavaScript es que las funciones se definen usando la palabra clave `function`, y, que al momento de importar, Solidity utiliza una sintaxis y semántica muy similar a la de JavaScript. Dejando de lado los puntos anteriores, Solidity se parece a la mayoría de lenguajes que utilizan llaves. Cabe recalcar, que Solidity ya no cuenta con una gran influencia de JavaScript.

Otra influencia para Solidity fue Python. Los modificadores de Solidity se agregaron tratando de modelar a los decoradores de Python, con una funcionalidad mucho más restringida. Además, la herencia múltiple, la linealización C3 y la palabra clave `super` se tomaron de Python, así como la asignación general y la semántica de copia de los tipos de valor y referencia.

# Índice

## Símbolos



## A



## B



## C

## D



## E



## F

## P



## R



## S

## T



## U



## V



## W



## Y