Solidity Documentation

Version 0.8.12

Ethereum

févr. 04, 2022

Principes de base

Solidity est un langage orienté objet et de haut niveau pour la mise en œuvre de contrats intelligents. Les contrats intelligents sont des programmes qui régissent le comportement des comptes dans l'état Ethereum.

Solidity est un langage d'accolades. Il est influencé par le C++, le Python et le JavaScript, et est conçu pour cibler la machine virtuelle Ethereum (EVM). Vous pouvez trouver plus de détails sur les langages dont Solidity s'est inspiré dans la section sur les *influences linguistiques*.

Solidity est typée statiquement, supporte l'héritage, les bibliothèques et les types complexes définis par l'utilisateur, entre autres caractéristiques.

Avec Solidity, vous pouvez créer des contrats pour des utilisations telles que le vote, le crowdfunding, les enchères à l'aveugle et les portefeuilles à signatures multiples.

Lorsque vous déployez des contrats, vous devez utiliser la dernière version publiée de Solidity. Sauf cas exceptionnel, seule la dernière version reçoit des correctifs de sécurité. En outre, les changements de rupture ainsi que les nouvelles fonctionnalités sont introduites régulièrement. Nous utilisons actuellement un numéro de version 0.y.z pour indiquer ce rythme rapide de changement.

Avertissement : Solidity a récemment publié la version 0.8.x qui a introduit de nombreux changements. Assurez-vous de lire *la liste complète*.

Les idées pour améliorer Solidity ou cette documentation sont toujours les bienvenues, lisez notre *guide des contributeurs* pour plus de détails.

Astuce : Vous pouvez télécharger cette documentation au format PDF, HTML ou Epub en cliquant sur le menu déroulant des versions dans le coin inférieur gauche et en sélectionnant le format de téléchargement préféré.

CHAPITRE 1

Pour commencer

1. Comprendre les bases des contrats intelligents

Si le concept des contrats intelligents est nouveau pour vous, nous vous recommandons de commencer par vous plonger dans la section « Introduction aux contrats intelligents ». dans la section « Introduction aux contrats intelligents », qui couvre :

- *Un exemple simple de smart contract* écrit sous Solidity.
- *Les bases de la blockchain*.
- *La Ethereum Virtual Machine*.

2. Apprenez à connaître Solidity

Une fois que vous êtes habitué aux bases, nous vous recommandons de lire les sections *« Solidity by Example »* et « Description du langage » pour comprendre les concepts fondamentaux du langage.

3. Installer le compilateur Solidity

Il existe plusieurs façons d'installer le compilateur Solidity. Il vous suffit de choisir votre option préférée et de suivre les étapes décrites sur la *installation page*.

Indication : Vous pouvez essayer des exemples de code directement dans votre navigateur grâce à la fonction Remix IDE. Remix est un IDE basé sur un navigateur web qui vous permet d'écrire, de déployer et d'administrer les smart contracts Solidity, sans avoir à sans avoir besoin d'installer Solidity localement.

Avertissement : Comme les humains écrivent des logiciels, ceux-ci peuvent comporter des bugs. Vous devez suivre les meilleures pratiques établies en matière de développement de logiciels lorsque vous écrivez vos contrats intelligents. Cela inclut la révision du code, les tests, les audits et les preuves de correction. Les utilisateurs de contrats intelligents sont parfois plus confiants dans le code que ses auteurs, et les blockchains et les contrats intelligents ont leurs propres problèmes à surveiller. Avant de travailler sur le code de production, assurez-vous de lire la section *Considérations de sécurité*.

4. En savoir plus

Si vous souhaitez en savoir plus sur la création d'applications décentralisées sur Ethereum, le programme Ethereum Developer Resources peut vous aider à trouver de la documentation générale sur Ethereum, ainsi qu'une large sélection de tutoriels, d'outils et de cadres de développement.

Si vous avez des questions, vous pouvez essayer de chercher des réponses ou de les poser sur Ethereum StackExchange, ou sur notre salon Gitter.

CHAPITRE 2

Traductions

Des bénévoles de la communauté aident à traduire cette documentation en plusieurs langues. Leur degré d'exhaustivité et de mise à jour varie. La version anglaise est une référence.

Note : Nous avons récemment mis en place une nouvelle organisation GitHub et un nouveau flux de traduction pour aider à rationaliser les efforts de la communauté. Veuillez vous référer au guide de traduction pour obtenir des informations sur la manière de contribuer aux traductions communautaires en cours.

— Français (en cours)
— Italien (en cours)
— Japonais
— Coréen (en cours)
— Russe (rather outdated)
— Chinois simplifié (en cours)
— Espagnol
— Turc (partiel)

CHAPITRE 3

Contenu

Index des mots-clés, Page de recherche

3.1 Introduction Aux Smart Contracts

3.1.1 Un Simple Smart Contract

Commençons par un exemple de base qui définit la valeur d'une variable et l'expose à l'accès d'autres contrats. Ce n'est pas grave si vous ne comprenez pas tout de suite, nous entrerons dans les détails plus tard.

Exemple de stockage

```
// 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 première ligne vous indique que le code source est sous la licence GPL version 3.0. Les spécificateurs de licence lisibles par machine sont importants dans un contexte où la publication du code source est le défaut.

La ligne suivante spécifie que le code source est écrit pour Solidity version 0.4.16, ou une version plus récente du langage jusqu'à, mais sans inclure, la version 0.9.0. Cela permet de s'assurer que le contrat n'est pas compilable avec une nouvelle version du compilateur (en rupture), où il pourrait se comporter différemment. *Pragmas* sont des instructions courantes pour les compilateurs sur la manière de traiter le code source (par exemple, pragma once).

Un contrat, au sens de Solidity, est une collection de code (ses *fonctions*) et de données (son *état*) qui réside à une adresse spécifique sur la blockchain. La ligne `uint storedData;` déclare une variable d'état appelée `storedData` de type `uint` (*u*nsigned *int*eger de *256* bits). Vous pouvez l'imaginer comme un emplacement unique dans une base de données que vous pouvez interroger et modifier en appelant des fonctions du code qui gère la base de données. Dans cet exemple, le contrat définit les fonctions `set` et `get` qui peuvent être utilisées pour modifier ou récupérer la valeur de la variable.

Pour accéder à un membre (comme une variable d'état) du contrat en cours, vous n'ajoutez généralement pas le préfixe `this.`, vous y accédez directement par son nom. Contrairement à d'autres langages, l'omettre n'est pas seulement une question de style, il en résulte une façon complètement différente d'accéder au membre, mais nous y reviendrons plus tard.

Ce contrat ne fait pas grand-chose pour l'instant, à part (en raison de l'infrastructure construite par Ethereum) permettant à quiconque de stocker un nombre unique qui est accessible par n'importe qui dans le monde sans un moyen (faisable) de vous empêcher de publier ce numéro. N'importe qui pourrait appeler `set` à nouveau avec une valeur différente et écraser votre numéro, mais le numéro est toujours stocké dans l'historique de la blockchain. Plus tard, vous verrez comment vous pouvez imposer des restrictions d'accès afin que vous soyez le seul à pouvoir modifier le numéro.

Avertissement : Soyez prudent lorsque vous utilisez du texte Unicode, car des caractères d'apparence similaire (ou même identiques) peuvent avoir des points de code différents et sont donc codés dans un tableau d'octets différent.

Note : Tous les identifiants (noms de contrats, noms de fonctions et noms de variables) sont limités au jeu de caractères ASCII. Il est possible de stocker des données encodées en UTF-8 dans des variables de type chaîne.

Exemple de sous-monnaie

Le contrat suivant met en œuvre la forme la plus simple d'une crypto-monnaie. Le contrat permet uniquement à son créateur de créer de nouvelles pièces (différents schémas d'émission sont possibles). Tout le monde peut s'envoyer des pièces sans avoir besoin de s'enregistrer avec un nom d'utilisateur et un mot de passe, tout ce dont vous avez besoin est une paire de clés Ethereum.

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

contract Coin {
    // Le mot clé "public" rend les variables
    // accessibles depuis d'autres contrats
    address public minter;
    mapping (address => uint) public balances;

    // Les événements permettent aux clients de réagir à des
    // changements de contrat que vous déclarez
    event Sent(address from, address to, uint amount);

    // Le code du constructeur n'est exécuté que lorsque le contrat
    // est créé
```

(suite sur la page suivante)

(suite de la page précédente)

```
    constructor() {
        minter = msg.sender;
    }

    // Envoie une quantité de pièces nouvellement créées à une adresse.
    // Ne peut être appelé que par le créateur du contrat
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Les erreurs vous permettent de fournir des informations sur
    // pourquoi une opération a échoué. Elles sont renvoyées
    // à l'appelant de la fonction.
    error InsufficientBalance(uint requested, uint available);

    // Envoie un montant de pièces existantes
    // de n'importe quel appelant à une adresse
    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);
    }
}
```

Ce contrat introduit quelques nouveaux concepts, passons-les en revue un par un.

La ligne `address public minter;` déclare une variable d'état de type *address*. Le type `address` est une valeur de 160 bits qui ne permet aucune opération arithmétique. Il convient pour stocker les adresses des contrats, ou un hachage de la moitié publique d'une paire de clés appartenant à comptes externes.

Le mot clé « public » génère automatiquement une fonction qui vous permet d'accéder à la valeur actuelle de la variable d'état depuis l'extérieur du contrat. depuis l'extérieur du contrat. Sans ce mot-clé, les autres contrats n'ont aucun moyen d'accéder à la variable. Le code de la fonction générée par le compilateur est équivalent à ce qui suit (ignorez `external` et `view` pour le moment) :

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

Vous pourriez ajouter vous-même une fonction comme celle ci-dessus, mais vous auriez une fonction et une variable d'état avec le même nom. Vous n'avez pas besoin de le faire, le compilateur s'en charge pour vous.

La ligne suivante, `mapping (adresse => uint) public balances;` crée également une variable d'état publique, mais il s'agit d'un type de données plus complexe. Le type *mapping* fait correspondre les adresses aux :ref :``internes non signés <integers>`.

Les mappings peuvent être vus comme des tableaux de hachage qui sont initialisées virtuellement, de telle sorte que chaque clé possible existe dès le départ et est mise en correspondance avec une valeur dont la représentation par octet est constituée de zéros. Cependant, il n'est pas possible d'obtenir une liste de toutes les clés d'un mappage, ni une liste de toutes les valeurs. Enregistrez ce que vous avez ajouté au mappage, ou utilisez-le dans un contexte où cela n'est pas

nécessaire. Ou encore mieux, gardez une liste, ou utilisez un type de données plus approprié.

La fonction *getter* créée par le mot-clé ``public`"". est plus complexe dans le cas d'un mapping. Elle ressemble à ce qui suit suivante :

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

Vous pouvez utiliser cette fonction pour demander le solde d'un seul compte.

La ligne `event Sent(adresse from, adresse to, uint amount);` déclare un *« événement »*, qui est émis dans la dernière ligne de la fonction `send`. Les clients Ethereum tels que les applications web peuvent écouter ces événements émis sur la blockchain sans trop de coût. Dès que l'événement est émis, l'écouteur reçoit les arguments « from », « to » et « amount », ce qui permet de suivre les transactions.

Pour écouter cet événement, vous pouvez utiliser le code suivant Du code JavaScript, qui utilise web3.js pour créer l'objet du contrat `Coin`, et toute interface utilisateur appelle la fonction `balances` générée automatiquement ci-dessus :

```
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));
    }
})
```

Le *constructeur* est une fonction spéciale qui est exécutée pendant la création du contrat et ne peut pas être appelée par la suite. Dans ce cas, elle stocke de manière permanente l'adresse de la personne qui crée le contrat. La variable `msg` (avec `tx` et `block`) est une *variable globale spéciale* qui contient des propriétés qui permettent d'accéder à la blockchain. `msg.sender` est toujours l'adresse d'où provient l'appel de fonction (externe) actuel.

Les fonctions qui constituent le contrat, et que les utilisateurs et les contrats peuvent appeler sont `mint` et `send`.

La fonction `mint` envoie une quantité de pièces nouvellement créées à une autre adresse. La fonction *require* définit des conditions qui annulent toutes les modifications si elles ne sont pas respectées. Dans cet exemple, `require(msg.sender == minter);` garantit que seul le créateur du contrat peut appeler `mint`. En général, le créateur peut monnayer autant de jetons qu'il le souhaite, mais à un moment donné, cela conduira à un phénomène appelé « overflow ». Notez qu'à cause de l'option par défaut *Checked arithmetic*, la transaction s'inversera si l'expression `balances[receiver] += amount;` déborde, c'est-à-dire lorsque `balances[receiver] + amount` en arithmétique de précision arbitraire est plus grand que la valeur maximale de `uint` (`2**256 - 1`). Ceci est également vrai pour l'instruction `balances[receiver] += amount;` dans la fonction `send`.

Les erreurs vous permettent de fournir plus d'informations à l'appelant sur pourquoi une condition ou une opération a échoué. Les erreurs sont utilisées avec l'instruction *revert statement*. L'instruction revert interrompt et annule sans condition inconditionnellement et annule toutes les modifications, de manière similaire à la fonction `require`, mais elle vous permet également de fournir le nom d'une erreur et des données supplémentaires qui seront fournies à l'appelant (et éventuellement à l'application frontale ou à l'explorateur de blocs) afin qu'un l'application frontale ou l'explorateur de blocs) afin de pouvoir déboguer ou réagir plus facilement à un échec.

La fonction « envoyer » peut être utilisée par n'importe qui (qui possède déjà certaines de ces pièces) pour envoyer un message à un autre utilisateur. qui possède déjà certaines de ces pièces) pour envoyer des pièces à quelqu'un d'autre. Si l'expéditeur n'a pas assez de pièces à envoyer, la condition `if` est évaluée à true. En conséquence, la condition `revert` fera échouer l'opération tout en fournissant à l'expéditeur les détails de l'erreur en utilisant l'erreur « InsufficientBalance ».

Note : Si vous utilisez ce contrat pour envoyer des pièces de monnaie à une adresse, vous ne verrez rien lorsque vous regardez cette adresse sur un explorateur de blockchain, parce que l'enregistrement que vous avez envoyé des pièces et les soldes modifiés sont uniquement stockés dans le stockage de données de ce contrat de pièces particulier. En utilisant des événements, vous pouvez créer un « explorateur de blockchain » qui suit les transactions et les soldes de votre nouvelle pièce, mais vous devez inspecter l'adresse du contrat de la pièce et non les adresses des propriétaires des pièces.

3.1.2 Les bases de la blockchain

Les blockchains en tant que concept ne sont pas trop difficiles à comprendre pour les programmeurs. La raison en est que la plupart des complications (minage, hashing, cryptographie à courbe elliptique, réseaux de pair à pair, etc.) sont juste là pour fournir un certain ensemble de fonctionnalités et de promesses pour la plate-forme. Une fois que vous acceptez ces caractéristiques comme données, vous n'avez pas à vous soucier de la technologie sous-jacente - ou vous n'avez pas à savoir comment le système AWS d'Amazon fonctionne en interne pour pouvoir l'utiliser ?

Transactions

Une blockchain est une base de données transactionnelle partagée à l'échelle mondiale. Cela signifie que tout le monde peut lire les entrées de la base de données simplement en participant au réseau. Si vous voulez modifier quelque chose dans la base de données, vous devez créer ce qu'on appelle une transaction qui doit être acceptée par tous les autres participants. Le mot « transaction » implique que la modification que vous souhaitez effectuer (supposons que vous souhaitiez modifier deux valeurs en même temps) n'est pas effectuée du tout ou est complètement appliquée. En outre, pendant que votre transaction est appliquée à la base de données, aucune autre transaction ne peut la modifier.

À titre d'exemple, imaginez une table qui répertorie les soldes de tous les comptes dans une monnaie électronique. Si un transfert d'un compte à un autre est demandé, la nature transactionnelle de la base de données garantit que si le montant est soustrait d'un compte, il est toujours ajouté à l'autre compte. Si pour pour une raison quelconque, l'ajout du montant au compte cible n'est pas possible, le compte source n'est pas non plus modifié.

En outre, une transaction est toujours signée de manière cryptographique par l'expéditeur (créateur). Cela permet de protéger facilement l'accès à certaines modifications de la base de données. Dans l'exemple de la monnaie électronique, un simple contrôle permet de s'assurer que seule la personne détenant les clés du compte peut transférer de l'argent depuis celui-ci.

Blocs

L'un des principaux obstacles à surmonter est ce que l'on appelle (en termes de bitcoin) une « attaque par double dépense » : Que se passe-t-il si deux transactions existent dans le réseau qui veulent toutes deux vider un compte ? Seule une des transactions peut être valide, généralement celle qui est acceptée en premier. Le problème est que « premier » n'est pas un terme objectif dans un réseau peer-to-peer.

La réponse abstraite à cette question est que vous n'avez pas à vous en soucier. Un ordre globalement accepté des transactions sera sélectionné pour vous, résolvant ainsi le conflit. Les transactions seront regroupées dans ce qu'on appelle un « bloc ». puis elles seront exécutées et distribuées entre tous les nœuds participants. Si deux transactions se contredisent, celle qui arrive en deuxième position sera rejetée et ne fera pas partie du bloc.

Ces blocs forment une séquence linéaire dans le temps et c'est de là que vient le mot « blockchain ». Les blocs sont ajoutés à la chaîne à intervalles assez réguliers. Ethereum, c'est à peu près toutes les 17 secondes.

Dans le cadre du « mécanisme de sélection des ordres » (appelé « minage »), il peut arriver que des blocs soient révoqués de temps en temps, mais seulement à la « pointe » de la chaîne. Plus de blocs sont ajoutés au-dessus d'un bloc

particulier, moins ce bloc a de chances d'être inversé. Il se peut donc que vos transactions soient inversées et même supprimées de la blockchain, mais plus vous attendez, moins cela est probable.

Note : Les transactions ne sont pas garanties d'être incluses dans le bloc suivant ou dans un bloc futur spécifique, puisque ce n'est pas à celui qui soumet une transaction, mais aux mineurs de déterminer dans quel bloc la transaction est incluse.

Si vous souhaitez planifier les appels futurs de votre contrat, vous pouvez utiliser un outil d'automatisation de contrat intelligent ou un service oracle.

3.1.3 La machine virtuelle Ethereum

Vue d'ensemble

La machine virtuelle d'Ethereum ou EVM est l'environnement d'exécution pour les contrats intelligents dans Ethereum. Il n'est pas seulement sandboxé mais complètement isolé, ce qui signifie que le code s'exécutant dans l'EVM n'a pas accès au réseau, au système de fichiers ou à d'autres processus. Les smart contracts ont même un accès limité aux autres smart contracts.

Comptes

Il y a deux sortes de comptes dans Ethereum qui partagent le même espace d'adresse : **Les comptes externes** qui sont contrôlés par paires de clés publiques-privées (c'est-à-dire les humains) et **les comptes de contrat** qui sont contrôlés par le code stocké avec le compte.

L'adresse d'un compte externe est déterminée à partir de de la clé publique, tandis que l'adresse d'un contrat est déterminée au moment où le contrat est créé (elle est dérivée de l'adresse du créateur et du nombre de transactions envoyées depuis cette adresse, le fameux « nonce »).

Que le compte stocke ou non du code, les deux types sont traités de la même manière par l'EVM.

Chaque compte dispose d'une mémoire persistante clé-valeur qui met en correspondance des mots de 256 bits avec des mots de 256 bits, appelés **storage**.

En outre, chaque compte dispose d'un **solde** en Ether (en « Wei » pour être exact, « 1 ether » est « 10**18 wei ») qui peut être modifié en envoyant des transactions qui incluent de l'Ether.

Transactions

Une transaction est un message qui est envoyé d'un compte à un autre compte (qui peut être le même ou vide, voir ci-dessous). Il peut contenir des données binaires (appelées « charge utile ») et de l'Ether.

Si le compte cible contient du code, ce code est exécuté et les données utiles sont fournies comme données d'entrée.

Si le compte cible n'est pas défini (la transaction n'a pas de destinataire ou que le destinataire a la valeur `null`), la transaction crée un **nouveau contrat**. Comme nous l'avons déjà mentionné, l'adresse de ce contrat n'est pas l'adresse zéro mais une adresse dérivée de l'émetteur et de son nombre de transactions envoyées (le « nonce »). La charge utile d'une telle transaction de création de contrat est considérée comme étant bytecode EVM et est exécutée. Les données de sortie de cette exécution sont stockées de façon permanente en tant que code du contrat. Cela signifie que pour créer un contrat, vous n'envoyez pas le code réel du contrat, mais en fait du code qui renvoie ce code lorsqu'il est exécuté.

Note : Pendant qu'un contrat est en cours de création, son code est encore vide. Pour cette raison, vous ne devriez pas faire appel au contrat en cours de construction avant que son constructeur n'ait fini de s'exécuter.

Gas

Lors de sa création, chaque transaction est chargée d'une certaine quantité de **gaz**, dont le but est de limiter la quantité de travail nécessaire pour exécuter la transaction et de payer en même temps pour cette exécution. Pendant que l'EVM exécute la transaction, le gaz est progressivement épuisé selon des règles spécifiques.

Le **prix du gaz** est une valeur fixée par le créateur de la transaction, qui doit payer « prix du gaz * gaz » à l'avance à partir du compte d'envoi. S'il reste du gaz après l'exécution, il est remboursé au créateur de la même manière.

Si le gaz est épuisé à un moment donné (c'est-à-dire qu'il serait négatif), une exception pour épuisement du gaz est déclenchée, ce qui rétablit toutes les modifications apportées à l'état dans la trame d'appel actuelle.

Stockage, mémoire et pile

La machine virtuelle d'Ethereum a trois zones où elle peut stocker des données- stockage, la mémoire et la pile, qui sont expliqués dans les paragraphes suivants.

Chaque compte dispose d'une zone de données appelée **storage**, qui est persistante entre les appels de fonction et les transactions. Le stockage est un magasin clé-valeur qui fait correspondre des mots de 256 bits à des mots de 256 bits. Il n'est pas possible d'énumérer le stockage à partir d'un contrat. Relativement coûteux à lire, et encore plus à initialiser et à modifier le stockage. En raison de ce coût, vous devez limiter ce que vous stockez dans le stockage persistant à ce dont le contrat a besoin pour fonctionner. Stockez les données telles que les calculs dérivés, la mise en cache et les agrégats en dehors du contrat. Un contrat ne peut ni lire ni écrire dans un stockage autre que le sien.

La deuxième zone de données est appelée **memory**, dont un contrat obtient une instance fraîchement effacée pour chaque appel de message. La mémoire est linéaire et peut être adressée au niveau de l'octet, mais la lecture est limitée à une largeur de 256 bits, tandis que l'écriture peuvent avoir une largeur de 8 bits ou de 256 bits. La mémoire est étendue d'un mot (256 bits), lorsqu'on accède (en lecture ou en écriture) à un mot de mémoire qui n'a pas encore été touché (c'est-à-dire à l'intérieur d'un mot). Au moment de l'expansion, le coût en gaz doit être payé. La mémoire est d'autant plus coûteuse qu'elle est grande (elle s'étend de façon quadratique).

L'EVM n'est pas une machine à registre mais une machine à pile. Tous les calculs sont effectués dans une zone de données appelée la **stack**. Sa taille maximale est de 1024 éléments et contient des mots de 256 bits. L'accès à la pile est limitée à l'extrémité supérieure de la manière suivante : Il est possible de copier l'un des 16 éléments les plus élevés au sommet de la pile ou d'échanger l'élément le plus élevé avec l'un des 16 éléments inférieurs. Il est possible de copier l'un des 16 éléments supérieurs au sommet de la pile ou d'échanger l'élément supérieur avec l'un des 16 éléments inférieurs. Toutes les autres opérations prennent les deux (ou un, ou plusieurs, selon l'opération) de la pile et poussent le résultat sur la pile. Bien sûr, il est possible de déplacer les éléments de la pile vers le stockage ou la mémoire afin d'avoir un accès plus profond à la pile, mais il n'est pas possible d'accéder à des éléments arbitraires plus profondément dans la pile sans avoir préalablement retiré le sommet de la pile.

Jeu d'instructions

Le jeu d'instructions de l'EVM est maintenu à un niveau minimal afin d'éviter les implémentations incorrectes ou incohérentes qui pourraient causer des problèmes de consensus. Toutes les instructions opèrent sur le type de données de base, les mots de 256 bits ou les tranches de mémoire (ou autres tableaux d'octets). Les opérations arithmétiques, binaires, logiques et de comparaison habituelles sont présentes. Les sauts conditionnels et inconditionnels sont possibles. En outre, les contrats peuvent accéder aux propriétés pertinentes du bloc actuel comme son numéro et son horodatage.

Pour une liste complète, veuillez consulter la *liste des opcodes* faisant partie de la documentation de l'assemblage en ligne.

Appels de messages

Les contrats peuvent appeler d'autres contrats ou envoyer de l'Ether à des comptes par le biais d'appels de messages. Les appels de messages sont similaires aux transactions, en ce sens qu'ils ont une source, une cible, des données utiles, de l'Ether, du gaz et des données de retour. En fait, chaque transaction consiste en un appel de message de niveau supérieur qui, à son tour, peut créer d'autres appels de message.

Un contrat peut décider quelle quantité de son **gaz** restant doit être envoyée avec l'appel de message interne et combien il souhaite conserver. Si une exception d'épuisement du gaz se produit dans l'appel interne (ou toute autre exception), cela sera signalé par une valeur d'erreur placée sur la pile. Dans ce cas, seul le gaz envoyé avec l'appel est consommé. Dans Solidity, le contrat d'appel provoque par défaut une exception manuelle dans de telles situations, de sorte que les exceptions « s'accumulent » dans la pile.

Comme déjà dit, le contrat appelé (qui peut être le même que l'appelant) recevra une instance de mémoire fraîchement nettoyée et aura accès à la charge utile de l'appel - qui sera fournie dans une zone séparée appelée **calldata**. Après avoir terminé son exécution, il peut retourner des données qui seront stockées à un emplacement dans la mémoire de l'appelant pré-alloué par ce dernier. Tous ces appels sont entièrement synchrones.

Les appels sont **limités** à une profondeur de 1024, ce qui signifie que pour des opérations plus complexes, les boucles doivent être préférées aux appels récursifs. En outre, seuls 63/64ème du gaz peuvent être transmis dans un appel de message, ce qui entraîne une limite de profondeur d'un peu moins de 1000 en pratique.

Delegatecall / Callcode et bibliothèques

Il existe une variante spéciale d'un appel de message, appelée **delegatecall**, qui est identique à un appel de message, à l'exception du fait que le code à l'adresse cible est exécuté dans le contexte du contrat d'appel et appelant et que les valeurs de `msg.sender` et `msg.value` ne changent pas.

Cela signifie qu'un contrat peut charger dynamiquement du code provenant d'une autre différente au moment de l'exécution. Le stockage, l'adresse actuelle et le solde font toujours référence au contrat appelant, seul le code est pris de l'adresse appelée.

Cela permet de mettre en œuvre la fonctionnalité de « bibliothèque » dans Solidity : Un code de bibliothèque réutilisable qui peut être appliqué au stockage d'un contrat, par exemple pour mettre en œuvre une structure de données complexe.

Logs

Il est possible de stocker des données dans une structure de données spécialement indexée qui s'applique jusqu'au niveau du bloc. Cette fonctionnalité appelée **logs** est utilisée par Solidity afin d'implémenter *events*. Les contrats ne peuvent pas accéder aux données des logs après leur création, mais elles peuvent être efficacement accessibles depuis l'extérieur de la blockchain. Puisqu'une partie des données du journal est stockée dans bloom filters, il est possible de rechercher ces données de manière efficace et cryptographique, de sorte que les pairs du réseau qui ne téléchargent pas l'ensemble de la blockchain (appelés « clients légers ») peuvent toujours trouver ces journaux.

Créer

Les contrats peuvent même créer d'autres contrats en utilisant un opcode spécial (c'est-à-dire qu'ils n'appellent pas simplement l'adresse zéro comme le ferait une transaction). La seule différence entre ces appels **create** et les appels de message normaux est que les données utiles sont exécutées et le résultat reçoit l'adresse du nouveau contrat sur la pile.

Désactivation et autodestruction

Le seul moyen de supprimer un code de la blockchain est lorsqu'un contrat à cette adresse effectue l'opération d'"« autodestruction ». L'Ether restant stocké à cette adresse est envoyé à une cible désignée et ensuite le stockage et le code est retiré de l'état. En théorie, supprimer le contrat semble être une bonne idée, mais elle est potentiellement dangereuse, car si quelqu'un envoie de l'Ether à des contrats supprimés, l'Ether est perdu à jamais.

Avertissement : Même si un contrat est supprimé par « autodestruction », il fait toujours partie de l'histoire de la blockchain et probablement conservé par la plupart des nœuds Ethereum. Ainsi, utiliser « l'autodestruction » n'est pas la même chose que de supprimer des données d'un disque dur.

Note : Même si le code d'un contrat ne contient pas d'appel à `selfdestruct``, il peut quand même effectuer cette opération en utilisant ``delegatecall` ou `callcode`.

Si vous voulez désactiver vos contrats, vous devriez plutôt **désactiver** ceux-ci en modifiant un état interne qui entraîne le retour en arrière de toutes les fonctions. Ceci rend impossible l'utilisation du contrat, car il retourne immédiatement de l'Ether.

Contrats précompilés

Il existe un petit ensemble d'adresses de contrat qui sont spéciales : La plage d'adresses comprise entre `1` et (y compris) `8` contient des « contrats précompilés » qui peuvent être appelés comme n'importe quel autre contrat, mais leur comportement (et leur consommation de gaz) n'est pas défini par le code EVM stocké à cette adresse (ils ne contiennent pas de code), mais est plutôt mis en œuvre dans l'environnement d'exécution EVM lui-même.

Différentes chaînes compatibles EVM peuvent utiliser un ensemble différent de contrats précompilés. Il est également possible que de nouveaux contrats précompilés soient ajoutés à la chaîne principale d'Ethereum à l'avenir, mais vous pouvez raisonnablement vous attendre à ce qu'ils soient toujours dans la gamme entre `1` et `0xffff` (inclus).

3.2 Installation du compilateur Solidity

3.2.1 Versionnage

Les versions de Solidity suivent le versionnement sémantique et en plus des versions, **des builds de développement nocturnes** sont également mis à disposition. Les nightly builds ne sont pas garanties et, malgré tous les efforts, elles peuvent contenir et/ou des changements non documentés. Nous recommandons d'utiliser la dernière version. Les installateurs de paquets ci-dessous utiliseront la dernière version.

3.2.2 Remix

Nous recommandons Remix pour les petits contrats et pour apprendre rapidement Solidity.

Access Remix en ligne, vous n'avez pas besoin d'installer quoi que ce soit. Si vous voulez l'utiliser sans connexion à l'Internet, allez sur https://github.com/ethereum/remix-live/tree/gh-pages et téléchargez le fichier `.zip` comme comme expliqué sur cette page. Remix est également une option pratique pour tester les constructions nocturnes sans installer plusieurs versions de Solidity.

D'autres options sur cette page détaillent l'installation du compilateur Solidity en ligne de commande sur votre ordinateur. Choisissez un compilateur en ligne de commande si vous travaillez sur un contrat plus important ou si vous avez besoin de plus d'options de compilation.

3.2.3 npm / Node.js

Utilisez `npm` pour une manière pratique et portable d'installer `solcjs`, un compilateur Solidity. Le programme *solcjs* a moins de fonctionnalités que les façons d'accéder au compilateur décrites plus bas dans cette page. La documentation *Utilisation du compilateur en ligne de commande* suppose que vous utilisez le compilateur complet, `solc`. L'utilisation de `solcjs` est documentée à l'intérieur de son propre repository.

Note : Le projet solc-js est dérivé du projet C++ *solc*. *solc* en utilisant Emscripten ce qui signifie que les deux utilisent le même code source du compilateur. *solc-js* peut être utilisé directement dans des projets JavaScript (comme Remix). Veuillez vous référer au dépôt solc-js pour les instructions.

```
npm install -g solc
```

Note : L'exécutable en ligne de commande est nommé `solcjs`.

Les options en ligne de commande de `solcjs` ne sont pas compatibles avec `solc` et les outils (tels que `geth`) qui attendent le comportement de `solc` ne fonctionneront pas avec `solcjs`.

3.2.4 Docker

Les images Docker des constructions Solidity sont disponibles en utilisant l'image `solc` de l'organisation `ethereum`. Utilisez la balise `stable` pour la dernière version publiée, et `nightly` pour les changements potentiellement instables dans la branche de développement.

L'image Docker exécute l'exécutable du compilateur, vous pouvez donc lui passer tous les arguments du compilateur. Par exemple, la commande ci-dessous récupère la version stable de l'image `solc` (si vous ne l'avez pas déjà), et l'exécute dans un nouveau conteneur, en passant l'argument `--help`.

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

Vous pouvez également spécifier les versions de build de la version dans la balise, par exemple, pour la version 0.5.4.

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

Pour utiliser l'image Docker afin de compiler les fichiers Solidity sur la machine hôte, montez un dossier local pour l'entrée et la sortie, et spécifier le contrat à compiler. Par exemple.

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

Vous pouvez également utiliser l'interface JSON standard (ce qui est recommandé lorsque vous utilisez le compilateur avec des outils). Lors de l'utilisation de cette interface, il n'est pas nécessaire de monter des répertoires tant que l'entrée JSON est autonome (c'est-à-dire qu'il ne fait pas référence à des fichiers externes qui devraient être *chargés par la callback d'importation*).

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

3.2.5 Paquets Linux

Les paquets binaires de Solidity sont disponibles à l'adresse solidity/releases.

Nous avons également des PPA pour Ubuntu, vous pouvez obtenir la dernière version stable en utilisant les commandes suivantes :

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

La version nocturne peut être installée en utilisant ces commandes :

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

Nous publions également un paquet snap, qui est installable dans toutes les distros Linux supportées. Pour installer la dernière version stable de solc :

```
sudo snap install solc
```

Si vous voulez aider à tester la dernière version de développement de Solidity avec les changements les plus récents, veuillez utiliser ce qui suit :

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

Note : Le snap `solc` utilise un confinement strict. Il s'agit du mode le plus sûr pour les paquets snap mais il comporte des limitations, comme l'accès aux seuls fichiers de vos répertoires `/home` et `/media`. Pour plus d'informations, consultez la page Démystifier le confinement de Snap.

Arch Linux dispose également de paquets, bien que limités à la dernière version de développement :

```
pacman -S solidity
```

Gentoo Linux possède un Ethereum overlay qui contient un paquet Solidity. Après la configuration de l'overlay, `solc` peut être installé dans les architectures x86_64 par :

```
emerge dev-lang/solidity
```

3.2.6 Paquets macOS

Nous distribuons le compilateur Solidity via Homebrew comme une version construite à partir des sources. Les bouteilles préconstruites ne sont actuellement pas supportées.

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

Pour installer la plus récente version 0.4.x / 0.5.x de Solidity, vous pouvez également utiliser `brew install solidity@4` et `brew install solidity@5`, respectivement.

Si vous avez besoin d'une version spécifique de Solidity, vous pouvez installer une formule Homebrew directement depuis Github.

Voir solidity.rb commits sur Github.

Copiez le hash de commit de la version que vous voulez et vérifiez-la sur votre machine.

```
git clone https://github.com/ethereum/homebrew-ethereum.git
cd homebrew-ethereum
git checkout <your-hash-goes-here>
```

Installez-le en utilisant `brew` :

```
brew unlink solidity
# eg. Install 0.4.8
brew install solidity.rb
```

3.2.7 Binaires statiques

Nous maintenons un dépôt contenant des constructions statiques des versions passées et actuelles du compilateur pour toutes les plateformes supportées. plates-formes supportées à solc-bin. C'est aussi l'endroit où vous pouvez trouver les nightly builds.

Le dépôt n'est pas seulement un moyen rapide et facile pour les utilisateurs finaux d'obtenir des binaires prêts à l'emploi, mais il est également conçu pour être convivial pour les outils tiers :

- Le contenu est mis en miroir sur https://binaries.soliditylang.org, où il peut être facilement téléchargé via HTTPS sans authentification, ni contrôle. HTTPS sans authentification, limitation de débit ou nécessité d'utiliser git.
- Le contenu est servi avec des en-têtes *Content-Type* corrects et une configuration CORS indulgente afin qu'il puisse être directement chargé par des outils s'exécutant dans le navigateur.
- Les binaires ne nécessitent pas d'installation ou de déballage (à l'exception des anciennes versions de Windows fournies avec les DLL nécessaires).

— Nous nous efforçons d'assurer un haut niveau de compatibilité ascendante. Les fichiers, une fois ajoutés, ne sont pas supprimés ou déplacés sans fournir un lien symbolique/une redirection à l'ancien emplacement. Ils ne sont jamais modifiés non plus en place et doivent toujours correspondre à la somme de contrôle d'origine. La seule exception serait les fichiers cassés ou inutilisables, susceptibles de causer plus de tort que de bien s'ils sont laissés en l'état.
— Les fichiers sont servis à la fois par HTTP et HTTPS. Tant que vous obtenez la liste des fichiers d'une manière sécurisée (via git, HTTPS, IPFS ou simplement en la mettant en cache localement) et que vous vérifiez les hachages des binaires après les avoir téléchargés, vous n'avez pas besoin d'utiliser HTTPS pour les binaires eux-mêmes.

Les mêmes binaires sont dans la plupart des cas disponibles sur la page **`Solidity release page on Github`_**. La différence est que nous ne mettons généralement pas à jour les anciennes versions sur la page Github. Cela signifie que que nous ne les renommons pas si la convention de nommage change et que nous n'ajoutons pas de builds pour les plates-formes qui n'étaient pas supportées au moment de la publication. Ceci n'arrive que dans `solc-bin`.

Le dépôt `solc-bin` contient plusieurs répertoires de haut niveau, chacun représentant une seule plate-forme. Chacun contient un fichier `list.json` listant les binaires disponibles. Par exemple dans `emscripten-wasm32/list.json`, vous trouverez les informations suivantes sur la version 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",
    "dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS"
  ]
}
```

Cela signifie que :

— Vous pouvez trouver le binaire dans le même répertoire sous le nom de solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. Notez que le fichier pourrait être un lien symbolique, et vous devrez le résoudre vous-même si vous n'utilisez pas git pour le télécharger ou si votre système de fichiers ne supporte pas les liens symboliques.
— Le binaire est également mis en miroir à https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. Dans ce cas, git n'est pas nécessaire et les liens symboliques sont résolus de manière transparente, soit en fournissant une copie du fichier ou en renvoyant une redirection HTTP.
— Le fichier est également disponible sur IPFS à l'adresse QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS.
— Le fichier pourrait à l'avenir être disponible sur Swarm à l'adresse 16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1.
— Vous pouvez vérifier l'intégrité du binaire en comparant son hachage keccak256 à `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3`. Le hachage peut être calculé en ligne de commande à l'aide de l'utilitaire `keccak256sum` fourni par sha3sum ou de la fonction *keccak256()* de ethereumjs-util`_ en JavaScript.
— Vous pouvez également vérifier l'intégrité du binaire en comparant son hachage sha256 à `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2`.

Avertissement : En raison de la forte exigence de compatibilité ascendante, le référentiel contient quelques éléments anciens mais vous devriez éviter de les utiliser lorsque vous écrivez de nouveaux outils :

— Utilisez `emscripten-wasm32/` (avec une solution de repli sur `emscripten-asmjs/`) au lieu de `bin/` si vous voulez les meilleures performances. Jusqu'à la version 0.6.1, nous ne fournissions que les binaires

asm.js. À partir de la version 0.6.2, nous sommes passés à des constructions **`WebAssembly`_** avec de bien meilleures performances. Nous avons reconstruit les anciennes versions pour wasm mais les fichiers asm.js originaux restent dans `bin/`. Les nouveaux fichiers ont dû être placés dans un répertoire séparé pour éviter les conflits de noms.
— Utilisez `emscripten-asmjs/` et `emscripten-wasm32/` au lieu des répertoires `bin/` et `wasm/` si vous voulez être sûr que vous téléchargez un binaire wasm ou asm.js.
— Utilisez `list.json` au lieu de `list.js` et `list.txt`. Le format de liste JSON contient toutes les informations des anciens formats et plus encore.
— Utilisez https://binaries.soliditylang.org au lieu de https://solc-bin.ethereum.org. Pour garder les choses simples, nous avons déplacé presque tout ce qui concerne le compilateur sous le nouveau domaine `soliditylang.org`, et cela s'applique aussi à `solc-bin`. Bien que le nouveau domaine soit recommandé, l'ancien domaine est toujours entièrement supporté et garanti pour pointer au même endroit.

Avertissement : Les binaires sont également disponibles à https://ethereum.github.io/solc-bin/ mais cette page a cessé d'être mise à jour juste après la sortie de la version 0.7.2, ne recevra pas de nouvelles versions ou nightly builds pour n'importe quelle plateforme et ne sert pas la nouvelle structure de répertoire, y compris les les constructions non-emscriptées.

Si vous l'utilisez, veuillez basculer vers https://binaries.soliditylang.org, qui est une solution de remplacement. Ceci nous permet d'apporter des changements à l'hébergement sous-jacent de manière transparente et de minimiser les perturbations. Contrairement au domaine `ethereum.github.io`, sur lequel nous n'avons aucun contrôle, ``binaries.github.io`"" est un domaine sur lequel nous n'avons aucun contrôle, « binaries.soliditylang.org » est garanti de fonctionner et de maintenir la même structure d'URL à long terme.

3.2.8 Construire à partir de la source

Conditions préalables - Tous les systèmes d'exploitation

Les éléments suivants sont des dépendances pour toutes les versions de Solidity :

Logiciel	Notes
CMake (version 3.13+)	Générateur de fichiers de construction multiplateforme.
Boost (version 1.77+ sur Windows, 1.65+ sinon)	Librairies C++.
Git	Outil en ligne de commande pour la récupération du code source.
z3 (version 4.8+, Optionnel)	À utiliser avec le vérificateur SMT.
cvc4 (Optionnel)	À utiliser avec le vérificateur SMT.

Note : Les versions de Solidity antérieures à 0.5.10 ne parviennent pas à se lier correctement avec les versions Boost 1.70+. Une solution possible est de renommer temporairement le répertoire `<Chemin d'installation de Boost>/lib/cmake/Boost-1.70.0` avant d'exécuter la commande cmake pour configurer solidity.

A partir de la 0.5.10, la liaison avec Boost 1.70+ devrait fonctionner sans intervention manuelle.

Note : La configuration de construction par défaut requiert une version spécifique de Z3 (la plus récente au moment de la dernière mise à jour du code). Les changements introduits entre les versions de Z3 entraînent souvent des résultats résultats légèrement différents (mais toujours valides). Nos tests SMT ne tiennent pas compte de ces différences et échoueront probablement avec une version différente de celle pour laquelle ils ont été écrits. Cela ne veut pas dire qu'une

compilation utilisant une version différente est défectueuse. Si vous passez l'option `-DSTRICT_Z3_VERSION=OFF` à CMake, vous pouvez construire avec n'importe quelle version qui satisfait aux exigences données dans la table ci-dessus. Si vous faites cela, cependant, n'oubliez pas de passer l'option `--no-smt` à `scripts/tests.sh` pour sauter les tests SMT.

Versions minimales du compilateur

Les compilateurs C++ suivants et leurs versions minimales peuvent construire la base de code Solidity :

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

Conditions préalables - macOS

Pour les builds macOS, assurez-vous que vous avez la dernière version de Xcode installée. Cela contient le compilateur Clang C++, l'" Xcode IDE et d'autres outils qui sont nécessaires à la création d'applications C++ sous OS X. Si vous installez Xcode pour la première fois, ou si vous venez d'installer une nouvelle nouvelle version, vous devrez accepter la licence avant de pouvoir effectuer des des constructions en ligne de commande :

```
sudo xcodebuild -license accept
```

Notre script de construction OS X utilise le gestionnaire de paquets Homebrew <https://brew.sh>`_ pour installer les dépendances externes. Voici comment désinstaller Homebrew, si vous voulez un jour repartir de zéro.

Conditions préalables - Windows

Vous devez installer les dépendances suivantes pour les versions Windows de Solidity :

Logiciel	Notes
Visual Studio 2019 Outils de construction	C++ compiler
Visual Studio 2019 (Optionnel)	Compilateur C++ et environnement de développement.
Boost (version 1.77+)	Librairies C++.

Si vous avez déjà un IDE et que vous avez seulement besoin du compilateur et des bibliothèques, vous pouvez installer Visual Studio 2019 Build Tools.

Visual Studio 2019 fournit à la fois l'IDE et le compilateur et les bibliothèques nécessaires. Donc, si vous n'avez pas d'IDE et que vous préférez développer Solidity, Visual Studio 2019 peut être un choix pour vous afin de tout configurer facilement.

Voici la liste des composants qui doivent être installés dans Visual Studio 2019 Build Tools ou Visual Studio 2019 :

- Fonctions de base de Visual Studio C++
- VC++ 2019 v141 toolset (x86,x64)
- SDK CRT universel Windows
- SDK Windows 8.1
- Support C++/CLI

Nous avons un script d'aide que vous pouvez utiliser pour installer toutes les dépendances externes requises :

```
scripts\install_deps.ps1
```

Ceci installera `boost` et `cmake` dans le sous-répertoire `deps`.

Clonez le référentiel

Pour cloner le code source, exécutez la commande suivante :

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

Si vous voulez aider à développer Solidity, vous devez forker Solidity et ajouter votre fork personnel en tant que second remote :

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

Note : Cette méthode aboutira à une construction preerelease conduisant par exemple à ce qu'un drapeau dans chaque bytecode produit par un tel compilateur. Si vous souhaitez recompiler un compilateur Solidity déjà publié, alors veuillez utiliser le tarball source sur la page de publication github :

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

(et non le « code source » fourni par github).

Construction en ligne de commande

Assurez-vous d'installer les dépendances externes (voir ci-dessus) avant la construction.

Le projet Solidity utilise CMake pour configurer la construction. Vous pourriez vouloir installer ccache pour accélérer les constructions répétées, CMake le récupérera automatiquement. La construction de Solidity est assez similaire sur Linux, macOS et autres Unices :

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

ou encore plus facilement sur Linux et macOS, vous pouvez exécuter :

```
#note: this will install binaries solc and soltest at usr/local/bin
./scripts/build.sh
```

Avertissement : Les versions BSD devraient fonctionner, mais ne sont pas testées par l'équipe Solidity.

Et pour Windows :

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

Si vous voulez utiliser la version de boost installée par `scripts\install_deps.ps1`, vous aurez vous devrez en plus passer `-DBoost_DIR="deps\boost\lib\cmake\Boost-*"` et `-DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded` comme arguments à l'appel à `cmake`.

Cela devrait entraîner la création de **solidity.sln** dans ce répertoire de construction. En double-cliquant sur ce fichier, Visual Studio devrait se lancer. Nous suggérons de construire **Release**, mais toutes les autres configurations fonctionnent.

Alternativement, vous pouvez construire pour Windows sur la ligne de commande, comme ceci :

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

3.2.9 Options CMake

Si vous êtes intéressé par les options CMake disponibles, lancez `cmake .. -LH`.

Solveurs SMT

Solidity peut être construit avec des solveurs SMT et le fera par défaut s'ils sont trouvés dans le système. Chaque solveur peut être désactivé par une option *cmake*.

Note : Dans certains cas, cela peut également être une solution de contournement potentielle pour les échecs de construction.

Dans le dossier de construction, vous pouvez les désactiver, puisqu'ils sont activés par défaut :

```
# disables only Z3 SMT Solver.
cmake .. -DUSE_Z3=OFF

# disables only CVC4 SMT Solver.
cmake .. -DUSE_CVC4=OFF

# disables both Z3 and CVC4
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

3.2.10 La chaîne de version en détail

La chaîne de la version de Solidity contient quatre parties :

- le numéro de version
- l'étiquette de préversion, généralement définie par `development.YYYY.MM.DD` ou `nightly.YYYY.MM.DD`.
- le commit au format `commit.GITHASH`.
- platform, qui comporte un nombre arbitraire d'éléments, contenant des détails sur la plate-forme et le compilateur.

S'il y a des modifications locales, le commit sera postfixé avec `.mod`.

Ces parties sont combinées comme requis par SemVer, où la balise pre-release Solidity est égale à la pre-release SemVer et le commit Solidity et la plateforme combinés constituent les métadonnées de construction SemVer.

Exemple de version : » 0.4.8+commit.60cc1668.Emscripten.clang « .

Exemple de préversion : » 0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang « .

3.2.11 Informations importantes sur les versions

Après la sortie d'une version, le niveau de version du patch est augmenté, car nous supposons que seuls les changements de niveau patch suivent. Lorsque les changements sont fusionnés, la version doit être augmentée en fonction de SemVer et de la gravité de la modification. Enfin, une version est toujours faite avec la version du nightly build actuel, mais sans le spécificateur ``prerelease`".

Exemple :

0. La version 0.4.0 est faite.
1. Le nightly build a une version 0.4.1 à partir de maintenant.

2. Des changements non cassants sont introduits –> pas de changement de version.
3. Un changement de rupture est introduit –> la version passe à 0.5.0.
4. La version 0.5.0 est publiée.

Ce comportement fonctionne bien avec la version *pragma*.

3.3 Solidity par l'exemple

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
    }

    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`.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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);
        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.voted, "You already voted.");

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

        // Forward the delegation as long as
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // `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.");
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[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];
        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++) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            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. Can you think of a better way ?

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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // 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
    ) {
        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();
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // 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
            // 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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
    // 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /// 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)
        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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    /// 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);
        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_);
        _;
    }

    /// 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_) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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)
    {
        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;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /// 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.
        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 authorise 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 authorises 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
↪"); });
```

Note : 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
// 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;
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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)
    {
        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.

Note : 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 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.soliditySHA3(
        ["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.

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;
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;
    }

    /// 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    /// 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
            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));
    }
}
```

Note : 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"],
        ["\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 or 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
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);
        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 Mise en page d'un fichier source Solidity

Les fichiers sources peuvent contenir un nombre arbitraire de *définitions des contrats*, directives d'importation, *directives pragmatiques* et *struct*, *enum*, *function*, *error* et *constant variable* définitions.

3.4.1 Identificateur de licence SPDX

La confiance dans les contrats intelligents peut être mieux établie si leur code source est disponible. Puisque la mise à disposition du code source touche toujours à des problèmes juridiques en ce qui concerne le droit d'auteur, le compilateur Solidity encourage l'utilisation d'identifiants de licence SPDX lisibles par machine. Chaque fichier source doit commencer par un commentaire indiquant sa licence :

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

Le compilateur ne valide pas que la licence fait partie de la liste autorisée par SPDX, mais il inclut la chaîne fournie dans les *métadonnées du code source*.

Si vous ne voulez pas spécifier une licence ou si le code source n'est pas pas open-source, veuillez utiliser la valeur spéciale `UNLICENSED`.

Le fait de fournir ce commentaire ne vous libère bien sûr pas des autres obligations liées à la licence, comme l'obligation de mentionner un en-tête de licence spécifique dans chaque fichier source ou le détenteur du droit d'auteur original.

Le commentaire est reconnu par le compilateur à n'importe quel endroit du fichier, mais il est recommandé de le placer en haut du fichier.

Plus d'informations sur la façon d'utiliser les identifiants de licence SPDX peuvent être trouvées sur le site web de SPDX.

3.4.2 Pragmas

Le mot-clé `pragma` est utilisé pour activer certaines fonctionnalités du compilateur ou des vérifications. Une directive pragma est toujours locale à un fichier source. vous devez ajouter la directive pragma à tous vos fichiers si vous voulez l'activer dans l'ensemble de votre projet. Si vous *import* un autre fichier, la directive pragma de ce fichier ne s'applique pas automatiquement au fichier d'importation.

Pragma de version

Les fichiers sources peuvent (et doivent) être annotés avec un pragma de version pour rejeter la compilation avec de futures versions du compilateur qui pourraient introduire des changements incompatibles. Nous essayons de limiter ces changements au strict minimum et de les introduire de manière à ce que les changements sémantiques nécessitent aussi dans la syntaxe, mais cela n'est pas toujours possible. Pour cette raison, il est toujours une bonne idée de lire le journal des modifications, au moins pour les versions qui contiennent des des changements de rupture. Ces versions ont toujours des versions de la forme `0.x.0` ou `x.0.0`.

Le pragma de version est utilisé comme suit : `pragma solidity ^0.5.2;`

Un fichier source avec la ligne ci-dessus ne compile pas avec un compilateur antérieur à la version 0.5.2, et il ne fonctionne pas non plus avec un compilateur à partir de la version 0.6.0 (cette deuxième condition est ajoutée en utilisant `^`). Parce que il n'y aura pas de changements de rupture jusqu'à la version `0.6.0`, vous pouvez être sûr que votre code compile comme vous l'aviez prévu. La version exacte du compilateur n'est pas fixée, de sorte que les versions de correction de bogues sont toujours possibles.

Il est possible de spécifier des règles plus complexes pour la version du compilateur, celles-ci suivent la même syntaxe que celle utilisée par npm.

Note : L'utilisation du pragma version *ne change pas* la version du compilateur. Il ne permet pas non plus d'activer ou de désactiver des fonctionnalités du compilateur. Il indique simplement au compilateur de vérifier si sa version correspond à celle requise par le pragma. Si elle ne correspond pas, le compilateur émet une une erreur.

Pragma du codeur ABI

En utilisant `pragma abicoder v1` ou `pragma abicoder v2`, vous pouvez choisir entre les deux implémentations du codeur et du décodeur ABI.

Le nouveau codeur ABI (v2) est capable de coder et de décoder tableaux et structs. Il peut produire un code moins optimal et n'a pas été testé autant que l'ancien codeur, mais est considéré comme non expérimental à partir de Solidity 0.6.0. Vous devez toujours explicitement l'activer en utilisant `pragma abicoder v2;`. Puisqu'il sera activé par défaut à partir de Solidity 0.8.0, il existe une option pour sélectionner l'ancien codeur en utilisant `pragma abicoder v1;`.

L'ensemble des types supportés par le nouveau codeur est un sur-ensemble strict de ceux supportés par l'ancien. Les contrats qui l'utilisent peuvent interagir avec ceux qui ne l'utilisent pas sans limitations. L'inverse n'est possible que dans la mesure où le contrat non-`abicoder v2` n'essaie pas de faire des appels qui nécessiteraient de décoder des types uniquement supportés par le nouvel encodeur. Le compilateur peut détecter cela et émettra une erreur. Il suffit d'activer « abicoder v2 » pour votre contrat pour que l'erreur disparaisse.

Note : Ce pragma s'applique à tout le code défini dans le fichier où il est activé, quel que soit l'endroit où ce code se retrouve finalement. Cela signifie qu'un contrat dont le fichier source est sélectionné pour être compilé avec le codeur ABI v1 peut toujours contenir du code qui utilise le nouveau codeur en l'héritant d'un autre contrat. Ceci est autorisé si les nouveaux types sont uniquement utilisés en interne et non dans les signatures de fonctions externes.

Note : Jusqu'à Solidity 0.7.4, il était possible de sélectionner le codeur ABI v2 en utilisant `pragma experimental ABIEncoderV2`, mais il n'était pas possible de sélectionner explicitement le codeur v1 parce qu'il était par défaut.

Pragma expérimental

Le deuxième pragma est le pragma expérimental. Il peut être utilisé pour activer des fonctionnalités du compilateur ou du langage qui ne sont pas encore activées par défaut. Les pragmes expérimentaux suivants sont actuellement supportés :

ABIEncoderV2

Parce que le codeur ABI v2 n'est plus considéré comme expérimental, il peut être sélectionné via `pragma abicoder v2` (voir ci-dessus) depuis Solidity 0.7.4.

SMTChecker

Ce composant doit être activé lorsque le compilateur Solidity est construit, et n'est donc pas disponible dans tous les binaires Solidity. Les *instructions de construction* expliquent comment activer cette option. Elle est activée pour les versions PPA d'Ubuntu dans la plupart des versions, mais pas pour les images Docker, les binaires Windows ou les binaires Linux construits de manière statique. Elle peut être activée pour solc-js via l'option smtCallback si vous avez un solveur SMT installé localement et que vous exécutez solc-js via node (et non via le navigateur).

Si vous utilisez `pragma experimental SMTChecker;`, alors vous obtenez des *avertissements de sécurité* supplémentaires qui sont obtenus en interrogeant un solveur SMT. Ce composant ne prend pas encore en charge toutes les fonctionnalités du langage Solidity et produit probablement de nombreux avertissements. S'il signale des fonctionnalités non supportées, l'analyse n'est peut-être pas entièrement solide.

3.4.3 Importation d'autres fichiers sources

Syntaxe et sémantique

Solidity prend en charge des déclarations d'importation pour aider à modulariser votre code. Ils sont similaires à celles disponibles en JavaScript (à partir de ES6). Cependant, Solidity ne supporte pas le concept de l'"exportation par défaut.

Au niveau global, vous pouvez utiliser des déclarations d'importation de la forme suivante :

```
import "filename";
```

La partie `filename` est appelée un « chemin d'importation ». Cette déclaration importe tous les symboles globaux de « nom de fichier » (et les symboles qui y sont importés) dans la portée globale actuelle (différent de ES6 mais compatible avec Solidity). L'utilisation de cette forme n'est pas recommandée, car elle pollue l'espace de noms de manière imprévisible. Si vous ajoutez de nouveaux éléments de haut niveau à l'intérieur de « filename », ils apparaissent automatiquement dans tous les fichiers qui importent de la sorte depuis « nom de fichier ». Il est préférable d'importer des symboles spécifiques de manière explicite.

L'exemple suivant crée un nouveau symbole global `symbolName` dont les membres sont tous les symboles globaux de « filename ». les symboles globaux de « nom_de_fichier » :

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

ce qui a pour conséquence que tous les symboles globaux sont disponibles dans le format `symbolName.symbol`.

Une variante de cette syntaxe qui ne fait pas partie de ES6, mais qui peut être utile, est la suivante :

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

qui est équivalent à `import * as symbolName from "filename";`.

S'il y a une collision de noms, vous pouvez renommer les symboles pendant l'importation. Par exemple, le code ci-dessous crée de nouveaux symboles globaux `alias` et `symbol2` qui font référence à `symbol1` et `symbole2` à l'intérieur de « filename », respectivement.

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

Importation de chemins

Afin de pouvoir supporter des constructions reproductibles sur toutes les plateformes, le compilateur Solidity doit faire abstraction des détails du système de fichiers dans lequel les fichiers sources sont stockés. Pour cette raison, les chemins d'importation ne se réfèrent pas directement aux fichiers dans le système de fichiers hôte. Au lieu de cela, le compilateur maintient une base de données interne (*système de fichiers virtuel* ou *VFS* en abrégé) dans laquelle chaque unité source se voit attribuer un *nom d'unité source* unique qui est un identifiant opaque et non structuré. Le chemin d'importation spécifié dans une instruction d'importation est traduit en un nom d'unité source et utilisé pour trouver l'unité source correspondante dans cette base de données.

En utilisant l'API *Standard JSON*, il est possible de fournir directement les noms et le contenu de tous les fichiers sources comme une partie de l'entrée du compilateur. Dans ce cas, les noms des unités sources sont vraiment arbitraires. Si, par contre, vous voulez que le compilateur trouve et charge automatiquement le code source dans le VFS, vos noms d'unité source doivent être structurés de manière à rendre possible un *import callback* de les localiser. Lorsque vous utilisez le compilateur en ligne de commande, le callback d'importation par défaut ne supporte que le chargement du code source depuis le système de fichiers de l'hôte, ce qui signifie que les noms de vos unités sources doivent être des chemins. Certains environnements fournissent des callbacks personnalisés qui sont plus polyvalents. Par exemple l'IDE Remix en fournit une qui vous permet d'importer des fichiers à partir d'URL HTTP, IPFS et Swarm ou de vous référer directement à des paquets dans le registre NPM..

Pour une description complète du système de fichiers virtuel et de la logique de résolution de chemin utilisée par le compilateur, voir *Résolution de chemin*.

3.4.4 Commentaires

Les commentaires d'une seule ligne (`//`) et les commentaires de plusieurs lignes (`/*...*/`) sont possibles.

```
// Il s'agit d'un commentaire d'une seule ligne.

/*
Ceci est un
commentaire de plusieurs lignes.
*/
```

Note : Un commentaire d'une seule ligne est terminé par n'importe quel terminateur de ligne unicode (LF, VF, FF, CR, NEL, LS ou PS) en codage UTF-8. Le terminateur fait toujours partie du code source après le commentaire, donc s'il ne s'agit pas d'un symbole ASCII (il s'agit de NEL, LS et PS), cela entraînera une erreur d'analyse syntaxique.

En outre, il existe un autre type de commentaire appelé commentaire NatSpec, qui est détaillé dans le guide de style. Ils sont écrits avec une triple barre oblique (`///`) ou un double astérisque (`/** ... */`). Ils doivent être utilisés directement au-dessus des déclarations de fonctions ou des instructions.

3.5 Structure d'un contrat

Les contrats dans Solidity sont similaires aux classes dans les langages orientés objet. Chaque contrat peut contenir des déclarations de *Variables d'état*, *Fonctions*, *Modificateurs de fonction*, *Événements*, *Erreurs*, structure-structure-types et *Types d'Enum*. De plus, les contrats peuvent hériter d'autres contrats.

Il existe également des types spéciaux de contrats appelés *libraries* et *interfaces*.

La section sur les contrats contient plus de détails que cette section, qui sert à donner un aperçu rapide.

3.5.1 Variables d'état

Les variables d'état sont des variables dont les valeurs sont stockées de manière permanente dans le contrat stockage.

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

contract SimpleStorage {
    uint storedData; // Variable d'état
    // ...
}
```

Voir la section *Types* pour les types de variables d'état valides et la section *Visibility and Getters* pour les choix possibles en matière de visibilité.

3.5.2 Fonctions

Les fonctions sont les unités exécutables du code. Les fonctions sont généralement définies à l'intérieur d'un contrat, mais elles peuvent aussi être définies en dehors des contrats.

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

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

// Fonction d'aide définie en dehors d'un contrat
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

Appels de fonction peut se produire en interne ou en externe et avoir différents niveaux de *visibilité* vers d'autres contrats. *Les fonctions* acceptent les *paramètres et variables de retour* pour passer des paramètres et des valeurs entre elles.

3.5.3 Modificateurs de fonction

Les modificateurs de fonctions peuvent être utilisés pour modifier la sémantique des fonctions de manière déclarative (voir *Function Modifiers* dans la section sur les contrats).

La surcharge, c'est-à-dire le fait d'avoir le même nom de modificateur avec différents paramètres, n'est pas possible.

Comme les fonctions, les modificateurs peuvent être *overridden*.

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

contract Purchase {
    address public seller;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    modifier onlySeller() { // Modificateur
        require(
            msg.sender == seller,
            "Seul le vendeur peut l'appeler."
        );
        _;
    }

    function abort() public view onlySeller { // Utilisation des modificateurs
        // ...
    }
}
```

3.5.4 Événements

Les événements sont des interfaces pratiques avec les fonctions de journalisation de l'EVM.

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

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

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // Événement déclencheur
    }
}
```

Voir *Events* dans la section contrats pour des informations sur la façon dont les événements sont déclarés et peuvent être utilisés à l'intérieur d'une application.

3.5.5 Erreurs

Les erreurs vous permettent de définir des noms et des données descriptives pour les situations d'échec. Les erreurs peuvent être utilisées dans *revert statements*. Par rapport aux descriptions de chaînes de caractères, les erreurs sont beaucoup moins coûteuses et vous permettent d'encoder des données supplémentaires. Vous pouvez utiliser NatSpec pour décrire l'erreur à l'utilisateur.

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

/// Pas assez de fonds pour le transfert. Demandé `requested`,
/// mais seulement `available` disponible.
error NotEnoughFunds(uint requested, uint available);

contract Token {
    mapping(address => uint) balances;
    function transfer(address to, uint amount) public {
        uint balance = balances[msg.sender];
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if (balance < amount)
            revert NotEnoughFunds(amount, balance);
        balances[msg.sender] -= amount;
        balances[to] += amount;
        // ...
    }
}
```

Voir *Errors and the Revert Statement* dans la section sur les contrats pour plus d'informations.

3.5.6 Types de structures

Les structures sont des types personnalisés qui peuvent regrouper plusieurs variables (voir *Structs* dans la section sur les types).

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

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

3.5.7 Types d'Enum

Les Enums peuvent être utilisées pour créer des types personnalisés avec un ensemble fini de « valeurs constantes » (voir *Enums* dans la section sur les types).

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

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

3.6 Types

Solidity est un langage statiquement typé, ce qui signifie que le type de chaque variable (état et locale) doit être spécifié. Solidity fournit plusieurs types élémentaires qui peuvent être combinés pour former des types complexes.

De plus, les types peuvent interagir entre eux dans des expressions contenant des opérateurs. Pour une référence rapide des différents opérateurs, voir *Ordre de Préséance des Opérateurs*.

Le concept de valeurs « indéfinies » ou « nulles » n'existe pas dans Solidity, mais les variables nouvellement déclarées ont toujours une *valeur par défaut* dépendant de son type. Pour gérer toute valeur inattendue, vous devez utiliser la

fonction *revert* pour annuler toute la transaction, ou retourner un tuple avec une seconde valeur `bool` indiquant le succès.

3.6.1 Value Types

The following types are also called value types because variables of these types 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.

Avertissement : 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 », which mean that if the result of an operation 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.

Avertissement : 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).

Note : 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.

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

Note : 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)`

Note : 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.

Note : 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.

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

Fixed Point Numbers

Avertissement : 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)

Note : 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 flavours, which are largely identical :
- `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 a plain `address` cannot be sent 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.

Note : 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.

Operators :
- <=, <, ==, !=, >= and >

Avertissement : 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 version 0.4.24 and higher of the compiler force you 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`.

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

Members of Addresses

For a quick reference of all members of address, see *Membres des types d'adresses*.
- `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.

Note : 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`.

Avertissement : 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);
```

Avertissement : 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()`).

Note : 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.

Note : 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`.

Note : 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.

Note : 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*.

Note : 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).

Note : 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.

Note : 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.

Note : 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 on one side. Examples include `1.`, `.1` and `1.3`.

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 a non-literal expression 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).

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.

Avertissement : 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`.

Note : 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.

Note : 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.

Note : 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);
    }

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

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

Note : 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 bound 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`.
    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.

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. 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*

Note : 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 {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

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

3.6.2 Reference Types

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored : `memory` (whose lifetime is limited to an external function call), `storage` (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or `calldata` (special data location that contains the function arguments).

An assignment or type conversion that changes the data location will always incur an automatic copy operation, while assignments inside the same data location only copy in some cases for storage types.

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.

Note : 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.

Note : 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.

Note : 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;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // 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 the array, where `.push()` appends a zero-initialized element and returns a reference to it.

`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 `bytes.concat(bytes(s1), bytes(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.

Note : 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.

`bytes.concat` function

You can concatenate a variable number of `bytes` or `bytes1 ... bytes32` using `bytes.concat`. 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, you need to convert them to `bytes` or `bytes1`/.../`bytes32` first.

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

contract C {
    bytes s = "Storage";
    function f(bytes calldata c, string memory m, bytes16 b) public view {
        bytes memory a = bytes.concat(s, c, c[:2], "Literal", bytes(m), b);
        assert((s.length + c.length + 2 + 7 + bytes(m).length + 16) == a.length);
    }
}
```

If you call `bytes.concat` without arguments it will return an empty `bytes` 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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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
        // 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.

Note : 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.

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

Note : 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] m_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).
    // Because of that, T[] is always a dynamic array of T, even if T
    // itself is an array.
    // Data location for all state variables is storage.
    bool[2][] m_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 ``m_pairsOfFlags``.
        m_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
        m_pairsOfFlags[index][0] = flagA;
        m_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 < m_pairsOfFlags.length) {
            while (m_pairsOfFlags.length > newSize)
```

(suite sur la page suivante)

(suite de la page précédente)

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

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

    bytes m_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[]"
        m_byteData = data;
        for (uint i = 0; i < 7; i++)
            m_byteData.push();
        m_byteData[3] = 0x08;
        delete m_byteData[2];
    }

    function addFlag(bool[2] memory flag) public returns (uint) {
        m_pairsOfFlags.push(flag);
        return m_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;
    }
}
```

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.

Note : 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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            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`.

Note : 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 Mapping Types

Mapping types use the syntax `mapping(_KeyType => _ValueType)` and variables of mapping type are declared using the syntax `mapping(_KeyType => _ValueType) _VariableName`. The `_KeyType` can be any built-in value type, `bytes`, `string`, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. `_ValueType` can be any type, including mappings, arrays and structs.

You can think of mappings as hash tables, which are virtually initialised such that every possible key exists and is mapped to a value whose byte-representation is all zeros, a type's *default value*. The similarity ends there, the key data is not stored in a mapping, only its `keccak256` hash is used to look up the value.

Because of this, mappings do not have a length or a concept of a key or value being set, and therefore cannot be erased without extra information regarding the assigned keys (see *Effacement des mappages*).

Mappings can only have a data location of `storage` and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.

You can mark state variables of mapping type as `public` and Solidity creates a *getter* for you. The `_KeyType` becomes a parameter for the getter. If `_ValueType` is a value type or a struct, the getter returns `_ValueType`. If `_ValueType` is an array or a mapping, the getter has one parameter for each `_KeyType`, recursively.

In the example below, the `MappingExample` contract defines a public `balances` mapping, with the key type an `address`, and a value type a `uint`, mapping an Ethereum address to an unsigned integer value. As `uint` is a value type, the getter returns a value that matches the type, which you can see in the `MappingUser` contract that returns the value at the specified address.

```
// 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));
    }
}
```

The example below is a simplified version of an ERC20 token. `_allowances` is an example of a mapping type inside another mapping type. The example below uses `_allowances` to record the amount someone else is allowed to withdraw from your account.

```
// 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;

    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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    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);
    }
}
```

Iterable Mappings

You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that. For example, the code below implements an `IterableMapping` library that the `User` contract then adds data too, and the `sum` function iterates over to sum all the values.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.8 <0.9.0;

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

struct itmap {
    mapping(uint => IndexValue) data;
    KeyFlag[] keys;
    uint size;
}

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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            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 iterate_start(itmap storage self) internal view returns (uint keyIndex) {
        return iterate_next(self, type(uint).max);
    }

    function iterate_valid(itmap storage self, uint keyIndex) internal view returns␣
↪(bool) {
        return keyIndex < self.keys.length;
    }

    function iterate_next(itmap storage self, uint keyIndex) internal view returns (uint␣
↪r_keyIndex) {
        keyIndex++;
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return keyIndex;
    }

    function iterate_get(itmap storage self, uint keyIndex) internal view returns (uint␣
↪key, uint value) {
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
    }
}

// How to use it
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        for (
            uint i = data.iterate_start();
            data.iterate_valid(i);
            i = data.iterate_next(i)
        ) {
            (, uint value) = data.iterate_get(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 `int32`. 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 arbitrary precision.

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.

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);
    }
}
```

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
```

Note : 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 from `bytes20` or any integer type to `address` result in `address payable`.

An `address a` can be converted to `address payable` via `payable(a)`.

3.7 Unités et variables disponibles dans le monde entier

3.7.1 Unités d'éther

Un nombre littéral peut prendre un suffixe de `wei`, `gwei` ou `ether` pour spécifier une sous-dénomination d'Ether, où les nombres d'Ether sans postfixe sont supposés être Wei.

```
assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 ether == 1e18);
```

Le seul effet du suffixe de sous-dénomination est une multiplication par une puissance de dix.

Note : Les dénominations `finney` et `szabo` ont été supprimées dans la version 0.7.0.

3.7.2 Unités de temps

Les suffixes comme `seconds`, `minutes`, `hours`, `days` et `weeks`, après des nombres littéraux, peuvent être utilisés pour spécifier des unités de temps où les secondes sont l'unité de base et les unités sont considérées naïvement de la manière suivante :

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`

Faites attention si vous effectuez des calculs de calendrier en utilisant ces unités, car chaque année n'est pas égale à 365 jours et chaque jour n'a pas 24 heures à cause des secondes intercalaires. En raison du fait que les secondes intercalaires ne peuvent pas être prédites, un calendrier exact doit être mis à jour par une bibliothèque doit être mise à jour par un oracle externe.

Note : Le suffixe `years` a été supprimé dans la version 0.5.0 pour les raisons ci-dessus.

Ces suffixes ne peuvent pas être appliqués aux variables. Par exemple, si vous voulez interpréter un paramètre de fonction en jours, vous pouvez le faire de la manière suivante :

```
function f(uint start, uint daysAfter) public {
    if (block.timestamp >= start + daysAfter * 1 days) {
      // ...
    }
}
```

3.7.3 Variables et fonctions spéciales

Certaines variables et fonctions spéciales existent toujours dans l'espace de nom global, et sont principalement utilisées pour fournir des informations sur la blockchain, ou sont des fonctions utilitaires d'usage général.

Propriétés des blocs et des transactions

- `blockhash(uint blockNumber) retourne (bytes32)` : hachage du bloc donné si `blocknumber` est l'un des 256 blocs les plus récents ; sinon retourne zéro.
- `block.basefee` (`uint`) : la redevance de base du bloc actuel (EIP-3198 et EIP-1559)
- `block.chainid` (`uint`) : identifiant de la chaîne actuelle
- `block.coinbase` (`address payable`) : adresse du mineur du bloc actuel
- `block.difficulty` (`uint`) : difficulté actuelle du bloc
- `block.gaslimit` (`uint`) : limite de gaz du bloc actuel
- `block.number` (`uint`) : numéro du bloc actuel
- `block.timestamp` (`uint`) : horodatage du bloc actuel en secondes depuis l'époque unix
- `gasleft() returns (uint256)` : gaz résiduel
- `msg.data` (`bytes calldata`) : données d'appel complètes
- `msg.sender` (`address`) : expéditeur du message (appel en cours)
- `msg.sig` (`bytes4`) : les quatre premiers octets des données d'appel (c'est-à-dire l'identifiant de la fonction)
- `msg.value` (`uint`) : nombre de wei envoyés avec le message
- `tx.gasprice` (`uint`) : prix du gaz de la transaction
- `tx.origin` (`address`) : expéditeur de la transaction (chaîne d'appel complète)

Note : Les valeurs de tous les membres de `msg`, y compris `msg.sender` et `msg.value` peuvent changer à chaque appel de fonction **externe**. Cela inclut les appels aux fonctions de la bibliothèque.

Note : Lorsque les contrats sont évalués hors chaîne plutôt que dans le contexte d'une transaction comprise dans un bloc, vous ne devez pas supposer que `block.*` et `tx.*` font référence à des valeurs d'un bloc ou d'une transaction spécifique. Ces valeurs sont fournies par l'implémentation EVM qui exécute le contrat et peuvent être arbitraires.

Note : Ne comptez pas sur `block.timestamp` ou `blockhash` comme source d'aléatoire, à moins que vous ne sachiez ce que vous faites.

L'horodatage et le hachage du bloc peuvent tous deux être influencés par les mineurs dans une certaine mesure. De mauvais acteurs dans la communauté minière peuvent par exemple exécuter une fonction de paiement de casino sur un hash choisi et réessayer un autre hash s'ils n'ont pas reçu d'argent.

L'horodatage du bloc actuel doit être strictement plus grand que l'horodatage du dernier bloc, mais la seule garantie est qu'il se situera quelque part entre les horodatages de deux blocs consécutifs dans la chaîne canonique.

Note : Les hachages des blocs ne sont pas disponibles pour tous les blocs pour des raisons d'évolutivité. Vous ne pouvez accéder qu'aux hachages des 256 blocs les plus récents. autres valeurs seront nulles.

Note : La fonction `blockhash` était auparavant connue sous le nom de `block.blockhash`, qui a été dépréciée dans la version 0.4.22 et supprimée dans la version 0.5.0.

Note : La fonction `gasleft` était auparavant connue sous le nom de `msg.gas`, qui a été dépréciée dans la version 0.4.21 et supprimée dans la version 0.5.0.

Note : Dans la version 0.7.0, l'alias `now` (pour `block.timestamp`) a été supprimé.

Fonctions de codage et de décodage de l'ABI

- `abi.decode(bytes memory encodedData, (...)) retourne (...)` : ABI-décode les données données, tandis que les types sont donnés entre parenthèses comme deuxième argument. Exemple : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)` : ABI-encode les arguments donnés
- `abi.encodePacked(...) returns (bytes memory)` : Effectue *l'encodage emballé* des arguments donnés. Notez que l'encodage emballé peut être ambigu !
- `abi.encodeWithSelector(bytes4 selector, ...) retourne (bytes memory)` : ABI-encode les arguments donnés en commençant par le deuxième et ajoute en préambule le sélecteur de quatre octets donné.
- `abi.encodeWithSignature(string memory signature, ...) retourne (bytes memory)` : Équivalent à `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) retourne (bytes memory)` : ABI-encode un appel à `functionPointer` avec les arguments trouvés dans le tuple. Effectue un contrôle de type complet, en s'assurant que les types correspondent à la signature de la fonction. Le résultat est égal à `abi.encodeWithSelector(functionPointer.selector, (...))`

Note : Ces fonctions d'encodage peuvent être utilisées pour créer des données pour les appels de fonctions externes sans réellement appeler une fonction externe. De plus, `keccak256(abi.encodePacked(a, b))` est un moyen de calculer le hachage de données structurées (attention, il est possible de créer une « collision de hachage » en utilisant différents types de paramètres de fonction).

Reportez-vous à la documentation sur le *ABI* et le *codage étroitement emballé* pour plus de détails sur le codage.

Membres des octets

- `bytes.concat(...) retourne (bytes memory)` : *Concatène un nombre variable d'octets et les arguments bytes1, ..., bytes32 dans un tableau d'octets.*

Traitement des erreurs

Consultez la section dédiée à *assert et require* pour plus de détails sur la gestion des erreurs et quand utiliser telle ou telle fonction.

assert(bool condition) provoque une erreur de panique et donc un changement d'état si la condition n'est pas remplie - à utiliser pour les erreurs internes.

require(bool condition) revient en arrière si la condition n'est pas remplie - à utiliser pour les erreurs dans les entrées ou les composants externes.

require(bool condition, string memory message) fait marche arrière si la condition n'est pas remplie - à utiliser pour les erreurs dans les entrées ou les composants externes. Fournit également un message d'erreur.

revert() interrompt l'exécution et renverse les changements d'état

revert(string memory reason) interrompt l'exécution et annule les changements d'état, en fournissant une chaîne explicative.

Fonctions mathématiques et cryptographiques

addmod(uint x, uint y, uint k) retourne (uint) calcule `(x + y) % k` où l'addition est effectuée avec une précision arbitraire et ne s'arrête pas à `2**256`. Affirme que `k != 0` à partir de la version 0.5.0.

mulmod(uint x, uint y, uint k) retourne (uint) calcule `(x * y) % k` où la multiplication est effectuée avec une précision arbitraire et ne s'arrête pas à `2**256`. Affirme que `k != 0` à partir de la version 0.5.0.

keccak256(octets mémoire) retourne (octets32) calcule le hachage Keccak-256 de l'entrée

---

Note : Il y avait auparavant un alias pour `keccak256` appelé `sha3`, qui a été supprimé dans la version 0.5.0.

---

sha256(bytes memory) retourne (bytes32) calcule le hachage SHA-256 de l'entrée

ripemd160(bytes memory) retourne (bytes20) calcule le hachage RIPEMD-160 de l'entrée

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) retourne (address) récupère l'adresse associée à la clé publique de la signature à courbe elliptique ou renvoie zéro en cas d'erreur. Les paramètres de la fonction correspondent aux valeurs ECDSA de la signature :
— `r` = premiers 32 octets de la signature
— `s` = deuxième 32 octets de la signature
— `v` = dernier 1 octet de la signature
`ecrecover` retourne une `adresse`, et non une `adresse payable`. Voir *address payable* pour la conversion, au cas où vous auriez besoin de transférer des fonds à l'adresse récupérée.

Pour plus de détails, lisez example usage.

Avertissement : Si vous utilisez `ecrecover`, soyez conscient qu'une signature valide peut être transformée en une signature valide différente sans avoir besoin de connaître la clé privée correspondante. Dans le hard fork de Homestead, ce problème a été corrigé pour les signatures _transaction_ (voir EIP-2), mais la fonction ecrecover est restée inchangée.

Ce n'est généralement pas un problème, à moins que vous n'exigiez que les signatures soient uniques ou que vous les utilisiez pour identifier des éléments. OpenZeppelin a une ECDSA helper library que vous pouvez utiliser comme un wrapper pour `ecrecover` sans ce problème.

Note : Lorsque vous exécutez les fonctions `sha256`, `ripemd160` ou `ecrecover` sur une *blockchain privée*, vous pouvez rencontrer des problèmes d'épuisement. Cela est dû au fait que ces fonctions sont implémentées en tant que « contrats

précompilés » et n'existent réellement qu'après avoir reçu le premier message (bien que leur code de contrat soit codé en dur). Les messages destinés à des contrats inexistants sont plus coûteux et l'exécution peut donc se heurter à une erreur Out-of-Gas. Une solution à ce problème consiste à envoyer d'abord du Wei (1 par exemple) à chacun des contrats avant de les utiliser dans vos contrats réels. Ce n'est pas un problème sur le réseau principal ou le réseau de test.

Membres des types d'adresses

<address>.balance (uint256) solde de l'"adresse dans Wei

<address>.code (bytes memory) code à l'"adresse (peut être vide)

<address>.codehash (bytes32) le codehash de l'adresse *Address*.

<address payable>.transfer(uint256 amount) envoie une quantité donnée de Wei à adress, revient en arrière en cas d'échec, envoie 2300 de gaz, non réglable

<address payable>.send(uint256 amount) returns (bool) envoie un montant donné de Wei à *Address*, renvoie `false` en cas d'échec, envoie 2300 de gaz, non réglable

<address>.call(bytes memory) returns (bool, bytes memory) émet un `CALL` de bas niveau avec la charge utile donnée, renvoie la condition de succès et les données de retour, transmet tous les gaz disponibles, ajustable

<address>.delegatecall(bytes memory) returns (bool, bytes memory) émet un `DELEGATECALL` de bas niveau avec la charge utile donnée, renvoie la condition de succès et les données de retour, transmet tous les gaz disponibles, réglable

<address>.staticcall(bytes memory) returns (bool, bytes memory) émet un `STATICCALL` de bas niveau avec la charge utile donnée, renvoie la condition de succès et les données de retour, transmet tous les gaz disponibles, réglable

Pour plus d'informations, consultez la section sur adress.

Avertissement : Vous devez éviter d'utiliser `.call()` chaque fois que possible lors de l'exécution d'une autre fonction de contrat car elle contourne la vérification de type le contrôle d'existence de la fonction et l'emballage des arguments.

Avertissement : Il y a quelques dangers à utiliser `send` : Le transfert échoue si la profondeur de la pile d'appel est à 1024 (ceci peut toujours être forcé par l'appelant) et il échoue également si le destinataire tombe en panne sèche. Donc, afin de de faire des transferts d'Ether sûrs, vérifiez toujours la valeur de retour de `send`, utilisez `transfer` ou encore mieux : Utilisez un modèle où le destinataire retire l'argent.

Avertissement : En raison du fait que l'EVM considère qu'un appel à un contrat inexistant réussit toujours, Solidity inclut une vérification supplémentaire en utilisant l'opcode `extcodesize` lors des appels externes. Cela garantit que le contrat qui est sur le point d'être appelé existe réellement (il contient du code) soit une exception est levée.

Les appels de bas niveau qui opèrent sur des adresses plutôt que sur des instances de contrat (c'est-à-dire `.call()`, `.delegatecall()`, `.staticcall()`, `.send()` et `.transfer()`) **n'incluent pas** cette vérification, ce qui les rend moins coûteux en termes de gaz mais aussi moins sûrs.

Note : Avant la version 0.5.0, Solidity permettait d'accéder aux membres adresse par une instance de contrat, par exemple `this.balance`. Ceci est maintenant interdit et une conversion explicite en adresse doit être faite :

`address(this).balance`.

Note : Si l'on accède à des variables d'état via un appel de délégué de bas niveau, la disposition de stockage des deux contrats doit s'aligner pour que le contrat appelé puisse accéder correctement aux variables de stockage du contrat appelant par leur nom. Ce n'est évidemment pas le cas si les pointeurs de stockage sont passés comme arguments de fonction, comme dans le cas des bibliothèques de haut niveau.

Note : Avant la version 0.5.0, `.call`, `.delegatecall` et `.staticcall` retournaient uniquement la condition de réussite et non les données de retour.

Note : Avant la version 0.5.0, il existait un membre appelé `callcode`' avec une sémantique similaire mais légèrement différente de celle de ``deallcode`, sémantique similaire mais légèrement différente de celle de `delegatecall`.

Concernant les contrats

`this` (le type du contrat actuel) le contrat actuel, explicitement convertible en *Address*.

`selfdestruct(address payable recipient)` Détruit le contrat actuel, en envoyant ses fonds à l'adresse *Address* donnée et mettre fin à l'exécution. Notez que `selfdestruct` a quelques particularités héritées de l'EVM :
- la fonction de réception du contrat récepteur n'est pas exécutée.
- le contrat n'est réellement détruit qu'à la fin de la transaction et les `revert` peuvent « annuler » la destruction.

En outre, toutes les fonctions du contrat en cours sont appelables directement, y compris la fonction en cours.

Note : Avant la version 0.5.0, il existait une fonction appelée `suicide` ayant la même sémantique que la fonction `selfdestruct`.

Informations sur le type de produit

L'expression `type(X)` peut être utilisée pour récupérer des informations sur le type `X`. Actuellement, la prise en charge de cette fonctionnalité est limitée (`X` peut être soit un contrat ou un type entier) mais elle pourrait être étendue dans le futur.

Les propriétés suivantes sont disponibles pour un type de contrat `C` :

`type(C).name` Le nom du contrat.

`type(C).creationCode` Tableau d'octets en mémoire qui contient le bytecode de création du contrat. Ceci peut être utilisé dans l'assemblage en ligne pour construire des routines de création personnalisées, notamment en utilisant l'opcode `create2`. Cette propriété n'est **pas** accessible dans le contrat lui-même ou dans un contrat dérivé. Elle provoque l'inclusion du bytecode dans le bytecode du site d'appel et donc les références circulaires de ce genre ne sont pas possibles.

`type(C).runtimeCode` Tableau d'octets en mémoire qui contient le bytecode d'exécution du contrat. Il s'agit du code qui est généralement déployé par le constructeur de `C`. Si `C` a un constructeur qui utilise l'assemblage en ligne, cela peut être différent du bytecode réellement déployé. Notez également que les bibliothèques modifient leur code d'exécution au moment du déploiement pour se prémunir contre les appels réguliers. Les mêmes restrictions que pour `.creationCode` s'appliquent à cette propriété.

En plus des propriétés ci-dessus, les propriétés suivantes sont disponibles pour une interface de type `I` :

`type(I).interfaceId :` Une valeur `bytes4` contenant le EIP-165 de l'interface `I` donnée. Cet identificateur est défini comme étant le `XOR` de tous les sélecteurs de fonctions définis dans l'interface elle-même - à l'exclusion de toutes les fonctions héritées.

Les propriétés suivantes sont disponibles pour un type entier `T` :

`type(T).min` La plus petite valeur représentable par le type `T`.

`type(T).max` La plus grande valeur représentable par le type `T`.

3.8 Expressions et structures de contrôle

3.8.1 Structures de contrôle

La plupart des structures de contrôle connues des langages à accolades sont disponibles dans Solidity :

Il y a : » if « , » else « , « while « , » do « , » for « , » break « , » continue « , » return « , avec la sémantique la sémantique habituelle connue en C ou en JavaScript.

Solidity prend également en charge la gestion des exceptions sous la forme de déclarations » try » et » catch « , mais seulement pour *les appels de fonctions externes* et pour les appels de création de contrat. Les erreurs peuvent être créées en utilisant l'instruction *revert*.

Les parenthèses ne peuvent *pas* être omises pour les conditionnels, mais les accolades peuvent être omises autour des corps d'énoncés simples.

Notez qu'il n'y a pas de conversion de type de non-booléen à booléen comme en C et JavaScript. booléens comme c'est le cas en C et en JavaScript, donc « if (1) { … }`` n'est *pas* valide Solidité.

3.8.2 Appels de fonction

Appels de fonctions internes

Les fonctions du contrat en cours peuvent être appelées directement (« en interne »), également de manière récursive, comme on le voit dans cet exemple absurde :

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

// Ceci signalera un avertissement
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(); }
}
```

Ces appels de fonction sont traduits en simples sauts à l'intérieur de l'EVM. Cela a pour l'effet que la mémoire courante n'est pas effacée, c'est-à-dire que le passage des références de mémoire aux fonctions appelées en interne est très efficace. Seules les fonctions de la même instance de contrat peuvent être appelées en interne.

Vous devez néanmoins éviter toute récursion excessive, car chaque appel de fonction interne utilise au moins un emplacement de pile et il n'y a que 1024 emplacements disponibles.

External Function Calls

Les fonctions peuvent également être appelées en utilisant la notation » this.g(8);`` et » c.g(2);``, où `c` est une instance de contrat et `g` est une fonction appartenant à `c`. L'appel de la fonction *g* ` de l'une ou l'autre façon a pour conséquence qu'elle est appelée « en externe », en utilisant appel de message et non directement via des sauts. Veuillez noter que les appels de fonction sur `this` ne peuvent pas être utilisés dans le constructeur, car le contrat réel n'a pas encore été créé.

Les fonctions des autres contrats doivent être appelées en externe. Pour un appel externe, tous les arguments de la fonction doivent être copiés en mémoire.

Note : Un appel de fonction d'un contrat à un autre ne crée pas sa propre transaction, il s'agit d'un appel de message faisant partie de la transaction globale.

Lorsque vous appelez des fonctions d'autres contrats, vous pouvez préciser la quantité de Wei ou de gaz envoyée avec l'appel avec les options spéciales `{valeur : 10, gaz : 10000}`. Notez qu'il est déconseillé de spécifier des valeurs de gaz explicitement, puisque les coûts de gaz des opcodes peuvent changer dans le futur. Tout Wei que vous envoyez au contrat est ajouté au solde total de ce contrat :

```
// 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}(); }
}
```

Vous devez utiliser le modificateur `payable` avec la fonction `info` parce que sinon, l'option `value` ne serait pas disponible.

Avertissement : Attention, `feed.info{value : 10, gaz : 800}` ne définit que localement la valeur et la quantité de `gaz` envoyée avec l'appel de la fonction, et que les parenthèses à la fin effectuent l'appel réel. Donc `feed.info{value : 10, gaz : 800}` n'appelle pas la fonction et et les paramètres « valeur » et `gaz` sont perdus, mais seulement `feed.info{value : 10, gaz : 800}()` effectue l'appel de fonction.

En raison du fait que l'EVM considère qu'un appel vers un contrat inexistant toujours réussir, Solidity utilise l'opcode `extcodesize` pour vérifier que le contrat qui est sur le point d'être appelé existe réellement (il contient du code) et provoque une exception si ce n'est pas le cas. Cette vérification est ignorée si les données de retour seront décodées après l'appel et donc le décodeur ABI va attraper le cas d'un contrat inexistant.

Notez que cette vérification n'est pas effectuée dans le cas de *appels de bas niveau* qui opèrent sur des adresses plutôt que sur des instances de contrat.

Note : Soyez prudent lorsque vous utilisez des appels de haut niveau à *contrats précompilés*, car le compilateur les considère comme inexistants selon la logique logique ci-dessus, même s'ils exécutent du code et peuvent retourner des données.

Les appels de fonction provoquent également des exceptions si le contrat appelé lui-même lève une exception ou tombe en panne.

Avertissement : Toute interaction avec un autre contrat impose un danger potentiel, surtout si le code source du contrat n'est pas connu à l'avance. Le contrat en cours transmet le contrôle au contrat appelé et celui-ci peut potentiellement faire à peu près n'importe quoi. Même si le contrat appelé hérite d'un contrat parent connu, le contrat hérité est seulement tenu d'avoir une interface correcte. Le site L'implémentation du contrat, cependant, peut être complètement arbitraire et donc.., constituer un danger. En outre, il faut se préparer à l'éventualité qu'il fasse appel à d'autres contrats de votre système ou même de revenir au contrat appelant avant que le premier appel ne revienne. Cela signifie que le contrat appelé peut modifier les variables d'état du contrat appelant via ses fonctions. Écrivez vos fonctions de manière à ce que, par exemple, les appels aux fonctions externes se produisent après toute modification des variables d'état dans votre contrat afin que votre contrat ne soit pas vulnérable à un exploit de réentraînement.

Note : Avant Solidity 0.6.2, la manière recommandée de spécifier la valeur et le gaz était de utiliser « f.value(x).gas(g)()``. Cette méthode a été dépréciée dans Solidity 0.6.2 et n'est plus possible depuis Solidity 0.7.0.

Appels nominatifs et paramètres de fonctions anonymes

Les arguments d'un appel de fonction peuvent être donnés par leur nom, dans n'importe quel ordre, s'ils sont entourés de { } comme on peut le voir dans l'exemple suivant. La liste d'arguments doit coïncider par son nom avec la liste des paramètres de la déclaration de la fonction, mais peut être dans un ordre arbitraire.

```
// 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;
    }

}
```

Noms des paramètres de la fonction omise

Les noms des paramètres non utilisés (en particulier les paramètres de retour) peuvent être omis. Ces paramètres seront toujours présents sur la pile, mais ils seront inaccessibles.

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

contract C {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // nom omis pour le paramètre
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

3.8.3 Créer des contrats via `new` (nouveau)

Un contrat peut créer d'autres contrats en utilisant le mot-clé `new`. Le code complet du contrat en cours de création doit être connu lorsque le contrat créateur est compilé afin que les dépendances récursives de création ne soient pas possibles.

```
// 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); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}
```

Comme on le voit dans l'exemple, il est possible d'envoyer de l'Ether en créant une instance de `D` en utilisant l'option `value`, mais il n'est pas possible de de limiter la quantité d'éther. Si la création échoue (à cause d'un dépassement de pile, d'un équilibre insuffisant ou d'autres problèmes), une exception est levée.

Créations de contrats salés / create2

Lors de la création d'un contrat, l'adresse du contrat est calculée à partir de l'adresse du contrat créateur et d'un compteur qui est augmenté à chaque création de chaque création de contrat.

Si vous spécifiez l'option `salt` (une valeur bytes32), alors la création de contrat utilisera un un mécanisme différent pour trouver l'adresse du nouveau contrat :

Elle calculera l'adresse à partir de l'adresse du contrat en cours de création, la valeur du sel donnée, le bytecode (de création) du contrat créé et les arguments du constructeur.

En particulier, le compteur (« nonce ») n'est pas utilisé. Cela permet une plus grande flexibilité dans la création de contrats : Vous pouvez dériver l'adresse du nouveau contrat avant qu'il ne soit créé. En outre, vous pouvez vous fier à cette adresse également dans le cas où le créateur contrat crée d'autres contrats entre-temps.

Le principal cas d'utilisation ici est celui des contrats qui agissent en tant que juges pour les interactions hors chaîne, qui n'ont besoin d'être créés que s'il y a un différend.

```
// 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 {
        // Cette expression compliquée vous indique simplement comment l'adresse
        // peut être précalculée. Elle n'est là qu'à titre d'illustration.
        // En fait, vous n'avez besoin que de ``new D{salt : salt}(arg)``.
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                arg
            ))
        )))));

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

Avertissement : Il existe quelques particularités en ce qui concerne la création salée. Un contrat peut être recréé à la même adresse après avoir été détruit. Pourtant, il est possible pour ce contrat nouvellement créé d'avoir un bytecode déployé différent, même si le bytecode de création a été le même (ce qui est une exigence parce que sinon l'adresse changerait). Ceci est dû au fait que le constructeur peut interroger l'état externe qui pourrait avoir changé entre les deux créations et l'incorporer dans le bytecode déployé avant qu'il ne soit stocké.

3.8.4 Ordre d'évaluation des expressions

L'ordre d'évaluation des expressions n'est pas spécifié (de manière plus formelle, l'ordre dans lequel les enfants d'un noeud de l'arbre des expressions sont évalués n'est pas spécifié, mais ils sont bien sûr évalués avant le noeud lui-même). Il est seulement garantie que les instructions sont exécutées dans l'ordre et que le court-circuitage des expressions booléennes est effectué.

3.8.5 Affectation

Déstructurer les affectations et renvoyer des valeurs multiples

Solidity autorise en interne les types tuple, c'est-à-dire une liste d'objets potentiellement différents dont le nombre est une constante à la constante au moment de la compilation. Ces tuples peuvent être utilisés pour retourner plusieurs valeurs en même temps. Celles-ci peuvent alors être affectées à des variables nouvellement déclarées soit à des variables préexistantes (ou à des valeurs LV en général).

Les tuples ne sont pas des types à proprement parler dans Solidity, ils ne peuvent être utilisés que pour former des groupements syntaxiques d'expressions.

```
// 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 déclarées avec le type et assignées à partir du tuple retourné,
        // il n'est pas nécessaire de spécifier tous les éléments (mais le nombre doit␣
↪correspondre).
        (uint x, , uint y) = f();
        // Truc commun pour échanger des valeurs -- ne fonctionne pas pour les types de␣
↪stockage sans valeur.
        (x, y) = (y, x);
        // Les composants peuvent être laissés de côté (également pour les déclarations␣
↪de variables).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

Il n'est pas possible de mélanger les déclarations de variables et les affectations non déclarées. Par exemple, l'exemple suivant n'est pas valide : `(x, uint y) = (1, 2);`

Note : Avant la version 0.5.0, il était possible d'assigner à des tuples de taille plus petite, soit en remplissant le côté gauche ou le côté droit (celui qui était vide). Ceci est maintenant interdit, donc les deux côtés doivent avoir le même nombre de composants.

Avertissement : Soyez prudent lorsque vous assignez à plusieurs variables en même temps lorsque des types de référence sont impliqués, car cela pourrait conduire à un comportement de copie inattendu.

Complications pour les tableaux et les structures

La sémantique des affectations est plus compliquée pour les types non-valeurs comme les tableaux et les structs, y compris les `octets` et les `chaînes`, voir *L'emplacement des données et le comportement d'affectation* pour plus de détails.

Dans l'exemple ci-dessous, l'appel à `g(x)` n'a aucun effet sur `x` parce qu'il crée une copie indépendante de la valeur de stockage en mémoire. Cependant, `h(x)` modifie avec succès `x` car seule une référence et non une copie est transmise.

```
// 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 Champ d'application et déclarations

Une variable qui est déclarée aura une valeur initiale par défaut dont la représentation en octets est constituée de zéros. Les « valeurs par défaut » des variables sont l'"« état zéro » typique de leur type. Par exemple, la valeur par défaut d'un `bool` est `false`. La valeur par défaut des types `uint` ou `int` est `0`. Pour les tableaux de taille statique et les types `bytes1` à `bytes32`, chaque élément sera initialisé à la valeur par défaut correspondant à son à son type. Pour les tableaux de taille dynamique, les `octets` et `string`, la valeur par défaut est un tableau ou une chaîne vide. Pour le type `enum`, la valeur par défaut est son premier membre.

Le scoping dans Solidity suit les règles de scoping répandues de C99 (et de nombreux autres langages) : Les variables sont visibles à partir du point juste après leur déclaration jusqu'à la fin du plus petit bloc `{ }` qui contient la déclaration. Les variables déclarées dans la partie d'initialisation d'une boucle `for` font exception à cette partie d'initialisation d'une boucle for ne sont visibles que jusqu'à la fin de la boucle for.

Les variables qui sont des paramètres (paramètres de fonction, paramètres de modificateur, paramètres de capture, ...) sont visibles à l'intérieur du bloc de code qui suit - le corps de la fonction/modificateur pour un paramètre de fonction et de modificateur et le bloc catch pour un paramètre catch.

Les variables et autres éléments déclarés en dehors d'un bloc de code, par exemple les fonctions, les contrats, les types définis par l'utilisateur, etc., sont visibles avant même d'avoir été déclarés. Cela signifie que vous pouvez utiliser des variables d'état avant qu'elles ne soient déclarées et appeler des fonctions de manière récursive.

En conséquence, les exemples suivants compileront sans avertissement, puisque les deux variables ont le même nom mais des portées disjointes.

```
// 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;
        }
    }
}
```

Comme exemple spécial des règles de scoping de C99, notez que dans ce qui suit, la première affectation à `x` va en fait affecter la variable externe et non la variable interne. Dans tous les cas, vous obtiendrez un avertissement sur le fait que la variable externe est cachée.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// Ceci signalera un avertissement
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // ceci sera assigné à la variable externe
            uint x;
        }
        return x; // x a la valeur 2
    }
}
```

Avertissement : Avant la version 0.5.0, Solidity suivait les mêmes règles de portée que le langage JavaScript, c'est-à-dire qu'une variable déclarée n'importe où dans une fonction avait une portée pour l'ensemble de la fonction, indépendamment de l'endroit où elle était déclarée. L'exemple suivant montre un extrait de code qui utilisait pour compiler mais qui conduit à une erreur à partir de la version 0.5.0.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// Cela ne compilera pas
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

3.8.7 Arithmétique vérifiée ou non vérifiée

Un débordement ou un sous-débordement est la situation où la valeur résultante d'une opération arithmétique, lorsqu'elle est exécutée sur un entier non limité, tombe en dehors de la plage du type de résultat.

Avant la version 0.8.0 de Solidity, les opérations arithmétiques s'emballaient toujours en cas de débordement ou de sous-débordement, ce qui a conduit à l'utilisation répandue de bibliothèques qui vérifications supplémentaires.

Depuis la version 0.8.0 de Solidity, toutes les opérations arithmétiques s'inversent par défaut en cas de dépassement inférieur ou supérieur, rendant ainsi inutile l'utilisation de ces bibliothèques.

Pour obtenir le comportement précédent, un bloc `unchecked` peut être utilisé :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // Cette soustraction se terminera par un dépassement de capacité.
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // Cette soustraction s'inversera en cas de dépassement de capacité.
        return a - b;
    }
}
```

L'appel à `f(2, 3)` retournera `2**256-1`, alors que `g(2, 3)` provoquera une assertion qui échoue.

Le bloc « non vérifié » peut être utilisé partout à l'intérieur d'un bloc, mais pas en remplacement pour un bloc. Il ne peut pas non plus être imbriqué.

Le paramètre n'affecte que les déclarations qui sont syntaxiquement à l'intérieur du bloc. Les fonctions appelées à l'intérieur d'un bloc « non vérifié » n'héritent pas de cette propriété.

Note : Pour éviter toute ambiguïté, vous ne pouvez pas utiliser `_;` à l'intérieur d'un bloc `non vérifié`.

Les opérateurs suivants provoqueront une assertion d'échec en cas de débordement ou de sous-débordement et s'enrouleront sans erreur s'ils sont utilisés à l'intérieur d'un bloc non vérifié :

`++`, `--`, `+`, binaire `-`, unaire `-`, `*`, `/`, `%`, `**`

`+=`, `-=`, `*=`, `/=`, `%=`

Avertissement : Il n'est pas possible de désactiver la vérification de la division par zéro ou modulo par zéro en utilisant le bloc `unchecked`.

Note : Les opérateurs binaires n'effectuent pas de vérification de dépassement de capacité ou de sous-dépassement. Ceci est particulièrement visible lors de l'utilisation de décalages binaires (`<<`, `>>`, `<=`, `>=`) à la place de la division d'entiers et de la multiplication par une puissance de 2. Par exemple, `type(uint256).max << 3` ne s'inverse pas alors que `type(uint256).max * 8` le ferait.

Note : La deuxième instruction dans `int x = type(int).min ; -x;` entraînera un dépassement de capacité car

l'intervalle négatif peut contenir une valeur de plus que l'intervalle positif.

Les conversions de type explicites seront toujours tronquées et ne provoqueront jamais une assertion d'échec à l'exception de la conversion d'un entier en un type enum.

3.8.8 Gestion des erreurs : Assert, Require, Revert et Exceptions

Solidity utilise des exceptions de retour à l'état initial pour gérer les erreurs. Une telle exception annule toutes les modifications apportées à l'état dans l'appel actuel (et tous ses sous-appels) et signale une erreur à l'appelant.

Lorsque des exceptions se produisent dans un sous-appel, elles « remontent » (c'est-à-dire que les exceptions sont rejetées) automatiquement à moins qu'elles ne soient capturées dans dans une instruction `try/catch`. Les exceptions à cette règle sont `send` et les fonctions de bas niveau `call`, `delegatecall` et `staticcall` : elles retournent *false* ` comme première valeur de retour en cas d'une exception, au lieu de « bouillonner ».

Avertissement : Les fonctions de bas niveau `call`, `delegatecall` et `staticcall` retournent *true* ` comme première valeur de retour si le compte appelé est inexistant, ce qui fait partie de la conception de l'EVM. L'existence du compte doit être vérifiée avant l'appel si nécessaire.

Les exceptions peuvent contenir des données d'erreur qui sont renvoyées à l'appelant sous la forme de *error instances*. Les erreurs intégrées « Erreur(string) » et « Panique(uint256) » sont utilisées par des fonctions spéciales, comme expliqué ci-dessous. `Error` est utilisé pour les conditions d'erreurs « normales ». Tandis que `Panic` est utilisé pour les erreurs qui ne devraient pas être présentes dans un code sans bogues.

Panique via « Assert » et erreur via « Require ».

Les fonctions pratiques ``assert"" et ``require"" peuvent être utilisées pour vérifier les conditions et lancer une exception si la condition n'est pas remplie.

La fonction `assert` crée une erreur de type `Panic(uint256)`. La même erreur est créée par le compilateur dans certaines situations, comme indiqué ci-dessous.

Assert ne doit être utilisée que pour tester les erreurs internes et pour vérifier les invariants. Un code qui fonctionne correctement ne devrait jamais créer un Panic, même pas sur une entrée externe invalide. Si cela se produit, alors il y a un bogue dans votre contrat que vous devez corriger. Les outils d'analyse du langage peuvent évaluer votre contrat pour identifier les conditions et les appels de fonction qui provoquent une panique.

Une exception de panique est générée dans les situations suivantes. Le code d'erreur fourni avec les données d'erreur indique le type de panique.

1. 0x00 : Utilisé pour les paniques génériques insérées par le compilateur.
2. 0x01 : Si vous appelez `assert` avec un argument qui évalue à false.
3. 0x11 : Si une opération arithmétique résulte en un débordement ou un sous-débordement en dehors d'un bloc « non vérifié { .... }``.
4. 0x12 : Si vous divisez ou modulez par zéro (par exemple, `5 / 0` ou `23 % 0`).
5. 0x21 : Si vous convertissez une valeur trop grande ou négative en un type d'enum.
6. 0x22 : Si vous accédez à un tableau d'octets de stockage qui est incorrectement codé.
7. 0x31 : Si vous appelez `.pop()` sur un tableau vide.
8. 0x32 : Si vous accédez à un tableau, à `bytesN` ou à une tranche de tableau à un index hors limites ou négatif (c'est-à-dire `x[i]` où `i >= x.length` ou `i < 0`).
9. 0x41 : Si vous allouez trop de mémoire ou créez un tableau trop grand.

10. 0x51 : Si vous appelez une variable zéro initialisée de type fonction interne.

La fonction `require` crée soit une erreur sans aucune donnée, soit une erreur de type `Error(string)`. Elle doit être utilisée pour garantir des conditions valides qui ne peuvent pas être détectées avant le moment de l'exécution. Cela inclut les conditions sur les entrées ou les valeurs de retour des appels à des contrats externes.a

---

Note : Il n'est actuellement pas possible d'utiliser des erreurs personnalisées en combinaison avec `require`. Veuillez utiliser `if (!condition) revert CustomError();` à la place.

Une exception `Error(string)` (ou une exception sans données) est générée par le compilateur dans les situations suivantes :

1. Appeler `require(x)` où `x` est évalué à `false`.
2. Si vous utilisez `revert()` ou `revert("description")`.
3. Si vous effectuez un appel de fonction externe ciblant un contrat qui ne contient pas de code.
4. Si votre contrat reçoit de l'Ether via une fonction publique sans modificateur `payable` (y compris le constructeur et la fonction de repli).
5. Si votre contrat reçoit de l'Ether via une fonction publique getter.

Dans les cas suivants, les données d'erreur de l'appel externe (s'il est fourni) sont transférées. Cela signifie qu'il peut soit causer une *Error* ou une *Panic* (ou toute autre donnée) :

1. Si un `.transfer()` échoue.
2. Si vous appelez une fonction via un appel de message mais qu'elle ne se termine pas correctement (c'est-à-dire qu'elle tombe en panne sèche, qu'il n'y a pas de lève elle-même une exception), sauf lorsqu'une opération de bas niveau `call`, `send`, `delegatecall`, `callcode` ou `staticcall` est utilisé. Les opérations de bas niveau ne lèvent jamais d'exceptions mais indiquent les échecs en retournant `false`.
3. Si vous créez un contrat en utilisant le mot-clé `new` mais que le contrat création *ne se termine pas correctement*.

Vous pouvez éventuellement fournir une chaîne de message pour `require`, mais pas pour `assert`.

Note : Si vous ne fournissez pas un argument de type chaîne à `require`, il se retournera avec des données d'erreur vides, sans même inclure le sélecteur d'erreur.

L'exemple suivant montre comment vous pouvez utiliser `require` pour vérifier les conditions sur les entrées et `assert` pour vérifier les erreurs internes.

```
// 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);
        // Puisque le transfert lève une exception en cas d'échec et que
        // ne peut pas rappeler ici, il ne devrait pas y avoir de moyen pour nous
        // d'avoir encore la moitié de l'argent.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

En interne, Solidity effectue une opération de retour en arrière (instruction `0xfd`). Cela provoque l'EVM à revenir sur toutes les modifications apportées à l'état. La raison de ce retour en arrière est qu'il n'y a pas de moyen sûr de poursuivre l'exécution, parce qu'un effet attendu ne s'est pas produit. Parce que nous voulons conserver l'atomicité des transactions, l'action la plus sûre est d'annuler tous les changements et de rendre la transaction entière (ou au moins l'appel) sans effet.

Dans les deux cas, l'appelant peut réagir à de tels échecs en utilisant `try/catch`, mais mais les changements dans l'appelant seront toujours annulés.

Note : Les exceptions de panique utilisaient l'opcode `invalid`' avant Solidity `0.8.0`, qui consommait tout le gaz disponible pour l'appel. Les exceptions qui utilisent ``require consommaient tout le gaz jusqu'à la version Metropolis.

---

revert

Une réversion directe peut être déclenchée à l'aide de l'instruction `revert` et de la fonction `revert`.

L'instruction `revert` prend une erreur personnalisée comme argument direct sans parenthèses :

revert CustomError(arg1, arg2) ;

Pour des raisons de rétrocompatibilité, il existe également la fonction `revert()`, qui utilise des parenthèses et accepte une chaîne de caractères :

revert() ; revert(« description ») ;

Les données d'erreur seront renvoyées à l'appelant et pourront être capturées à cet endroit. L'utilisation de `revert()` provoque un revert sans aucune donnée d'erreur alors que `revert("description")` créera une erreur `Error(string)`.

L'utilisation d'une instance d'erreur personnalisée sera généralement beaucoup plus économique qu'une description sous forme de chaîne, car vous pouvez utiliser le nom de l'erreur pour la décrire, qui est encodé dans seulement quatre octets. Une description plus longue peut être fournie via NatSpec, ce qui n'entraîne aucun coût.

L'exemple suivant montre comment utiliser une chaîne d'erreur et une instance d'erreur personnalisée avec `revert` et l'équivalent `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.");
        // Autre façon de faire :
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Effectuer l'achat.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();
```

(suite sur la page suivante)

(suite de la page précédente)

```
        payable(msg.sender).transfer(address(this).balance);
    }
}
```

Les deux façons de faire `si (!condition) revert(...);` et `require(condition, ...);` sont équivalentes tant que les arguments de `revert` et `require` n'ont pas d'effets secondaires, par exemple si ce ne sont que des chaînes de caractères.

Note : La fonction `require` est évaluée comme n'importe quelle autre fonction. Cela signifie que tous les arguments sont évalués avant que la fonction elle-même ne soit exécutée. En particulier, dans `require(condition, f())` la fonction `f` est exécutée même si `condition` est vraie.

La chaîne fournie est *abi-encoded* comme s'il s'agissait d'un appel à une fonction `Error(string)`. Dans l'exemple ci-dessus, `revert("Not enough Ether provided.");` renvoie l'hexadécimal suivant comme données de retour d'erreur :

```
0x08c379a0                                                         // Sélecteur de␣
↪fonction pour Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Décalage des␣
↪données
0x000000000000000000000000000000000000000000000000000000000000001a // Longueur de la␣
↪chaîne
0x4e6f7420656e6f756768204574686572207072f6696465642e000000000000 // Données en chaîne
```

Le message fourni peut être récupéré par l'appelant à l'aide de `try/catch` comme indiqué ci-dessous.

Note : Il existait auparavant un mot-clé appelé « throw » avec la même sémantique que « reverse()``, qui a été déprécié dans la version 0.4.13 et supprimé dans la version 0.5.0.

---

try/catch

Il existait auparavant un mot-clé appelé « throw » avec la même sémantique que `reverse()`.

```
// 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) {
        // Désactiver définitivement le mécanisme s'il y a
        // plus de 10 erreurs.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            // Ceci est exécuté dans le cas où
            // le revert a été appelé dans getData
            // et qu'une chaîne de raison a été fournie.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // Ceci est exécuté en cas de panique,
            // c'est-à-dire une erreur grave comme une division par zéro
            // ou un dépassement de capacité. Le code d'erreur peut être utilisé
            // pour déterminer le type d'erreur.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // Ceci est exécuté au cas où revert() a été utilisé.
            errorCount++;
            return (0, false);
        }
    }
}
```

Le mot-clé `try` doit être suivi d'une expression représentant un appel de fonction externe ou une création de contrat (`new ContractName()`). Les erreurs à l'intérieur de l'expression ne sont pas prises en compte (par exemple s'il s'agit d'une expression complexe qui implique aussi des appels de fonctions internes), seul un retour en arrière se produisant dans l'appel externe lui-même. La partie `returns` (qui est optionnelle) qui suit déclare des variables de retour correspondant aux types retournés par l'appel externe. Dans le cas où il n'y a pas eu d'erreur ces variables sont assignées et l'exécution du contrat continue à l'intérieur du premier bloc de succès. Si la fin du bloc de succès est atteinte, l'exécution continue après les blocs `catch`.

Solidity prend en charge différents types de blocs catch en fonction du type d'erreur :

- `catch Error(string memory reason) { ... }` : Cette clause catch est exécutée si l'erreur a été provoquée par `revert("reasonString")` ou par `require(false, "reasonString")` (ou une erreur interne qui provoque une telle exception).
- `catch Panic(uint errorCode) { ... }` : Si l'erreur a été causée par une panique, c'est-à-dire par un `assert` défaillant, division par zéro, un accès invalide à un tableau, un débordement arithmétique et autres, cette clause catch sera exécutée.
- `catch (bytes memory lowLevelData) { ... }` : Cette clause est exécutée si la signature de l'erreur signature d'erreur ne correspond à aucune autre clause, s'il y a eu une erreur lors du décodage du message d'erreur, ou si aucune donnée d'erreur n'a été fournie avec l'exception. La variable déclarée donne accès aux données d'erreur de bas niveau dans ce cas.
- `catch { ... }` : Si vous n'êtes pas intéressé par les données d'erreur, vous pouvez simplement utiliser `catch { ... }` (même comme seule clause catch) au lieu de la clause précédente.

Il est prévu de supporter d'autres types de données d'erreur dans le futur. Les chaînes « Erreur » et « Panique » sont actuellement analysées telles quelles et ne sont pas traitées comme des identifiants.

Afin d'attraper tous les cas d'erreur, vous devez avoir au moins la clause suivante `catch { ...}` ou la clause `catch (bytes memory lowLevelData) { ... }`.

Les variables déclarées dans la clause `returns` et la clause `catch` sont uniquement dans le bloc qui suit.

Note : Si une erreur se produit pendant le décodage des données de retour dans un énoncé try/catch, cela provoque une exception dans le contrat en cours d'exécution et, pour cette raison, elle n'est pas attrapée dans la clause catch. S'il y a une erreur pendant le décodage de `catch Error(string memory reason)` et qu'il existe une clause catch de bas niveau, cette erreur y est attrapée.

Note : Si l'exécution atteint un bloc de capture, alors les effets de changement d'état de l'appel externe ont été annulés. Si l'exécution atteint le bloc de succès, les effets n'ont pas été annulés. Si les effets ont été inversés, alors l'exécution continue soit dans un bloc catch ou bien l'exécution de l'instruction try/catch elle-même s'inverse (par exemple, en raison d'échecs de décodage comme indiqué ci-dessus ou en raison de l'absence d'une clause catch de bas niveau).

Note : Les raisons de l'échec d'un appel peuvent être multiples. Ne supposez pas que le message d'erreur provient directement du contrat appelé : L'erreur peut s'être produite plus bas dans la chaîne d'appels et le contrat appelé n'a fait que la transmettre. De même, elle peut être due à une situation de panne sèche et non d'une condition d'erreur délibérée : L'appelant conserve toujours au moins 1/64ème du gaz dans un appel et donc l'appelant a encore du gaz.

3.9 Contrats

Les contrats dans Solidity sont similaires aux classes dans les langages orientés objet. Ils contiennent des données persistantes dans des variables d'état, et des fonctions qui peuvent modifier ces variables. L'appel d'une fonction sur un contrat (instance) différent va effectuer un appel de fonction EVM et donc un changement de contexte de telle sorte que les variables d'état dans le contrat appelant sont inaccessibles. Un contrat et ses fonctions doivent être appelés pour que quelque chose se produise. Il n'y a pas de concept de « cron » dans Ethereum pour appeler une fonction à un événement particulier automatiquement.

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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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))
            owner = newOwner;
    }
}


contract TokenCreator {
    function createToken(bytes32 name)
        public
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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

Solidity knows two kinds of function calls : internal ones that do not create an actual EVM call (also called a « message call ») and external ones that do. Because of that, there are four types of visibility for functions and state variables.

Functions have to be specified as being `external`, `public`, `internal` or `private`. For state variables, `external` is not possible.

`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 messages. For public state variables, an automatic getter function (see below) is generated.

`internal` Those functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it), without using `this`. This is the default visibility level for state variables.

`private` Private functions and state variables are only visible for the contract they are defined in and not in derived contracts.

Note : Everything that is inside a contract is visible to all observers external to the blockchain. Making something `private` 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; }
}

// 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
        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
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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;

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // 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 {
    // 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;
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.

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.

Avertissement : 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 Constant and Immutable State Variables

State variables can be declared as `constant` or `immutable`. In both cases, the variables cannot be modified after the contract has been constructed. For `constant` variables, the value has to be fixed at compile-time, while for `immutable`, it can still be assigned at construction time.

It is also possible to define `constant` variables at the file level.

The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

Compared to regular state variables, the gas costs of constant and immutable variables are much lower. For a constant variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time. This allows for local optimizations. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, constant values can sometimes be cheaper than immutable values.

Not all types for constants and immutables are implemented at this time. The only supported types are *strings* (only for constants) and *value types*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;

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 _reference) {
        decimals = _decimals;
        // Assignments to immutables can even access the environment.
        maxBalance = _reference.balance;
    }

    function isBalanceTooHigh(address _other) public view returns (bool) {
        return _other.balance > maxBalance;
    }
}
```

Constant

For `constant` variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. `block.timestamp`, `address(this).balance` or `block.number`) or execution data (`msg.value` or `gasleft()`) or makes calls to external contracts is disallowed. Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not. The built-in functions `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` and `mulmod` are allowed (even though, with the exception of `keccak256`, they do call external contracts).

The reason behind allowing side-effects on the memory allocator is that it should be possible to construct complex objects like e.g. lookup-tables. This feature is not yet fully usable.

Immutable

Variables declared as `immutable` are a bit less restricted than those declared as `constant` : Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They can be assigned only once and can, from that point on, be read even during construction time.

The contract creation code generated by the compiler will modify the contract's runtime code before it is returned by replacing all references to immutables by the values assigned to the them. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain.

Note : Immutables that are assigned at their declaration are only considered initialized once the constructor of the contract is executing. This means you cannot initialize immutables inline with a value that depends on another immutable. You can do this, however, inside the constructor of the contract.

This is a safeguard against different interpretations about the order of state variable initialization and constructor execution, especially with regards to inheritance.

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;
    }
}
```

Note : Functions defined outside a contract are still always executed in the context of a contract. They still have access to the variable `this`, 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 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;

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.

Note : An *external function* cannot accept a multi-dimensional array as an input parameter. This functionality is possible if you enable the ABI coder v2 by adding `pragma abicoder v2;` to your source file.

An *internal function* can accept a multi-dimensional array without enabling the feature.

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 o_sum, uint o_product)
    {
        o_sum = _a + _b;
        o_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 o_sum, uint o_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.

Note : You cannot return some types from non-internal functions, notably multi-dimensional dynamic arrays and structs. If you enable the ABI coder v2 by adding `pragma abicoder v2;` to your source file then more types are available, but `mapping` types are still limited to inside a single contract and you cannot transfer them.

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.

Note : 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 {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + block.timestamp;
    }
}
```

Note : `constant` on functions used to be an alias to `view`, but this was dropped in version 0.5.0.

Note : Getter methods are automatically marked `view`.

Note : 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.

Note : 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.

Avertissement : 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).

Note : 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.

Note : 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 regular transactions 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

Avertissement : Contracts that receive Ether directly (without a function call, i.e. using `send` or `transfer`) but do not define a receive Ether function or a payable fallback function throw an exception, sending back the Ether (this was different before Solidity v0.4.0). So 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 it would not fail on interface confusions).

Avertissement : 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.

Avertissement : 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.

Note : 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(
↪"nonExistingFunction()"));
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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 _in) public pure returns (uint out) {
        out = _in;
    }

    function f(uint _in, bool _really) public pure returns (uint out) {
        if (_really)
            out = _in;
    }
}
```

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 _in) public pure returns (B out) {
        out = _in;
    }

    function f(address _in) public pure returns (address out) {
        out = _in;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}

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.

Note : 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 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

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.

Note : 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.

```
// 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
    );

    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 Errors and the Revert Statement

Errors in Solidity provide a convenient and gas-efficient way to explain to the user why an operation failed. They can be defined inside and outside of contracts (including interfaces and libraries).

They have to be used together with the *revert statement* which causes all changes in the current call to be reverted and passes the error data back to the caller.

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

(suite sur la page suivante)

(suite de la page précédente)

```
/// 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;
    }
    // ...
}
```

Errors cannot be overloaded or overridden but are inherited. The same error can be defined in multiple places as long as the scopes are distinct. Instances of errors can only be created using `revert` statements.

The error creates data that is then passed to the caller with the revert operation to either return to the off-chain component or catch it in a *try/catch statement*. Note that an error can only be caught when coming from an external call, reverts happening in internal calls or inside the same function cannot be caught.

If you do not provide any parameters, the error only needs four bytes of data and you can use *NatSpec* as above to further explain the reasons behind the error, which is not stored on chain. This makes this a very cheap and convenient error-reporting feature at the same time.

More specifically, an error instance is ABI-encoded in the same way as a function call to a function of the same name and types would be and then used as the return data in the `revert` opcode. This means that the data consists of a 4-byte selector followed by *ABI-encoded* data. The selector consists of the first four bytes of the keccak256-hash of the signature of the error type.

Note : It is possible for a contract to revert with different errors of the same name or even with errors defined in different places that are indistinguishable by the caller. For the outside, i.e. the ABI, only the name of the error is relevant, not the contract or file where it is defined.

The statement `require(condition, "description");` would be equivalent to `if (!condition) revert Error("description")` if you could define `error Error(string)`. Note, however, that `Error` is a built-in type and cannot be defined in user-supplied code.

Similarly, a failing `assert` or similar conditions will revert with an error of the built-in type `Panic(uint256)`.

Note : Error data should only be used to give an indication of failure, but not as a means for control-flow. The reason is that the revert data of inner calls is propagated back through the chain of external calls by default. This means that an inner call can « forge » revert data that looks like it could have come from the contract that called it.

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;


contract Owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}


// 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);
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.
            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;

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 {
    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;

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(); }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}

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;

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
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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.

Note : Functions with the `private` visibility cannot be `virtual`.

Note : Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered `virtual`.

Note : 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
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function f() external view virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

Note : 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 {_;}
}

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) {
    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.

Avertissement : 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.

Avertissement : 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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) {}
}
```

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 will be abstract.

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() {}
}

// 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 Contrats abstraits

Les contrats doivent être marqués comme abstraits lorsqu'au moins une de leurs fonctions n'est pas implémentée. Les contrats peuvent être marqués comme abstraits même si toutes les fonctions sont implémentées.

Cela peut être fait en utilisant le mot-clé `abstract` comme le montre l'exemple suivant. Notez que ce contrat doit être défini comme abstrait, car la fonction `utterance()` a été définie, mais aucune implémentation n'a été fournie (aucun corps d'implémentation `{ }` n'a été donné).

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

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}
```

Ces contrats abstraits ne peuvent pas être instanciés directement. Cela est également vrai si un contrat abstrait met en œuvre toutes les fonctions définies. L'utilisation d'un contrat abstrait comme classe de base est illustrée dans l'exemple suivant :

```
// 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 contrat hérite d'un contrat abstrait et qu'il n'implémente pas toutes les fonctions non implémentées en les surchargeant, il doit également être marqué comme abstrait.

Notez qu'une fonction sans implémentation est différente d'une *Fonction Type*, même si leur syntaxe est très similaire.

Exemple de fonction sans implémentation (une déclaration de fonction) :

```
function foo(address) external returns (address);
```

Exemple de déclaration d'une variable dont le type est un type de fonction :

```
function(address) external returns (address) foo;
```

Les contrats abstraits découplent la définition d'un contrat de son implémentation fournissant une meilleure extensibilité et auto-documentation et facilitant les modèles comme la méthode Template et supprimant la duplication du code. Les contrats abstraits sont utiles de la même façon que définir des méthodes dans une interface est utile. C'est un moyen pour le concepteur du contrat abstrait de dire « tout enfant de moi doit implémenter cette méthode ».

Note : Les contrats abstraits ne peuvent pas remplacer une fonction virtuelle implémentée par une fonction virtuelle non implémentée.

3.9.10 Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented. There are further restrictions :

- They cannot inherit from other contracts, but they can inherit from other interfaces.
- All declared functions must be external.
- They cannot declare a constructor.
- They cannot declare state variables.
- They cannot declare modifiers.

Some of these restrictions might be lifted in the future.

Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

Interfaces are denoted by their own keyword :

```
// 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;
}
```

Contracts can inherit interfaces as they would inherit other contracts.

All functions declared in interfaces are implicitly `virtual` and any functions that override them do not need the `override` keyword. This does not automatically mean that an overriding function can be overridden again - this is only possible if the overriding function is marked `virtual`.

Interfaces can inherit from other interfaces. This has the same rules as normal inheritance.

```
// 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);
}
```

Types defined inside interfaces and other contract-like structures can be accessed from other contracts : `Token.TokenType` or `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.

Note : 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`.

Note : 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])
```

(suite sur la page suivante)

(suite de la page précédente)

```
            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)
        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 {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 a = limb(_a, i);
            uint b = limb(_b, i);
            unchecked {
                r.limbs[i] = a + b + carry;

                if (a + b < a || (a + b == 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 _limb) internal pure returns (uint) {
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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 *Liens entre les bibliothèques* 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 {}
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

The directive `using A for B;` can be used to attach library functions (from the library `A`) to any type (`B`) in the context of a contract. These functions will receive the object they are called on as their first parameter (like the `self` variable in Python).

The effect of `using A for *;` is that the functions from the library `A` are attached to *any* type.

In both situations, *all* functions in the library are attached, even those where the type of the first parameter does not match the type of the object. The type is checked at the point the function is called and function overload resolution is performed.

The `using A for B;` directive is active only within the current contract, including within all of its functions, and has no effect outside of the contract in which it is used. The directive may only be used inside a contract, not inside any of its functions.

Let us rewrite the set example from the *Libraries* in this way :

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


// This is the same code as before, just without comments
struct Data { mapping(uint => bool) flags; }

library Set {
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}


contract C {
    using Set for Data; // this is the crucial change
    Data knownValues;

    function register(uint value) public {
        // Here, all variables of type Data have
        // corresponding member functions.
        // The following function call is identical to
        // `Set.insert(knownValues, value)`
        require(knownValues.insert(value));
    }
}
```

It is also possible to extend elementary types in that way :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.8 <0.9.0;

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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return type(uint).max;
    }
}

contract C {
    using Search for uint[];
    uint[] data;

    function append(uint value) public {
        data.push(value);
    }

    function replace(uint _old, uint _new) public {
        // This performs the library function call
        uint index = data.indexOf(_old);
        if (index == type(uint).max)
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Note that all external library calls are actual EVM function calls. This means that if you pass memory or value types, a copy will be performed, even of the `self` variable. The only situation where no copy will be performed is when storage reference variables are used or when internal library functions are called.

3.10 Assemblage en ligne

Vous pouvez intercaler des instructions Solidity avec de l'assemblage en ligne dans un langage proche de celui de la machine virtuelle Ethereum. de celui de la machine virtuelle Ethereum. Cela vous donne un contrôle plus fin, ce qui est particulièrement utile lorsque vous améliorez le langage en écrivant des bibliothèques.

Le langage utilisé pour l'assemblage en ligne dans Solidity est appelé *Yul*. et il est documenté dans sa propre section. Cette section couvrira uniquement comment le code d'assemblage en ligne peut s'interfacer avec le code Solidity environnant.

Avertissement : L'assemblage en ligne est un moyen d'accéder à la machine virtuelle d'Ethereum à un faible niveau. Cela contourne plusieurs fonctions importantes de sécurité et de vérification de Solidity. Vous ne devez l'utiliser que pour des tâches qui en ont besoin, et seulement si vous avez confiance en son utilisation.

Un bloc d'assemblage en ligne est marqué par `assembly { ... }`, où le code à l'intérieur des les accolades est du code dans le langage *Yul*.

Le code d'assemblage en ligne peut accéder aux variables locales de Solidity comme expliqué ci-dessous.

Les différents blocs d'assemblage en ligne ne partagent aucun espace de nom, c'est-à-dire qu'il n'est pas possible d'appeler une fonction Yul ou d'accéder à des variables Solidity.

3.10.1 Exemple

L'exemple suivant fournit du code de bibliothèque pour accéder au code d'un autre contrat et le et de le charger dans une variable `bytes`. Ceci est également possible avec « plain Solidity », en utilisant `<adresse>.code`. Mais le point important ici est que les bibliothèques d'assemblage réutilisables peuvent améliorer le langage Solidity sans changer le compilateur. langage Solidity sans changer de compilateur.

```
// 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 o_code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(_addr)
            // allocate output byte array - this could also be done without assembly
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // store length in memory
            mstore(o_code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

L'assemblage en ligne est également bénéfique dans les cas où l'optimiseur ne parvient pas à produire code efficace, par exemple :

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


library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] memory _data) public pure returns (uint sum) {
        for (uint i = 0; i < _data.length; ++i)
            sum += _data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    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))))
            }
        }
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Same as above, but accomplish the entire code within inline assembly.
    function sumPureAsm(uint[] memory _data) public pure returns (uint sum) {
        assembly {
            // Load the length (first 32 bytes)
            let len := mload(_data)

            // Skip over the length field.
            //
            // Keep temporary variable so it can be incremented in place.
            //
            // NOTE: incrementing _data would result in an unusable
            //       _data variable after this assembly block
            let data := add(_data, 0x20)

            // Iterate until the bound is not met.
            for
                { let end := add(data, mul(len, 0x20)) }
                lt(data, end)
                { data := add(data, 0x20) }
            {
                sum := add(sum, mload(data))
            }
        }
    }
}
```

3.10.2 Accès aux variables, fonctions et bibliothèques externes

Vous pouvez accéder aux variables Solidity et autres identifiants en utilisant leur nom.

Les variables locales de type valeur sont directement utilisables dans l'assemblage en ligne. Elles peuvent à la fois être lues et assignées.

Les variables locales qui font référence à la mémoire sont évaluées à l'adresse de la variable en mémoire et non à la valeur elle-même. Ces variables peuvent également être assignées, mais notez qu'une assignation ne modifie que le pointeur et non les données. et qu'il est de votre responsabilité de respecter la gestion de la mémoire de Solidity. Voir *Conventions dans Solidity*.

De même, les variables locales qui font référence à des tableaux de calldonnées ou à des structures de calldonnées de taille statique sont évaluées à l'adresse de la variable dans calldata, et non à la valeur elle-même. La variable peut également être assignée à un nouveau décalage, mais notez qu'aucune validation pour assurer que que la variable ne pointera pas au-delà de `calldatasize()` n'est effectuée.

Pour les pointeurs de fonctions externes, l'adresse et le sélecteur de fonction peuvent être accessible en utilisant `x.address` et `x.selector`. Le sélecteur est constitué de quatre octets alignés à droite. Les deux valeurs peuvent être assignées. Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.10 <0.9.0;

contract C {
    // Assigns a new selector and address to the return variable @fun
    function combineToFunctionPointer(address newAddress, uint newSelector) public pure␣
↪returns (function() external fun) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        assembly {
            fun.selector := newSelector
            fun.address  := newAddress
        }
    }
}
```

Pour les tableaux de calldonnées dynamiques, vous pouvez accéder à leur offset (en octets) et leur longueur (nombre d'éléments) en utilisant `x.offset` et `x.length`. Les deux expressions peuvent également être assignées à, mais comme pour le cas statique, aucune validation ne sera effectuée pour s'assurer que la zone de données résultante est dans les limites de `calldatasize()`.

Pour les variables de stockage local ou les variables d'état, un seul identifiant Yul n'est pas suffisant, car elles n'occupent pas nécessairement un seul emplacement de stockage complet. Par conséquent, leur « adresse » est composée d'un slot et d'un byte-offset à l'intérieur de cet emplacement. Pour récupérer le slot pointé par la variable `x`, on utilise vous utilisez `x.slot`, et pour récupérer le byte-offset vous utilisez `x.offset`. L'utilisation de la variable `x` elle-même entraînera une erreur.

Vous pouvez également assigner à la partie `.slot` d'un pointeur de variable de stockage local. Pour celles-ci (structs, arrays ou mappings), la partie `.offset` est toujours zéro. Il n'est pas possible d'assigner à la partie `.slot` ou ``.offset`"" d'une variable d'état, cependant.

Les variables locales de Solidity sont disponibles pour les affectations, par exemple :

```
// 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 {
            // We ignore the storage slot offset, we know it is zero
            // in this special case.
            r := mul(x, sload(b.slot))
        }
    }
}
```

Avertissement : Si vous accédez à des variables d'un type qui s'étend sur moins de 256 bits (par exemple `uint64`, `adresse`, ou `bytes16`), vous ne pouvez pas faire d'hypothèses sur les bits qui ne font pas partie du codage du type. En particulier, ne supposez pas qu'ils soient nuls. Pour être sûr, effacez toujours les données correctement avant de les utiliser dans un contexte où cela est important : `uint32 x = f() ; assembly { x := and(x, 0xffffffff) /* maintenant utiliser x */ }` Pour nettoyer les types signés, vous pouvez utiliser l'opcode `signextend` : `assembly { signextend(<nombre_bytes_de_x_moins_un>, x) }`

Depuis Solidity 0.6.0, le nom d'une variable d'assemblage en ligne ne peut pas suivre aucune déclaration visible dans la portée du bloc d'assemblage en ligne (y compris les déclarations de variables, de contrats et de fonctions).

Depuis la version 0.7.0 de Solidity, les variables et les fonctions déclarées à l'intérieur du bloc d'assemblage en ligne ne peuvent pas contenir `.`, mais l'utilisation de `.` est valide valide pour accéder aux variables Solidity depuis l'extérieur du bloc d'assemblage en ligne.

3.10.3 Choses à éviter

L'assemblage en ligne peut avoir une apparence de haut niveau, mais il est en fait extrêmement bas niveau. Les appels de fonction, les boucles, les ifs et les switchs sont convertis par de simples règles de réécriture. règles de réécriture et après cela, la seule chose que l'assembleur fait pour vous est de réarranger opcodes de style fonctionnel, le comptage de la hauteur de la pile pour pour l'accès aux variables et la suppression des emplacements de pile pour les variables locales à l'assemblage lorsque la fin de leur bloc est atteinte.

3.10.4 Conventions dans Solidity

Contrairement à l'assemblage EVM, Solidity possède des types dont la taille est inférieure à 256 bits, par exemple uint24. Pour des raisons d'efficacité, la plupart des opérations arithmétiques ignorent le fait que les types peuvent être plus courts que 256 bits, et les bits d'ordre supérieur sont nettoyés lorsque cela est nécessaire, c'est-à-dire peu de temps avant qu'ils ne soient écrits en mémoire ou avant que les comparaisons ne soient effectuées. Cela signifie que si vous accédez à une telle variable à partir d'un assemblage en ligne, vous devrez peut-être d'abord nettoyer manuellement les bits d'ordre supérieur.

Solidity gère la mémoire de la manière suivante. Il existe un » pointeur de mémoire libre » à la position 0x40 dans la mémoire. Si vous voulez allouer de la mémoire, utilisez la mémoire à partir de l'endroit où pointe ce pointeur et mettez-la à jour. Il n'y a aucune garantie que la mémoire n'a pas été utilisée auparavant et vous ne pouvez donc pas supposer que son contenu est de zéro octet. Il n'existe pas de mécanisme intégré pour libérer la mémoire allouée. Voici un extrait d'assemblage que vous pouvez utiliser pour allouer de la mémoire qui suit le processus décrit ci-dessus.

```
function allocate(length) -> pos {
  pos := mload(0x40)
  mstore(0x40, add(pos, length))
}
```

Les 64 premiers octets de la mémoire peuvent être utilisés comme « espace de grattage » pour l'allocation à court terme. Les 32 octets après le pointeur de mémoire libre (c'est-à-dire, à partir de `0x60`) sont censés être zéro de manière permanente et sont utilisés comme valeur initiale pour les tableaux de mémoire dynamique vides. Cela signifie que la mémoire allouable commence à 0x80, qui est la valeur initiale du pointeur de mémoire libre.

Les éléments des tableaux de mémoire dans Solidity occupent toujours des multiples de 32 octets (c'est même vrai pour les « octets »). Même vrai pour `bytes1[]`, mais pas pour `bytes` et `string`). Les tableaux de mémoire multidimensionnels sont des pointeurs vers des tableaux de mémoire. La longueur d'un tableau dynamique est stockée dans le premier emplacement du tableau et suivie par les éléments du tableau.

Avertissement : Les tableaux de mémoire de taille statique n'ont pas de champ de longueur, mais celui-ci pourrait être ajouté ultérieurement pour permettre une meilleure convertibilité entre les tableaux de taille statique et dynamique. Pour permettre une meilleure convertibilité entre les tableaux de taille statique et dynamique. Donc ne vous y fiez pas.

3.11 Aide-mémoire

3.11.1 Ordre de Préséance des Opérateurs

Voici l'ordre de préséance des opérateurs, classés par ordre d'évaluation.

Prédominance	Description	Opérateur
1	Incrément et décrément de Postfix	`++`, `--`
	Nouvelle expression	`new <nomdutilisateur>`
	Subscription de tableau	`<array>[<index>]`
	Accès des membres	`<objet>.<membre>`
	Appel de type fonctionnel	`<func>(<args...>)`
	Parenthèses	`(<déclaration>)`
2	Préfixe d'incrémentation et de décrémentation	`++`, `--`
	Moins unaire	`-`
	Opérations unaires	`delete`
	Logique NON	`!`
	NON par bit	`~`
3	Exponentité	`**`
4	Multiplication, division et modulo	`*`, `/`, `%`
5	Addition et soustraction	`+`, `-`
6	Opérateurs de décalage par bit	`<<`, `>>`
7	ET par bit	`&`
8	XOR par bit	`^`
9	OU par bit	`|`
10	Opérateurs d'inégalité	`<`, `>`, `<=`, `>=`
11	Opérateurs d'égalité	`==`, `!=`
12	ET logique	`&&`
13	OU logique	`||`
14	Opérateur ternaire	`<conditional> ? <if-true> : <if-false>`
	Opérateurs d'assignation	`=`, `|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=`
15	Opérateur de virgule	`,`

3.11.2 Variables Globales

— `abi.decode(bytes memory encodedData, (...)) returns (...)` : *ABI*-décode les données fournies. Les types sont donnés entre parenthèses comme deuxième argument. Exemple : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
— `abi.encode(...) returns (bytes memory)` : *ABI*-encode les arguments donnés
— `abi.encodePacked(...) returns (bytes memory)` : Performe l'"*encodage emballé* des arguments donnés. Notez que cet encodage peut être ambigu !
— `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` : *ABI*-encode les arguments donnés en commençant par le deuxième et en ajoutant au début le sélecteur de quatre octets donné.
— `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)` : ABI-encode un appel à `functionPointer` avec les arguments trouvés dans le tuple. Effectue une vérification complète des types, en s'assurant que les types correspondent à la signature de la fonction. Le résultat est égal à `abi.encodeWithSelector(functionPointer.selector, (...))`

— `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)` : Equivalent à `abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...)`
— `bytes.concat(...) returns (bytes memory)` : *Concatène un nombre variable d'arguments d'arguments dans un tableau d'un octet*
— `block.basefee` (`uint`) : redevance de base du bloc actuel (EIP-3198 et EIP-1559)
— `block.chainid` (`uint`) : identifiant de la chaîne actuelle
— `block.coinbase` (`address payable`) : adresse du mineur du bloc actuel
— `block.difficulty` (`uint`) : difficulté actuelle du bloc
— `block.gaslimit` (`uint`) : limite de gaz du bloc actuel
— `block.number` (`uint`) : numéro du bloc actuel
— `block.timestamp` (`uint`) : Horodatage du bloc actuel
— `gasleft() returns (uint256)` : gaz résiduel
— `msg.data` (`bytes`) : données d'appel complètes
— `msg.sender` (`address`) : expéditeur du message (appel en cours)
— `msg.value` (`uint`) : nombre de wei envoyés avec le message
— `tx.gasprice` (`uint`) : prix du gaz de la transaction
— `tx.origin` (`address`) : expéditeur de la transaction (chaîne d'appel complète)
— `assert(bool condition)` : interrompt l'exécution et annule les changements d'état si la condition est « fausse » (à utiliser pour les erreurs internes).
— `require(bool condition)` : interrompre l'exécution et annuler les changements d'état si la condition est « fausse » (à utiliser pour une entrée malformée ou une erreur dans un composant externe)
— `require(bool condition, string memory message)` : interrompt l'exécution et annule les changements d'état si la condition est « fausse » (à utiliser en cas d'entrée malformée ou d'erreur dans un composant externe). Fournit également un message d'erreur.
— `revert()` : interrompre l'exécution et revenir sur les changements d'état
— `revert(string memory message)` : interrompre l'exécution et revenir sur les changements d'état en fournissant une chaîne explicative
— `blockhash(uint blockNumber) returns (bytes32)` : hachage du bloc donné - ne fonctionne que pour les 256 blocs les plus récents
— `keccak256(bytes memory) returns (bytes32)` : calculer le hachage Keccak-256 de l'entrée
— `sha256(bytes memory) returns (bytes32)` : calculer le hachage SHA-256 de l'entrée
— `ripemd160(bytes memory) returns (bytes20)` : calculer le hachage RIPEMD-160 de l'entrée
— `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)` : récupérer l'adresse associée à la clé publique de la signature de la courbe elliptique, renvoie zéro en cas d'erreur
— `addmod(uint x, uint y, uint k) returns (uint)` : compute `(x + y) % k` où l'addition est effectuée avec une précision arbitraire et ne s'arrête pas à `2**256`. Affirmer que `k != 0` à partir de la version 0.5.0.
— `mulmod(uint x, uint y, uint k) returns (uint)` : compute `(x * y) % k` où la multiplication est effectuée avec une précision arbitraire et ne s'arrête pas à `2**256`. Affirmer que `k != 0` à partir de la version 0.5.0.
— `this` (current contract's type) : le contrat en cours, explicitement convertible en « adresse » ou « adresse payable ».
— `super` : le contrat un niveau plus haut dans la hiérarchie d'héritage
— `selfdestruct(address payable recipient)` : détruire le contrat en cours, en envoyant ses fonds à l'adresse donnée
— `<address>.balance` (`uint256`) : solde de la *Address* dans Wei
— `<address>.code` (`bytes memory`) : le code à *Address* (peut être vide)
— `<address>.codehash` (`bytes32`) : le codehash de l'adresse *Address*
— `<address payable>.send(uint256 amount) returns (bool)` : envoie une quantité donnée de Wei à *Address*, renvoie `false` en cas d'échec
— `<address payable>.transfer(uint256 amount)` : envoie une quantité donnée de Wei à *Address*, lance en cas d'échec
— `type(C).name` (`string`) : le nom du contrat
— `type(C).creationCode` (`bytes memory`) : bytecode de création du contrat donné, voir *Type Information*.
— `type(C).runtimeCode` (`bytes memory`) : le bytecode d'exécution du contrat donné, voir *Type Information*.

— `type(I).interfaceId` (`bytes4`) : contenant l'identificateur d'interface EIP-165 de l'interface donnée, voir *Type Information*.
— `type(T).min` (`T`) : la valeur minimale représentable par le type entier `T`, voir *Type Information*.
— `type(T).max` (`T`) : la valeur maximale représentable par le type entier `T`, voir *Type Information*.

Note : Lorsque les contrats sont évalués hors chaîne plutôt que dans le contexte d'une transaction comprise dans un bloc, vous ne devez pas supposer que `block.*` et `tx.*` font référence à des valeurs d'un bloc ou d'une transaction d'un bloc ou d'une transaction spécifique. Ces valeurs sont fournies par l'implémentation EVM qui exécute le contrat et peuvent être arbitraires. contrat et peuvent être arbitraires.

Note : Ne comptez pas sur `block.timestamp` ou `blockhash` comme source d'aléatoire, à moins que vous ne sachiez ce que vous faites.

L'horodatage et le hachage du bloc peuvent tous deux être influencés par les mineurs dans une certaine mesure. De mauvais acteurs dans la communauté minière peuvent par exemple exécuter une fonction de paiement de casino sur un hash choisi et réessayer un autre hash s'ils n'ont pas reçu d'argent.

L'horodatage du bloc actuel doit être strictement plus grand que l'horodatage du dernier bloc, mais la seule garantie est qu'il se situera quelque part entre les horodatages de deux blocs consécutifs dans la chaîne canonique.

Note : Les hachages des blocs ne sont pas disponibles pour tous les blocs pour des raisons d'évolutivité. Vous ne pouvez accéder qu'aux hachages des 256 blocs les plus récents. autres valeurs seront nulles.

Note : Dans la version 0.5.0, les alias suivants ont été supprimés : `suicide` comme alias pour `selfdestruct`, `msg.gas` comme alias pour `gasleft`, `block.blockhash` comme alias pour `blockhash` et `sha3` comme alias pour `keccak256`.

Note : Dans la version 0.7.0, l'alias `now` (pour `block.timestamp`) a été supprimé.

3.11.3 Spécification de la Visibilité des Fonctions

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

— `public` : visible en externe et en interne (crée une *fonction réceptrice* pour les variables de stockage/d'état)
— `private` : uniquement visible dans le contrat en cours
— `external` : visible uniquement en externe (uniquement pour les fonctions) - c'est-à-dire qu'il ne peut être appelé que par message (via `this.func`)
— `internal` : uniquement visible en interne

3.11.4 Modificateurs

— `pure` pour les fonctions : Interdit la modification ou l'accès à l'état.
— `view` pour les fonctions : Interdit la modification de l'état.
— `payable` pour les fonctions : Leur permet de recevoir de l'Ether en même temps qu'un appel.
— `constant` pour les variables d'état : Ne permet pas l'affectation (sauf l'initialisation), n'occupe pas d'emplacement de stockage.
— `immutable` pour les variables d'état : Permet exactement une affectation au moment de la construction et est constante par la suite. Est stockée dans le code.
— `anonymous` pour les événements : Ne stocke pas la signature de l'événement comme sujet.
— `indexed` pour les paramètres d'événements : Stocke le paramètre en tant que sujet.
— `virtual` pour les fonctions et les modificateurs : Permet de modifier le comportement de la fonction ou du modificateur dans les contrats dérivés.
— `override` : Indique que cette fonction, ce modificateur ou cette variable d'état publique change le comportement d'une fonction ou d'un modificateur dans un contrat de base.

3.11.5 Mots clés réservés

Ces mots-clés sont réservés dans Solidity. Ils pourraient faire partie de la syntaxe à l'avenir :

`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`.

3.12 Utilisation du compilateur

3.12.1 Utilisation du compilateur en ligne de commande

Note : Cette section ne s'applique pas à *solcjs*, même s'il est utilisé en mode ligne de commande.

Utilisation de base

L'une des cibles de construction du référentiel Solidity est `solc`, le compilateur en ligne de commande de Solidity. L'utilisation de `solc --help` vous fournit une explication de toutes les options. Le compilateur peut produire diverses sorties, allant de simples binaires et assemblages sur un arbre syntaxique abstrait (parse tree) à des estimations de l'utilisation du gaz. Si vous voulez seulement compiler un seul fichier, vous le lancez comme `solc --bin sourceFile.sol` et il imprimera le binaire. Si vous voulez obtenir certaines des variantes de sortie plus avancées de `solc`, il est probablement préférable de lui dire de tout sortir dans des fichiers séparés en utilisant `solc -o outputDirectory --bin --ast-compact-json --asm sourceFile.sol`.

Options de l'optimiseur

Avant de déployer votre contrat, activez l'optimiseur lors de la compilation en utilisant `solc --optimize --bin sourceFile.sol`. Par défaut, l'optimiseur optimisera le contrat en supposant qu'il est appelé 200 fois au cours de sa durée de vie (plus précisément, il suppose que chaque opcode est exécuté environ 200 fois). Si vous voulez que le déploiement initial du contrat soit moins cher et que les exécutions de fonctions ultérieures soient plus coûteuses, définissez-le à `--optimize-runs=1`. Si vous vous attendez à de nombreuses transactions et que vous ne vous souciez pas des coûts de la taille de la sortie, définissez `--optimize-runs` à un nombre élevé. Ce paramètre a des effets sur les éléments suivants (cela pourrait changer dans le futur) :

— la taille de la recherche binaire dans la routine d'envoi des fonctions
— la façon dont les constantes comme les grands nombres ou les chaînes de caractères sont stockées.

Chemin de base et remappage des importations

Le compilateur en ligne de commande lira automatiquement les fichiers importés depuis le système de fichiers, mais il est également possible de fournir des redirections *path* en utilisant `prefix=path` de la manière suivante :

```
solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ file.sol
```

Ceci indique essentiellement au compilateur de rechercher tout ce qui commence par `github.com/ethereum/dapp-bin/` sous `/usr/local/lib/dapp-bin`.

Lorsque vous accédez au système de fichiers pour rechercher des importations, les *chemins qui ne commencent pas par ./ ou ../* sont traités comme relatifs aux répertoires spécifiés en utilisant les options `--base-path` et `-include-path` (ou le répertoire de travail actuel si le chemin de base n'est pas spécifié). De plus, la partie du chemin ajoutée via ces options n'apparaîtra pas dans les métadonnées du contrat.

Pour des raisons de sécurité, le compilateur a des *restrictions sur les répertoires auxquels il peut accéder*. Les répertoires des fichiers sources spécifiés sur la ligne de commande et les chemins cibles des remappings sont automatiquement autorisés à être accédés par le lecteur de fichiers, mais tout le reste est rejeté par défaut. Des chemins supplémentaires (et leurs sous-répertoires) peuvent être autorisés via la commande `--allow-paths /sample/path,/another/sample/path`. Tout ce qui se trouve à l'intérieur du chemin spécifié par `--base-path` est toujours autorisé.

Ce qui précède n'est qu'une simplification de la façon dont le compilateur gère les chemins d'importation. Pour une explication détaillée avec des exemples et une discussion des cas de coin, veuillez vous référer à la section sur *résolution de chemin*.

Liens entre les bibliothèques

Si vos contrats utilisent *libraries*, vous remarquerez que le bytecode contient des sous-chaînes de la forme `__$53aea86b7d70b31448b230b20ae141a537$__`. Il s'agit de caractères de remplacement pour les adresses réelles des bibliothèques. Le placeholder est un préfixe de 34 caractères de l'encodage hexadécimal du hachage keccak256 du nom de bibliothèque entièrement qualifié. Le fichier de bytecode contiendra également des lignes de la forme `// <placeholder> -> <fq library name>` à la fin pour aider à identifier les bibliothèques que les placeholders représentent. Notez que le nom de bibliothèque pleinement qualifié est le chemin de son fichier source et le nom de la bibliothèque séparés par `:`. Vous pouvez utiliser `solc` comme linker, ce qui signifie qu'il insérera les adresses des bibliothèques pour vous à ces endroits :

Soit vous ajoutez `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890 file.sol:Heap=0xabCD5678901234567890123458901234567890"` à votre commande pour fournir une adresse pour chaque bibliothèque (utilisez des virgules ou des espaces comme séparateurs) ou stockez la chaîne dans un fichier (une bibliothèque par ligne) et lancez `solc` en utilisant *–libraries fileName* `.

Note : À partir de la version 0.8.1 de Solidity, on accepte `=` comme séparateur entre bibliothèque et adresse, et `:` comme séparateur est déprécié. Il sera supprimé à l'avenir. Actuellement,

`--libraries "file.sol:Math:0x1234567890123456789012345678901234589012345678901234567890 file.sol:Heap:0xabCD5678901234567890123458901234567890"` fonctionnera également.

Si `solc` est appelé avec l'option `--standard-json`, il attendra une entrée JSON (comme expliqué ci-dessous) sur l'entrée standard, et retournera une sortie JSON sur la sortie standard. C'est l'interface recommandée pour des utilisations plus complexes et particulièrement automatisées. Le processus se terminera toujours dans un état « success » et rapportera toute erreur via la sortie JSON. L'option `--base-path` est également traitée en mode standard-json.

Si `solc` est appelé avec l'option `--link`, tous les fichiers d'entrée sont interprétés comme des binaires non liés (encodés en hexadécimal) dans le format `__$53aea86b7d70b31448b230b20ae141a537$__` donné ci-dessus et sont liés in-place (si l'entrée est lue depuis stdin, elle est écrite sur stdout). Toutes les options sauf `--libraries` sont ignorées (y compris `-o`) dans ce cas.

Avertissement : La liaison manuelle des bibliothèques sur le bytecode généré est déconseillée car elle ne permet pas de mettre à jour les métadonnées du contrat. Puisque les métadonnées contiennent une liste de bibliothèques spécifiées au moment de la compilation et le bytecode contient un hash de métadonnées, vous obtiendrez des binaires différents, selon du moment où la liaison est effectuée.

Vous devez demander au compilateur de lier les bibliothèques au moment où un contrat est compilé, soit en en utilisant l'option `--libraries` de `solc` ou la clé `libraries` si vous utilisez l'interface standard-JSON au compilateur.

Note : L'espace réservé à la bibliothèque était auparavant le nom pleinement qualifié de la bibliothèque elle-même au lieu du hash de celui-ci. Ce format est toujours pris en charge par `solc --link` mais mais le compilateur ne l'affichera plus. Ce changement a été fait pour réduire la probabilité de collision entre les bibliothèques, puisque seuls les 36 premiers caractères du nom de du nom complet de la bibliothèque pouvaient être utilisés.

3.12.2 Réglage de la version de l'EVM sur la cible

Lorsque vous compilez le code de votre contrat, vous pouvez spécifier la version de la machine virtuelle d'Ethereum pour laquelle compiler afin d'éviter des caractéristiques ou des comportements particuliers.

Avertissement : La compilation pour la mauvaise version EVM peut entraîner un comportement erroné, étrange et défaillant. Veuillez vous assurer, en particulier si vous exécutez une chaîne privée, que vous utilisez les versions EVM correspondantes.

Sur la ligne de commande, vous pouvez sélectionner la version EVM comme suit :

```
solc --evm-version <VERSION> contract.sol
```

Dans l'interface *standard JSON*, utilisez la clé `"evmVersion"` dans le champ `"settings"` :

```
{
  "sources": {/* ... */},
  "settings": {
    "optimizer": {/* ... */},
    "evmVersion": "<VERSION>"
  }
}
```

Options de la cible

Vous trouverez ci-dessous une liste des versions EVM cibles et des modifications relatives au compilateur introduites à chaque version. La rétrocompatibilité n'est pas garantie entre chaque version.

- **homestead**
  - (version la plus ancienne)
- **tangerineWhistle**
  - Le coût du gaz pour l'accès à d'autres comptes a augmenté, ce qui est pertinent pour l'estimation du gaz et l'optimiseur.
  - Tout le gaz est envoyé par défaut pour les appels externes, auparavant une certaine quantité devait être conservée.
- **spuriousDragon**
  - Le coût du gaz pour l'opcode `exp` a augmenté, ce qui est important pour l'estimation du gaz et l'optimiseur.
- **byzantium**
  - Les opcodes `returndatacopy`, `returndatasize` et `staticcall` sont disponibles en assembly.
  - L'opcode `staticcall` est utilisé lors de l'appel de fonctions de vue ou de fonctions pures non libérées, ce qui empêche les fonctions de modifier l'état au niveau de l'EVM, c'est-à-dire qu'il s'applique même lorsque vous utilisez des conversions de type invalides.
  - Il est possible d'accéder aux données dynamiques renvoyées par les appels de fonctions.
  - Introduction de l'opcode `revert`, ce qui signifie que `revert()` ne gaspillera pas de gaz.
- **constantinople**
  - Les opcodes `create2`, `extcodehash''`, ``shl`, `shr` et `sar` sont disponibles en assembleur.
  - Les opérateurs de décalage utilisent des opcodes de décalage et nécessitent donc moins de gaz.
- **petersburg**
  - Le compilateur se comporte de la même manière qu'avec constantinople.
- **istanbul**
  - Les opcodes `chainid` et ``selfbalance`" sont disponibles en assemblage.
- **berlin**
  - Les coûts du gaz pour `LOAD`, `*CALL`, `BALANCE`, `EXT` et `SELFDESTRUCT` ont augmenté. Le compilateur suppose des coûts de gaz froid pour de telles opérations. Ceci est pertinent pour l'estimation des gaz et l'optimiseur.
- **london (default)**
  - Le tarif de base du bloc (EIP-3198 et EIP-1559) est accessible via le global `block.basefee` ou `basefee()` en assemblage inline.

3.12.3 Description JSON des entrées et sorties du compilateur

La manière recommandée de s'interfacer avec le compilateur Solidity, surtout pour les configurations plus complexes et automatisées est l'interface dite d'entrée-sortie JSON. La même interface est fournie par toutes les distributions du compilateur.

Les champs sont généralement susceptibles d'être modifiés, certains sont optionnels (comme indiqué), mais nous essayons de ne faire que des changements compatibles avec le passé.

L'API du compilateur attend une entrée au format JSON et produit le résultat de la compilation dans une sortie au format JSON. La sortie d'erreur standard n'est pas utilisée et le processus se terminera toujours dans un état de « succès », même s'il y a eu des erreurs. Les erreurs sont toujours signalées dans le cadre de la sortie JSON.

Les sous-sections suivantes décrivent le format à travers un exemple. Les commentaires ne sont bien sûr pas autorisés et sont utilisés ici uniquement à des fins explicatives.

Description de l'entrée

```
{
  // Requis : Langue du code source. Les langages actuellement pris en charge sont
↪"Solidity" et "Yul".
  "language": "Solidity",
  // Requis
  "sources":
  {
    // Les clés ici sont les noms "globaux" des fichiers sources,
    // les importations peuvent utiliser d'autres fichiers via les remappings (voir ci-
↪dessous).
    "myFile.sol":
    {
      // Facultatif : hachage keccak256 du fichier source.
      // Il est utilisé pour vérifier le contenu récupéré s'il est importé via des URL.
      "keccak256": "0x123...",
      // Obligatoire (sauf si "content" est utilisé, voir ci-dessous) : URL(s) vers le␣
↪fichier source.
      // Les URL doivent être importées dans cet ordre et le résultat doit être vérifié␣
↪par rapport à l'empreinte
      // le hachage keccak256 (si disponible). Si le hachage ne correspond pas ou si␣
↪aucune des
      // URL n'aboutit à un succès, une erreur doit être signalée.
      // En utilisant l'interface en ligne de commande, seuls les chemins de systèmes de␣
↪fichiers sont pris en charge.
      // Avec l'interface JavaScript, l'URL sera transmise au callback de lecture fourni␣
↪par l'utilisateur.
      // Ainsi, toute URL prise en charge par le callback peut être utilisée.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "/tmp/path/to/file.sol"
        // Si des fichiers sont utilisés, leurs répertoires doivent être ajoutés à la␣
↪ligne de commande via
        // `--allow-paths <path>`.
      ]
    },
    "destructible":
    {
      // Facultatif : keccak256 hash du fichier source
      "keccak256": "0x234...",
      // Obligatoire (sauf si "urls" est utilisé) : contenu littéral du fichier source
      "content": "contract destructible is owned { function shutdown() { if (msg.sender␣
↪== owner) selfdestruct(owner); } }"
    }
  },
  // Optionnel
  "settings":
  {
    // Facultatif : Arrête la compilation après l'étape donnée. Actuellement, seul
↪"parsing" est valide ici
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "stopAfter": "parsing",
    // Facultatif : Liste triée de réaffectations
    "remappings": [ ":g=/dir" ],
    // Facultatif : Paramètres de l'optimiseur
    "optimizer": {
      // Désactivé par défaut.
      // NOTE : enabled=false laisse toujours certaines optimisations activées. Voir les␣
↪commentaires ci-dessous.
      // ATTENTION : Avant la version 0.8.6, l'omission de la clé 'enabled' n'était pas␣
↪équivalente à la mise en place de la clé 'enabled'.
      // l'activer à false et désactiverait en fait toutes les optimisations.
      "enabled": true,
      // Optimisez en fonction du nombre de fois que vous avez l'intention d'exécuter le␣
↪code.
      // Les valeurs les plus basses optimisent davantage le coût de déploiement initial,
↪ les valeurs les plus élevées optimisent davantage les utilisations à haute fréquence.
      // Plus les valeurs sont faibles, plus l'optimisation est axée sur le coût du␣
↪déploiement initial.
      // Plus les valeurs sont élevées, plus l'optimisation est axée sur un usage␣
↪fréquent.
      "runs": 200,
      // Activez ou désactivez les composants de l'optimiseur en détail.
      // L'interrupteur "enabled" ci-dessus fournit deux valeurs par défaut qui peuvent␣
↪être
      // modifiables ici. Si "details" est donné, "enabled" peut être omis.
      "details": {
        // L'optimiseur de trou d'homme est toujours activé si aucun détail n'est donné,
        // utilisez les détails pour le désactiver.
        "peephole": true,
        // L'inliner est toujours activé si aucun détail n'est donné,
        // utilisez les détails pour le désactiver.
        "inliner": true,
        // L'enlèvement du jumpdest inutilisé est toujours activé si aucun détail n'est␣
↪donné,
        // utilisez les détails pour le désactiver.
        "jumpdestRemover": true,
        // Réorganise parfois les littéraux dans les opérations commutatives.
        "orderLiterals": false,
        // Supprime les blocs de code dupliqués
        "deduplicate": false,
        // L'élimination des sous-expressions communes, c'est l'étape la plus compliquée␣
↪mais
        // peut également fournir le gain le plus important.
        "cse": false,
        // Optimiser la représentation des nombres littéraux et des chaînes de␣
↪caractères dans le code.
        "constantOptimizer": false,
        // Le nouvel optimiseur Yul. Opère principalement sur le code du codeur ABI v2
        // et de l'assemblage en ligne.
        // Il est activé en même temps que le réglage de l'optimiseur global
        // et peut être désactivé ici.
        // Avant Solidity 0.6.0, il devait être activé par ce commutateur.
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "yul": false,
        // Tuning options for the Yul optimizer.
        "yulDetails": {
          // Améliore l'allocation des emplacements de pile pour les variables, peut␣
↪libérer les emplacements de pile plus tôt.
          // Activé par défaut si l'optimiseur Yul est activé.
          "stackAllocation": true,
          // Sélectionnez les étapes d'optimisation à appliquer.
          // Facultatif, l'optimiseur utilisera la séquence par défaut si elle est omise.
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    // Version de l'EVM pour laquelle il faut compiler.
    // Affecte la vérification de type et la génération de code. Peut être homestead,
    // tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg, istanbul␣
↪ou berlin.
    "evmVersion": "byzantium",
    // Facultatif : Modifier le pipeline de compilation pour passer par la␣
↪représentation intermédiaire de Yul.
    // Il s'agit d'une fonctionnalité hautement EXPERIMENTALE, à ne pas utiliser en␣
↪production. Elle est désactivée par défaut.
    "viaIR": true,
    // Facultatif : Paramètres de débogage
    "debug": {
      // Comment traiter les chaînes de motifs de retour (et d'exigence). Les paramètres␣
↪sont
      // "default", "strip", "debug" et "verboseDebug".
      // "default" n'injecte pas les chaînes revert générées par le compilateur et␣
↪conserve celles fournies par l'utilisateur.
      // "strip" supprime toutes les chaînes revert (si possible, c'est-à-dire si des␣
↪littéraux sont utilisés) en conservant les effets secondaires.
      // "debug" injecte des chaînes pour les revert internes générés par le compilateur,
↪ implémenté pour les encodeurs ABI V1 et V2 pour le moment.
      // "verboseDebug" ajoute même des informations supplémentaires aux chaînes de␣
↪revert fournies par l'utilisateur (pas encore implémenté).
      "revertStrings": "default",
      // Facultatif : quantité d'informations de débogage supplémentaires à inclure dans␣
↪les commentaires de l'EVM
      // produit et dans le code Yul. Les composants disponibles sont :
      // - `location` : Annotations de la forme `@src <index>:<start>:<end>` indiquant
      // l'emplacement de l'élément correspondant dans le fichier Solidity original, où :
      // - `<index>` est l'index du fichier correspondant à l'annotation `@use-src`,
      // - `<start>` est l'indice du premier octet à cet emplacement,
      // - `<end>` est l'indice du premier octet après cet emplacement.
      // - `snippet` : Un extrait de code d'une seule ligne provenant de l'emplacement␣
↪indiqué par `@src`.
      // L'extrait est cité et suit l'annotation `@src` correspondante.
      // - `*` : Valeur joker qui peut être utilisée pour tout demander.
      "debugInfo": ["location", "snippet"]
    },
    // Paramètres des métadonnées (facultatif)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "metadata": {
      // Utiliser uniquement le contenu littéral et non les URL (faux par défaut)
      "useLiteralContent": true,
      // Utilisez la méthode de hachage donnée pour le hachage des métadonnées qui est␣
↪ajouté au bytecode.
      // Le hachage des métadonnées peut être supprimé du bytecode via l'option "none".
      // Les autres options sont "ipfs" et "bzzr1".
      // Si l'option est omise, "ipfs" est utilisé par défaut.
      "bytecodeHash": "ipfs"
    },
    // Adresses des bibliothèques. Si toutes les bibliothèques ne sont pas données ici,
    // il peut en résulter des objets non liés dont les données de sortie sont␣
↪différentes.
    "libraries": {
      // La clé de premier niveau est le nom du fichier source dans lequel la␣
↪bibliothèque est utilisée.
      // Si des remappages sont utilisés, ce fichier source doit correspondre au chemin␣
↪global
      // après que les remappages aient été appliqués.
      // Si cette clé est une chaîne vide, cela fait référence à un niveau global.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    },
    // Ce qui suit peut être utilisé pour sélectionner les sorties souhaitées en se␣
↪basant
    // sur les noms de fichiers et de contrats.
    // Si ce champ est omis, alors le compilateur charge et effectue une vérification de␣
↪type,
    // mais ne générera aucune sortie en dehors des erreurs.
    // La clé de premier niveau est le nom du fichier et la clé de second niveau est le␣
↪nom du contrat.
    // Un nom de contrat vide est utilisé pour les sorties qui ne sont pas liées à un␣
↪contrat
    // mais à l'ensemble du fichier source, comme l'AST.
    // Une étoile comme nom de contrat fait référence à tous les contrats du fichier.
    // De même, une étoile comme nom de fichier correspond à tous les fichiers.
    // Pour sélectionner toutes les sorties que le compilateur peut éventuellement␣
↪générer, utilisez
    // "outputSelection : { "*" : { "*" : [ "*" ], "" : [ "*" ] } }"
    // mais notez que cela pourrait ralentir inutilement le processus de compilation.
    //
    // Les types de sortie disponibles sont les suivants :
    //
    // Niveau fichier (nécessite une chaîne vide comme nom de contrat) :
    // ast - AST de tous les fichiers sources
    //
    // Niveau du contrat (nécessite le nom du contrat ou "*") :
    // abi - ABI
    // devdoc - Documentation du développeur (natspec)
    // userdoc - Documentation utilisateur (natspec)
    // metadata - Métadonnées
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // ir - Représentation intermédiaire Yul du code avant optimisation
    // irOptimized - Représentation intermédiaire après optimisation
    // storageLayout - Emplacements, décalages et types des variables d'état du contrat.
    // evm.assembly - Nouveau format d'assemblage
    // evm.legacyAssembly - Ancien format d'assemblage en JSON
    // evm.bytecode.functionDebugData - Informations de débogage au niveau des fonctions.
    // evm.bytecode.object - Objet bytecode
    // evm.bytecode.opcodes - Liste d'opcodes
    // evm.bytecode.sourceMap - Cartographie de la source (utile pour le débogage)
    // evm.bytecode.linkReferences - Références de liens (si objet non lié)
    // evm.bytecode.generatedSources - Sources générées par le compilateur.
    // evm.deployedBytecode* - Bytecode déployé (a toutes les options que evm.bytecode a)
    // evm.deployedBytecode.immutableReferences - Correspondance entre les identifiants␣
↪AST et les plages de bytecode qui font référence aux immutables.
    // evm.methodIdentifiers - La liste des hachages de fonctions
    // evm.gasEstimates - Estimations des gaz de fonction
    // ewasm.wast - Ewasm au format S-expressions de WebAssembly
    // ewasm.wasm - Ewasm au format binaire WebAssembly
    //
    // Notez que l'utilisation d'un `evm`, `evm.bytecode`, `ewasm`, etc. sélectionnera␣
↪chaque
    // partie cible de cette sortie. De plus, `*` peut être utilisé comme un joker pour␣
↪tout demander.
    //
    "outputSelection": {
      "*": {
        "*": [
          "metadata", "evm.bytecode" // Activez les sorties de métadonnées et de␣
↪bytecode de chaque contrat.
          , "evm.bytecode.sourceMap" // Activez la sortie de la carte des sources pour␣
↪chaque contrat.
        ],
        "": [
          "ast" // Active la sortie AST de chaque fichier.
        ]
      },
      // Active la sortie de l'abi et des opcodes de MonContrat définis dans le fichier␣
↪def.
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      }
    },
    // L'objet modelChecker est expérimental et sujet à des modifications.
    "modelChecker":
    {
      // Choisir les contrats qui doivent être analysés comme ceux qui sont déployés.
      "contracts":
      {
        "source1.sol": ["contract1"],
        "source2.sol": ["contract2", "contract3"]
      },
      // Choisir si les opérations de division et de modulo doivent être remplacées par
```

(suite sur la page suivante)

(suite de la page précédente)

```
      // multiplication avec des variables de type slack. La valeur par défaut est `true`.
      // L'utilisation de `false` ici est recommandée si vous utilisez le moteur CHC
      // et que vous n'utilisez pas Spacer comme solveur de Horn (en utilisant Eldarica,␣
↪par exemple).
      // Voir la section Vérification formelle pour une explication plus détaillée de␣
↪cette option.
      "divModWithSlacks": true,
      // Choisissez le moteur de vérification de modèle à utiliser : all (par défaut),␣
↪bmc, chc, none.
      "engine": "chc",
      // Choisissez quels types d'invariants doivent être signalés à l'utilisateur :␣
↪contrat, réentrance.
      "invariants": ["contract", "reentrancy"],
      // Choisissez si vous souhaitez afficher toutes les cibles non prouvées. La valeur␣
↪par défaut est `false`.
      "showUnproved": true,
      // Choisissez les solveurs à utiliser, s'ils sont disponibles.
      // Voir la section Vérification formelle pour la description des solveurs.
      "solvers": ["cvc4", "smtlib2", "z3"],
      // Choisissez les cibles à vérifier : constantCondition,
      // underflow, overflow, divByZero, balance, assert, popEmptyArray, outOfBounds.
      // Si l'option n'est pas donnée, toutes les cibles sont vérifiées par défaut,
      // sauf underflow/overflow pour Solidity >=0.8.7.
      // Voir la section Vérification formelle pour la description des cibles.
      "targets": ["underflow", "overflow", "assert"],
      // Délai d'attente pour chaque requête SMT en millisecondes.
      // Si cette option n'est pas donnée, le SMTChecker utilisera une limite de␣
↪ressources déterministe
      // par défaut.
      // Un délai d'attente de 0 signifie qu'il n'y a aucune restriction de ressources ou␣
↪de temps pour les requêtes.
      "timeout": 20000
    }
  }
}
```

Description de la sortie

```
{
  // Facultatif : non présent si aucune erreur/avis/infos n'a été rencontrée.
  "errors": [
    {
      // Facultatif : Emplacement dans le fichier source.
      "sourceLocation": {
        "file": "sourceFile.sol",
        "start": 0,
        "end": 100
      },
      // Facultatif : Autres lieux (par exemple, lieux de déclarations conflictuelles)
      "secondarySourceLocations": [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        {
          "file": "sourceFile.sol",
          "start": 64,
          "end": 92,
          "message": "L'autre déclaration est ici :"
        }
      ],
      // Obligatoire : Type d'erreur, tel que "TypeError", "InternalCompilerError",
↪"Exception", etc.
      // Voir ci-dessous pour la liste complète des types.
      "type": "TypeError",
      // Obligatoire : Composant d'où provient l'erreur, tel que "general", "ewasm", etc.
      "component": "general",
      // Obligatoire ("error", "warning" ou "info", mais veuillez noter que cela␣
↪pourrait être étendu à l'avenir)
      "severity": "error",
      // Facultatif : code unique pour la cause de l'erreur.
      "errorCode": "3141",
      // Obligatoire
      "message": "Mot clé invalide",
      // Facultatif : le message formaté avec l'emplacement de la source
      "formattedMessage": "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // Il contient les sorties au niveau du fichier.
  // Il peut être limité/filtré par les paramètres outputSelection.
  "sources": {
    "sourceFile.sol": {
      // Identifiant de la source (utilisé dans les cartes de sources)
      "id": 1,
      // L'objet AST
      "ast": {}
    }
  },
  // Il contient les sorties au niveau du contrat.
  // Il peut être limité/filtré par les paramètres outputSelection.
  "contracts": {
    "sourceFile.sol": {
      // Si la langue utilisée ne comporte pas de noms de contrat, ce champ doit être␣
↪égal à une chaîne vide.
      "ContractName": {
        // L'ABI du contrat Ethereum. S'il est vide, il est représenté comme un tableau␣
↪vide.
        // See https://docs.soliditylang.org/en/develop/abi-spec.html
        "abi": [],
        // Voir la documentation sur la sortie des métadonnées (chaîne JSON sérialisée).
        "metadata": "{/* ... */}",
        // Documentation utilisateur (natspec)
        "userdoc": {},
        // Documentation pour les développeurs (natspec)
        "devdoc": {},
        // Représentation intermédiaire (chaîne de caractères)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "ir": "",
        // Voir la documentation sur l'agencement du stockage.
        "storageLayout": {"storage": [/* ... */], "types": {/* ... */} },
        // Sorties liées à l'EVM
        "evm": {
          // Assemblée (chaîne de caractères)
          "assembly": "",
          // Assemblage à l'ancienne (objet)
          "legacyAssembly": {},
          // Bytecode et détails connexes.
          "bytecode": {
            // Débogage des données au niveau des fonctions.
            "functionDebugData": {
              // Suit maintenant un ensemble de fonctions incluant des fonctions␣
↪définies par l'utilisateur.
              // L'ensemble ne doit pas nécessairement être complet.
              "@mint_13": { // Nom interne de la fonction
                "entryPoint": 128, // Décalage d'octet dans le bytecode où la fonction␣
↪commence (facultatif)
                "id": 13, // AST ID de la définition de la fonction ou null pour les␣
↪fonctions internes au compilateur (facultatif)
                "parameterSlots": 2, // Nombre d'emplacements de pile EVM pour les␣
↪paramètres de fonction (facultatif)
                "returnSlots": 1 // Nombre d'emplacements de pile EVM pour les valeurs de␣
↪retour (facultatif)
              }
            },
            // Le bytecode sous forme de chaîne hexagonale.
            "object": "00fe",
            // Liste des opcodes (chaîne de caractères)
            "opcodes": "",
            // Le mappage de la source sous forme de chaîne. Voir la définition du␣
↪mappage de la source.
            "sourceMap": "",
            // Tableau des sources générées par le compilateur. Actuellement, il ne
            // contient qu'un seul fichier Yul.
            "generatedSources": [{
              // Yul AST
              "ast": {/* ... */},
              // Fichier source sous sa forme texte (peut contenir des commentaires)
              "contents":"{ function abi_decode(start, end) -> data { data :=␣
↪calldataload(start) } }",
              // ID du fichier source, utilisé pour les références aux sources, même
↪"namespace" que les fichiers sources de Solidity.
              "id": 2,
              "language": "Yul",
              "name": "#utility.yul"
            }],
            // S'il est donné, il s'agit d'un objet non lié.
            "linkReferences": {
              "libraryFile.sol": {
                // Décalage des octets dans le bytecode.
```

(suite sur la page suivante)

(suite de la page précédente)

```
                // La liaison remplace les 20 octets qui s'y trouvent.
                "Library1": [
                  { "start": 0, "length": 20 },
                  { "start": 200, "length": 20 }
                ]
              }
            }
          },
          "deployedBytecode": {
            /* ..., */ // La même disposition que ci-dessus.
            "immutableReferences": {
              // Il existe deux références à l'immuable avec l'ID AST 3, toutes deux d'une␣
↪longueur de 32 octets. L'une se trouve
              // à l'offset 42 du bytecode, l'autre à l'offset 80 du bytecode.
              "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
            }
          },
          // La liste des hachages de fonctions
          "methodIdentifiers": {
            "delegate(address)": "5c19a95c"
          },
          // Estimation des gaz de fonction
          "gasEstimates": {
            "creation": {
              "codeDepositCost": "420000",
              "executionCost": "infinite",
              "totalCost": "infinite"
            },
            "external": {
              "delegate(address)": "25000"
            },
            "internal": {
              "heavyLifting()": "infinite"
            }
          }
        },
        // Sorties liées à l'Ewasm
        "ewasm": {
          // Format des expressions S
          "wast": "",
          // Format binaire (chaîne hexagonale)
          "wasm": ""
        }
      }
    }
  }
}
```

Types d'erreurs

1. `JSONError` : L'entrée JSON n'est pas conforme au format requis, par exemple, l'entrée n'est pas un objet JSON, la langue n'est pas supportée, etc.
2. `IOError` : Erreurs de traitement des entrées/sorties et des importations, telles qu'une URL non résoluble ou une erreur de hachage dans les sources fournies.
3. `ParserError` : Le code source n'est pas conforme aux règles du langage.
4. `DocstringParsingError` : Les balises NatSpec du bloc de commentaires ne peuvent pas être analysées.
5. `SyntaxError` : Erreur de syntaxe, comme l'utilisation de « continue » en dehors d'une boucle « for ».
6. `DeclarationError` : Noms d'identifiants invalides, impossibles à résoudre ou contradictoires. Par exemple, « Identifiant non trouvé ».
7. `TypeError` : Erreur dans le système de types, comme des conversions de types invalides, des affectations invalides, etc.
8. `UnimplementedFeatureError` : La fonctionnalité n'est pas supportée par le compilateur, mais devrait l'être dans les futures versions.
9. `InternalCompilerError` : Bogue interne déclenché dans le compilateur - il doit être signalé comme un problème.
10. `Exception` : Echec inconnu lors de la compilation - ceci devrait être signalé comme un problème.
11. `CompilerError` : Utilisation non valide de la pile du compilateur - ceci devrait être signalé comme un problème.
12. `FatalError` : Une erreur fatale n'a pas été traitée correctement - ceci devrait être signalé comme un problème.
13. `Warning` : Un avertissement, qui n'a pas arrêté la compilation, mais qui devrait être traité si possible.
14. `Info` : Une information que le compilateur pense que l'utilisateur pourrait trouver utile, mais qui n'est pas dangereuse et ne doit pas nécessairement être traitée.

3.12.4 Outils de compilation

solidity-upgrade

`solidity-upgrade` peut vous aider à mettre à jour semi-automatiquement vos contrats en fonction des changements de langue. Bien qu'il n'implémente pas et ne puisse pas implémenter tous changements requis pour chaque version de rupture, il prend en charge ceux qui, autrement, nécessiteraient de nombreux ajustements manuels répétitifs.

Note : `solidity-upgrade` effectue une grande partie du travail, mais vos contrats nécessiteront très probablement d'autres ajustements manuels. Nous vous recommandons d'utiliser un système de contrôle de version pour vos fichiers. Cela permet de réviser et éventuellement de revenir en arrière sur les modifications apportées.

Avertissement : `solidity-upgrade` n'est pas considéré comme complet ou exempt de bogues, donc veuillez l'utiliser avec précaution.

Comment cela fonctionne

Vous pouvez passer un ou plusieurs fichiers sources Solidity à `solidity-upgrade [files]`. Si ceux-ci utilisent l'instruction `import` qui fait référence à des fichiers en dehors du répertoire du fichier source actuel, vous devez spécifier des répertoires qui sont autorisés à lire et à importer des fichiers, en passant l'instruction `--allow-paths [directory]`. Vous pouvez ignorer les fichiers manquants en passant `--ignore-missing`.

`solidity-upgrade` est basé sur `libsolidity` et peut analyser vos fichiers sources, et peut y trouver des mises à jour applicables.

Les mises à jour de source sont considérées comme de petits changements textuels à votre code source. Elles sont appliquées à une représentation en mémoire des fichiers sources donnés. Le fichier source correspondant est mis à jour par défaut, mais vous pouvez passer la commande `--dry-run` pour simuler l'ensemble du processus de mise à jour sans écrire dans aucun fichier.

Le processus de mise à jour lui-même a deux phases. Dans la première phase, les fichiers sources sont analysés, et puisqu'il n'est pas possible de mettre à jour le code source à ce niveau, les erreurs sont collectées et peuvent être enregistrées en passant `--verbose`. Aucune mise à jour de la source n'est disponible à ce stade.

Dans la deuxième phase, toutes les sources sont compilées et tous les modules d'analyse de mise à niveau activés sont exécutés en même temps que la compilation. Par défaut, tous les modules disponibles sont activés. Veuillez lire la documentation sur les *modules disponibles* pour plus de détails.

Cela peut entraîner des erreurs de compilation qui peuvent être corrigées par des mises à jour des sources. Si aucune erreur ne se produit, aucune mise à niveau des sources n'est signalée et vous avez terminé. Si des erreurs se produisent et qu'un module de mise à niveau a signalé une mise à niveau de la source, la première source, la première signalée est appliquée et la compilation est déclenchée à nouveau pour tous les fichiers sources donnés. L'étape précédente est répétée aussi longtemps que des mises à jour de sources sont signalées. Si des erreurs surviennent encore, vous pouvez les enregistrer en passant le paramètre `--verbose`. Si aucune erreur ne se produit, vos contrats sont à jour et peuvent être compilés avec la dernière version du compilateur.

Modules de mise à niveau disponibles

Module	Version	Description
`constructor`	0.5.0	Les constructeurs doivent maintenant être définis à l'aide dumot-clé « constructeur ».
`visibility`	0.5.0	La visibilité explicite des fonctions est désormais obligatoire, La valeur par défaut est `public`.
`abstract`	0.6.0	Le mot-clé `abstract` doit être utilisé si le contrat ne met pas en œuvre toutes ses fonctions.
`virtual`	0.6.0	Fonctions sans implémentation en dehors d'un doivent être marquées `virtual`.
`override`	0.6.0	Lorsque vous remplacez une fonction ou un modificateur, la nouvelle fonction le mot clé `override` doit être utilisé.
`dotsyntax`	0.7.0	La syntaxe suivante est obsolète : `f.gas(...)()`, `f.value(...)()` et `(new C).value(...)()`. Remplacez ces appels par `f{gas: ..., value: ...}()` et `(new C){value: ...}()`.
`now`	0.7.0	Le mot clé `now` est obsolète. Utilisez `block.timestamp` à la place.
`constructor-visibility`	0.7.0	Supprime la visibilité des constructeurs.

Veuillez lire *0.5.0 notes de mise à jour*, *0.6.0 notes de mise à jour*, *0.7.0 notes de mise à jour* et *0.8.0 notes de mise à jour* pour plus de détails.

Synopsis

```
Usage: solidity-upgrade [options] contract.sol

Allowed options:
    --help                Show help message and exit.
    --version             Show version and exit.
    --allow-paths path(s)
                          Allow a given path for imports. A list of paths can be
                          supplied by separating them with a comma.
    --ignore-missing      Ignore missing files.
    --modules module(s)   Only activate a specific upgrade module. A list of
                          modules can be supplied by separating them with a comma.
    --dry-run             Apply changes in-memory only and don't write to input
                          file.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    --verbose           Print logs, errors and changes. Shortens output of
                        upgrade patches.
    --unsafe            Accept *unsafe* changes.
```

Rapports de bogue / Demandes de fonctionnalités

Si vous avez trouvé un bogue ou si vous avez une demande de fonctionnalité, veuillez déposer une question sur Github.

Exemple

Supposons que vous ayez le contrat suivant dans `Source.sol` :

```
pragma solidity >=0.6.0 <0.6.4;
// Ceci ne compilera pas après la version 0.7.0.
// SPDX-License-Identifier: GPL-3.0
contract C {
    // FIXME : supprimer la visibilité du constructeur et rendre le contrat␣
↪abstrait.
    constructor() internal {}
}

contract D {
    uint time;

    function f() public payable {
        // FIXME : remplacer maintenant par block.timestamp
        time = now;
    }
}

contract E {
    D d;

    // FIXME : supprimer la visibilité du constructeur
    constructor() public {}

    function g() public {
        // FIXME : change .value(5) => {value : 5}
        d.f.value(5)();
    }
}
```

Changements requis

Le contrat ci-dessus ne sera pas compilé à partir de la version 0.7.0. Pour mettre le contrat à jour avec la version actuelle de Solidity, les modules de mise à jour suivants doivent être exécutés : `constructor-visibility`, `now` et `dotsyntax`. Veuillez lire la documentation sur *modules disponibles* pour plus de détails.

Exécution de la mise à niveau

Il est recommandé de spécifier explicitement les modules de mise à niveau en utilisant l'argument `--modules`.

```
solidity-upgrade --modules constructor-visibility,now,dotsyntax Source.sol
```

The command above applies all changes as shown below. Please review them carefully (the pragmas will have to be updated manually.)

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
abstract contract C {
    // FIXME : supprimer la visibilité du constructeur et rendre le contrat␣
↪abstrait.
    constructor() {}
}

contract D {
    uint time;

    function f() public payable {
        // FIXME : remplacer maintenant par block.timestamp
        time = block.timestamp;
    }
}

contract E {
    D d;

    // FIXME : supprimer la visibilité du constructeur
    constructor() {}

    function g() public {
        // FIXME : change .value(5) => {value : 5}
        d.f{value: 5}();
    }
}
```

3.13 Analyse de la sortie du compilateur

Il est souvent utile d'examiner le code d'assemblage généré par le compilateur. Le binaire généré, c'est-à-dire la sortie de `solc --bin contract.sol`, est généralement difficile à lire. Il est recommandé d'utiliser l'indicateur `--asm` pour analyser la sortie de l'assemblage. Même pour les gros contrats, regarder une visuel de l'assemblage avant et après un changement est souvent très instructif.

Considérons le contrat suivant (nommé, disons `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;
    }
}
```

Voici le résultat de l'opération « 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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
```

(suite sur la page suivante)

(suite de la page précédente)

```
      pop
      jump  // out

    auxdata:␣
↪0xa2646970667358221220a5874f19737ddd4c5d77ace1619e5160c67b3d4bedac75fce908fed32d98899864736f6c6378273
}
```

Alternativement, la sortie ci-dessus peut également être obtenue à partir de Remix, sous l'option « Compilation Details » après avoir compilé un contrat.

Remarquez que la sortie `asm` commence par le code de création / constructeur. Le code de déploiement est fourni comme partie du sous-objet (dans l'exemple ci-dessus, il fait partie du sous-objet `sub_0`). Le champ `auxdata`'' correspond au contrat :ref:`metadata <encodage des métadonnées dans le bytecode>`. Les commentaires dans la sortie de l'assemblage pointent vers la emplacement de la source. Notez que ``#utility.yul` est un fichier généré en interne de fonctions utilitaires qui peut être obtenu en utilisant les drapeaux `--combined-json generated-sources,generated-sources-runtime`.

De même, l'assemblage optimisé peut être obtenu avec la commande : `solc --optimize --asm contract.sol`. Souvent, il est intéressant de voir si deux sources différentes dans Solidity aboutissent au même code optimisé. le même code optimisé. Par exemple, pour voir si les expressions `(a * b) / c`, `a * b / c` génèrent le même bytecode. Cela peut être facilement fait en prenant un `diff` de la sortie assembleur correspondante, après avoir éventuellement supprimé les commentaires. d'assemblage correspondant, après avoir éventuellement supprimé les commentaires qui font référence aux emplacements des sources.

Note : La sortie `--asm` n'est pas conçue pour être lisible par une machine. Par conséquent, il peut y avoir des des changements de rupture sur la sortie entre les versions mineures de solc.

3.14 Changements apportés au Codegen basé sur Solidity IR

Solidity peut générer du bytecode EVM de deux manières différentes : Soit directement de Solidity vers les opcodes EVM (« old codegen »), soit par le biais d'une représentation intermédiaire (« IR ») dans Yul (« new codegen » ou « IR-based codegen »).

Le générateur de code basé sur l'IR a été introduit dans le but non seulement de permettre génération de code plus transparente et plus vérifiable, mais aussi de permettre des passes d'optimisation plus puissantes qui couvrent plusieurs fonctions.

Actuellement, le générateur de code basé sur IR est toujours marqué comme expérimental, mais il supporte toutes les fonctionnalités du langage et a fait l'objet de nombreux tests. Nous considérons donc qu'il est presque prêt à être utilisé en production.

Vous pouvez l'activer sur la ligne de commande en utilisant `--experimental-via-ir`. ou avec l'option `{"viaIR" : true}` dans le standard-json et nous encourageons tout le monde à l'essayer !

Pour plusieurs raisons, il existe de minuscules différences sémantiques entre l'ancien générateur de code basé sur l'IR, principalement dans des domaines où nous ne nous attendons pas à ce que les gens se fient à ce comportement de toute façon. Cette section met en évidence les principales différences entre l'ancien et le générateur de code basé sur la RI.

3.14.1 Changements uniquement sémantiques

Cette section énumère les changements qui sont uniquement sémantiques, donc potentiellement cacher un comportement nouveau et différent dans le code existant.

— Lorsque les structures de stockage sont supprimées, chaque emplacement de stockage qui contient un membre de la structure est entièrement mis à zéro. Auparavant, l'espace de remplissage n'était pas modifié. Par conséquent, si l'espace de remplissage dans une structure est utilisé pour stocker des données (par exemple, dans le contexte d'une mise à jour de contrat), vous devez être conscient que que `delete` effacera maintenant aussi le membre ajouté (alors qu'il n'aurait pas été effacé dans le passé).

```
// 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 occupe seulement les 16 premiers octets de l'emplacement de 32 octets.
        // delete écrira zéro dans l'emplacement complet
    }
}
```

Nous avons le même comportement pour la suppression implicite, par exemple lorsque le tableau de structs est raccourci.

— Les modificateurs de fonction sont mis en œuvre d'une manière légèrement différente en ce qui concerne les paramètres de fonction et les variables de retour. Cela a notamment un effet si le caractère générique `_;` est évalué plusieurs fois dans un modificateur. Dans l'ancien générateur de code, chaque paramètre de fonction et variable de retour a un emplacement fixe sur la pile. Si la fonction est exécutée plusieurs fois parce que `_;` est utilisé plusieurs fois ou utilisé dans une boucle, alors un changement de la valeur du paramètre de fonction ou de la variable de retour est visible lors de la prochaine exécution de la fonction. Le nouveau générateur de code implémente les modificateurs à l'aide de fonctions réelles et transmet les paramètres de fonction. Cela signifie que plusieurs évaluations du corps d'une fonction obtiendront les mêmes valeurs pour les paramètres, et l'effet sur les variables de retour est qu'elles sont réinitialisées à leur valeur par défaut (zéro) à chaque exécution.

```
// 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() { _; _; }
}
```

Si vous exécutez `f(0)` dans l'ancien générateur de code, il retournera 2, alors qu'il retournera 1 en utilisant le nouveau générateur de code.

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

contract C {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    bool active = true;
    modifier mod()
    {
        _;
        active = false;
        _;
    }
    function foo() external mod() returns (uint ret)
    {
        if (active)
            ret = 1; // Same as ``return 1``
    }
}
```

La fonction `C.foo()` renvoie les valeurs suivantes :

— Ancien générateur de code : `1` comme variable de retour est initialisé à `0` une seule fois avant la première évaluation `_;` et ensuite écrasée par la variable `return 1;`. Elle n'est pas initialisée à nouveau pour la seconde évaluation et `foo()` ne l'assigne pas explicitement non plus (à cause de `active == false`), il garde donc sa première valeur.

— Nouveau générateur de code : `0` car tous les paramètres, y compris les paramètres de retour, seront réinitialisés avant chaque évaluation `_;`.

— L'ordre d'initialisation des contrats a changé en cas d'héritage.

L'ordre était auparavant le suivant :

— Toutes les variables d'état sont initialisées à zéro au début.
— Évaluer les arguments du constructeur de base du contrat le plus dérivé au contrat le plus basique.
— Initialiser toutes les variables d'état dans toute la hiérarchie d'héritage, de la plus basique à la plus dérivée.
— Exécuter le constructeur, s'il est présent, pour tous les contrats dans la hiérarchie linéarisée du plus bas au plus dérivé.

Nouvel ordre :

— Toutes les variables d'état sont initialisées à zéro au début.
— Évaluer les arguments du constructeur de base du contrat le plus dérivé au contrat le plus basique.
— Pour chaque contrat dans l'ordre du plus basique au plus dérivé dans la hiérarchie linéarisée, exécuter :

1. Si elles sont présentes à la déclaration, les valeurs initiales sont assignées aux variables d'état.
2. Le constructeur, s'il est présent.

Cela entraîne des différences dans certains contrats, par exemple :

```
// 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();
}
```

Auparavant, `y` était fixé à 0. Cela est dû au fait que nous initialisions d'abord les variables d'état : D'abord,

`x` est mis à 0, et lors de l'initialisation de `y`, `f()` renvoie 0, ce qui fait que `y` est également 0. Avec les nouvelles règles, `y`' sera fixé à 42. Nous commençons par initialiser ``x à 0, puis nous appelons le constructeur de A qui fixe `x` à 42. Enfin, lors de l'initialisation de `y`, `f()` renvoie 42, ce qui fait que `y` est 42.

— La copie de tableaux d'"« octets » de la mémoire vers le stockage est implémentée d'une manière différente. L'ancien générateur de code copie toujours des mots entiers, alors que le nouveau coupe le tableau d'octets après sa fin. L'ancien comportement peut conduire à ce que des données sales soient copiées après la fin du tableau (mais toujours dans le même emplacement de stockage). Cela entraîne des différences dans certains contrats, par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

contract C {
    bytes x;
    function f() public returns (uint _r) {
        bytes memory m = "tmp";
        assembly {
            mstore(m, 8)
            mstore(add(m, 32), "deadbeef15dead")
        }
        x = m;
        assembly {
            _r := sload(x.slot)
        }
    }
}

Auparavant, ``f()`` retournait␣
↪``0x6465616462656566313564656164000000000000000000000000000000000010``
```

(il a une longueur correcte, et les 8 premiers éléments sont corrects, mais ensuite il contient des données sales qui ont été définies via l'assemblage). Maintenant, il renvoie `0x6465616462656566000000000000000000000000000000000000000000000010` (il a une longueur correcte, et des éléments corrects, mais il ne contient pas de données superflues).

— Pour l'ancien générateur de code, l'ordre d'évaluation des expressions n'est pas spécifié. Pour le nouveau générateur de code, nous essayons d'évaluer dans l'ordre de la source (de gauche à droite), mais nous ne le garantissons pas. Cela peut conduire à des différences sémantiques.
Par exemple :

```
// 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;
    }
}
```

La fonction `preincr_u8(1)` retourne les valeurs suivantes :
— Ancien générateur de code : 3 (`1 + 2`) mais la valeur de retour n'est pas spécifiée en général.
— Nouveau générateur de code : 4 (`2 + 2`) mais la valeur de retour n'est pas garantie
D'autre part, les expressions des arguments de fonction sont évaluées dans le même ordre par les deux générateurs de code, à l'exception des fonctions globales `addmod` et `mulmod`. Par exemple :

```
// 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);
    }
}
```

La fonction `g(1, 2)` renvoie les valeurs suivantes :
— Ancien générateur de code : `10` (`add(2 + 3, 2 + 3)`) mais la valeur de retour n'est pas spécifiée en général.
— Nouveau générateur de code : `10` mais la valeur de retour n'est pas garantie
Les arguments des fonctions globales `addmod` et `mulmod` sont évalués de droite à gauche par l'ancien générateur de code et de gauche à droite par le nouveau générateur de code. Par exemple :

```
// 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);
    }
}
```

La fonction `f()` renvoie les valeurs suivantes :
— Ancien générateur de code : » aMod = 0 » et » mMod = 2 « .
— Nouveau générateur de code : » aMod = 4 » et » mMod = 0 « .
— Le nouveau générateur de code impose une limite dure de `type(uint64).max` (`0xffffffffffffffff`) pour le pointeur de mémoire libre. Les allocations qui augmenteraient sa valeur au-delà de cette limite. L'ancien générateur de code n'a pas n'a pas cette limite.
Par exemple :

```
// 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);
    }
}
```

La fonction *f()* se comporte comme suit :
— Ancien générateur de code : manque de gaz lors de la mise à zéro du contenu du tableau après la grande allocation de mémoire.

— Nouveau générateur de code : retour en arrière en raison d'un débordement du pointeur de mémoire libre (ne tombe pas en panne sèche).

3.14.2 Internes

Pointeurs de fonctions internes

L'ancien générateur de code utilise des décalages de code ou des balises pour les valeurs des pointeurs de fonctions internes. Ceci est particulièrement compliqué car ces offsets sont différents au moment de la construction et après le déploiement et les valeurs peuvent traverser cette frontière via le stockage. Pour cette raison, les deux offsets sont codés au moment de la construction dans la même valeur (dans différents octets).

Dans le nouveau générateur de code, les pointeurs de fonction utilisent des ID internes qui sont alloués en séquence. Comme les appels via des pointeurs de fonction doivent toujours utiliser une fonction de distribution interne qui utilise l'instruction `switch` pour sélectionner la bonne fonction.

L'ID `0` est réservé aux pointeurs de fonction non initialisés qui provoquent une panique dans la fonction de répartition lorsqu'ils sont appelés.

Dans l'ancien générateur de code, les pointeurs de fonctions internes sont initialisés avec une fonction spéciale qui provoque toujours une panique. Cela provoque une écriture en mémoire au moment de la construction pour les pointeurs de fonctions internes en mémoire.

Nettoyage

L'ancien générateur de code n'effectue le nettoyage qu'avant une opération dont le résultat pourrait être affecté par les valeurs des bits sales. Le nouveau générateur de code effectue le nettoyage après toute opération qui peut entraîner des bits sales. L'espoir est que l'optimiseur sera suffisamment puissant pour éliminer les opérations de nettoyage redondantes.

Par exemple :

```
// 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
        }
        _r2 = _a;
    }
}
```

La fonction `f(1)` renvoie les valeurs suivantes :

— Ancien générateur de code : (`fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe`, `0000000000000000000000000000000000000000000000000000000000000000fe`)

— Nouveau générateur de codes : (`0000000000000000000000000000000000000000000000000000000000000000fe`, `0000000000000000000000000000000000000000000000000000000000000000fe`)

Notez que, contrairement au nouveau générateur de code, l'ancien générateur de code n'effectue pas de nettoyage après l'affectation bit-non (`_a = ~_a`). Il en résulte que des valeurs différentes sont assignées (dans le bloc d'assemblage en ligne) à la valeur de retour `_r1` entre l'ancien et le nouveau générateur de code. Cependant, les deux générateurs de code effectuent un nettoyage avant que la nouvelle valeur de `_a` soit assignée à `_r2`.

3.15 Layout of State Variables in Storage

State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings (see below), data is stored contiguously item after item starting with the first state variable, which is stored in slot `0`. For each variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules :

- The first item in a storage slot is stored lower-order aligned.
- Value types use only as many bytes as are necessary to store them.
- If a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot.
- Structs and array data always start a new slot and their items are packed tightly according to these rules.
- Items following struct or array data always start a new storage slot.

For contracts that use inheritance, the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the above rules, state variables from different contracts do share the same storage slot.

The elements of structs and arrays are stored after each other, just as if they were given as individual values.

Avertissement : When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

It might be beneficial to use reduced-size types if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. If you are not reading or writing all the values in a slot at the same time, this can have the opposite effect, though : When one value is written to a multi-value storage slot, the storage slot has to be read first and then combined with the new value such that other data in the same slot is not destroyed.

When dealing with function arguments or memory values, there is no inherent benefit because the compiler does not pack these values.

Finally, in order to allow the EVM to optimize for this, ensure that you try to order your storage variables and `struct` members such that they can be packed tightly. For example, declaring your storage variables in the order of `uint128, uint128, uint256` instead of `uint128, uint256, uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

Note : The layout of state variables in storage is considered to be part of the external interface of Solidity due to the fact that storage pointers can be passed to libraries. This means that any change to the rules outlined in this section is considered a breaking change of the language and due to its critical nature should be considered very carefully before being executed.

3.15.1 Mappings and Dynamic Arrays

Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored « in between » the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the *rules above* and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

Assume the storage location of the mapping or array ends up being a slot `p` after applying *the storage layout rules*. For dynamic arrays, this slot stores the number of elements in the array (byte arrays and strings are an exception, see *below*). For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations.

Array data is located starting at `keccak256(p)` and it is laid out in the same way as statically-sized array data would : One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes. Dynamic arrays of dynamic arrays apply this rule recursively. The location of element `x[i][j]`, where the type of `x` is `uint24[][]`, is computed as follows (again, assuming `x` itself is stored at slot `p`) : The slot is `keccak256(keccak256(p) + i) + floor(j / floor(256 / 24))` and the element can be obtained from the slot data `v` using `(v >> ((j % floor(256 / 24)) * 24)) & type(uint24).max`.

The value corresponding to a mapping key `k` is located at `keccak256(h(k) . p)` where `.` is concatenation and `h` is a function that is applied to the key depending on its type :

— for value types, `h` pads the value to 32 bytes in the same way as when storing the value in memory.
— for strings and byte arrays, `h` computes the `keccak256` hash of the unpadded data.

If the mapping value is a non-value type, the computed slot marks the start of the data. If the value is of struct type, for example, you have to add an offset corresponding to the struct member to reach the member.

As an example, consider the following contract :

```
// 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;
}
```

Let us compute the storage location of `data[4][9].c`. The position of the mapping itself is `1` (the variable `x` with 32 bytes precedes it). This means `data[4]` is stored at `keccak256(uint256(4) . uint256(1))`. The type of `data[4]` is again a mapping and the data for `data[4][9]` starts at slot `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1)))`. The slot offset of the member `c` inside the struct `S` is `1` because `a` and `b` are packed in a single slot. This means the slot for `data[4][9].c` is `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`. The type of the value is `uint256`, so it uses a single slot.

`bytes` and `string`

`bytes` and `string` are encoded identically. In general, the encoding is similar to `bytes1[]`, in the sense that there is a slot for the array itself and a data area that is computed using a `keccak256` hash of that slot's position. However, for short values (shorter than 32 bytes) the array elements are stored together with the length in the same slot.

In particular : if the data is at most `31` bytes long, the elements are stored in the higher-order bytes (left aligned) and the lowest-order byte stores the value `length * 2`. For byte arrays that store data which is `32` or more bytes long, the main slot `p` stores `length * 2 + 1` and the data is stored as usual in `keccak256(p)`. This means that you can distinguish a short array from a long array by checking if the lowest bit is set : short (not set) and long (set).

Note : Handling invalidly encoded slots is currently not supported but may be added in the future. If you are compiling via the experimental IR-based compiler pipeline, reading an invalidly encoded slot results in a `Panic(0x22)` error.

3.15.2 JSON Output

The storage layout of a contract can be requested via the *standard JSON interface*. The output is a JSON object containing two keys, `storage` and `types`. The `storage` object is an array where each element has the following form :

```
{
    "astId": 2,
    "contract": "fileA:A",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

The example above is the storage layout of `contract A { uint x; }` from source unit `fileA` and

- `astId` is the id of the AST node of the state variable's declaration
- `contract` is the name of the contract including its path as prefix
- `label` is the name of the state variable
- `offset` is the offset in bytes within the storage slot according to the encoding
- `slot` is the storage slot where the state variable resides or starts. This number may be very large and therefore its JSON value is represented as a string.
- `type` is an identifier used as key to the variable's type information (described in the following)

The given `type`, in this case `t_uint256` represents an element in `types`, which has the form :

```
{
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32",
}
```

where

- `encoding` how the data is encoded in storage, where the possible values are :
  - `inplace` : data is laid out contiguously in storage (see *above*).
  - `mapping` : Keccak-256 hash-based method (see *above*).
  - `dynamic_array` : Keccak-256 hash-based method (see *above*).
  - `bytes` : single slot or Keccak-256 hash-based depending on the data size (see *above*).
- `label` is the canonical type name.
- `numberOfBytes` is the number of used bytes (as a decimal string). Note that if `numberOfBytes > 32` this means that more than one slot is used.

Some types have extra information besides the four above. Mappings contain its `key` and `value` types (again referencing an entry in this mapping of types), arrays have its `base` type, and structs list their `members` in the same format as the top-level `storage` (see *above*).

Note : The JSON output format of a contract's storage layout is still considered experimental and is subject to change in non-breaking releases of Solidity.

The following example shows a contract and its storage layout, containing value and reference types, types that are encoded packed, and nested types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
contract A {
    struct S {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    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,
```

(suite sur la page suivante)

(suite de la page précédente)


```
      "contract": "fileA:A",
      "label": "map",
      "offset": 0,
      "slot": "7",
      "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
    },
    {
      "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",
```


(suite sur la page suivante)

(suite de la page précédente)


```
      "numberOfBytes": "1"
    },
    "t_bytes_storage": {
      "encoding": "bytes",
      "label": "bytes",
      "numberOfBytes": "32"
    },
    "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",
```


(suite sur la page suivante)

(suite de la page précédente)

```
          "type": "t_array(t_uint256)2_storage"
        },
        {
          "astId": 12,
          "contract": "fileA:A",
          "label": "dynArray",
          "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"
    }
  }
}
```

3.16 Layout in Memory

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows :

- `0x00` - `0x3f` (64 bytes) : scratch space for hashing methods
- `0x40` - `0x5f` (32 bytes) : currently allocated memory size (aka. free memory pointer)
- `0x60` - `0x7f` (32 bytes) : zero slot

Scratch space can be used between statements (i.e. within inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory pointer points to `0x80` initially).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `bytes1[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

Avertissement : There are some operations in Solidity that need a temporary memory area larger than 64 bytes and therefore will not fit into the scratch space. They will be placed where the free memory points to, but given their short lifetime, the pointer is not updated. The memory may or may not be zeroed out. Because of this, one should not expect the free memory to point to zeroed out memory.

While it may seem like a good idea to use `msize` to arrive at a definitely zeroed out memory area, using such a pointer non-temporarily without updating the free memory pointer can have unexpected results.

3.16.1 Differences to Layout in Storage

As described above the layout in memory is different from the layout in *storage*. Below there are some examples.

Example for Difference in Arrays

The following array occupies 32 bytes (1 slot) in storage, but 128 bytes (4 items with 32 bytes each) in memory.

```
uint8[4] a;
```

Example for Difference in Struct Layout

The following struct occupies 96 bytes (3 slots of 32 bytes) in storage, but 128 bytes (4 items with 32 bytes each) in memory.

```
struct S {
    uint a;
    uint b;
    uint8 c;
    uint8 d;
}
```

3.17 Layout of Call Data

The input data for a function call is assumed to be in the format defined by the *ABI specification*. Among others, the ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

Arguments for the constructor of a contract are directly appended at the end of the contract's code, also in ABI encoding. The constructor will access them through a hard-coded offset, and not by using the `codesize` opcode, since this of course changes when appending data to the code.

3.18 Cleaning Up Variables

When a value is shorter than 256 bit, in some cases the remaining bits must be cleaned. The Solidity compiler is designed to clean such remaining bits before any operations that might be adversely affected by the potential garbage in the remaining bits. For example, before writing a value to memory, the remaining bits need to be cleared because the memory contents can be used for computing hashes or sent as the data of a message call. Similarly, before storing a value in the storage, the remaining bits need to be cleaned because otherwise the garbled value can be observed.

Note that access via inline assembly is not considered such an operation : If you use inline assembly to access Solidity variables shorter than 256 bits, the compiler does not guarantee that the value is properly cleaned up.

Moreover, we do not clean the bits if the immediately following operation is not affected. For instance, since any non-zero value is considered `true` by `JUMPI` instruction, we do not clean the boolean values before they are used as the condition for `JUMPI`.

In addition to the design principle above, the Solidity compiler cleans input data when it is loaded onto the stack.

Different types have different rules for cleaning up invalid values :

Type	Valid Values	Invalid Values Mean
enum of n members	0 until n - 1	exception
bool	0 or 1	1
signed integers	sign-extended word	currently silently wraps ; in the future exceptions will be thrown
unsigned integers	higher bits zeroed	currently silently wraps ; in the future exceptions will be thrown

3.19 Source Mappings

As part of the AST output, the compiler provides the range of the source code that is represented by the respective node in the AST. This can be used for various purposes ranging from static analysis tools that report errors based on the AST and debugging tools that highlight local variables and their uses.

Furthermore, the compiler can also generate a mapping from the bytecode to the range in the source code that generated the instruction. This is again important for static analysis tools that operate on bytecode level and for displaying the current position in the source code inside a debugger or for breakpoint handling. This mapping also contains other information, like the jump type and the modifier depth (see below).

Both kinds of source mappings use integer identifiers to refer to source files. The identifier of a source file is stored in `output['sources'][sourceName]['id']` where `output` is the output of the standard-json compiler interface parsed as JSON. For some utility routines, the compiler generates « internal » source files that are not part of the original input but are referenced from the source mappings. These source files together with their identifiers can be obtained via `output['contracts'][sourceName][contractName]['evm']['bytecode']['generatedSources']`.

Note : In the case of instructions that are not associated with any particular source file, the source mapping assigns an integer identifier of `-1`. This may happen for bytecode sections stemming from compiler-generated inline assembly statements.

The source mappings inside the AST use the following notation :

```
s:l:f
```

Where `s` is the byte-offset to the start of the range in the source file, `l` is the length of the source range in bytes and `f` is the source index mentioned above.

The encoding in the source mapping for the bytecode is more complicated : It is a list of `s:l:f:j:m` separated by `;`. Each of these elements corresponds to an instruction, i.e. you cannot use the byte offset but have to use the instruction offset (push instructions are longer than a single byte). The fields `s`, `l` and `f` are as above. `j` can be either `i`, `o` or `-` signifying whether a jump instruction goes into a function, returns from a function or is a regular jump as part of e.g. a loop. The last field, `m`, is an integer that denotes the « modifier depth ». This depth is increased whenever the placeholder statement (_) is entered in a modifier and decreased when it is left again. This allows debuggers to track tricky cases like the same modifier being used twice or multiple placeholder statements being used in a single modifier.

In order to compress these source mappings especially for bytecode, the following rules are used :

- If a field is empty, the value of the preceding element is used.
- If a `:` is missing, all following fields are considered empty.

This means the following source mappings represent the same information :

```
1:2:1;1:9:1;2:1:2;2:1:2;2:1:2
```

```
1:2:1;:9;2:1:2;;
```

Important to note is that when the *verbatim* builtin is used, the source mappings will be invalid : The builtin is considered a single instruction instead of potentially multiple.

3.20 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 experimental Yul IR for a Solidity source. Similarly, one can use `solc --strict-assembly --optimize` for a stand-alone Yul mode.

You can find more details on both optimizer modules and their optimization steps below.

3.20.1 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.

3.20.2 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`).

3.20.3 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`.

Note : 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.

3.20.4 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 »).

3.20.5 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.

Avertissement : 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.

— *BlockFlattener*.
— *CircularReferencesPruner*.
— *CommonSubexpressionEliminator*.
— *ConditionalSimplifier*.
— *ConditionalUnsimplifier*.
— *ControlFlowSimplifier*.
— *DeadCodeEliminator*.
— *EqualStoreEliminator*.
— *EquivalentFunctionCombiner*.
— *ExpressionJoiner*.
— *Expression Simplifier*.
— *ExpressionSplitter*.
— *ForLoopConditionIntoBody*.
— *ForLoopConditionOutOfBody*.
— *ForLoopInitRewriter*.
— *ExpressionInliner*.
— *FullInliner*.
— *FunctionGrouper*.
— *FunctionHoister*.
— *FunctionSpecializer*.
— *LiteralRematerialiser*.
— *LoadResolver*.
— *LoopInvariantCodeMotion*.
— *RedundantAssignEliminator*.
— *ReasoningBasedSimplifier*.
— *Rematerialiser*.
— *SSAReverser*.
— *SSATransform*.
— *StructuralSimplifier*.
— *UnusedFunctionParameterPruner*.
— *UnusedPruner*.
— *VarDeclInitializer*.

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'
```

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.

Available abbreviations are listed in the Yul optimizer docs.

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 or revert.

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.

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.

3.21 Métadonnées du contrat

Le compilateur Solidity génère automatiquement un fichier JSON, le contrat qui contient des informations sur le contrat compilé. Vous pouvez utiliser ce fichier pour interroger la version du compilateur, les sources utilisées, l'ABI et la documentation NatSpec, pour interagir de manière plus sûre avec le contrat et vérifier son code source.

Le compilateur ajoute par défaut le hash IPFS du fichier de métadonnées à la fin du bytecode (pour plus de détails, voir ci-dessous) de chaque contrat, de sorte que vous pouvez le fichier de manière authentifiée sans avoir à recourir à un fournisseur de données centralisé. Les autres options disponibles sont le hachage Swarm et ne pas ajouter le hachage des métadonnées au bytecode. Elles peuvent être configurées via l'interface *Standard JSON Interface*.

Vous devez publier le fichier de métadonnées sur IPFS, Swarm, ou un autre service pour que que d'autres puissent y accéder. Vous créez le fichier en utilisant la commande `solc --metadata`. qui génère un fichier appelé

`ContractName_meta.json`. Ce fichier contient les références IPFS et Swarm au code source et le fichier de métadonnées.

Le fichier de métadonnées a le format suivant. L'exemple ci-dessous est présenté de manière lisible par l'homme. Des métadonnées correctement formatées doivent utiliser correctement les guillemets, réduire les espaces blancs au minimum et trier les clés de tous les objets pour arriver à un formatage unique. Les commentaires ne sont pas autorisés et ne sont utilisés ici qu'à à des fins explicatives.

```
{
  // Obligatoire : La version du format de métadonnées
  "version": "1",
  // Obligatoire : Langue du code source, sélectionne essentiellement une "sous-version"
  // de la spécification
  "language": "Solidity",
  // Obligatoire : Détails sur le compilateur, le contenu est spécifique
  // au langage.
  "compiler": {
    // Requis pour Solidity : Version du compilateur
    "version": "0.4.6+commit.2dabbdf0.Emscripten.clang",
    // Facultatif : hachage du binaire du compilateur qui a produit cette sortie.
    "keccak256": "0x123..."
  },
  // Requis : Fichiers source de compilation/unités de source, les clés sont des noms de␣
↪fichiers.
  "sources":
  {
    "myFile.sol": {
      // Requis : keccak256 hash du fichier source
      "keccak256": "0x123...",
      // Obligatoire (sauf si "content" est utilisé, voir ci-dessous) : URL(s) triée(s)
      // vers le fichier source, le protocole est plus ou moins arbitraire, mais une
      // une URL Swarm est recommandée
      "urls": [ "bzzr://56ab..." ],
      // Facultatif : Identifiant de la licence SPDX tel qu'indiqué dans le fichier␣
↪source.
      "license": "MIT"
    },
    "destructible": {
      // Requis : keccak256 hash du fichier source
      "keccak256": "0x234...",
      // Obligatoire (sauf si "url" est utilisé) : contenu littéral du fichier source.
      "content": "contract destructible is owned { function destroy() { if (msg.sender␣
↪== owner) selfdestruct(owner); } }"
    }
  },
  // Requis : Paramètres du compilateur
  "settings":
  {
    // Requis pour Solidity : Liste triée de réaffectations
    "remappings": [ ":g=/dir" ],
    // Facultatif : Paramètres de l'optimiseur. Les champs "enabled" et "runs" sont␣
↪obsolètes
    // et ne sont fournis que pour des raisons de compatibilité ascendante.
    "optimizer": {
```

(suite sur la page suivante)

(suite de la page précédente)

```
      "enabled": true,
      "runs": 500,
      "details": {
        // peephole a la valeur par défaut "true".
        "peephole": true,
        // la valeur par défaut de l'inliner est "true".
        "inliner": true,
        // jumpdestRemover a la valeur par défaut "true".
        "jumpdestRemover": true,
        "orderLiterals": false,
        "deduplicate": false,
        "cse": false,
        "constantOptimizer": false,
        "yul": true,
        // Facultatif : Présent uniquement si "yul" est "true".
        "yulDetails": {
          "stackAllocation": false,
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    "metadata": {
      // Reflète le paramètre utilisé dans le json d'entrée, la valeur par défaut est␣
↪false.
      "useLiteralContent": true,
      // Reflète le paramètre utilisé dans le json d'entrée, la valeur par défaut est
↪"ipfs".
      "bytecodeHash": "ipfs"
    },
    // Requis pour Solidity : Fichier et nom du contrat ou de la bibliothèque pour␣
↪lesquels ces
    // métadonnées est créée pour.
    "compilationTarget": {
      "myFile.sol": "MyContract"
    },
    // Requis pour Solidity : Adresses des bibliothèques utilisées
    "libraries": {
      "MyLib": "0x123123..."
    }
  },
  // Requis : Informations générées sur le contrat.
  "output":
  {
    // Requis : Définition ABI du contrat
    "abi": [/* ... */],
    // Requis : Documentation du contrat par l'utilisateur de NatSpec
    "userdoc": [/* ... */],
    // Requis : Documentation du contrat par le développeur NatSpec
    "devdoc": [/* ... */]
  }
}
```

Avertissement : Comme le bytecode du contrat résultant contient le hachage des métadonnées par défaut, toute modification des métadonnées peut entraîner une modification du bytecode. Cela inclut changement de nom de fichier ou de chemin, et puisque les métadonnées comprennent un hachage de toutes les sources utilisées, un simple changement d'espace résulte en des métadonnées différentes, et un bytecode différent.

Note : La définition ABI ci-dessus n'a pas d'ordre fixe. Il peut changer avec les versions du compilateur. Cependant, à partir de la version 0.5.12 de Solidity, le tableau maintient un certain ordre. ordre.

3.21.1 Encodage du hachage des métadonnées dans le bytecode

Parce que nous pourrions supporter d'autres façons de récupérer le fichier de métadonnées à l'avenir, le mappage `{"ipfs" : <Hachage IPFS>, "solc" : <version du compilateur>}` est stockée CBOR-encodé. Puisque la cartographie peut contenir plus de clés (voir ci-dessous) et que le début de cet encodage n'est pas facile à trouver, sa longueur est ajoutée dans un encodage big-endian de deux octets. La version actuelle du compilateur Solidity ajoute généralement l'élément suivant à la fin du bytecode déployé.

```
0xa2
0x64 'i' 'p' 'f' 's' 0x58 0x22 <34 octets hachage IPFS>
0x64 's' 'o' 'l' 'c' 0x43 <Codage de la version sur 3 octets>
0x00 0x33
```

Ainsi, afin de récupérer les données, la fin du bytecode déployé peut être vérifiée, pour correspondre à ce modèle et utiliser le hachage IPFS pour récupérer le fichier.

Alors que les versions de solc utilisent un encodage de 3 octets de la version comme indiqué ci-dessus (un octet pour chaque numéro de version majeure, mineure et de patch), les versions préversées utiliseront à la place une chaîne de version complète incluant le hachage du commit et la date de construction.

Note : Le mappage CBOR peut également contenir d'autres clés, il est donc préférable de décoder complètement les données plutôt que de se fier à ce qu'elles commencent par `0xa264`. Par exemple, si des fonctionnalités expérimentales qui affectent la génération de code sont utilisées, le mappage contiendra également `"experimental" : true`.

Note : Le compilateur utilise actuellement le hachage IPFS des métadonnées par défaut, mais il peut aussi utiliser le hachage bzzr1 ou un autre hachage à l'avenir, donc ne vous ne comptez pas sur cette séquence pour commencer avec `0xa2 0x64 'i' 'p' 'f' 's'`. Nous ajouterons peut-être des données supplémentaires à cette structure CBOR.

3.21.2 Utilisation pour la génération automatique d'interface et NatSpec

Les métadonnées sont utilisées de la manière suivante : Un composant qui veut interagir avec un contrat (par exemple Mist ou tout autre porte-monnaie) récupère le code du contrat, à partir de là, le hachage IPFS/Swarm d'un fichier qui est ensuite récupéré. Ce fichier est décodé en JSON dans une structure comme ci-dessus.

Le composant peut alors utiliser l'ABI pour générer automatiquement une interface utilisateur rudimentaire pour le contrat.

En outre, le portefeuille peut utiliser la documentation utilisateur NatSpec pour afficher un message de confirmation à l'utilisateur chaque fois qu'il interagit avec le contrat, ainsi qu'une demande d'autorisation pour la signature de la transaction.

Pour plus d'informations, lisez *Format de la spécification en langage naturel d'Ethereum (NatSpec)*.

3.21.3 Utilisation pour la vérification du code source

Afin de vérifier la compilation, les sources peuvent être récupérées sur IPFS/Swarm via le lien dans le fichier de métadonnées. Le compilateur de la version correcte (qui est vérifié pour faire partie des compilateurs « officiels ») est invoqué sur cette entrée avec les paramètres spécifiés. Le bytecode résultant est comparé aux données de la transaction de création ou aux données de l'opcode `CREATE`. Cela vérifie automatiquement les métadonnées puisque leur hachage fait partie du bytecode. Les données en excès correspondent aux données d'entrée du constructeur, qui doivent être décodées selon l'interface et présentées à l'utilisateur.

Dans le référentiel sourcify (npm package) vous pouvez voir un exemple de code qui montre comment utiliser cette fonctionnalité.

3.22 Spécification ABI pour les contrats

3.22.1 Conception de base

L'interface binaire d'application de contrat (ABI) est le moyen standard d'interagir avec les contrats dans l'écosystème Ethereum, à la fois depuis l'extérieur de la blockchain et pour l'interaction entre les contrats. de l'extérieur de la blockchain que pour l'interaction entre contrats. Les données sont codées en fonction de leur type, comme décrit dans cette spécification. L'encodage n'est pas autodécrit et nécessite donc un schéma pour être décodé.

Nous supposons que les fonctions d'interface d'un contrat sont fortement typées, connues au moment de la compilation et statiques. Nous supposons que tous les contrats auront les définitions d'interface de tous les contrats qu'ils appellent disponibles au moment de la compilation.

Cette spécification ne concerne pas les contrats dont l'interface est dynamique ou connue uniquement au moment de l'exécution.

3.22.2 Sélecteur de fonctions

Les quatre premiers octets des données d'appel d'une fonction spécifient la fonction à appeler. Il s'agit des premiers (gauche, ordre supérieur en big-endian) quatre octets du hachage Keccak-256 de la signature de la fonction. la fonction. La signature est définie comme l'expression canonique du prototype de base sans spécificateur d'emplacement de données, c'est-à-dire qu'il s'agit de l'expression canonique de la fonction. spécificateur d'emplacement de données, c'est-à-dire le nom de la fonction avec la liste des types de paramètres entre parenthèses. Les types de paramètres sont séparés par une simple virgule - aucun espace n'est utilisé.

Note : Le type de retour d'une fonction ne fait pas partie de cette signature. Dans *Solidity's function overloading* les types de retour ne sont pas pris en compte. La raison est de garder la résolution d'appel de fonction indépendante du contexte. La description *JSON de l'ABI* contient cependant des entrées et des sorties.

3.22.3 Codage des arguments

À partir du cinquième octet, les arguments codés suivent. Ce codage est également utilisé à d'autres d'autres endroits, par exemple les valeurs de retour et les arguments d'événements sont codés de la même manière, sans les quatre octets spécifiant la fonction.

3.22.4 Types

Les types élémentaires suivants existent :

- `uint<M>` : type de nombre entier non signé de `M` bits, `0 < M <= 256`, `M % 8 == 0`. Par exemple, `uint32`, `uint8`, `uint256`.
- `int<M>` : type d'entier signé en complément à deux de `M` bits, `0 < M <= 256`, `M % 8 == 0`.
- `address` : équivalent à `uint160`, sauf pour l'interprétation supposée et le typage du langage. Pour calculer le sélecteur de fonction, on utilise `address`.
- `uint`, `int` : synonymes de `uint256`, `int256` respectivement. Pour calculer le sélecteur de fonction sélecteur de fonction, `uint256` et `int256` doivent être utilisés.
- `bool` : équivalent à `uint8` restreint aux valeurs 0 et 1. Pour le calcul du sélecteur de fonction, `bool` est utilisé.
- `fixed<M>x<N>` : nombre décimal signé en virgule fixe de `M` bits, `8 <= M <= 256`, `M % 8 == 0`, et `0 < N <= 80`, qui désigne la valeur *v* ` comme `v / (10 ** N)`.
- `ufixed<M>x<N>` : variante non signée de `fixed<M>x<N>`.
- `fixed`, `ufixed` : synonymes de `fixed128x18`, `ufixed128x18` respectivement. Pour calculer le sélecteur de fonction, il faut utiliser *fixed128x18* ` et *ufixed128x18* `.
- `bytes<M>` : type binaire de `M` octets, `0 < M <= 32`.
- `fonction` : une adresse (20 octets) suivie d'un sélecteur de fonction (4 octets). Encodé de manière identique à `bytes24`.

Le type de tableau (de taille fixe) suivant existe :

- `<type>[M]` : un tableau de longueur fixe de `M` éléments, `M >= 0`, du type donné.

Les types de taille non fixe suivants existent :

- `bytes` : séquence d'octets de taille dynamique.
- `string` : chaîne unicode de taille dynamique supposée être encodée en UTF-8.
- `<type>[]` : un tableau de longueur variable d'éléments du type donné.

Les types peuvent être combinés en un tuple en les mettant entre parenthèses, séparés par des virgules :

- `(T1,T2,...,Tn)` : tuple constitué des types `T1`, …, `Tn`, `n >= 0`

Il est possible de former des tuples de tuples, des tableaux de tuples et ainsi de suite. Il est également possible de former des n-uplets zéro (où `n == 0`).

Correspondance entre Solidity et les types ABI

Solidity supporte tous les types présentés ci-dessus avec les mêmes noms, à l'exception des tuples. l'exception des tuples. Par contre, certains types Solidity ne sont pas supportés par l'ABI par l'ABI. Le tableau suivant montre sur la colonne de gauche les types Solidity qui qui ne font pas partie de l'ABI et, dans la colonne de droite, les types ABI qui les représentent. qui les représentent.

Solidity	ABI
addresse payable<address>`|``address`	
contract	`address`
enum	`uint8`
types de valeurs définies par l'utilisateur	
struct	`tuple`

Avertissement : Avant la version `0.8.0` les enums pouvaient avoir plus de 256 membres et étaient représentés par le plus petit type de plus petit type d'entier juste assez grand pour contenir la valeur de n'importe quel membre.

3.22.5 Critères de conception pour l'encodage

Le codage est conçu pour avoir les propriétés suivantes, qui sont particulièrement utiles si certains arguments sont des tableaux imbriqués :

1. Le nombre de lectures nécessaires pour accéder à une valeur est au plus égal à la profondeur de la valeur dans la structure du tableau d'arguments. dans la structure du tableau d'arguments, c'est-à-dire que quatre lectures sont nécessaires pour récupérer `a_i[k][l][r]`. Dans une version version précédente de l'ABI, le nombre de lectures était linéairement proportionnel au nombre total de paramètres dynamiques dans le pire des cas. dynamiques dans le pire des cas.
2. Les données d'une variable ou d'un élément de tableau ne sont pas entrelacées avec d'autres données et elles sont relocalisables, c'est-à-dire qu'elles n'utilisent que des « adresses » relatives.

3.22.6 Spécification formelle de l'encodage

Nous distinguons les types statiques et dynamiques. Les types statiques sont codés sur place et les types dynamiques sont dynamiques sont codés à un emplacement alloué séparément après le bloc actuel.

Définition : Les types suivants sont appelés « dynamiques » :
— `bytes`
— Chaîne de caractères
— `T[]` pour tout `T`
— `T[k]` pour tout `T` dynamique et tout `k >= 0`
— `(T1,...,Tk)` si `Ti` est dynamique pour tout `1 <= i <= k`
Tous les autres types sont dits « statiques ».

Définition : `len(a)` est le nombre d'octets dans une chaîne binaire `a`. Le type de `len(a)` est supposé être `uint256`.

Nous définissons `enc`, le codage réel, comme une correspondance entre les valeurs des types ABI et les chaînes binaires telles que que `len(enc(X))` dépend de la valeur de `X` si et seulement si le type de `X` est dynamique.

Définition : Pour toute valeur ABI `X`, on définit récursivement `enc(X)`, en fonction du type de `X`. du type de `X` qui est
— `(T1,...,Tk)` pour `k >= 0` et tout type `T1`, ..., `Tk`
`enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`
où `X = (X(1), ..., X(k))` et tête » et » queue » sont définies comme suit pour » Ti » :
si `Ti` est statique :
`head(X(i)) = enc(X(i))` et `tail(X(i)) = ""` (la chaîne vide)
sinon, c'est-à-dire si *Ti* ` est dynamique :
`head(X(i)) = enc(len( head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1)) ))`
`tail(X(i)) = enc(X(i))`
Notez que dans le cas dynamique, `head(X(i))` est bien défini car les longueurs des parties de tête parties de la tête ne dépendent que des types et non des valeurs. La valeur de `head(X(i))` est le décalage du début de `tail(X(i))`. du début de `tail(X(i))` par rapport au début de `enc(X)`.
— `T[k]` pour tout `T` et `k` :
`enc(X) = enc((X[0], ..., X[k-1]))`
c'est-à-dire qu'il est codé comme s'il s'agissait d'un tuple avec `k` éléments du même type.
— `T[]` où `X` a *k* ` éléments (`k` est supposé être de type `uint256`) :
`enc(X) = enc(k) enc([X[0], ..., X[k-1]])`
c'est-à-dire qu'il est encodé comme s'il s'agissait d'un tableau de taille statique `k`, préfixé par le le nombre d'éléments.

- `bytes`, de longueur `k` (qui est supposé être de type `uint256`) :
  `enc(X) = enc(k) pad_right(X)`, c'est-à-dire que le nombre d'octets est codé sous forme de `uint256` suivi de la valeur réelle de `X` en tant que séquence d'octets, suivie par le nombre minimal d'octets zéro pour que `len(enc(X))` soit un multiple de 32.
- `Chaîne de caractères` :
  `enc(X) = enc(enc_utf8(X))`, c'est-à-dire que `X` est codé en UTF-8 et que cette valeur est interprétée comme étant du type `bytes` et encodée plus loin. Notez que la longueur utilisée dans ce codage est le nombre d'octets de la chaîne encodée en UTF-8, et non son nombre de caractères.
- `uint<M>` : `enc(X)` est le codage big-endian de `X`, complété du côté gauche par des octets zéro. d'ordre supérieur (gauche) avec des octets zéro de sorte que la longueur soit de 32 octets.
- Adresse : comme dans le cas de `uint160`.
- `int<M>` : `enc(X)` est le code de complément à deux big-endian de `X`, complété sur le côté supérieur (gauche) par des octets `0xff` pour les `X` négatifs et par des octets zéro pour les `X` non négatifs, de sorte que la longueur soit de 32 octets.
- `bool` : comme dans le cas de `uint8`, où `1` est utilisé pour `vrai` et `0` pour `false`.
- `fixed<M>x<N>` : ``enc(X)` est `enc(X * 10**N)` où `X * 10**N` est interprété comme un `int256`.
- `fixed` : comme dans le cas `fixed128x18`
- `ufixed<M>x<N>` : ``enc(X)` est `enc(X * 10**N)` où `X * 10**N` est interprété comme un `uint256`.
- `ufixed` : comme dans le cas `ufixed128x18`
- `bytes<M>` : `enc(X)` est la séquence d'octets dans `X` remplie de zéros de queue jusqu'à une longueur de 32 octets.

Notez que pour tout `X`, `len(enc(X))` est un multiple de 32.

3.22.7 Sélecteur de fonctions et codage des arguments

En somme, un appel à la fonction `f` avec les paramètres `a_1, ..., a_n` est encodé comme suit

```
fonction_selector(f) enc((a_1, ..., a_n))
```

et les valeurs de retour `v_1, ..., v_k` de `f` sont codées en tant que

```
enc((v_1, ..., v_k))
```

c'est-à-dire que les valeurs sont combinées en un tuple et codées.

3.22.8 Exemples

Étant donné le contrat :

```
// 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 {}
}
```

Ainsi, pour notre exemple `Foo`, si nous voulions appeler `baz` avec les paramètres `69` et `true`, nous passerions 68 octets au total, qui peuvent être décomposés en :

- `0xcdcd77c0` : l'ID de la méthode. Il s'agit des 4 premiers octets du hachage de Keccak de la forme la forme ASCII de la signature `baz(uint32,bool)`.
- `0x0000000000000000000000000000000000000000000000000000000000000045` : le premier paramètre, une valeur uint32 69 remplie de 32 octets

— `0x0000000000000000000000000000000000000000000000000000000000000001` : le deuxième paramètre, un booléen `vrai`, padded to 32 bytes

Au total :

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000
```

Elle renvoie un seul `bool`. Si, par exemple, elle devait retourner `false`, sa sortie serait le tableau d'octets unique `0x0000000000000000000000000000000000000000000000000000000000000000`, un seul bool.

Si nous voulions appeler `bar` avec l'argument `["abc", "def"]`, nous passerions 68 octets au total, répartis en :

— `0xfce353f6` : l'identifiant de la méthode. Celui-ci est dérivé de la signature `bar(bytes3[2])`.

— `0x6162630000000000000000000000000000000000000000000000000000000000` : la première partie du premier paramètre, une valeur `bytes3` « abc »`` (alignée à gauche).

— `0x6465660000000000000000000000000000000000000000000000000000000000` : la deuxième partie du premier paramètre, une valeur `bytes3` (alignée à gauche). paramètre, un `bytes3` de valeur `"def"` (aligné à gauche).

Au total :

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000000000006465660000000000000000000000000
```

Si nous voulions appeler `sam` avec les arguments `"dave"`, `true` et `[1,2,3]`, nous devrions passerait 292 octets au total, répartis comme suit :

— `0xa5643bf2` : l'identifiant de la méthode. Celui-ci est dérivé de la signature `sam(bytes,bool,uint256[])`. Notez que `uint` est remplacé par sa représentation canonique `uint256`.

— `0x0000000000000000000000000000000000000000000000000000000000000060` : l'emplacement de la partie données du premier paramètre (type dynamique), mesuré en octets à partir du début du bloc d'arguments. Dans ce cas, `0x60`.

— `0x0000000000000000000000000000000000000000000000000000000000000001` : le deuxième paramètre : booléen vrai.

— `0x00000000000000000000000000000000000000000000000000000000000000a0` : l'emplacement de la partie données du troisième paramètre (type dynamique), mesuré en octets. Dans ce cas, `0xa0`.

— `0x0000000000000000000000000000000000000000000000000000000000000004` : la partie données du premier argument, elle commence par la longueur du tableau d'octets en éléments, dans ce cas, 4.

— `0x6461766500000000000000000000000000000000000000000000000000000000` : le contenu du premier argument : l'encodage UTF-8 (équivalent à l'ASCII dans ce cas) de `"dave"`, padded sur la droite à 32 octets.

— `0x0000000000000000000000000000000000000000000000000000000000000003` : la partie données du troisième argument, elle commence par la longueur du tableau en éléments, dans ce cas, 3.

— `0x0000000000000000000000000000000000000000000000000000000000000001` : la première entrée du troisième paramètre.

— `0x0000000000000000000000000000000000000000000000000000000000000002` : la deuxième entrée du troisième paramètre.

— `0x0000000000000000000000000000000000000000000000000000000000000003` : la troisième entrée du troisième paramètre.

Au total :

```
0xa5643bf2000000000000000000000000000000000000000000000000000000000000006000000000000000000000000000000000
```

3.22.9 Utilisation des types dynamiques

Un appel à une fonction dont la signature est `f(uint,uint32[],bytes10,bytes)` avec les valeurs suivantes `(0x123, [0x456, 0x789], "1234567890", "Hello, world !")` est codé de la manière suivante :

Nous prenons les quatre premiers octets de `sha3("f(uint256,uint32[],bytes10,bytes)")`, c'est-à-dire `0x8be65246`. Ensuite, nous encodons les parties de tête des quatre arguments. Pour les types statiques `uint256` et `bytes10`, ce sont directement les valeurs que nous voulons passer, alors que pour les types dynamiques `uint32[]` et `bytes`, nous utilisons le décalage en octets par rapport au début de leur zone de données, mesuré à partir du début de l'encodage de la valeur (c'est-à-dire pas de l'encodage de la valeur). (c'est-à-dire sans compter les quatre premiers octets contenant le hachage de la signature de la fonction). Ces valeurs sont les suivantes

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (décalage du début de la partie données du second paramètre, 4*32 octets, exactement la taille de la partie tête)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (décalage du début de la partie données du quatrième paramètre = décalage du début de la partie données du premier paramètre dynamique + taille de la partie données du premier paramètre dynamique = 4*32 + 3*32 (voir ci-dessous))

Ensuite, la partie données du premier argument dynamique, `[0x456, 0x789]`, est la suivante :

- `0x0000000000000000000000000000000000000000000000000000000000000002` (nombre d'éléments du tableau, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (premier élément)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (deuxième élément)

Enfin, nous encodons la partie données du second argument dynamique, « Hello, world ! » :

- `0x000000000000000000000000000000000000000000000000000000000000000d` (nombre d'éléments (octets dans ce cas) : 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` (`"Hello, world !"` padded to 32 bytes on the right)

Au total, le codage est le suivant (nouvelle ligne après le sélecteur de fonction et chaque 32 octets pour plus de clarté) :

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Appliquons le même principe pour encoder les données d'une fonction de signature `g(uint[][],string[])` avec les valeurs `([1, 2], [3]], ["un", "deux", "trois"])` mais commençons par les parties les plus atomiques de l'encodage :

D'abord, nous encodons la longueur et les données du premier tableau dynamique intégré `[1, 2]` du premier tableau racine `[[1, 2], [3]]` :

- `0x0000000000000000000000000000000000000000000000000000000000000002` (nombre d'éléments du premier tableau, 2 ; les éléments eux-mêmes sont 1 et 2)
- `0x0000000000000000000000000000000000000000000000000000000000000001` (premier élément)
- `0x0000000000000000000000000000000000000000000000000000000000000002` (deuxième élément)

Ensuite, nous codons la longueur et les données du deuxième tableau dynamique intégré `[3]` du premier tableau racine `[[1, 2], [3]]` :

— 0x0000000000000000000000000000000000000000000000000000000000000001 (nombre d'éléments dans le second tableau, 1 ; l'élément est 3)
— 0x0000000000000000000000000000000000000000000000000000000000000003 (premier élément)

Nous devons ensuite trouver les décalages *a`* et *b`* pour leurs tableaux dynamiques respectifs `[1, 2]` et `[3]`. Pour calculer les décalages, nous pouvons examiner les données codées du premier tableau racine `[[1, 2], [3]]`. en énumérant chaque ligne du codage :

```
0 - a                                                               - décalage de [1, 2]
1 - b                                                               - décalage de [3]
2 - 0000000000000000000000000000000000000000000000000000000000000002 - compte pour [1, 2]
3 - 0000000000000000000000000000000000000000000000000000000000000001 - codage de 1
4 - 0000000000000000000000000000000000000000000000000000000000000002 - codage de 2
5 - 0000000000000000000000000000000000000000000000000000000000000001 - compte pour [3]
6 - 0000000000000000000000000000000000000000000000000000000000000003 - codage de 3
```

Le décalage `a` pointe vers le début du contenu du tableau `[1, 2]` qui est la ligne 2 (64 octets). 2 (64 octets) ; ainsi `a` = `0x0000000000000000000000000000000000000000000000000000000000000040`.

Le décalage `b` pointe vers le début du contenu du tableau `[3]` qui est la ligne 5 (160 octets) ; donc `b` = `0x00000000000000000000000000000000000000000000000000000000000000a0`.

Ensuite, nous encodons les chaînes intégrées du deuxième tableau racine :

— 0x0000000000000000000000000000000000000000000000000000000000000003 (nombre de caractères dans le mot `"one"`)
— 0x6f6e650000000000000000000000000000000000000000000000000000000000 (représentation utf8 du mot `"one"`)
— 0x0000000000000000000000000000000000000000000000000000000000000003 (nombre de caractères dans le mot `"two"`)
— 0x74776f0000000000000000000000000000000000000000000000000000000000 (représentation utf8 du mot `"two"`)
— 0x0000000000000000000000000000000000000000000000000000000000000005 (nombre de caractères dans le mot `"three"`)
— 0x7468726565000000000000000000000000000000000000000000000000000000 (représentation utf8 du mot `"three"`)

Parallèlement au premier tableau racine, puisque les chaînes sont des éléments dynamiques, nous devons trouver leurs décalages `c`, `d` et `e` :

```
0 - c                                                               - décalage pour "un"
1 - d                                                               - décalage pour
↪"deux"
2 - e                                                               - décalage pour
↪"trois"
3 - 0000000000000000000000000000000000000000000000000000000000000003 - compte pour "un"
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - codage pour "un"
5 - 0000000000000000000000000000000000000000000000000000000000000003 - compte pour "deux"
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - codage pour "deux"
7 - 0000000000000000000000000000000000000000000000000000000000000005 - compte pour "trois
↪"
8 - 7468726565000000000000000000000000000000000000000000000000000000 - codage pour "trois
↪"
```

L'offset `c` pointe vers le début du contenu de la chaîne `"one"` qui est la ligne 3 (96 octets) ; donc `c` = `0x0000000000000000000000000000000000000000000000000000000000000060`.

Le décalage `d` pointe vers le début du contenu de la chaîne `"two"` qui est la ligne 5 (160 octets) ; donc `d` = `0x00000000000000000000000000000000000000000000000000000000000000a0`.

Le décalage `e` pointe vers le début du contenu de la chaîne `"trois"` qui est la ligne 7 (224 octets) ; donc `e = 0x00000000000000000000000000000000000000000000000000000000000000e0`.

Notez que les encodages des éléments intégrés des tableaux racines ne sont pas dépendants les uns des autres et ont les mêmes encodages pour une fonction avec une signature `g(string[],uint[][])`.

Ensuite, nous encodons la longueur du premier tableau racine :

— `0x0000000000000000000000000000000000000000000000000000000000000002` (nombre d'éléments dans le premier tableau racine, 2 ; les éléments eux-mêmes sont `[1, 2]` et `[3]`)

Ensuite, nous codons la longueur du deuxième tableau racine :

— `0x0000000000000000000000000000000000000000000000000000000000000003` (nombre de chaînes dans le deuxième tableau racine, 3 ; les chaînes elles-mêmes sont `"un"`, `"deux"` et `"trois"`)

Enfin, nous trouvons les décalages *f*` et `g` pour leurs tableaux dynamiques racines respectifs `[[1, 2], [3]]` et `["un", "deux", "trois"]`, et assemblons les pièces dans le bon ordre :

```
0x2289b18c                                                          - signature de la␣
↪fonction
 0 - f                                                              - décalage de [[1,␣
↪2], [3]]
 1 - g                                                              - décalage de ["un
↪", "deux", "trois"]
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - compte pour [[1,␣
↪2], [3]]
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - décalage de [1,␣
↪2]
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - décalage de [3]
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - compte pour [1,␣
↪2]
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - codage de 1
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - codage de 2
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - compte pour [3]
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - codage de 3
10 - 0000000000000000000000000000000000000000000000000000000000000003 - compte pour ["un
↪", "deux", "trois"]
11 - 0000000000000000000000000000000000000000000000000000000000000060 - décalage pour "un
↪"
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - décalage pour
↪"deux"
13 - 00000000000000000000000000000000000000000000000000000000000000e0 - décalage pour
↪"trois"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - compte pour "un"
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - codage de "un"
16 - 0000000000000000000000000000000000000000000000000000000000000003 - compte pour "deux
↪"
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - codage de "deux"
18 - 0000000000000000000000000000000000000000000000000000000000000005 - compte pour
↪"trois"
19 - 7468726565000000000000000000000000000000000000000000000000000000 - codage de "trois"
```

Le décalage `f` pointe vers le début du contenu du tableau `[[1, 2], [3]]` qui est la ligne 2 (64 octets) ; donc `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Le décalage `g` pointe vers le début du contenu du tableau `["one", "two", "three"]` qui est la ligne 10 (320 octets) ; donc `g = 0x0000000000000000000000000000000000000000000000000000000000000140`.

3.22.10 Événements

Les événements sont une abstraction du protocole de journalisation et de surveillance des événements d'Ethereum. Les entrées de journal fournissent l'adresse du contrat du contrat, une série de quatre sujets maximum et des données binaires de longueur arbitraire. Les événements exploitent la fonction existante ABI existante afin d'interpréter ceci (avec une spécification d'interface) comme une structure correctement typée.

Étant donné un nom d'événement et une série de paramètres d'événement, nous les divisons en deux sous-séries : celles qui sont indexées et celles qui ne le sont pas. ceux qui ne le sont pas. Ceux qui sont indexés, dont le nombre peut aller jusqu'à 3 (pour les événements non anonymes) ou 4 (pour les événements anonymes), sont utilisés avec le hachage Keccak de la signature de l'événement pour former les sujets de l'entrée du journal. Ceux qui ne sont pas indexés forment le tableau d'octets de l'événement.

En fait, une entrée de journal utilisant cette ABI est décrite comme suit :

- `address` : l'adresse du contrat (intrinsèquement fournie par Ethereum) ;
- `topics[0]` : `keccak(EVENT_NAME+ "("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` est une fonction qui renvoie simplement le nom du contrat. est une fonction qui renvoie simplement le type canonique d'un argument donné, par exemple, pour `uint indexé foo`, elle renverrait retournerait `uint256`). Cette valeur n'est présente dans `topics[0]` que si l'événement n'est pas déclaré comme `anonyme` ;
- `topics[n]` : `abi_encode(EVENT_INDEXED_ARGS[n - 1])` si l'événement n'est pas déclaré comme étant `anonyme`. ou `abi_encode(EVENT_INDEXED_ARGS[n])` s'il l'est (`EVENT_INDEXED_ARGS` est la série des `EVENT_ARGS` qui sont sont indexées) ;
- `data` : qui ne sont pas indexés, `abi_encode` est la fonction d'encodage ABI utilisée pour retourner une série de valeurs typées d'une fonction, comme décrit ci-dessus).

Pour tous les types d'une longueur maximale de 32 octets, le tableau `EVENT_INDEXED_ARGS` contient la valeur directement, avec un padding ou une extension de signe (pour les entiers signés) à 32 octets, comme pour le codage ABI normal. Cependant, pour tous les types « complexes » ou de longueur dynamique, y compris tous les tableaux, `string`, `bytes` et structs, `EVENT_INDEXED_ARGS` contiendra le hachage *Keccak* d'une valeur spéciale encodée sur place (voir *Codage des paramètres d'événements indexés*), plutôt que la valeur encodée directement. Cela permet aux applications d'interroger efficacement les valeurs de types de longueur dynamique dynamiques (en définissant le hachage de la valeur encodée comme sujet), mais les applications ne peuvent pas de décoder les valeurs indexées qu'elles n'ont pas demandées. Pour les types de longueur dynamique, les développeurs d'applications doivent faire un compromis entre la recherche rapide de valeurs prédéterminées prédéterminées (si l'argument est indexé) et la lisibilité de valeurs arbitraires (ce qui exige que les arguments ne soient pas indexés). que les arguments ne soient pas indexés). Les développeurs peuvent surmonter ce compromis et atteindre à la fois recherche efficace et la lisibilité arbitraire en définissant des événements avec deux arguments - un indexés, l'autre non - destinés à contenir la même valeur.

3.22.11 Erreurs

En cas d'échec à l'intérieur d'un contrat, celui-ci peut utiliser un opcode spécial pour interrompre l'exécution et annuler tous les changements d'état. tous les changements d'état. En plus de ces effets, des données descriptives peuvent être retournées à l'appelant. Ces données descriptives sont le codage d'une erreur et de ses arguments de la même manière que les données d'un appel de fonction. d'une fonction.

A titre d'exemple, considérons le contrat suivant dont la fonction `transfer` se retourne toujours se retourne avec une erreur personnalisée de « solde insuffisant » :

```
// 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 {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        revert InsufficientBalance(0, amount);
    }
}
```

Les données de retour seraient codées de la même manière que l'appel de fonction `InsufficientBalance(0, amount)` à la fonction `InsufficientBalance(uint256,uint256)`, c'est-à-dire `0xcf479181`, `uint256(0)`, `uint256(montant)`.

Les sélecteurs d'erreur `0x00000000` et `0xffffff` sont réservés pour une utilisation future.

Avertissement : Ne faites jamais confiance aux données d'erreur. Par défaut, les données d'erreur remontent à travers la chaîne d'appels externes, ce qui signifie que ce qui signifie qu'un contrat peut recevoir une erreur qui n'est définie dans aucun des contrats qu'il appelle directement. De plus, tout contrat peut simuler n'importe quelle erreur en renvoyant des données qui correspondent à une signature d'erreur, même si l'erreur n'est définie nulle part.

3.22.12 JSON

Le format JSON de l'interface d'un contrat est donné par un tableau de descriptions de fonctions, d'événements et d'erreurs. Une description de fonction est un objet JSON avec les champs :

- `type` : `fonction"`, `constructeur"`, `receive"` (la fonction *« receive Ether »*) ou `"fallback"` (la fonction *« default »*) ;
- `name` : le nom de la fonction ;
- `inputs` : un tableau d'objets, chacun d'entre eux contenant :
  - `name` : le nom du paramètre.
  - `type` : le type canonique du paramètre (plus bas).
  - `components` : utilisé pour les types de tuple (plus bas).
- `outputs` : un tableau d'objets similaires aux ``inputs`".
- `stateMutability` : une chaîne avec l'une des valeurs suivantes : `pure` (spécifié pour ne pas lire l" état de la blockchain), `view` (*spécifié pour ne pas modifier l'état de la blockchain state*), *nonpayable`* (la fonction n'accepte pas les Ether - la valeur par défaut) et `payable` (la fonction accepte les Ether).

Le constructeur et la fonction de repli n'ont jamais de `name` ou de ``outputs`". La fonction de repli n'a pas non plus de ``inputs`".

Note : Envoyer un Ether non nul à une fonction non payante inversera la transaction.

Note : L'état de mutabilité « non-payable » est reflété dans Solidity en ne spécifiant pas de modificateur d'état du tout. un modificateur d'état mutable.

Une description d'événement est un objet JSON avec des champs assez similaires :

- `type` : toujours « événement ».
- `name` : le nom de l'événement.
- `inputs` : un tableau d'objets, chacun d'entre eux contenant :
  - `name` : le nom du paramètre.
  - `type` : le type canonique du paramètre (plus bas).
  - `components` : utilisé pour les types de tuple (plus bas).
  - `indexed` : `true` si le champ fait partie des sujets du journal, `false` s'il fait partie du segment de données du journal.

— `anonymous` : `true` si l'événement a été déclaré comme ``anonymous`"".

Les erreurs se présentent comme suit :

— `type` : toujours `"erreur"`.
— `name` : le nom de l'erreur.
— `inputs` : un tableau d'objets, chacun d'entre eux contenant :
  — `name` : le nom du paramètre.
  — `type` : le type canonique du paramètre (plus bas).
  — `components` : utilisé pour les types de tuple (plus bas).

Note : Il peut y avoir plusieurs erreurs avec le même nom et même avec une signature identique signature identique dans le tableau JSON, par exemple si les erreurs proviennent de différents fichiers différents dans le contrat intelligent ou sont référencées à partir d'un autre contrat intelligent. Pour l'ABI, seul le nom de l'erreur elle-même est pertinent et non l'endroit où elle est définie.

Par exemple,

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


contract Test {
    constructor() { b = hex"1234567890123456789012345678901234567890123456789012"; }
    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;
}
```

donnerait le 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

Bien que les noms ne fassent intentionnellement pas partie de l'encodage ABI, il est tout à fait logique de les inclure dans le JSON pour pouvoir l'afficher à l'utilisateur final. La structure est imbriquée de la manière suivante :

Un objet avec des membres `name`, `type`` et potentiellement ``components`` décrit une variable typée. Le type canonique est déterminé jusqu'à ce qu'un type de tuple soit atteint et la description de la chaîne de caractères jusqu'à ce point est stockée dans ``l'objet''. jusqu'à ce point est stockée dans le préfixe ``type` avec le mot `tuple`, c'est-à-dire que ce sera `tuple` suivi par une séquence de `[]` et de `[k]` avec des entiers `k`. Les composants du tuple sont ensuite stockés dans le membre `components`, qui est de type tableau et a la même structure que l'objet de niveau supérieur, sauf que `indexed` n'y est pas autorisé.

A titre d'exemple, le 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) {}
}
```

donnerait le 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",
```

(suite sur la page suivante)

(suite de la page précédente)

```
                "type": "uint256"
              }
            ]
          }
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

3.22.13 Mode de codage strict

Le mode d'encodage strict est le mode qui conduit exactement au même encodage que celui défini dans la spécification formelle ci-dessus. Cela signifie que les décalages doivent être aussi petits que possible tout en ne créant pas de chevauchements dans les zones de données. autorisés.

Habituellement, les décodeurs ABI sont écrits de manière simple en suivant simplement les pointeurs de décalage, mais certains décodeurs peuvent appliquer un mode strict. Le décodeur Solidity ABI n'applique pas actuellement le mode strict, mais l'encodeur crée toujours des données en mode strict. crée toujours les données en mode strict.

3.22.14 Mode Packed non standard

Grâce à `abi.encodePacked()`, Solidity prend en charge un mode packed non standard dans lequel :

- les types plus courts que 32 octets sont concaténés directement, sans remplissage ni extension de signe
- les types dynamiques sont encodés in-place et sans la longueur.
- les éléments de tableaux sont rembourrés, mais toujours encodés in-place.

De plus, les structs ainsi que les tableaux imbriqués ne sont pas supportés.

A titre d'exemple, l'encodage de `int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world !")` donne le résultat suivant :

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                                int16(-1)
      ^^                              bytes1(0x42)
        ^^^^                          uint16(0x03)
            ^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") sans champ de longueur
```

Plus précisément :

- Pendant l'encodage, tout est encodé sur place. Cela signifie qu'il n'y a pas de distinction entre la tête et la queue, comme dans l'encodage ABI, et la longueur d'un tableau n'est pas encodée.
- Les arguments directs de `abi.encodePacked` sont encodés sans padding, tant qu'ils ne sont pas des tableaux (ou des `string` ou des `bytes`).
- L'encodage d'un tableau est la concaténation de l'encodage de ses éléments **avec*****. codage de ses éléments **avec** remplissage.
- Les types de taille dynamique comme `string`, `bytes` ou `uint[]` sont encodés sans leur champ de longueur.
- L'encodage de `string` ou `bytes` n'applique pas de remplissage à la fin sauf s'il s'agit d'une partie d'un tableau ou d'une structure (dans ce cas, il s'agit d'un multiple de 32 octets). 32 octets).

En général, l'encodage est ambigu dès qu'il y a deux éléments de taille dynamique, à cause du champ de longueur manquant.

Si le remplissage est nécessaire, des conversions de type explicites peuvent être utilisées : `abi.encodePacked(uint16(0x12)) == hex "0012"`.

Puisque le codage empaqueté n'est pas utilisé lors de l'appel de fonctions, il n'y a pas de prise en charge particulière pour faire précéder un sélecteur de fonction. Comme l'encodage est ambigu, il n'y a pas de fonction de décodage.

Avertissement : Si vous utilisez `keccak256(abi.encodePacked(a, b))` et que `a` et `b` sont tous deux des types dynamiques, il est facile de créer des collisions dans la valeur de hachage en déplaçant des parties de `a` dans `b` et et vice-versa. Plus précisément, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. Si vous utilisez `abi.encodePacked` pour des signatures, l'authentification ou l'intégrité de données d'utiliser toujours les mêmes types et de vérifier qu'au plus l'un d'entre eux est dynamique. À moins qu'il n'y ait une raison impérative, `abi.encode` devrait être préféré.

3.22.15 Codage des paramètres d'événements indexés

Les paramètres d'événements indexés qui ne sont pas des types de valeur, c'est-à-dire les tableaux et les stockés directement, mais un keccak256-hash d'un encodage est stocké. Ce codage est défini comme suit :

- l'encodage d'une valeur de type `bytes` et ``chaîne`"" est juste le contenu de la chaîne de caractères sans aucun padding ou préfixe de longueur.
- l'encodage d'une structure est la concaténation de l'encodage de ses membres, toujours complétés par un multiple de 32 octets (même `bytes` et `string`).
- Le codage d'un tableau (de taille dynamique ou statique) est le suivant concaténation des encodages de ses éléments, toujours complétés par un multiple de 32 de 32 octets (même `bytes` et `string`) et sans préfixe de longueur.

Dans l'exemple ci-dessus, comme d'habitude, un nombre négatif est paddé par extension de signe et non paddé à zéro. Les types `bytesNN` sont paddés à droite tandis que les types `uintNN` / `intNN` sont paddés à gauche.

Avertissement : Le codage d'une structure est ambigu s'il contient plus d'un tableau de taille dynamique. dynamique. Pour cette raison, vérifiez toujours à nouveau les données de l'événement et ne vous fiez pas au résultat de la recherche basé uniquement sur les paramètres indexés.

3.23 Solidity v0.5.0 Changements de rupture

Cette section met en évidence les principaux changements introduits dans la version 0.5.0 de Solidity, ainsi que les raisons de ces changements et la façon de mettre à jour le code concerné. Pour la liste complète, consultez le journal des modifications de la version.

Note : Les contrats compilés avec Solidity v0.5.0 peuvent toujours s'interfacer avec des contrats et même des bibliothèques compilés avec des versions plus anciennes sans avoir à les recompiler ou à les redéployer. Il suffit de modifier les interfaces pour inclure les emplacements des données et les spécificateurs de visibilité et de mutabilité. Voir la section *Interopérabilité avec les contrats plus anciens* en dessous.

3.23.1 Changements uniquement sémantiques

Cette section énumère les changements qui sont uniquement sémantiques, donc potentiellement cacher un comportement nouveau et différent dans le code existant.

— Le décalage signé vers la droite utilise maintenant le décalage arithmétique
approprié, c'est-à-dire qu'il arrondit vers l'infini négatif au lieu d'arrondir vers zéro. l'infini négatif, au lieu d'arrondir vers zéro. Les décalages signés et non signés auront des opcodes dédiés dans Constantinople, et sont émulés par Solidity pour le moment. Solidity pour le moment.

— La déclaration `continue` dans une boucle `do...while` saute maintenant au comportement commun dans de tels cas. Auparavant, il sautait vers le corps de la boucle. Ainsi, si la condition est fausse, la boucle se termine.
— Les fonctions `.call()`, `.delegatecall()` et `.staticcall()` ne tamponnent plus lorsqu'on leur donne un seul paramètre `bytes`.
— Les fonctions Pure et View sont désormais appelées en utilisant l'opcode `STATICCALL` au lieu de `CALL` si la version de l'EVM est Byzantium ou ultérieure. Cela interdit les changements d'état au niveau de l'EVM.
— L'encodeur ABI pallie désormais correctement les tableaux d'octets et les chaînes de caractères des données d'appel (`msg.data` et paramètres de fonctions externes) lorsqu'ils sont utilisés dans des appels externes et dans `abi.encode`. Pour un encodage non codé, utilisez `abi.encodePacked`.
— Le décodeur ABI revient en arrière au début des fonctions et dans `abi.decode()` si les données d'appel passées sont trop courtes ou pointent hors des limites. Notez que les bits d'ordre supérieur sales sont toujours simplement ignorés.
— Transférer tout le gaz disponible avec des appels de fonctions externes à partir de Tangerine Whistle.

3.23.2 Changements sémantiques et syntaxiques

Cette section met en évidence les changements qui affectent la syntaxe et la sémantique.

— Les fonctions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` et `ripemd160()` n'acceptent plus qu'un seul argument `bytes`. unique, `bytes`. De plus, l'argument n'est pas paddé. Ceci a été changé pour rendre plus explicite et clair la façon dont les arguments sont concaténés. Changez chaque `.call()` (et famille) en un `.call("")` et chaque `.call(signature, a, b, c)` en utilisant `.call(abi.encodeWithSignature(signature, a, b, c))` (le dernier ne fonctionne que pour les types dernière ne fonctionne que pour les types de valeurs). Changez chaque `keccak256(a, b, c)` en `keccak256(abi.encodePacked(a, b, c))`. Même s'il ne s'agit pas d'une il est suggéré que les développeurs changent `x.call(bytes4(keccak256("f(uint256)")), a, b)` en `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.
— Les fonctions `.call()`, `.delegatecall()` et `.staticcall()` retournent maintenant `(bool, bytes memory)` pour donner accès aux données de retour. Modifier `bool success = otherContract.call("f")` en `(bool success, bytes memory données) = otherContract.call("f")`.
— Solidity met désormais en œuvre les règles de délimitation du style C99 pour les locales de fonctions, c'est-à-dire que les variables ne peuvent être utilisées que déclarées et seulement dans le même périmètre ou dans des

périmètres imbriqués. Les variables déclarées dans le bloc d'initialisation d'une boucle ``for`"" sont valides en tout point de la boucle. boucle.

3.23.3 Exigences d'explicitation

Cette section liste les modifications pour lesquelles le code doit être plus explicite. Pour la plupart des sujets, le compilateur fournira des suggestions.

- La visibilité explicite des fonctions est maintenant obligatoire. Ajouter `public` à chaque fonction et constructeur fonction et constructeur, et `external` à chaque fonction de fallback ou d'interface d'interface qui ne spécifie pas déjà sa visibilité.
- La localisation explicite des données pour toutes les variables de type struct, array ou mapping est maintenant obligatoire. Ceci s'applique également aux paramètres des fonctions et aux de retour. Par exemple, changez `uint[] x = m_x` en `uint[] storage x = m_x`, et `fonction f(uint[][] x)` en `fonction f(uint[][] mémoire x)` où « memory » est l'emplacement des données et peut être remplacé par « storage » ou « calldata ». `calldata` en conséquence. Notez que les fonctions `externes` requièrent des paramètres dont l'emplacement des données est `calldata`.
- Les types de contrats n'incluent plus les membres `addresses` afin de afin de séparer les espaces de noms. Par conséquent, il est maintenant nécessaire de convertir explicitement les valeurs du type de contrat en adresses avant d'utiliser une membre `address`. Exemple : si `c` est un contrat, changez `c.transfert(...)` en `adresse(c).transfert(...)`, et `c.balance` en `address(c).balance`.
- Les conversions explicites entre des types de contrats non liés sont désormais interdites. Vous pouvez seulement convertir un type de contrat en l'un de ses types de base ou ancêtres. Si vous êtes sûr que un contrat est compatible avec le type de contrat vers lequel vous voulez le convertir, bien qu'il n'en hérite pas. bien qu'il n'en hérite pas, vous pouvez contourner ce problème en convertissant d'abord en `adresse`. Exemple : si `A` et `B` sont des types de contrat, `B` n'hérite pas de `A` et `b` est un contrat de type `B`, vous pouvez toujours convertir `b` en type `A` en utilisant `A(adresse(b))`. Notez que vous devez toujours faire attention aux fonctions de repli payantes correspondantes, comme expliqué ci-dessous.
- Le type « adresse » a été divisé en « adresse » et « adresse payable », où seule « l'adresse payable » fournit la fonction « transfert ». Un site Une « adresse payable » peut être directement convertie en une « adresse », mais l'inverse n'est pas autorisé. l'inverse n'est pas autorisé. La conversion de `adresse` en `adresse payable" est possible par conversion via ``uint160`. Si `c` est un contrat, `address(c)` résulte en `address payable` seulement si `c` possède une fonction de repli payable. Si vous utilisez le modèle *withdraw pattern*, vous n'avez probablement pas à modifier votre code car `transfer` est uniquement utilisé sur `msg.sender` au lieu des adresses stockées et `msg.sender` est une `adresse`. est une `adresse payable`.
- Les conversions entre `bytesX` et `uintY` de taille différente sont maintenant sont désormais interdites en raison du remplissage de `bytesX` à droite et du remplissage de `uintY` à gauche. gauche, ce qui peut entraîner des résultats de conversion inattendus. La taille doit maintenant être ajustée dans le type avant la conversion. Par exemple, vous pouvez convertir un `bytes4` (4 octets) en un `uint64` (8 octets) en convertissant d'abord le `bytes4` en un `uint64`'. en convertissant d'abord la variable ``bytes4` en `bytes8`, puis en `uint64`'. Vous obtenez le inverse en convertissant en ``uint32`. Avant la version 0.5.0, toute conversion entre `bytesX` et `uintY` passait par `uint8X`. Pour Par exemple, `uint8(bytes3(0x291807))` sera converti en `uint8(uint24(bytes3(0x291807))` (le résultat est (le résultat est `0x07`).
- L'utilisation de `msg.value` dans des fonctions non payantes (ou son introduction par le biais d'un modificateur) est interdit par mesure de sécurité. Transformez la fonction en payante » ou créez une nouvelle fonction interne pour la logique du programme qui utilise `msg.value`.
- Pour des raisons de clarté, l'interface de la ligne de commande exige maintenant - si l" l'entrée standard est utilisée comme source.

3.23.4 Éléments dépréciés

Cette section liste les changements qui déprécient des fonctionnalités ou des syntaxes antérieures. Notez que plusieurs de ces changements étaient déjà activés dans le mode expérimental `v0.5.0`.

Interfaces en ligne de commande et JSON

- L'option de ligne de commande `--formal` (utilisée pour générer la sortie de Why3 pour une pour une vérification formelle plus poussée) était dépréciée et est maintenant supprimée. Un nouveau module de vérification formelle, le SMTChecker, est activé via `pragma experimental SMTChecker;`.
- L'option de ligne de commande `--julia` a été renommée en `--yul` en raison du changement de nom du langage intermédiaire ``. en raison du changement de nom du langage intermédiaire « Julia » en « Yul ».
- Les options de ligne de commande `--clone-bin` et `--combined-json clone-bin` ont été supprimées. ont été supprimées.
- Les remappages avec un préfixe vide ne sont pas autorisés.
- Les champs AST JSON `constant` et ``payable`" ont été supprimés. L'adresse informations sont maintenant présentes dans le champ ``stateMutability`".
- Le champ JSON AST `isConstructor` du noeud `FunctionDefinition` a été remplacé par un champ appelé `Fonctions''. a été remplacé par un champ appelé ``kind` qui peut avoir la valeur valeur `"constructor"`, `"fallback"` ou `"function"`.
- Dans les fichiers hexadécimaux binaires non liés, les adresses des bibliothèques sont maintenant les 36 premiers caractères hexadécimaux de la clé. sont désormais les 36 premiers caractères hexadécimaux du hachage keccak256 du nom de bibliothèque nom de bibliothèque entièrement qualifié, entouré de « $…$ ». Auparavant, seul le nom complet de la bibliothèque était utilisé. Cela réduit les risques de collisions, en particulier lorsque de longs chemins sont utilisés. Les fichiers binaires contiennent maintenant aussi une liste de correspondances entre ces caractères de remplacement vers les noms pleinement qualifiés.

Constructeurs

- Les constructeurs doivent désormais être définis à l'aide du mot clé « constructeur ».
- L'appel de constructeurs de base sans parenthèses est désormais interdit.
- La spécification des arguments des constructeurs de base plusieurs fois dans la même même hiérarchie d'héritage est maintenant interdit.
- L'appel d'un constructeur avec des arguments mais avec un nombre d'arguments incorrect est maintenant désapprouvé. Si vous souhaitez seulement spécifier une relation d'héritage sans sans donner d'arguments, ne fournissez pas de parenthèses du tout.

Fonctions

- La fonction `callcode` est maintenant désapprouvée (en faveur de `delegatecall`). Il est Il est toujours possible de l'utiliser via l'assemblage en ligne.
- La fonction `suicide` n'est plus autorisée (au profit de `selfdestruct`).
- `sha3` n'est plus autorisé (au profit de `keccak256`).
- `throw` est maintenant désapprouvé (en faveur de `revert`, `require` et de `assert`).

Conversions

— Les conversions explicites et implicites des littéraux décimaux en types ``bytesXX`"" sont maintenant désactivées. est désormais interdit.
— Les conversions explicites et implicites de littéraux hexadécimaux en types ``bytesXX`"" de taille différente sont désormais interdites. de taille différente sont désormais interdites.

Littéraux et suffixes

— L'unité de dénomination « années » n'est plus autorisée en raison de complications et de confusions concernant les années bissextiles. complications et de confusions concernant les années bissextiles.
— Les points de fin de ligne qui ne sont pas suivis d'un nombre ne sont plus autorisés.
— La combinaison de nombres hexadécimaux avec des unités (par exemple, « 0x1e wei ») n'est plus autorisée. interdites.
— Le préfixe `0X` pour les nombres hexadécimaux n'est plus autorisé, seul `0x` est possible.

Variables

— La déclaration de structures vides n'est plus autorisée pour des raisons de clarté.
— Le mot clé « var » n'est plus autorisé pour favoriser l'explicitation.
— Les affectations entre les tuples avec un nombre différent de composants sont maintenant interdites. désapprouvé.
— Les valeurs des constantes qui ne sont pas des constantes de compilation ne sont pas autorisées.
— Les déclarations multi-variables avec un nombre de valeurs non concordant sont maintenant désapprouvées.
— Les variables de stockage non initialisées ne sont plus autorisées.
— Les composants de tuple vides ne sont plus admis.
— La détection des dépendances cycliques dans les variables et les structures est limitée en récursion à 256. récursion à 256.
— Les tableaux de taille fixe avec une longueur de zéro ne sont plus autorisés.

Syntaxe

— L'utilisation de `constant` comme modificateur de mutabilité de l'état de la fonction est désormais interdite.
— Les expressions booléennes ne peuvent pas utiliser d'opérations arithmétiques.
— L'opérateur unaire « + » n'est plus autorisé.
— Les littéraux ne peuvent plus être utilisés avec `abi.encodePacked` sans conversion conversion préalable vers un type explicite.
— Les déclarations de retour vides pour les fonctions avec une ou plusieurs valeurs de retour ne sont plus sont désormais interdites.
— La syntaxe « loose assembly », c'est-à-dire les étiquettes de saut, est maintenant totalement interdite, les sauts et les instructions non fonctionnelles ne peuvent plus être utilisés. Utilisez les nouvelles fonctions `while`, `switch` et `if` à la place.
— Les fonctions sans implémentation ne peuvent plus utiliser de modificateurs.
— Les types de fonctions avec des valeurs de retour nommées ne sont plus autorisés.
— Les déclarations de variables d'une seule déclaration à l'intérieur de corps if/while/for qui ne sont pas qui ne sont pas des blocs ne sont plus autorisées.
— Nouveaux mots-clés : `calldata` et `constructor`.
— Nouveaux mots-clés réservés : `alias`, `apply`, `auto`, `copyof`, `définir''`, ``immutable'', ``implements'', ``macro'', ``mutable'', ``override`, `partiel`, `promise`, `reference`, `sealed`, ``sizeof"", ``supports"", ``typedef"" et ``unchecked"".

3.23.5 Interopérabilité avec les anciens contrats

Il est toujours possible de s'interfacer avec des contrats écrits pour des versions de Solidity antérieures à la v0.5.0 (ou l'inverse) en définissant des interfaces pour eux. Considérons que vous avez le contrat suivant, antérieur à la version 0.5.0, déjà déployé :

```
// 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) {
        //...
    }
    // ...
}
```

Il ne compilera plus avec Solidity v0.5.0. Cependant, vous pouvez lui définir une interface compatible :

```
// 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);
}
```

Notez que nous n'avons pas déclaré « anotherOldFunction » comme étant « view », bien qu'elle soit déclarée « constante » dans le contrat original. contrat original. Cela est dû au fait qu'à partir de la version 0.5.0 de Solidity, l'option `staticcall` est utilisée pour appeler les fonctions `view`. Avant la v0.5.0, le mot-clé `constant` n'était pas appliqué, donc appeler une fonction déclarée `constante` avec `staticcall` peut encore se retourner, puisque la fonction `constant` peut encore tenter de modifier le stockage. Par conséquent, lorsque vous définissez une pour des contrats plus anciens, vous ne devriez utiliser `view` à la place de `constant` que si vous êtes absolument sûr que la fonction fonctionnera avec `staticcall`.

Avec l'interface définie ci-dessus, vous pouvez maintenant facilement utiliser le contrat pré-0.5.0 déjà déployé :

```
// 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();
    }
}
```

De même, les bibliothèques pré-0.5.0 peuvent être utilisées en définissant les fonctions de la bibliothèque sans implé-

mentation et en en fournissant l'adresse de la bibliothèque pré-0.5.0 lors de l'édition de liens (voir :ref :``commandline-compiler` pour savoir comment utiliser le pour savoir comment utiliser le compilateur en ligne de commande pour l'édition de liens) :

```
// 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);
    }
}
```

3.23.6 Exemple

L'exemple suivant montre un contrat et sa version mise à jour pour Solidity v0.5.0 avec certaines des modifications énumérées dans cette section.

Ancienne 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        bytes memory b = new bytes(x);
        y = -3 >> 1;
        // y == -1 (wrong, should be -2)
        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;
    }
}
```

Nouvelle 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; }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
        // 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 address_make_payable 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.make_payable();
        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
```

(suite sur la page suivante)

(suite de la page précédente)

```
        myNumber += y + msg.value;
    }
}

// We can define a library for explicitly converting ``address``
// to ``address payable`` as a workaround.
library address_make_payable {
    function make_payable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

3.24 Solidity v0.6.0 Changements de rupture

Cette section met en évidence les principaux changements de rupture introduits dans Solidity version 0.6.0, ainsi que le raisonnement derrière ces changements et la façon de mettre à jour code affecté. Pour la liste complète, consultez le changelog de la version.

3.24.1 Changements dont le compilateur pourrait ne pas être averti

Cette section liste les changements pour lesquels le comportement de votre code pourrait changer sans que le compilateur vous en avertisse.

— Le type résultant d'une exponentiation est le type de la base. Il s'agissait auparavant du plus petit type qui peut contenir à la fois le type de la base et le type de l'exposant, comme pour les opérations symétriques. symétriques. De plus, les types signés sont autorisés pour la base de l'exponentiation.

3.24.2 Exigences d'explicitation

Cette section liste les changements pour lesquels le code doit être plus explicite, mais la sémantique ne change pas. Pour la plupart des sujets, le compilateur fournira des suggestions.

— Les fonctions ne peuvent maintenant être surchargées que lorsqu'elles sont marquées avec la clef `virtual` ou définies dans une interface. Les fonctions sans Les fonctions sans implémentation en dehors d'une interface doivent être marquées `virtual`. Lorsqu'on surcharge une fonction ou un modificateur, le nouveau mot-clé `override` doit être utilisé. doit être utilisé. Lorsqu'on remplace une fonction ou un modificateur défini dans plusieurs bases parallèles bases parallèles, toutes les bases doivent être listées entre parenthèses après le mot-clé comme ceci : `override(Base1, Base2)`.

— L'accès des membres à `length` des tableaux est maintenant toujours en lecture seule, même pour les tableaux de stockage. Il n'est plus possible de plus possible de redimensionner des tableaux de stockage en assignant une nouvelle valeur à leur longueur. Utilisez `push()`, `push(value)` ou `pop()` à la place, ou assignez un tableau complet, qui écrasera bien sûr le contenu existant. La raison derrière cela est d'éviter les collisions de stockage de gigantesques de stockage gigantesques.

— Le nouveau mot-clé `abstract` peut être utilisé pour marquer les contrats comme étant abstraits. Il doit être utilisé si un contrat n'implémente pas toutes ses fonctions. Les contrats abstraits ne peuvent pas être créés en utilisant l'opérateur `new`, et il n'est pas possible de générer du bytecode pour eux pendant la compilation.

— Les bibliothèques doivent implémenter toutes leurs fonctions, pas seulement les fonctions internes.

— Les noms des variables déclarées en inline assembly ne peuvent plus se terminer par `_slot` ou `_offset`.

— Les déclarations de variables dans l'assemblage en ligne ne peuvent plus suivre une déclaration en dehors du bloc d'assemblage en ligne. Si le nom contient un point, son préfixe jusqu'au point ne doit pas entrer en conflit avec une déclaration en dehors du bloc d'assemblage en ligne. d'assemblage.

— Le shadowing de variables d'état est désormais interdit. Un contrat dérivé peut seulement déclarer une variable d'état `x`, que s'il n'y a pas de variable d'état visible avec le même nom d'état visible portant le même nom dans l'une de ses bases.

3.24.3 Changements sémantiques et syntaxiques

Cette section liste les changements pour lesquels vous devez modifier votre code et il fait quelque chose d'autre après.

— Les conversions de types de fonctions externes en `adresse` sont maintenant interdites. A la place, les types de fonctions externes Au lieu de cela, les types de fonctions externes ont un membre appelé `address`, similaire au membre `selector` existant.
— La fonction `push(value)` pour les tableaux de stockage dynamique ne retourne plus la nouvelle longueur (elle ne retourne rien).
— La fonction sans nom communément appelée « fonction de repli » a été divisée en une nouvelle fonction de repli définie à l'aide de la fonction de repli. nouvelle fonction de repli définie à l'aide du mot-clé `fallback` et une fonction de réception d'éther définie à l'aide du mot-clé `receive`.
  — Si elle est présente, la fonction de réception de l'éther est appelée chaque fois que les données d'appel sont vides (que l'éther soit reçu ou non). (que l'éther soit reçu ou non). Cette fonction est implicitement `payable`.
  — La nouvelle fonction de repli est appelée lorsqu'aucune autre fonction ne correspond (si la fonction receive ether n'existe pas, cela inclut les appels avec des données d'appel vides). Vous pouvez rendre cette fonction `payable` ou non. Si elle n'est pas « payante », alors les transactions ne correspondant à aucune autre fonction qui envoie une valeur seront inversées. Vous n'aurez besoin d'implémenter implémenter la nouvelle fonction de repli que si vous suivez un modèle de mise à niveau ou de proxy.

3.24.4 Nouvelles fonctionnalités

Cette section énumère des choses qui n'étaient pas possibles avant la version 0.6.0 de Solidity ou qui étaient plus difficiles à réaliser.

— L'instruction *try/catch* vous permet de réagir à l'échec d'appels externes.
— Les types `struct` et `enum` peuvent être déclarés au niveau du fichier.
— Les tranches de tableau peuvent être utilisées pour les tableaux de données d'appel, par exemple « abi.decode(msg.data[4 :], (uint, uint))``. est un moyen de bas niveau pour décoder les données utiles de l'appel de fonction.
— Natspec prend en charge les paramètres de retour multiples dans la documentation du développeur, en appliquant le même contrôle de nommage que `@param`.
— Yul et Inline Assembly ont une nouvelle instruction appelée `leave` qui quitte la fonction courante.
— Les conversions de `adresse` en `adresse payable` sont maintenant possibles via `payable(x)`, où `x` doit être de type `adresse`.

3.24.5 Changements d'interface

Cette section liste les changements qui ne sont pas liés au langage lui-même, mais qui ont un effet sur les interfaces du compilateur. le compilateur. Ces modifications peuvent changer la façon dont vous utilisez le compilateur sur la ligne de commande, la façon dont vous utilisez son interface programmable, ou la façon dont vous analysez la sortie qu'il produit. ou comment vous analysez la sortie qu'il produit.

Nouveau rapporteur d'erreurs

Un nouveau rapporteur d'erreur a été introduit, qui vise à produire des messages d'erreur plus accessibles sur la ligne de commande. Il est activé par défaut, mais si vous passez `--old-reporter`, vous revenez à l'ancien rapporteur d'erreurs, qui est déprécié.

Options de hachage des métadonnées

Le compilateur ajoute maintenant le hash IPFS du fichier de métadonnées à la fin du bytecode par défaut. (pour plus de détails, voir la documentation sur *contract metadata*). Avant la version 0.6.0, le compilateur ajoutait la balise Swarm hash par défaut, et afin de toujours supporter ce comportement, la nouvelle option de ligne de commande `--metadata-hash` a été introduite. Elle permet de sélectionner le hachage à produire et à ajouter ajouté, en passant soit `ipfs` soit `swarm` comme valeur à l'option de ligne de commande `--metadata-hash`. Passer la valeur `none` supprime complètement le hachage.

Ces changements peuvent également être utilisés via l'interface *Standard JSON Interface* et affecter les métadonnées JSON générées par le compilateur.

La façon recommandée de lire les métadonnées est de lire les deux derniers octets pour déterminer la longueur de l'encodage CBOR et d'effectuer un décodage correct sur ce bloc de données, comme expliqué dans la section *metadata*.

Optimiseur de Yul

Avec l'optimiseur de bytecode hérité, l'optimiseur *Yul* est maintenant activé par défaut lorsque vous appelez le compilateur avec *–optimize*. avec `--optimize`. Il peut être désactivé en appelant le compilateur avec *–no-optimize-yul* `. Ceci affecte principalement le code qui utilise ABI coder v2.

Modifications de l'API C

Le code client qui utilise l'API C de `libsolc` a maintenant le contrôle de la mémoire utilisée par le compilateur. Pour rendre Pour rendre ce changement cohérent, `solidity_free` a été renommé en `solidity_reset`, les fonctions `solidity_alloc` et `solidity_free` ont été modifiées. `solidity_free` ont été ajoutées et `solidity_compile` retourne maintenant une chaîne de caractères qui doit être explicitement libérée par la fonction `solidity_free()`.

3.24.6 Comment mettre à jour votre code

Cette section donne des instructions détaillées sur la façon de mettre à jour le code antérieur pour chaque changement de rupture.

- Changez `address(f)` en `f.address` pour que `f` soit de type fonction externe.
- Remplacer `fonction () externe [payable] { ... }` par soit `receive() externe [payable] { ... }`, `fallback() externe [payable] { ... }` ou les deux. Préférez l'utilisation d'une fonction `receive` uniquement, lorsque cela est possible.
- Remplacez `uint length = array.push(value)` par `array.push(value);`. La nouvelle longueur peut être accessible via `array.length`.
- Changez `array.length++` en `array.push()` pour augmenter, et utilisez `pop()` pour diminuer la longueur d'un tableau de stockage.
- Pour chaque paramètre de retour nommé dans la documentation `@dev` d'une fonction, définissez une entrée `@return` contenant le nom du paramètre. qui contient le nom du paramètre comme premier mot. Par exemple, si vous avez une fonction « f()`` définie comme suit comme « fonction f() public returns (uint value)`` et une annotation *@dev* `, documentez ses paramètres de retour comme suit de retour comme suit : `@return value La valeur de retour.`. Vous pouvez mélanger des paramètres de retour nommés et non nommés documentation tant que les annotations sont dans l'ordre où elles apparaissent dans le type de retour du tuple.

— Choisissez des identifiants uniques pour les déclarations de variables dans l'assemblage en ligne qui n'entrent pas en conflit avec les déclarations en dehors de l'assemblage en ligne. avec des déclarations en dehors du bloc d'assemblage en ligne.
— Ajoutez « virtual » à chaque fonction non interface que vous avez l'intention de remplacer. Ajoutez `virtual` à toutes les fonctions sans implémentation en dehors des interfaces. à toutes les fonctions sans implémentation en dehors des interfaces. Pour l'héritage simple, ajoutez `override` à chaque fonction de remplacement. Pour l'héritage multiple, ajoutez `override(A, B, ..)`, où vous listez entre parenthèses tous les contrats qui définissent la fonction surchargée. Lorsque plusieurs bases définissent la même fonction, le contrat qui hérite doit remplacer toutes les fonctions conflictuelles.

3.25 Solidity v0.7.0 Changements de dernière minute

Cette section met en évidence les principaux changements de rupture introduits dans Solidity version 0.7.0, ainsi que le raisonnement derrière ces changements et la façon de mettre à jour code affecté. Pour la liste complète, consultez le changelog de la version <https://github.com/ethereum/solidity/releases/tag/v0.7.0>`_.

3.25.1 Changements silencieux de la sémantique

— L'exponentiation et les décalages de littéraux par des non-littéraux (par exemple, `1 << x` ou `2 ** x`) utiliseront toujours soit le type `uint256` (pour les littéraux non négatifs), soit le type `int256` (pour les littéraux négatifs) pour effectuer l'opération. Auparavant, l'opération était effectuée dans le type de la quantité de décalage / l'exposant, ce qui peut être trompeur. exposant, ce qui peut être trompeur.

3.25.2 Modifications de la syntaxe

— Dans les appels de fonctions externes et de création de contrats, l'éther et le gaz sont maintenant spécifiés en utilisant une nouvelle syntaxe : `x.f{gaz : 10000, valeur : 2 éther}(arg1, arg2)`. L'ancienne syntaxe – `x.f.gas(10000).value(2 ether)(arg1, arg2)` – provoquera une erreur.
— La variable globale `now` est obsolète, `block.timestamp` devrait être utilisée à la place. L'identifiant unique `now` est trop générique pour une variable globale et pourrait donner l'impression qu'elle change pendant le traitement de la transaction, alors que `block.timestamp` reflète correctement reflète correctement le fait qu'il s'agit d'une propriété du bloc.
— Les commentaires NatSpec sur les variables ne sont autorisés que pour les variables d'état publiques et non pour les variables locales ou internes.
— Le jeton `gwei` est maintenant un mot-clé (utilisé pour spécifier, par exemple, `2 gwei` comme un nombre) et ne peut pas être utilisé comme un identifiant.
— Les chaînes de caractères ne peuvent plus contenir que des caractères ASCII imprimables, ce qui inclut une variété de séquences d'échappement, telles que les hexadécimales. séquences d'échappement, telles que les échappements hexadécimaux (`xff`) et unicode (`u20ac`).
— Les chaînes littérales Unicode sont désormais prises en charge pour accueillir les séquences UTF-8 valides. Ils sont identifiés avec le préfixe `unicode` : `unicode "Hello "`.
— Mutabilité d'état : La mutabilité d'état des fonctions peut maintenant être restreinte pendant l'héritage : Les fonctions avec une mutabilité d'état par défaut peuvent être remplacées par des fonctions `pure'' et ``view''. tandis que les fonctions ``view` peuvent être remplacées par des fonctions `pure`. En même temps, les variables d'état publiques sont considérées comme `view` et même `pure` si elles sont constantes. si elles sont des constantes.

Assemblage en ligne

— Interdire `.` dans les noms de fonctions et de variables définies par l'utilisateur dans l'assemblage en ligne. C'est toujours valable si vous utilisez Solidity en mode Yul-only.
— L'emplacement et le décalage de la variable pointeur de stockage `x` sont accessibles via `x.slot` et `x.offset`. et `x.offset` au lieu de `x_slot` et `x_offset`.

3.25.3 Suppression des fonctionnalités inutilisées ou dangereuses

Mappages en dehors du stockage

— Si une structure ou un tableau contient un mappage, il ne peut être utilisé que dans le stockage. Auparavant, les membres du mappage étaient ignorés en mémoire, ce qui est déroutant et source d'erreurs. ce qui est déroutant et source d'erreurs.
— Les affectations aux structures ou tableaux dans le stockage ne fonctionnent pas s'ils contiennent des mappings. mappings. Auparavant, les mappings étaient ignorés silencieusement pendant l'opération de copie, ce qui ce qui est trompeur et source d'erreurs.

Fonctions et événements

— La visibilité (`public / internal`') n'est plus nécessaire pour les constructeurs : Pour empêcher un contrat d'être créé, il peut être marqué ``abstract`. Cela rend le concept de visibilité pour les constructeurs obsolète.
— Contrôleur de type : Désaccorder `virtual` pour les fonctions de bibliothèque : Puisque les bibliothèques ne peuvent pas être héritées, les fonctions de bibliothèque ne devraient pas être virtuelles.
— Plusieurs événements avec le même nom et les mêmes types de paramètres dans la même hiérarchie d'héritage sont interdits. même hiérarchie d'héritage sont interdits.
— `utiliser A pour B` n'affecte que le contrat dans lequel il est mentionné. Auparavant, l'effet était hérité. Maintenant, vous devez répéter l'instruction « using » dans tous les contrats dérivés qui font usage de cette instruction. dans tous les contrats dérivés qui utilisent cette fonctionnalité.

Expressions

— Les décalages par des types signés ne sont pas autorisés. Auparavant, les décalages par des montants négatifs étaient autorisés, mais ils étaient annulés à l'exécution.
— Les dénominations `finney` et ``szabo`" sont supprimées. Elles sont rarement utilisées et ne rendent pas le montant réel facilement visible. A la place, des valeurs explicites valeurs explicites comme « 1e20 » ou le très commun « gwei » peuvent être utilisées.

Déclarations

— Le mot-clé `var` ne peut plus être utilisé. Auparavant, ce mot-clé était analysé mais donnait lieu à une erreur de type et à une suggestion sur le type à utiliser. une suggestion sur le type à utiliser. Maintenant, il résulte en une erreur d'analyse.

3.25.4 Changements d'interface

— JSON AST : Marquer les littéraux de chaînes hexagonales avec `kind : "hexString"`.
— JSON AST : Les membres avec la valeur `null` sont supprimés de la sortie JSON.
— NatSpec : Les constructeurs et les fonctions ont une sortie userdoc cohérente.

3.25.5 Comment mettre à jour votre code

Cette section donne des instructions détaillées sur la façon de mettre à jour le code antérieur pour chaque changement de rupture.

— Changez `x.f.value(...)()` en `x.f{value : ...}()`. De même, `(new C).value(...)()` en `nouveau C{valeur : ...}()` et `x.f.gas(...).valeur(...)()` en `x.f{gas : ..., valeur : ...}()`.
— Remplacez `now` par `block.timestamp`.
— Changez les types de l'opérande droit dans les opérateurs de décalage en types non signés. Par exemple, remplacez `x >> (256 - y)` par `x >> uint(256 - y)`.
— Répétez les déclarations `utilisant A pour B` dans tous les contrats dérivés si nécessaire.
— Supprimez le mot-clé « public » de chaque constructeur.
— Supprimer le mot-clé « interne » de chaque constructeur et ajouter « abstrait » au contrat (s'il n'est pas déjà présent).
— Changez les suffixes `_slot` et `_offset`` ' dans l'assemblage en ligne en ``.slot` et ``.offset`", respectivement.

3.26 Solidity v0.8.0 Changements de rupture

Cette section met en évidence les principaux changements de rupture introduits dans Solidity version 0.8.0. Pour la liste complète, consultez le changelog de la version 0.8.0.

3.26.1 Changements silencieux de la sémantique

Cette section répertorie les modifications où le code existant change de comportement sans que le compilateur vous en informe.

— Les opérations arithmétiques s'inversent en cas de sous-dépassement et de dépassement. Vous pouvez utiliser `unchecked { ... }` pour utiliser le comportement d'enveloppement précédent.
Les vérifications pour le débordement sont très communes, donc nous les avons faites par défaut pour augmenter la lisibilité du code, même si cela entraîne une légère augmentation du coût de l'essence.
— ABI coder v2 est activé par défaut.
Vous pouvez choisir d'utiliser l'ancien comportement en utilisant `pragma abicoder v1;`. Le pragma `pragma experimental ABIEncoderV2;` est toujours valide, mais il est déprécié et n'a aucun effet. Si vous voulez être explicite, veuillez utiliser le pragma `pragma abicoder v2;` à la place.
Notez que ABI coder v2 supporte plus de types que v1 et effectue plus de contrôles d'intégrité sur les entrées. ABI coder v2 rend certains appels de fonctions plus coûteux et il peut aussi faire des appels de contrats réversibles qui n'étaient pas réversibles avec ABI coder v1 lorsqu'ils contiennent des données qui ne sont pas conformes aux types de paramètres. types de paramètres.
— L'exponentiation est associative à droite, c'est-à-dire que l'expression `a**b**c` est interprétée comme `a**(b**c)`. Avant la version 0.8.0, elle était interprétée comme `(a**b)**c`.
C'est la façon courante d'analyser l'opérateur d'exponentiation.
— Les assertions qui échouent et d'autres vérifications internes comme la division par zéro ou le dépassement arithmétique n'utilisent pas l'opcode invalide mais plutôt l'opcode de retour. Plus précisément, ils utiliseront des données d'erreur égales à un appel de fonction à `Panic(uint256)` avec un code d'erreur spécifique aux circonstances. aux circonstances.

Cela permettra d'économiser du gaz sur les erreurs tout en permettant aux outils d'analyse statique de distinguer ces situations d'un retour sur invalidité. distinguer ces situations d'un retour en arrière sur une entrée invalide, comme un `require` échoué.

— Si l'on accède à un tableau d'octets en stockage dont la longueur est mal codée, une panique est provoquée. Un contrat ne peut pas se retrouver dans cette situation à moins que l'assemblage en ligne soit utilisé pour modifier la représentation brute des tableaux d'octets de stockage.

— Si des constantes sont utilisées dans les expressions de longueur de tableau, les versions précédentes de Solidity utilisaient une précision arbitraire dans toutes les branches de l'arbre d'évaluation. dans toutes les branches de l'arbre d'évaluation. Maintenant, si des variables constantes sont utilisées comme expressions intermédiaires, leurs valeurs seront correctement arrondies de la même manière que lorsqu'elles sont utilisées dans des expressions d'exécution.

— Le type `byte` a été supprimé. C'était un alias de `bytes1`.

3.26.2 Nouvelles restrictions

Cette section énumère les changements qui pourraient empêcher les contrats existants de se compiler.

— Il existe de nouvelles restrictions liées aux conversions explicites de littéraux. Le comportement précédent dans les cas suivants était probablement ambigu :

1. Les conversions explicites de littéraux négatifs et de littéraux plus grands que `type(uint160).max` en `adresse` sont interdites.
2. Les conversions explicites entre des littéraux et un type de nombre entier `T` ne sont autorisées que si le littéral se situe entre `type(T).min` et `type(T).max`. En particulier, remplacez les utilisations de `uint(-1)` par `type(uint)`. par `type(uint).max`.
3. Les conversions explicites entre les littéraux et les énumérations ne sont autorisées que si le littéral peut représenter une valeur de l'énumération.
4. Les conversions explicites entre les littéraux et le type `adresse` (par exemple `address(literal)`) ont le type `address`. type `adresse` au lieu de `adresse payable`. On peut obtenir un type d'adresse payable en utilisant une conversion explicite, c'est-à-dire `payable(literal)`.

— *Les littéraux d'adresse* ont le type `address` au lieu de `address payable`. Ils peuvent être convertis en `adresse payable` en utilisant une conversion explicite, par exemple `payable(0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF)`.

— Il y a de nouvelles restrictions sur les conversions de type explicites. La conversion n'est autorisée que lorsqu'il y a lorsqu'il y a au plus un changement de signe, de largeur ou de catégorie de type (`int`, `address`, `bytesNN`, etc.). Pour effectuer plusieurs changements, il faut utiliser plusieurs conversions.
Utilisons la notation `T(S)` pour désigner la conversion explicite `T(x)`, où, `T` et `S` sont des types, et `x` est une variable arbitraire de type `S`. Un exemple d'une telle exemple d'une telle conversion non autorisée serait `uint16(int8)` puisqu'elle change à la fois la largeur (8 bits à 16 bits) et le signe (d'entier signé à entier non signé). Pour effectuer la conversion, il faut passer par un type intermédiaire. passer par un type intermédiaire. Dans l'exemple précédent, ce serait `uint16(uint8(int8))` ou `uint16(int16(int8))`. Notez que les deux façons de convertir produiront des résultats différents, par ex, pour `-1`. Voici quelques exemples de conversions qui ne sont pas autorisées par cette règle.
  — `address(uint)` et `uint(address)` : conversion à la fois de la catégorie de type et de la largeur. Remplacez-les par `address(uint160(uint))` et `uint(uint160(address))` respectivement.
  — `payable(uint160)`, `payable(bytes20)` et `payable(integer-literal)` : conversion de la catégorie de type et de la la catégorie de type et la mutabilité d'état. Remplacez-les par `payable(address(uint160))`, `payable(address(bytes20))` et `payable(address(integer-literal))` respectivement. Notez que `payable(0)` est valide et constitue une exception à la règle.
  — `int80(bytes10)` et `bytes10(int80)` : conversion de la catégorie de type et du signe. Remplacez-les par `int80(uint80(bytes10))` et `bytes10(uint80(int80)` respectivement.
  — `Contract(uint)` : convertit à la fois la catégorie de type et le signe. Remplacez-la par `Contract(adresse(uint160(uint)))`.

Ces conversions ont été interdites pour éviter toute ambiguïté. Par exemple, dans l'expression `uint16 x = uint16(int8(-1))`, la valeur de `x` dépendrait de la conversion du signe ou de la largeur appliquée en premier lieu. a été appliquée en premier.

— Les options d'appel de fonction ne peuvent être données qu'une seule fois, c'est-à-dire que `c.f{gas : 10000}{value : 1}()` est invalide et doit être changé en `c.f{gas : 10000, value : 1}()`.

— Les fonctions globales `log0`, `log1`, `log2`, `log3` et `log4` ont été supprimées.
Ce sont des fonctions de bas niveau qui étaient largement inutilisées. Leur comportement est accessible depuis l'assemblage en ligne.

— Les définitions de `enum` ne peuvent pas contenir plus de 256 membres.
Cela permet de supposer que le type sous-jacent dans l'ABI est toujours `uint8`.

— Les déclarations portant les noms « this », « super » et « _ » ne sont pas autorisées, à l'exception des fonctions et événements publics. fonctions et événements publics. Cette exception a pour but de permettre la déclaration d'interfaces de contrats implémentées dans des langages autres que Solidity qui autorisent de tels noms de fonctions.

— Suppression de la prise en charge des séquences d'échappement `b`, `f` et `v`'' dans le code. Elles peuvent toujours être insérées par le biais d'échappements hexadécimaux, par exemple, respectivement, " ``X08`, » `X0c` et » `X0b`.

— Les variables globales `tx.origin` et `msg.sender` ont le type `address` au lieu de `adresse payable`. On peut les convertir en `adresse payable` en utilisant une conversion explicite, c'est-à-dire `payable(tx.origin)` ou `payable(msg.sender)`.
Ce changement a été fait car le compilateur ne peut pas déterminer si ces adresses sont payables ou non. sont payables ou non, donc il faut maintenant une conversion explicite pour rendre cette exigence visible.

— La conversion explicite en type `adresse` retourne toujours un type `adresse` non payable. Dans En particulier, les conversions explicites suivantes ont le type `adresse` au lieu de ``adresse payable » :
  — `adresse(u)` où `u` est une variable de type `uint160`. On peut convertir `u` dans le type `adresse payable` en utilisant deux conversions explicites, c'est-à-dire, `payable(adresse(u))`.
  — `adresse(b)` où `b` est une variable de type `bytes20`. On peut convertir `b` dans le type `adresse payable` en utilisant deux conversions explicites, c'est-à-dire, `payable(adresse(b))`.
  — `adresse(c)` où `c` est un contrat. Auparavant, le type de retour de cette conversion dépendait de la possibilité pour le contrat de recevoir de l'Ether (soit en ayant une fonction de réception ou une fonction de repli payable). La conversion `payable(c)` a le type `adresse payable" et n'est autorisée que si le contrat "c" peut recevoir de l'éther. En général, on peut convertir ``c` en type `adresse payable` en utilisant la conversion explicite suivante explicite suivante : `payable(adresse(c))`. Notez que `address(this)` tombe sous la même catégorie que `address(c)` et les mêmes règles s'appliquent pour elle.

— La construction de « chainid » dans l'assemblage en ligne est maintenant considérée comme une « vue » au lieu d'une « pure ».

— La négation unaire ne peut plus être utilisée sur les entiers non signés, seulement sur les entiers signés.

3.26.3 Changements d'interface

— La sortie de `--combined-json` a changé : Les champs JSON `abi`, `devdoc`, `userdoc` et `storage-layout` sont maintenant des sous-objets. Avant la version 0.8.0, ils étaient sérialisés sous forme de chaînes de caractères.

— L'« ancien AST » a été supprimé (`--ast-json` sur l'interface de la ligne de commande et `legacyAST` pour le JSON standard). Utilisez l'« AST compact » (`--ast-compact--json` resp. `AST`) en remplacement.

— L'ancien rapporteur d'erreurs (`--old-reporter`) a été supprimé.

3.26.4 Comment mettre à jour votre code

— Si vous comptez sur l'arithmétique enveloppante, entourez chaque opération de `unchecked { ... }`.
— Optionnel : Si vous utilisez SafeMath ou une bibliothèque similaire, changez `x.add(y)` en `x + y`, `x.mul(y)` en `x * y` etc.
— Ajoutez `pragma abicoder v1;` si vous voulez rester avec l'ancien codeur ABI.
— Supprimez éventuellement `pragma experimental ABIEncoderV2` ou `pragma abicoder v2` car ils sont redondants.
— Changez `byte` en `bytes1`.
— Ajouter des conversions de types explicites intermédiaires si nécessaire.
— Combinez `c.f{gas : 10000}{value : 1}()` en `c.f{gas : 10000, value : 1}()`.
— Remplacez `msg.sender.transfer(x)` par `payable(msg.sender).transfer(x)` ou utilisez une variable stockée de type `adresse payable`.
— Remplacez `x**y**z` par `(x**y)**z`.
— Utilisez l'assemblage en ligne en remplacement de `log0`, ..., `log4`.
— Négation des entiers non signés en les soustrayant de la valeur maximale du type et en ajoutant 1 (par exemple, `type(uint256).max - x + 1`, tout en s'assurant que x n'est pas zéro)

3.27 Format NatSpec

Les contrats Solidity peuvent utiliser une forme spéciale de commentaires pour fournir une documentation riche pour les fonctions, les variables de retour et autres. Cette forme spéciale est nommé Ethereum Natural Language Specification Format (NatSpec).

Note : NatSpec a été inspiré par Doxygen. Bien qu'il utilise des commentaires et des balises de style Doxygen, il n'y a aucune intention de garder une compatibilité stricte avec Doxygen. Veuillez examiner attentivement les balises supportées listées ci-dessous.

Cette documentation est segmentée en messages destinés aux développeurs et en messages destinés aux l'utilisateur finaux. Ces messages peuvent être présentés à l'utilisateur final (l'humain) au moment où il interagit avec le contrat (c'est-à-dire lorsqu'il signe une transaction).

Il est recommandé que les contrats Solidity soient entièrement annotés à l'aide de NatSpec pour toutes les interfaces publiques (tout ce qui se trouve dans l'ABI).

NatSpec inclut le formatage des commentaires que l'auteur du contrat intelligent utilisera et qui sont compris par le compilateur Solidity. Ils sont également détaillés ci-dessous sortie du compilateur Solidity, qui extrait ces commentaires dans un format lisible par la machine.

NatSpec peut également inclure des annotations utilisées par des outils tiers. Celles-ci sont très probablement via la balise `@custom:<name>`, et un bon cas d'utilisation est celui des outils d'analyse et de vérification.

3.27.1 Exemple de documentation

La documentation est insérée au-dessus de chaque `contrat`, `interface`, `fonction`, et `event` en utilisant le format de notation Doxygen. Une variable d'état `public` est équivalente à une `function` pour les besoins de NatSpec.

— **Pour Solidity, vous pouvez choisir `///` pour les commentaires d'une ou plusieurs lignes** commentaires, ou `/**` et se terminant par `*/`.
— **Pour Vyper, utilisez `"""` indenté jusqu'au contenu intérieur avec des** commentaires. Voir la documentation de Vyper.

L'exemple suivant montre un contrat et une fonction utilisant toutes les balises disponibles.

Note : Le compilateur Solidity n'interprète les balises que si elles sont externes ou publiques. Vous pouvez utiliser des commentaires similaires pour vos fonctions internes et privées, mais elles ne seront pas interprétées.

Ceci pourrait changer à l'avenir.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title Un simulateur pour les arbres
/// @author Larry A. Gardner
/// @notice Vous ne pouvez utiliser ce contrat que pour la simulation la plus␣
↪élémentaire.
/// Tous les appels de fonctions sont actuellement implémentés sans effets␣
↪secondaires.
/// @custom:experimental Il s'agit d'un contrat expérimental.
contract Tree {
    /// @notice Calculer l'âge de l'arbre en années, arrondi à l'unité supérieure,␣
↪pour les arbres vivants.
    /// @dev L'algorithme d'Alexandr N. Tetearing pourrait améliorer la précision.
    /// @param rings Le nombre de cernes de l'échantillon dendrochronologique.
    /// @return Âge en années, arrondi au chiffre supérieur pour les années␣
↪partielles
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice Renvoie le nombre de feuilles de l'arbre.
    /// @dev Renvoie uniquement un nombre fixe.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

contract Plant {
    function leaves() external virtual pure returns(uint256) {
        return 3;
    }
}

contract KumquatTree is Tree, Plant {
    function age(uint256 rings) external override pure returns (uint256) {
        return rings + 2;
    }

    /// Retourne le nombre de feuilles que possède ce type d'arbre spécifique.
    /// @inheritdoc Arbre
    function leaves() external override(Tree, Plant) pure returns(uint256) {
        return 3;
    }
}
```

3.27.2 Tags

Toutes les balises sont facultatives. Le tableau suivant explique le but de chaque balise NatSpec et où elle peut être utilisée. Dans un cas particulier, si aucune balise n'est utilisée, le compilateur Solidity interprétera un commentaire `///` ou `/**` de la même manière que s'il était balisé avec *@notice* `.

Tag		Contexte
`@title`	Un titre qui doit décrire le contrat/interface	contract, library, interface
`@author`	Le nom de l'auteur	contract, library, interface
`@notice`	Expliquer à un utilisateur final ce que cela fait	contract, library, interface, function, public state variable, event
`@dev`	Expliquez à un développeur tout détail supplémentaire	contract, library, interface, function, state variable, event
`@param`	Documente un paramètre comme dans Doxygen (doit être suivi du nom du paramètre)	function, event
`@return`	Documente les variables de retour de la fonction d'un contrat	function, public state variable
`@inheritdoc`	Copie toutes les étiquettes manquantes de la fonction de base (doit être suivi du nom du contrat).	function, public state variable
`@custom:...`	Balise personnalisée, la sémantique est définie par l'application.	everywhere

Si votre fonction renvoie plusieurs valeurs, comme `(int quotient, int remainder)`, alors utilisez plusieurs instructions `return` dans le même format que les instructions `@param`.

Les balises personnalisées commencent par `@custom:` et doivent être suivies d'une ou plusieurs lettres minuscules ou d'un trait d'union. Elles ne peuvent cependant pas commencer par un trait d'union. Elles peuvent être utilisées partout et font partie de la documentation du développeur.

Expressions dynamiques

Le compilateur Solidity fera passer la documentation NatSpec de votre code source Solidity jusqu'à la sortie JSON, comme décrit dans ce guide. Le consommateur de ce JSON, par exemple le logiciel client de l'utilisateur final, peut le présenter directement à l'utilisateur final ou appliquer un prétraitement.

Par exemple, certains logiciels clients effectueront un rendu :

```
/// @notice Cette fonction va multiplier `a` par 7
```

to the end-user as :

```
Cette fonction va multiplier 10 par 7
```

Si une fonction est appelée et que la valeur 10 est attribuée à l'entrée `a`.

La spécification de ces expressions dynamiques n'entre pas dans le cadre de la documentation de Solidity. et vous pouvez en savoir plus à l'adresse suivante le projet radspec.

Notes sur l'héritage

Les fonctions sans NatSpec hériteront automatiquement de la documentation de leur fonction de base. Les exceptions à cette règle sont :

— Lorsque les noms des paramètres sont différents.
— Quand il y a plus d'une fonction de base.
— Quand il y a une balise explicite `@inheritdoc` qui spécifie quel contrat doit être utilisé pour hériter.

3.27.3 Sortie de documentation

Lorsqu'elle est analysée par le compilateur, une documentation telle que celle de l'exemple ci-dessus produira deux fichiers JSON différents. L'un est destiné à être consommé par l'utilisateur final comme un avis lorsqu'une fonction est exécutée et l'autre à être utilisé par le développeur.

Si le contrat ci-dessus est enregistré sous le nom de `ex1.sol`, alors vous pouvez générer la documentation en utilisant :

```
solc --userdoc --devdoc ex1.sol
```

Et la sortie est ci-dessous.

Note : À partir de la version 0.6.11 de Solidity, la sortie NatSpec contient également un champ `version` et un champ `kind`. Actuellement, la `version` est fixée à `1` et le `kind` doit être l'un de `user` ou `dev`. Dans le futur, il est possible que de nouvelles versions soient introduites et que les anciennes soient supprimées.

Documentation pour les utilisateurs

La documentation ci-dessus produira la documentation utilisateur suivante Fichier JSON en sortie :

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculez l'âge de l'arbre en années, arrondi au chiffre supérieur,␣
↪pour les arbres vivants."
    }
  },
  "notice" : "Vous pouvez utiliser ce contrat uniquement pour la simulation la plus␣
↪basique"
}
```

Notez que la clé permettant de trouver les méthodes est la signature canonique de la fonction telle que définie dans le *Contrat ABI* et non le simple nom de la fonction.

Documentation pour les développeurs

Outre le fichier de documentation utilisateur, un fichier JSON de documentation pour les développeurs doit également être produit et doit ressembler à ceci :

```
{
  "version" : 1,
  "kind" : "dev",
  "author" : "Larry A. Gardner",
  "details" : "Tous les appels de fonction sont actuellement mis en œuvre sans effets␣
↪secondaires",
  "custom:experimental" : "Il s'agit d'un contrat expérimental.",
  "methods" :
  {
    "age(uint256)" :
    {
      "details" : "L'algorithme d'Alexandr N. Tetearing pourrait augmenter la précision",
      "params" :
      {
        "rings" : "Le nombre de cernes de l'échantillon dendrochronologique"
      },
      "return" : "âge en années, arrondi au chiffre supérieur pour les années incomplètes
↪"
    }
  },
  "title" : "Un simulateur pour les arbres"
}
```

3.28 Considérations de sécurité

Alors qu'il est généralement assez facile de construire un logiciel qui fonctionne comme prévu, il est beaucoup plus difficile de vérifier que personne ne peut l'utiliser d'une manière **non** prévue.

Dans Solidity, cela est encore plus important car vous pouvez utiliser des contrats intelligents pour gérer des jetons ou, éventuellement, des choses encore plus précieuses. De plus, chaque exécution d'un contrat intelligent se fait en public et, en plus de cela, le code source est souvent disponible.

Bien sûr, il faut toujours tenir compte de l'importance de l'enjeu : Vous pouvez comparer un contrat intelligent avec un service web qui est ouvert au public (et donc, également aux acteurs malveillants) et peut-être même open source. Si vous ne stockez que votre liste de courses sur ce service web, vous n'aurez peut-être pas à prendre trop de précautions, mais si vous gérez votre compte bancaire en utilisant ce service web, vous devriez être plus prudent.

Cette section énumère quelques pièges et recommandations générales en matière de sécurité mais ne peut, bien entendu, jamais être complète. Gardez également à l'esprit que même si le code de votre smart contrat intelligent est exempt de bogues, le compilateur ou la plateforme elle-même peuvent en bug. Une liste de certains bogues du compilateur liés à la sécurité et connus du public peut être trouvée dans la *liste des bugs connus*, qui est également lisible par machine. Notez qu'il existe un programme de prime de bogue qui couvre le générateur de code du compilateur Solidity.

Comme toujours, avec la documentation open source, merci de nous aider à étendre cette section (surtout, quelques exemples ne feraient pas de mal) !

NOTE : En plus de la liste ci-dessous, vous pouvez trouver plus de recommandations de sécurité et de meilleures pratiques dans la liste de connaissances de Guy Lando et le repo GitHub de Consensys.

3.28.1 Pièges

Information privée et aléatoire

Tout ce que vous utilisez dans un contrat intelligent est visible publiquement, même les variables locales et les variables d'état marquées `private`.

L'utilisation de nombres aléatoires dans les contrats intelligents est assez délicat si vous ne voulez pas que les mineurs soient capables de tricher.

Ré-entrée en scène

Toute interaction d'un contrat (A) avec un autre contrat (B) et tout transfert d'Ether transmet le contrôle à ce contrat (B). Il est donc possible pour B de rappeler A avant que cette interaction ne soit terminée. Pour donner un exemple, le code suivant contient un bug (il ne s'agit que d'un extrait et non d'un contrat complet) :

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

// CE CONTRAT CONTIENT UN BUG - NE PAS UTILISER
contract Fund {
    /// @dev Cartographie des parts d'éther du contrat.
    mapping(address => uint) shares;
    /// Retirez votre part.
    function withdraw() public {
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

Le problème n'est pas trop grave ici en raison du gaz limité dans le cadre de de `send`, mais il expose quand même une faiblesse : Le transfert d'éther peut toujours inclure l'exécution de code, donc le destinataire pourrait être un contrat qui appelle dans `withdraw`. Cela lui permettrait d'obtenir de multiples remboursements et de récupérer tout l'Ether du contrat. En particulier, le contrat suivant permettra à un attaquant de rembourser plusieurs fois car il utilise `call` qui renvoie tout le gaz restant par défaut :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

// CE CONTRAT CONTIENT UN BUG - NE PAS UTILISER
contract Fund {
    /// @dev Cartographie des parts d'éther du contrat.
    mapping(address => uint) shares;
    /// Retirez votre part.
    function withdraw() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

Pour éviter la ré-entrance, vous pouvez utiliser le modèle Checks-Effects-Interactions comme comme indiqué ci-dessous :

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

contract Fund {
    /// @dev Cartographie des parts d'éther du contrat.
    mapping(address => uint) shares;
    /// Retirez votre part.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        payable(msg.sender).transfer(share);
    }
}
```

Notez que la ré-entrance n'est pas seulement un effet du transfert d'Ether mais de tout appel de fonction sur un autre contrat. De plus, vous devez également prendre en compte les situations de multi-contrats. Un contrat appelé pourrait modifier l'état d'un autre contrat dont vous dépendez.

Limite et boucles de gaz

Les boucles qui n'ont pas un nombre fixe d'itérations, par exemple les boucles qui dépendent de valeurs de stockage, doivent être utilisées avec précaution : En raison de la limite de gaz de bloc, les transactions ne peuvent consommer qu'une certaine quantité de gaz. Que ce soit explicitement ou simplement en raison du fonctionnement normal, le nombre d'itérations d'une boucle peut dépasser la limite de gaz en bloc, ce qui peut entraîner que le contrat complet soit bloqué à un certain point. Cela peut ne pas s'appliquer aux fonctions `view` qui sont uniquement exécutées pour lire les données de la blockchain. Cependant, de telles fonctions peuvent être appelées par d'autres contrats dans le cadre d'opérations sur la blockchain et les bloquer. Veuillez être explicite sur ces cas dans la documentation de vos contrats.

Envoi et réception d'Ether

- Ni les contrats ni les « comptes externes » ne sont actuellement capables d'empêcher que quelqu'un leur envoie de l'Ether. Les contrats peuvent réagir et rejeter un transfert régulier, mais il existe des moyens de déplacer de l'Ether sans créer un appel de message. Une façon est de simplement « miner vers » l'adresse du contrat et la seconde façon est d'utiliser `selfdestruct(x)`.
- Si un contrat reçoit de l'Ether (sans qu'une fonction soit appelée), soit la *receive Ether*, soit la fonction *fallback* est exécutée. S'il n'a ni fonction de réception ni fonction de repli, l'éther sera rejeté (en lançant une exception). Pendant l'exécution d'une de ces fonctions, le contrat ne peut compter que sur le « supplément de gaz » qui lui est transmis (2300 gaz) dont il dispose à ce moment-là. Cette allocation n'est pas suffisante pour modifier le stockage (ne considérez pas cela comme acquis, l'allocation pourrait changer avec les futures hard forks). Pour être sûr que votre contrat peut recevoir de l'Ether de cette manière, vérifiez les exigences en matière de gaz des fonctions de réception et de repli (par exemple dans la section « details » de Remix).
- Il existe un moyen de transmettre plus de gaz au contrat récepteur en utilisant `addr.call{value : x}("")`. C'est essentiellement la même chose que `addr.transfer(x)`, sauf qu'elle transmet tout le gaz restant et donne la possibilité au destinataire d'effectuer des actions plus coûteuses (et il renvoie un code d'échec au lieu de propager automatiquement l'erreur). Cela peut inclure le rappel dans le contrat d'envoi ou d'autres changements d'état auxquels vous n'auriez peut-être pas pensé. Cela permet donc une grande flexibilité pour les utilisateurs honnêtes mais aussi pour les acteurs malveillants.
- Utilisez les unités les plus précises possibles pour représenter le montant du wei, car vous perdez tout ce qui est arrondi en raison d'un manque de précision.
- Si vous voulez envoyer des Ether en utilisant `address.transfer`, il y a certains détails à connaître :
  1. Si le destinataire est un contrat, il provoque l'exécution de sa fonction de réception ou de repli qui peut, à son tour, rappeler le contrat émetteur.

2. L'envoi d'Ether peut échouer si la profondeur d'appel dépasse 1024. Puisque l'appelant a le contrôle total de la profondeur d'appel, il peut faire échouer le transfert ; tenez compte de cette possibilité ou utilisez `send` et assurez-vous de toujours vérifier sa valeur de retour. Mieux encore, écrivez votre contrat en utilisant un modèle où le destinataire peut retirer de l'Ether à la place.
3. L'envoi d'Ether peut également échouer parce que l'exécution du contrat du destinataire nécessite plus que la quantité d'essence allouée (explicitement en utilisant *require*, *assert*, *revert* ou parce que l'opération est trop coûteuse) - il « tombe en panne sèche » (OOG). Si vous utilisez `transfer` ou `send` avec une vérification de la valeur de retour, cela pourrait être un moyen pour le destinataire de bloquer la progression du contrat d'envoi. Là encore, la meilleure pratique consiste à :ref :``utiliser un motif « withdraw » plutôt qu'un motif « send » <withdrawal_pattern>`.

Profondeur de la pile d'appel

Les appels de fonctions externes peuvent échouer à tout moment parce qu'ils dépassent la limite de taille de la pile d'appels de 1024. Dans de telles situations, Solidity lève une exception. Les acteurs malveillants pourraient être en mesure de forcer la pile d'appels à une valeur élevée avant d'interagir avec votre contrat. Notez que, depuis que Tangerine Whistle hardfork, la règle 63/64 rend l'attaque de la profondeur de la pile d'appels impraticable. Notez également que la pile d'appel et la pile d'expression ne sont pas liées, même si toutes deux ont une limite de taille de 1024 emplacements de pile.

Notez que `.send()` ne lève **pas** d'exception si la pile d'appels est épuisée, mais renvoie plutôt `false` dans ce cas. Les fonctions de bas niveau `.call()`, `.delegatecall()` et `.staticcall()` se comportent de la même manière.

Procurations autorisées

Si votre contrat peut agir comme un proxy, c'est-à-dire s'il peut appeler des contrats arbitraires avec des données fournies par l'utilisateur, alors l'utilisateur peut essentiellement assumer l'identité du contrat proxy. Même si vous avez mis en place d'autres mesures de protection, il est préférable de construire votre système de contrat de telle sorte que le proxy n'a aucune autorisation (même pas pour lui-même). Si nécessaire, vous pouvez y parvenir en utilisant un deuxième 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);
    }
    // Autres fonctions et autres fonctionnalités
}

// Il s'agit du contrat complet, il n'a pas d'autre fonctionnalités et
// ne nécessite aucun privilège pour fonctionner.
contract PermissionlessProxy {
    function callOther(address _addr, bytes memory _payload) public
            returns (bool, bytes memory) {
        return _addr.call(_payload);
    }
}
```

tx.origin

N'utilisez jamais tx.origin pour l'autorisation. Disons que vous avez un contrat de portefeuille comme celui-ci :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// CE CONTRAT CONTIENT UN BUG - NE PAS UTILISER
contract TxUserWallet {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint amount) public {
        // LE BOGUE EST ICI, vous devez utiliser msg.sender au lieu de tx.origin
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Maintenant, quelqu'un vous incite à envoyer de l'Ether à l'adresse de ce portefeuille d'attaque :

```
// 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 votre porte-monnaie avait vérifié l'autorisation de `msg.sender`, il aurait obtenu l'adresse du porte-monnaie attaqué, au lieu de l'adresse du propriétaire. Mais en vérifiant `tx.origin`, il obtient l'adresse originale qui a déclenché la transaction, qui est toujours l'adresse du propriétaire. Le porte-monnaie attaqué draine instantanément tous vos fonds.

Complément à deux / Débordements / Débordements

Comme dans de nombreux langages de programmation, les types entiers de Solidity ne sont pas réellement des entiers. Ils ressemblent à des entiers lorsque les valeurs sont petites, mais ne peuvent pas représenter des nombres arbitrairement grands.

Le code suivant provoque un dépassement de capacité parce que le résultat de l'addition est trop grand pour être stocké dans le type `uint8` :

```
uint8 x = 255;
uint8 y = 1;
return x + y;
```

Solidity a deux modes dans lesquels il traite ces débordements : Le mode vérifié et le mode non vérifié ou le mode « enveloppant ».

Le mode vérifié par défaut détecte les dépassements et provoque l'échec de l'assertion. Vous pouvez désactiver cette vérification en utilisant `unchecked { ... }`, ce qui aura pour effet d'ignorer le débordement en silence. Le code ci-dessus renverrait `0` s'il était enveloppé dans `unchecked { ... }`.

Même en mode vérifié, ne pensez pas que vous êtes protégé des bogues de débordement. Dans ce mode, les débordements se retourneront toujours. S'il n'est pas possible d'éviter le débordement, cela peut conduire à ce qu'un contrat intelligent soit bloqué dans un certain état.

En général, il faut lire les limites de la représentation par complément à deux, qui présente même des cas limites plus spéciaux pour les nombres signés.

Essayez d'utiliser `require` pour limiter la taille des entrées à un intervalle raisonnable et utilisez la fonction *SMT checker* pour trouver les débordements potentiels.

Effacement des mappages

Le type Solidity `mapping` (voir *Mapping Types*) est une structure de données de type clé-valeur qui ne garde pas la trace des clés auxquelles qui ont reçu une valeur non nulle. Pour cette raison, le nettoyage d'un mappage sans informations supplémentaires sur les clés écrites n'est pas possible. Si un `mapping` est utilisé comme type de base d'un tableau de stockage dynamique, la suppression ou l'éclatement du tableau n'aura aucun effet sur les éléments du `mapping`. Il en va de même, par exemple, si un `mapping` est utilisé comme type d'un champ d'une `structure` qui est le type de base d'un tableau de stockage dynamique. Le site `mapping` est également ignoré dans les affectations de structs ou de tableaux contenant 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;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function readMap(uint _map, uint _key) public view returns (uint) {
        return array[_map][_key];
    }

    function eraseMaps() public {
        delete array;
    }
}
```

Considérons l'exemple ci-dessus et la séquence d'appels suivante : `allocate(10)`, `writeMap(4, 128, 256)`. À ce stade, l'appel à `readMap(4, 128)` renvoie 256. Si on appelle `eraseMaps`, la longueur de la variable d'état `array` est remise à zéro, mais mais comme ses éléments `mapping` ne peuvent être mis à zéro, leurs informations restent vivantes dans le stockage du contrat. Après avoir supprimé `array`, l'appel à `allocate(5)` nous permet d'accéder à `array[4]` à nouveau, et l'appel à `readMap(4, 128)` renvoie 256 même sans un autre appel à `writeMap`.

Si vos informations de `mapping` doivent être effacées, envisagez d'utiliser une bibliothèque similaire à iterable mapping, vous permettant de parcourir les clés et de supprimer leurs valeurs dans le `mapping` approprié.

Détails mineurs

— Les types qui n'occupent pas la totalité des 32 octets peuvent contenir des « bits d'ordre supérieur sales ». Ceci est particulièrement important si vous accédez à `msg.data` - cela pose un risque de malléabilité : Vous pouvez créer des transactions qui appellent une fonction `f(uint8 x)` avec un argument brut de 32 octets de `0xff000001` et avec `0x00000001`. Les deux sont envoyés au contrat et les deux ressemblent au nombre 1 en ce qui concerne `x`, mais `msg.data` sera différente, donc si vous utilisez `keccak256(msg.data)` pour quoi que ce soit, vous obtiendrez des résultats différents.

3.28.2 Recommandations

Prenez les avertissements au sérieux

Si le compilateur vous avertit de quelque chose, vous devez le modifier. Même si vous ne pensez pas que cet avertissement particulier a des implications de sécurité, il peut y avoir un autre problème caché. Tout avertissement du compilateur que nous émettons peut être réduit au silence par de légères modifications du code.

Utilisez toujours la dernière version du compilateur pour être informé de tous les avertissements récemment introduits.

Les messages de type `info` émis par le compilateur ne sont pas dangereux, et représentent simplement des suggestions supplémentaires et des informations optionnelles que le compilateur pense pourrait être utile à l'utilisateur.

Limiter la quantité d'éther

Restreindre la quantité d'Ether (ou d'autres jetons) qui peut être stockée dans un contrat intelligent. Si votre code source, le compilateur ou la plateforme a un bug, ces fonds peuvent être perdus. Si vous voulez limiter vos pertes, limitez la quantité d'Ether.

Restez petit et modulaire

Gardez vos contrats petits et facilement compréhensibles. Isolez les fonctionnalités sans rapport dans d'autres contrats ou dans des bibliothèques. Les recommandations générales sur la qualité du code source s'appliquent bien sûr : Limitez la quantité de variables locales, la longueur des fonctions et ainsi de suite. Documentez vos fonctions afin que les autres puissent voir quelle était votre intention et si elle est différente de ce que fait le code.

Utiliser le modèle Verifications-Effects-Interactions

La plupart des fonctions vont d'abord effectuer quelques vérifications (qui a appelé la fonction, les arguments sont-ils à portée, ont-ils envoyé assez d'Ether, la personne a-t-elle des jetons, etc.) Ces vérifications doivent être effectuées en premier.

Dans un second temps, si toutes les vérifications sont passées, les effets sur les variables d'état du contrat en cours. L'interaction avec d'autres contrats doit être la toute dernière étape de toute fonction.

Les premiers contrats retardaient certains effets et attendaient que les appels de fonctions externes reviennent dans un état de non-erreur. C'est souvent une grave erreur à cause du problème de ré-entrance expliqué ci-dessus.

Notez également que les appels à des contrats connus peuvent à leur tour provoquer des appels à des contrats inconnus, il est donc probablement préférable de toujours appliquer ce modèle.

Inclure un mode de sécurité intégrée

Bien que le fait de rendre votre système entièrement décentralisé supprime tout intermédiaire, ce serait une bonne idée, surtout pour un nouveau code, d'inclure une sorte de mécanisme de sécurité :

Vous pouvez ajouter une fonction dans votre contrat intelligent qui effectue quelques des auto-vérifications comme « Y a-t-il eu une fuite d'Ether ? », « La somme des jetons est-elle égale au solde du contrat ? » ou des choses similaires. Gardez à l'esprit que vous ne pouvez pas utiliser trop d'essence pour cela, donc de l'aide par des calculs hors-chaîne peut être nécessaire.

Si l'auto-vérification échoue, le contrat passe automatiquement dans une sorte de mode « failsafe », qui, par exemple, désactive la plupart des fonctions, remet le contrôle à un tiers fixe et de confiance ou simplement convertir le contrat en un simple contrat « rendez-moi mon argent ».

Demandez un examen par les pairs

Plus il y a de personnes qui examinent un morceau de code, plus on découvre de problèmes. Demander à des personnes d'examiner votre code permet également de vérifier par recoupement si votre code est facile à comprendre - un critère très important pour les bons contrats intelligents.

3.29 SMTChecker et vérification formelle

En utilisant la vérification formelle, il est possible d'effectuer une preuve mathématique automatisée que votre code source répond à une certaine spécification formelle. La spécification est toujours formelle (tout comme le code source), mais généralement beaucoup plus simple.

Notez que la vérification formelle elle-même ne peut vous aider qu'à comprendre la différence entre ce que vous avez fait (la spécification) et la manière dont vous l'avez fait (l'implémentation réelle). Vous devez toujours vérifier si la spécification correspond à ce que vous vouliez et que vous n'avez pas manqué d'effets involontaires.

Solidity met en œuvre une approche de vérification formelle basée sur SMT (Satisfiability Modulo Theories) et la Horn de résolution. Le module SMTChecker essaie automatiquement de prouver que le code satisfait à la spécification

donnée par les déclarations `require` et `assert`. C'est-à-dire qu'il considère les déclarations `require` comme des hypothèses et essaie de prouver que les conditions contenues dans les déclarations `assert` sont toujours vraies. Si un échec d'assertion est trouvé, un contre-exemple peut être donné à l'utilisateur montrant comment l'assertion peut être violée. Si aucun avertissement n'est donné par le SMTChecker pour une propriété, cela signifie que la propriété est sûre.

Les autres cibles de vérification que le SMTChecker vérifie au moment de la compilation sont :
- Les débordements et les sous-écoulements arithmétiques.
- La division par zéro.
- Conditions triviales et code inaccessible.
- Extraction d'un tableau vide.
- Accès à un index hors limites.
- Fonds insuffisants pour un transfert.

Toutes les cibles ci-dessus sont automatiquement vérifiées par défaut si tous les moteurs sont activés, sauf underflow et overflow pour Solidity >=0.8.7.

Les avertissements potentiels que le SMTChecker rapporte sont :
- `<failing property> happens here.`. Cela signifie que le SMTChecker a prouvé qu'une certaine propriété est défaillante. Un contre-exemple peut être donné, cependant dans des situations complexes, il peut aussi ne pas montrer de contre-exemple. Ce résultat peut aussi être un faux positif dans certains cas, lorsque l'encodage SMT ajoute des abstractions pour le code Solidity qui est difficile ou impossible à exprimer.
- `<failing property> might happen here`. Cela signifie que le solveur n'a pas pu prouver l'un ou l'autre cas dans le délai imparti. Comme le résultat est inconnu, le SMTChecker rapporte l'échec potentiel pour la solidité. Cela peut être résolu en augmentant le délai d'interrogation, mais le problème peut aussi être simplement trop difficile à résoudre pour le moteur.

Pour activer le SMTChecker, vous devez sélectionner *quel moteur doit fonctionner*, où la valeur par défaut est aucun moteur. La sélection du moteur active le SMTChecker sur tous les fichiers.

Note : Avant Solidity 0.8.4, la manière par défaut d'activer le SMTChecker était via `pragma experimental SMTChecker;` et seuls les contrats contenant le pragma seraient analysés. Ce pragme a été déprécié, et bien qu'il active toujours le qu'il active toujours le SMTChecker pour une compatibilité ascendante, il sera supprimé dans Solidity 0.9.0. Notez également que maintenant l'utilisation du pragma même dans un seul fichier active le SMTChecker pour tous les fichiers.

Note : L'absence d'avertissement pour une cible de vérification représente une preuve mathématique incontestable de l'exactitude, en supposant l'absence de bogues dans le SMTChecker et le solveur sous-jacent. Gardez à l'esprit que ces problèmes sont *très difficiles* et parfois *impossibles* à résoudre automatiquement dans le cas général. Par conséquent, plusieurs propriétés pourraient ne pas être résolues ou pourraient conduire à des faux positifs pour les grands contrats. Chaque propriété prouvée doit être être considérée comme une réalisation importante. Pour les utilisateurs avancés, voir *SMTChecker Tuning* pour apprendre quelques options qui pourraient aider à prouver des propriétés complexes.

3.29.1 Tutoriel

Débordement

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

contract Overflow {
    uint immutable x;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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);
    }
}
```

Le contrat ci-dessus montre un exemple de vérification de débordement (overflow). Le SMTChecker ne vérifie pas l'underflow et l'overflow par défaut pour Solidity >=0.8.7, donc nous devons utiliser l'option de ligne de commande `--model-checker-targets "underflow,overflow"` ou l'option JSON `settings.modelChecker.targets = ["underflow", "overflow"]`. Voir *cette section pour la configuration des cibles*. Ici, il signale ce qui suit :

```
Warning: CHC: Overflow (resulting value larger than 2**256 - 1) happens here.
Counterexample:
x = 1, y = 1157920892373161954235709850086879078532699846656405640394575840079131296399335
 = 0

Transaction trace:
Overflow.constructor(1,␣
↪1157920892373161954235709850086879078532699846656405640394575840079131296399335)
State: x = 1, y =␣
↪1157920892373161954235709850086879078532699846656405640394575840079131296399335
Overflow.stateAdd()
    Overflow.add(1,␣
↪1157920892373161954235709850086879078532699846656405640394575840079131296399335) --␣
↪internal call
 --> o.sol:9:20:
  |
9 |             return _x + _y;
  |                    ^^^^^^^
```

Si nous ajoutons des déclarations `require` qui filtrent les cas de débordement, le SMTChecker prouve qu'aucun débordement n'est atteignable (en ne signalant pas d'avertissement) :

```
// 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;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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);
    }
}
```

Affirmer

Une assertion représente un invariant dans votre code : une propriété qui doit être vraie *pour toutes les opérations, y compris toutes les valeurs d'entrée et de stockage*, sinon il y a un bug.

Le code ci-dessous définit une fonction `f` qui garantit l'absence de débordement. La fonction `inv` définit la spécification que `f` est monotone et croissante : pour chaque paire possible `(_a, _b)`, si `_b > _a` alors `f(_b) > f(_a)`. Puisque `f` est effectivement monotone et croissante, le SMTChecker prouve que notre propriété est correcte. Nous vous encourageons à jouer avec la propriété et la définition de la fonction pour voir les résultats qui en découlent !

```
// 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));
    }
}
```

Nous pouvons également ajouter des assertions à l'intérieur des boucles pour vérifier des propriétés plus complexes. Le code suivant recherche l'élément maximum d'un tableau non restreint de nombres, et affirme la propriété selon laquelle l'élément trouvé doit être supérieur ou égal à chaque élément du tableau.

```
// 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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            assert(m >= _a[i]);

        return m;
    }
}
```

Notez que dans cet exemple, le SMTChecker va automatiquement essayer de prouver trois propriétés :

1. ++i dans la première boucle ne déborde pas.
2. ++i dans la deuxième boucle ne déborde pas.
3. L'assertion est toujours vraie.

Note : Les propriétés impliquent des boucles, ce qui rend l'exercice *beaucoup plus difficile* que les exemples précédents, alors faites attention aux boucles !

Toutes les propriétés sont correctement prouvées sûres. N'hésitez pas à modifier et/ou d'ajouter des restrictions sur le tableau pour obtenir des résultats différents. Par exemple, en changeant le code en

```
// 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;
    }
}
```

nous donne :

```
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]);
```

Propriétés de l'État

Jusqu'à présent, les exemples ont seulement démontré l'utilisation du SMTChecker sur du code pur, prouvant des propriétés sur des opérations ou des algorithmes spécifiques. Un type commun de propriétés dans les contrats intelligents sont les propriétés qui impliquent l'état du contrat. Plusieurs transactions peuvent être nécessaires pour faire échouer pour une telle propriété.

À titre d'exemple, considérons une grille 2D où les deux axes ont des coordonnées dans la plage (-2^128, 2^128 - 1). Plaçons un robot à la position (0, 0). Le robot ne peut se déplacer qu'en diagonale, un pas à la fois, et ne peut pas se déplacer en dehors de la grille. La machine à états du robot peut être représentée par le contrat intelligent ci-dessous.

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

contract Robot {
    int x = 0;
    int y = 0;

    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);
    }
}
```

La fonction `inv` représente un invariant de la machine à états selon lequel `x + y` doit être pair. Le SMTChecker parvient à prouver que quelque soit le nombre de commandes que l'on donne au robot, même s'ils sont infinis, l'invariant ne peut *jamais* échouer. Le lecteur intéressé peut vouloir prouver ce fait manuellement aussi. Indice : cet invariant est inductif.

Nous pouvons aussi tromper le SMTChecker pour qu'il nous donne un chemin vers une position que nous pensons être atteignable. Nous pouvons ajouter la propriété que (2, 4) est *non* accessible, en ajoutant la fonction suivante.

```
function reach_2_4() public view {
    assert(!(x == 2 && y == 4));
}
```

Cette propriété est fausse, et tout en prouvant que la propriété est fausse, le SMTChecker nous dit exactement *comment* atteindre (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
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));
   |            ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Notez que le chemin ci-dessus n'est pas nécessairement déterministe, car il y a d'autres chemins qui pourraient atteindre (2, 4). Le choix du chemin affiché peut changer en fonction du solveur utilisé, de sa version, ou simplement au hasard.

Appels externes et réentrance

Chaque appel externe est traité comme un appel à un code inconnu par le SMTChecker. Le raisonnement derrière cela est que même si le code du contrat appelé est disponible au moment de la compilation, il n'y a aucune garantie que le contrat déployé sera bien le même que le contrat d'où provient l'interface au moment de la compilation.

Dans certains cas, il est possible de déduire automatiquement des propriétés sur les variables d'état qui restent vraies même si le code appelé de l'extérieur peut faire n'importe quoi, y compris réintroduire le contrat de l'appelant.

```
// 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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;
    }

    function run() mutex public {
        uint xPre = x;
        unknown.run();
        assert(xPre == x);
    }
}
```

L'exemple ci-dessus montre un contrat qui utilise un drapeau mutex pour interdire la réentrance. Le solveur est capable de déduire que lorsque `unknown.run()` est appelé, le contrat est déjà « verrouillé », donc il ne serait pas possible de changer la valeur de `x`, indépendamment de ce que fait le code appelé inconnu.

Si nous « oublions » d'utiliser le modificateur `mutex` sur la fonction `set`, le SMTChecker est capable de synthétiser le comportement du code appelé de manière externe que l'assertion échoue :

```
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);
   |                 ^^^^^^^^^^^^^^^^
```

3.29.2 Options et réglages de SMTChecker

Délai d'attente

Le SMTChecker utilise une limite de ressource codée en dur (`rlimit`) choisie par solveur, qui n'est pas précisément liée au temps. Nous avons choisi l'option `rlimit` comme défaut car elle donne plus de garanties de déterminisme que le temps à l'intérieur du solveur.

Cette option se traduit approximativement par « un délai de quelques secondes » par requête. Bien sûr de nombreuses propriétés sont très complexes et nécessitent beaucoup de temps pour être résolus, où le déterminisme n'a pas d'importance. Si le SMTChecker ne parvient pas à résoudre les propriétés du contrat avec le `rlimit` par défaut, un timeout peut être donné en millisecondes via l'option CLI `--model-checker-timeout <time>` ou l'option JSON `settings.modelChecker.timeout=<time>`, où 0 signifie pas de délai d'attente.

Objectifs de vérification

Les types de cibles de vérification créées par le SMTChecker peuvent aussi être personnalisés via l'option CLI `--model-checker-target <targets>` ou l'option JSON `settings.modelChecker.targets=<targets>`. Dans le cas de l'interface CLI, `<targets>` est une liste non séparée par des virgules d'une ou plusieurs cibles de vérification, et un tableau d'une ou plusieurs cibles comme l'entrée JSON. Les mots-clés qui représentent les cibles sont :

- Assertions : `assert`.
- Débordement arithmétique : `underflow`.
- Débordement arithmétique : `overflow`.
- La division par zéro : `divByZero`.
- Conditions triviales et code inaccessible : `constantCondition`.
- Extraire un tableau vide : `popEmptyArray`.
- Accès hors limites aux tableaux et aux index d'octets fixes : `outOfBounds`.
- Fonds insuffisants pour un transfert : `balance`.
- Tous ces éléments : `défaut` (CLI uniquement).

Un sous-ensemble commun de cibles pourrait être, par exemple : `--model-checker-targets assert,overflow`.

Toutes les cibles sont vérifiées par défaut, sauf underflow et overflow pour Solidity >=0.8.7.

Il n'y a pas d'heuristique précise sur comment et quand diviser les cibles de vérification, mais cela peut être utile, surtout lorsqu'il s'agit de grands contrats.

Cibles non vérifiées

S'il existe des cibles non vérifiées, le SMTChecker émet un avertissement indiquant combien de cibles non vérifiées il y a. Si l'utilisateur souhaite voir toutes les cibles non corrigées, l'option CLI `--model-checker-show-unproved` et l'option JSON `settings.modelChecker.showUnproved = true` peuvent être utilisées.

Contrats vérifiés

Par défaut, tous les contrats déployables dans les sources données sont analysés séparément en tant que celui qui sera déployé. Cela signifie que si un contrat a de nombreux parents d'héritage direct et indirect, ils seront tous analysés séparément, même si seul le plus dérivé sera accessible directement sur la blockchain. Cela entraîne une charge inutile pour le SMTChecker et le solveur. Pour aider les cas comme celui-ci, les utilisateurs peuvent spécifier quels contrats doivent être analysés comme le déployé. Les contrats parents sont bien sûr toujours analysés, mais seulement dans le contexte du contrat le plus dérivé, ce qui réduit la complexité de l'encodage et des requêtes générées. Notez que les contrats abstraits ne sont par défaut pas analysés comme les plus dérivés par le SMTChecker.

Les contrats choisis peuvent être donnés via une liste séparée par des virgules (les espaces blancs ne sont pas autorisés) de paires <source> :<contrat> dans le CLI : `--model-checker-contracts "<source1.sol:contract1>, <source2.sol:contract2>,<source2.sol:contract3>"`, et via l'objet `settings.modelChecker.contracts` dans le *JSON input*, qui a la forme suivante :

```
"contracts": {
    "source1.sol": ["contract1"],
    "source2.sol": ["contract2", "contract3"]
}
```

Invariants inductifs rapportés et inférés

Pour les propriétés qui ont été prouvées sûres avec le moteur CHC, le SMTChecker peut récupérer les invariants inductifs qui ont été inférés par le solveur de Horn dans le cadre de la preuve. Actuellement, deux types d'invariants peuvent être rapportés à l'utilisateur :

- Invariants de contrat : ce sont des propriétés sur les variables d'état du contrat qui sont vraies avant et après chaque transaction possible que le contrat peut exécuter. Par exemple, `x >= y`, où `x` et `y` sont les variables d'état d'un contrat.
- Propriétés de réentraînement : elles représentent le comportement du contrat en présence d'appels externes à du code inconnu. Ces propriétés peuvent exprimer une relation entre la valeur des variables d'état avant et après l'appel externe, où l'appel externe est libre de faire n'importe quoi, y compris d'effectuer des appels réentrants au contrat analysé. Les variables amorcées représentent les valeurs des variables d'état après ledit appel externe. Exemple : `lock -> x = x'`.

L'utilisateur peut choisir le type d'invariants à rapporter en utilisant l'option CLI `--model-checker-invariants "contract,reentrancy"` ou comme un tableau dans le champ `settings.modelChecker.invariants` dans l'entrée *JSON*. Par défaut, le SMTChecker ne rapporte pas les invariants.

Division et modulo avec des variables muettes

Spacer, le solveur de Corne par défaut utilisé par le SMTChecker, n'aime souvent pas les opérations de division et de modulation dans les règles de Horn. Pour cette raison, par défaut, les opérations de division et de modulo de Solidity sont codées en utilisant la contrainte suivante `a = b * d + m` où `d = a / b` et `m = a % b`. Cependant, d'autres solveurs, comme Eldarica, préfèrent les opérations syntaxiquement précises. L'indicateur de ligne de commande `--model-checker-div-mod-no-slacks` et l'option JSON `settings.modelChecker.divModNoSlacks` peuvent être utilisés pour basculer le codage en fonction des préférences du solveur utilisé.

Abstraction des fonctions Natspec

Certaines fonctions, y compris les méthodes mathématiques courantes telles que `pow` et `sqrt` peuvent être trop complexes pour être analysées de manière entièrement automatisée. Ces fonctions peuvent être annotées avec des balises Natspec qui indiquent au contrôleur SMTChecker que ces fonctions doivent être abstraites. Cela signifie que de la fonction n'est pas utilisé et que, lorsqu'elle est appelée, la fonction :

- retournera une valeur non déterministe, et soit gardera les variables d'état inchangées si la fonction abstraite est view/pure, soit fixera également les variables d'état à des valeurs non déterministes dans le cas contraire. Ceci peut être utilisé via l'annotation `/// @custom:smtchecker abstract-function-nondet`.
- Agir comme une fonction non interprétée. Cela signifie que la sémantique de la fonction (donnée par le corps) est ignorée, et que la seule propriété de cette fonction est que, pour une même entrée, elle garantit la même sortie. Ceci est actuellement en cours de développement et sera disponible via l'annotation `/// @custom:smtchecker abstract-function-uf`.

Moteurs de vérification de modèles réduits

Le module SMTChecker implémente deux moteurs de raisonnement différents, un Bounded Model Checker (BMC) et un système de Clauses de Corne Contraintes (CHC). Les deux moteurs sont actuellement en cours de développement, et ont des caractéristiques différentes. Les moteurs sont indépendants et chaque avertissement de propriété indique de quel moteur il provient. Notez que tous les exemples ci-dessus avec des contre-exemples ont été rapportés par CHC, le moteur le plus puissant.

Par défaut, les deux moteurs sont utilisés, CHC s'exécute en premier, et chaque propriété qui n'a pas été prouvée est transmise à BMC. Vous pouvez choisir un moteur spécifique via l'interface CLI `--model-checker-engine {all,bmc,chc,none}` ou l'option JSON `settings.modelChecker.engine={all,bmc,chc,none}`.

Contrôleur de modèles délimités (BMC)

Le moteur BMC analyse les fonctions de manière isolée, c'est-à-dire qu'il ne prend pas en compte le comportement global du contrat sur plusieurs transactions lorsqu'il analyse chaque fonction. Les boucles sont également ignorées dans ce moteur pour le moment. Les appels de fonctions internes sont inlined tant qu'ils ne sont pas récursifs, directement ou indirectement. Les appels de fonctions externes sont inlined si possible. Connaissance qui est potentiellement affectée par la réentrance est effacée.

Les caractéristiques ci-dessus font que la BMC est susceptible de signaler des faux positifs, mais il est également léger et devrait être capable de trouver rapidement de petits bogues locaux.

Clauses de corne contraintes (CHC)

Le graphique de flux de contrôle (CFG) d'un contrat est modélisé comme un système de clauses de Horn, où le cycle de vie du contrat est représenté par une boucle qui peut visiter chaque fonction publique/externe de manière non-déterministe. De cette façon, le comportement de l'ensemble du contrat sur un nombre illimité de transactions est pris en compte lors de l'analyse de toute fonction. Les boucles sont entièrement prises en charge par ce moteur. Les appels de fonctions internes sont pris en charge, et les appels de fonctions externes supposent que le code appelé est inconnu et peut faire n'importe quoi.

Le moteur CHC est beaucoup plus puissant que BMC en termes de ce qu'il peut prouver, et peut nécessiter plus de ressources informatiques.

Solveurs SMT et Horn

Les deux moteurs détaillés ci-dessus utilisent des prouveurs de théorèmes automatisés comme leur logique. BMC utilise un solveur SMT, tandis que CHC utilise un solveur de Horn. Souvent le même outil peut agir comme les deux, comme on le voit dans z3, qui est principalement un solveur SMT et qui rend Spacer disponible comme solveur de Horn, et Eldarica qui fait les deux.

L'utilisateur peut choisir quels solveurs doivent être utilisés, s'ils sont disponibles, via l'option CLI `--model-checker-solvers {all,cvc4,smtlib2,z3}` ou l'option JSON `settings.modelChecker.solvers=[smtlib2,z3]`, où :

- `cvc4` n'est disponible que si le binaire `solc` est compilé avec. Seul BMC utilise `cvc4`.
- `smtlib2` produit des requêtes SMT/Horn dans le format smtlib2. Celles-ci peuvent être utilisées avec le mécanisme de rappel du compilateur de sorte que tout solveur binaire du système peut être employé pour renvoyer de manière synchrone les résultats des requêtes au compilateur. C'est actuellement la seule façon d'utiliser Eldarica, par exemple, puisqu'il ne dispose pas d'une API C++. Cela peut être utilisé à la fois par BMC et CHC, selon les solveurs appelés.
- `z3` est disponible
  - si `solc` est compilé avec lui ;

— si une bibliothèque dynamique `z3` de version 4.8.x est installée dans un système Linux (à partir de Solidity 0.7.6) ;
— statiquement dans `soljson.js` (à partir de Solidity 0.6.9), c'est-à-dire le binaire Javascript du compilateur.

Étant donné que BMC et CHC utilisent tous deux `z3`, et que `z3` est disponible dans une plus grande variété d'environnements, y compris dans le navigateur, la plupart des utilisateurs n'auront presque jamais à se préoccuper de cette option. Les utilisateurs plus avancés peuvent utiliser cette option pour essayer des solveurs alternatifs sur des problèmes plus complexes.

Veuillez noter que certaines combinaisons de moteur et de solveur choisis conduiront à ce que SMTChecker ne fera rien, par exemple choisir CHC et `cvc4`.

3.29.3 Abstraction et faux positifs

Le SMTChecker implémente les abstractions d'une manière incomplète et saine : Si un bogue est signalé, il peut s'agir d'un faux positif introduit par les abstractions (dû à l'effacement de connaissances ou l'utilisation d'un type non précis). S'il détermine qu'une cible de vérification est sûre, elle est effectivement sûre, c'est-à-dire qu'il n'y a pas de faux négatifs (à moins qu'il y ait un bug dans le SMTChecker).

Si une cible ne peut pas être prouvée, vous pouvez essayer d'aider le solveur en utilisant les options de réglage dans la section précédente. Si vous êtes sûr d'un faux positif, ajouter des déclarations `require` dans le code avec plus d'informations peut également donner plus de puissance au solveur.

Encodage et types SMT

L'encodage SMTChecker essaye d'être aussi précis que possible, en faisant correspondre les types et expressions Solidity à leur représentation SMT-LIB la plus proche, comme le montre le tableau ci-dessous.

Type Solidity	Triage SMT	Théories
Booléen	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
autres types	Integer	LIA

Les types qui ne sont pas encore pris en charge sont abstraits par un seul entier non signé de 256 bits, où leurs opérations non supportées sont ignorées.

Pour plus de détails sur la façon dont l'encodage SMT fonctionne en interne, voir l'article Vérification basée sur SMT des contrats intelligents Solidity.

Appels de fonction

Dans le moteur BMC, les appels de fonctions vers le même contrat (ou contrats de base) sont inlined lorsque cela est possible, c'est-à-dire lorsque leur implémentation est disponible. Les appels de fonctions dans d'autres contrats ne sont pas inlined même si leur code est disponible, car nous ne pouvons pas garantir que le code déployé est le même.

Le moteur CHC crée des clauses Horn non linéaires qui utilisent des résumés des fonctions appelées pour prendre en charge les appels de fonctions internes. Les appels de fonctions externes sont traités comme des appels à du code inconnu, y compris les appels réentrants potentiels.

Les fonctions pures complexes sont abstraites par une fonction non interprétée (UF) sur les arguments.

Fonctions	Comportement BMC/CHC
`assert`	Objectif de vérification.
`require`	Assomption.
appel interne	BMC : Appel de fonction en ligne. CHC : Résumés des fonctions.
appel externe à un code connu	BMC : Appel de fonction en ligne ou L'appel de fonction en ligne ou l'effacement des connaissances sur les variables d'état et des références de stockage local. CHC : Supposer que le code appelé est inconnu. Essayer de déduire les invariants qui tiennent après le retour de l'appel.
Réseau de stockage push/pop	Supporté précisément. Vérifie s'il s'agit de faire sauter un tableau vide.
Fonctions ABI	Abstracted with UF.
`addmod`, `mulmod`	Supported precisely.
`gasleft`, `blockhash`, `keccak256`, `ecrecover` `ripemd160`	Abstracted with UF.
Fonctions pures sans implémentation (externe ou complexe)	Abstraitement avec UF
fonctions externes sans mise en œuvre	BMC : Effacer les connaissances de l'État et assumer Le résultat est indéterminé. CHC : Résumé non déterministe. Essayez d'inférer des invariants qui tiennent après le retour de l'appel.
transfert	BMC : Vérifie si le solde du contrat est suffisant. CHC : n'effectue pas encore le contrôle.
autres	Actuellement non pris en charge

L'utilisation de l'abstraction signifie la perte de connaissances précises, mais dans de nombreux cas, elle ne signifie pas une perte de puissance de preuve.

```
// 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;
    }
}
```

Dans l'exemple ci-dessus, le SMTChecker n'est pas assez expressif pour calculer réellement « ecrecover », mais en modélisant les appels de fonctions comme des fonctions non interprétées, nous savons que la valeur de retour est la

même lorsqu'elle est appelée avec des paramètres équivalents. Ceci est suffisant pour prouver que l'assertion ci-dessus est toujours vraie.

L'abstraction d'un appel de fonction avec un UF peut être faite pour des fonctions connues pour être déterministes, et peut être facilement réalisée pour les fonctions pures. Il est cependant difficile de le faire avec des fonctions externes générales, puisqu'elles peuvent de variables d'état.

Types de référence et alias

Solidity implémente l'aliasing pour les types de référence avec le même *data emplacement*. Cela signifie qu'une variable peut être modifiée à travers une référence à la même données. Le SMTChecker ne garde pas trace des références qui font référence aux mêmes données. Cela implique que chaque fois qu'une référence locale ou une variable d'état de type référence est assignée, toutes les connaissances concernant les variables de même type et de même emplacement données est effacée. Si le type est imbriqué, la suppression de la connaissance inclut également tous les 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;
        // Effacer les connaissances sur les références mémoire ne devrait pas
        // effacer les connaissances sur les variables d'état.
        assert(array1[0] == 42);
        // Cependant, une affectation à une référence de stockage effacera
        // la connaissance du stockage en conséquence.
        d[0] = 2;
        // Échoue en tant que faux positif à cause de l'affectation ci-dessus.
        assert(array1[0] == 42);
        // Échoue car `a == b` est possible.
        assert(a[0] == 2);
        // Échoue car `c[i] == b` est 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 {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        f(a, b, c, array2[x]);
    }
}
```

Après l'affectation à `b[0]`, nous devons effacer la connaissance de `a`, puisqu'il a le même type (`uint[]`) et le même emplacement de données (mémoire). Nous devons également effacer les connaissances sur `c`, puisque son type de base est également un `uint[]` situé dans la mémoire. Cela implique qu'un `c[i]` pourrait faire référence aux mêmes données que `b` ou `a`.

Remarquez que nous n'avons pas de connaissances claires sur `array` et `d`, parce qu'ils sont situés dans le stockage, même s'ils ont aussi le type `uint[]`. Cependant, si `d` était assigné, nous devrions effacer la connaissance sur `array` et et vice-versa.

Bilan des contrats

Un contrat peut être déployé avec des fonds qui lui sont envoyés, si `msg.value` > 0 dans la transaction de déploiement. Cependant, l'adresse du contrat peut déjà avoir des fonds avant le déploiement, qui sont conservés par le contrat. Par conséquent, le SMTChecker suppose que `adress(this).balance >= msg.value` dans le constructeur afin d'être cohérent avec les règles EVM. Le solde du contrat peut également augmenter sans déclencher d'appel au contrat contrat, si :

- `selfdestruct` est exécuté par un autre contrat avec le contrat analysé comme cible des fonds restants,
- le contrat est la base de données de pièces de monnaie (i.e., `block.coinbase`) d'un bloc.

Pour modéliser cela correctement, le SMTChecker suppose qu'à chaque nouvelle transaction le solde du contrat peut augmenter d'au moins `msg.value`.

3.29.4 Hypothèses du monde réel

Certains scénarios peuvent être exprimés dans Solidity et dans l'EVM, mais on s'attend à ce qu'ils ne se produisent jamais se produire dans la pratique. L'un de ces cas est la longueur d'un tableau de stockage dynamique qui déborde pendant un processus de poussée : Si l'opération `push` est appliquée à un tableau de longueur 2^256 - 1, sa longueur déborde silencieusement. Cependant, il est peu probable que cela se produise dans la pratique, car les opérations nécessaires pour faire croître le tableau à ce point prendraient des milliards d'années à être exécutées. Une autre hypothèse similaire prise par le SMTChecker est que le solde d'une adresse ne peut jamais déborder.

Une idée similaire a été présentée dans EIP-1985.

3.30 Ressources

3.30.1 Ressources générales

- Ethereum.org Developer Portal
- Ethereum StackExchange
- Solidity Portal
- Solidity Changelog
- Solidity Source Code sur GitHub
- Solidity Chat des utilisateurs de langues
- Solidity Chat des développeurs de compilateurs
- Awesome Solidity
- Solidity par Example

3.30.2 Environnements de développement intégrés (Ethereum)

— **Brownie** Cadre de développement et de test basé sur Python pour les contrats intelligents ciblant la machine virtuelle Ethereum.
— **Dapp** Outil pour construire, tester et déployer des contrats intelligents à partir de la ligne de commande.
— **Embark** Plateforme de développeurs pour la création et le déploiement d'applications décentralisées.
— **Hardhat** Environnement de développement Ethereum avec réseau Ethereum local, fonctions de débogage et écosystème de plugins.
— **Remix** IDE basé sur un navigateur avec compilateur intégré et environnement d'exécution Solidity sans composants côté serveur.
— **Scaffold-ETH** Pile de développement Ethereum axée sur des itérations rapides du produit.
— **Truffle** Cadre de développement Ethereum.

3.30.3 Intégrations de l'éditeur

— Atom
  — **Etheratom** Plugin pour l'éditeur Atom qui propose la coloration syntaxique, la compilation et un environnement d'exécution (compatible avec les nœuds Backend et VM).
  — **Atom Solidity Linter** Plugin pour l'éditeur Atom qui fournit le linting Solidity.
  — **Atom Solium Linter** Linter Solidity configurable pour Atom utilisant Solium (maintenant Ethlint) comme base.
— Emacs
  — **Emacs Solidity** Plugin pour l'éditeur Emacs fournissant la coloration syntaxique et le signalement des erreurs de compilation.
— IntelliJ
  — **IntelliJ IDEA plugin** Plugin Solidity pour IntelliJ IDEA (et tous les autres IDE de JetBrains)
— Sublime
  — **Package for SublimeText - Solidity language syntax** Coloration syntaxique Solidity pour l'éditeur SublimeText.
— Vim
  — **Vim Solidity** Plugin pour l'éditeur Vim fournissant une coloration syntaxique.
  — **Vim Syntastic** Plugin pour l'éditeur Vim permettant de vérifier la compilation.
— Visual Studio Code
  — **Visual Studio Code extension** Plugin Solidity pour Microsoft Visual Studio Code qui comprend la coloration syntaxique et le compilateur Solidity.

3.30.4 Outils Solidity

— **ABI to Solidity interface converter** Un script pour générer des interfaces de contrat à partir de l'ABI d'un contrat intelligent.
— **abi-to-sol** Outil permettant de générer une source d'interface Solidity à partir d'un JSON ABI donné.
— **Doxity** Générateur de documentation pour Solidity.
— **Ethlint** Linter pour identifier et corriger les problèmes de style et de sécurité dans Solidity.
— **evmdis** EVM Disassembler qui effectue une analyse statique sur le bytecode pour fournir un niveau d'abstraction plus élevé que les opérations EVM brutes.
— **EVM Lab** Ensemble d'outils riches pour interagir avec l'EVM. Comprend une VM, une API Etherchain et un visualiseur de traces avec affichage du coût du gaz.
— **hevm** Débogueur EVM et moteur d'exécution symbolique.
— **leafleth** Un générateur de documentation pour les smart-contracts de Solidity.
— **PIET** Un outil pour développer, auditer et utiliser les contrats intelligents Solidity à travers une interface graphique simple.

— **sol2uml** Générateur de diagrammes de classe en langage de modélisation unifié (UML) pour les contrats Solidity.
— **solc-select** A script to quickly switch between Solidity compiler versions.
— **Solidity prettier plugin** Un plugin Prettier pour Solidity.
— **Solidity REPL** Essayez Solidity instantanément avec une console Solidity en ligne de commande.
— **solgraph** Visualisez le flux de contrôle Solidity et mettez en évidence les vulnérabilités potentielles en matière de sécurité.
— **Solhint** Linter Solidity qui fournit la sécurité, un guide de style et des règles de bonnes pratiques pour la validation des contrats intelligents.
— **Sūrya** Outil utilitaire pour les systèmes de contrats intelligents, offrant un certain nombre de sorties visuelles et des informations sur la structure des contrats. Il permet également d'interroger le graphe des appels de fonction.
— **Universal Mutator** Un outil pour la génération de mutations, avec des règles configurables et le support de Solidity et Vyper.

3.30.5 Analyseurs et grammaires Solidity tiers

— **Solidity Parser for JavaScript** Un analyseur Solidity pour JS construit à partir d'une grammaire ANTLR4 robuste.

3.31 Résolution du chemin d'importation

Afin de pouvoir supporter des constructions reproductibles sur toutes les plateformes, le compilateur Solidity doit faire abstraction des détails du système de fichiers où sont stockés les fichiers sources. Les chemins utilisés dans les importations doivent fonctionner de la même manière partout, tandis que l'interface de la ligne de commande doit être capable de travailler avec des chemins spécifiques à la plate-forme pour fournir une bonne expérience utilisateur. Cette section vise à expliquer en détail comment Solidity concilie ces exigences.

3.31.1 Système de fichiers virtuel

Le compilateur maintient une base de données interne (*système de fichiers virtuel* ou *VFS* en abrégé) dans laquelle chaque unité source se voit attribuer un *nom d'unité source* unique qui est un identifiant opaque et non structuré. Lorsque vous utilisez l'instruction *import*, vous spécifiez un *chemin d'accès à l'importation* qui fait référence à un nom d'unité source.

Rappel d'importation

Le VFS n'est initialement peuplé que de fichiers que le compilateur a reçus en entrée. Des fichiers supplémentaires peuvent être chargés pendant la compilation en utilisant un *import callback*, qui est différent selon le type de compilateur que vous utilisez (voir ci-dessous). Si le compilateur ne trouve pas de nom d'unité source correspondant au chemin d'importation dans le VFS, il invoque le callback, qui est chargé d'obtenir le code source à placer sous ce nom. Un callback d'importation est libre d'interpréter les noms d'unité source d'une manière arbitraire, pas seulement comme des chemins. S'il n'y a pas de callback disponible lorsqu'on en a besoin ou s'il ne parvient pas à localiser le code source, la compilation échoue.

Le compilateur en ligne de commande fournit le *Host Filesystem Loader* - un rappel rudimentaire qui interprète un nom d'unité source comme un chemin dans le système de fichiers local. L'interface JavaScript n'en fournit pas par défaut, mais un peut être fourni par l'utilisateur. Ce mécanisme peut être utilisé pour obtenir du code source à partir d'emplacements autres que le système de fichiers local (qui peut même ne pas être accessible, par exemple lorsque le compilateur est exécuté dans un navigateur). Par exemple l'IDE Remix fournit un callback polyvalent qui vous permet

d'importer des fichiers à partir d'URL HTTP, IPFS et Swarm ou de vous référer directement à des paquets dans le registre NPM.

Note : La recherche de fichiers du Host Filesystem Loader dépend de la plate-forme. Par exemple, les barres obliques inverses dans le nom d'une unité source peuvent être interprétées comme des séparateurs de répertoire ou non, et la recherche peut être sensible à la casse ou non, selon la plate-forme sous-jacente.

Pour des raisons de portabilité, il est recommandé d'éviter d'utiliser des chemins d'importation qui ne fonctionnent correctement qu'avec avec une fonction d'appel d'importation spécifique ou uniquement sur une plate-forme. Par exemple, vous devriez toujours utiliser des slashs avant car ils fonctionnent comme des séparateurs de chemin également sur plateformes qui prennent en charge les barres obliques inversées.

Contenu initial du système de fichiers virtuel

Le contenu initial du VFS dépend de la façon dont vous invoquez le compilateur :

1. **solc / command-line interface**
   Lorsque vous compilez un fichier à l'aide de l'interface de ligne de commande du compilateur, vous fournissez un ou plusieurs chemins d'accès à des fichiers contenant du code Solidity :

   ```
   solc contract.sol /usr/local/dapp-bin/token.sol
   ```

   Le nom de l'unité source d'un fichier chargé de cette façon est construit en convertissant son chemin d'accès à une forme canonique et, si possible, en le rendant relatif au chemin de base ou à l'un des chemins d'inclusion. Reportez-vous à *CLI Path Normalization and Stripping* pour une une description détaillée de ce processus.
2. **Standard JSON**
   Le nom de l'unité source d'un fichier chargé de cette façon est construit en convertissant son chemin d'accès à une forme canonique et, si possible, en le rendant relatif au chemin de base ou à l'un des chemins d'inclusion. Reportez-vous à *CLI Path Normalization and Stripping* pour une description détaillée de ce processus.

   ```
   {
       "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"]}}}
   }
   ```

   Le dictionnaire `sources` devient le contenu initial du système de fichiers virtuel et ses clés sont utilisées comme noms d'unités sources.
3. **Standard JSON (via import callback)**
   Avec Standard JSON, il est également possible d'indiquer au compilateur d'utiliser le callback d'importation pour obtenir le code source :

```
{
    "language": "Solidity",
    "sources": {
        "/usr/local/dapp-bin/token.sol": {
            "urls": [
                "/projects/mytoken.sol",
                "https://example.com/projects/mytoken.sol"
            ]
        }
    },
    "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
}
```

Si un import callback est disponible, le compilateur lui donnera les chaînes spécifiées dans `urls` une par une, jusqu'à ce qu'une soit chargée avec succès ou que la fin de la liste soit atteinte.

Les noms des unités de sources sont déterminés de la même manière que lors de l'utilisation de `content` - ce sont des clés du dictionnaire `sources` et le contenu de `urls` ne les affecte en aucune façon.

4. **Entrée standard**

   En ligne de commande, il est également possible de fournir la source en l'envoyant à l'entrée standard du compilateur :

```
echo 'import "./util.sol"; contract C {}' | solc -
```

   - utilisé comme l'un des arguments indique au compilateur de placer le contenu de l'entrée standard dans le système de fichiers virtuel sous un nom d'unité source spécial : <stdin>.

Une fois le VFS initialisé, des fichiers supplémentaires ne peuvent y être ajoutés que par le biais de la fonction import pour y ajouter des fichiers.

3.31.2 Importations

L'instruction d'importation spécifie un *chemin d'importation*. En fonction de la façon dont le chemin d'importation est spécifié, nous pouvons diviser les importations en deux catégories :

— *Imports directs*, où vous spécifiez directement le nom complet de l'unité source.

— *Relative imports*, où vous spécifiez un chemin commençant par `./` ou `../` à combiner avec le nom de l'unité source du fichier d'importation.

Code source 1 – contracts/contract.sol

```
import "./math/math.sol";
import "contracts/tokens/token.sol";
```

Dans l'exemple ci-dessus, `./math/math.sol` et `contracts/tokens/token.sol` sont des chemins d'importation alors que les noms d'unités sources vers lesquels ils sont traduits sont respectivement `contracts/math/math.sol` et `contracts/tokens/token.sol`.

Importations directes

Une importation qui ne commence pas par `./` ou `../` est une *importation directe*.

```
import "/project/lib/util.sol";         // nom de l'unité source: /project/lib/util.sol
import "lib/util.sol";                  // nom de l'unité source: lib/util.sol
import "@openzeppelin/address.sol";     // nom de l'unité source: @openzeppelin/address.
↪sol
import "https://example.com/token.sol"; // nom de l'unité source: https://example.com/
↪token.sol
```

Après avoir appliqué tout *import remappings*, le chemin d'importation devient simplement le nom de l'unité source.

Note : Le nom d'une unité source n'est qu'un identifiant et même si sa valeur ressemble à un chemin, il n'est pas soumis aux règles de normalisation que l'on peut attendre d'un shell. Tous les segments `/./` ou `../` ou les séquences de barres obliques multiples en font toujours partie. Lorsque la source est fournie via une interface JSON standard, il est tout à fait possible d'associer différents contenus à des noms d'unités de source qui feraient référence au même fichier sur le disque.

Lorsque la source n'est pas disponible dans le système de fichiers virtuel, le compilateur transmet le nom de l'unité source à l'import callback. Le Host Filesystem Loader tentera de l'utiliser comme chemin et de rechercher le fichier sur le disque. À ce stade, les règles de normalisation spécifiques à la plate-forme entrent en jeu et les noms qui étaient considérés comme différents dans le VFS peuvent en fait aboutir au chargement du même fichier. Par exemple, `/projet/lib/math.sol` et `/projet/lib/../lib///math.sol` sont considérés comme complètement différents dans le VFS même s'ils font référence au même fichier sur le disque.

Note : Même si un callback d'importation finit par charger du code source pour deux noms d'unité source différents à partir du même fichier sur le disque, le compilateur les verra toujours comme des unités sources distinctes. C'est le nom de l'unité source qui importe, pas l'emplacement physique du code.

Importations relatives

Une importation commençant par `./` ou `../` est une importation *relative*. Ces importations spécifient un chemin relatif au nom de l'unité source de l'unité source importatrice :

Code source 2 – /project/lib/math.sol

```
import "./util.sol" as util;    // nom de l'unité source: /project/lib/util.sol
import "../token.sol" as token; // nom de l'unité source: /project/token.sol
```

Code source 3 – lib/math.sol

```
import "./util.sol" as util;    // nom de l'unité source: lib/util.sol
import "../token.sol" as token; // nom de l'unité source: token.sol
```

Note : Les importations relatives commencent toujours par `./` ou `../`. `import "./util.sol"`, est une importation directe. Alors que les deux chemins seraient considérés comme relatifs dans le système de fichiers hôte, `util.sol` est en fait absolu dans le VFS.

Définissons un *segment de chemin* comme toute partie non vide du chemin qui ne contient pas de séparateur et qui est délimitée par deux séparateurs de chemin. Un séparateur est un slash avant ou le début/la fin de la chaîne. Par exemple, dans `./abc/..//`, il y a trois segments de chemin : `.`, `abc` et `..`.

Le compilateur calcule un nom d'unité source à partir du chemin d'importation de la manière suivante :

1. Un préfixe est d'abord calculé
   — Le préfixe est initialisé avec le nom de l'unité source de l'unité source importatrice.
   — Le dernier segment de chemin avec les barres obliques précédentes est supprimé du préfixe.
   — Ensuite, la partie avant du chemin d'importation normalisé, composée uniquement de caractères `/` et `.`, est prise en compte. Pour chaque segment `..` trouvé dans cette partie, le dernier segment de chemin avec les barres obliques précédant est supprimé du préfixe.
2. Ensuite, le préfixe est ajouté au chemin d'importation normalisé. Si le préfixe n'est pas vide, une seule barre oblique est insérée entre lui et le chemin d'importation.

L'élimination du dernier segment de chemin avec les barres obliques précédentes fonctionne comme suit :

1. Tout ce qui dépasse la dernière barre oblique est supprimé (c'est-à-dire que `a/b//c.sol` devient `a/b//`).
2. Toutes les barres obliques de fin de ligne sont supprimées (par exemple, `a/b//` devient `a/b`).

Les règles de normalisation sont les mêmes que pour les chemins UNIX, à savoir :
— Tous les segments internes `.` sont supprimés.
— Chaque segment interne `..` remonte d'un niveau dans la hiérarchie.
— Les slashs multiples sont écrasés en un seul.
Notez que la normalisation est effectuée uniquement sur le chemin d'importation. Le nom de l'unité source du module d'importation qui est utilisé pour le préfixe n'est pas normalisé. Cela garantit que la partie `protocol://` ne se transforme pas en `protocol:/` si le fichier d'importation est identifié par une URL.

Si vos chemins d'importation sont déjà normalisés, vous pouvez vous attendre à ce que l'algorithme ci-dessus produise des résultats très intuitifs. Voici quelques exemples de ce que vous pouvez attendre s'ils ne le sont pas :

Code source 4 – lib/src/../contract.sol

```
import "./util/./util.sol";         // nom de l'unité source: lib/src/../util/util.sol
import "./util//util.sol";          // nom de l'unité source: lib/src/../util/util.sol
import "../util/../array/util.sol"; // nom de l'unité source: lib/src/array/util.sol
import "../../../../util.sol";      // nom de l'unité source: util.sol
import "../../../../../util.sol";   // nom de l'unité source: util.sol
```

Note : L'utilisation d'importations relatives contenant des segments `..` en tête n'est pas recommandée. Le même effet peut être obtenu de manière plus fiable en utilisant des importations directes avec base path et include path.

3.31.3 Chemin de base et chemins d'inclusion

Le chemin de base et les chemins d'inclusion représentent les répertoires à partir desquels le Host Filesystem Loader chargera les fichiers. Lorsqu'un nom d'unité source est transmis au chargeur, il y ajoute en préambule le chemin de base et effectue une recherche dans le système de fichiers. Si la recherche n'aboutit pas, la même chose est faite avec tous les répertoires de la liste des chemins d'inclusion.

Il est recommandé de définir le chemin de base au répertoire racine de votre projet et d'utiliser les chemins d'inclusion pour spécifier des emplacements supplémentaires qui peuvent contenir des bibliothèques dont dépend votre projet. Cela vous permet d'importer à partir de ces bibliothèques d'une manière uniforme, peu importe où elles sont situées dans le système de fichiers par rapport à votre projet. Par exemple, si vous utilisez npm pour installer des paquets et que votre contrat importe `@openzeppelin/contracts/utils/Strings.sol`, vous pouvez utiliser ces options pour indiquer au compilateur que que la bibliothèque peut être trouvée dans l'un des répertoires de paquets npm :

```
solc contract.sol \
    --base-path . \
    --include-path node_modules/ \
    --include-path /usr/local/lib/node_modules/
```

Votre contrat sera compilé (avec les mêmes métadonnées exactes), peu importe que vous installiez la bibliothèque dans le répertoire du paquetage local ou global ou même directement sous la racine de votre projet.

Par défaut, le chemin de base est vide, ce qui laisse le nom de l'unité source inchangé. Lorsque le nom de l'unité source est un chemin relatif, cela a pour conséquence que le fichier est recherché dans le répertoire à partir duquel le compilateur a été invoqué. C'est aussi la seule valeur qui permet d'interpréter les chemins absolus dans les noms d'unités sources interprétés comme des chemins absolus sur le disque. Si le chemin de base est lui-même relatif, il est interprété comme relatif au répertoire de travail actuel du compilateur. du compilateur.

Note : Les chemins d'inclusion ne peuvent pas avoir de valeurs vides et doivent être utilisés avec un chemin de base non vide.

Note : Les chemins d'inclusion et de base peuvent se chevaucher tant que cela ne rend pas la résolution des importations ambiguë. Par exemple, vous pouvez spécifier un répertoire à l'intérieur du chemin de base comme un répertoire d'inclusion ou avoir un répertoire d'inclusion qui est un sous-répertoire d'un autre répertoire include. Le compilateur n'émettra une erreur que si le nom de l'unité source transmis au Host Filesystem Loader représente un chemin existant lorsqu'il est combiné avec plusieurs chemins d'inclusion ou un chemin d'inclusion et un chemin de base.

Normalisation et suppression des chemins CLI

Sur la ligne de commande, le compilateur se comporte comme vous le feriez avec n'importe quel autre programme : Il accepte les chemins dans un format natif de la plate-forme et les chemins relatifs sont relatifs au répertoire de travail actuel. Les noms d'unités sources attribués aux fichiers dont les chemins sont spécifiés sur la ligne de commande, cependant, ne doivent pas changer simplement parce que le projet est compilé sur une plate-forme différente ou parce que le compilateur a été invoqué à partir d'un répertoire différent. Pour cela, les chemins des fichiers sources provenant de la ligne de commande doivent être convertis en une forme canonique et, si possible, rendus relatifs au chemin de base ou à l'un des chemins d'inclusion.

Les règles de normalisation sont les suivantes :

- Si un chemin est relatif, il est rendu absolu en y ajoutant le répertoire de travail actuel.
- Les segments internes `.` et ``.`”” sont réduits.
- Les séparateurs de chemin spécifiques à la plate-forme sont remplacés par des barres obliques.

— Les séquences de plusieurs séparateurs de chemin consécutifs sont écrasées en un seul séparateur (à moins qu'il s'agisse des barres obliques de tête d'un chemin UNC).
— Si le chemin comprend un nom de racine (par exemple une lettre de lecteur sous Windows) et que la racine est la même que la racine du répertoire de travail actuel, la racine est remplacée par `/`.
— Les liens symboliques dans le chemin ne sont **pas** résolus.
  — La seule exception est le chemin d'accès au répertoire de travail actuel ajouté aux chemins relatifs dans le but de les rendre absolus. Sur certaines plateformes, le répertoire de travail est toujours signalé avec les liens symboliques résolus, donc pour des raisons de cohérence, le compilateur les résout partout.
— La casse originale du chemin est préservée même si le système de fichiers est insensible à la casse mais case-preserving et que la casse réelle sur le disque est différent.

Note : Il existe des situations où les chemins ne peuvent pas être rendus indépendants de la plate-forme. Par exemple, sous Windows, le compilateur peut éviter d'utiliser les lettres de lecteur en se référant au répertoire racine du lecteur actuel comme `/` mais les lettres de lecteur sont toujours nécessaires pour les chemins menant à d'autres lecteurs. Vous pouvez éviter de telles situations en vous assurant que tous les fichiers sont disponibles dans une seule arborescence de répertoire sur le même lecteur.

Après la normalisation, le compilateur essaie de rendre le chemin du fichier source relatif. Il essaie d'abord le chemin de base, puis les chemins d'inclusion dans l'ordre où ils ont été donnés. Si le chemin de base est vide ou non spécifié, il est traité comme s'il était égal au chemin du répertoire de travail actuel (avec tous les liens symboliques résolus). Le résultat est accepté seulement si le chemin du répertoire normalisé est le préfixe exact du chemin du fichier normalisé. Sinon, le chemin du fichier reste absolu. Cela rend la conversion non ambiguë et assure que le chemin relatif ne commence pas par `../`. Le chemin de fichier résultant devient le nom de l'unité source.

Note : Le chemin relatif produit par le dépouillement doit rester unique dans le chemin de base et les chemins d'inclusion. Par exemple, le compilateur émettra une erreur pour la commande suivante si à la fois `/projet/contract.sol` et `/lib/contract.sol` existent :

```
solc /project/contract.sol --base-path /project --include-path /lib
```

Note : Avant la version 0.8.8, la suppression des chemins d'accès de l'interface CLI n'était pas effectuée et la seule normalisation appliquée était la conversion des séparateurs de chemin. Lorsque vous travaillez avec des versions plus anciennes du compilateur, il est recommandé d'invoquer le compilateur à partir du chemin de base et de n'utiliser que des chemins relatifs sur la ligne de commande.

3.31.4 Chemins autorisés

Par mesure de sécurité, le Host Filesystem Loader refusera de charger des fichiers en dehors de quelques emplacements qui sont considérés comme sûrs par défaut :
— En dehors du mode JSON standard :
  — Les répertoires contenant les fichiers d'entrée listés sur la ligne de commande.
  — Les répertoires utilisés comme cibles *remapping*. Si la cible n'est pas un répertoire (c'est-à-dire ne se termine pas par `/`, `/.` ou `/..`), le répertoire contenant la cible est utilisé à la place.
  — Chemin de base et chemins d'inclusion.
— En mode JSON standard :
  — Le chemin de base et les chemins d'inclusion.

Des répertoires supplémentaires peuvent être mis sur une liste blanche en utilisant l'option `--allow-paths`. L'option accepte une liste de chemins séparés par des virgules :

```
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
```

Lorsque le compilateur est invoqué avec la commande indiquée ci-dessus, le Host Filesystem Loader permet d'importer des fichiers depuis les répertoires suivants :

- `/home/user/project/token/` (parce que `token/` contient le fichier d'entrée et aussi parce qu'il s'agit du chemin de base),
- `/lib/` (parce que `/lib/` est un des chemins d'inclusion),
- *`/home/user/project/libs/`* (parce que *`libs/`* est un répertoire contenant une cible de remappage),
- `/home/user/utils/` (à cause de *`../utils/`* passé à *`–allow-paths`*),
- `/tmp/libraries/` (à cause de `/tmp/libraries` passé dans *`–allow-paths`*),

Note : Le répertoire de travail du compilateur est l'un des chemins autorisés par défaut uniquement s'il se trouve être le chemin de base (ou le chemin de base n'est pas spécifié ou a une valeur vide).

Note : Le compilateur ne vérifie pas si les chemins autorisés existent réellement et s'ils sont des répertoires. Les chemins inexistants ou vides sont simplement ignorés. Si un chemin autorisé correspond à un fichier plutôt qu'à un répertoire, le fichier est également considéré comme étant sur la liste blanche.

Note : Les chemins autorisés sont sensibles à la casse, même si le système de fichiers ne l'est pas. La casse doit correspondre exactement à celle utilisée dans vos importations. Par exemple, `--allow-paths tokens` ne correspondra pas à `import "Tokens/IERC20.sol"`.

Avertissement : Les fichiers et répertoires accessibles uniquement par des liens symboliques à partir de répertoires autorisés ne sont pas automatiquement sur la liste blanche. Par exemple, si `token/contract.sol` dans l'exemple ci-dessus était en fait un lien symbolique pointant sur `/etc/passwd`, le compilateur refuserait de le charger à moins que `/etc/` ne fasse aussi partie des chemins autorisés.

3.31.5 Remappage des importations

Le remappage des importations vous permet de rediriger les importations vers un emplacement différent dans le système de fichiers virtuel. Le mécanisme fonctionne en modifiant la traduction entre les chemins d'importation et les noms d'unités sources. Par exemple, vous pouvez configurer un remappage de sorte que toute importation à partir du répertoire virtuel `github.com/ethereum/dapp-bin/library/` soit considérée comme une importation depuis `dapp-bin/library/`.

Vous pouvez limiter la portée d'un remappage en spécifiant un *contexte*. Cela permet de créer des remappages qui ne s'appliquent qu'aux importations situées dans une bibliothèque spécifique ou un fichier spécifique. Sans contexte, un remappage est appliqué à chaque import correspondant dans tous les fichiers du système de fichiers virtuel.

Les remappages d'importation ont la forme de `context:prefix=target` :

- `context` doit correspondre au début du nom de l'unité source du fichier contenant l'importation.
- `prefix` doit correspondre au début du nom de l'unité source résultant de l'importation.
- `target` est la valeur avec laquelle le préfixe est remplacé.

Par exemple, si vous clonez https://github.com/ethereum/dapp-bin/ localement dans `/projet/dapp-bin` et que vous exécutez le compilateur avec :

```
solc github.com/ethereum/dapp-bin/=dapp-bin/ --base-path /project source.sol
```

vous pouvez utiliser ce qui suit dans votre fichier source :

```
import "github.com/ethereum/dapp-bin/library/math.sol"; // source unit name: dapp-bin/
↪library/math.sol
```

Le compilateur cherchera le fichier dans le VFS sous `dapp-bin/library/math.sol`. Si le fichier n'est pas disponible à cet endroit, le nom de l'unité source sera transmis au Host Filesystem Loader, qui cherchera alors dans `/project/dapp-bin/library/iterable_mapping.sol`.

Avertissement : Les informations sur les remappages sont stockées dans les métadonnées du contrat. Comme le binaire produit par le compilateur contient un hachage des métadonnées, toute modification des réaffectations se traduira par un bytecode différent.

C'est pourquoi vous devez veiller à ne pas inclure d'informations locales dans les cibles de remappage. Par exemple, si votre bibliothèque est située dans le répertoire `/home/user/packages/mymath/math.sol`, un remappage comme `@math/=/home/user/packages/mymath/` aurait pour conséquence d'inclure votre répertoire personnel dans les métadonnées. Pour être en mesure de reproduire le même bytecode avec un tel remappage sur une autre machine, vous devrez recréer des parties de votre structure de répertoire locale dans le VFS et (si vous utilisez le Host Filesystem Loader) également dans le système de fichiers de l'hôte.

Pour éviter que votre structure de répertoire locale ne soit intégrée dans les métadonnées, il est recommandé de désigner les répertoires contenant les bibliothèques comme des *chemins d'inclusion*. Par exemple, dans l'exemple ci-dessus, `--include-path /home/user/packages/` vous permettrait d'utiliser les importations commençant par `mymath/`. Contrairement au remappage, l'option seule ne fera pas apparaître `mymath` comme `@math`, mais cela peut être réalisé en créant un lien symbolique ou en renommant le sous-répertoire du paquetage.

Pour un exemple plus complexe, supposons que vous dépendez d'un module qui utilise une ancienne version de dapp-bin que vous avez extraite vers `/project/dapp-bin_old`, alors vous pouvez exécuter :

```
solc module1:github.com/ethereum/dapp-bin/=dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=dapp-bin_old/ \
     --base-path /project \
     source.sol
```

Cela signifie que tous les imports de `module2` pointent vers l'ancienne version mais que les imports de `module1` pointent vers la nouvelle version.

Voici les règles détaillées qui régissent le comportement des remappages :

1. **Les remappages n'affectent que la traduction entre les chemins d'importation et les noms d'unités sources.**

   Les noms d'unités sources ajoutés au VFS de toute autre manière ne peuvent pas être remappés. Par exemple, les chemins que vous spécifiez sur la ligne de commande et ceux qui se trouvent dans `sources.urls` en JSON standard ne sont pas affectés.

   ```
   solc /project/=/contracts/ /project/contract.sol # source unit name: /project/
   ↪contract.sol
   ```

   Dans l'exemple ci-dessus, le compilateur chargera le code source à partir de `/project/contract.sol` et le placera sous ce nom exact d'unité source dans le VFS, et non sous `/contract/contract.sol`.

2. **Le contexte et le préfixe doivent correspondre aux noms des unités sources, et non aux chemins d'importation.**
   — Cela signifie que vous ne pouvez pas remapper `./` ou `./` directement puisqu'ils sont remplacés pendant la traduction en nom d'unité source, mais vous pouvez remapper la partie du nom par laquelle ils sont remplacés avec :

```
solc ./=a/ /project/=b/ /project/contract.sol # source unit name: /project/
↪contract.sol
```

Code source 5 – /project/contract.sol

```
import "./util.sol" as util; // source unit name: b/util.sol
```

   — Vous ne pouvez pas remapper le chemin de base ou toute autre partie du chemin qui est seulement ajouté en interne par un rappel d'importation :

```
solc /project/=/contracts/ /project/contract.sol --base-path /project # source␣
↪unit name: contract.sol
```

Code source 6 – /project/contract.sol

```
import "util.sol" as util; // source unit name: util.sol
```

3. **La cible est insérée directement dans le nom de l'unité source et ne doit pas nécessairement être un chemin d'accès valide.**
   — Il peut s'agir de n'importe quoi tant que le callback d'importation peut le gérer. Dans le cas du Host Filesystem Loader, cela inclut également les chemins relatifs. Lorsque vous utilisez l'interface JavaScript, vous pouvez même utiliser des URL et des identifiants abstraits si votre callback peut les gérer.
   — Le remappage se produit après que les importations relatives aient déjà été résolues en noms d'unités sources. Cela signifie que les cibles commençant par `./` et `./` n'ont pas de signification particulière et sont relatives au chemin de base plutôt qu'à l'emplacement du fichier source.
   — Les cibles de remappage ne sont pas normalisées, donc `@root/=./a/b//` remappera `@root/contract.sol` en `./a/b/`. vers `./a/b//contract.sol` et non `a/b/contract.sol`.
   — Si la cible ne se termine pas par un slash, le compilateur ne l'ajoutera pas automatiquement :

```
solc /project/=/contracts /project/contract.sol # source unit name: /project/
↪contract.sol
```

Code source 7 – /project/contract.sol

```
import "/project/util.sol" as util; // source unit name: /contractsutil.sol
```

4. **Le contexte et le préfixe sont des modèles et les correspondances doivent être exactes.**
   — `a//b=c` ne correspondra pas à *a/b* `.
   — Les noms des unités sources ne sont pas normalisés, donc `a/b=c` ne correspondra pas non plus à `a//b`.
   — Les parties des noms de fichiers et de répertoires peuvent également correspondre. `/newProject/con:/new=old` correspondra à `/newProject/contract.sol` et le remappera à `oldProject/contrat.sol`.
5. **Un remappage au maximum est appliqué à une seule importation.**
   — Si plusieurs réaffectations correspondent au même nom d'unité source, celle dont le préfixe est le plus long est choisi.
   — Si les préfixes sont identiques, celui qui est spécifié en dernier l'emporte.
   — Les réaffectations ne fonctionnent pas sur d'autres réaffectations. Par exemple, `a=b b=c c=d` n'aura pas pour résultat de transformer *a* ` en `d`.
6. **Le préfixe ne peut être vide, mais le contexte et la cible sont facultatifs.**
   — Si `target` est une chaîne vide, `prefix` est simplement supprimé des chemins d'importation.

— Un `context` vide signifie que le remappage s'applique à toutes les importations dans toutes les unités sources.

3.31.6 Utilisation des URLs dans les importations

La plupart des préfixes d'URL tels que `https://` ou `data://` n'ont pas de signification particulière dans les chemins d'importation. La seule exception est `file://` qui est supprimé des noms d'unités sources par le Host Filesystem Loader.

Lorsque vous compilez localement, vous pouvez utiliser le remappage d'importation pour remplacer la partie protocole et domaine par une partie chemin local :

```
solc :https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin contract.sol
```

Notez le premier `:`, qui est nécessaire lorsque le contexte de remappage est vide. Sinon, la partie `https:` serait interprétée par le compilateur comme le contexte.

3.32 Yul

Yul (précédemment aussi appelé JULIA ou IULIA) est un langage intermédiaire qui peut être compilé en bytecode pour différents backends.

Le support d'EVM 1.0, EVM 1.5 et Ewasm est prévu, et il est conçu pour être un dénominateur commun utilisable pour ces trois plateformes. Il peut déjà être utilisé en mode autonome et pour « l'assemblage en ligne » dans Solidity et il existe une implémentation expérimentale du compilateur Solidity qui utilise Yul comme langage intermédiaire. Le Yul est une bonne cible pour étapes d'optimisation de haut niveau qui peuvent bénéficier à toutes les plates-formes cibles de manière égale.

3.32.1 Motivation et description de haut niveau

La conception de Yul vise à atteindre plusieurs objectifs :

1. Les programmes écrits en Yul doivent être lisibles, même si le code est généré par un compilateur de Solidity ou d'un autre langage de haut niveau.
2. Le flux de contrôle doit être facile à comprendre pour faciliter l'inspection manuelle, la vérification formelle et l'optimisation.
3. La traduction de Yul en bytecode doit être aussi simple que possible.
4. Yul doit être adapté à l'optimisation de l'ensemble du programme.

Afin d'atteindre le premier et le second objectif, Yul fournit des constructions de haut niveau comme les boucles `for`, les instructions `if` et `switch` et les appels de fonctions. Ces éléments devraient être suffisantes pour représenter adéquatement le flux de contrôle des programmes assembleurs. Par conséquent, il n'y a pas d'instructions explicites pour `SWAP`, `DUP`, `JUMPDEST`, `JUMP` et `JUMPI` sont fournis, parce que les deux premiers obscurcissent le flux de données et les deux derniers obfusquent le flux de contrôle. De plus, les instructions fonctionnelles de la forme `mul(add(x, y), 7)` sont préférées aux instructions opcode pures telles que `7 y x add mul` car dans la première forme, il est beaucoup plus facile de voir quel opérande est utilisé pour quel opcode.

Même s'il a été conçu pour les machines à pile, Yul n'expose pas la complexité de la pile elle-même. Le programmeur ou l'auditeur ne devrait pas avoir à se soucier de la pile.

Le troisième objectif est atteint en compilant les constructions de niveau supérieur en bytecode de manière très régulière. La seule opération non-locale effectuée par l'assembleur est la recherche de noms d'identifiants définis par l'utilisateur (fonctions, variables, ...) et le nettoyage des variables locales de la pile.

Pour éviter les confusions entre des concepts comme les valeurs et les références, Yul est typée statiquement. En même temps, il existe un type par défaut (généralement le mot entier de la machine cible) qui peut toujours être omis pour faciliter la lisibilité.

Pour garder le langage simple et flexible, Yul n'a pas d'opérations, de fonctions ou de types intégrés dans sa forme pure. Ceux-ci sont ajoutés avec leur sémantique lors de la spécification d'un dialecte de Yul, ce qui permet de spécialiser Yul pour répondre aux exigences de différentes plateformes et ensembles de fonctionnalités cibles.

Actuellement, il n'existe qu'un seul dialecte spécifié de Yul. Ce dialecte utilise les opcodes EVM en tant que fonctions intégrées (voir ci-dessous) et ne définit que le type `u256`, qui est le type natif 256-bit de l'EVM. Pour cette raison, nous ne fournirons pas de types dans les exemples ci-dessous.

3.32.2 Exemple simple

Le programme d'exemple suivant est écrit dans le dialecte EVM et calcule l'exponentiation. Il peut être compilé en utilisant `solc --strict-assembly`. Les fonctions intégrées `mul` et `div` calculent le produit et la division, respectivement.

```
{
    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) }
        }
    }
}
```

Le programme d'exemple suivant est écrit dans le dialecte EVM et calcule l'exponentiation. Il peut être compilé en utilisant `solc --strict-assembly`. Les fonctions intégrées `mul` et `div` calculent le produit et la division, respectivement.

```
{
    function power(base, exponent) -> result
    {
        result := 1
        for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
        {
            result := mul(result, base)
        }
    }
}
```

À la *fin de la section*, une implémentation complète du standard de la norme ERC-20 peut être trouvée.

3.32.3 Utilisation autonome

Vous pouvez utiliser Yul sous sa forme autonome dans le dialecte EVM en utilisant le compilateur Solidity. Il utilisera la notation d'objet *Yul* afin qu'il soit possible de se référer au code comme à des données pour déployer des contrats. Ce mode Yul est disponible pour le compilateur en ligne de commande (utilisez `--strict-assembly`) et pour l'interface *standard-json* :

```
{
    "language": "Yul",
    "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
    "settings": {
        "outputSelection": { "*": { "*": ["*"], "": [ "*" ] } },
        "optimizer": { "enabled": true, "details": { "yul": true } }
    }
}
```

Avertissement : Yul est en cours de développement actif et la génération de bytecode n'est entièrement implémentée que pour le dialecte EVM de Yul avec EVM 1.0 comme cible.

3.32.4 Description informelle de Yul

Dans ce qui suit, nous allons parler de chaque aspect individuel du langage Yul. Dans les exemples, nous utiliserons le dialecte EVM par défaut.

Syntaxe

Yul analyse les commentaires, les littéraux et les identifiants de la même manière que Solidity, donc vous pouvez par exemple utiliser `//` et `/* */` pour désigner des commentaires. Il y a une exception : Les identificateurs dans Yul peuvent contenir des points : `..`

Yul peut spécifier des « objets » qui se composent de code, de données et de sous-objets. Veuillez consulter *Yul Objects* ci-dessous pour plus de détails à ce sujet. Dans cette section, nous ne sommes concernés que par la partie code d'un tel objet. Cette partie code consiste toujours en un bloc délimité par des accolades. La plupart des outils supportent la spécification d'un seul bloc de code où un objet est attendu.

Inside a code block, the following elements can be used (see the later sections for more details) :

- des littéraux, par exemple `0x123`, `42` ou `"abc"` (chaînes de caractères jusqu'à 32 caractères)
- les appels à des fonctions intégrées, par exemple `add(1, mload(0))`
- les déclarations de variables, par exemple `let x := 7`, « let x := add(y, 3)`` ou `let x` (la valeur initiale de 0 est attribuée)
- des identificateurs (variables), par exemple `add(3, x)`
- des affectations, par exemple `x := add(y, 3)`
- les blocs à l'intérieur desquels les variables locales ont une portée, par exemple `{ let x := 3 { let y := add(x, 1) } }`
- les instructions if, par exemple `if lt(a, b) { sstore(0, 1) }`
- les instructions switch, par exemple : `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- Boucles for, par exemple : `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- des définitions de fonctions, par exemple : `fonction f(a, b) -> c { c := add(a, b) }`

Plusieurs éléments syntaxiques peuvent se succéder en étant simplement séparés par un espace, c'est-à-dire qu'il n'est pas nécessaire de mettre un `;` ou un saut de ligne à la fin.

Littéraux

En tant que littéraux, vous pouvez utiliser :
- Des constantes entières en notation décimale ou hexadécimale.
- Des chaînes ASCII (par exemple, `"abc"`), qui peuvent contenir des échappatoires hexagonales `xNN` et des échappatoires Unicode `uNNNN` où `N` sont des chiffres hexadécimaux.
- Chaînes hexadécimales (par exemple, `hex "616263"`).

Dans le dialecte EVM de Yul, les littéraux représentent des mots de 256 bits comme suit :
- Les constantes décimales ou hexadécimales doivent être inférieures à `2**256`. Elles représentent le mot de 256 bits avec cette valeur comme un entier non signé en codage big endian.
- Une chaîne de caractères ASCII est d'abord vue comme une séquence d'octets, en voyant un caractère ASCII non échappé comme un seul octet dont la valeur est le code ASCII, un caractère d'échappement `\xNN` comme un octet unique ayant cette valeur, et un échappement `uNNNN` comme la séquence d'octets UTF-8 pour ce point de code. La séquence d'octets ne doit pas dépasser 32 octets. La séquence d'octets est complétée par des zéros sur la droite pour atteindre une longueur de 32 octets ; En d'autres termes, la chaîne est stockée alignée à gauche. La séquence d'octets remplie représente un mot de 256 bits dont les 8 bits les plus significatifs sont les uns du premier octet, c'est-à-dire que les octets sont interprétés sous la forme big endian.
- Une chaîne hexadécimale est d'abord considérée comme une séquence d'octets, en regardant chaque paire de chiffres hexadécimaux contigus comme un octet. La séquence d'octets ne doit pas dépasser 32 octets (c'est-à-dire 64 chiffres hexadécimaux) et est traitée comme ci-dessus.

Lors de la compilation pour l'EVM, ceci sera traduit en une instruction `PUSHi` appropriée. Dans l'exemple suivant, 3 et 2 sont additionnés, ce qui donne 5. avec la chaîne « abc » est calculée. La valeur finale est affectée à une variable locale appelée `x`.

La limite de 32 octets ci-dessus ne s'applique pas aux chaînes de caractères passées aux fonctions intégrées qui requièrent des arguments littéraux (par exemple, `setimmutable` ou ``loadimmutable`"). Ces chaînes de caractères ne se retrouvent jamais dans le dans le bytecode généré.

```
let x := and("abc", add(3, 2))
```

À moins qu'il ne s'agisse du type par défaut, le type d'un littéral doit être spécifié après un deux-points :

```
// Cela ne compilera pas (les types u32 et u256 ne sont pas encore implémentés).
let x := and("abc":u32, add(3:u256, 2:u256))
```

Appels de fonction

Les fonctions intégrées et les fonctions définies par l'utilisateur (voir ci-dessous) peuvent être appelées de la même manière que dans l'exemple précédent. Si la fonction renvoie une seule valeur, elle peut être directement utilisée à l'intérieur d'une expression. Si elle renvoie plusieurs valeurs, elles doivent être assignées à des variables locales.

```
function f(x, y) -> a, b { /* ... */ }
mstore(0x80, add(mload(0x80), 3))
// Ici, la fonction définie par l'utilisateur `f` renvoie deux valeurs.
let x, y := f(1, mload(0))
```

Pour les fonctions intégrées de l'EVM, les expressions fonctionnelles peuvent être directement traduites en un flux d'opcodes : Il suffit de lire l'expression de droite à gauche pour obtenir les opcodes. Dans le cas de la première ligne de l'exemple, il s'agit de `PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE`.

Pour les appels aux fonctions définies par l'utilisateur, les arguments sont également placés sur la pile de droite à gauche et c'est dans cet ordre dans lequel les listes d'arguments sont évaluées. Les valeurs de retour, par contre, sont attendues sur la pile de gauche à droite, c'est-à-dire que dans cet exemple, `y` est en haut de la pile et `x` est en dessous.

Déclarations de variables

Vous pouvez utiliser le mot-clé `let` pour déclarer des variables. Une variable n'est visible qu'à l'intérieur du bloc `{..}` dans lequel elle a été définie. Lors de la compilation vers l'EVM, un nouvel emplacement de pile est créé, qui est réservé pour la variable et est automatiquement supprimé lorsque la fin du bloc est atteinte. Vous pouvez fournir une valeur initiale pour la variable. Si vous ne fournissez pas de valeur, la variable sera initialisée à zéro.

Comme les variables sont stockées sur la pile, elles n'ont pas d'influence directe sur la mémoire ou le stockage, mais elles peuvent être utilisées comme pointeurs vers des emplacements de mémoire ou de stockage dans les fonctions intégrées `mstore`, `mload`, `sstore` et `sload`. De futurs dialectes pourraient introduire des types spécifiques pour ces pointeurs.

Quand une variable est référencée, sa valeur actuelle est copiée. Pour l'EVM, cela se traduit par une instruction `DUP`.

```
{
    let zero := 0
    let v := calldataload(zero)
    {
        let y := add(sload(v), 1)
        v := y
    } // y est "désalloué" ici
    sstore(v, zero)
} // v et zéro sont "désalloués" ici
```

Si la variable déclarée doit avoir un type différent du type par défaut, vous l'indiquez en suivant les deux points. Vous pouvez également déclarer plusieurs variables dans une déclaration lorsque vous effectuez une assignation à partir d'un appel de fonction qui renvoie plusieurs valeurs.

```
// Cela ne compilera pas (les types u32 et u256 ne sont pas encore implémentés).
{
    let zero:u32 := 0:u32
    let v:u256, t:u32 := f()
    let x, y := g()
}
```

Selon les paramètres de l'optimiseur, le compilateur peut libérer les emplacements de pile déjà après que la variable ait été utilisée pour pour la dernière fois, même si elle est encore dans la portée.

Affectations

Les variables peuvent être assignées après leur définition en utilisant l'opérateur `:=`. Il est possible d'affecter plusieurs variables en même temps. Pour cela, le nombre et le type des valeurs doivent correspondre. Si vous voulez affecter les valeurs renvoyées par une fonction qui a plusieurs paramètres de retour, vous devez fournir plusieurs variables. La même variable ne peut pas apparaître plusieurs fois dans la partie gauche d'une une affectation, par exemple : `x, x := f()` n'est pas valide.

```
let v := 0
// réassignation de v
v := 2
let t := add(v, 2)
function f() -> a, b { }
// assigner des valeurs multiples
v, t := f()
```

If

L'instruction if peut être utilisée pour exécuter du code de manière conditionnelle. Aucun bloc « else » ne peut être défini. Envisagez d'utiliser « switch » à la place (voir ci-dessous) si vous avez besoin de plusieurs alternatives.

```
if lt(calldatasize(), 4) { revert(0, 0) }
```

Les accolades pour le corps sont nécessaires.

Interrupteur

Vous pouvez utiliser une instruction switch comme une version étendue de l'instruction if. Elle prend la valeur d'une expression et la compare à plusieurs constantes littérales. La branche correspondant à la constante correspondante est prise. Contrairement aux autres langages de programmation, le flux de contrôle ne se poursuit pas d'un cas à l'autre. Il peut y avoir un cas de repli ou par défaut appelé `default` qui est pris si aucune des constantes littérales ne correspond.

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

La liste des cas n'est pas entourée d'accolades, mais le corps d'un cas en a besoin.

Boucles

Yul supporte les boucles for qui consistent en un en-tête contenant une partie d'initialisation, une condition, une partie de post-itération et un corps. La condition doit être une expression, tandis que les trois autres sont des blocs. Si la partie d'initialisation déclare des variables au niveau supérieur, la portée de ces variables s'étend à toutes les autres parties de la boucle.

Les instructions `break` et `continue` peuvent être utilisées dans le corps de la boucle pour en sortir ou passer à la partie suivante, respectivement.

L'exemple suivant calcule la somme d'une zone en mémoire.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

Les boucles for peuvent également être utilisées en remplacement des boucles while : Il suffit de laisser les parties d'initialisation et de post-itération vides.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {     // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

Déclarations de fonctions

Yul permet de définir des fonctions. Celles-ci ne doivent pas être confondues avec les fonctions dans Solidity, car elles ne font jamais partie d'une interface externe d'un contrat et font partie d'un espace de noms distinct de celui des fonctions Solidity.

Pour l'EVM, les fonctions Yul prennent leurs arguments (et un PC de retour) de la pile et mettent également les résultats sur la pile. Les fonctions définies par l'utilisateur et les fonctions intégrées sont appelées exactement de la même manière.

Les fonctions peuvent être définies n'importe où et sont visibles dans le bloc dans lequel elles sont déclarées. À l'intérieur d'une fonction, vous ne pouvez pas accéder aux variables locales définies en dehors de cette fonction.

Les fonctions déclarent des paramètres et renvoient des variables, comme dans Solidity. Pour retourner une valeur, vous l'affectez à la ou aux variables de retour.

Si vous appelez une fonction qui renvoie plusieurs valeurs, vous devez les affecter à plusieurs variables en utilisant `a, b := f(x)` ou `let a, b := f(x)`.

L'instruction `leave` peut être utilisée pour quitter la fonction en cours. Elle fonctionne comme l'instruction `return` dans d'autres langages, mais elle ne prend pas de valeur à retourner, elle quitte juste la fonction et la fonction retournera les valeurs qui sont actuellement assignées à la ou aux variables de retour.

Notez que le dialecte EVM a une fonction intégrée appelée `return` qui quitte le contexte d'exécution complet (appel de message interne) et non pas seulement la fonction yul courante.

L'exemple suivant implémente la fonction puissance par carré et multiplication.

```
{
    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) }
        }
    }
}
```

3.32.5 Spécification de Yul

Ce chapitre décrit le code Yul de manière formelle. Le code Yul est généralement placé à l'intérieur d'objets Yul, qui sont expliqués dans leur propre chapitre.

```
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 =
    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 sur la grammaire

En dehors de celles qui sont directement imposées par la grammaire, les restrictions suivantes s'appliquent :

Les commutateurs doivent avoir au moins un cas (y compris le cas par défaut). Toutes les valeurs de cas doivent avoir le même type et des valeurs distinctes. Si toutes les valeurs possibles du type d'expression sont couvertes, un cas par défaut n'est pas autorisé (par exemple, un commutateur avec une expression `bool` qui a à la fois un cas vrai et un cas faux ne permet pas de cas par défaut).

Chaque expression est évaluée à zéro ou plusieurs valeurs. Identificateurs et littéraux évaluent à exactement une valeur et les appels de fonction sont évalués à un nombre de valeurs égal au nombre de variables de retour de la fonction appelée.

Dans les déclarations de variables et les affectations, l'expression de droite (si elle est présente) doit être évaluée sur un nombre de valeurs égal au nombre de variables du côté gauche. C'est la seule situation dans laquelle une expression évaluant à plus d'une valeur est autorisée. Le même nom de variable ne peut pas apparaître plus d'une fois dans la partie gauche d'une affectation ou d'une déclaration de variable.

Les expressions qui sont également des instructions (c'est-à-dire au niveau du bloc) doivent être évaluées à des valeurs nulles.

Dans toutes les autres situations, les expressions doivent être évaluées à une seule valeur.

Une instruction `continue` ou `break` ne peut être utilisée que dans le corps d'une boucle for, comme suit. Considérez la boucle la plus interne qui contient l'instruction. La boucle et l'instruction doivent être dans la même fonction, ou les deux doivent être au niveau supérieur. L'instruction doit se trouver dans le bloc de corps de la boucle ; elle ne peut pas se trouver dans le bloc d'initialisation ou le bloc de mise à jour de la boucle. Il est important de souligner que cette restriction ne s'applique que à la boucle la plus interne qui contient l'instruction `continue` ou `break` : cette boucle la plus interne, et donc l'instruction `continue` ou `break`, peut apparaître n'importe où dans une boucle externe, éventuellement dans le bloc d'initialisation ou le bloc de mise à jour d'une boucle externe. Par exemple, ce qui suit est légal, car l'instruction `break` apparaît dans le bloc body de la boucle interne, bien qu'elle apparaisse également dans le bloc de mise à jour de la boucle externe :

```
for {} true { for {} true {} { break } }
{
}
```

La partie condition de la boucle for doit être évaluée à une seule valeur.

L'instruction `leave` ne peut être utilisée qu'à l'intérieur d'une fonction.

Les fonctions ne peuvent pas être définies n'importe où dans les blocs d'init de la boucle for.

Les littéraux ne peuvent pas être plus grands que leur type. Le plus grand type défini est d'une largeur de 256 bits.

Pendant les affectations et les appels de fonction, les types des valeurs respectives doivent correspondre. Il n'y a pas de conversion de type implicite. La conversion de type en général ne peut être réalisée que si le dialecte fournit une fonction intégrée appropriée qui prend une valeur d'un type et retourne une valeur d'un type différent.

Règles de scoping

Dans Yul, les champs d'application sont liés aux blocs (à l'exception des fonctions et de la boucle for comme expliqué ci-dessous) et toutes les déclarations (`FunctionDefinition`, `VariableDeclaration`) introduisent de nouveaux identifiants dans ces champs d'application.

Les identificateurs sont visibles dans le bloc dans lequel ils sont définis (y compris tous les sous-noeuds et sous-blocs) : Les fonctions sont visibles dans tout le bloc (même avant leurs définitions) alors que les variables ne sont visibles qu'à partir de la déclaration qui suit la `VariableDeclaration`.

En particulier, variables ne peuvent pas être référencées dans la partie droite de leur propre déclaration de variable. Les fonctions peuvent être référencées dès avant leur déclaration (si elles sont visibles).

En tant qu'exception à la règle générale de délimitation, la portée de la partie « init » de la boucle for (le premier bloc) s'étend à toutes les autres parties de la boucle for. Cela signifie que les variables (et les fonctions) déclarées dans la partie init (mais pas dans un bloc à l'intérieur de la partie init) sont visibles dans toutes les autres parties de la boucle for.

Les identificateurs déclarés dans les autres parties de la boucle for respectent les règles syntaxiques de scoping.

Cela signifie qu'une boucle for de la forme `for { I... } C { P... } { B... }` est équivalent à `I... for {} C { P... } { B... } }`.

Les paramètres et les paramètres de retour des fonctions sont visibles dans le corps de la fonction et leurs noms doivent être distincts.

À l'intérieur des fonctions, il n'est pas possible de référencer une variable qui a été déclarée en dehors de cette fonction.

L'ombrage est interdit, c'est-à-dire que vous ne pouvez pas déclarer un identificateur à un endroit où un autre identificateur portant le même nom est également visible, même s'il n'est pas possible de le référencer parce qu'il a été déclaré en dehors de la fonction courante.

Spécification formelle

Nous spécifions formellement Yul en fournissant une fonction d'évaluation E surchargée sur les différents nœuds de l'AST. Comme les fonctions intégrées peuvent avoir des effets secondaires, E prend deux objets d'état et le noeud AST et retourne deux nouveaux objets d'état et un nombre variable d'autres valeurs. Les deux objets d'état sont l'objet d'état global (qui, dans le contexte de l'EVM, est la mémoire, le stockage et l'état de la blockchain) et l'objet d'état local (l'état des variables locales, c'est-à-dire un segment de la pile dans l'EVM).

Si le noeud AST est une déclaration, E retourne les deux objets d'état et un « mode », qui est utilisé pour les instructions `break`, `continue`` ' et ``leave`. Si le noeud de l'AST est une expression, E retourne les deux objets d'état et autant de valeurs que l'expression en évalue.

La nature exacte de l'état global n'est pas spécifiée dans cette description de haut niveau. L'état local `L` est une correspondance entre les identifiants `i` et les valeurs `v`, noté `L[i] = v`.

Pour un identifiant `v`, on note `$v` le nom de l'identifiant.

Nous utiliserons une notation de déstructuration pour les noeuds de l'AST.

```
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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
            E(G1, L1, St2, ..., Stn)
        otherwise
            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
    G, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L, mode = E(G, L, i1, ..., in)
        // le mode doit être régulier ou congé en raison des restrictions syntaxiques
        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:
                    G2, 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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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) =
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // Evaluer les littéraux, le contexte n'a pas d'importance.
    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
```

Dialecte EVM

Le dialecte par défaut de Yul est actuellement le dialecte EVM avec une version de l'EVM. Le seul type disponible dans ce dialecte est `u256`, le type natif 256 bits de la machine virtuelle Ethereum. Comme il s'agit du type par défaut de ce dialecte, il peut être omis.

Le tableau suivant liste toutes les fonctions intégrées (selon la version de la machine virtuelle Ethereum) et fournit une brève description de la sémantique de la fonction / opcode. Ce document ne veut pas être une description complète de la machine virtuelle Ethereum. Veuillez vous référer à un autre document si vous êtes intéressé par la sémantique précise.

Les opcodes marqués avec `-` ne retournent pas de résultat et tous les autres retournent exactement une valeur. Les opcodes marqués par `F`, `H`, `B`, `C`, `I` et `L` sont présents depuis Frontier, Homestead, Byzance, Constantinople, Istanbul ou Londres respectivement.

Dans ce qui suit, `mem[a...b]` signifie les octets de mémoire commençant à la position *a* ` et allant jusqu'à mais sans inclure la position `b` et `storage[p]` signifie le contenu de la mémoire à l'emplacement `p`.

Puisque Yul gère les variables locales et le flux de contrôle, les opcodes qui interfèrent avec ces fonctionnalités ne sont pas disponibles. Ceci inclut les instructions `dup` et `swap` ainsi que les instructions `jump`, les labels et les instructions `push`.

Instruction			Explication
stop()	-	F	arrête l'exécution, iden- tique à 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 ou 0 if y == 0
sdiv(x, y)		F	x / y, pour les nombres signés en complément à deux, 0 if y == 0
mod(x, y)		F	x % y, 0 if y == 0
smod(x, y)		F	x % y, pour les nombres signés en complément à deux, 0 if y == 0
exp(x, y)		F	x au pouvoir de y
not(x)		F	bitwise « not » of x (chaque bit de x est annulé)
lt(x, y)		F	1 if x < y, 0 sinon
gt(x, y)		F	1 if x > y, 0 sinon
slt(x, y)		F	1 if x < y, 0 sinon, pour les nombres signés en complément à deux
sgt(x, y)		F	1 if x > y, 0 sinon, pour les nombres signés en complément à deux
eq(x, y)		F	1 if x == y, 0 sinon
iszero(x)		F	1 if x == 0, 0 sinon
and(x, y)		F	par bit « and » of x et y
or(x, y)		F	par bit « or » of x et y
xor(x, y)		F	par bit « xor » of x et y
byte(n, x)		F	le nième octet de x, où l'octet le plus significatif est le 0ième octet
shl(x, y)		C	décalage logique à gauche de y par x bits
shr(x, y)		C	décalage logique vers la droite de y par x bits
sar(x, y)		C	décalage arithmétique signé vers la droite de y par x bits
addmod(x, y, m)		F	(x + y) % m avec une précision arithmétique arbitraire, 0 if m == 0
mulmod(x, y, m)		F	(x * y) % m avec une précision arithmétique arbitraire, 0 if m == 0
signextend(i, x)		F	le signe s'étend du (i*8+7)ème bit en comptant à partir du moins significatif
keccak256(p, n)		F	keccak(mem[p…(p+n)))
pc()		F	position actuelle dans le code
pop(x)	-	F	valeur de rejet 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 (ne modifie qu'un seul octet)
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
msize()		F	taille de la mémoire, c.à.d l'indice de mémoire le plus important auquel on accède
gas()		F	gaz encore disponible pour l'exécution
address()		F	adresse du contrat actuel / contexte d'exécution
balance(a)		F	wei balance à l'adresse a
selfbalance()		I	équivalent à balance(address()), mais moins cher
caller()		F	expéditeur de l'appel (à l'exclusion de « delegatecall »)
callvalue()		F	wei envoyé avec l'appel en cours
calldataload(p)		F	données d'appel à partir de la position p (32 octets)
calldatasize()		F	taille des données d'appel en octets

Instruction			Explication
calldatacopy(t, f, s)	-	F	copier s octets de calldata à la position f vers mem à la position t
codesize()		F	taille du code du contrat / contexte d'exécution actuel
codecopy(t, f, s)	-	F	copier s octets du code à la position f vers la mémoire à la position t
extcodesize(a)		F	taille du code à l'adresse a
extcodecopy(a, t, f, s)	-	F	comme codecopy(t, f, s) mais prendre le code à l'adresse a
returndatasize()		B	taille de la dernière donnée retournée
returndatacopy(t, f, s)	-	B	copier s octets de returndata à la position f vers mem à la position t
extcodehash(a)		C	code de hachage de l'adresse a
create(v, p, n)		F	créer un nouveau contrat avec le code mem[p…(p+n)) et envoyer v wei et renvoie
create2(v, p, n, s)		C	créer un nouveau contrat avec le code mem[p…(p+n)) à l'adresse keccak256(0xff
call(g, a, v, in, insize, out, outsize)		F	appeler le contrat à l'adresse a avec l'entrée mem[in…(in+insize)) fournir g gaz e
callcode(g, a, v, in, insize, out, outsize)		F	identique à `call` mais n'utilise que le code de a et reste dans le contexte du contra
delegatecall(g, a, in, insize, out, outsize)		H	identique à `callcode` mais conserve aussi `caller`. et `callvalue` *Voir plus*
staticcall(g, a, in, insize, out, outsize)		B	identique à `call(g, a, 0, in, insize, out, outsize)` mais font ne pas a
return(p, s)	-	F	fin de l'exécution, retour des données mem[p…(p+s))
revert(p, s)	-	B	terminer l'exécution, annuler les changements d'état, retourner les données mem[
selfdestruct(a)	-	F	mettre fin à l'exécution, détruire le contrat en cours et envoyer les fonds à un orga
invalid()	-	F	terminer l'exécution avec une instruction invalide
log0(p, s)	-	F	journal sans sujets et données mem[p…(p+s))
log1(p, s, t1)	-	F	journal avec sujet t1 et données mem[p…(p+s))
log2(p, s, t1, t2)	-	F	journal avec les sujets t1, t2 et les données mem[p…(p+s))
log3(p, s, t1, t2, t3)	-	F	journal avec les sujets t1, t2, t3 et les données mem[p…(p+s))
log4(p, s, t1, t2, t3, t4)	-	F	journal avec les sujets t1, t2, t3, t4 et les données mem[p…(p+s))
chainid()		I	ID de la chaîne d'exécution (EIP-1344)
basefee()		L	les frais de base du bloc actuel (EIP-3198 et EIP-1559)
origin()		F	émetteur de la transaction
gasprice()		F	prix du gaz de la transaction
blockhash(b)		F	hash du bloc nr b - uniquement pour les 256 derniers blocs, à l'exclusion du bloc a
coinbase()		F	bénéficiaire actuel de l'exploitation minière
timestamp()		F	Horodatage du bloc actuel en secondes depuis l'époque.
number()		F	numéro du bloc actuel
difficulty()		F	difficulté du bloc actuel
gaslimit()		F	limite de gaz du bloc en cours

Note : Les instructions `call*` utilisent les paramètres `out` et `outsize` pour définir une zone de mémoire où les données de retour ou d'échec sont placées. Cette zone est écrite en fonction du nombre d'octets que le contrat appelé renvoie. S'il retourne plus de données, seuls les premiers octets `outsize` sont écrits. Vous pouvez accéder au reste des données en utilisant l'opcode `returndatacopy`. S'il retourne moins de données, les octets restants ne sont pas touchés du tout. Vous devez utiliser l'opcode ``returndatasize`` pour vérifier quelle partie de cette zone mémoire contient les données retournées. Les autres octets conserveront leurs valeurs d'avant l'appel.

Dans certains dialectes internes, il existe des fonctions supplémentaires :

datasize, dataoffset, datacopy

Les fonctions `datasize(x)`, `dataoffset(x)` et `datacopy(t, f, l)` sont utilisées pour accéder à d'autres parties d'un objet Yul.

`datasize` et `dataoffset` ne peuvent prendre que des chaînes de caractères (les noms d'autres objets) comme arguments et renvoient respectivement la taille et le décalage dans la zone de données. Pour l'EVM, la fonction `datacopy` est équivalente à `codecopy`.

setimmutable, loadimmutable

Les fonctions `setimmutable(offset, "name", value)` et `loadimmutable("name")` sont utilisées pour le mécanisme d'immuabilité de Solidity et ne sont pas adaptées à Yul. L'appel à `setimmutable(offset, "name", value)` suppose que le code d'exécution du contrat contenant l'immuable donné a été copié en mémoire à l'offset `offset` et écrira `value` à toutes les positions en mémoire (par rapport à `offset`') qui contiennent le placeholder généré pour les appels à ``loadimmutable("name")` dans le code d'exécution.

linkersymbol

La fonction `linkersymbol("library_id")` est un espace réservé pour un littéral d'adresse à substituer par l'éditeur de liens. Son premier et seul argument doit être une chaîne de caractères et représente de manière unique l'adresse à insérer. Les identifiants peuvent être arbitraires mais lorsque le compilateur produit du code Yul à partir de sources Solidity, il utilise un nom de bibliothèque qualifié avec le nom de l'unité source qui définit cette bibliothèque. Pour lier le code avec une adresse de bibliothèque particulière, le même identifiant doit être fourni à la commande `--libraries` sur la ligne de commande.

Par exemple, ce code

```
let a := linkersymbol("file.sol:Math")
```

est équivalent à

```
let a := 0x1234567890123456789012345678901234567890
```

lorsque le linker est invoqué avec l'option `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890`.

Voir *Utilisation du compilateur en ligne de commande* pour plus de détails sur l'éditeur de liens Solidity.

memoryguard

Cette fonction est disponible dans le dialecte EVM avec des objets. L'appelant de `let ptr := memoryguard(size)` (où `size` doit être un nombre littéral) promet qu'il n'utilisera la mémoire que dans l'intervalle `[0, size)` ou dans l'intervalle non borné commençant à `ptr`.

Puisque la présence d'un appel `memoryguard` indique que tous les accès à la mémoire adhère à cette restriction, il permet à l'optimiseur d'effectuer des étapes d'optimisation supplémentaires, par exemple l'évasion de la limite de la pile, qui tente de déplacer les les variables de la pile qui seraient autrement inaccessibles à la mémoire.

L'optimiseur Yul promet de n'utiliser que la plage de mémoire `[size, ptr)` pour ses besoins. Si l'optimiseur n'a pas besoin de réserver de la mémoire, il considère que `ptr == size`.

`memoryguard` peut être appelé plusieurs fois, mais doit avoir le même littéral comme argument dans un seul sous-objet Yul. Si au moins un appel `memoryguard` est trouvé dans un sous-objet, les étapes supplémentaires d'optimisation seront exécutées sur lui.

verbatim

L'ensemble des fonctions intégrées `verbatim...` vous permet de créer du bytecode pour des opcodes qui ne sont pas connus du compilateur Yul. Il vous permet également de créer séquences de bytecode qui ne seront pas modifiées par l'optimiseur.

Les fonctions sont `verbatim_<n>i_<m>o("<data>", ...)`, où

- `n` est une valeur décimale comprise entre 0 et 99 qui spécifie le nombre d'emplacements de pile / variables d'entrée
- `m` est une décimale entre 0 et 99 qui spécifie le nombre d'emplacements de pile / variables de sortie
- `data` est une chaîne littérale qui contient la séquence d'octets.

Si vous voulez, par exemple, définir une fonction qui multiplie par deux, sans que l'optimiseur ne touche à la constante deux, vous pouvez utiliser

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

Ce code résultera en un opcode `dup1` pour récupérer `x`. (l'optimiseur pourrait réutiliser directement le résultat de l'opcode `calldataload`, cependant) directement suivi de `600202`. Le code est supposé consommer la valeur copiée de `x` et de produire le résultat en haut de la pile. Le compilateur génère alors du code pour allouer un slot de pile pour `double` et y stocker le résultat.

Comme avec tous les opcodes, les arguments sont disposés sur la pile avec l'argument le plus à gauche en haut, tandis que les valeurs de retour sont supposées être disposées de telle sorte que la variable la plus à droite se trouve en haut de la pile.

Puisque `verbatim` peut être utilisé pour générer des opcodes arbitraires ou même des opcodes inconnus du compilateur Solidity, il faut être prudent lorsqu'on utilise `verbatim` avec l'optimiseur. Même lorsque l'optimiseur est désactivé, le générateur de code doit déterminer la disposition de la pile, ce qui signifie que, par exemple, l'utilisation de `verbatim` pour modifier la hauteur de la pile peut conduire à un comportement non défini.

La liste suivante est une liste non exhaustive des restrictions sur le bytecode verbatim qui ne sont pas vérifiées par le compilateur. La violation de ces restrictions peut entraîner un comportement non défini.

- Le flux de contrôle ne doit pas sauter dans ou hors des blocs verbatim, mais il peut sauter à l'intérieur d'un même bloc verbatim
- Le contenu des piles, hormis les paramètres d'entrée et de sortie ne doit pas être accessible
- La différence de hauteur de la pile doit être exactement `m - n` (emplacements de sortie moins emplacements d'entrée)
- Le bytecode verbatim ne peut pas faire d'hypothèses sur le bytecode environnant. Tous les paramètres requis doivent être passés en tant que variables de pile

L'optimiseur n'analyse pas le bytecode verbatim et suppose toujours qu'il modifie tous les aspects de l'état et peut donc seulement faire que très peu d'optimisations à travers les appels de fonction `verbatim`.

L'optimiseur traite le bytecode verbatim comme un bloc de code opaque. Il ne le divise pas, mais peut le déplacer, le dupliquer ou le combiner avec des blocs de bytecode verbatim identiques. Si un bloc de bytecode verbatim est inaccessible par le flux de contrôle, il peut être supprimé.

Avertissement : Pendant les discussions sur le fait que les améliorations de l'EVM ne risquent pas de casser les contrats intelligents existants, les caractéristiques de `verbatim` ne peuvent pas recevoir la même considération que celles utilisées par le compilateur Solidity lui-même.

Note : Pour éviter toute confusion, tous les identificateurs commençant par la chaîne `verbatim` sont réservés et ne peuvent pas être utilisés pour des identificateurs définis par l'utilisateur.

3.32.6 Spécification de l'objet Yul

Les objets Yul sont utilisés pour regrouper des sections de code et de données nommées. Les fonctions `datasize`, `dataoffset` et `datacopy` peuvent être utilisées pour accéder à ces sections à partir du code. Les chaînes hexadécimales peuvent être utilisées pour spécifier des données en codage hexadécimal, les chaînes régulières en codage natif. Pour le code, `datacopy` accédera à sa représentation binaire assemblée.

```
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\\] | '\\' .)* '"'
```

Ci-dessus, `Block` fait référence à `Block` dans la grammaire de code Yul expliquée dans le chapitre précédent.

Note : Les objets de données ou les sous-objets dont le nom contient un `.` peuvent être définis mais il n'est pas possible d'y accéder via `datasize`, `dataoffset` ou `datacopy` parce que `.` est utilisé comme un séparateur pour accéder à des objets à l'intérieur d'un autre objet.

Note : L'objet de données appelé `".metadata"` a une signification particulière : Il n'est pas accessible depuis le code et il est toujours ajouté à la toute fin du bytecode, quelle que soit sa position dans l'objet.

D'autres objets de données avec une signification particulière pourraient être ajoutés dans le futur, mais leurs noms commenceront toujours par un `.`.

Un exemple d'objet Yul est présenté ci-dessous :

```
// Un contrat consiste en un objet unique avec des sous-objets représentant
// le code à déployer ou d'autres contrats qu'il peut créer.
// Le noeud unique "code" est le code exécutable de l'objet.
// Chaque (autre) objet nommé ou section de données est sérialisé et // rendu
// accessible aux fonctions spéciales intégrées datacopy / dataoffset / datasize.
// L'objet actuel, les sous-objets et les éléments de données à l'intérieur de l
↪'objet actuel
// sont dans le champ d'application.
object "Contract1" {
    // C'est le code du constructeur du contrat.
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // créer d'abord "Contract2"
        let size := datasize("Contract2")
        let offset := allocate(size)
        // Ceci se transformera en codecopie pour EVM
        datacopy(offset, dataoffset("Contract2"), size)
        // le paramètre du constructeur est un seul nombre 0x1234
        mstore(add(offset, size), 0x1234)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        pop(create(offset, add(size, 32), 0))

        // retourne maintenant l'objet d'exécution (le code
        // actuellement exécuté est le code du constructeur)
        size := datasize("runtime")
        offset := allocate(size)
        // Cela se transformera en une copie mémoire->mémoire pour Ewasm et
        // une codecopie pour EVM
        datacopy(offset, dataoffset("runtime"), size)
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "runtime" {
        code {
            function allocate(size) -> ptr {
                ptr := mload(0x40)
                if iszero(ptr) { ptr := 0x60 }
                mstore(0x40, add(ptr, size))
            }

            // code d'exécution

            mstore(0, "Hello, World!")
            return(0, 0x20)
        }
    }

    // Objet embarqué. Le cas d'utilisation est que l'extérieur est un contrat d
↪'usine,
    // et Contract2 est le code à créer par la fabrique
    object "Contract2" {
        code {
            // code ici ...
        }

        object "runtime" {
            code {
            // code ici ...
            }
        }

        data "Table1" hex"4123"
    }
}
```

3.32.7 Optimiseur de Yul

L'optimiseur Yul fonctionne sur du code Yul et utilise le même langage pour l'entrée, la sortie et les états intermédiaires. Cela permet de faciliter le débogage et la vérification de l'optimiseur.

Veuillez vous référer à la documentation générale *optimizer* pour plus de détails sur les différentes étapes d'optimisation et l'utilisation de l'optimiseur.

Si vous voulez utiliser Solidity en mode autonome Yul, vous activez l'optimiseur en utilisant `--optimize` et spécifiez éventuellement le *nombre attendu d'exécutions de contrats* avec `--optimize-runs` :

```
solc --strict-assembly --optimize --optimize-runs 200
```

En mode Solidity, l'optimiseur Yul est activé en même temps que l'optimiseur normal.

Séquence des étapes d'optimisation

Par défaut, l'optimiseur Yul applique sa séquence prédéfinie d'étapes d'optimisation à l'assemblage généré. Vous pouvez remplacer cette séquence et fournir la vôtre en utilisant l'option `--yul-optimizations` :

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul'
```

L'ordre des étapes est significatif et affecte la qualité du résultat. De plus, l'application d'une étape peut révéler de nouvelles possibilités d'optimisation pour d'autres qui ont déjà été appliquées. La répétition des étapes est donc souvent bénéfique. En plaçant une partie de la séquence entre crochets (`[]`), vous indiquez à l'optimiseur d'appliquer cette partie jusqu'à ce qu'elle n'améliore plus la taille de l'assemblage résultant. Vous pouvez utiliser les crochets plusieurs fois dans une même séquence mais ils ne peuvent pas être imbriqués.

Les étapes d'optimisation suivantes sont disponibles :

Abréviation	Nom complet
f	BlockFlattener
l	CircularReferencesPruner
c	CommonSubexpressionEliminator
C	ConditionalSimplifier
U	ConditionalUnsimplifier
n	ControlFlowSimplifier
D	DeadCodeEliminator
v	EquivalentFunctionCombiner
e	ExpressionInliner
j	ExpressionJoiner
s	ExpressionSimplifier
x	ExpressionSplitter
I	ForLoopConditionIntoBody
O	ForLoopConditionOutOfBody
o	ForLoopInitRewriter
i	FullInliner
g	FunctionGrouper
h	FunctionHoister
F	FunctionSpecializer
T	LiteralRematerialiser
L	LoadResolver
M	LoopInvariantCodeMotion

suite sur la page suivante

Tableau 2 – suite de la page précédente

Abréviation	Nom complet
r	RedundantAssignEliminator
R	ReasoningBasedSimplifier - highly experimental
m	Rematerialiser
V	SSAReverser
a	SSATransform
t	StructuralSimplifier
u	UnusedPruner
p	UnusedFunctionParameterPruner
d	VarDeclInitializer

Certaines étapes dépendent de propriétés assurées par BlockFlattener, FunctionGrouper, ForLoopInitRewriter. Pour cette raison, l'optimiseur Yul les applique toujours avant d'appliquer les étapes fournies par l'utilisateur.

Le ReasoningBasedSimplifier est une étape de l'optimiseur qui n'est actuellement pas activée dans le jeu d'étapes par défaut. Elle utilise un solveur SMT pour simplifier les expressions arithmétiques et les conditions booléennes. Il n'a pas encore été testé ou validé de manière approfondie et peut produire des résultats non reproductibles, veuillez donc l'utiliser avec précaution !

3.32.8 Exemple complet d'ERC20

```
object "Token" {
    code {
        // Enregistrez le créateur dans l'emplacement zéro.
        sstore(0, caller())

        // Déployer le contrat
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            // Protection contre l'envoi d'Ether
            require(iszero(callvalue()))

            // Distributeur
            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()
```

(suite sur la page suivante)

(suite de la page précédente)

```
            }
            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)
            }


            /* ---------- fonctions de décodage des données d'appel ----------- */
            function selector() -> s {
                s := div(calldataload(0),␣
↪0x100000000000000000000000000000000000000000000000000000000)
            }

            function decodeAsAddress(offset) -> v {
                v := decodeAsUint(offset)
```

(suite sur la page suivante)

(suite de la page précédente)

```
                if iszero(iszero(and(v,␣
↪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)
            }
            /* ---------- fonctions d'encodage des données d'appel ---------- */
            function returnUint(v) {
                mstore(0, v)
                return(0, 0x20)
            }
            function returnTrue() {
                returnUint(1)
            }

            /* -------- événements ---------- */
            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)
            }

            /* -------- schéma de stockage ---------- */
            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)
            }

            /* -------- accès au stockage ---------- */
            function owner() -> o {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            o := sload(ownerPos())
        }
        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))
        }

        /* ---------- fonctions d'utilité ---------- */
        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) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                if iszero(condition) { revert(0, 0) }
            }
        }
    }
}
```

3.33 Guide de style

3.33.1 Introduction

Ce guide est destiné à fournir des conventions de codage pour l'écriture du code Solidity. Ce guide doit être considéré comme un document évolutif qui changera au fur et à mesure que des conventions utiles seront trouvées et que les anciennes conventions seront rendues obsolètes.

De nombreux projets mettront en place leurs propres guides de style. En cas de conflits, les guides de style spécifiques au projet sont prioritaires.

La structure et un grand nombre de recommandations de ce guide de style ont été tirées du guide de style de python pep8 style guide.

Le but de ce guide n'est *pas* d'être la bonne ou la meilleure façon d'écrire du code Solidity. Le but de ce guide est la *consistance*. Une citation de python pep8 résume bien ce concept.

Note : Un guide de style est une question de cohérence. La cohérence avec ce guide de style est importante. La cohérence au sein d'un module ou d'une fonction est la plus importante.

Mais le plus important : **savoir quand être incohérent** - parfois le guide de style ne s'applique tout simplement pas. En cas de doute, utilisez votre meilleur jugement. Regardez d'autres exemples et décidez de ce qui vous semble le mieux. Et n'hésitez pas à demander !

3.33.2 Présentation du code

Indentation

Utilisez 4 espaces par niveau d'indentation.

Tabs ou Espaces

Les espaces sont la méthode d'indentation préférée.

Il faut éviter de mélanger les tabulations et les espaces.

Lignes vierges

Entourer les déclarations de haut niveau dans le code source de solidity de deux lignes vides.

Oui :

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

contract A {
    // ...
}


contract B {
    // ...
}


contract C {
    // ...
}
```

Non :

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

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Dans un contrat, les déclarations de fonctions sont entourées d'une seule ligne vierge.

Les lignes vides peuvent être omises entre des groupes de déclarations d'une seule ligne (comme les fonctions de base d'un contrat abstrait).

Oui :

```
// 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;
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}
```

Non :

```
// 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 {
        // ...
    }
}
```

Longueur maximale de la ligne

Garder les lignes sous la recommandation PEP 8 à un maximum de 79 (ou 99) caractères aide les lecteurs à analyser facilement le code.

Les lignes enveloppées doivent se conformer aux directives suivantes.

1. Le premier argument ne doit pas être attaché à la parenthèse ouvrante.
2. Une, et une seule, indentation doit être utilisée.
3. Chaque argument doit être placé sur sa propre ligne.
4. L'élément de terminaison, `);`, doit être placé seul sur la dernière ligne.

Appels de fonction

Oui :

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

Non :

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Déclarations d'affectation

Oui :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

Non :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(argument1,
                                                                   argument2,
                                                                   argument3,
                                                                   argument4);
```

Définitions d'événements et émetteurs d'événements

Oui :

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
```

(suite sur la page suivante)

(suite de la page précédente)

```
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

Non «

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Codage du fichier source

L'encodage UTF-8 ou ASCII est préféré.

Importations

Les déclarations d'importation doivent toujours être placées en haut du fichier.

Oui :

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

import "./Owned.sol";

contract A {
    // ...
}

contract B is Owned {
    // ...
}
```

Non :

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

(suite sur la page suivante)

(suite de la page précédente)

```
contract A {
    // ...
}


import "./Owned.sol";


contract B is Owned {
    // ...
}
```

Ordre des fonctions

L'ordre aide les lecteurs à identifier les fonctions qu'ils peuvent appeler et à trouver plus facilement les définitions des constructeurs et des fonctions de repli.

Les fonctions doivent être regroupées en fonction de leur visibilité et ordonnées :

- constructor
- receive function (si elle existe)
- fallback function (si elle existe)
- external
- public
- internal
- private

Dans un regroupement, placez les fonctions `view` et `pure` en dernier.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {
    constructor() {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
    }

    // Fonctions externes
    // ...

    // Fonctions externes qui sont view
    // ...

    // Fonctions externes qui sont pure
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // ...

    // Fonctions publiques
    // ...

    // Fonctions internes
    // ...

    // Fonctions privées
    // ...
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {

    // External functions
    // ...

    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // Fonctions privées
    // ...

    // Fonctions publiques
    // ...

    constructor() {
        // ...
    }

    // Fonctions internes
    // ...
}
```

Espaces blancs dans les expressions

Évitez les espaces blancs superflus dans les situations suivantes :

Immédiatement à l'intérieur des parenthèses, des crochets ou des accolades, à l'exception des déclarations de fonctions sur une seule ligne.

Oui :

```
spam(ham[1], Coin({name: "ham"}));
```

Non :

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception :

```
function singleLine() public { spam(); }
```

Immédiatement avant une virgule, un point-virgule :

Oui :

```
function spam(uint i, Coin coin) public;
```

Non ;

```
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;
long_variable = 3;
```

Non :

```
x             = 1;
y             = 2;
long_variable = 3;
```

Ne pas inclure d'espace dans les fonctions de réception et de repli :

Oui :

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

Non :

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

Structures de contrôle

Les accolades désignant le corps d'un contrat, d'une bibliothèque, de fonctions et de structs doivent :

- s'ouvrir sur la même ligne que la déclaration
- se fermer sur leur propre ligne au même niveau d'indentation que le début de la déclaration.
- L'accolade d'ouverture doit être précédée d'un espace.

Oui :

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

contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

Non :

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

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

Les mêmes recommandations s'appliquent aux structures de contrôle `if`, `else`, `while`, et `for`.

En outre, les structures de contrôle suivantes doivent être séparées par un espace unique `if`, `while` et `for` et le bloc entre parenthèses représentant le conditionnel, ainsi qu'un espace entre le bloc parenthétique conditionnel et l'accolade ouvrante.

Oui :

```
if (...) {
    ...
}

for (...) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ...
}
```

Non :

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

Pour les structures de contrôle dont le corps contient une seule déclaration, l'omission des accolades est acceptable *si* la déclaration est contenue sur une seule ligne.

Oui :

```
if (x < 10)
    x += 1;
```

Non :

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

Pour les blocs `if` qui ont une clause `else` ou `else if`, la clause `else` doit être placée sur la même ligne que l'accolade fermant le bloc `if`. Il s'agit d'une exception par rapport aux règles des autres structures de type bloc.

Oui :

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

Non :

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Déclaration de fonction

Pour les déclarations de fonction courtes, il est recommandé de garder l'accolade d'ouverture du corps de la fonction sur la même ligne que la déclaration de la fonction.

L'accolade fermante doit être au même niveau d'indentation que la déclaration de fonction. de la fonction.

L'accolade ouvrante doit être précédée d'un seul espace.

Oui :

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}
```

Non :

```
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;}
```

L'ordre des modificateurs pour une fonction doit être :

1. Visibilité
2. Mutabilité
3. Virtuel
4. Remplacer
5. Modificateurs personnalisés

Oui :

```
function balance(uint from) public view override returns (uint)  {
    return balanceOf[from];
}

function shutdown() public onlyOwner {
    selfdestruct(owner);
}
```

Non :

```
function balance(uint from) public override view returns (uint)  {
    return balanceOf[from];
}

function shutdown() onlyOwner public {
    selfdestruct(owner);
}
```

Pour les longues déclarations de fonctions, il est recommandé de déposer chaque argument sur sa propre ligne au même niveau d'indentation que le corps de la fonction. La parenthèse fermante et la parenthèse ouvrante doivent être placées sur leur propre ligne au même niveau d'indentation que la déclaration de fonction.

Oui :

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
{
    doSomething();
}
```

Non :

```
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(
```

(suite sur la page suivante)

(suite de la page précédente)

```
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

Si une longue déclaration de fonction comporte des modificateurs, chaque modificateur doit être déposé sur sa propre ligne.

Oui :

```
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();
}
```

Non :

```
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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public
    onlyOwner
    priced
    returns (address) {
    doSomething();
}
```

Les paramètres de sortie et les instructions de retour multilignes doivent suivre le même style que celui recommandé pour l'habillage des longues lignes dans la section Longueur de ligne maximale.

Oui :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

Non :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
             uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

Pour les fonctions constructrices sur les contrats hérités dont les bases nécessitent des arguments, il est recommandé de déposer les constructeurs de base sur de nouvelles lignes de la même manière que les modificateurs si la déclaration

de la fonction est longue ou difficile à lire.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// Contrats de base juste pour que cela 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)
    {
        // do something with param5
        x = param5;
    }
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Contrats de base juste pour que cela compile
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}


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;
    }
}


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;
        }
}
```

Lorsque vous déclarez des fonctions courtes avec une seule déclaration, il est permis de le faire sur une seule ligne.

C'est autorisé :

```
function shortFunction() public { doSomething(); }
```

Ces directives pour les déclarations de fonctions sont destinées à améliorer la lisibilité. Les auteurs doivent faire preuve de discernement car ce guide ne prétend pas couvrir toutes les permutations possibles pour les déclarations de fonctions.

Mappages

Dans les déclarations de variables, ne séparez pas le mot-clé `mapping` de son type par un espace. Ne séparez pas un mot-clé `mapping` imbriqué de son type par un espace.

Oui :

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

Non :

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

Déclarations de variables

Les déclarations de variables de tableau ne doivent pas comporter d'espace entre le type et les parenthèses.

Oui :

```
uint[] x;
```

Non :

```
uint [] x;
```

Autres recommandations

— Les chaînes de caractères devraient être citées avec des guillemets doubles au lieu de guillemets simples.

Oui :

```
str = "foo";
str = "Hamlet dit : 'Être ou ne pas être...'";
```

Non :

```
str = 'bar';
str = '"Soyez vous-même ; tous les autres sont déjà pris." -Oscar Wilde';
```

— Entourer les opérateurs d'un espace unique de chaque côté.

Oui :

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

Non :

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

— Les opérateurs ayant une priorité plus élevée que les autres peuvent exclure les espaces afin d'indiquer la préséance. Ceci a pour but de permettre d'améliorer la lisibilité d'une déclaration complexe. Vous devez toujours utiliser la même quantité d'espaces blancs de part et d'autre d'un opérateur :

Oui :

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

Non :

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

3.33.3 Ordre de mise en page

Disposez les éléments du contrat dans l'ordre suivant :

1. Déclarations de pragmatisme
2. Instructions d'importation
3. Interfaces
4. Bibliothèques
5. Contrats

À l'intérieur de chaque contrat, bibliothèque ou interface, utilisez l'ordre suivant :

1. Les déclarations de type
2. Variables d'état
3. Événements
4. Fonctions

Note : Il peut être plus clair de déclarer les types à proximité de leur utilisation dans les événements ou les variables d'état.

3.33.4 Conventions d'appellation

Les conventions de dénomination sont puissantes lorsqu'elles sont adoptées et utilisées à grande échelle. L'utilisation de différentes conventions peut véhiculer des informations *méta* significatives qui, autrement, ne seraient pas immédiatement disponibles.

Les recommandations de nommage données ici sont destinées à améliorer la lisibilité, et ne sont donc pas des règles, mais plutôt des lignes directrices pour essayer d'aider à transmettre le plus d'informations à travers les noms des choses.

Enfin, la cohérence au sein d'une base de code devrait toujours prévaloir sur les conventions décrites dans ce document.

Styles de dénomination

Pour éviter toute confusion, les noms suivants seront utilisés pour faire référence à différents styles d'appellation.

— `b` (lettre minuscule simple)
— `B` (lettre majuscule simple)
— `lettresminuscules`
— `minuscule_avec_underscores`
— `MAJUSCULE`
— `MAJUSCULE_AVEC_UNDERSCORES`
— `MotsEnMajuscule` (ou MotsEnMaj)
— `casMixe` (diffère des CapitalizedWords par le caractère minuscule initial !)
— `Mots_Capitalisés_Avec_Underscores`

Note : Lorsque vous utilisez des sigles dans CapWords, mettez toutes les lettres des sigles en majuscules. Ainsi, HTTPServerError est préférable à HttpServerError. Lors de l'utilisation d'initiales en mixedCase, mettez toutes les lettres des initiales en majuscules, mais gardez la première en minuscule si elle est le début du nom. Ainsi, xmlHTTPRequest est préférable à XMLHTTPRequest.

Noms à éviter

- `l` - Lettre minuscule el
- `O` - Lettre majuscule oh
- `I` - Lettre majuscule eye

N'utilisez jamais l'un de ces noms pour des noms de variables à une seule lettre. Elles sont souvent impossibles à distinguer des chiffres un et zéro.

Noms de contrats et de bibliothèques

- Les contrats et les bibliothèques doivent être nommés en utilisant le style CapWords. Exemples : `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
- Les noms des contrats et des bibliothèques doivent également correspondre à leurs noms de fichiers.
- Si un fichier de contrat comprend plusieurs contrats et/ou bibliothèques, alors le nom du fichier doit correspondre au *contrat principal*. Cela n'est cependant pas recommandé si cela peut être évité.

Comme le montre l'exemple ci-dessous, si le nom du contrat est `Congress` et celui de la bibliothèque `Owned`, les noms de fichiers associés doivent être `Congress.sol` et `Owned.sol`.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Owned.sol
contract Owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

et dans `Congress.sol` :

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

(suite sur la page suivante)

(suite de la page précédente)

```
import "./Owned.sol";


contract Congress is Owned, TokenRecipient {
    //...
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// owned.sol
contract owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

et dans `Congress.sol` :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;


import "./owned.sol";


contract Congress is owned, tokenRecipient {
    //...
}
```

Noms de structures

Les structures doivent être nommées en utilisant le style CapWords. Exemples :`MonCoin`, `Position`, `PositionXY`.

Noms d'événements

Les événements doivent être nommés en utilisant le style CapWords. Exemples : `Dépôt`, `Transfert`, `Approbation`, `AvantTransfert`, `AprèsTransfert`.

Noms des fonctions

Les fonctions doivent utiliser la casse mixte. Exemples : `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Noms des arguments de la fonction

Les arguments des fonctions doivent utiliser des majuscules et des minuscules. Exemples : `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

Lorsque vous écrivez des fonctions de bibliothèque qui opèrent sur un struct personnalisé, le struct doit être le premier argument et doit toujours être nommée `self`.

Noms des variables locales et des variables d'état

Utilisez la casse mixte. Exemples : `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constantes

Les constantes doivent être nommées avec des lettres majuscules et des caractères de soulignement pour séparer les mots. Exemples : `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Noms des modificateurs

Utilisez la casse mixte. Exemples : `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enums

Les Enums, dans le style des déclarations de type simples, doivent être nommés en utilisant le style CapWords. Exemples : `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Éviter les collisions de noms

— `single_trailing_underscore_`

Cette convention est suggérée lorsque le nom souhaité entre en collision avec celui d'un nom intégré ou autrement réservé.

3.33.5 NatSpec

Les contrats Solidity peuvent également contenir des commentaires NatSpec. Ils sont écrits avec une triple barre oblique (`///`) ou un double astérisque (`/** ... */`). Ils doivent être utilisés directement au-dessus des déclarations de fonctions ou des instructions.

Par exemple, le contrat de *un smart contract simple* avec les commentaires ajoutés, ressemble à celui ci-dessous :

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

/// @author L'équipe Solidity
/// @title Un exemple simple de stockage
contract SimpleStorage {
    uint storedData;

    /// Stocke `x`.
    /// @param x la nouvelle valeur à stocker
    /// @dev stocke le nombre dans la variable d'état `storedData`.
    function set(uint x) public {
        storedData = x;
    }

    /// Retourner la valeur stockée.
    /// @dev récupère la valeur de la variable d'état `storedData`.
    /// @retourne la valeur stockée
    function get() public view returns (uint) {
        return storedData;
    }
}
```

Il est recommandé que les contrats Solidity soient entièrement annotés en utilisant *NatSpec* pour toutes les interfaces publiques (tout ce qui se trouve dans l'ABI).

Veuillez consulter la section sur *NatSpec* pour une explication détaillée.

3.34 Modèles communs

3.34.1 Retrait des contrats

La méthode recommandée pour envoyer des fonds après un effet est d'utiliser le modèle de retrait. Bien que la méthode la plus intuitive la méthode la plus intuitive pour envoyer de l'Ether, suite à un effet, est un appel direct de « transfert », ce n'est pas recommandé car il introduit un car elle introduit un risque potentiel de sécurité. Vous pouvez lire plus d'informations à ce sujet sur la page *Considérations de sécurité*.

Voici un exemple du schéma de retrait en pratique dans un contrat où l'objectif est d'envoyer le plus d'argent vers le contrat afin de devenir le plus « riche », inspiré de King of the Ether.

Dans le contrat suivant, si vous n'êtes plus le plus riche, vous recevez les fonds de la personne qui est maintenant la plus riche.

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

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    /// La quantité d'Ether envoyé n'était pas supérieur au
    /// montant le plus élevé actuellement.
    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;
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // N'oubliez pas de mettre à zéro le remboursement en attente avant
        // l'envoi pour éviter les attaques de ré-entrance
        pendingWithdrawals[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

Cela s'oppose au modèle d'envoi plus intuitif :

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

contract SendContract {
    address payable public richest;
    uint public mostSent;

    /// La quantité d'Ether envoyée n'était pas plus élevée que
    /// le montant le plus élevé actuellement.
    error NotEnoughEther();

    constructor() payable {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        richest = payable(msg.sender);
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        // Cette ligne peut causer des problèmes (expliqués ci-dessous).
        richest.transfer(msg.value);
        richest = payable(msg.sender);
        mostSent = msg.value;
    }
}
```

Remarquez que, dans cet exemple, un attaquant pourrait piéger le contrat dans un état inutilisable en faisant en sorte que `richest` soit l'adresse d'un contrat qui possède une fonction de réception ou de repli qui échoue (par exemple en utilisant `revert()` ou simplement en consommant plus que l'allocation de 2300 gaz qui leur a été transférée). De cette façon, chaque fois que `transfer` est appelé pour livrer des fonds au contrat « empoisonné », il échouera et donc aussi `becomeRichest` échouera aussi, et le contrat sera bloqué pour toujours.

En revanche, si vous utilisez le motif « withdraw » du premier exemple, l'attaquant ne peut faire échouer que son propre retrait, et pas le reste le reste du fonctionnement du contrat.

3.34.2 Restriction de l'accès

La restriction de l'accès est un modèle courant pour les contrats. Notez que vous ne pouvez jamais empêcher un humain ou un ordinateur de lire le contenu de vos transactions ou l'état de votre contrat. Vous pouvez rendre les choses un peu plus difficiles en utilisant le cryptage, mais si votre contrat est supposé lire les données, tout le monde le fera aussi.

Vous pouvez restreindre l'accès en lecture à l'état de votre contrat par **d'autres contrats**. C'est en fait le cas par défaut sauf si vous déclarez vos variables d'état `public`.

De plus, vous pouvez restreindre les personnes qui peuvent apporter des modifications l'état de votre contrat ou appeler les fonctions de votre contrat. fonctions de votre contrat et c'est ce dont il est question dans cette section.

L'utilisation de **modificateurs de fonction** permet de rendre ces restrictions très lisibles.

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

contract AccessRestriction {
    // Ils seront attribués lors de la construction
    // phase de construction, où `msg.sender` est le compte
    // qui crée ce contrat.
    address public owner = msg.sender;
    uint public creationTime = block.timestamp;

    // Suit maintenant une liste d'erreurs que
    // ce contrat peut générer ainsi que
    // avec une explication textuelle dans des
    // commentaires spéciaux.

    /// L'expéditeur n'est pas autorisé pour cette
    /// opération.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    error Unauthorized();

    /// La fonction est appelée trop tôt.
    error TooEarly();

    /// Pas assez d'Ether envoyé avec l'appel de fonction.
    error NotEnoughEther();

    // Les modificateurs peuvent être utilisés pour changer
    // le corps d'une fonction.
    // Si ce modificateur est utilisé, il
    // ajoutera une vérification qui ne se passe
    // que si la fonction est appelée depuis
    // une certaine adresse.
    modifier onlyBy(address _account)
    {
        if (msg.sender != _account)
            revert Unauthorized();
        // N'oubliez pas le "_;"! Il sera
        // remplacé par le corps de la fonction
        // réelle lorsque le modificateur est utilisé.
        _;
    }

    /// Faire de `_newOwner` le nouveau propriétaire de ce
    /// contrat.
    function changeOwner(address _newOwner)
        public
        onlyBy(owner)
    {
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        if (block.timestamp < _time)
            revert TooEarly();
        _;
    }

    /// Effacer les informations sur la propriété.
    /// Ne peut être appelé que 6 semaines après
    /// que le contrat ait été créé.
    function disown()
        public
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // Ce modificateur exige qu'un certain
    // frais étant associé à un appel de fonction.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Si l'appelant a envoyé trop de frais, il ou elle est
    // remboursé, mais seulement après le corps de la fonction.
    // Ceci était dangereux avant la version 0.4.0 de Solidity,
    // où il était possible de sauter la partie après `_;`.
    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;
        // juste quelques exemples de conditions
        if (uint160(owner) & 0 == 1)
            // Cela n'a pas remboursé pour Solidity
            // avant la version 0.4.0.
            return;
        // rembourser les frais payés en trop
    }
}
```

Une manière plus spécialisée de restreindre l'accès aux appels peut être restreint, sera abordée dans l'exemple suivant.

3.34.3 Machine à états

Les contrats se comportent souvent comme une machine à états, ce qui signifie qu'ils ont certaines **étapes** dans lesquelles ils se comportent différemment ou dans lesquelles différentes fonctions peuvent être appelées. Un appel de fonction termine souvent une étape et fait passer le contrat à l'étape suivante (surtout si le contrat modélise une **interaction**). Il est également courant que certaines étapes soient automatiquement à un certain moment dans le **temps**.

Par exemple, un contrat d'enchères à l'aveugle qui commence à l'étape « accepter des offres à l'aveugle », puis qui passe ensuite à l'étape « révéler les offres » et qui se termine par « déterminer le résultat de l'enchère ».

Les modificateurs de fonction peuvent être utilisés dans cette situation pour modéliser les états et se prémunir contre l'utilisation incorrecte du contrat.

Exemple

Dans l'exemple suivant, le modificateur `atStage` assure que la fonction ne peut être appelée qu'à un certain stade.

Les transitions automatiques temporisées sont gérées par le modificateur `timedTransitions`, devrait être utilisé pour toutes les fonctions.

Note : L'ordre des modificateurs est important. Si atStage est combiné avec timedTransitions, assurez-vous que vous le mentionnez après cette dernière, afin que la nouvelle étape soit prise en compte.

Enfin, le modificateur `transitionNext` peut être utilisé pour passer automatiquement à l'étape suivante lorsque la fonction se termine.

Note : Le Modificateur Peut Être Ignoré. Ceci s'applique uniquement à Solidity avant la version 0.4.0 : Puisque les modificateurs sont appliqués en remplaçant simplement code et non en utilisant un appel de fonction, le code dans le modificateur transitionNext peut être ignoré si la fonction elle-même utilise return. Si vous voulez faire cela, assurez-vous d'appeler nextStage manuellement à partir de ces fonctions. À partir de la version 0.4.0, le code du modificateur sera exécuté même si la fonction retourne explicitement.

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

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }
    /// La fonction ne peut pas être appelée pour le moment.
    error FunctionInvalidAtThisStage();

    // Il s'agit de l'étape actuelle.
    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);
    }

    // Effectuez des transitions chronométrées. Veillez à mentionner
    // ce modificateur en premier, sinon les gardes
    // ne tiendront pas compte de la nouvelle étape.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    block.timestamp >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                block.timestamp >= creationTime + 12 days)
            nextStage();
        // Les autres étapes se déroulent par transaction
        _;
    }

    // L'ordre des modificateurs est important ici !
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // Nous n'implémenterons pas cela ici
    }

    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // Ce modificateur passe à l'étape suivante
    // après que la fonction soit terminée.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

3.35 Liste des bogues connus

Ci-dessous, vous trouverez une liste, formatée en JSON, de certains des bogues connus relatifs à la sécurité dans le compilateur Solidity. Le fichier lui-même est hébergé dans le dépositaire Github. La liste remonte jusqu'à la version 0.3.0, les bogues connus pour être présents uniquement dans les versions précédentes ne sont pas listés.

Il existe un autre fichier appelé bugs_by_version.json, qui peut être utilisé pour vérifier quels bugs affectent une version spécifique du compilateur.

Les outils de vérification des sources des contrats et aussi les autres outils interagissant avec les contrats doivent consulter cette liste selon les critères suivants :

- Il est légèrement suspect qu'un contrat ait été compilé avec une version nocturne du compilateur au lieu d'une version publiée. Cette liste ne garde pas des versions non publiées ou des versions nocturnes.
- Il est également légèrement suspect qu'un contrat ait été compilé avec une version qui n'était pas la plus récente au moment où le contrat a été établi. Pour les contrats contrats créés à partir d'autres contrats, vous devez suivre la chaîne de création jusqu'à une transaction et utiliser la date de cette transaction comme date de création.
- Il est très suspect qu'un contrat ait été compilé à l'aide d'un compilateur qui contient un bogue connu et que le contrat a été créé à un moment où une version plus récente du compilateur contenant un correctif était déjà disponible.

Le fichier JSON des bogues connus ci-dessous est un tableau d'objets, un pour chaque bogue, avec les clés suivantes :

uid Identifiant unique donné au bogue sous la forme `SOL-<year>-<number>`. Il est possible que plusieurs entrées existent avec le même uid. Cela signifie que que plusieurs gammes de versions sont affectées par le même bogue.

name Nom unique donné au bogue

summary Brève description du bogue

description Description détaillée du bogue

link URL d'un site web contenant des informations plus détaillées, facultatif

introduced La première version du compilateur publiée qui contenait le bogue, facultatif

fixed La première version du compilateur publiée qui ne contenait plus le bogue

publish La date à laquelle le bogue a été connu publiquement, facultative.

severity Gravité du bug : très faible, faible, moyenne, élevée. Prend en compte la possibilité de découverte dans les tests contractuels, la probabilité d'occurrence et les dommages potentiels par des exploits.

conditions Les conditions qui doivent être remplies pour déclencher le bug. Les touches suivantes suivantes peuvent être utilisées : `optimizer`, valeur booléenne qui signifie que l'optimiseur booléen qui signifie que l'optimiseur doit être activé pour activer le bogue. `evmVersion`, une chaîne qui indique quelle version de EVM les paramètres de compilation déclenche le bogue. La chaîne peut contenir des opérateurs opérateurs de comparaison. Par exemple, `">=constantinople"` signifie que le bug bogue est présent lorsque la version de l'EVM est définie sur `constantinople` ou ou plus. Si aucune condition n'est donnée, on suppose que le bogue est présent.

check Ce champ contient différentes vérifications qui indiquent si le contrat intelligent contient ou non le bogue. Le premier type de vérification est constitué d'expressions régulières Javascript qui doivent être comparées au code source (« source-regex ») si le bogue est présent. S'il n'y a pas de correspondance, alors le bogue est

très probablement pas présent. S'il y a une correspondance, le bogue pourrait être présent. Pour une meilleure précision, les vérifications doivent être appliquées au code source après avoir enlevé les commentaires. commentaires. Le deuxième type de vérification concerne les motifs à vérifier sur l'AST compact du programme le programme Solidity (« ast-compact-json-path »). La requête de recherche spécifiée est une expression JsonPath. Si au moins un chemin de l'AST Solidity correspond à la requête, le bogue est probablement présent.

```
[
    {
        "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.
↪",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "link": "https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/",
        "introduced": "0.4.16",
        "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"
    },
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "uid": "SOL-2020-10",
        "name": "DynamicArrayCleanup",
        "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␣
↪and will instead jump out to whatever was on the stack before the call.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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␣
↪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.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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",
        "severity": "medium",
        "conditions": {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "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␣
↪to the regular optimizer in the compiler settings.",
        "introduced": "0.5.14",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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*,␣
↪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.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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␣
↪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/",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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␣
↪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.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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␣
↪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"
```

(suite sur la page suivante)

(suite de la page précédente)

```
    },
    {
        "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␣
↪data is corrupted.",
        "fixed": "0.4.12",
        "severity": "low"
    },
    {
        "uid": "SOL-2017-1",
        "name": "ConstantOptimizerSubtraction",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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␣
↪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/",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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"
    },
    {
        "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.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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",
        "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"
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
]
```

3.36 Contribution

L'aide est toujours la bienvenue et il existe de nombreuses possibilités de contribuer à Solidity.

En particulier, nous apprécions le soutien dans les domaines suivants :

- Signaler les problèmes.
- Corriger et répondre aux problèmes de Solidity's GitHub issues., en particulier ceux marqués comme « good first issue » qui sont destinés à servir de problèmes d'introduction pour les contributeurs externes.
- Améliorer la documentation.
- Traduire la documentation dans plus de langues.
- Répondre aux questions des autres utilisateurs sur StackExchange et le Solidity Gitter Chat.
- S'impliquer dans le processus de conception du langage en proposant des changements de langage ou de nouvelles fonctionnalités sur le forum Solidity et en fournissant des commentaires.

Pour commencer, vous pouvez essayer *Construire à partir de la source* afin de vous familiariser avec les composants de Solidity et le processus de construction. En outre, il peut être utile de vous familiariser avec l'écriture de contrats intelligents dans Solidity.

Veuillez noter que ce projet est publié avec un Code de conduite du contributeur. En participant à ce projet - dans les problèmes, les demandes de pull, ou les canaux Gitter - vous acceptez de respecter ses termes.

3.36.1 Appels de l'équipe

Si vous avez des problèmes ou des demandes de pull à discuter, ou si vous êtes intéressé à entendre ce sur quoi l'équipe et les contributeurs travaillent, vous pouvez rejoindre nos appels d'équipe publics :

- Les lundis à 15h CET/CEST.
- Les mercredis à 14h CET/CEST.

Les deux appels ont lieu sur Jitsi.

3.36.2 Comment signaler des problèmes

Pour signaler un problème, veuillez utiliser le GitHub issues tracker. Lorsque rapportant des problèmes, veuillez mentionner les détails suivants :

- Version de Solidity.
- Code source (le cas échéant).
- Système d'exploitation.
- Étapes pour reproduire le problème.
- Le comportement réel par rapport au comportement attendu.

Il est toujours très utile de réduire au strict minimum le code source à l'origine du problème. Très utile et permet même parfois de clarifier un malentendu.

3.36.3 Flux de travail pour les demandes de Pull

Pour contribuer, merci de vous détacher de la branche `develop` et d'y faire vos modifications ici. Vos messages de commit doivent détailler *pourquoi* vous avez fait votre changement en plus de *ce que vous avez fait (sauf si c'est un changement minuscule)*.

Si vous avez besoin de retirer des changements de la branche `develop` après avoir fait votre fork (par (par exemple, pour résoudre des conflits de fusion potentiels), évitez d'utiliser `git merge` et à la place, `git rebase` votre branche. Cela nous aidera à revoir votre changement plus facilement.

De plus, si vous écrivez une nouvelle fonctionnalité, veuillez vous assurer que vous ajoutez des tests appropriés sous `test/` (voir ci-dessous).

Cependant, si vous effectuez un changement plus important, veuillez consulter le canal Gitter du développement de Solidity (différent de celui mentionné ci-dessus, celui-ci est axé sur le développement du compilateur et du langage plutôt que sur l'utilisation du langage) en premier lieu.

Les nouvelles fonctionnalités et les corrections de bogues doivent être ajoutées au fichier `Changelog.md` : veuillez suivre le style des entrées précédentes, le cas échéant.

Enfin, veillez à respecter le ``style de codage <https://github.com/ethereum/solidity/blob/develop/CODING_STYLE.md>`_ pour ce projet. De plus, même si nous effectuons des tests CI, veuillez tester votre code et assurez-vous qu'il se construit localement avant de soumettre une demande de pull.

Merci pour votre aide !

3.36.4 Exécution des tests du compilateur

Conditions préalables

Pour exécuter tous les tests du compilateur, vous pouvez vouloir installer facultativement quelques dépendances (evmone, libz3, et libhera).

Sur macOS, certains des scripts de test attendent que GNU coreutils soit installé. Ceci peut être accompli plus facilement en utilisant Homebrew : `brew install coreutils`.

Exécution des tests

Solidity inclut différents types de tests, la plupart d'entre eux étant regroupés dans l'application `Boost C++ Test Framework`. Boost C++ Test Framework application `soltest`. Exécuter `build/test/soltest` ou son wrapper `scripts/soltest.sh` est suffisant pour la plupart des modifications.

Le script *./scripts/tests.sh* ` exécute automatiquement la plupart des tests Solidity, y compris ceux inclus dans le Boost C++ Test Framework l'application `soltest` (ou son enveloppe `scripts/soltest.sh`), ainsi que les tests en ligne de commande et les tests de compilation.

Le système de test essaie automatiquement de découvrir l'emplacement du evmone pour exécuter les tests sémantiques.

La bibliothèque `evmone` doit être située dans le répertoire `deps` ou `deps/lib` relativement au répertoire de travail actuel, à son parent ou au parent de son parent. Alternativement, un emplacement explicite pour l'objet partagé `evmone` peut être spécifié via la variable d'environnement `ETH_EVMONE`.

`evmone` est principalement nécessaire pour l'exécution de tests sémantiques et de gaz. Si vous ne l'avez pas installé, vous pouvez ignorer ces tests en passant l'option `--no-semantic-tests` à `scripts/soltest.sh`.

L'exécution des tests Ewasm est désactivée par défaut et peut être explicitement activée via `./scripts/soltest.sh --ewasm` et nécessite que hera soit trouvé par `soltest.sh`. Pour être trouvé par `soltest`. Le mécanisme de

localisation de la bibliothèque `hera` est le même que pour `evmone`, sauf que la variable permettant de spécifier un emplacement explicite est appelée `ETH_HERA`.

Les bibliothèques `evmone` et `hera`` doivent toutes deux se terminer par l'extension de fichier avec l'extension ``.so` sur Linux, `.dll` sur les systèmes Windows et `.dylib` sur macOS.

Pour exécuter les tests SMT, la bibliothèque `libz3` doit être installée et localisable par `cmake` pendant l'étape de configuration du compilateur.

Si la bibliothèque `libz3` n'est pas installée sur votre système, vous devriez désactiver les tests SMT en exportant `SMT_FLAGS=--no-smt` avant de lancer `./scripts/tests.sh` ou de en exécutant *./scripts/soltest.sh –no-smt*`. Ces tests sont `libsolidity/smtCheckerTests` et `libsolidity/smtCheckerTestsJSON`.

Note : Pour obtenir une liste de tous les tests unitaires exécutés par Soltest, exécutez `./build/test/soltest --list_content=HRF`.

Pour obtenir des résultats plus rapides, vous pouvez exécuter un sous-ensemble de tests ou des tests spécifiques.

Pour exécuter un sous-ensemble de tests, vous pouvez utiliser des filtres : `./scripts/soltest.sh -t TestSuite/TestName`, où `TestName` peut être un joker `*`.

Ou, par exemple, pour exécuter tous les tests pour le désambiguïsateur yul : `./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`.

`./build/test/soltest --help` a une aide étendue sur toutes les options disponibles.

Voir en particulier :
- show_progress (-p) pour montrer l'achèvement du test,
- run_test (-t) pour exécuter des cas de tests spécifiques, et
- report-level (-r) donner un rapport plus détaillé.

Note : Ceux qui travaillent dans un environnement Windows et qui veulent exécuter les jeux de base ci-dessus sans libz3. En utilisant Git Bash, vous utilisez : `./build/test/Release/soltest.exe -- --no-smt`. Si vous exécutez ceci dans une Invite de Commande simple, utilisez : `./build/test/Release/soltest.exe -- --no-smt`.

Si vous voulez déboguer à l'aide de GDB, assurez-vous que vous construisez différemment de ce qui est « habituel ». Par exemple, vous pouvez exécuter la commande suivante dans votre dossier `build` : .. code-block : : bash

    cmake -DCMAKE_BUILD_TYPE=Debug .. make

Cela crée des symboles de sorte que lorsque vous déboguez un test en utilisant le drapeau `--debug`, vous avez accès aux fonctions et aux variables avec lesquelles vous pouvez casser ou imprimer.

Le CI exécute des tests supplémentaires (y compris `solc-js` et le test de frameworks Solidity tiers) qui nécessitent la compilation de la cible Emscripten.

Écrire et exécuter des tests de syntaxe

Les tests de syntaxe vérifient que le compilateur génère les messages d'erreur corrects pour le code invalide et accepte correctement le code valide. Ils sont stockés dans des fichiers individuels à l'intérieur du dossier `tests/libsolidity/syntaxTests`. Ces fichiers doivent contenir des annotations, indiquant le(s) résultat(s) attendu(s) du test respectif. La suite de tests les compile et les vérifie par rapport aux attentes données.

Par exemple : `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifiant déjà déclaré.
```

Un test de syntaxe doit contenir au moins le contrat testé lui-même, suivi du séparateur `// ----`. Les commentaires qui suivent le séparateur sont utilisés pour décrire les erreurs ou les avertissements attendus du compilateur. La fourchette de numéros indique l'emplacement dans le code source où l'erreur s'est produite. Si vous voulez que le contrat compile sans aucune erreur ou avertissement, vous pouvez omettre le séparateur et les commentaires qui le suivent.

Dans l'exemple ci-dessus, la variable d'état `variable` a été déclarée deux fois, ce qui n'est pas autorisé. Il en résulte un `DeclarationError` indiquant que l'identifiant a déjà été déclaré.

L'outil `isoltest` est utilisé pour ces tests et vous pouvez le trouver sous `./build/test/tools/`. C'est un outil interactif qui permet d'éditer les contrats défaillants en utilisant votre éditeur de texte préféré. Essayons de casser ce test en supprimant la deuxième déclaration de `variable` :

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifiant déjà déclaré.
```

Lancer `./build/test/tools/isoltest` à nouveau entraîne un échec du test :

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
        contract test {
            uint256 variable;
        }

    Expected result:
        DeclarationError: (36-52): Identifiant déjà déclaré.
    Obtained result:
        Success
```

`isoltest` imprime le résultat attendu à côté du résultat obtenu, et fournit aussi un moyen de modifier, de mettre à jour ou d'ignorer le fichier de contrat actuel, ou de quitter l'application.

Il offre plusieurs options pour les tests qui échouent :

- `edit` : `isoltest` essaie d'ouvrir le contrat dans un éditeur pour que vous puissiez l'ajuster. Il utilise soit l'éditeur donné sur la ligne de commande (comme `isoltest --editor /path/to/editor`), dans la variable d'environnement `EDITOR` ou juste `/usr/bin/editor` (dans cet ordre).
- `update` : Met à jour les attentes pour le contrat en cours de test. Cela met à jour les annotations en supprimant les attentes non satisfaites et en ajoutant les attentes manquantes. Le test est ensuite exécuté à nouveau.
- `skip` : Ignore l'exécution de ce test particulier.
- `quit'' : Quitte ``isoltest`.

Toutes ces options s'appliquent au contrat en cours, à l'exception de `quit` qui arrête l'ensemble du processus de test.

La mise à jour automatique du test ci-dessus le change en

```
contract test {
    uint256 variable;
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
// ----
```

et relancez le test. Il passe à nouveau :

```
Ré-exécution du cas de test...
syntaxTests/double_stateVariable_declaration.sol: OK
```

Note : Choisissez un nom pour le fichier du contrat qui explique ce qu'il teste, par exemple « double_variable_declaration.sol ». Ne mettez pas plus d'un contrat dans un seul fichier, sauf si vous testez l'héritage ou les appels croisés de contrats. Chaque fichier doit tester un aspect de votre nouvelle fonctionnalité.

3.36.5 Exécution du Fuzzer via AFL

Le fuzzing est une technique qui consiste à exécuter des programmes sur des entrées plus ou moins aléatoires afin de trouver des états d'exécution exceptionnels (défauts de segmentation, exceptions, etc.). Les fuzzers modernes sont intelligents et effectuent une recherche dirigée à l'intérieur de l'entrée. Nous avons un binaire spécialisé appelé `solfuzzer` qui prend le code source comme entrée et échoue chaque fois qu'il rencontre une erreur interne du compilateur, un défaut de segmentation ou similaire. mais n'échoue pas si, par exemple, le code contient une erreur. De cette façon, les outils de fuzzing peuvent trouver des problèmes internes dans le compilateur.

Nous utilisons principalement AFL pour le fuzzing. Vous devez télécharger et installer les paquets AFL depuis vos dépôts (afl, afl-clang) ou les construire manuellement. Ensuite, construisez Solidity (ou juste le binaire `solfuzzer`) avec AFL comme compilateur :

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

À ce stade, vous devriez pouvoir voir un message similaire à celui qui suit :

```
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>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

Si les messages d'instrumentation n'apparaissent pas, essayez de changer les drapeaux cmake pointant vers les binaires clang de l'AFL :

```
# si l'échec précédent
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Sinon, lors de l'exécution, le fuzzer s'arrête avec une erreur disant que le binaire n'est pas instrumenté :

```
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
```

Ensuite, vous avez besoin de quelques fichiers sources d'exemple. Cela permet au fuzzer de trouver des erreurs plus facilement. Vous pouvez soit copier certains fichiers des tests de syntaxe, soit extraire des fichiers de test de la documentation ou des autres 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
```

La documentation de l'AFL indique que le corpus (les fichiers d'entrée initiaux) ne doit pas être trop volumineux. Les fichiers eux-mêmes ne devraient pas être plus grands que 1 kB et il devrait y avoir au maximum un fichier d'entrée par fonctionnalité, donc mieux vaut commencer avec un petit nombre de fichiers. Il existe également un outil appelé `afl-cmin` qui peut couper les fichiers d'entrée qui ont pour résultat un comportement similaire du binaire.

Maintenant, lancez le fuzzer (le `-m` étend la taille de la mémoire à 60 Mo) :

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

Le fuzzer crée des fichiers sources qui conduisent à des échecs dans `/tmp/fuzzer_reports`. Il trouve souvent de nombreux fichiers sources similaires qui produisent la même erreur. Vous pouvez utiliser l'outil `scripts/uniqueErrors.sh` pour filtrer les erreurs uniques.

3.36.6 Moustaches

Whiskers est un système de modélisation de chaînes de caractères similaire à Mustache. Il est utilisé par le compilateur à divers endroits pour faciliter la lisibilité, et donc la maintenabilité et la vérifiabilité, du code.

La syntaxe présente une différence par rapport à Mustache. Les marqueurs de template *{{* ` et }} sont remplacés par < et > afin de faciliter l'analyse et d'éviter les conflits avec yul`. (Les symboles < ` et > ` sont invalides dans l'assemblage en ligne, tandis que { et } sont utilisés pour délimiter les blocs). Une autre limitation est que les listes ne sont résolues qu'à une seule profondeur et qu'elles ne sont pas récursives. Cela peut changer dans le futur.

Une spécification approximative est la suivante :

Toute occurrence de <name> est remplacée par la valeur de la variable fournie name sans aucun échappement et sans remplacement itératif. Une zone peut être délimitée par <#name>...</name>`. Elle est remplacée par autant de concaténations de son contenu qu'il y avait d'ensembles de variables fournis au système de modèles, en remplaçant chaque fois les éléments ``<inner> par leur valeur respective. Les variables de haut niveau peuvent également être utilisées à l'intérieur de ces zones.

Il existe également des conditionnels de la forme <?name>...<!name>...</name>, où les remplacements de modèles se poursuivent récursivement dans le premier ou le second segment, en fonction de la valeur du paramètre booléen name. Si <?+name>...<!+name>...</+name>` est utilisé, alors la vérification consiste à savoir si le paramètre chaîne de caractères ``name est non vide.

3.36.7 Guide de style de la documentation

Dans la section suivante, vous trouverez des recommandations de style spécifiquement axées sur la documentation des contributions à Solidity.

Langue anglaise

Utilisez l'anglais, avec une préférence pour l'orthographe anglaise britannique, sauf si vous utilisez des noms de projets ou de marques. Essayez de réduire l'utilisation de l'argot et les références locales, en rendant votre langage aussi clair que possible pour tous les lecteurs. Vous trouverez ci-dessous quelques références pour vous aider :

— L'anglais technique simplifié.
— Anglais international
— L'orthographe de l'anglais britannique

Note : Bien que la documentation officielle de Solidity soit écrite en anglais, il existe des traductions contribuées par la communauté dans d'autres langues. dans d'autres langues sont disponibles. Veuillez vous référer au guide de traduction pour savoir comment contribuer aux traductions de la communauté.

Cas de titre pour les en-têtes

Utilisez la casse des titres <https://titlecase.com>`_ pour les titres. Cela signifie qu'il faut mettre en majuscule tous les mots principaux dans titres, mais pas les articles, les conjonctions et les prépositions, sauf s'ils commencent le titre.

Par exemple, les exemples suivants sont tous corrects :

— Title Case for Headings.
— Pour les titres, utilisez la casse du titre.
— Noms de variables locales et d'État.
— Ordre de mise en page.

Développer les contractions

Utilisez des contractions développées pour les mots, par exemple :

— « Do not » au lieu de « Don't ».
— Can not » au lieu de « Can't ».

Voix active et passive

La voix active est généralement recommandée pour la documentation de type tutoriel car elle car elle aide le lecteur à comprendre qui ou quoi effectue une tâche. Cependant, comme la documentation de Solidity est un mélange de tutoriels et de contenu de référence, la voix passive est parfois plus appropriée.

En résumé :

— Utilisez la voix passive pour les références techniques, par exemple la définition du langage et les éléments internes de la VM Ethereum.
— Utilisez la voix active pour décrire des recommandations sur la façon d'appliquer un aspect de Solidity.

Par exemple, le texte ci-dessous est à la voix passive car il spécifie un aspect de Solidity :

> Les fonctions peuvent être déclarées « pures », auquel cas elles promettent de ne pas lire ou de modifier l'état.

Par exemple, le texte ci-dessous est à la voix active car il traite d'une application de Solidity :

> Lorsque vous invoquez le compilateur, vous pouvez spécifier comment découvrir le premier élément d'un chemin, ainsi que les remappages de préfixes de chemin.

Termes courants

— « Paramètres de fonction » et « variables de retour », et non pas paramètres d'entrée et de sortie.

Exemples de code

Un processus CI teste tous les exemples de code formatés en blocs de code qui commencent par » pragma solidity « , » contrat « , » bibliothèque » ou » interface « . ou » interface » en utilisant le script » ./test/cmdlineTests.sh » lorsque vous créez un PR. Si vous ajoutez de nouveaux exemples de code, assurez-vous qu'ils fonctionnent et passent les tests avant de créer le PR.

Assurez-vous que tous les exemples de code commencent par une version de `pragma` qui couvre la plus grande partie où le code du contrat est valide. Par exemple, `pragma solidity >=0.4.0 <0.9.0;`.

Exécution des Tests de Documentation

Assurez-vous que vos contributions passent nos tests de documentation en exécutant `./scripts/docs.sh` qui installe les dépendances nécessaires à la documentation et vérifie les problèmes éventuels. Nécessaires à la documentation et vérifie l'absence de problèmes tels que des liens brisés ou des problèmes de syntaxe.

3.36.8 Conception du langage Solidity

Pour vous impliquer activement dans le processus de conception du langage et partager vos idées concernant l'avenir de Solidity, veuillez rejoindre le forum Solidity.

Le forum Solidity sert de lieu pour proposer et discuter de nouvelles fonctionnalités du langage et de leur mise en œuvre dans les premiers stades de l'idéation ou des modifications de fonctionnalités existantes.

Dès que les propositions deviennent plus tangibles, leur implémentation sera également discutée dans le dépôt Solidity GitHub sous la forme de questions.

En plus du forum et des discussions sur les problèmes, nous organisons régulièrement des appels de discussion sur la conception du langage dans lesquels des sujets, questions ou implémentations de fonctionnalités sélectionnés sont débattus en détail. L'invitation à ces appels est partagée via le forum.

Nous partageons également des enquêtes de satisfaction et d'autres contenus pertinents pour la conception des langues sur le forum.

Si vous voulez savoir où en est l'équipe en termes d'implémentation de nouvelles fonctionnalités, vous pouvez suivre le statut de l'implémentation dans le projet Solidity Github. Les questions dans le backlog de conception nécessitent une spécification plus approfondie et seront soit discutées dans un appel de conception de langue ou dans un appel d'équipe régulier. Vous pouvez voir les changements à venir pour la prochaine version de rupture en passant de la branche par défaut (*develop*) à la breaking branch.

Pour les cas particuliers et les questions, vous pouvez nous contacter via le canal Solidity-dev Gitter, un chatroom dédié aux conversations autour du compilateur Solidity et du développement du langage.

Nous sommes heureux d'entendre vos réflexions sur la façon dont nous pouvons améliorer le processus de conception du langage pour qu'il soit encore plus collaboratif et transparent.

3.37 Guide de la marque Solidity

Ce guide de la marque contient des informations sur la politique de la marque Solidity et les directives d'utilisation du logo.

3.37.1 La marque Solidity

Le langage de programmation Solidity est un projet communautaire à code source ouvert dirigé par une équipe centrale. L'équipe centrale est parrainée par la Ethereum Foundation.

Ce document a pour objectif de fournir des informations sur la meilleure façon d'utiliser la marque et le logo Solidity.

Nous vous encourageons à lire attentivement ce document avant d'utiliser la marque ou le logo. Votre coopération est très appréciée !

3.37.2 Nom de marque Solidity

Le terme « Solidity » doit être utilisé pour faire référence au langage de programmation Solidity uniquement.

Veuillez ne pas utiliser « Solidity » :

- Pour faire référence à tout autre langage de programmation.
- D'une manière qui pourrait induire en erreur ou impliquer l'association de modules, d'outils, de documentation ou d'autres ressources sans rapport avec le langage Solidity.
- D'une manière qui sème la confusion dans la communauté quant à savoir si le langage de programmation Solidity est open-source et libre d'utilisation.

3.37.3 Licence du logo Solidity

Le logo Solidity est distribué et mis sous licence Creative Commons". Attribution 4.0 International License.

Il s'agit de la licence Creative Commons la plus permissive, qui autorise la réutilisation et les modifications à toutes fins. et les modifications dans n'importe quel but.

Vous êtes libre de :

- **Partager** - Copier et redistribuer le matériel sur tout support ou format.

— **Adapter** - Remixer, transformer et construire à partir de ce matériel dans n'importe quel but, même commercial. dans n'importe quel but, même commercial.

Aux conditions suivantes :

— **Attribution** - Vous devez donner le crédit approprié, fournir un lien à la la licence, et indiquer si des modifications ont été apportées. Vous pouvez le faire de toute manière raisonnable, mais pas d'une manière qui suggère que l'équipe centrale de Solidity vous approuve ou approuve votre utilisation.

Lorsque vous utilisez le logo Solidity, veuillez respecter les directives relatives au logo Solidity.

3.37.4 Directives relatives au logo Solidity

(Cliquez avec le bouton droit de la souris sur le logo pour le télécharger.)

Veuillez ne pas :

— Modifier le ratio du logo (ne pas l'étirer ou le couper).

— Modifier les couleurs du logo, sauf si cela est absolument nécessaire.

3.37.5 Crédits

Ce document a été, en partie, dérivé de la Python Software Foundation sur l'utilisation des marques déposées et du Guide des médias de Rust.

3.38 Influences de la langue

Solidity est un langage à virgule flottante qui a été influencé et inspiré par plusieurs langages de programmation bien connus.

Solidity est le plus profondément influencé par le C++, mais a également emprunté des concepts à des langages comme Python, JavaScript, et autres.

L'influence du C++ est visible dans la syntaxe des déclarations de variables, les boucles for, le concept de surcharge des fonctions, les conversions de type implicites et explicites et de nombreux autres détails.

Aux premiers jours du langage, Solidity était en partie influencé par JavaScript. Cela était dû à la détermination de la portée des variables au niveau des fonctions et à l'utilisation du mot-clé « var ». L'influence de JavaScript a été réduite à partir de la version 0.4.0. Maintenant, la principale similitude restante avec JavaScript est que les fonctions sont définies en utilisant le mot-clé `function`. Solidity prend également en charge la syntaxe et la sémantique de l'importation qui qui sont similaires à celles disponibles en JavaScript. En dehors de ces points, Solidity ressemble à la plupart des autres langages à accolades et n'a plus d'influence majeure de JavaScript.

Python a également influencé Solidity. Les modificateurs de Solidity ont été ajoutés en essayant de modéliser les décorateurs de Python avec plus d'efficacité. les décorateurs de Python avec une fonctionnalité beaucoup plus restreinte. De plus, l'héritage multiple, la linéarisation C3, et le mot-clé « super » sont tirés de Python, ainsi que la sémantique générale de l'assignation et des types de référence.

Index

D

E

F

G

H

I

T

U

V

W

Y