Solidity Documentation

Version 0.5.0

Ethereum

oct. 28, 2018

Table des matières

Solidity est un langage haut-niveau, orienté objet dédié à l'implémentation de smart contracts. Les smart contracts (littéralement contrats intelligents) sont des programes qui régissent le comportement de comptes dans l'état d'Ethereum.

Solidity a été influencé par C++, Python et JavaScript et est conçu pour cibler la machine virtuelle Ethereum (EVM).

Solidity est statiquement typé, supporte l'héritage, les librairies et les bibliothèques, ainsi que les types complexes définis par l'utilisateur parmi d'autres caractéristiques.

Avec Solidity, vous pouvez créer des contrats pour des usages tels que le vote, le crowdfunding, les enchères à l'aveugle, et portefeuilles multi-signature.

Note : La meilleure façon d'essayer Solidity à ce jour est d'utiliser Remix https://remix.ethereum.org/ (le chargement peut prendre un certain temps, merci d'être patient). Remix est un IDE basé sur un navigateur Web qui vous permet d'écrire des contrats intelligents Solidity, puis de déployer et exécuter les contrats intelligents.

Avertissement : Puisque le logiciel est écrit par des humains, il peut contenir des bugs. Ainsi, des contrats intelligents devraient également être créés selon les meilleures pratiques bien connues en matière de développement de logiciels. Cela comprend l'examen du code, les essais, les vérifications et les preuves d'exactitude. Notez également que les utilisateurs ont parfois plus confiance dans le code que ses auteurs. Enfin, les blockchains ont leurs propres choses à surveiller, alors jetez un coup d'oeil à la section *Security Considerations*.

CHAPITRE 1

Traductions

Cette documentation est traduite en plusieurs langues par des bénévoles de la communauté avec divers degrés d'exhaustivité et d'actualité. La version anglaise sert de référence.

- Anglais
- Chinois simplifié (en cours)
- Espagnol
- Russe (plutôt périmé)
- Coréen (en cours)

CHAPITRE 2

Liens utiles

2.1 Général

— Ethereum
— Changelog
— Code Source
— Ethereum Stackexchange
— Language Users Chat
— Compiler Developers Chat

2.2 Intégrations de Solidity disponibles

— Génériques :
  — **Remix** IDE basé sur navigateur avec compilateur intégré et environnement d'exécution Solidity sans composants côté serveur.
  — **Solium** Linter pour identifier et résoudre les problèmes de style et de sécurité dans Solidity.
  — **Solhint** Solidity linter qui fournit la sécurité, le guide de style et les règles de bonnes pratiques pour la validation intelligente des contrats.
— Atom :
  — **Etheratom** Plugin pour l'éditeur Atom qui comprend une coloration syntaxique, une compilation et un environnement d'exécution (Backend node & VM compatible).
  — **Atom Solidity Linter** Plugin pour l'éditeur Atom qui fournit un linter Solidity.
  — **Atom Solium Linter** Linter Solidty configurable pour Atom utilisant Solium comme base.
— Eclipse :
  — **YAKINDU Solidity Tools** IDE basé sur Eclipse. Caractéristiques : aide et complétion de code contextuelle, navigation dans le code, coloration syntaxique, compilateur intégré, corrections rapides et modèles.
— Emacs :
  — **Emacs Solidity** Plugin pour l'éditeur Emacs fournissant la coloration syntaxique et le reporting des erreurs de compilation.
— IntelliJ :

  - **IntelliJ IDEA plugin** Solidity plugin pour IntelliJ IDEA (et tous les autres IDE JetBrains)
- Sublime :
  - **Package for SublimeText — Solidity language syntax** Coloration syntaxique pour l'éditeur SublimeText.
- Vim :
  - **Vim Solidity** Plugin apportant la coloration syntaxique pour l'éditeur Vim.
  - **Vim Syntastic** Plugin pour l'éditeur Vim fournissant des checks de compilation.
- Visual Studio Code :
  - **Visual Studio Code extension** Solidity plugin pour Microsoft Visual Studio Code qui inclus la coloration syntaxique et un compilateur Solidity.

Discontinued :

- **Mix IDE** Qt IDE pour designer, debugger et tester les smarts contracts Solidity.
- **Ethereum Studio** Web IDE spécialisé qui apporte un environnement Ethereum complet.
- **Visual Studio Extension** Solidity plugin pour Microsoft Visual Studio qui inclut le compilateur Solidity.

2.3 Outils Solidity

- **Dapp** Outil de création, gestionnaire de paquets et assistant de déploiement pour Solidity.
- **Solidity REPL** Essayez Solidity instantanément avec une console Solidity en ligne de commande.
- **solgraph** Visualisez le flux de contrôle de Solidity et mettez en évidence les vulnérabilités potentielles en matière de sécurité.
- **Doxity** Générateur de documentation pour Solidity.
- **evmdis** Désassembleur EVM qui effectue une analyse statique sur le bytecode pour fournir un niveau d'abstraction plus élevé que les opérations EVM brutes.
- **ABI to solidity interface converter** Un script pour générer des interfaces de contrat à partir de l'ABI d'un smart contract.
- **Securify** Analyseur statique en ligne entièrement automatisé pour les smart contracts, fournissant un rapport de sécurité basé sur les modèles de vulnérabilité.
- **Sūrya** Outil utilitaire pour les systèmes de smart contracts, offrant un certain nombre de résultats visuels et d'informations sur la structure des contrats. Prend également en charge l'interrogation du graphe d'appel de fonction.
- **EVM Lab** Riche ensemble d'outils pour interagir avec l'EVM. Comprend une VM, une API Etherchain et un traceur avec affichage du coût du gaz.

Note : Des informations telles que les noms de variables, les commentaires et le formatage du code source sont perdus dans le processus de compilation et il n'est pas possible de récupérer complètement le code source original. Décompiler les contrats intelligents pour afficher le code source original pourrait ne pas être possible, ou le résultat final pourrait être utile.

2.4 Parsers et grammaires de Solidity de tierce parties

- **solidity-parser** Parser Solidity pour JavaScript
- **Solidity Grammar for ANTLR 4** Vérification de grammaire Solidity pour le générateur de parsers ANTLR 4

CHAPITRE 3

Documentation du langage

Dans les pages suivantes, nous verrons d'abord un *smart contract simple* écrit en Solidity suivi par les bases des *blockchains* et la *Machine virtuelle*.

La section suivante expliquera plusieurs *caractéristiques* de Solidity en donnant des exemples de contrats utiles *contrats d'exemple*. Rappelez-vous que vous pouvez toujours essayer les contrats dans votre navigateur !

La quatrième et plus vaste section couvrira en profondeur tous les aspects de Solidity.

Si vous avez encore des questions, vous pouvez essayer de chercher ou de poser des questions sur le site Web de Ethereum Stackexchange ou venez sur notre gitter channel. Les idées pour améliorer Solidity ou cette documentation sont toujours les bienvenues !

Traduit de l'anglais par Kevin Azoulay.

CHAPITRE 4

Sommaire

Keyword Index, Search Page

4.1 Introduction aux Smart Contracts

4.1.1 Un Smart Contract simple

Commençons par un exemple de base qui fixe la valeur d'une variable et l'expose pour l'accès par d'autres contrats. C'est très bien si vous ne comprenez pas tout maintenant, nous entrerons plus en détail plus tard.

Storage

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
    uint storedData;

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

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

La première ligne indique simplement que le code source est écrit pour Solidity version 0.4.0 ou tout ce qui est plus récent qui ne casse pas la fonctionnalité (jusqu'à la version 0.6.0, mais non comprise). Il s'agit de s'assurer que le contrat n'est pas compilable avec une nouvelle version du compilateur (de rupture), où il pourrait se comporter différemment. Les pré-cités pragmas sont des instructions courantes pour les compilateurs sur la façon de traiter le code source (par exemple pragma once).

Un contrat au sens de Solidity est un ensemble de code (ses *fonctions*) et les données (son *état*) qui résident à une adresse spécifique sur la blockchain Ethereum. La ligne `uint storedData;` déclare une variable d'état appelée `storedData`` de type `uint` (*u*nsigned *int*eger de *256* bits). Vous pouvez le considérer comme une case mémoire dans une base de données qui peut être interrogée et modifiée en appelant les fonctions du code qui gèrent la base de données. Dans le cas d'Ethereum, c'est toujours le contrat propriétaire. Et dans ce cas, les fonctions `set` et `get` peuvent être utilisées pour modifier ou récupérer la valeur de la variable.

Pour accéder à une variable d'état, vous n'avez pas besoin du préfixe `this.` d'autres langues.

Ce contrat ne fait pas encore grand-chose en dehors de (en raison de l'infrastructure construite par Ethereum) permettre à n'importe qui de stocker un numéro unique qui est accessible par n'importe qui dans le monde sans un moyen (faisable) pour vous empêcher de publier ce numéro. Bien sûr, n'importe qui peut simplement appeler `set` à nouveau avec une valeur différente. et écraser votre numéro, mais le numéro sera toujours stocké dans l'historique de la blockchain. Plus tard, nous verrons comment vous pouvez imposer des restrictions d'accès pour que vous seul puissiez modifier le numéro.

Note : Tous les identifiants (noms de contrat, 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 string.

Avertissement : Soyez prudent lorsque vous utilisez du texte Unicode, car des caractères d'apparence similaire (ou même identique) peuvent avoir des codages unicode différents et seront donc codés sous la forme d'un tableau d'octets différent.

Exemple de sous-monnaie

Le contrat suivant mettra en œuvre la forme la plus simple d'un contrat de cryptomonnaie. Il est possible de générer des pièces à partir de rien, mais seule la personne qui a créé le contrat sera en mesure de le faire (il est facile de mettre en œuvre un schéma d'émission différent). De plus, n'importe qui 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.

```
pragma solidity >0.4.99 <0.6.0;

contract Coin {
    // Le mot-clé "public" rend ces variables
    // facilement accessible de l'exterieur.
    address public minter;
    mapping (address => uint) public balances;

    // Les Events authowisent les clients légers à réagir
    // aux changements efficacement.
    event Sent(address from, address to, uint amount);

    // C'est le constructor, code qui n'est exécuté
    // qu'à la création du contrat.
    constructor() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Ce contrat introduit quelques nouveaux concepts, passons-les en revue un à un. La ligne `address public minter;` déclare une variable d'état de type adress qui est accessible au public. Le type `adress` est une valeur de 160 bits qui ne permet aucune opération arithmétique. Il convient pour le stockage des adresses de contrats ou de paires de clés appartenant à des tiers. Le mot-clé « public » génère automatiquement une fonction qui permet d'accéder à la valeur courante de la variable d'état de 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 à peu près équivalent à ce qui suit (ignorez `external'' et ``view` pour l'instant) :

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

Bien sûr, l'ajout d'une fonction exactement comme celle-là ne fonctionnera pas parce que nous aurions une fonction et une variable d'état avec le même nom, mais vous avez l'idée - le compilateur réalisera cela pour vous.

La ligne suivante, `mapping (" adress => uint ") public balances;` crée également une variable d'état publique, mais c'est un type de données plus complexe. Le type fait correspondre les adresses aux entiers non signés. Les mappings peuvent être vus comme des tables de hachage qui sont virtuellement initialisées de sorte que toutes les clés possibles existent dès le début et sont mappées à un fichier dont la représentation octale n'est que de zéros. Cette analogie ne va pas trop loin, car il n'est pas non plus possible d'obtenir une liste de toutes les clés d'un mapping, ni une liste de toutes les valeurs. Il faut donc garder à l'esprit (ou bien mieux, gardez une liste ou utilisez un type de données plus avancé) ce que vous avez ajouté à la cartographie ou l'utiliser dans un contexte où cela n'est pas nécessaire. La fonction getter créé par le mot-clé `public` est un peu plus complexe dans ce cas. Ça ressemble grossièrement à ça :

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

Comme vous pouvez le voir, vous pouvez utiliser cette fonction pour interroger facilement le solde d'un seul compte.

La ligne `event Sent(address from, address to, uint amount);` déclare un bien-nommé « event » qui est émis dans la dernière ligne de la fonction `send`. Les interfaces utilisateur (ainsi que les applications serveur bien sûr) peuvent écouter les événements qui sont émis sur la blockchain sans trop de frais. Dès qu'elle est émise, l'auditeur reçoit également le message des arguments « from », « to » et « amount », ce qui facilite le suivi des transactions. Pour écouter cet événement, vous devriez utiliser le code JavaScript suivant (qui suppose que ‘‘Coin‘ est un objet de contrat créé via web3.js ou un module similaire) :

```
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));
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
})
```

Notez comment la fonction `balances` générée automatiquement est appelée depuis l'interface utilisateur.

Le constructor est une fonction spéciale qui est exécutée pendant la création du contrat et ne peut pas être appelée ultérieurement. Il stocke de façon permanente l'adresse de la personne qui crée le contrat : `msg` (avec `tx` et `block`) est une variable globale spéciale qui contient certaines propriétés qui permettent d'accéder à la blockchain. `msg.sender` est toujours l'adresse d'où vient l'appel de la fonction courante (externe).

Enfin, les fonctions qui finiront avec le contrat et qui peuvent être appelées par les utilisateurs et les contrats sont « mint » et « send ». Si « mint » est appelé par quelqu'un d'autre que le compte qui a créé le contrat, rien ne se passera. Ceci est assuré par la fonction spéciale `require` qui fait que tous les changements sont annulés si son argument est évalué à faux. Le deuxième appel à `require` permet de s'assurer qu'il n'y aura pas trop de pièces, ce qui pourrait causer des erreurs de débordement de buffer plus tard.

D'un autre côté, `send` peut être utilisé par n'importe qui (qui a déjà certaines de ces pièces) pour envoyer des pièces à n'importe qui d'autre. Si vous n'avez pas assez de pièces à envoyer, l'appel `require` échouera et fournira également à l'utilisateur un message d'erreur approprié.

Note : Si vous utilisez ce contrat pour envoyer des pièces à une adresse, vous ne verrez rien lorsque vous regarderez cette adresse sur un explorateur de chaîne de blocs, parce que le fait que vous avez envoyé des pièces et les soldes modifiés sont seulement stockés dans le stockage de données de ce contrat de pièces particulier. Par l'utilisation d'événements, il est relativement facile de créer un « explorateur de chaîne » qui suit les transactions et les soldes de votre nouvelle pièce, mais vous devez inspecter l'adresse du contrat de pièces et non les adresses des propriétaires des pièces.

4.1.2 Blockchain Basics

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 (mining, hashing, elliptic-curve cryptography, réseaux pair-à-pair, etc.) sont juste là pour fournir un certain nombre de fonctionnalités et de promesses pour la plate-forme. Une fois que vous prenez ces fonctions pour aquises, vous n'avez pas à vous soucier de la technologie sous-jacente - ou devez-vous savoir comment fonctionne le cloud AWS d'Amazon en interne afin de 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 une transaction qui doit être acceptée par tous les autres. Le mot transaction implique que la modification que vous voulez effectuer (en supposant que vous voulez modifier deux valeurs en même temps) n'est pas effectuée du tout ou est complètement appliquée. De plus, pendant que votre transaction est appliquée à la base de données, aucune autre transaction ne peut la modifier.

Par exemple, imaginez un tableau qui énumère les soldes de tous les comptes dans une devise 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 quelque raison que ce soit, il n'est pas possible d'ajouter le montant au compte cible, le compte source n'est pas non plus modifié.

De plus, une transaction est toujours signée cryptographiquement par l'expéditeur (créateur). Il est donc facile de garder l'accès à des modifications spécifiques de la base de données. Dans l'exemple de la monnaie électronique, un

simple contrôle permet de s'assurer que seule la personne qui détient les clés du compte peut transférer de l'argent à partir de celui-ci.

Blocs

Un obstacle majeur à surmonter est ce que l'on appelle (en termes Bitcoin) une » attaque de double dépense » : Que se passe-t-il si deux transactions existent dans le réseau et que toutes deux veulent vider un compte ? Une seule 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 pair-à-pair.

La réponse abstraite à cette question est que vous n'avez pas à vous en soucier. Un ordre des transactions accepté dans le monde entier sera sélectionné pour vous, résolvant ainsi le conflit. Les transactions seront regroupées dans ce que l'on appelle un « bloc », puis elles seront exécutées et réparties entre tous les nœuds participants. Si deux transactions se contredisent, celle qui finit deuxième 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 ». Des blocs sont ajoutés à la chaîne à des intervalles assez réguliers - pour Ethereum, c'est à peu près toutes les 17 secondes.

Dans le cadre du mécanisme de sélection d'ordre (qu'on appelle « mining »), il peut arriver que des blocs soient retournés de temps à autre, mais seulement au « sommet » de la chaîne. Plus il y a de blocs ajoutés au-dessus d'un bloc particulier, moins il y a de chances que ce bloc soit retourné. Il se peut donc que vos transactions soient annulées et même supprimées de la blockchain, mais plus vous attendez, moins il est probable qu'elles le soient.

Note : Il n'est pas garanti que les transactions seront incluses dans le bloc suivant ou dans tout bloc futur spécifique, puisque ce n'est pas à l'auteur d'une transaction, mais aux mineurs de déterminer dans quel bloc la transaction est incluse.

Si vous voulez programmer des appels futurs de votre contrat, vous pouvez utiliser le service "alarm clock <http://www.ethereum-alarm-clock.com/>‘_ ou un service oracle similaire.

4.1.3 La Machine Virtuelle Ethereum

Définition

La Machine Virtuelle Ethereum ou EVM est l'environnement d'exécution des contrats intelligents dans Ethereum. Il n'est pas seulement cloisonné, il est aussi complètement isolé, ce qui signifie que le code fonctionnant à l'intérieur de 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 à d'autres Smart Contracts.

Comptes

Il y a deux types de comptes dans Ethereum qui partagent le même espace d'adresses : **Comptes externes** qui sont contrôlés par des paires de clés public-privé (c'est-à-dire des humains) et **comptes contrats** qui sont contrôlés par le code stocké avec le compte.

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

Indépendamment du fait 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 base de données persistante de clés-valeurs qui associe des mots de 256 bits à des mots de 256 bits appelée **storage**.

De plus, chaque compte a une **balance** en Ether (dans « Wei » pour être exact, *1 ether* est *10**18 wei*) qui peut être modifié en envoyant des transactions qui incluent des Ether.

Transactions

Une transaction est un message envoyé d'un compte à un autre (qui peut être identique ou vide, voir ci-dessous). Il peut inclure des données binaires (ce qu'on appelle charge utile ou « payload ») et de l'éther.

Si le compte cible contient du code, ce code est exécuté et le payload est fourni comme données d'entrée.

Si le compte cible n'est pas défini (la transaction n'a pas de destinataire ou le destinataire est défini sur `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'adresse de l'expéditeur et de son nombre de transactions envoyées (le « nonce »). Le payload d'une telle transaction de création de contrat est considérée comme étant du bytecode EVM et exécuté. Les données de sortie de cette exécution sont stockées en permanence comme code du contrat. Cela signifie que pour créer un contrat, vous n'envoyez pas le code réel du contrat, mais en fait un code qui retourne ce code lorsqu'il est exécuté.

Note : Pendant la création d'un contrat, son code est toujours vide. Pour cette raison, vous ne devez pas rappeler le contrat en cours de construction tant que son constructeur n'a pas terminé son exécution.

Gas

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

Le **gas price** (prix du gas) est une valeur fixée par le créateur de la transaction, qui doit payer `gas_price * gas` à l'avance à partir du compte émetteur. S'il reste du gaz après l'exécution, il est remboursé au créateur de la même manière.

Si le gaz est épuisé à n'importe quel moment (c'est-à-dire qu'il serait négatif), une exception « à court de gas » est déclenchée, qui annule toutes les modifications apportées à l'état dans la trame d'appel en cours.

Storage, Memory et la Stack

La machine virtuelle Ethereum dispose de trois zones où elle peut stocker les données, stockage (« storage »), la mémoire (« memory ») et la pile (« stack »), qui sont expliquées dans les paragraphes suivants.

Chaque compte possède une zone de données appelée **storage**, qui est persistante entre les appels de fonction et les transactions. Storage est un stockage de valeur clé qui mappe les mots de 256 bits en 256 bits. Il n'est pas possible d'énumérer storage à partir d'un contrat et il est comparativement coûteux à lire, et encore plus à modifier le storage. Un contrat ne peut ni lire ni écrire dans un storage autre que le sien.

La deuxième zone de données est appelée **memory**, dont un contrat obtient une instance fraîchement rapprochée pour chaque appel de message. La mémoire est linéaire et peut être adressée au niveau de l'octet, mais les lectures sont limitées à une largeur de 256 bits, tandis que les écritures peuvent être de 8 bits ou de 256 bits. La mémoire est augmentée d'un mot (256 bits), lors de l'accès (en lecture ou en écriture) à un mot de mémoire qui n'a pas été touché auparavant (c.-à-d. tout décalage dans un mot). Au moment de l'agrandissement, le coût en gaz doit être payé. La mémoire est d'autant plus coûteuse qu'elle s'agrandit (le coût grandit de façon quadratique).

L'EVM n'est pas une machine à registre mais une machine à pile, donc tous les calculs sont effectués sur une zone de données appelée la **stack**. Elle a une taille maximale de 1024 éléments et contient des mots de 256 bits. L'accès à la stack est limitée à l'extrémité supérieure de la façon suivante : Il est possible de copier l'un des 16 éléments les plus

hauts au sommet de la stack ou d'inverser l'élément le plus en haut avec l'un des 16 éléments en dessous. Toutes les autres opérations prennent les deux éléments les plus hauts (ou un, ou plus, selon l'opération) de la stack et poussent le résultat sur la stack. Bien sûr, il est possible de déplacer les éléments de la pile vers le stockage ou la mémoire afin d'obtenir un accès plus profond à la stack, mais il n'est pas possible d'accéder à des éléments arbitraires plus profondément dans la stack sans d'abord en enlever le haut.

Jeu d'Instructions

Le jeu d'instructions de l'EVM est maintenu au minimum afin d'éviter des impl'ementations incorrectes ou incohérentes qui pourraient causer des problèmes de consensus. Toutes les instructions fonctionnent sur le type de données de base, les mots de 256 bits ou sur des tranches de mémoire (ou d'autres tableaux d'octets). Les opérations arithmétiques, binaires, logiques et de comparaison habituelles sont présentes. Des 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 :ref :‘ liste des opcodes <opcodes>‘ dans la documentation de l'insertion de langage assembleur.

Les Message Calls

Les contrats peuvent appeler d'autres contrats ou envoyer des Ether sur des comptes non contractuels par le biais d'appels de messages (« message calls »). Les Message Calls sont similaires aux transactions, en ce sens qu'ils ont une source, une cible, une charge utile de données, d'éventuels Ether, le gas et le retour. En fait, chaque transaction consiste en un message call de niveau supérieur qui, à son tour, peut créer d'autres message calls.

Un contrat peut décider de la quantité de **gas** qu'il doit envoyer avec l'appel de message interne et de la quantité qu'il souhaite conserver. Si une exception fin de gas se produit dans l'appel interne (ou toute autre exception), elle sera signalée par une valeur d'erreur placée sur la stack. Dans ce cas, seul le gas envoyé avec l'appel est épuisé. Dans Solidity, le contrat appelant provoque une exception manuelle par défaut dans de telles situations, de sorte que les exceptions « remontent en surface » de la pile d'appels.

Comme déjà dit, le contrat appelé (qui peut être le même que celui de l'appelant) recevra une instance de mémoire fraîchement effacée et aura accès à la charge utile de l'appel - qui sera fournie dans une zone séparée appelée **calldata**. Une fois l'exécution terminée, il peut renvoyer des données qui seront stockées à un emplacement de 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 les opérations plus complexes, les boucles doivent être préférées aux appels récursifs. De plus, seul 63/64ème du gaz peut être transféré lors d'un appel de message, ce qui entraîne une limite de profondeur d'un peu moins de 1000 en pratique.

Delegatecall / Callcode et Libraries

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

Cela signifie qu'un contrat peut charger dynamiquement du code à partir d'une adresse différente lors de l'exécution. Le stockage, l'adresse actuelle et le solde se réfèrent toujours au contrat d'appel, seul le code est repris de l'adresse appelée.

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

Logs / Journalisation

Il est possible de stocker les données dans une structure de données spécialement indexée qui s'étend jusqu'au niveau du bloc. Cette fonction appelée **logs** (journalisation) est utilisé par Solidity pour implémenter les *events*. Les contrats ne peuvent pas accéder aux données du journal une fois qu'elles ont été créées, mais ils peut être accédé efficacement de l'extérieur de la chaîne de blocs. Puisqu'une partie des données du journal est stockée dans des bloom filters, il est possible de rechercher ces données de manière efficace et cryptographique de manière sécurisée, afin que les pairs du réseau qui ne téléchargent pas la totalité de la blockchain (appelés « clients légers ») peuvent encore trouver ces logs.

Création

Les contrats peuvent même créer d'autres contrats à l'aide d'un opcode spécial (càd qu'ils n'appellent pas simplement l'adresse zéro comme le ferait une transaction). La seule différence entre ces **appels de création** et des appels de message normaux est que les données de charge utile sont exécutées, le résultat stocké sous forme de code et l'appelant / créateur reçoit l'adresse du nouveau contrat sur la stack.

Désactivation et Auto-Destruction

La seule façon de supprimer du code de la blockchain est lorsqu'un contrat à cette adresse exécute l'opération d'auto-destruction `selfdestruct`. L'Ether restant stocké à cette adresse est envoyé à une cible désignée, puis le stockage et le code sont retirés de l'état. Supprimer le contrat en théorie semble être une bonne idée, mais c'est potentiellement dangereux, comme en cas d'envoi d'éther à des contrats supprimés, où l'éther est perdu à jamais.

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

Si vous souhaitez désactiver vos contrats, vous devez plutôt les désactiver en changeant un état interne qui provoque un échec (`revert`) de toutes les fonctions. Il est donc impossible d'utiliser le contrat, car il retourne immédiatement l'éther.

Avertissement : Même si un contrat est supprimé par `selfdestruct`, il fait toujours partie de l'historique de la blockchain et probablement conservé par la plupart des nœuds Ethereum. L'utilisation de l'autodestruction n'est donc pas la même chose que la suppression de données d'un disque dur.

4.2 Installer le Compilateur Solidity

4.2.1 Versionnage

Les versions de Solidity suivent un versionnage sémantique et en plus des versions stables, des versions de développement **nightly** sont également disponibles. Les versions nightly ne sont pas garanties de fonctionner et malgré tous les efforts, elles peuvent contenir des changements non documentés et/ou cassés. Nous vous recommandons d'utiliser la dernière version. Les installateurs de paquets ci-dessous utilisent la dernière version.

4.2.2 Remix

Nous recommandons Remix pour les petits contrats et pour l'apprentissage rapide de Solidity.

Accédez à Remix en ligne, vous n'avez rien à installer. Si vous voulez l'utiliser sans connexion à Internet, allez à https://github.com/ethereum/remix-live/tree/gh-pages et téléchargez le fichier `.zip` tel qu'expliqué sur cette page.

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

4.2.3 npm / Node.js

Utilisez *npm" pour un moyen pratique et portable d'installer 'solcjs", un compilateur Solidity. Le programme 'solcjs* a moins de fonctionnalités que le compilateur décrit plus bas sur cette page. La documentation du *Using the Commandline Compiler* suppose que vous utilisez le compilateur complet, *solc*. L'utilisation de *solcjs* est documentée dans son propre dépot.

Note : Le projet solc-js est dérivé du projet C++ *solc* en utilisant Emscripten, ce qui signifie que les deux utilisent le même code source du compilateur. 'solc-js" peut être utilisé directement dans les 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 de la ligne de commande de 'solcjs" ne sont pas compatibles avec 'solc" et les outils (tels que 'geth") attendant le comportement de 'solc" ne fonctionneront pas avec 'solcjs".

4.2.4 Docker

Nous fournissons des images dockers à jour pour le compilateur. Le dépot `stable` contient les versions publiées tandis que le dépôt `nightly` contient des changements potentiellement instables dans la branche develop.

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

Actuellement, l'image du docker ne contient que l'exécutable du compilateur, donc vous devez faire un peu plus de travail pour lier le code source et répertoires de sortie.

4.2.5 Paquets binaires

Les binaires de Solidity sont disponibles à solidity/releases.

Nous avons également des PPAs for Ubuntu, vous pouvez obtenir la dernière version via la commande :

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

La version nigthly peut s'installer avec la commande :

```
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 package snap, installable dans toutes les distributionss linux supportées. Pour installer la dernière evrsion stable de solc :

```
sudo snap install solc
```

Si vous voulez aider aux tests en utilisant la dernière version de développement, avec les changements l ;es plus récents, merci d'utiliser :

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

Arch Linux a aussi des paquets, bien que limités à la dernière version de développement :

```
pacman -S solidity
```

Nous distribuons également le compilateur Solidity via homebrew dans une version compilée à partir des sources. Les « bottles » pré-compilées ne sont pas encore supportées pour l'instant.

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

Si vous avec besoin d'une version spécifique, vous pouvez exécuter la formule homebrew correspondante disponible sur GitHub.

Regarder commits de solidity.rb sur Github.

Suivez l'historique des liens jusqu'à avoir un lien de ficher brut (« raw ») d'un commit spécifique de `solidity.rb`.

Installez-le via `brew` :

```
brew unlink solidity
# Install 0.4.8
brew install https://raw.githubusercontent.com/ethereum/homebrew-ethereum/
↪77cce03da9f289e5a3ffe579840d3c5dc0a62717/solidity.rb
```

Gentoo Linux dispose aussi d" un paquet Solidity installable via `emerge` :

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

4.2.6 Compilation à partir des sources

Prérequis - Linux

Vous aurez besoin des dépendances suivantes pour les compilations de Solidity sous Linux :

Software	Notes
Git pour Linux	Outils en ligne de commande pour r'ecup'erer des fichiers sur github

Prérequis - macOS

Pour macOS, assurez-vous d'avoir installer la dernière version de Xcode. Ceci contient le compilateur C++ Clang, l'IDE Xcode et d'autres outils de développement Apple qui sont nécessaires pour construire des applications C++ sous OS X. Si vous installez Xcode pour la première fois, ou si vous venez d'installer une nouvelle version, vous devrez accepter la licence avant de pouvoir compiler en ligne de commande :

```
sudo xcodebuild -license accept
```

Nos versions pour OS X exigent que vous installiez Homebrew <http://brew.sh>‘_http://brew.sh pour l’installation des dépendances externes. Voici comment ‘désinstaller Homebrew, si vous voulez recommencer à zéro.

Prérequis - Windows

Vous aurez besoin des dépendances suivants pour la compilation de solidity sous Windows :

Software	Notes
Git pour Linux	Outils en ligne de commande pour r’ecup’erer des fichiers sur github
CMake	Générateur de fichiers d’installation multi-plateformes
Visual Studio 2017 Build Tools	Compilateur C++
Visual Studio 2017 (Optional)	Environment de développement et compilateur C++.

Si vous avez déjà eu un IDE et que vous n’avez besoin que du compilateur et des bibliothèques, vous pouvez installer Visual Studio 2017 Build Tools.

Visual Studio 2017 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 en Solidity, Visual Studio 2017 peut être un choix pour tout installer facilement.

Voici la liste des composants à installer dans Visual Studio 2017 Build Tools ou Visual Studio 2017 :
- Visual Studio C+++ fonctionnalités de base
- VC+++ 2017 v141 toolset (x86,x64)
- Windows Universal CRT SDK
- Windows 8.1 SDK
- Support C+++/CLI

Clonez le dépot

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 devriez forker Solidity et ajouter votre fork comme un second remote (dépot distant) :

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

Solidity a des submodules Git. Vérifiez qu’ils sont proprement chargés :

```
git submodule update --init --recursive
```

Dépendances externes

Nous avons un script d’aide qui installe toutes les dépendances externes requises sur macOS, Windows et de nombreuses distributions Linux.

```
./scripts/install_deps.sh
```

Ou, sous Windows :

```
scripts\install_deps.bat
```

Compilation en ligne de commande

Soyez sûrs d'installer les dépendances externes avant de compiler.

Le projet Solidity utilise CMake pour la configuration de compilation. Vous voulez peut-être installer ccache pour accélérer des compilations successives. CMake l'utilisera automatiquement. Compiler Solidity est similaire sur Linux, macOS et autres systèmes Unix :

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

ou encore plus simplement :

```
#note: les binaires de solc et les tests seront installés dans usr/local/bin
./scripts/build.sh
```

Et pour Windows :

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
```

Ce dernier ensemble d'instructions devrait aboutir à la création de **solidity.sln** dans ce répertoire de compilation. Double-cliquer sur ce fichier devrait faire démarrer Visual Studio. Nous suggérons de construire la configuration **RelWithDebugInfo**, mais toutes les autres fonctionnent.

Alternativement, vous pouvez compiler pour Windows en ligne de commande, comme ça :

```
cmake --build . --config RelWithDebInfo
```

4.2.7 Options de CMake

La liste des options de Cmake est disponible via la commande : `cmake .. -LH`.

Solveurs SMT

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

Remarque : Dans certains cas, cela peut également être une solution de contournement potentielle en cas d'échec de compilation.

Dans le dossier de compilation, vous pouvez les désactiver, car ils sont activés par défaut :

```
# désactive seulement Z3 SMT Solver.
cmake .. -DUSE_Z3=OFF

# désactive seulement CVC4 SMT Solver.
cmake .. -DUSE_CVC4=OFF
```

(suite sur la page suivante)

(suite de la page précédente)

```
# désactive Z3 et CVC4
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

4.2.8 La string de version en détail

La string de version de Solidity contient 4 parties :

— le numéro de version
— la balise de pre-version, généralement définie sur `develop.YYYY.MM.DD` ou `nightly.YYYY.MM.DD`.
— commit au format `commit.GITHASH`.
— plate-forme, qui a 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 suffixé avec `.mod`.

Ces parties sont combinées comme l'exige Semver, où la balise de pré-version Solidity est identique à la pré-version de Semver. et le commit Solidity et la plate-forme Solidity combinés constituent les métadonnées de la construction Semver.

Un exemple de version : `` `0.4.8+commit.60cc1668.Emscripten.clang ``.

Un exemple de pré-version : `0.4.9-nightly.2017.1.17+commit.6ecb4aaa3.Emscripten.clang`

4.2.9 Informations importantes concernant le versionnage

Après la sortie d'une version, la version de correctif est incrémentée, parce que nous supposons que seulement les changements de niveau patch suivent. Lorsque les modifications sont fusionnées, la version doit être supprimée en fonction des éléments suivants et la gravité du changement. Enfin, une version est toujours basée sur la nigthly actuelle, mais sans le spécificateur `prerelease`.

Exemple :

0. la version 0.4.0 est faite
1. nightly build a une version de 0.4.1 à partir de maintenant
2. des modifications incessantes sont introduites - pas de changement de version
3. un changement de rupture est introduit - la version est augmentée à 0.5.0
4. la version 0.5.0 est faite

Ce comportement fonctionne bien avec le *version pragma*.

4.3 Solidity par l'Exemple

4.3.1 Vote

Le contrat suivant est assez complexe, mais il présente de nombreuses caractéristiques de Solidity. Il implémente un contrat de vote. Bien entendu, le principal problème du vote électronique est de savoir comment attribuer les droits de vote aux bonnes personnes et éviter les manipulations. Nous ne résoudrons pas tous les problèmes ici, mais nous montrerons au moins comment le vote délégué peut être effectué de manière à ce que le dépouillement soit à la fois **automatique et totalement transparent**.

L'idée est de créer un contrat par bulletin de vote, en donnant un nom court à chaque option. Ensuite, le créateur du contrat qui agit à titre de président donnera le droit de vote à chaque adresse individuellement.

Les personnes derrière les adresses peuvent alors choisir de voter elles-mêmes ou de déléguer leur vote à une personne en qui elles ont confiance.

A la fin du temps de vote, la `winningProposal()` (proposition gagnante) retournera la proposition avec le plus grand nombre de votes.

```
pragma solidity >=0.4.22 <0.6.0;

/// @title Vote par délegation.
contract Ballot {
    // Ceci déclare un type complexe, représentant
    // un votant, qui sera utilisé
    // pour les variables plus tard.
    struct Voter {
        uint weight; // weight (poids), qui s'accumule avec les délégations
        bool voted;  // si true, cette personne a déjà voté
        address delegate; // Cette personne a délégué son vote à
        uint vote;   // index la la proposition choisie
    }

    // Type pour une proposition.
    struct Proposal {
        bytes32 name;   // nom court (jusqu'à 32 octets)
        uint voteCount; // nombre de votes cumulés
    }

    address public chairperson;

    // Ceci déclare une variable d'état qui stocke
    // un élément de structure 'Voters' pour  chaque votant.
    mapping(address => Voter) public voters;

    // Un tableau dynamique de structs `Proposal`.
    Proposal[] public proposals;

    /// Créé un nouveau bulletin pour choisir l'un des `proposalNames`.
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // Pour chacun des noms proposés,
        // crée un nouvel objet proposal
        // à la fin du tableau.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` créé un objet temporaire
            // Proposal et `proposals.push(...)`
            // l'ajoute à la fin du tableau `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Donne à un `voter` un droit de vote pour ce scrutin.
    // Peut seulement être appelé par `chairperson`.
    function giveRightToVote(address voter) public {
        // Si le premier argument passé à `require` s'évalue
        // à `false`, l'exécution s'arrete et tous les changements
        // à l'état et aux soldes sont annulés.
        // Cette opération consommait tout le gas dans
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // d'anciennes versions de l'EVM, plus maintenant.
        // Il est souvent une bonne idée d'appeler `require`
        // pour vérifier si les appels de fonctions
        // s'effectuent correctement.
        // Comme second argument, vous pouvez fournir une
        // phrase explicative de ce qui est allé de travers.
        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;
    }

    /// Delegue son vote au votant `to`.
    function delegate(address to) public {
        // assigne les références
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

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

        // Relaie la délégation tant que `to`
        // est également en délégation de vote.
        // En général, ce type de boucles est très dangereux,
        // puisque s'il tourne trop longtemps, l'opération
        // pourrait demander plus de gas qu'il n'est possible
        // d'en avoir dans un bloc.
        // Dans ce cas, la délégation ne se ferait pas,
        // mais dans d'autres circonstances, ces boucles
        // peuvent complètement paraliser un contrat.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // On a trouvé une boucle dans la chaine
            // de délégations => interdit.
            require(to != msg.sender, "Found loop in delegation.");
        }

        // Comme `sender` est une référence, ceci
        // modifie `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // Si le délégué a déjà voté,
            // on ajoute directement le vote aux autres
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // Sinon, on l'ajoute au poids de son vote.
            delegate_.weight += sender.weight;
        }
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /// Voter (incluant les procurations par délégation)
    /// pour la proposition `proposals[proposal].name`.
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // Si `proposal` n'est pas un index valide,
        // une erreur sera levée et l'exécution annulée
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Calcule la proposition gagnante
    /// en prenant tous les votes précédents en compte.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Appelle la fonction winningProposal() pour avoir
    // l'index du gagnant dans le tableau de propositions
    // et retourne le nom de la proposition gagnante.
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

Améliorations possibles

À l'heure actuelle, de nombreuses opérations sont nécessaires pour attribuer les droits de vote à tous les participants. Pouvez-vous trouver un meilleur moyen ? .. index : : auction ;blind, auction ;open, blind auction, open auction

4.3.2 Enchères à l'aveugle

Dans cette section, nous allons montrer à quel point il est facile de créer un contrat d'enchères à l'aveugle sur Ethereum. Nous commencerons par une enchère ouverte où tout le monde pourra voir les offres qui sont faites, puis nous prolongerons ce contrat dans une enchère aveugle où il n'est pas possible de voir l'offre réelle avant la fin de la période de soumission.

Enchère ouverte simple

L'idée générale du contrat d'enchère simple suivant est que chacun peut envoyer ses offres pendant une période d'enchère. Les ordres incluent l'envoi d'argent / éther afin de lier les soumissionnaires à leur offre. Si l'enchère est la plus haute, l'enchérisseur qui avait fait l'offre la plus élevée auparavant récupère son argent. Après la fin de la période de soumission, le contrat doit être appelé manuellement pour que le bénéficiaire reçoive son argent - les contrats ne peuvent pas s'activer eux-mêmes.

```
pragma solidity >=0.4.22 <0.6.0;

contract SimpleAuction {
    // Paramètres de l'enchère
    // temps unix absolus (secondes depuis 01-01-1970)
    // ou des durées en secondes.
    address payable public beneficiary;
    uint public auctionEndTime;

    // État actuel de l'enchère.
    address public highestBidder;
    uint public highestBid;

    // Remboursements autorisés d'enchères précédentes
    mapping(address => uint) pendingReturns;

    // Mis à true à la fin, interdit tout changement.
    // Par defaut à `false`, comme un grand.
    bool ended;

    // Évènements déclenchés aux changements.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Ce ui suit est appelé commentaire natspec,
    // reconaissable à ses 3 slashes.
    // Ce message sera affiché quand l'utilisateur
    // devra confirmer une transaction.

    /// Créée une enchère simple de `_biddingTime`
    /// secondes au profit de l'addresse
    /// beneficaire address `_beneficiary`.
    constructor(
        uint _biddingTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEndTime = now + _biddingTime;
    }

    /// Faire une offre avec la valeur envoyée
    /// avec cette transaction.
    /// La valeur ne sera remboursée que si
    // l'enchère est perdue.
    function bid() public payable {
        // Aucun argument n'est nécessaire, toute
        // l'information fait déjà partie
        // de la transaction. Le mot-clé payable
        // est requis pour autoriser la fonction
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // à recevoir de l'Ether.

        // Annule l'appel si l'enchère est termminée
        require(
            now <= auctionEndTime,
            "Auction already ended."
        );

        // Rembourse si l'enchère est trop basse
        require(
            msg.value > highestBid,
            "There already is a higher bid."
        );

        if (highestBid != 0) {
            // Renvoyer l'argent avec un simple
            // highestBidder.send(highestBid) est un risque de sécurité
            // car ça pourrait déclencher un appel à un contrat.
            // Il est toujours plus sûr de laisser les utilisateurs
            // retirer leur argent eux-mêmes.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Retirer l'argent d'une enchère dépassée
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // Il est important de mettre cette valeur à zéro car l'utilisateur
            // pourrait rappeler cette fonction avant le retour de `send`.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // Pas besoin d'avorter avec un throw ici, juste restaurer le montant
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// Met fin à l'enchère et envoie
    /// le montant de l'enchère la plus haute au bénéficiaire.
    function auctionEnd() public {
        // C'est une bonne pratique de structurer les fonctions qui
        // intéragissent avec d'autres contrats (appellent des
        // fonctions ou envoient de l'Ether) en trois phases:
        // 1. Vérifier les conditions
        // 2. éffectuer les actions (potentiellement changeant les conditions)
        // 3. interagir avec les autres contrats
        // Si ces phases sont mélangées, l'autre contrat pourrait rappeler
        // le contrat courant et modifier l'état ou causer des effets
        // (paiements en Ether par ex) qui se produiraient plusieurs fois.
        // Si des fonctions appelées en interne effectuent des appels
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // à des contrats externes, elles doivent aussi êtres considérées
        // comme concernées par cette norme.

        // 1. Conditions
        require(now >= auctionEndTime, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");

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

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

Enchère aveugle

L'enchère ouverte précédente est étendue en une enchère aveugle dans ce qui suit. L'avantage d'une enchère aveugle est qu'il n'y a pas de pression temporelle vers la fin de la période de soumission. La création d'une enchère aveugle sur une plate-forme informatique transparente peut sembler une contradiction, mais la cryptographie vient à la rescousse.

Pendant la **période de soumission**, un soumissionnaire n'envoie pas son offre, mais seulement une version hachée de celle-ci. Puisqu'il est actuellement considéré comme pratiquement impossible de trouver deux valeurs (suffisamment longues) dont les valeurs de hachage sont égales, le soumissionnaire s'engage à l'offre par cela. Après la fin de la période de soumission, les soumissionnaires doivent révéler leurs offres : Ils envoient leurs valeurs en clair et le contrat vérifie que la valeur de hachage est la même que celle fournie pendant la période de soumission.

Un autre défi est de savoir comment rendre l'enchère contraignante et aveugle en même temps : La seule façon d'éviter que l'enchérisseur n'envoie pas l'argent après avoir gagné l'enchère est de le lui faire envoyer avec l'enchère. Puisque les transferts de valeur ne peuvent pas être aveuglés dans Ethereum, tout le monde peut voir la valeur.

Le contrat suivant résout ce problème en acceptant toute valeur supérieure à l'offre la plus élevée. Comme cela ne peut bien sûr être vérifié que pendant la phase de révélation, certaines offres peuvent être invalides, et c'est fait exprès (il fournit même un marqueur explicite pour placer des offres invalides avec des transferts de grande valeur) : Les soumissionnaires peuvent brouiller la concurrence en plaçant plusieurs offres invalides hautes ou basses.

```
pragma solidity >0.4.23 <0.6.0;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

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

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

    address public highestBidder;
    uint public highestBid;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Remboursements autorisés d'enchères précédentes
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Les Modifiers sont une façon pratique de valider des entrées.
    /// `onlyBefore` est appliqué à `bid` ci-dessous:
    /// Le corps de la fonction sera placé dans le modifier
    /// où `_` est placé.
    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }

    constructor(
        uint _biddingTime,
        uint _revealTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    /// Placer une enchère à l'aveugle avec `_blindedBid` =
    /// keccak256(abi.encodePacked(value, fake, secret)).
    ///  L'éther envoyé n'est remboursé que si l'enchère est correctement
    /// révélée dans la phase de révélation. L'offre est valide si
    /// l'éther envoyé avec l'offre est d'au moins "valeur" et
    /// "fake" n'est pas true. Régler "fake" à true et envoyer
    /// envoyer un montant erroné sont des façons de masquer l'enchère
    /// mais font toujours le dépot requis. La même addresse peut placer
    /// plusieurs ordres
    function bid(bytes32 _blindedBid)
        public
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

    /// Révèle vos ench1eres aveugles. Vous serez remboursé pour toutes
    /// les enchères invalides et toutes les autres exceptée la plus haute
    /// le cas échéant.
    function reveal(
        uint[] memory _values,
        bool[] memory _fake,
        bytes32[] memory _secret
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        require(_secret.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], _fake[i], _secret[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,␣
↪secret))) {
                // L'enchère n'a pas été révélée.
                // Ne pas rembourser.
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Rendre impossible un double remboursement
            bidToCheck.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }

    // Cette fonction interne ("internal") ne peut être appelée que
    // que depuis l'intérieur du contrat (ou ses contrats dérivés).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Rembourse la précédent leader.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// Se faire rembourser une enchère battue.
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // Il est important de mettre cette valeur à zéro car l'utilisateur
            // pourrait rappeler cette fonction avant le retour de `send`.
            // (voir remarque sur conditions -> effets -> interaction).
            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    /// Met fin à l'enchère et envoie
    /// le montant de l'enchère la plus haute au bénéficiaire.
    function auctionEnd()
```

(suite sur la page suivante)

(suite de la page précédente)

```
        public
        onlyAfter(revealEnd)
    {
        require(!ended);
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

4.3.3 Achat distant sécurisé

```
pragma solidity >=0.4.22 <0.6.0;

contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    // Vérifie que `msg.value` est un nombre pair.
    // La division tronquerait un nombre impair.
    // On multiplie pour vérifier que ce n'était pas un impair.
    constructor() public payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value, "Value has to be even.");
    }

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(
            msg.sender == buyer,
            "Only buyer can call this."
        );
        _;
    }

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

    modifier inState(State _state) {
        require(
            state == _state,
            "Invalid state."
```

(suite sur la page suivante)

(suite de la page précédente)

```
        );
        _;
    }

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

    /// Annule l'achat et rembourse l'ether du dépot.
    /// Peut seulement être appelé par le vendeur
    /// avant le verrouillage du contrat
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        seller.transfer(address(this).balance);
    }

    /// Confirme l'achat en tant qu'acheteur.
    /// La transaction doit inclure `2 * value` ether.
    /// L'Ether sera bloqué jusqu'à ce que confirmReceived
    /// soit appelé.
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// Confirmer que vous (l'acheteur) avez reçu l'objet,
    /// ce qui débloquera l'Ether bloqué.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // Il est important de changer l'état d'abord car sinon
        // les contrats appelés avec `send` ci-dessous
        // pourraient rappeler la fonction.
        state = State.Inactive;

        // NOTE: Ce schéma autorise les deux acteurs à bloquer
        // la transaction par une exception "our of gas" ( pas
        // assez de gas). Un fonction de retrait distincte devrait
        // être utilisée.

        buyer.transfer(value);
        seller.transfer(address(this).balance);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

4.3.4 Canaux de micro-paiement

Dans cette section, nous allons apprendre comment construire une implémentation simple d'un canal de paiement. Il utilise des signatures cryptographiques pour effectuer des transferts répétés d'Ether entre les mêmes parties en toute sécurité, instantanément et sans frais de transaction. Pour ce faire, nous devons comprendre comment signer et vérifier les signatures, et configurer le canal de paiement.

Création et vérification des signatures

Imaginez qu'Alice veuille envoyer une quantité d'Ether à Bob, c'est-à-dire qu'Alice est l'expéditeur et Bob est le destinataire. Alice n'a qu'à envoyer des messages cryptographiquement signés hors chaîne (par exemple par e-mail) à Bob et cela sera très similaire à la rédaction de chèques.

Les signatures sont utilisées pour autoriser les transactions et sont un outil généraliste à la disposition des contrats intelligents. Alice construira un simple contrat intelligent qui lui permettra de transmettre des Ether, mais d'une manière inhabituelle, au lieu d'appeler une fonction elle-même pour initier un paiement, elle laissera Bob le faire, et donc payer les frais de transaction. Le contrat fonctionnera comme suit :

1. Alice déploie le contrat `ReceiverPays` en y attachant suffisamment d'éther pour couvrir les paiements qui seront effectués.
2. Alice autorise un paiement en signant un message avec sa clé privée.
3. Alice envoie le message signé cryptographiquement à Bob. Le message n'a pas besoin d'être gardé secret (vous le comprendrez plus tard), et le mécanisme pour l'envoyer n'a pas d'importance.
4. Bob réclame leur paiement en présentant le message signé au contrat intelligent, il vérifie l'authenticité du message et libère ensuite les fonds.

Création de la signature

Alice n'a pas besoin d'interagir avec le réseau Ethereum pour signer la transaction, le processus est complètement hors ligne. Dans ce tutoriel, nous allons signer les messages dans le navigateur en utilisant `web3.js` et `MetaMask`. En particulier, nous utiliserons la méthode standard décrite dans EIP-762, car elle offre un certain nombre d'autres avantages en matière de sécurité.

```
/// Hasher d'abord simplifie un peu les choses
var hash = web3.sha3("message to sign");
web3.personal.sign(hash, web3.eth.defaultAccount, function () {...});
```

Notez que `web3.personal.sign` préfixe les données signées de la longueur du message. Mais comme nous avons hashé en premier, le message sera toujours exactement 32 octets de long, et donc ce préfixe de longueur est toujours le même, ce qui facilite tout.

Que signer

Dans le cas d'un contrat qui effectue des paiements, le message signé doit inclure :

1. Adresse du destinataire
2. le montant à transférer
3. Protection contre les attaques de rediffusion

Une attaque de rediffusion se produit lorsqu'un message signé est réutilisé pour revendiquer l'autorisation pour une deuxième action. Pour éviter les attaques par rediffusion, nous utiliserons la même méthode que pour les transactions Ethereum elles-mêmes, ce qu'on appelle un nonce, qui est le nombre de transactions envoyées par un compte. Le contrat intelligent vérifiera si un nonce est utilisé plusieurs fois.

Il existe un autre type d'attaques de redifussion, il se produit lorsque le propriétaire déploie un smart contract `ReceiverPays`, effectue certains paiements, et ensuite détruit le contrat. Plus tard, il décide de déployer `ReceiverPays` encore une fois, mais le nouveau contrat ne peut pas connaître les nonces utilisés dans le déploiement précédent, donc l'attaquant peut réutiliser les anciens messages.

Alice peut s'en protéger, notamment en incluant l'adresse du contrat dans le message, et seulement les messages contenant l'adresse du contrat lui-même seront acceptés. Cette fonctionnalité se trouve dans les deux premières lignes de la fonction `claimPayment()` du contrat complet à la fin de ce chapitre.

Encoder les arguments

Maintenant que nous avons déterminé quelles informations inclure dans le message signé, nous sommes prêts à assembler le message, à le hacher, et le signer. Par souci de simplicité, nous ne faisons que concaténer les données. La bibliothèque ethereumjs-abi fournit une fonction appelée `soliditySHA3` qui imite le comportement de la fonction `keccak256` de Solidity appliquée aux arguments codés en utilisant `abi.encododePacked`. En résumé, voici une fonction JavaScript qui crée la signature appropriée pour l'exemple `ReceiverPays` :

```
// recipient est l'addresse à payer.
// amount, en wei, spécifie combien d'Ether doivent être envoyés.
// nonce peut être n'importe quel nombre unique pour prévenir les attques par␣
↪redifusion
// contractAddress est utilisé pour éviter les attaque par redifusion de messages␣
↪inter-contrats
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + ethereumjs.ABI.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

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

Récupérer le signataire du message en Solidity

En général, les signatures ECDSA se composent de deux paramètres, `r` et `s`. Les signatures dans Ethereum incluent un troisième paramètre appelé `v`, qui peut être utilisé pour récupérer la clé privée du compte qui a été utilisée pour signer le message, l'expéditeur de la transaction. Solidity fournit une fonction intégrée ecrecover qui accepte un message avec les paramètres `r`, `s` et `v` et renvoie l'adresse qui a été utilisée pour signer le message.

Récupérer le signataire du message en Solidity

En général, les signatures ECDSA se composent de deux paramètres, `r` et `s`. Les signatures dans Ethereum incluent un troisième paramètre appelé « v », qui peut être utilisé pour récupérer la clé privée du compte qui a été utilisée pour signer le message, l'expéditeur de la transaction. La solidité offre une fonction intégrée Récupérer <fonctions-mathématiques et cryptographiques>'_ qui accepte un message avec les paramètres `r`, `s` et `v` et renvoie l'adresse qui a été utilisée pour signer le message.

Extraire les paramètres de signature

Les signatures produites par web3.js sont la concaténation de `r`, `s` et `v`, donc la première étape est de re-séparer ces paramètres. Cela peut être fait sur le client, mais le faire à l'intérieur du smart contract signifie qu'un seul paramètre de signature peut être envoyé au lieu de trois. Diviser un tableau d'octets en plusieurs parties est un peu compliqué. Nous utiliserons l'"assembleur en ligne pour faire le travail dans la fonction `splitSignature` (la troisième fonction dans le contrat complet à la fin du présent chapitre).

Calculer le hash du message

Le smart contract doit savoir exactement quels paramètres ont été signés, et doit donc recréer le message à partir des paramètres et utiliser cette fonction pour la vérification des signatures. Les fonctions `prefixed` et `recoverSigner` s'occupent de cela et leur utilisation peut se trouver dans la fonction `claimPayment`.

Le contrat complet

```
pragma solidity >=0.4.24 <0.6.0;

contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() public payable {}

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

        // Cette ligne recrée le message signé par le client
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount,␣
↪nonce, this)));

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

        msg.sender.transfer(amount);
    }

    /// détruit le contrat et réclame son solde.
    function kill() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }

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

        assembly {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            // premiers 32 octets, après le préfixe
            r := mload(add(sig, 32))
            // 32 octets suivants
            s := mload(add(sig, 64))
            // Octet final (premier du prochain lot de 32)
            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);
    }

    /// construit un hash préfixé pour mimer le comportement de eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Écrire un canal de paiement simple

Alice va maintenant construire une implémentation simple mais complète d'un canal de paiement. Les canaux de paiement utilisent des signatures cryptographiques pour effectuer des virements répétés d'Ether en toute sécurité, instantanément et sans frais de transaction.

Qu'est-ce qu'un canal de paiement ?

Les canaux de paiement permettent aux participants d'effectuer des transferts répétés d'Ether sans utiliser de transactions. Cela signifie que les délais et frais associés aux transactions peuvent être évités. Nous allons explorer un canal de paiement unidirectionnel simple entre deux parties (Alice et Bob). Son utilisation implique trois étapes :

1. Alice déploie un smart contract avec de l'Ether. Cela « ouvre » (`opens`) le canal de paiement.
2. Alice signe des messages qui précisent combien d'éther est dû au destinataire. Cette étape est répétée pour chaque paiement.
3. Bob « ferme » (`closes`) le canal de paiement, retirant leur part d'Ether et renvoyant le reste à l'expéditeur.

Non seulement les étapes 1 et 3 exigent des transactions Ethereum, mais l'étape 2 signifie que l'expéditeur transmet un message signé cryptographiquement au destinataire par des moyens hors chaîne (par exemple, par courrier électronique). Cela signifie que seulement deux transactions sont nécessaires pour traiter un nombre quelconque de transferts.

Bob est assuré de recevoir ses fonds parce que le contrat bloque les fonds en Ether et respecte des ordres valides et signés. Le smart contract impose également un délai d'attente, Alice est donc assurée de recouvrer ses fonds même si le bénéficiaire refuse de fermer le canal. C'est aux participants d'un canal de paiement de décider combien de temps il doit rester ouvert. Pour une transaction de courte durée, comme payer un cybercafé pour chaque minute d'accès au réseau, ou dans le cas d'une relation de plus longue durée, comme le versement d'un salaire horaire à un employé, un paiement pourrait durer des mois ou des années.

Ouverture du canal de paiement

Pour ouvrir le canal de paiement, Alice déploie le contrat, y attachant de l'Ether en dépot et spécifiant le destinataire prévu, ainsi qu'une durée de vie maximale du canal. C'est la fonction `SimplePaymentChannel` dans le contrat.

Effectuer des paiements

Alice effectue des paiements en envoyant des messages signés à Bob. Cette étape est entièrement réalisée en dehors du réseau Ethereum. Les messages sont signés cryptographiquement par l'expéditeur puis transmis directement au destinataire.

Chaque message contient les informations suivantes :

— L'adresse du contrat, utilisé pour empêcher les attaques de redifussion par contrats croisés.
— Le montant total d'Ether qui est dû au bénéficiaire jusqu'alors.

Un canal de paiement est fermé une seule fois, à la fin d'une série de virements. De ce fait, un seul des messages envoyés sera échangé. C'est pourquoi chaque message spécifie un montant total cumulatif d'éther dû, plutôt que le montant total d'un micropaiement individuel. Le destinataire réclamera naturellement le message le plus récent parce que c'est celui dont le total est le plus élevé. Le nonce par message n'est plus nécessaire, car le smart contract ne va honorer qu'un seul message. L'adresse du contrat intelligent est toujours utilisée pour éviter qu'un message destiné à un canal de paiement ne soit utilisé pour un autre canal.

Voici le code javascript modifié pour signer cryptographiquement un message du chapitre précédent :

```
function constructPaymentMessage(contractAddress, amount) {
    return ethereumjs.ABI.soliditySHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

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

// contractAddress détectera la rediffusion de messages à d'autres contrats.
// amount, en wei, précise combien d'Ether doivent être envoyés.

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

Fermeture du canal de paiement

Lorsque Bob est prêt à recevoir leurs ses, il est temps de fermer le canal de paiement en appelant une fonction `close` sur le smart contract. La fermeture du canal paie au destinataire l'Ether qui lui est dû et détruit le contrat, en renvoyant tout Ether restant à Alice. Pour fermer le canal, Bob doit fournir un message signé par Alice.

Le contrat doit vérifier que le message contient une signature valide de l'expéditeur. Le processus de vérification est le même que celui utilisé par le destinataire. Les fonctions Solidity `isValidSignature` et `recoverSigner`

fonctionnent de la même manière que leurs fonctions JavaScript dans la section précédente. Ce dernier est emprunté au Le contrat `ReceiverPays` du chapitre précédent.

La fonction `close` ne peut être appelée que par le destinataire du canal de paiement, qui enverra naturellement le message de paiement le plus récent car c'est celui qui comporte le plus haut total dû. Si l'expéditeur était autorisé à appeler cette fonction, il pourrait fournir un message avec un montant inférieur et escroquer le destinataire de ce qui lui est dû.

La fonction vérifie que le message signé correspond aux paramètres donnés. Si tout se passe bien, le destinataire reçoit sa part d'Ether, et l'expéditeur reçoit le reste par `selfdestruct` (autodestruction) du contrat. Vous pouvez voir la fonction `close` dans le contrat complet.

Expiration du canal

Bob peut fermer le canal de paiement à tout moment, mais s'il ne le fait pas, Alice a besoin d'un moyen de récupérer les fonds bloqués. Une durée d'*expiration* a été définie au moment du déploiement du contrat. Une fois cette heure atteinte, Alice peut appeler pour récupérer leurs fonds. Vous pouvez voir la fonction `claimTimeout` dans le contrat complet.

Après l'appel de cette fonction, Bob ne peut plus recevoir d'Ether. Il est donc important que Bob ferme le canal avant que l'expiration ne soit atteinte.

Le contrat complet

```
pragma solidity >=0.4.24 <0.6.0;

contract SimplePaymentChannel {
    address payable public sender;      // Le compte emvoyant les paiements.
    address payable public recipient;   // Le compte destinataire des paiements.
    uint256 public expiration;  // Expitration si le destinataire ne clot pas le␣
↪canal.

    constructor (address payable _recipient, uint256 duration)
        public
        payable
    {
        sender = msg.sender;
        recipient = _recipient;
        expiration = now + duration;
    }

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

        // Vérifier que la signature est bien de l'emetteur
        return recoverSigner(message, signature) == sender;
    }

    /// Le destinataire peut clore le canal à tout moment en présentant le dernier␣
↪montant
    /// signé par l'expéditeur des fonds. Le destinataire se verra verser ce montant,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /// et le reste sera rendu à l'emetteur des fonds.
    function close(uint256 amount, bytes memory signature) public {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

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

    /// L'emetteur peut modifier la date d'expiration à tout moment
    function extend(uint256 newExpiration) public {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// Si l'expiration est atteinte avant cloture par le destinataire,
    /// l'Ether est renvoyé à l'emetteur
    function claimTimeout() public {
        require(now >= expiration);
        selfdestruct(sender);
    }

    /// Toutes les fonctions ci-dessous sont tirées
    /// du chapitre 'créer et vérifier les signatures'.

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

        assembly {
            // premiers 32 octets, après le préfixe
            r := mload(add(sig, 32))
            // 32 octets suivants
            s := mload(add(sig, 64))
            // Octet final (premier du prochain lot de 32)
            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);
    }

    /// construit un hash préfixé pour mimer le comportement de eth_sign.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Note : La fonction `splitSignature` est très simple et n'utilise pas tous les contrôles de sécurité. Une implémentation réelle devrait utiliser une bibliothèque plus rigoureusement testée de ce code, tel que le fait openzepellin .

Vérification des paiements

Contrairement à notre chapitre précédent, les messages dans un canal de paiement ne sont pas appliqués tout de suite. Le destinataire conserve la trace du dernier message et le fait parvenir au réseau quand il est temps de fermer le canal de paiement. Cela signifie qu'il est essentiel que le destinataire effectue sa propre vérification de chaque message. Sinon, il n'y a aucune garantie que le destinataire sera en mesure d'être payé à la fin.

Le destinataire doit vérifier chaque message à l'aide du processus suivant :

1. Vérifiez que l'adresse du contact dans le message correspond au canal de paiement.
2. Vérifiez que le nouveau total est le montant prévu.
3. Vérifier que le nouveau total ne dépasse pas la quantité d'éther déposée.
4. Vérifiez que la signature est valide et provient de l'expéditeur du canal de paiement.

Nous utiliserons la librairie *ethereumjs-util <https ://github.com/ethereumjs/ethereumjs-util>'_https ://github.com/ethereumjs/ethereumjs-util pour écrire ces vérifications. L'étape finale peut se faire de plusieurs façons, ici en **JavaScript**, Le code suivant emprunte la fonction 'constructMessage* du **code JavaScript** de signature ci-dessus :

```
// Cette ligne mine le fonctionnement de la méthode JSON-RPC de eth_sign.
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();
}
```

4.4 Solidity en Profondeur

Cette section devrait vous fournir toute l'information dont vous avez besoin pour Solidity. Si quelque chose manque ici, merci de nous contacter, en anglais, sur gitter ou à nous faire parvenir un problème sur github, ou en français sur le git de cette traduction

4.4.1 Structure d'un fichier source Solidity

Les fichiers sources peuvent contenir un nombre arbitraire de *définitions de contrats*, directives d'*import* et *directives pragma*.

Pragmas

Le mot-clé `pragma` peut être utilisé pour activer certaines fonctions ou vérifications du compilateur. Une directive pragma est toujours locale à un fichier source, vous devez donc ajouter pragma à tous vos fichiers si vous voulez l'activer dans tout votre projet. Si vous *importez* un autre fichier, le pragma de ce fichier ne s'appliquera pas automatiquement au fichier à importer.

Version Pragma

Les fichiers sources peuvent (et devraient) être annotés avec un `version pragma` pour refuser d'être compilés avec de futures versions de compilateurs qui pourraient introduire des changements incompatibles. Nous essayons de limiter ces changements au strict minimum, et en particulier introduire des changements d'une manière telle que les changements de sémantique nécessiteront également des changements de syntaxe, mais ce n'est bien sûr pas toujours possible. Pour cette raison, c'est toujours une bonne idée de lire le fichier des modifications (« changelog ») au moins pour les versions qui contiennent des changements de rupture, ces versions auront toujours des versions de la forme `0.x.0` ou `x.0.0`.

Le `version pragma` est utilisée comme suit :

```
pragma solidity ^0.4.0;
```

Un tel fichier source ne compilera pas avec un compilateur antérieur à la version 0.4.0 et ne fonctionnera pas non plus sur un compilateur à partir de la version 0.5.0 (cette deuxième condition est ajoutée en utilisant `^`). L'idée derrière cela est la supposition qu'il n'y aura pas de changements de rupture jusqu'à la version `0.5.0`, donc nous pouvons toujours être sûrs que notre code compilera la façon dont nous l'avons prévu. Nous ne précisons pas la version exacte de correctif du compilateur, de sorte que les versions corrigées sont toujours possibles.

Il est possible de spécifier des règles beaucoup plus complexes pour la version du compilateur, la syntaxe suit celle utilisée par npm.

Note : L'utilisation de `version pragma` ne changera pas la version du compilateur. Il n'activera ou désactivera pas non plus les fonctions du compilateur. Il demandera simplement au compilateur de vérifier si sa version correspond à celle requise par le pragma. S'il ne correspond pas, le compilateur affichera une erreur.

Pragma Expérimental

Le deuxième pragma est le `experimental pragma`. Il peut être utilisé pour activer des fonctions du compilateur ou de la langue qui ne sont pas encore activées par défaut. Les pragmas expérimentaux suivants sont actuellement pris en charge :

ABIEncoderV2

Le nouvel encodeur ABI est capable d'encoder et de décoder arbitrairement des tableaux et des structures imbriqués. Il produit un code moins optimal (l'optimiseur pour cette partie du code est encore en développement) et n'a pas reçu autant de tests que l'ancien codeur. Vous pouvez l'activer en utilisant `pragma experimental ABIEncoderV2;` .

SMTChecker

Ce composant doit être activé lors de la compilation du compilateur et n'est par conséquent pas forcément présent dans tous les binaires Solidity. Les *instructions de compilation* expliquent comment activer cette option. Elle est activée pour les versions PPA d'Ubuntu dans la plupart des versions, mais pas pour solc-js, les images Docker, les binaires Windows ni les binaires Linux pré-compilés.

Si vous utilisez `pragma experimental SMTChecker;`, vous aurez des avertissements de sécurité supplémentaires qui sont obtenus en interrogeant un solveur SMT. Le composant ne prend pas encore en charge toutes les fonctionnalités du langage Solidity et émet probablement de nombreux avertissements. Dans le cas où il signale des caractéristiques non prises en charge, l'analyse peut ne pas être cohérente.

Importation d'autres fichiers sources

Syntaxe et sémantique

Solidity supporte les instructions d'importation qui sont très similaires à celles disponibles en JavaScript (à partir de ES6), bien que Solidity ne connaisse pas le concept de « default export ».

Au niveau global, vous pouvez utiliser les instructions d'importation sous la forme suivante :

```
import "filename";
```

Cette instruction importe tous les symboles globaux de « nom de fichier » (et les symboles qui y sont importés) dans le champ d'application global actuel (différent de celui de ES6 mais rétrocompatible pour Solidity). Cette syntaxe simple n'est pas recommandée car elle pollue l'espace de nommage d'une manière imprévisible : Si vous ajoutez de nouveaux éléments de niveau supérieur dans « nom de fichier », ils apparaîtront automatiquement dans tous les fichiers qui importent ainsi à partir de « nom de fichier ». Il est préférable d'importer explicitement des symboles spécifiques.

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

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

En cas de collision de noms, vous pouvez également renommer les symboles lors de l'importation. Ce code crée de nouveaux symboles globaux `alias` et `symbole2` qui font référence à `symbole1` et `symbole2` de `"nom de fichier"`, respectivement.

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

Une autre syntaxe ne fait pas partie de ES6, mais probablement pratique :

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

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

Chemins

Ci-dessus, `nom-de-fichier` est toujours traité comme un chemin avec `/` comme séparateur de répertoire, `.` comme le répertoire courant et `..` comme le répertoire parent. Lorsque `.``ou ``..` est suivi d'un caractère autre que `/`, il n'est pas considéré comme le répertoire courant ou parent. Tous les noms de chemins sont traités comme des chemins absolus à moins qu'ils ne commencent par le répertoire courant `.` ou le répertoire parent `..`.

Pour importer un fichier `x` du même répertoire que le fichier courant, utilisez `import "./x" as x;`. Si vous utilisez `import "x" as x;` à la place, un fichier différent pourrait être référencé (d'un plus global « include directory »).

Il repose sur le compilateur (voir ci-dessous) de résoudre les chemins. En général, la hiérarchie des répertoires n'a pas besoin de pointer strictement sur votre système de fichiers local, elle peut aussi pointer vers les ressources en ipfs, http ou git par exemple.

Note : Utilisez toujours des importations relatives comme `import "./filename.sol";` et évitez d'utiliser `..` dans les spécificateurs de chemins. Dans ce dernier cas, il est probablement préférable d'utiliser des chemins globaux et de configurer les remappages comme expliqué ci-dessous.

Utilisation dans les compilateurs

Lorsque vous invoquez le compilateur, vous pouvez spécifier comment découvrir le premier élément d'un chemin, ainsi que les remappages de préfixes de chemins. Par exemple, vous pouvez configurer un remappage de sorte que tout ce qui est importé du répertoire virtuel `github.com/ethereum/dapp-bin/library` soit réellement lu depuis votre répertoire local `/usr/local/dapp-bin/library`. Si plusieurs remappages s'appliquent, celui avec la clé la plus longue est essayé en premier. Un préfixe vide n'est pas autorisé. Les remappages peuvent dépendre d'un contexte, ce qui vous permet de configurer des paquets à importer, par exemple différentes versions d'une bibliothèque du même nom.

solc :

Pour solc (le compilateur de ligne de commande), vous fournissez ces chemins d'accès sous la forme d'arguments `context:prefix=target`, où les parties `context:``et ``=target` sont optionnelles (`prefix` est la valeur par défaut de `target` dans ce cas). Toutes les valeurs de remappage qui sont des fichiers réguliers sont compilées (y compris leurs dépendances).

Ce mécanisme est rétrocompatible (tant qu'aucun nom de fichier ne contient = ou `:`) et ne constitue donc pas un changement de rupture. Tous les fichiers dans ou sous le répertoire `context` qui importent un fichier commençant par `prefix` sont redirigés en remplaçant `prefix` par `target`.

Par exemple, si vous clonez `github.com/ethereum/dapp-bin/` localement vers `/usr/local/dapp-bin`, vous pouvez utiliser ce qui suit dans votre fichier source :

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

Puis lancer le compilateur :

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

Comme exemple plus complexe, supposons que vous utilisiez un module qui utilise une ancienne version de dapp-bin que vous avez extraite vers `/usr/local/dapp-bin_old`, alors vous pouvez exécuter :

```
solc module1:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin_old/ \
     source.sol
```

Cela signifie que toutes les importations du `module2` pointent vers l'ancienne version mais les importations du `module1` pointent vers la nouvelle version.

Note : `solc` vous permet seulement d'inclure des fichiers de certains répertoires. Ils doivent être dans le répertoire (ou sous-répertoire) d'un des fichiers sources explicitement spécifiés ou dans le répertoire (ou sous-répertoire) d'une cible de remappage. Si vous voulez autoriser les includes absolus directs, ajoutez le remapping `/=//`.

S'il y a plusieurs remappages qui mènent à un fichier valide, le remappage avec le préfixe commun le plus long est choisi.

Remix :

Remix fournit un remappage automatique pour GitHub et récupère automatiquement le fichier en ligne. Vous pouvez importer le mappage itérable comme ci-dessus, par exemple :

: : import « github.com/ethereum/dapp-bin/library/iterable_mapping.sol » as it_mapping ;

Remix may add other source code providers in the future.

Commentaires

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

```
// Ceci est un commentaire sur une ligne.

/*
Ceci est un commentaire
multi-lignes.
*/
```

Note : Un commentaire d'une seule ligne est terminé par tout terminateur de ligne unicode (LF, VF, FF, CR, NEL, LS ou PS) en codage utf8. Le terminateur fait toujours partie du code source après le commentaire, donc si ce n'est pas un symbole ascii (que sont NEL, LS et PS), il conduira à une erreur d'analyse.

De plus, il existe un autre type de commentaire appelé commentaire natspec, pour lequel la documentation n'est pas encore écrite. Ils sont écrits avec une triple barre oblique (`///`) ou un double bloc d'astérisque (`/**... */`) et ils doivent être utilisés directement au-dessus des déclarations ou instructions de fonction. Vous pouvez utiliser les balises de style Doxygen à l'intérieur de ces commentaires pour documenter les fonctions, annoter les conditions de vérification, et fournir un **texte de confirmation** qui est montré aux utilisateurs lorsqu'ils tentent d'appeler une fonction.

Dans l'exemple suivant, nous documentons le titre du contrat, l'explication des deux paramètres d'entrée et les deux valeurs retournées.

```
pragma solidity >=0.4.0 <0.6.0;

/** @title Shape calculator. */
contract ShapeCalculator {
    /** @dev Calculates a rectangle's surface and perimeter.
      * @param w Width of the rectangle.
      * @param h Height of the rectangle.
      * @return s The calculated surface.
      * @return p The calculated perimeter.
      */
    function rectangle(uint w, uint h) public pure returns (uint s, uint p) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        s = w * h;
        p = 2 * (w + h);
    }
}
```

4.4.2 Structure d'un contrat

Les contrats Solidity sont similaires à des classes dans des langages orientés objet. Chaque contrat peut contenir des déclarations de *Variables d'état*, structure-fonctions, structure-fonction-modificateurs, structure-événements, *Types Structure* et *Types Enum*. De plus, les contrats peuvent hériter d'autres contrats.

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

La section sur les contrats contient plus de détails que cette section, qui permet d'avoir une vue d'ensemble rapide.

Variables d'état

Les variables d'état sont des variables dont les valeurs sont stockées en permanence dans le storage du contrat.

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

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

Fonctions

Les fonctions sont les unités exécutables du code d'un contrat.

```
pragma solidity >=0.4.0 <0.6.0;

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

Les *function-calls" peuvent se faire en interne ou en externe et ont différents niveaux de :ref :'visibilité* pour d'autres contrats.

Modificateurs de fonction

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

Modificateurs de Fonctions

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

```
pragma solidity >=0.4.22 <0.6.0;

contract Purchase {
    address public seller;

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

    function abort() public view onlySeller { // Utilisation du modificateur
        // ...
    }
}
```

Évènements

Les évènements (`event`) sont une interface d'accès aux fonctionnalités de journalisation (logs) de l'EVM. >>>>>>> develop

```
pragma solidity >=0.4.21 <0.6.0;

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

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

Voir *Events* dans la section contrats pour plus d'informations sur la façon dont les événements sont déclarés et peuvent être utilisés à partir d'une dapp.

Types Structure

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

```
pragma solidity >=0.4.0 <0.6.0;

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

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Types Enum

Les Enumérateurs (`enum`) peuvent être utilisés pour créer des types personnalisés avec un ensemble fini de "valeurs constantes" (voir *Énumérations* dans la section Types).

```
pragma solidity >=0.4.0 <0.6.0;

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

4.4.3 Types

Solidity est un langage à typage statique, ce qui signifie que le type de chaque variable (état et locale) doit être spécifié. Solidity propose 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 liste synthétique des différents opérateurs, voir *Order of Precedence of Operators*.

Types Valeur

Les types suivants sont également appelés types valeur parce que les variables de ces types sont toujours transmises par valeur, c'est-à-dire qu'elles sont toujours copiées lorsqu'elles sont utilisées comme arguments de fonction ou dans les affectations.

Booléens

`bool` : Les valeurs possibles sont les constantes `true` et `false`.

Opérateurs :
- `!` (négation logique)
- `&&` (conjonction logique, « and »)
- `||` (disjonction logique, « or »)
- `==` (égalité)
- `!=` (inégalité)

Les opérateurs `||` et `&&` appliquent les règles communes de court-circuitage. Celà signifie que `f(x) || g(y)`, if `f(x)` evalue comme `true`, `g(y)` ne sera pas exécutée même si ses effets étaient attendus.

Entiers

`int` / `uint` : Entiers signés et non-signés de différentes tailles. Les mots-clé `uint8` à `uint256` par pas de `8` (entier non signé de 8 à 256 bits) et `int8` à `int256`. `uint` et `int` sont des alias de `uint256` et `int256`, respectivement.

Opérateurs :
- Comparaisons : `<=`, `<`, `==`, `!=`, `>=`, `>` (retournent un `bool`)
- Opérateurs binaires : `&`, `|`, `^` (ou exclusif binaire), `~` (négation binaire)
- Opérateurs de décalage : `<<` (décalage vers la gauche), `>>` (décalage vers la droite)

— Opérateurs arithmétiques : +, −, l" opérateur unaire −, *, /, % (modulo), ** (exponentiation)

Comparaisons

La valeur d'une comparaison est celle obtenue en comparant la valeur entière.

Opérations binaires

Les opérations binaires sont effectuées sur la représentation du nombre par *complément à deux<https ://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux>*. Cela signifie que, par exemple, `~int256(0) == int256(-1)`.

Décalages

Le résultat d'une opération de décalage a le type de l'opérande de gauche. L'expression `x << y` est équivalente à `x * 2**y` et, pour les entiers positifs, `x >> y` est équivalente à `x / 2**y`. Pour un `x` négatif, `x >> y` équivaut à diviser par une puissance de `2` en arrondissant vers le bas (vers l'infini négatif). Décaler d'un nombre négatif de bits déclenche une exception.

Avertissement : Avant la version `0.5.0.0`, un décalage vers la droite `x >> y` pour un `x` négatif était équivalent à `x / 2**y`, c'est-à-dire que les décalages vers la droite étaient arrondis vers zéro plutôt que vers l'infini négatif.

Addition, Soustraction et Multiplication

L'addition, la soustraction et la multiplication ont la sémantique habituelle. Ils utilisent également la représentation du complément de deux, ce qui signifie, par exemple, que `uint256(0) - uint256(1) == 2**256 - 1`. Vous devez tenir compte de ces débordements (« overflows ») pour la conception de contrats sûrs.

L'expression `x` équivaut à `(T(0) - x)` où `T` est le type de `x`. Cela signifie que `-x` ne sera pas négatif si le type de `x` est un type entier non signé. De plus, `x` peut être positif si `x` est négatif. Il y a une autre mise en garde qui découle également de la représentation en compléments de deux :

```
int x = -2**255;
assert(-x == x);
```

Cela signifie que même si un nombre est négatif, vous ne pouvez pas supposer que sa négation sera positive.

Division

Puisque le type du résultat d'une opération est toujours le type d'un des opérandes, la division sur les entiers donne toujours un entier. Dans Solidity, la division s'arrondit vers zéro. Cela signifie que `int256(-5) / int256(2) == int256(-2)`.

Notez qu'en revanche, la division sur les littéraux donne des valeurs fractionnaires de précision arbitraire.

Note : La division par zéra cause un échec d"`assert`.

Modulo

L'opération modulo `a % n` donne le reste `r` après la division de l'opérande `a` par l'opérande `n`, où `q = int(a / n)` et `r = a - (n * q)`. Cela signifie que modulo donne le même signe que son opérande gauche (ou zéro) et `a % n == -(abs(a) % n)` est valable pour un `a` négatif :

- `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 : La division par zéra cause un échec d"`assert`.

Exponentiation

l'exponentiation n'est disponible que p[our les types signés. Veillez à ce que les types que vous utilisez soient suffisamment grands pour conserver le résultat et vous préparer à un éventuel effet d'enroulage (wrapping/int overflow).

Note : `0**0` est défini par l'EVM comme étant `1`.

Nombre à virgule fixe

Avertissement : Les numéros à point fixe ne sont pas encore entièrement pris en charge par Solidity. Ils peuvent être déclarés, mais ne peuvent pas être affectés à ou de.

`fixed` / `ufixed` : Nombre *a virgule fixe signés et non-signés de taille variable. Les mots-clés ''ufixedMxN*' et `fixedMxN`, où `M` représente le nombre de bits pris par le type et `N` représente combien de décimales sont disponibles. `M` doit être divisible par 8 et peut aller de 8 à 256 bits. `N` doit être compris entre 0 et 80, inclusivement. `ufixed` et `fixed` sont des alias pour `ufixed128x18` et `fixed128x18`, respectivement.

Opérateurs :

- Comparaisons : <=, <, ==, !=, >=, > (évalue à `bool`)
- Operateurs arithmétiques : +, -, l'opérateur unaire -, *, /, % (modulo)

Note : La principale différence entre les nombres à virgule flottante (`float``et ``double` dans de nombreux langages, plus précisément les nombres IEEE 754) et les nombres à virgule fixe est que le nombre de bits utilisés pour l'entier et la partie fractionnaire (la partie après le point décimal) est flexible dans le premier, alors qu'il est strictement défini dans le second. Généralement, en virgule flottante, presque tout l'espace est utilisé pour représenter le nombre, alors que seul un petit nombre de bits définit où se trouve le point décimal.

Adresses

Le type d'adresse se décline en deux versions, qui sont en grande partie identiques :

- `address` : Contient une valeur de 20 octets (taille d'une adresse Ethereum).
- `address payable` : Même chose que « adresse », mais avec les membres additionnels `transfert` et `envoi`.

L'idée derrière cette distinction est que l'`address payable` est une adresse à laquelle vous pouvez envoyer de l'éther, alors qu'une simple `address` ne peut être envoyée de l'éther.

Conversions de type :

Les conversions implicites de `address payable` à `address` sont autorisées, tandis que les conversions de `address` à `address payable` ne sont pas possibles. .. note :

```
La seule façon d'effectuer une telle conversion est d'utiliser une conversion␣
↪intermédiaire en ``uint160``.
```

Les adresses littérales peuvent être implicitement converties en `address payable`.

Les conversions explicites vers et à partir de `address` sont autorisées pour les entiers, les entiers littéraux, les `bytes20` et les types de contrats avec les réserves suivantes : Les conversions sous la forme `address payable(x)` ne sont pas permises. Au lieu de cela, le résultat d'une conversion sous forme `adresse(x)` donne une `address payable` si `x` est un contrat disposant d'une fonction par défaut (`fallback`) `payable`, ou si `x` est de type entier, bytes fixes, ou littéral. Sinon, l'adresse obtenue sera de type `address`. Dans les fonctions de signature externes, `address` est utilisé à la fois pour le type `address``et ``address payable`.

Note :

Il se peut fort bien que vous n'ayez pas à vous soucier de la distinction entre `address` et `address payable` et que vous u
`msg.sender`, qui est une `address payable`.

Opérateurs :
— <=, <, ==, !=, >= and >

Avertissement :

Si vous convertissez un type qui utilise une taille d'octet plus grande en `address`, par exemple `bytes32`, alors l'adresse e
Pour réduire l'ambiguïté de conversion à partir de la version 0.4.24 du compilateur vous force à rendre la troncature explicite dans la conversion. Prenons par exemple l'adresse `0x1111222222323333434444545555666666777777788889999999AAAABBBBBBCCDDDDEEFEFFFFFFCC`.
Vous pouvez utiliser `address(uint160(octets20(b)))`, ce qui donne `0x1111212222323333434444545555666677778888889999aAaaa`, ou vous pouvez utiliser `address(uint160(uint256(b)))`, ce qui donne `0x777777888888999999AaAAbBbbCcccddDdeeeEfFFfCcCcCc`.

Note :

La distinction entre `address``et ``address payable` a été introduite avec la version 0.5.0. À partir de cette version également, les contrats ne dérivent pas du type d'adresse, mais peuvent toujours être convertis explicitement en adresse » ou à » adresse payable « , s'ils ont une fonction par défaut payable.

Membres de Address

Pour une liste des membres de address, voir *Membres du type address*.
— `balance` et `transfer`.
Il est possible d'interroger le solde d'une adresse en utilisant la propriété `balance` et d'envoyer des Ether (en unités de wei) à une adresse payable à l'aide de la fonction `transfert` :

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

La fonction `transfer` échoue si le solde du contrat en cours n'est pas suffisant ou si le transfert d'Ether est rejeté par le compte destinataire. La fonction `transfert` s'inverse en cas d'échec.

Note : Si `x` est une adresse de contrat, son code (plus précisément : sa *Fallback Function*, si présente) sera exécutée avec l'appel `transfer` (c'est une caractéristique de l'EVM et ne peut être empêché). Si cette exécution échoue ou s'il n'y a plus de gas, le transfert d'Ether sera annulé et le contrat en cours s'arrêtera avec une exception.

— `send`

`send` est la contrepartie de bas niveau du `transfer`. Si l'exécution échoue, le contrat en cours ne s'arrêtera pas avec une exception, mais `send` retournera `false`.

Avertissement : Il y a certains dangers à utiliser la fonction `send` : Le transfert échoue si la profondeur de la stack atteint 1024 (cela peut toujours être forcé par l'appelant) et il échoue également si le destinataire manque de gas. Donc, afin d'effectuer des transferts d'Ether en toute sécurité, vérifiez toujours la valeur de retour de `send`, utilisez `transfer` ou mieux encore : utilisez un modèle où le destinataire retire l'argent.

— `call`, `delegatecall` et `staticcall`

Afin de s'interfacer avec des contrats qui ne respectent pas l'ABI, ou d'obtenir un contrôle plus direct sur l'encodage, les fonctions `call`, `delegatecall` et `staticcall` sont disponibles. Elles prennent tous pour argument un seul `bytes memory` comme entrée et retournent la condition de succès (en tant que `bool`) et les données (`bytes memory`). Les fonctions `abi.encoder`, `abi.encoderPacked`, `abi.encoderWithSelector` et `abi.encoderWithSignature` peuvent être utilisées pour coder des données structurées.

Exemple :

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

Avertissement :

Toutes ces fonctions sont des fonctions de bas niveau et doivent être utilisées avec précaution. Plus précisément, tout contrat inconnu peut être malveillant et si vous l'appelez, vous transférez le contrôle à ce contrat qui, à son tour, peut revenir dans votre contrat, donc soyez prêt à modifier les variables de votre état. quand l'appel revient. La façon habituelle d'interagir avec d'autres contrats est d'appeler une fonction sur un objet `contract` (`x.f()`)..

: : note : : Les versions précédentes de Solidity permettaient à ces fonctions de recevoir des arguments arbitraires et de traiter différemment un premier argument de type `bytes4`. Ces cas rares ont été supprimés dans la version 0.5.0.

Il est possible de régler le gas fourni avec le modificateur `.gas()` :

```
namReg.call.gas(1000000)(abi.encodeWithSignature("register(string)", "MyName"));
```

De même, la valeur en Ether fournie peut également être contrôlée : : :

```
nameReg.call.value(1 ether)(abi.encodeWithSignature("register(string)", "MyName"));
```

Enfin, ces modificateurs peuvent être combinés. Leur ordre n'a pas d'importance :

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

De la même manière, la fonction `delegatecall` peut être utilisée : la différence est que seul le code de l'adresse donnée est utilisé, tous les autres aspects (stockage, balance,...) sont repris du contrat actuel. Le but de `delegatecall` est d'utiliser du code de bibliothèque qui est stocké dans un autre contrat. L'utilisateur doit s'assurer que la disposition du stockage dans les deux contrats est adaptée à l'utilisation de `delegatecall`.

Note : Avant Homestead, il n'existait qu'une variante limitée appelée `callcode` qui ne donnait pas accès aux valeurs originales `msg.sender` et `msg.value`. Cette fonction a été supprimée dans la version 0.5.0.

Depuis Byzantium, `staticcall` peut aussi être utilisé. C'est fondamentalement la même chose que `call`, mais reviendra en arrière si la fonction appelée modifie l'état d'une manière ou d'une autre.

Les trois fonctions `call`, `delegatecall``et ``staticcall` sont des fonctions de très bas niveau et ne devraient être utilisées qu'en *dernier recours* car elles brisent la sécurité de type de Solidity.

L'option `.gas()` est disponible sur les trois méthodes, tandis que l'option `.value()` n'est pas supportée pour `delegatecall`.

Note : Tous les contrats pouvant être convertis en type `address`, il est possible d'interroger le solde du contrat en cours en utilisant `address(this).balance`.

Types Contrat

Chaque *contrat* définit son propre type. Vous pouvez implicitement convertir des contrats en contrats dont ils héritent. Les contrats peuvent être explicitement convertis de et vers tous les autres types de contrats et le type `address`.

La conversion explicite vers et depuis le type `address payable` n'est possible que si le type de contrat dispose d'une fonction de repli payante. La conversion est toujours effectuée en utilisant `address(x)` et non `address payable(x)`. Vous trouverez plus d'informations dans la section sur le *type address*.

Note : Avant la version 0.5.0, les contrats dérivaient directement du type address et il n'y avait aucune distinction entre `address` et `address payable`.

Si vous déclarez une variable locale de type contrat (*MonContrat c*), vous pouvez appeler des fonctions sur ce contrat. Prenez bien soin de l'assigner à un contrat d'un type correspondant.

Vous pouvez également instancier les contrats (ce qui signifie qu'ils sont nouvellement créés). Vous trouverez plus de détails dans la section "contrats de création".

La représentation des données d'un contrat est identique à celle du type `address` et ce type est également utilisé dans l'*ABI*.

Les contrats ne supportent aucun opérateur.

Les membres du type contrat sont les fonctions externes du contrat, y compris les variables d'état publiques.

Tableaux d'octets de taille fixe

Les types valeur `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` contiennent une séquence de 1 à 32 octets. `byte` est un alias de `bytes1`.

Opérateurs :
- Comparaisons : <=, <, ==, !=, >=, > (retournent un `bool`)
- Opérateurs binaires : `&`, `|`, ^ (ou exclusif binaire), ~ (négation binaire)
- Opérateurs de décalage : << (décalage vers la gauche), >> (décalage vers la droite)
- Accès par indexage : Si `x` estd e type `bytesI`, alors `x[k]` pour `0 <= k < I` retourne le `k` ème byte (lecture seule).

L'opérateur de décalage travaille avec n'importe quel type d'entier comme opérande droite (mais retourne le type de l'opérande gauche), qui indique le nombre de bits à décaler. Le décalage d'un montant négatif entraîne une exception d'exécution.

Membres :

*`.length` donne la longueur fixe du tableau d'octets (lecture seule).

Note : Le type `byte[]` est un tableau d'octets, mais en raison des règles de bourrage, il gaspille 31 octets d'espace pour chaque élément (sauf en storage). Il est préférable d'utiliser le type « bytes » à la place.

Tableaux dynamiques d'octets

`bytes` : Tableau d'octets de taille dynamique, voir :ref :`arrays`. Ce n'est pas un type valeur !

`string` : Chaîne codée UTF-8 de taille dynamique, voir *Tableaux*. Ce n'est pas un type valeur !

Adresses Littérales

Les caractères hexadécimaux qui réussissent un test de somme de contrôle d'adresse (« address checksum »), par exemple `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` sont de type `address payable`. Les nombres hexadécimaux qui ont entre 39 et 41 chiffres et qui ne passent pas le test de somme de contrôle produisent un avertissement et sont traités comme des nombres rationnels littéraux réguliers.

Note : Le format de some de contr^ole multi-casse est décrit dans EIP-55.

Rationels et entiers littéraux

Les nombres entiers littéraux sont formés à partir d'une séquence de nombres compris entre 0 et 9 interprétés en décimal. Par exemple, `69` signifie soixante-neuf. Les littéraux octaux n'existent pas dans Solidity et les zéros précédant un nombre sont invalides.

Les fractions décimales sont formées par un `.` avec au moins un chiffre sur un côté. Exemples : `1.1`, `.1 `` et ``1.3`.

La notation scientifique est également supportée, où la base peut avoir des fractions, alors que l'exposant ne le peut pas. Exemples : `2e10`, `-2e10`, `2e-10`, `2e-10`, `2.5e1`.

Les soulignements (underscore) peuvent être utilisés pour séparer les chiffres d'un nombre littéral numérique afin d'en faciliter la lecture. Par exemple, la décimale `123_000`, l'hexadécimale `0x2eff_abde`, la notation décimale

scientifique `1_2e345_678` sont toutes valables. Les tirets de soulignement ne sont autorisés qu'entre deux chiffres et un seul tiret de soulignement consécutif est autorisé. Il n'y a pas de signification sémantique supplémentaire ajoutée à un nombre contenant des tirets de soulignement, les tirets de soulignement sont ignorés.

Les expressions littérales numériques conservent une précision arbitraire jusqu'à ce qu'elles soient converties en un type non littéral (c'est-à-dire en les utilisant avec une expression non littérale ou par une conversion explicite). Cela signifie que les calculs ne débordent pas (overflow) et que les divisions ne tronquent pas les expressions littérales des nombres.

Par exemple, `(2**800 + 1) - 2**800` produit la constante `1` (de type `uint8`) bien que les résultats intermédiaires ne rentrent même pas dans la taille d'un mot machine. De plus, `.5 * 8` donne l'entier `4` (bien que des nombres non entiers aient été utilisés entre les deux).

N'importe quel opérateur qui peut être appliqué aux nombres entiers peut également être appliqué aux expressions littérales des nombres tant que les opérandes sont des nombres entiers. Si l'un des deux est fractionnaire, les opérations sur bits sont interdites et l'exponentiation est interdite si l'exposant est fractionnaire (parce que cela pourrait résulter en un nombre non rationnel).

Note :

Solidity a un type de nombre littéral pour chaque nombre rationnel. Les nombres entiers littéraux et les nombres rationnels appartiennent à des types de nombres littéraux. De plus, toutes les expressions numériques littérales (c'est-à-dire les expressions qui ne contiennent que des nombres et des opérateurs) appartiennent à des types littéraux de nombres. Ainsi, les expressions littérales `1 + 2` et `2 + 1` appartiennent toutes deux au même type littéral de nombre pour le nombre rationnel numéro trois.

Avertissement : La dvision d'entiers littéraux tronquait dans les versions de Solidity avant la version 0.4.0, mais elle donne maintenant en un nombre rationnel, c'est-à-dire que `5 / 2` n'est pas égal à `2`, mais à `2.5`.

Note : Les expressions littérales numériques sont converties en caractères non littéraux dès qu'elles sont utilisées avec des expressions non littérales. Indépendamment des types, la valeur de l'expression assignée à `b` ci-dessous est évaluée en entier. Comme `a` est de type `uint128`, l'expression `2,5 + a` doit cependant avoir un type. Puisqu'il n'y a pas de type commun pour les types `2.5` et `uint128`, le compilateur Solidity n'accepte pas ce code.

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

Chaines de caractères littérales

Les chaînes de caractères littérales sont écrites avec des guillemets simples ou doubles (`"foo"` ou `'bar'`). Elles n'impliquent pas de zéro final comme en C ; `foo` représente trois octets, pas quatre. Comme pour les entiers littéraux, leur type peut varier, mais ils sont implicitement convertibles en `bytes1`, ..., `bytes32`, ou s'ils conviennent, en `bytes` et en `string`.

Les chaînes de caractères littérales supportent les caractères d'échappement suivants :

- `\<newline>` (échappe un réel caractère newline)
- `\\` (barre oblique)
- `\'` (guillemet simple)
- `\"` (guillemet double)
- `\b` (backspace)

- \f (form feed)
- \n (newline)
- \r (carriage return)
- \t (tabulation horizontale)
- \v (tabulation verticale)
- \xNN (hex escape, see below)
- \uNNNN (echapement d'unicode, voir ci-dessous)

\xNN prend une valeur hexadécimale et insère l'octet approprié, tandis que \uNNNNN prend un codepoint Unicode et insère une séquence UTF-8.

La chaîne de caractères de l'exemple suivant a une longueur de dix octets. Elle commence par un octet de newline, suivi d'une guillemet double, d'une guillemet simple, d'un caractère barre oblique inversée et ensuite (sans séparateur) de la séquence de caractères abcdef.

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

Tout terminateur de ligne unicode qui n'est pas une nouvelle ligne (i.e. LF, VF, FF, CR, NEL, LS, PS) est considéré comme terminant la chaîne littérale. Newline ne termine la chaîne littérale que si elle n'est pas précédée d'un \.

Hexadécimaux littéraux

Les caractères hexadécimaux sont précédées du mot-clé hex et sont entourées de guillemets simples ou doubles (hex"001122FF"). Leur contenu doit être une chaîne hexadécimale et leur valeur sera la représentation binaire de ces valeurs.

Les littéraux hexadécimaux se comportent comme *chaînes de caractères littérales* et ont les mêmes restrictions de convertibilité.

Énumérations

Les enum sont une façon de créer un type défini par l'utilisateur en Solidity. Ils sont explicitement convertibles de et vers tous les types d'entiers mais la conversion implicite n'est pas autorisée. La conversion explicite à partir d'un nombre entier vérifie au moment de l'exécution que la valeur se trouve à l'intérieur de la plage de l'enum et provoque une affirmation d'échec autrement. Un enum a besoin d'au moins un membre.

La représentation des données est la même que pour les énumérations en C : Les options sont représentées par des valeurs entières non signées à partir de 0.

```
pragma solidity >=0.4.16 <0.6.0;

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

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

    // Comme le type enum ne fait pas partie de l' ABI, la signature de "getChoice"
    // sera automatoquement changée en "getChoice() returns (uint8)"
    // pour ce qui sort de Solidity. Le type entier utilisé est
    // assez grand pour contenir toutes valeurs, par exemple si vous en avez
    // plus de 256, ``uint16`` sera utilisé etc...
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

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

Types Fonction

Les types fonction sont les types des fonctions. Les variables du type fonction peuvent être passés et retournés pour transférer les fonctions vers et renvoyer les fonctions des appels de fonction. Les types de fonctions se déclinent en deux versions : les fonctions *internes* `internal` et les fonctions *externes* `external` :

Les fonctions internes ne peuvent être appelées qu'à l'intérieur du contrat en cours (plus précisément, à l'intérieur de l'unité de code en cours, qui comprend également les fonctions de bibliothèque internes et les fonctions héritées) car elles ne peuvent pas être exécutées en dehors du contexte du contrat actuel. L'appel d'une fonction interne est réalisé en sautant à son label d'entrée, tout comme lors de l'appel interne d'une fonction du contrat en cours.

Les fonctions externes se composent d'une adresse et d'une signature de fonction et peuvent être transférées et renvoyées à partir des appels de fonction externes.

Les types de fonctions sont notés comme suit : :

fonction (<types de paramètres>) {internallexternal} {purelviewlpayable][returns (<types de retour>)]

En contraste avec types de paramètres, les types de retour ne peuvent pas être vides - si le type de fonction ne retourne rien, toute la partie ‘‘returns (<types de retour>)‘‘doit être omise.

Par défaut, les fonctions sont de type `internal`, donc le mot-clé `internal` peut être omis. Notez que ceci ne s'applique qu'aux types de fonctions. La visibilité doit être spécifiée explicitement car les fonctions définies dans les contrats n'ont pas de valeur par défaut.

Conversions :

Une fonction de type `external` peut être explicitement convertie en `address` résultant en l'adresse du contrat de la fonction.

Un type de fonction `A` est implicitement convertible en un type de fonction `B` si et seulement si leurs types de paramètres sont identiques, leurs types de retour sont identiques, leurs propriétés internal/external sont identiques et la mutabilité d'état de `A` n'est pas plus restrictive que la mutabilité de l'état de `B`. En particulier :

- Les fonctions `pure` peuvent être converties en fonctions `view` et `non-payable`.
- Les fonctions `view` peuvent être converties en fonctions `non-payable`.
- les fonctions `payable` peuvent être converties en fonctions `non-payable`.

Aucune autre conversion entre les types de fonction n'est possible.

La règle concernant les fonctions `payable` et `non-payable` peut prêter à confusion, mais essentiellement, si une fonction est `payable`, cela signifie qu'elle accepte aussi un paiement de zéro Ether, donc elle est également `non-payable`. D'autre part, une fonction `non-payable` rejettera l'Ether qui lui est envoyé, de sorte que les fonctions `non-payable` ne peuvent pas être converties en fonctions `payable`.

Si une variable de type fonction n'est pas initialisée, l'appel de celle-ci entraîne l'échec d'une assertion. Il en va de même si vous appelez une fonction après avoir utilisé `delete` dessus.

Si des fonctions de type `external` sont appelées d'en dehors du contexte de Solidity, ils sont traités comme le type `function`, qui code l'adresse suivie de l'identificateur de fonction ensemble dans un seul type `bytes24`.

Notez que les fonctions publiques du contrat actuel peuvent être utilisées à la fois comme une fonction interne et comme une fonction externe. Pour utiliser `f` comme fonction interne, utilisez simplement `f`, si vous voulez utiliser sa forme externe, utilisez `this.f`.

Membres :

Les fonctions publiques (ou externes) ont aussi un membre spécial appelé `selector`, qui retourne le *sélecteur de fonction* :

```
pragma solidity >=0.4.16 <0.6.0;

contract Selector {
  function f() public pure returns (bytes4) {
    return this.f.selector;
  }
}
```

Exemple d'utilisation des fonctions de type `internal` :

```
pragma solidity >=0.4.16 <0.6.0;

library ArrayUtils {
  // les fonctions internes peuvent être utilisées dams des fonctions
  // de librairies internes car elles partagent le même contexte
  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) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return ArrayUtils.range(l).map(square).reduce(sum);
  }
  function square(uint x) internal pure returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal pure returns (uint) {
    return x + y;
  }
}
```

Exemple d" usage de fonction `external` :

```
pragma solidity >=0.4.22 <0.6.0;

contract Oracle {
  struct Request {
    bytes data;
    function(uint) external callback;
  }
  Request[] 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 {
    // Ici on checke que la réponse vient d'une source de confiance
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant oracle = Oracle(0x1234567); // known contract
  uint exchangeRate;
  function buySomething() public {
    oracle.query("USD", this.oracleResponse);
  }
  function oracleResponse(uint response) public {
    require(
        msg.sender == address(oracle),
        "Only oracle can call this."
    );
    exchangeRate = response;
  }
}
```

Note : Les fonctions lambda ou en in-line sont prévues mais pas encore prises en charge.

Types Référence

Les valeurs du type référence peuvent être modifiées par plusieurs noms différents. Comparez ceci avec les catégories de valeurs où vous obtenez une copie indépendante chaque fois qu'une variable de valeur est utilisée. Pour cette raison, les types référence doivent être traités avec plus d'attention que les types de valeur. Actuellement, les types référence comprennent les structures, les tableaux et les mappages. Si vous utilisez un type référence, vous devez

toujours indiquer explicitement la zone de données où le type est enregistré : (dont la durée de vie est limitée à un appel de fonction), `storage` (l'emplacement où les variables d'état sont stockées) ou `calldata` (emplacement de données spécial qui contient les arguments de fonction, disponible uniquement pour les paramètres d'appel de fonction externe).

Une affectation ou une conversion de type qui modifie l'emplacement des données entraîne toujours une opération de copie automatique, alors que les affectations à l'intérieur du même emplacement de données ne copient que dans certains cas selon le type de stockage.

Emplacement des données

Chaque type référence, c'est-à-dire *arrays* (tableaux) et *structs*, comporte une annotation supplémentaire, la `localisation des données`, indiquant où elles sont stockées. Il y a trois emplacements de données : `Memory`, `Storage` et `Calldata`. Calldata n'est valable que pour les paramètres des fonctions de contrat externes et n'est nécessaire que pour ce type de paramètre. Calldata est une zone non modifiable, non persistante où les arguments de fonction sont stockés, et se comporte principalement comme memory.

Note : Avant la version 0.5.0, l'emplacement des données pouvait être omis, et était par défaut à des emplacements différents selon le type de variable, le type de fonction, etc.

La localisation des données n'est sont pas seulement pertinente pour la persistance des données, mais aussi pour la sémantique des affectations : Les affectations entre le stockage et la mémoire (ou à partir des données de la calldata) créent toujours une copie indépendante. Les affectations de mémoire à mémoire ne créent que des références. Cela signifie que les modifications d'une variable mémoire sont également visibles dans toutes les autres variables mémoire qui se réfèrent aux mêmes données. Les affectations du stockage à une variable de stockage local n'affectent également qu'une référence. En revanche, toutes les autres affectations au stockage sont toujours copiées. Les affectations à des variables d'état ou à des membres de variables locales de type structure de stockage, même si la variable locale elle-même n'est qu'une référence, constituent des exemples dans ce cas.

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    uint[] x; // the data location of x is storage

    // Les données de memoryArray sont stockées en mémoire (memory)
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // marche, copie le tableau en storage
        uint[] storage y = x; // marche, assigne un pointeur, y est en storage
        y[7]; // bon, retourne le 8e élément
        y.length = 2; // bon, modifie x via y
        delete x; // bon, efface l'array, modifie y
        // L'exemple suivant ne fonctionne pas, il implique de créer un
        // tableau anonyme en storage, mais storage est alloué "statiquement"
        // y = memoryArray;
        // Ceci ne marche pas non plus, car ça redéfinirait le
        // pointeur mais ne pointe sur rien
        // delete y;
        g(x); // appelle g, avec un pointeur sur x
        h(x); // appelle h et crée une copie indépendante en memory
    }

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

Tableaux

Les tableaux peuvent avoir une taille fixe à la compilation ou peuvent être dynamiques. Il y a peu de restrictions concernant l'élément contenu, il peut aussi être un autre tableau, un mappage ou une structure. Les restrictions générales s'appliquent, cependant, en ce sens que les mappages ne peuvent être utilisés que dans le storage et que les fonctions visibles au public nécessitent des paramètres qui sont des types reconnus par l'ABI.

Un tableau de taille fixe `k` et de type d'élément `T` est écrit `T[k]`, un tableau de taille dynamique `T[]`. Par exemple, un tableau de 5 tableaux dynamiques de `uint` est `uint[][5]` (notez que la notation est inversée par rapport à certains autres langages). Pour accéder au deuxième uint du troisième tableau dynamique, vous utilisez `x[2][1]` (les indexs commencent à zéro et l'accès fonctionne dans le sens inverse de la déclaration, c'est-à-dire que `x[2]` supprime un niveau dans le type de déclaration à partir de la droite).

L'accès à un tableau après sa fin provoque un `revert`. Si vous voulez ajouter de nouveaux éléments, vous devez utiliser `.push()` ou augmenter le membre `.length` (voir ci-dessous).

Les variables de type `bytes` et `string` sont des tableaux spéciaux. Un `byte` est semblable à un `byte[]`, mais il est condensé en calldata et en mémoire. `string` est égal à `bytes`, mais ne permet pas l'accès à la longueur ou à l'index. Il faut donc généralement préférer les `bytes` aux `bytes[]` car c'est moins cher à l'usage. En règle générale, utilisez `bytes` pour les données en octets bruts de longueur arbitraire et `string` pour les données de chaîne de caractères de longueur arbitraire (UTF-8). Si vous pouvez limiter la longueur à un certain nombre d'octets, utilisez toujours un des `bytes1` à `bytes32`, car ils sont beaucoup moins chers également.

Note : Si vous voulez accéder à la représentation en octets d'une chaîne de caractères `s`, utilisez `bytes(s).length` / `bytes(s)[7] ='x';`. Gardez à l'esprit que vous accédez aux octets de bas niveau de la représentation UTF-8, et non aux caractères individuels !

Il est possible de marquer les tableaux `public` et de demander à Solidity de créer un *getter*. L'index numérique deviendra un paramètre obligatoire pour le getter.

Allouer des tableaux en mémoire

Vous pouvez utiliser le mot-clé `new` pour créer des tableaux dont la longueur dépend de la durée d'exécution en mémoire. Contrairement aux tableaux de stockage, il n'est **pas** possible de redimensionner les tableaux de mémoire (par exemple en les assignant au membre `.length`). Vous devez soit calculer la taille requise à l'avance, soit créer un nouveau tableau de mémoire et copier chaque élément.

```
pragma solidity >=0.4.16 <0.6.0;

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

Tableaux littéraux / Inline Arrays

Les littéraux de tableau sont des tableaux qui sont écrits comme une expression et ne sont pas assignés à une variable tout de suite.

```
pragma solidity >=0.4.16 <0.6.0;

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

Le type d'un tableau littéral est un tableau mémoire de taille fixe dont le type de base est le type commun des éléments donnés. Le type de `[1, 2, 3]` est `uint8[3] memory`, car le type de chacune de ces constantes est `uint8`. Pour cette raison, il est nécessaire de convertir le premier élément de l'exemple ci-dessus en « uint ». Notez qu'actuellement, les tableaux de taille fixe ne peuvent pas être assignées à des tableaux de taille dynamique, c'est-à-dire que ce qui suit n'est pas possible :

```
pragma solidity >=0.4.0 <0.6.0;

// Ceci ne compile pas.
contract C {
    function f() public {
        // La ligne suivant provoque une erreur car uint[3] memory
        // ne peut pas être convertit en uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

Il est prévu de supprimer cette restriction à l'avenir, mais crée actuellement certaines complications en raison de la façon dont les tableaux sont transmis dans l'ABI.

Membres

length :

Les tableaux ont un membre `length` qui contient leur nombre d'éléments. La longueur des tableaux memory est fixe (mais dynamique, c'est-à-dire qu'elle peut dépendre des paramètres d'exécution) une fois qu'ils sont créés. Pour les tableaux de taille dynamique (disponible uniquement en storage), ce membre peut être assigné pour redimensionner le tableau. L'accès à des éléments en dehors de la longueur courante ne redimensionne pas automatiquement le tableau et provoque plutôt un échec d'assertion. L'augmentation de la longueur ajoute de nouveaux éléments initialisés zéro au tableau. Réduire la longueur permet d'effectuer une suppression (:ref :`suppression`) implicite sur chacun des éléments supprimés.

push : Les tableaux de stockage dynamique et les `bytes` (et non `string`) ont une fonction membre appelée `push` que vous pouvez utiliser pour ajouter un élément à la fin du tableau. L'élément sera mis à zéro à l'initialisation. La fonction renvoie la nouvelle longueur.

pop : Les tableaux de stockage dynamique et les `bytes` (et non `string`) ont une fonction membre appelée `pop` que vous pouvez utiliser pour supprimer un élément à la fin du tableau. Ceci appelle aussi implicitement :ref :`delete` sur l'élément supprimé.

Avertissement : Si vous utilisez `.length--` sur un tableau vide, cela provoque un débordement par le bas et fixe donc la longueur à `2**256-1`.

Note : L'augmentation de la longueur d'un tableau en storage a des coûts en gas constants parce qu'on suppose que le stockage est nul, alors que la diminution de la longueur a au moins un coût linéaire (mais dans la plupart des cas pire que linéaire), parce qu'elle inclut explicitement l'élimination des éléments supprimés comme si on appelait :ref :`delete`.

Note : Il n'est pas encore possible d'utiliser les tableaux de tableaux dans les fonctions externes (mais ils sont supportés dans les fonctions publiques).

Note : Dans les versions EVM antérieures à Byzantium, il n'était pas possible d'accéder au retour de tableaux dynamique à partir des appels de fonctions. Si vous appelez des fonctions qui retournent des tableaux dynamiques, assurez-vous d'utiliser un EVM qui est configuré en mode Byzantium.

```
pragma solidity >=0.4.16 <0.6.0;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Notez que ce qui suit n'est pas une paire de tableaux dynamiques
    // mais un tableau tableau dynamique de paires (c'est-à-dire de
    // tableaux de taille fixe de longueur deux).
    // Pour cette raison, T[] est toujours un tableau dynamique
    // de T, même si T lui-même est un tableau.
    // L'emplacement des données pour toutes les variables d'état
    // est storage.
    bool[2][] m_pairsOfFlags;

    // newPairs est stocké en memory - seule possibilité
    // pour les arguments de fonction publique
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // l'assignation d' un tableau en storage implique la copie
        // de  ``newPairs`` et remplace l'array ``m_pairsOfFlags``.
        m_pairsOfFlags = newPairs;
    }

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

    function f(uint[] memory c) public {
        // stocke un pointeur sur ``s`` dans ``g``
        StructType storage g = s;
        // change aussi ``s.moreInfo``.
        g.moreInfo = 2;
        // assigne une copie car ``g.contents`` n'est
        // pas une variable locale mais un membre
        // d'une variable locale
        g.contents = c;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // accès à un index inexistant, déclenche une exception
```

(suite sur la page suivante)

(suite de la page précédente)

```
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // Si la nouvelle taille est plus petite, les
        // éléments en trop seront supprimés
        m_pairsOfFlags.length = newSize;
    }

    function clear() public {
        // supprime le tableau complet
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // meme effet avec
        m_pairsOfFlags.length = 0;
    }

    bytes m_byteData;

    function byteArrays(bytes memory data) public {
        // le tableau de byte ("bytes") sont différents car stockés sans
        // padding mais peuvent être traités comme des ``uint8[]``
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = 0x08;
        delete m_byteData[2];
    }

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

    function createMemoryArray(uint size) public pure returns (bytes memory) {
        // Un tableau dynamique est créé via `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        // Les tableaux déclarés à la volée sont toujours de taille statique
        // et en cas d' utilisation de littéraux uniquement, au moins
        // un type doit être spécifié.
        arrayOfPairs[0] = [uint(1), 2];

        // Créée un tableau dynamique de bytes:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(uint8(i));
        return b;
    }
}
```

Structs

Solidity permet de définir de nouveaux types sous forme de structs, comme le montre l'exemple suivant :

```
pragma solidity >=0.4.11 <0.6.0;

contract CrowdFunding {
    // Définit un nouveau type avec 2 champs.
    struct Funder {
        address addr;
        uint amount;
    }

    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
        // Crée une nouvelle structure en memory et la copie vers storage.
        // Nous omettons le type de mappage, car il n'est pas valide en mémoire.
        // Si les structures sont copiées (même d'un stockage à un autre), les //␣
↪types de mappage sont toujours omis, car ils ne peuvent pas
        // être énumérés.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // Créé une nouvelle struct temporaire en memory, initialisée
        // aux valeurs voulues, et copie-la en storage.
        // Notez que vous pouvez également utiliser (msg.sender, msg.value)
        // pour l'initialiser.
        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)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

Le contrat ne fournit pas toutes les fonctionnalités d'un contrat de crowdfunding, mais il contient les concepts de base nécessaires pour comprendre les `struct`. Les types structs peuvent être utilisés à l'intérieur des `mapping` et des `array` et peuvent eux-mêmes contenir des mappages et des tableaux.

Il n'est pas possible pour une structure de contenir un membre de son propre type, bien que la structure elle-même puisse être le type de valeur d'un membre de mappage ou peut contenir un tableau de taille dynamique de son type.

Cette restriction est nécessaire, car la taille de la structure doit être finie.

Notez que dans toutes les fonctions, un type structure est affecté à une variable locale avec l'emplacement de données `storage`. Ceci ne copie pas la structure mais stocke seulement une référence pour que les affectations aux membres de la variable locale écrivent réellement dans l'état.

Bien sûr, vous pouvez aussi accéder directement aux membres de la structure sans l'affecter à une variable locale, comme dans `campaigns[campaignID].amount = 0`.

Mappages

Vous déclarez le objets de type `mapping` avec la syntaxe `mapping(_KeyType => _ValueType)`. `_KeyType` peut être n'importe quel type élémentaire. Cela signifie qu'il peut s'agir de n'importe lequel des types de valeurs intégrés plus les octets et les chaînes de caractères. Les types définis par l'utilisateur ou les types complexes tels que les types de contrat, les énuménations, les mappages, les structs et tout type de tableau, à l'exception des `bytes` et des `string` qui ne sont pas autorisés. `_ValueType` peut être n'importe quel type, y compris les mappages.

Vous pouvez considérer les mappings comme des tables de hashage, qui sont virtuellement initialisées de telle sorte que chaque clé possible existe et est mappée à une valeur dont la représentation binaire est constituée de zéros, de type *valeur par défaut*. La similitude s'arrête là, les données « clés » ne sont pas stockées dans un mappage, seul son hachage `keccak256` est utilisé pour rechercher la valeur.

Pour cette raison, les mappages n'ont pas de longueur ou de concept de clé ou de valeur définie.

Les mappages ne peuvent avoir qu'un emplacement de données en `storage` et sont donc autorisés pour les variables d'état, comme types référence en storage dans les fonctions ou comme paramètres pour les fonctions de librairies. Ils ne peuvent pas être utilisés comme paramètres ou paramètres de retour de fonctions de contrat publiques.

Vous pouvez marquer les variables de type mapping comme `public` et Solidity crée un *getter* pour vous. Le `` `_KeyType `` devient un paramètre pour le getter. Si `_ValueType` est un type de valeur ou une structure, le getter retourne `_ValueType`. Si `_ValueType` est un tableau ou un mappage, le getter a un paramètre pour chaque `_KeyType`, de manière récursive. Par exemple :

```
pragma solidity >=0.4.0 <0.6.0;

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));
    }
}
```

Note : Les mappings ne sont pas itérables, mais il est possible d'y ajouter une structure de données. Pour un exemple, voir mapping iterable.

Opérateurs impliquant des LValues

Si `a` est une LValue (c.-à-d. une variable ou quelque chose qui peut être assigné à), les opérateurs suivants sont disponibles en version raccourcie :

```
``a += e`` équivaut à ``a = a + e``. Les opérateurs ``-=``, ``*=``, ``/=``, ``%=``, ``|=``, ``&=`` et ``^=`` sont définis de la même manière. ``a++`` et ``a--`` sont équivalents à ``a += 1`` / ``a -= 1`` mais l'expression elle-même a toujours la valeur précédente ``a``. Par contraste, ``--a`` et ``++a`` changent également ``a`` de ``1`` , mais retournent la valeur après le changement.
```

delete

`delete a` affecte la valeur initiale du type à `a`. C'est-à-dire que pour les entiers, il est équivalent à `a = 0`, mais il peut aussi être utilisé sur les tableaux, où il assigne un tableau dynamique de longueur zéro ou un tableau statique de la même longueur avec tous les éléments réinitialisés. Pour les structs, il assigne une structure avec tous les membres réinitialisés. En d'autres termes, la valeur de `a` après `delete a` est la même que si `a` était déclaré sans attribution, avec la réserve suivante :

`delete` n'a aucun effet sur les mappages (car les clés des mappages peuvent être arbitraires et sont généralement inconnues). Ainsi, si vous supprimez une structure, elle réinitialisera tous les membres qui ne sont pas des `mappings` et se propagera récursivement dans les membres à moins qu'ils ne soient des mappings. Toutefois, il est possible de supprimer des clés individuelles et ce à quoi elles correspondent : Si `a` est un mappage, alors `delete a[x]` supprimera la valeur stockée à `x`.

Il est important de noter que `delete a` se comporte vraiment comme une affectation à `a`, c'est-à-dire qu'il stocke un nouvel objet dans `a`. Cette distinction est visible lorsque `a` est une variable par référence : Il ne réinitialisera que `a` lui-même, et non la valeur à laquelle il se référait précédemment.

```
pragma solidity >=0.4.0 <0.6.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // met x à 0, n' affecte pas data
        delete data; // met data à 0, n'affecte pas x
        uint[] storage y = dataArray;
        delete dataArray; // ceci met dataArray.length à zéro, mais un uint[]
        // est un objet complexe, donc y est affecté est un alias
        // vers l' objet en storage.
        // D' un autre côté: "delete y" est invalid, car l' assignement à
        // une variable locale pointant vers un objet en storage n' est
        // autorisée que depuis un objet en storage.
        assert(y.length == 0);
    }
}
```

Conversions entre les types élémentaires

Conversions implicites

Si un opérateur est appliqué à différents types, le compilateur essaie de convertir implicitement l'un des opérandes au type de l'autre (c'est la même chose pour les assignations). En général, une conversion implicite entre les types valeur est possible si elle a un sens sémantique et qu'aucune information n'est perdue : `uint8` est convertible en `uint16` et `int128` en `int256`, mais `uint8` n'est pas convertible en `uint256` (car `uint256` ne peut contenir, par exemple, `-1`). Tout type d'entier qui peut être converti en `uint160` peut aussi être converti en `address`.

Pour plus de détails, veuillez consulter les sections concernant les types eux-mêmes.

Conversions explicites

Si le compilateur ne permet pas la conversion implicite mais que vous savez ce que vous faites, une conversion de type explicite est parfois possible. Notez que cela peut vous donner un comportement inattendu et vous permet de contourner certaines fonctions de sécurité du compilateur, donc assurez-vous de tester que le résultat est ce que vous voulez ! Prenons l'exemple suivant où l'on convertit un `int8` négatif en un `uint` :

```
int8 y = -3;
uint x = uint(y);
```

A la fin de cet extrait de code, `x` aura la valeur `0xfffffff...fd` (64 caractères hexadécimaux), qui est -3 dans la représentation en 256 bits du complément à deux.

Si un entier est explicitement converti en un type plus petit, les bits d'ordre supérieur sont coupés :

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b sera désormais 0x5678
```

Si un entier est explicitement converti en un type plus grand, il est rembourré par la gauche (c'est-à-dire à l'extrémité supérieure de l'ordre). Le résultat de la conversion sera comparé à l'entier original :

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

Les types à taille fixe se comportent différemment lors des conversions. Ils peuvent être considérés comme des séquences d'octets individuels et la conversion à un type plus petit coupera la séquence :

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b sera désormais 0x12
```

Si un type à taille fixe est explicitement converti en un type plus grand, il est rembourré à droite. L'accès à l'octet par un index fixe donnera la même valeur avant et après la conversion (si l'index est toujours dans la plage) :

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b sera désormais 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

Puisque les entiers et les tableaux d'octets de taille fixe se comportent différemment lorsqu'ils sont tronqués ou rembourrés, les conversions explicites entre entiers et tableaux d'octets de taille fixe ne sont autorisées que si les deux ont la même taille. Si vous voulez convertir entre des entiers et des tableaux d'octets de taille fixe de tailles différentes, vous devez utiliser des conversions intermédiaires qui font la troncature et le remplissage désirés. règles explicites :

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b sera désormais 0x00001234
uint32 c = uint32(bytes4(a)); // c sera désormais 0x12340000
uint8 d = uint8(uint16(a)); // d sera désormais 0x34
uint8 e = uint8(bytes1(a)); // d sera désormais 0x12
```

Conversions entre les types littéraux et élémentaires

Types nombres entiers

Les nombres décimaux et hexadécimaux peuvent être implicitement convertis en n'importe quel type entier suffisamment grand pour le représenter sans troncature :

```
uint8 a = 12; // Bon
uint32 b = 1234; // Bon
uint16 c = 0x123456; // échoue, car devrait tronquer en 0x3456
```

Tableaux d'octets de taille fixe

Les nombres décimaux ne peuvent pas être implicitement convertis en tableaux d'octets de taille fixe. Les nombres hexadécimaux peuvent être littéraux, mais seulement si le nombre de chiffres hexadécimaux correspond exactement à la taille du type de `bytes`. Par exception, les nombres décimaux et hexadécimaux ayant une valeur de zéro peuvent être convertis en n'importe quel type à taille fixe :

```
bytes2 a = 54321; // pas autorisé
bytes2 b = 0x12; // pas autorisé
bytes2 c = 0x123; // pas autorisé
bytes2 d = 0x1234; // bon
bytes2 e = 0x0012; // bon
bytes4 f = 0; // bon
bytes4 g = 0x0; // bon
```

Les littéraux de chaînes de caractères et les littéraux de chaînes hexadécimales peuvent être implicitement convertis en tableaux d'octets de taille fixe, si leur nombre de caractères correspond à la taille du type `bytes` :

```
bytes2 a = hex"1234"; // bon
bytes2 b = "xy"; // bon
bytes2 c = hex"12"; // pas autorisé
bytes2 d = hex"123"; // pas autorisé
bytes2 e = "x"; // pas autorisé
bytes2 f = "xyz"; // débile
```

Adresses

Comme décrit dans *Adresses Littérales*, les chaines de caractères hexadécimaux de la bonne taille qui passent le test de somme de contrôle sont de type `address`. Aucun autre littéral ne peut être implicitement converti au type `address`.

Les conversions explicites de `bytes20` ou de tout type entier en `address` aboutissent en une `address payable`.

4.4.4 Unités et variables globales

Unités d'Ether

Un nombre littéral peut prendre un suffixe de « wei », « finney », « szabo » ou « ether » pour spécifier une sous-dénomination d'éther, où les nombres d'éther sans postfix sont supposés être Wei.

```
assert(1 wei == 1);
assert(1 szabo == 1e12);
assert(1 finney == 1e15);
assert(1 ether == 1e18);
```

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

Unités de temps

Des suffixes comme `seconds`, `minutes`, `hours`, `days` et `weeks` peuvent être utilisés après les nombres littéraux pour spécifier les unités de temps où les secondes sont l'unité de base et les unités sont considérées naïvement de la façon 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 calendaires en utilisant ces unités, parce que chaque année n'est pas égale à 365 jours ni même chaque jour n'a 24 heures à cause des secondes bissextiles. Les secondes intercalaires étant imprévisibles, une bibliothèque de calendrier exacte 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. Si vous voulez interpréter une variable d'entrée en jours, par exemple, vous pouvez le faire de la manière suivante :

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

Variables spéciales et fonctions

Il y a des variables et des fonctions spéciales qui existent toujours dans l'espace de nommage global et qui sont principalement utilisées pour fournir des informations sur la chaîne de blocs ou sont des fonctions utilitaires générales.

Propriétés du bloc et des transactions

- `blockhash(uint blockNumber) returns (bytes32)` : hash du numéro de bloc passé - mnarche seulement pour les 256 plus récents, excluant le bloc courant
- `block.coinbase` (`address payable`) : addresse du mineur du bloc courant
- `block.difficulty` (`uint`) : difficulté du bloc courant
- `block.gaslimit` (`uint`) : limite de gas actuelle

— `block.number` (`uint`) : numéro du bloc courant
— `block.timestamp` (`uint`) : timestamp du bloc en temps unix (secondes)
— `gasleft() returns (uint256)` : gas restant
— `msg.data` (`bytes calldata`) : calldata complet
— `msg.sender` (`address payable`) : expéditeur du message (appel courant)
— `msg.sig` (`bytes4`) : 4 premiers octets calldata (i.e. identifiant de function)
— `msg.value` (`uint`) : nombre de wei envoyés avec le message
— `now` (`uint`) : alias pour `block.timestamp`
— `tx.gasprice` (`uint`) : prix de la transaction en gas
— `tx.origin` (`address payable`) : expéditeur de la transaction (appel global complet)

Note : Les valeurs de tous les membres de `msg`, y compris `msg.sender``et ``msg.value` peuvent changer pour chaque appel de fonction **external**. Ceci inclut les appels aux fonctions de librairies.

Note : Ne vous basez pas sur `block.timestamp`, `now` ou `blockhash` comme source de hasard, à moins de savoir ce que vous faites.

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

L'horodatage du bloc courant doit être strictement supérieur à celui du dernier bloc, mais la seule garantie est qu'il se situera entre les horodatages de deux blocs consécutifs dans la chaîne canonique.

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

Note : La fonction `blockhash` était auparavant connue sous le nom `block.blockhash`. Elle 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`. Elle a été dépréciée dans la version 0.4.21 et supprimée dans la version 0.5.0.

Fonctions d'encodage et de décodage de l'ABI

— `abi.decode(bytes memory encodedData, (...)) returns (...)` : l'ABI décode les données données, tandis que les types sont donnés entre parenthèses comme second argument. Exemple : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
— `abi.encode(...) returns (bytes memory)` : l'ABI encode les arguments passés.
— `abi.encodePacked(...) returns (bytes memory)` : exécute l'*encodage structuré* des arguments donnés
— `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` : l'ABI-encode les arguments donnés à partir du second et précède le sélecteur des quatre octets donnés.
— `abi.encodeWithSignature(string memory signature, ...) returns (bytes`

`memory)` : équivalent à `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)` `

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

Voir la documentation sur le mode d'encodage <abi_packed_mode> pour plus de détails sur l'encodage.

Gestion des erreurs

Voir la section dédiée sur *assert and require* pour plus de détails sur la gestion des erreurs et quand utiliser quelle fonction.

`assert(bool condition)` : entraîne l'utilisation d'un opcode invalide et donc la réversion du changement d'état si la condition n'est pas remplie - à utiliser pour les erreurs internes.

`require(bool condition)` : `revert` si la condition n'est pas remplie - à utiliser en cas d'erreurs dans les entrées ou les composants externes.

`require(bool condition, string memory message)` : `revert` si la condition n'est pas remplie - à utiliser en cas d'erreurs dans les entrées ou les composants externes. Fournit également un message d'erreur.

`revert()` : annuler l'exécution et annuler les changements d'état

`revert(string memory reason)` : annuler l'exécution et annuler les changements d'état, fournissant une phrase explicative

Fonctions mathématiques et cryptographiques

`addmod(uint x, uint y, uint k) returns (uint)` : calcule `(x + y) % k` où l'addition est effectuée avec une précision arbitraire et n'overflow pas à `2**256`. `assert` que `k != 0` à partir de la version 0.5.0.

`mulmod(uint x, uint y, uint k) returns (uint)` : calcule `(x * y) % k` où la multiplication est effectuée avec une précision arbitraire et n'overflow pas à `2**256`. `assert` que `k != 0` à partir de la version 0.5.0.

`keccak256(bytes memory) returns (bytes32)` : calcule le hash Keccak-256 du paramètre

`sha256(bytes memory) returns (bytes32)` : calcule le hash SHA-256 du paramètre

`ripemd160(bytes memory) returns (bytes20)` : calcule le hash RIPEMD-160 du paramètre

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)` : récupérer l'adresse associée à la clé publique à partir de la signature de la courbe elliptique ou retourner zéro sur erreur. (exemple)

Note : La fonction `ecrecover` renvoie une `address`, et non une `address payable`. Voir adresse payable pour la conversion, au cas où vous auriez besoin de transférer des fonds à l'adresse récupérée.

Il se peut que vous rencontriez `out-of-gas` pour `sha256`, `ripemd160` ou `erecover` sur une *blockchain privée*. La raison en est que ces contrats sont mis en œuvre sous la forme de contrats dits précompilés et que ces contrats n'existent réellement qu'après avoir reçu le premier message (bien que leur code contrat soit codé en dur). Les messages à des contrats inexistants sont plus coûteux et l'exécution se heurte donc à une erreur out-of-gas. Une solution de contournement pour ce problème est d'envoyer d'abord, par exemple, 1 Wei à chacun des contrats avant de les

utiliser dans vos contrats réels. Le problème n'existe pas sur la cha^ine publique Ethereum ni sur les différents testnets officiels.

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

---

Membres du type address

- **`<address>.balance (uint256)` :** balance de l'*Adresses* en Wei
- **`<address payable>.transfer(uint256 amount)` :** envoie la quantité donnée de Wei à adresse, `revert` en cas d'échec, envoie 2300 gas (non réglable)
- **`<address payable>.send(uint256 amount) returns (bool)` :** envoie la quantité donnée de Wei à adresse, retourne `false` en cas d'échec, envoie 2300 gas (non réglable)
- **`<address>.call(bytes memory) returns (bool, bytes memory)` :** émett un appel de bas niveau `CALL` avec la charge utile donnée, renvoie l'état de réussite et les données de retour, achemine tout le gas disponible ou un montant spécifié
- **`<address>.delegatecall(bytes memory) returns (bool, bytes memory)` :** émet un appel de bas niveau `DELEGATECALL` avec la charge utile donnée, retourne les données de succès et de retour, achemine tout le gas disponible ou un montant spécifié
- **`<address>.staticcall(bytes memory) returns (bool, bytes memory)` :** émettre un appel de bas niveau `STATICCALL` avec la charge utile donnée, retourne les conditions de succès et les données de retour, achemine tout le gas disponible ou un montant spécifié

Pour plus d'informations, voir la section sur adresse.

Avertissement : Il y a certains dangers à utiliser l'option `send` : Le transfert échoue si la profondeur de la pile d'appels est à 1024 (cela peut toujours être forcé par l'appelant) et il échoue également si le destinataire manque de gas. Donc, afin d'effectuer des transferts d'éther en toute sécurité, vérifiez toujours la valeur de retour de `send`, utilisez `transfer` ou mieux encore : Utilisez un modèle où le bénéficiaire retire l'argent.

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

Note : Si l'accès aux variables d'état s'effectue via un appel de délégation de bas niveau, le plan de stockage des deux contrats doit être alignée pour que le contrat appelé puisse accéder correctement aux variables de stockage du contrat appelant par leur nom. Ce n'est bien sûr pas le cas si les pointeurs de stockage sont passés comme arguments de fonction comme dans le cas des fonctions de librairies (bibliothèques) de haut niveau.

Note : Avant la version 0.5.0, `.call`, `.delegatecall` et `staticcall` ne renvoyaient que la condition de succès et non les données de retour.

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

Variables relatives au contrat

`this` (type du contrat courant) : le contrat en cours, explicitement convertible en *Adresses*.

`selfdestruct(address payable destinataire_des_fonds)` : détruire le contrat en cours, en envoyant ses fonds à l'adresse *Adresses* indiquée

En outre, toutes les fonctions du contrat en cours peuvent être appelées directement, y compris la fonction en cours.

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

4.4.5 Expressions et structures de contrôle

Paramètres d'entrée et de sortie

Comme en Javascript, les fonctions peuvent prendre des paramètres en entrée ; contrairement à Javascript et C, elles peuvent retourner plusieurs paramètres en sortie.

Paramètres d'entrée

Les paramètres d'entrée sont déclarés de la même manière que les variables. Le nom des paramètres non utilisés peut être omis. Par exemple, supposons que nous voulions que notre contrat accepte un type d'appels externes avec deux entiers, nous pourrions écrire quelque chose comme :

```
pragma solidity >=0.4.16 <0.6.0;

contract Simple {
    uint sum;
    function taker(uint _a, uint _b) public {
        sum = _a + _b;
    }
}
```

Les paramètres d'entrée peuvent être utilisés comme n'importe quelle autre variable locale, ils peuvent aussi être assignés.

Paramètres de sortie

Les paramètres de sortie peuvent être déclarés avec la même syntaxe après le mot-clé `returns`. Par exemple, supposons que nous voulions retourner deux résultats : la somme et le produit des deux entiers donnés, alors nous pourrions écrire :

```
pragma solidity >=0.4.16 <0.6.0;

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;
    }
}
```

Les noms des paramètres de sortie peuvent être omis. Les valeurs de sortie peuvent également être spécifiées à l'aide de l'instruction `returns`, également capables de *return multiple values*. Les paramètres de retour peuvent être utilisés comme n'importe quelle autre variable locale et sont initialisés zéro ; s'ils ne sont pas explicitement définis, ils restent à zéro.

Structures de controle

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

Nous disposons de : `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, avec la syntaxe famillière du C ou du JavaScript.

Les parenthèses ne peuvent *pas* être omises pour les conditions, mais les accolades peuvent être omises autour des déclaration en une opération.

Notez qu'il n'y a pas de conversion de types non booléens vers types booléens comme en C et JavaScript, donc `if (1) {...}` n'est pas valable en Solidity.

Retour de valeurs multiples

Lorsqu'une fonction a plusieurs paramètres de sortie, `return (v0, v1, ..., vn)` peut retourner plusieurs valeurs. Le nombre de composants doit être le même que le nombre de paramètres de sortie déclarés.

Appels de fonction

Appels de fonction internes

Les fonctions du contrat en cours peuvent être appelées directement (`internal`), également de manière récursive, comme le montre cet exemple absurde :

```
pragma solidity >=0.4.16 <0.6.0;

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(`JUMP`) à l'intérieur de l'EVM. Cela a pour effet que la mémoire actuelle n'est pas effacée, c'est-à-dire qu'il est très efficace de passer des références de mémoire aux fonctions appelées en interne. Seules les fonctions du même contrat peuvent être appelées en interne.

Vous devriez toujours éviter une récursivité excessive, car chaque appel de fonction interne utilise au moins un emplacement de pile et il y a au maximum un peu moins de 1024 emplacements disponibles.

Appels de fonction externes

Les expressions `this.g(8);` et `c.g(2);` (où `c` est une instance de contrat) sont aussi des appels de fonction valides, mais cette fois-ci, la fonction sera appelée `external`, via un 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 actuel n'a pas encore été créé.

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

Lors de l'appel de fonctions d'autres contrats, le montant de Wei envoyé avec l'appel et le gas peut être spécifié avec les options spéciales `.value()` et `.gas()`, respectivement :

```
pragma solidity >=0.4.0 <0.6.0;

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` pour pouvoir appeler `.value()` .

Avertissement : Veillez à ce que `feed.info.value(10).gas(800)` ne définisse que localement la `value` et la quantité de `gas` envoyés avec l'appel de fonction, et que les parenthèses à la fin sont bien présentes pour effectuer l'appel. Ainsi, dans cet exemple, la fonction n'est pas appelée.

Les appels de fonction provoquent des exceptions si le contrat appelé n'existe pas (dans le sens où le compte ne contient pas de code) ou si le contrat appelé lui-même lève une exception ou manque de gas.

Avertissement : Toute interaction avec un autre contrat présente un danger potentiel, surtout si le code source du contrat n'est pas connu à l'avance. Le contrat actuel cède le contrôle au contrat appelé et cela peut potentiellement faire à peu près n'importe quoi. Même si le contrat appelé hérite d'un contrat parent connu, le contrat d'héritage doit seulement avoir une interface correcte. L'exécution du contrat peut cependant être totalement arbitraire et donc représentent un danger. En outre, soyez prêt au cas où il appelle d'autres fonctions de votre contrat ou même de retour dans le contrat d'appel avant le retour du premier appel. 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 à les fonctions externes se produisent après tout changement de variables d'état dans votre contrat, de sorte que votre contrat n'est pas vulnérable à un exploit de réentrée.

Appels nommés et paramètres de fonction anonymes

Les arguments d'appel de fonction peuvent être donnés par leur nom, dans n'importe quel ordre, s'ils sont inclus dans `{ }` comme on peut le voir dans l'exemple qui suit. La liste d'arguments doit coïncider par son nom avec la liste des

paramètres de la déclaration de fonction, mais peut être dans un ordre arbitraire.

```
pragma solidity >=0.4.0 <0.6.0;

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 fonction omis

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

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    // omitted name for parameter
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

Création de contrats via `new`

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 lors de la compilation afin d'éviter les dépendances récursives liées à la création.

```
pragma solidity >0.4.99 <0.6.0;

contract D {
    uint public x;
    constructor(uint a) public payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // sera exécuté dans le constructor de C

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

    function createAndEndowD(uint arg, uint amount) public payable {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Envoyer des Ethers avec la création
        D newD = (new D).value(amount)(arg);
        newD.x();
    }
}
```

Comme dans l'exemple, il est possible d'envoyer des Ether en créant une instance de `D` en utilisant l'option `.value()`, mais il n'est pas possible de limiter la quantité de gas. Si la création échoue (à cause d'une rupture de pile, d'un manque de gas ou d'autres problèmes), une exception est levée.

Ordre d'évaluation des expressions

L'ordre d'évaluation des expressions n'est pas spécifié (plus formellement, 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). La seule garantie est que les instructions sont exécutées dans l'ordre et que les expressions booléennes sont court-circuitées correctement. Voir *Order of Precedence of Operators* pour plus d'informations.

Assignation

Déstructuration d'assignations et retour de valeurs multiples

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

Les tuples ne sont pas des types propres à Solidity, ils ne peuvent être utilisés que pour former des groupes syntaxiques d'expressions.

```
pragma solidity >0.4.23 <0.6.0;

contract C {
    uint[] data;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // Astuce simple pour un échange de valeurs -- Ne marche pas pour
        // les types autres que par valeur (voir Types).
        (x, y) = (y, x);
        // Certains composants peuvent être ignorés au besoin
        (data.length, , ) = f(); // Sets the length to 7
    }
}
```

Il n'est pas possible de mélanger les assignations à la déclarations et les assignations simples, c'est-à-dire que ce qui suit n'est pas valable : *(x, uint y) = (1, 2);'*

Note : Avant la version 0.5.0, il était possible d'assigner des tuples de plus petite taille, soit en les remplissant à gauche ou à droite (ce qui était vide). Ceci est maintenant interdit, de sorte que les deux côtés doivent avoir le même nombre de composants, laissés blancs si inutilisés.

Avertissement : Soyez prudent lorsque vous assignez plusieurs variables en même temps lorsqu'il s'agit de types de référence, car cela pourrait entraîner une copie inattendue.

Complications pour les tableaux et les structures

La sémantique des affectations est un peu plus compliquée pour les types autres que valeurs comme les tableaux et les structs. L'affectation *à* une variable d'état crée toujours une copie indépendante. D'autre part, l'affectation à une variable locale crée une copie indépendante uniquement pour les types élémentaires, c'est-à-dire les types statiques qui tiennent sur 32 octets. Si des structs ou des tableaux (y compris les `bytes` et les `string`) sont assignés d'une variable d'état à une variable locale, la variable locale contient une référence à la variable d'état originale. Une deuxième affectation à la variable locale ne modifie pas l'état mais seulement la référence. Les affectations aux membres (ou éléments) de la variable locale *changent* l'état.

Portée et déclarations

Une variable qui est déclarée aura une valeur par défaut initiale dont la représentation octale est égale à une suite de zéros. Les « valeurs par défaut » des variables sont les « états zéro » typiques quel que soit le type. Par exemple, la valeur par défaut d'un `bool` est `false`. La valeur par défaut pour les types `uint` ou `int` est `0`. Pour les tableaux de taille statique et les `bytes1` à `bytes32`, chaque élément individuel sera initialisé à la valeur par défaut correspondant à son type. Enfin, pour les tableaux de taille dynamique, les octets et les chaînes de caractères, la valeur par défaut est un tableau ou une chaîne vide.

La portée en Solidity suit les règles de portée très répandues du C99 (et de nombreux autres languages) : Les variables sont visibles du point situé juste après leur déclaration jusqu'à la fin du plus petit bloc `{ }` qui contient la déclaration. Par exception à cette règle, les variables déclarées dans la partie initialisation d'une boucle `for` ne sont visibles que jusqu'à la fin de la boucle for.

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 leur déclaration. Cela signifie que vous pouvez utiliser les variables d'état avant qu'elles ne soient déclarées et appeler les fonctions de manière récursive.

Par conséquent, les exemples suivants seront compilés sans avertissement, puisque les deux variables ont le même nom mais des portées disjointes.

```
pragma solidity >0.4.99 <0.6.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

À titre d'exemple particulier des règles de détermination de la portée héritées du C99, notons que, dans ce qui suit, la première affectation à `x` affectera en fait la variable externe et non la variable interne. Dans tous les cas, vous obtiendrez un avertissement concernant cette double déclaration.

```
pragma solidity >0.4.99 <0.6.0;
// Ceci déclenche un warning
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // Ceci assigne à la valeur externe
            uint x;
        }
        return x; // x == 2
    }
}
```

Avertissement :

Avant la version 0.5.0, Solidity suivait les mêmes règles de scoping que JavaScript, c'est-à-dire qu'une variable déclarée n'importe où dans une fonction était dans le champ d'application pour l'ensemble de la fonction, peu importe où elle était déclarée. L'exemple suivant montre un extrait de code qui compilait, mais conduit aujourd'hui à une erreur à partir de la version 0.5.0.

```
pragma solidity >0.4.99 <0.6.0;
// Ceci ne compile plus
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

Gestion d'erreurs : Assert, Require, Revert et Exceptions

Solidity utilise des exceptions qui restaurent l'état pour gérer les erreurs. Une telle exception annule toutes les modifications apportées à l'état de l'appel en cours (et de tous ses sous-appels) et signale également une erreur à l'appelant. Les fonctions bien 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` ne doit être utilisée que pour tester les erreurs internes, et pour vérifier les invariants. La fonction `require` doit être utilisée pour s'assurer que les conditions valides, telles que les entrées ou les variables d'état du contrat, sont remplies, ou pour valider les valeurs de retour des appels aux contrats externes. S'ils sont utilisés correctement, les outils d'analyse peuvent évaluer votre contrat afin d'identifier les conditions et les appels de fonction qui parviendront à un échec d'"`assert`. Un code fonctionnant correctement ne devrait jamais échouer un `assert` ; si cela se produit, il y a un bogue dans votre contrat que vous devriez corriger.

Il y a deux autres façons de déclencher des exceptions : La fonction `revert` peut être utilisée pour signaler une erreur et annuler l'appel en cours. Il est possible de fournir une chaîne de caractères contenant des détails sur l'erreur qui sera renvoyée à l'appelant.

Note : Il y avait un mot-clé appelé `throw` avec la même sémantique que `revert()` qui était déprécié dans la version 0.4.13 et supprimé dans la version 0.5.0.

Lorsque des exceptions se produisent dans un sous-appel, elles « remontent à la surface » automatiquement (c'est-à-dire que les exceptions sont déclenchées en casacade). Les exceptions à cette règle sont `send` et les fonctions de bas niveau `call`, `delegatecall` et `staticcall`, qui retournent `false` comme première valeur de retour en cas d'exception au lieu de lancer une chaine d'exceptions.

Avertissement : Les fonctions de bas niveau `call`, `delegatecall` et `staticcall` renvoient `true` comme première valeur de retour si le compte appelé est inexistant, dû à la conception de l'EVM. L'existence doit être vérifiée avant l'appel si désiré.

Il n'est pas encore possible de réellement réagir aux exceptions.

Dans l'exemple suivant, vous pouvez voir comment `require` peut être utilisé pour vérifier facilement les conditions sur les entrées et comment `assert` peut être utilisé pour vérifier les erreurs internes. Notez que vous pouvez facultativement fournir une chaîne de message pour `require`, mais pas pour `assert`.

```
pragma solidity >0.4.99 <0.6.0;

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

Une exception de type `assert` est générée dans les situations suivantes :

1. Si vous accédez à un tableau avec un index trop grand ou négatif (par ex. `x[i]` où `i >= x.length` ou `i < 0`).
2. Si vous accédez à une variable de longueur fixe `bytesN` à un indice trop grand ou négatif.
3. Si vous divisez ou modulez par zéro (par ex. `5 / 0` ou `23 % 0`).
4. Si vous décalez d'un montant négatif.
5. Si vous convertissez une valeur trop grande ou négative en un type enum.
6. Si vous appelez une variable initialisée nulle de type fonction interne.
7. Si vous appelez `assert` avec un argument qui s'évalue à `false`.

Une exception de type `assert` est générée dans les situations suivantes :

1. Appeler `require` avec un argument qui s'évalue à `false`.
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 n'a plus de gas, qu'elle n'a pas de fonction correspondante ou qu'elle lance une exception elle-même), sauf lorsqu'une opération de bas niveau `call`, `send`, `staticcall`, `delegatecall` ou `callcode` est utilisée. Les opérations de bas niveau ne lancent jamais d'exceptions mais indiquent les échecs en retournant `false`.

3. Si vous créez un contrat en utilisant le mot-clé `new` mais que la création du contrat ne se termine pas correctement (voir ci-dessus pour la définition de « ne pas terminer correctement »).
4. Si vous effectuez un appel de fonction externe ciblant un contrat qui ne contient aucun code.
5. Si votre contrat reçoit des Ether via une fonction publique sans modificateur `payable` (y compris le constructeur et la fonction par defaut).
6. Si votre contrat reçoit des Ether via une fonction de getter public.
7. Si un `.transfer()` échoue.

En interne, Solidity exécute une opération de retour en arrière (instruction `0xfd`) pour une exception de type `require` et exécute une opération invalide (instruction `0xfe`) pour lancer une exception de type `assert`. Dans les deux cas, cela provoque lánnulation toutes les modifications apportées à l'état de l'EVM dans l'appel courant. La raison du retour en arrière est qu'il n'y a pas de moyen sûr de continuer l'exécution, parce qu'un effet attendu ne s'est pas produit. Parce que nous voulons conserver l'atomicité des transactions, la chose la plus sûre à faire est d'annuler tous les changements et de faire toute la transaction (ou au moins l'appel) sans effet.

Note : Les exceptions de type `assert` consomment tout le gas disponible pour l'appel, alors que les exceptions de type `require` ne consommeront pas de gaz à partir du lancement de Metropolis.

L'exemple suivant montre comment une chaîne d'erreurs peut être utilisée avec `revert` et `require` :

```
pragma solidity >0.4.99 <0.6.0;

contract VendingMachine {
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Autre façon de le faire:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Effectuer l'achat
    }
}
```

La chaîne fournie sera *abi-encoded* comme si c'était un appel à une fonction `Error(string)`. Dans l'exemple ci-dessus, `revert("Not enough Ether provided.");` fera en sorte que les données hexadécimales suivantes soient définies comme données de retour d'erreur :

```
0x08c379a0                                                         // Selecteur de␣
↪fonction pour Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Décalage des␣
↪données
0x000000000000000000000000000000000000000000000000000000000000001a // Taille de la␣
↪string
0x4e6f7420656e6f756768204574686572207072f76696465642e000000000000 // Données de la␣
↪string
```

4.4.6 Contracts

Les contrats en Solidity sont similaires à des classes dans les langages orientés objets. Ils contiennent des données persistentes dans des variables et des fonctions peuvent les modifier. Appeler la fonction d'un autre contrat (une autre instance) executera un appel de fonction auprès de l'EVM et changera alors le contexte, rendant inaccessibles ces variables.

Créer des contrats

Les contrats peuvent être créés « en dehors » via des transactions Ethereum ou depuis des contrat en Solidity.

Les EDIs, comme Remix, facilite la tâche via des éléments visuels.

Créer des contrats via du code se fait le plus simplement en utilisant l'API Javascript web3.js. Elle possède une fonction appelée web3.eth.Contract qui facilite cette création

Quand un contrat est créé, son constructeur (une fonction déclarée via le mot-clé `constructor`) est executé, de manière unique.

Un constructeur est optionnel. Aussi, un seul constructeur est autorisé, ce qui signifie que l'overloading n'est pas supporté.

Après que le constructeur a été exécuté, le code final du contrat est déployé sur la Blockchain. Ce code inclut toutes les fonctions publiques et externes, et toutes les fonctions qui sont atteignables par des appels de fonctions. Le code déployé n'inclut pas le constructeur ou les fonctions internes uniquement appelées depuis le constructeur.

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.

```
pragma solidity >=0.4.22 <0.6.0;

contract OwnedToken {
    // TokenCreator is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // This is the constructor which registers the
    // creator and the assigned name.
    constructor(bytes32 _name) public {
        // State variables are accessed via their name
        // and not via e.g. this.owner. This also applies
        // to functions and especially in the constructors,
        // you can only call them like that ("internally"),
        // because the contract itself does not exist yet.
        owner = msg.sender;
        // We do 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 check that.
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) public {
        // Only the creator can alter the name --
        // the comparison is possible since contracts
        // are explicitly convertible to addresses.
        if (msg.sender == address(creator))
            name = newName;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function transfer(address newOwner) public {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;

        // We also want to ask the creator if the transfer
        // is fine. Note that this calls a function of the
        // 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
       returns (OwnedToken tokenAddress)
    {
        // Create a new Token contract and return its address.
        // From the JavaScript side, the return type is simply
        // `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);
    }

    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // Check some arbitrary condition.
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

Visibility and Getters

Since 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), there are four types of visibilities 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). External functions are sometimes more efficient when they receive large arrays of data.

`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`.

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 accessing and 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.

```
pragma solidity >=0.4.16 <0.6.0;

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`.

```
pragma solidity >=0.4.0 <0.6.0;

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.

```
pragma solidity >=0.4.0 <0.6.0;

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.

```
pragma solidity >=0.4.0 <0.6.0;

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 `data(0)`. If you want to return an entire array in one call, then you need to write a function, for example :

```
pragma solidity >=0.4.0 <0.6.0;

contract arrayExample {
  // public state variable
  uint[] public myArray;

  // Getter function generated by the compiler
  /*
  function myArray(uint i) returns (uint) {
      return myArray[i];
  }
  */

  // function that returns entire array
  function getArray() 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 :

```
pragma solidity >=0.4.0 <0.6.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

It generates a function of the following form. The mapping in the struct is omitted because there is no good way to provide the key for the mapping :

```
function data(uint arg1, bool arg2, uint arg3) public returns (uint a, bytes3 b) {
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
}
```

Function Modifiers

Modifiers can be used to easily change the behaviour of functions. For example, they can automatically check a condition prior to executing the function. Modifiers are inheritable properties of contracts and may be overridden by derived contracts.

```
pragma solidity >0.4.99 <0.6.0;

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

    // This contract only defines a modifier but does not use
    // it: it will be used in derived contracts.
    // The function body is inserted where the special symbol
    // `_;` in the definition of a modifier appears.
    // This means that if the owner calls this function, the
    // function is executed and otherwise, an exception is
    // thrown.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract mortal is owned {
    // This contract inherits the `onlyOwner` modifier from
    // `owned` and applies it to the `close` function, which
    // causes that calls to `close` only have an effect if
    // they are made by the stored owner.
    function close() public onlyOwner {
        selfdestruct(owner);
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract priced {
    // Modifiers can receive arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;

    constructor(uint initialPrice) public { price = initialPrice; }

    // It is important to also provide the
    // `payable` keyword here, otherwise the function will
    // automatically reject all Ether sent to it.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// This function is protected by a mutex, which means that
    /// reentrant calls from within `msg.sender.call` cannot call `f` again.
    /// The `return 7` statement assigns 7 to the return value but still
    /// executes the statement `locked = false` in the modifier.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.

Avertissement : In an earlier version of Solidity, `return` statements in functions having modifiers behaved differently.

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.

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).

Constant State Variables

State variables can be declared as `constant`. In this case, they have to be assigned from an expression which is a constant at compile time. Any expression that accesses storage, blockchain data (e.g. `now`, `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 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.

The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective constant expression (which might be computed to a single value by the optimizer).

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

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    uint constant x = 32**22 + 8;
    string constant text = "abc";
    bytes32 constant myHash = keccak256("abc");
}
```

Functions

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 for `view` functions 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.

```
pragma solidity >0.4.99 <0.6.0;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + now;
    }
}
```

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.

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.

```
pragma solidity >0.4.99 <0.6.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

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.

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).

Avertissement : Before 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.

Fallback Function

A contract can have exactly one unnamed function. This function cannot have arguments, cannot return anything and has to have `external` visibility. It is executed on a call to the contract if none of the other functions match the given function identifier (or if no data was supplied at all).

Furthermore, this function is executed whenever the contract receives plain Ether (without data). Additionally, in order to receive Ether, the fallback function must be marked `payable`. If no such function exists, the contract cannot receive Ether through regular transactions.

In the worst case, the fallback 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

Like any function, the fallback function can execute complex operations as long as there is enough gas passed on to it.

Note : Even though the fallback function cannot have arguments, one can still use `msg.data` to retrieve any payload supplied with the call.

Avertissement : The fallback function is also executed if the caller meant to call a function that is not available. If you want to implement the fallback function only to receive ether, you should add a check like `require(msg.data.length == 0)` to prevent invalid calls.

Avertissement : Contracts that receive Ether directly (without a function call, i.e. using `send` or `transfer`) but do not define a 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 payable fallback function.

Avertissement : A contract without a payable fallback 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 fallback function).

```
pragma solidity >0.4.99 <0.6.0;

contract Test {
    // 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.
    function() external { x = 1; }
    uint x;
}


// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    function() external payable { }
}

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␣
↪via an
        // intermediate conversion to ``uint160`` to even allow calling ``send`` on␣
↪it.
        address payable testPayable = address(uint160(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 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`.

```
pragma solidity >=0.4.16 <0.6.0;

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.

```
pragma solidity >=0.4.16 <0.6.0;

// 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;
    }
}

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.

```
pragma solidity >=0.4.16 <0.6.0;

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`.

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 the Frontier and Homestead releases, 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 simple payment verification (SPV) for logs, so if an external entity supplies a contract with such a verification, 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. If you use arrays (including `string` and `bytes`) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a topic can only hold a single word (32 bytes).

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.

```
pragma solidity >=0.4.21 <0.6.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 event = clientReceipt.Deposit();

// watch for changes
event.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 event = 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"]
   }
}
```

Low-Level Interface to Logs

It is also possible to access the low-level interface to the logging mechanism via the functions `log0`, `log1`, `log2`, `log3` and `log4`. `logi` takes `i + 1` parameter of type `bytes32`, where the first argument will be used for the data part of the log and the others as topics. The event call above can be performed in the same way as

```
pragma solidity >=0.4.10 <0.6.0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract C {
    function f() public payable {
        uint256 _id = 0x420042;
        log3(
            bytes32(msg.value),
            bytes32(0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20),
            bytes32(uint256(msg.sender)),
            bytes32(_id)
        );
    }
}
```

where the long hexadecimal number is equal to `keccak256("Deposit(address,bytes32,uint256)")`, the signature of the event.

Additional Resources for Understanding Events

- Javascript documentation
- Example usage of events
- How to access them in js

Inheritance

Solidity supports multiple inheritance by copying code including polymorphism.

All function calls are virtual, which means that the most derived function is called, except when the contract name is explicitly given.

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 copied into the created contract.

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.

```
pragma solidity >0.4.99 <0.6.0;

contract owned {
    constructor() public { owner = 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 mortal is owned {
    function kill() 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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract Config {
    function lookup(uint id) public returns (address adr);
}

contract NameReg {
    function register(bytes32 name) public;
    function unregister() public;
 }

// Multiple inheritance is possible. Note that `owned` is also a base class of␣
↪`mortal`, yet there is only a single instance of `owned` (as for virtual␣
↪inheritance in C++).
contract named is owned, mortal {
    constructor(bytes32 name) public {
        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.
    function kill() public {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // It is still possible to call a specific overridden function.
            mortal.kill();
        }
    }
}

// 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, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) public {
      if (msg.sender == owner) info = newInfo;
   }

   function get() public view returns(uint r) { return info; }

   uint info;
}
```

Note that above, we call `mortal.kill()` to « forward » the destruction request. The way this is done is problematic, as seen in the following example :

```
pragma solidity >=0.4.22 <0.6.0;

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

contract mortal is owned {
    function kill() public {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* do cleanup 1 */ mortal.kill(); }
}

contract Base2 is mortal {
    function kill() public { /* do cleanup 2 */ mortal.kill(); }
}

contract Final is Base1, Base2 {
}
```

A call to `Final.kill()` will call `Base2.kill` as the most derived override, but this function will bypass `Base1.kill`, basically because it does not even know about `Base1`. The way around this is to use `super` :

```
pragma solidity >=0.4.22 <0.6.0;

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

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* do cleanup 1 */ super.kill(); }
}


contract Base2 is mortal {
    function kill() public { /* do cleanup 2 */ super.kill(); }
}

contract Final is Base1, Base2 {
}
```

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.kill()` (note that the final inheritance sequence is – starting with the most derived contract : Final, Base2, Base1, mortal, 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.

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 zero 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.

Constructor functions can be either `public` or `internal`. If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() public {}`. For example :

```
pragma solidity >0.4.99 <0.6.0;

contract A {
    uint public a;

    constructor(uint _a) internal {
        a = _a;
    }
}

contract B is A(1) {
    constructor() public {}
}
```

A constructor set as `internal` causes the contract to be marked as *abstract*.

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.

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 :

```
pragma solidity >=0.4.22 <0.6.0;

contract Base {
    uint x;
    constructor(uint _x) public { x = _x; }
}

// Either directly specify in the inheritance list...
contract Derived1 is Base(7) {
    constructor() public {}
}

// or through a "modifier" of the derived constructor.
contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) public {}
}
```

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 theerror « Linearization of inheritance graph impossible ».

```
pragma solidity >=0.4.0 <0.6.0;

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.

Inheriting Different Kinds of Members of the Same Name

When the inheritance results in a contract with a function and a modifier of the same name, it is considered as an error. This error is produced also by an event and a modifier of the same name, and a function and an event of the same name. As an exception, a state variable getter can override a public function.

Abstract Contracts

Contracts are marked as abstract when at least one of their functions lacks an implementation as in the following example (note that the function declaration header is terminated by `;`) :

```
pragma solidity >=0.4.0 <0.6.0;

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

Such contracts cannot be compiled (even if they contain implemented functions alongside non-implemented functions), but they can be used as base contracts :

```
pragma solidity >=0.4.0 <0.6.0;

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

contract Cat is Feline {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function utterance() public returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it will itself be abstract.

Note that a function without implementation is different from a *Function Type* even though their syntax looks very similar.

Example of function without implementation (a function declaration) :

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

Example of a Function Type (a variable declaration, where the variable is of type `function`) :

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

Abstract contracts decouple the definition of a contract from its implementation providing better extensibility and self-documentation and facilitating patterns like the Template method and removing code duplication. Abstract contracts are useful in the same way that defining methods in an interface is useful. It is a way for the designer of the abstract contract to say « any child of mine must implement this method ».

Interfaces

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

- They cannot inherit other contracts or interfaces.
- All declared functions must be external.
- They cannot declare a constructor.
- They cannot declare state variables.

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 :

```
pragma solidity >=0.4.11 <0.6.0;

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.

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

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 (`L.f()` if `L` is the name of the library). Furthermore, `internal` functions of libraries are visible in all contracts, just as if the library were a base contract. 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, code of internal library functions and all functions called from therein will at compile time be pulled into the calling contract, and a regular `JUMP` call will be used instead of a `DELEGATECALL`.

The following example illustrates how to use libraries (but manual method be sure to check out *using for* for a more advanced example to implement a set).

```
pragma solidity >=0.4.22 <0.6.0;

library Set {
  // We define a new struct datatype that will be used to hold its data in the␣
↪calling contract.
  struct Data { mapping(uint => bool) flags; }

  // Note that the first parameter is of type "storage reference" and thus only its␣
↪storage address and not its contents is passed as part of the call.  This is a␣
↪special feature of library functions.  It is idiomatic to call the first parameter␣
↪`self`, if the function can be seen as a method of that object.
  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      public
      view
      returns (bool)
```

(suite sur la page suivante)

(suite de la page précédente)

```
  {
      return self.flags[value];
  }
}

contract C {
    Set.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 :

```
pragma solidity >=0.4.16 <0.6.0;

library BigInt {
    struct bigint {
        uint[] limbs;
    }

    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);
            r.limbs[i] = a + b + carry;
            if (a + b < a || (a + b == uint(-1) && 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
            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.bigint;

    function f() public pure {
        BigInt.bigint memory x = BigInt.fromUint(7);
        BigInt.bigint memory y = BigInt.fromUint(uint(-1));
        BigInt.bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}
```

As the compiler cannot know where the library will be deployed at, these addresses have to be filled into the final bytecode by a linker (see *Using the Commandline Compiler* for how to use the commandline compiler for linking). If the addresses are not given as arguments to the compiler, the compiled hex code will contain placeholders of the form `__Set_______` (where `Set` is the name of the library). The address can be filled manually by replacing all those 40 symbols by the hex encoding of the address of the library contract.

Note : Manually linking libraries on the generated bytecode is discouraged, because it is restricted to 36 characters. You should ask the compiler to link the libraries at the time a contract is compiled by either using the `--libraries` option of `solc` or the `libraries` key if you use the standard-JSON interface to the compiler.

Restrictions for libraries in comparison to contracts :

- No state variables
- Cannot inherit nor be inherited
- Cannot receive Ether

(These might be lifted at a later point.)

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.

Using For

The directive `using A for B;` can be used to attach library functions (from the library `A`) to any type (`B`). 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.

By including a library, its data types including library functions are available without having to add further code.

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

```
pragma solidity >=0.4.16 <0.6.0;

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

  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
        return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      public
      view
      returns (bool)
  {
      return self.flags[value];
  }
}
```

(suite sur la page suivante)

(suite de la page précédente)

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

    function register(uint value) public {
        // Here, all variables of type Set.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 :

```
pragma solidity >=0.4.16 <0.6.0;

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;
        return uint(-1);
    }
}

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 == uint(-1))
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Note that all 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.

4.4.7 Assembleur Solidity

L'EVM définit un langage assembleur que vous pouvez utiliser dans Solidity en assembleur en ligne (« inline assembly ») dans le code source ou de manière indépendante. Ce guide commence par décrire comment utiliser l'assembleur en ligne, en quoi il diffère de l'assemblage autonome, et spécifie l'assemblage lui-même.

Assembleur en ligne (inline)

Vous pouvez entrelacer les instructions en Solidity avec de l'assembleur en ligne, dans un langage proche de celui de la machine virtuelle. Cela vous donne un contrôle plus fin, en particulier lorsque vous améliorez le langage en écrivant des bibliothèques.

Comme l'EVM est une machine à pile, il est souvent difficile d'adresser le bon emplacement de pile et de fournir des arguments aux opcodes au bon endroit sur la pile. L'assembleur en ligne de Solidity vous aide à le faire, ainsi qu'avec d'autres problèmes qui surviennent lors de la rédaction manuelle d'assembleur.

L'assembleur en ligne présente les caractéristiques suivantes :

- opcodes de type fonctionnels : `mul(1, add(2, 3))`
- variables assembleur locales : `let x := add(2, 3) let y := mload(0x40) x := add(x, y)`
- accèss à des variables externes : `function f(uint x) public { assembly { x := sub(x, 1) } }`
- boucles : `for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }`
- conditions if : `if slt(x, 0) { x := sub(0, x) }`
- conditions switch : `switch x case 0 { y := mul(x, 2) } default { y := 0 }`
- appels de fonction : `function f(x) -> y { switch x case 0 { y := 1 } default { y := mul(x, f(sub(x, 1))) } }`

Avertissement : L'assembleur en ligne est un moyen d'accéder à la machine virtuelle Ethereum en bas niveau. Ceci permet de contourner plusieurs normes de sécurité importantes et contrôles de Solidity. Vous ne devriez l'utiliser que pour les tâches qui en ont besoin, et seulement si vous êtes sûr de pourquoi/comment l'utiliser.

Syntaxe

L'assembleur analyse les commentaires, les littéraux et les identificateurs de la même manière que Solidity, vous pouvez donc utiliser les commentaires habituels `//` et `/* */`. L'assembleur en ligne est délimité par `assembly { ... }` et à l'intérieur de ces accolades, vous pouvez utiliser ce qui suit (voir les sections suivantes pour plus de détails) :

- litéraux, par ex. `0x123`, `42` ou `"abc"` (strings jusqu'à 32 caractères)
- opcodes de type fonctionnels, exemple `add(1, mlod(0))`
- déclaration de variables, ex. `let x := 7`, `let x := add(y, 3)` or `let x` (Initialisées à 0)
- identifieurs (variables assembleur locales et externes si utilisée en tant qu'assembleur en ligne), ex. `add(3, x)`, `sstore(x_slot, 2)`
- assignations, ex. `x := add(y, 3)`
- blocks de délimitation de portée, ex. `{ let x := 3 { let y := add(x, 1) } }`

Les caractéristiques suivantes ne sont disponibles que dans l'assembleur utilisé seul :

- contrôle direct de la stack `dup1`, `swap1`, ...
- assignations directement sur la stack (de style instruction), e.g. `3 =: x`
- étiquettes, ex. `name:`
- opcodes de saut

Note : L'assembleur autonome est rétrocompatible mais n'est plus documenté ici.

À la fin du bloc `assembly { ... }`, la pile doit être équilibrée, à moins que vous n'en ayez besoin autrement. S'il n'est pas équilibré, le compilateur génère un avertissement.

Exemple

L'exemple suivant fournit le code de bibliothèque pour accéder au code d'un autre contrat et le charger dans une variable `bytes`. Ce n'est pas possible de base avec Solidity et l'idée est que les bibliothèques assembleur seront utilisées pour améliorer le langage Solidity.

```
pragma solidity >=0.4.0 <0.6.0;

library GetCode {
    function at(address _addr) public view returns (bytes memory o_code) {
        assembly {
            // récupère la taille du code, a besoin d'assembleur
            let size := extcodesize(_addr)
            // allouer le tableau de bytes de sortie - ceci serait fait en Solidity␣
↪via o_code = new bytes(size)
            o_code := mload(0x40)
            // nouvelle "fin de mémoire" en incluant le padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // stocke la taille en mémoire
            mstore(o_code, size)
            // récupère le code lui-même, nécessite de l'assembleur
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

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

```
pragma solidity >=0.4.16 <0.6.0;

library VectorSum {
    // Cette fonction est moins efficace car l'optimiseur ne parvient
    // pas à supprimer les contrôles de limites dans l'accès aux tableaux.
    function sumSolidity(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // Nous savons que nous n'accédons au tableau que dans ses
    // limites, ce qui nous permet d'éviter la vérification. 0x20
    // doit être ajouté à un tableau car le premier emplacement
    // contient la longueur du tableau.
    function sumAsm(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                o_sum := add(o_sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Même chose que ci-dessus, mais exécute le code entier en assembleur en ligne.
    function sumPureAsm(uint[] memory _data) public pure returns (uint o_sum) {
        assembly {
            // Charge la taille (premiers 32 bytes)
            let len := mload(_data)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            // Saute le champ de taille.
            //
            // Garde une variable temporaire pour pouvoir l'incrémenter.
            //
            // NOTE: incrémenter _data resulterait en une
            // variable _data inutilisable après ce bloc d'assembleur
            let data := add(_data, 0x20)

            // Itère jusqu'à la limite.
            for
                { let end := add(data, mul(len, 0x20)) }
                lt(data, end)
                { data := add(data, 0x20) }
            {
                o_sum := add(o_sum, mload(data))
            }
        }
    }
}
```

Opcodes

Ce document ne se veut pas une description complète de la machine virtuelle Ethereum, mais la liste suivante peut être utilisée comme référence de ses opcodes.

Si un opcode prend des arguments (toujours du haut de la pile), ils sont donnés entre parenthèses. Notez que l'ordre des arguments peut être vu comme étant inversé dans un style non fonctionnel (expliqué ci-dessous). Les opcodes marqués par − ne poussent pas un article sur la pile, ceux marqués par `*` sont spéciaux et tous les autres poussent exactement une valeur sur la pile. Les opcodes marqués avec `F`, `H`, `B` ou `C` sont présents depuis Frontier, Homestead, Byzantium ou Constantinople, respectivement. Constantinople est toujours en cours de planification et toutes les instructions marquées comme telles entraîneront une exception d'instruction invalide à ce stade.

Dans ce qui suit, `mem[a...b]` signifie les octets de mémoire commençant à la position `a` jusqu'à la position `b` mais non comprise et `storage[p]` signifie le contenu du stockage à la position `p`.

Les opcodes `pushi` et `jumpdest` ne peuvent pas être utilisés directement.

Dans la grammaire, les opcodes sont représentés comme des identificateurs prédéfinis.

Instruction			Explication
stop	-	F	arrêt de l'exécution, identique 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
sdiv(x, y)		F	x / y, pour les nombres signés en complément à deux
mod(x, y)		F	x % y
smod(x, y)		F	x % y, pour les nombres signés en complément à deux

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Instruction			Explication
exp(x, y)		F	x exposant y
not(x)		F	~x, chaque bit de x est inversé
lt(x, y)		F	1 si x < y, 0 sinon
gt(x, y)		F	1 si x > y, 0 sinon
slt(x, y)		F	1 si x<y, 0 sinon, pour les nombres signés en complément à deux
sgt(x, y)		F	1 si x>y, 0 sinon, pour les nombres signés en complément à deux
eq(x, y)		F	1 si x == y, 0 sinon
iszero(x)		F	1 si x == 0, 0 sinon
and(x, y)		F	and binaire de x et y
or(x, y)		F	or binaire de x et y
xor(x, y)		F	xor binaire de x et y
byte(n, x)		F	nème octet de x, où le bit de poids fort est le 0ème
shl(x, y)		C	décalage logique binaire de y à gauche de x bits
shr(x, y)		C	décalage logique binaire de y à droite de x bits
sar(x, y)		C	décalage arithmétique de y à droite de x bits
addmod(x, y, m)		F	(x + y) % m arithmétique de précision arbitraire
mulmod(x, y, m)		F	(x * y) % m arithmétique de précision arbitraire
signextend(i, x)		F	signe déplacé au i*8+7 ème bit en partant du bit de poids faible
keccak256(p, n)		F	keccak(mem[p...(p+n)))
jump(label)	-	F	saute à l'étiquette / position dans le code
jumpi(label, cond)	-	F	saute à l'étiquette si cond différent de 0
pc		F	position actuelle dans le code
pop(x)	-	F	retire l'élément poussé sur la stack par x
dup1 ... dup16		F	copie le nième emplacement (du haut) de la pile sur le dessus
swap1 ... swap16	*	F	échange l'élément du dessus de la pile avec le nième en dessous
mload(p)		F	mem[p...(p+32))
mstore(p, v)	-	F	mem[p...(p+32)) := v
mstore8(p, v)	-	F	mem[p] := v & 0xff (modifie uniquement un bit)

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Instruction			Explication
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
msize		F	taille actuelle de memory, c.à.d plus grand index mémoire
gas		F	gas toujours disponible à l'exécution
address		F	addresse du contrat en cours / du contexte d'exécution
balance(a)		F	solde en wei de l'adresse a
caller		F	emetteur du message (excluant `delegatecall`)
callvalue		F	wei envoyés avec l'appel courant
calldataload(p)		F	données d'appel calldata commençant à la position p (32 octets)
calldatasize		F	taille des données d'appel en octets
calldatacopy(t, f, s)	-	F	copie s octets de la position f de calldata vers t en memoire
codesize		F	size of the code of the current contract / execution context
codecopy(t, f, s)	-	F	copy s bytes from code at position f to mem at position t
extcodesize(a)		F	size of the code at address a
extcodecopy(a, t, f, s)	-	F	comme codecopy(t, f, s) mais prend le code à l'adresse a
returndatasize		B	taille du dernier returndata
returndatacopy(t, f, s)	-	B	copie s octets de la position f de returndata vers t en mémoire
extcodehash(a)		C	hash du code de l'adresse a
create(v, p, n)		F	créée un nouveau contrat avec le code mem[p...(p+n)) et envoie v wei puis retourne la nouvelle adresse

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Instruction			Explication
create2(v, p, n, s)		C	créée un nouveau contrat avec le code mem[p... (p+n)) à l'adresse keccak256(0xff . this . s . keccak256(mem[p... (p+n))) et envoie v wei puis retourne la nouvelle adresse, où `0xff` est une valeur sur 8 octets, `this` est l'adresse du contrat courant sur 20 octets et `s` est une valeur 256 bits en big-endian
call(g, a, v, in, insize, out, outsize)		F	appelle le contrat à l'adresse a avec les données d'entrée mem[in... (in+insize)), en fournissant g gas et v wei et l'espace mémoire de sortie mem[out... (out+outsize)), retournant 0 en cas d'erreur (ex. manque de gas) et 1 en cas de succès
callcode(g, a, v, in, insize, out, outsize)		F	identique à `call` mais utilise seulement le code de a en restant dans le contexte du contrat courant
delegatecall(g, a, in, insize, out, outsize)		H	identique à `callcode` mais garde également `caller` et `callvalue`
staticcall(g, a, in, insize, out, outsize)		B	identique à `call(g, a, 0, in, insize, out, outsize)` mais n'autorise pas de modifications de l'état
return(p, s)	-	F	termine l'exécution, retourne data mem[p... (p+s))
revert(p, s)	-	B	termine l'exécution, annule les changement de l'état, retourne data mem[p... (p+s))

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Instruction			Explication
selfdestruct(a)	-	F	termine l'exécution, détruit le contrat en cours et envoie ses fonds à a
invalid	-	F	termine l'exécution with invalid instruction
log0(p, s)	-	F	ajoute data mem[p...(p+s)) au journal sans topics
log1(p, s, t1)	-	F	ajoute data mem[p...(p+s)) au journal avec le topic t1
log2(p, s, t1, t2)	-	F	ajoute data mem[p...(p+s)) au journal avec les topics t1 et t2
log3(p, s, t1, t2, t3)	-	F	ajoute data mem[p...(p+s)) au journal avec les topics t1, t2, t3
log4(p, s, t1, t2, t3, t4)	-	F	ajoute data mem[p...(p+s)) au journal avec topics t1, t2, t3, t4
origin		F	émetteur de la transaction
gasprice		F	prix du gas pour cette transaction
blockhash(b)		F	hash du bloc numero b seulement pour els derniers 256 blocs excluant le courant
coinbase		F	bénéficiaire du minage courant
timestamp		F	timestamp du bloc courant en secondes depuis l'epoch UNIX
number		F	numéro du bloc courant
difficulty		F	difficulté du bloc courant
gaslimit		F	limite de gas du bloc courant

Literaux

You can use integer constants by typing them in decimal or hexadecimal notation and an appropriate `PUSHi` instruction will automatically be generated. The following creates code to add 2 and 3 resulting in 5 and then computes the bitwise and with the string « abc ». The final value is assigned to a local variable called `x`. Strings are stored left-aligned and cannot be longer than 32 bytes.

```
assembly { let x := and("abc", add(3, 2)) }
```

Functional Style

For a sequence of opcodes, it is often hard to see what the actual arguments for certain opcodes are. In the following example, `3` is added to the contents in memory at position `0x80`.

```
3 0x80 mload add 0x80 mstore
```

Solidity inline assembly has a « functional style » notation where the same code would be written as follows :

```
mstore(0x80, add(mload(0x80), 3))
```

If you read the code from right to left, you end up with exactly the same sequence of constants and opcodes, but it is much clearer where the values end up.

If you care about the exact stack layout, just note that the syntactically first argument for a function or opcode will be put at the top of the stack.

Access to External Variables, Functions and Libraries

You can access Solidity variables and other identifiers by using their name. For variables stored in the memory data location, this pushes the address, and not the value onto the stack. Variables stored in the storage data location are different, as they might not occupy a full storage slot, so their « address » is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x_slot`, and to retrieve the byte-offset you use `x_offset`.

Local Solidity variables are available for assignments, for example :

```
pragma solidity >=0.4.11 <0.6.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            r := mul(x, sload(b_slot)) // ignore the offset, we know it is zero
        }
    }
}
```

Avertissement : If you access variables of a type that spans less than 256 bits (for example `uint64`, `address`, `bytes16` or `byte`), you cannot make any assumptions about bits not part of the encoding of the type. Especially, do not assume them to be zero. To be safe, always clear the data properly before you use it in a context where this is important : `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` To clean signed types, you can use the `signextend` opcode.

Labels

Support for labels has been removed in version 0.5.0 of Solidity. Please use functions, loops, if or switch statements instead.

Declaring Assembly-Local Variables

You can use the `let` keyword to declare variables that are only visible in inline assembly and actually only in the current `{...}`-block. What happens is that the `let` instruction will create a new stack slot that is reserved for the variable and automatically removed again when the end of the block is reached. You need to provide an initial value for the variable which can be just `0`, but it can also be a complex functional-style expression.

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    function f(uint x) public view returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
                b := y
            } // y is "deallocated" here
            b := add(b, v)
        } // v is "deallocated" here
    }
}
```

Assignments

Assignments are possible to assembly-local variables and to function-local variables. Take care that when you assign to variables that point to memory or storage, you will only change the pointer and not the data.

Variables can only be assigned expressions that result in exactly one value. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables.

```
{
    let v := 0
    let g := add(v, 2)
    function f() -> a, b { }
    let c, d := f()
}
```

If

The if statement can be used for conditionally executing code. There is no « else » part, consider using « switch » (see below) if you need multiple alternatives.

```
{
    if eq(value, 0) { revert(0, 0) }
}
```

The curly braces for the body are required.

Switch

You can use a switch statement as a very basic version of « if/else ». It takes the value of an expression and compares it to several constants. The branch corresponding to the matching constant is taken. Contrary to the error-prone behaviour

of some programming languages, control flow does not continue from one case to the next. There can be a fallback or default case called `default`.

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

The list of cases does not require curly braces, but the body of a case does require them.

Loops

Assembly supports a simple for-style loop. For-style loops have a header containing an initializing part, a condition and a post-iteration part. The condition has to be a functional-style expression, while the other two are blocks. If the initializing part declares any variables, the scope of these variables is extended into the body (including the condition and the post-iteration part).

The following example computes the sum of an area in memory.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

For loops can also be written so that they behave like while loops : Simply leave the initialization and post-iteration parts empty.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {      // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

Functions

Assembly allows the definition of low-level functions. These take their arguments (and a return PC) from the stack and also put the results onto the stack. Calling a function looks the same way as executing a functional-style opcode.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function. There is no explicit `return` statement.

If you call a function that returns multiple values, you have to assign them to a tuple using `a, b := f(x)` or `let a, b := f(x)`.

The following example implements the power function by square-and-multiply.

```
{
    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) }
        }
    }
}
```

Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops, ifs and switches are converted by simple rewriting rules and after that, the only thing the assembler does for you is re-arranging functional-style opcodes, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached.

Conventions in Solidity

In contrast to EVM assembly, Solidity knows types which are narrower than 256 bits, e.g. `uint24`. In order to make them more efficient, most arithmetic operations just treat them as 256-bit numbers and the higher-order bits are only cleaned at the point where it is necessary, i.e. just shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher order bits first.

Solidity manages memory in a very simple way : There is a « free memory pointer » at position `0x40` in memory. If you want to allocate memory, just use the memory starting from where this pointer points at and update it accordingly. There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory. Here is an assembly snippet that can be used for allocating memory :

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

The first 64 bytes of memory can be used as « scratch space » for short-term allocation. The 32 bytes after the free memory pointer (i.e. starting at `0x60`) is meant to be zero permanently and is used as the initial value for empty dynamic memory arrays. This means that the allocatable memory starts at `0x80`, which is the initial value of the free memory pointer.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (yes, this is even true for `byte[]`, 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 : Statically-sized memory arrays do not have a length field, but it might be added later to allow better convertibility between statically- and dynamically-sized arrays, so please do not rely on that.

Standalone Assembly

The assembly language described as inline assembly above can also be used standalone and in fact, the plan is to use it as an intermediate language for the Solidity compiler. In this form, it tries to achieve several goals :

1. Programs written in it should be readable, even if the code is generated by a compiler from Solidity.
2. The translation from assembly to bytecode should contain as few « surprises » as possible.
3. Control flow should be easy to detect to help in formal verification and optimization.

In order to achieve the first and last goal, assembly provides high-level constructs like `for` loops, `if` and `switch` statements and function calls. It should be possible to write assembly programs that do not make use of explicit `SWAP`, `DUP`, `JUMP` and `JUMPI` statements, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form `mul(add(x, y), 7)` are preferred over pure opcode statements like `7 y x add mul` because in the first form, it is much easier to see which operand is used for which opcode.

The second goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, ...), which follow very simple and regular scoping rules and cleanup of local variables from the stack.

Scoping : An identifier that is declared (label, variable, function, assembly) is only visible in the block where it was declared (including nested blocks inside the current block). It is not legal to access local variables across function borders, even if they would be in scope. Shadowing is not allowed. Local variables cannot be accessed before they were declared, but functions and assemblies can. Assemblies are special blocks that are used for e.g. returning runtime code or creating contracts. No identifier from an outer assembly is visible in a sub-assembly.

If control flow passes over the end of a block, pop instructions are inserted that match the number of local variables declared in that block. Whenever a local variable is referenced, the code generator needs to know its current relative position in the stack and thus it needs to keep track of the current so-called stack height. Since all local variables are removed at the end of a block, the stack height before and after the block should be the same. If this is not the case, compilation fails.

Using `switch`, `for` and functions, it should be possible to write complex code without using `jump` or `jumpi` manually. This makes it much easier to analyze the control flow, which allows for improved formal verification and optimization.

Furthermore, if manual jumps are allowed, computing the stack height is rather complicated. The position of all local variables on the stack needs to be known, otherwise neither references to local variables nor removing local variables automatically from the stack at the end of a block will work properly.

Example :

We will follow an example compilation from Solidity to assembly. We consider the runtime bytecode of the following Solidity program :

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
  function f(uint x) public pure returns (uint y) {
    y = 1;
    for (uint i = 0; i < x; i++)
      y = 2 * y;
  }
}
```

The following assembly will be generated :

```
{
  mstore(0x40, 0x80) // store the "free memory pointer"
  // function dispatcher
  switch div(calldataload(0), exp(2, 226))
  case 0xb3de648b {
    let r := f(calldataload(4))
    let ret := $allocate(0x20)
    mstore(ret, r)
    return(ret, 0x20)
  }
  default { revert(0, 0) }
  // memory allocator
  function $allocate(size) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, size))
  }
  // the contract function
  function f(x) -> y {
    y := 1
    for { let i := 0 } lt(i, x) { i := add(i, 1) } {
      y := mul(2, y)
    }
  }
}
```

Assembly Grammar

The tasks of the parser are the following :

- Turn the byte stream into a token stream, discarding C++-style comments (a special comment exists for source references, but we will not explain it here).
- Turn the token stream into an AST according to the grammar below
- Register identifiers with the block they are defined in (annotation to the AST node) and note from which point on, variables can be accessed.

The assembly lexer follows the one defined by Solidity itself.

Whitespace is used to delimit tokens and it consists of the characters Space, Tab and Linefeed. Comments are regular JavaScript/C++ comments and are interpreted in the same way as Whitespace.

Grammar :

```
AssemblyBlock = '{' AssemblyItem* '}'
AssemblyItem =
    Identifier |
    AssemblyBlock |
    AssemblyExpression |
    AssemblyLocalDefinition |
    AssemblyAssignment |
    AssemblyStackAssignment |
    LabelDefinition |
    AssemblyIf |
    AssemblySwitch |
    AssemblyFunctionDefinition |
    AssemblyFor |
    'break' |
    'continue' |
    SubAssembly
```

(suite sur la page suivante)

(suite de la page précédente)

```
AssemblyExpression = AssemblyCall | Identifier | AssemblyLiteral
AssemblyLiteral = NumberLiteral | StringLiteral | HexLiteral
Identifier = [a-zA-Z_$] [a-zA-Z_0-9]*
AssemblyCall = Identifier '(' ( AssemblyExpression ( ',' AssemblyExpression )* )? ')'
AssemblyLocalDefinition = 'let' IdentifierOrList ( ':=' AssemblyExpression )?
AssemblyAssignment = IdentifierOrList ':=' AssemblyExpression
IdentifierOrList = Identifier | '(' IdentifierList ')'
IdentifierList = Identifier ( ',' Identifier)*
AssemblyStackAssignment = '=:' Identifier
LabelDefinition = Identifier ':'
AssemblyIf = 'if' AssemblyExpression AssemblyBlock
AssemblySwitch = 'switch' AssemblyExpression AssemblyCase*
    ( 'default' AssemblyBlock )?
AssemblyCase = 'case' AssemblyExpression AssemblyBlock
AssemblyFunctionDefinition = 'function' Identifier '(' IdentifierList? ')'
    ( '->' '(' IdentifierList ')' )? AssemblyBlock
AssemblyFor = 'for' ( AssemblyBlock | AssemblyExpression )
    AssemblyExpression ( AssemblyBlock | AssemblyExpression ) AssemblyBlock
SubAssembly = 'assembly' Identifier AssemblyBlock
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

4.4.8 Miscellaneous

Layout of State Variables in Storage

Statically-sized variables (everything except mapping and dynamically-sized array types) are laid out contiguously in storage starting from position `0`. Multiple 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.
- Elementary types use only that many bytes that are necessary to store them.
- If an elementary type does not fit the remaining part of a storage slot, it is moved to the next storage slot.
- Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

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 is only beneficial to use reduced-size arguments 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. 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.

The elements of structs and arrays are stored after each other, just as if they were given explicitly.

Mappings and Dynamic Arrays

Due to their unpredictable size, mapping and dynamically-sized array types use a Keccak-256 hash computation to find the starting position of the value or the array data. These starting positions are always full stack slots.

The mapping or the dynamic array itself occupies a slot in storage at some position `p` according to the above rule (or by recursively applying this rule for mappings of mappings or arrays of arrays). 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 is unused (but it is needed so that two equal mappings after each other will use a different hash distribution). Array data is located at `keccak256(p)` and the value corresponding to a mapping key `k` is located at `keccak256(k . p)` where `.` is concatenation. If the value is again a non-elementary type, the positions are found by adding an offset of `keccak256(k . p)`.

So for the following contract snippet :

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
  struct s { uint a; uint b; }
  uint x;
  mapping(uint => mapping(uint => s)) data;
}
```

The position of `data[4][9].b` is at `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`.

`bytes` and `string`

`bytes` and `string` are encoded identically. For short byte arrays, they store their data in the same slot where the length is also stored. In particular : if the data is at most `31` bytes long, it is stored in the higher-order bytes (left aligned) and the lowest-order byte stores `length * 2`. For byte arrays that store data which is `32` or more bytes long, the main slot stores `length * 2 + 1` and the data is stored as usual in `keccak256(slot)`. 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.

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).

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 shouldn't 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 adverse results.

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.

Internals - 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 the 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.

On the other hand, 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

Internals - The Optimizer

The Solidity optimizer operates on assembly, so it can be and also is used by other languages. It splits the sequence of instructions into basic blocks at `JUMPs` and `JUMPDESTs`. Inside these blocks, the instructions are analysed and every modification to the stack, to memory or storage is recorded as an expression which consists of an instruction and a list of arguments which are essentially pointers to other expressions. The main idea is now to find expressions that are always equal (on every input) and combine them into an expression class. The optimizer first tries to find each new expression in a list of already known expressions. If this does not work, the expression is simplified according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is done recursively, we can also apply the latter rule if the second factor is a more complex expression where we know that it will always evaluate to one. Modifications to storage and memory locations have to erase knowledge about storage and memory locations which are not known to be different : If we first write to location x and then to location y and both are input variables, the second could overwrite the first, so we actually do not know what is stored at x after we wrote to y. On

the other hand, if a simplification of the expression x - y evaluates to a non-zero constant, we know that we can keep our knowledge about what is stored at x.

At the end of this process, we know which expressions have to be on the stack in 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 a `JUMPI` is found whose condition evaluates to a constant, it is transformed to an unconditional jump.

As the last step, the code in each block is completely re-generated. A dependency graph is created from the expressions on the stack at the end of the block and every operation that is not part of this graph is essentially dropped. Now code is generated 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) and finally, 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 depending on the value of the constant, and thus code like

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2)
  return 2;
else
  return 1;
```

is simplified to code which can also be compiled from

```
data[7] = 9;
return 1;
```

even though the instructions contained a jump in the beginning.

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.

Both kinds of source mappings use integer identifiers to refer to source files. These are regular array indices into a list of source files usually called `"sourceList"`, which is part of the combined-json and the output of the json / npm compiler.

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` 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 and `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.

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;;
```

Tips and Tricks

- Use `delete` on arrays to delete all its elements.
- Use shorter types for struct elements and sort them such that short types are grouped together. This can lower the gas costs as multiple `SSTORE` operations might be combined into a single (`SSTORE` costs 5000 or 20000 gas, so this is what you want to optimise). Use the gas price estimator (with optimiser enabled) to check !
- Make your state variables public - the compiler will create *getters* for you automatically.
- If you end up checking conditions on input or state a lot at the beginning of your functions, try using *Function Modifiers*.
- Initialize storage structs with a single assignment : `x = MyStruct({a: 1, b: 2});`

Note : If the storage struct has tightly packed properties, initialize it with separate assignments : `x.a = 1; x.b = 2;`. In this way it will be easier for the optimizer to update storage in one go, thus making assignment cheaper.

Cheatsheet

Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	`++`, `--`
	New expression	`new <typename>`
	Array subscripting	`<array>[<index>]`
	Member access	`<object>.<member>`
	Function-like call	`<func>(<args...>)`
	Parentheses	`(<statement>)`
2	Prefix increment and decrement	`++`, `--`
	Unary minus	`-`
	Unary operations	`delete`
	Logical NOT	`!`
	Bitwise NOT	`~`
3	Exponentiation	`**`
4	Multiplication, division and modulo	`*`, `/`, `%`
5	Addition and subtraction	`+`, `-`
6	Bitwise shift operators	`<<`, `>>`
7	Bitwise AND	`&`
8	Bitwise XOR	`^`
9	Bitwise OR	`|`
10	Inequality operators	`<`, `>`, `<=`, `>=`
11	Equality operators	`==`, `!=`
12	Logical AND	`&&`
13	Logical OR	`||`
14	Ternary operator	`<conditional> ? <if-true> : <if-false>`
15	Assignment operators	`=`, `|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=`
16	Comma operator	`,`

Global Variables

— `abi.decode(bytes memory encodedData, (...)) returns (...)` : *ABI*-decodes the provided data. The types are given in parentheses as second argument. Example : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
— `abi.encode(...) returns (bytes memory)` : *ABI*-encodes the given arguments
— `abi.encodePacked(...) returns (bytes memory)` : Performs *packed encoding* of the given arguments
— **`abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` : *ABI*-encodes the given a** starting from the second and prepends the given four-byte selector
— `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)` : Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...)` `
— `block.coinbase` (`address payable`) : current block miner's address
— `block.difficulty` (`uint`) : current block difficulty
— `block.gaslimit` (`uint`) : current block gaslimit
— `block.number` (`uint`) : current block number
— `block.timestamp` (`uint`) : current block timestamp
— `gasleft() returns (uint256)` : remaining gas
— `msg.data` (`bytes`) : complete calldata
— `msg.sender` (`address payable`) : sender of the message (current call)
— `msg.value` (`uint`) : number of wei sent with the message
— `now` (`uint`) : current block timestamp (alias for `block.timestamp`)
— `tx.gasprice` (`uint`) : gas price of the transaction

- `tx.origin` (`address payable`) : sender of the transaction (full call chain)
- `assert(bool condition)` : abort execution and revert state changes if condition is `false` (use for internal error)
- `require(bool condition)` : abort execution and revert state changes if condition is `false` (use for malformed input or error in external component)
- `require(bool condition, string memory message)` : abort execution and revert state changes if condition is `false` (use for malformed input or error in external component). Also provide error message.
- `revert()` : abort execution and revert state changes
- `revert(string memory message)` : abort execution and revert state changes providing an explanatory string
- `blockhash(uint blockNumber) returns (bytes32)` : hash of the given block - only works for 256 most recent blocks
- `keccak256(bytes memory) returns (bytes32)` : compute the Keccak-256 hash of the input
- `sha256(bytes memory) returns (bytes32)` : compute the SHA-256 hash of the input
- `ripemd160(bytes memory) returns (bytes20)` : compute the RIPEMD-160 hash of the input
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)` : recover address associated with the public key from elliptic curve signature, return zero on error
- `addmod(uint x, uint y, uint k) returns (uint)` : compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `mulmod(uint x, uint y, uint k) returns (uint)` : compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `this` (current contract's type) : the current contract, explicitly convertible to `address` or `address payable`
- `super` : the contract one level higher in the inheritance hierarchy
- `selfdestruct(address payable recipient)` : destroy the current contract, sending its funds to the given address
- `<address>.balance` (`uint256`) : balance of the *Adresses* in Wei
- `<address payable>.send(uint256 amount) returns (bool)` : send given amount of Wei to *Adresses*, returns `false` on failure
- `<address payable>.transfer(uint256 amount)` : send given amount of Wei to *Adresses*, throws on failure

---

Note : Do not rely on `block.timestamp`, `now` and `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

Note : The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Note : In version 0.5.0, the following aliases were removed : `suicide` as alias for `selfdestruct`, `msg.gas` as alias for `gasleft`, `block.blockhash` as alias for `blockhash` and `sha3` as alias for `keccak256`.

Function Visibility Specifiers

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

- `public` : visible externally and internally (creates a *getter function* for storage/state variables)
- `private` : only visible in the current contract
- `external` : only visible externally (only for functions) - i.e. can only be message-called (via `this.func`)
- `internal` : only visible internally

Modifiers

- `pure` for functions : Disallows modification or access of state.
- `view` for functions : Disallows modification of state.
- `payable` for functions : Allows them to receive Ether together with a call.
- `constant` for state variables : Disallows assignment (except initialisation), does not occupy storage slot.
- `anonymous` for events : Does not store event signature as topic.
- `indexed` for event parameters : Stores the parameter as topic.

Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future :

`abstract`, `after`, `alias`, `apply`, `auto`, `case`, `catch`, `copyof`, `default`, `define`, `final`, `immutable`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `override`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `try`, `type`, `typedef`, `typeof`, `unchecked`.

Language Grammar

```
SourceUnit = (PragmaDirective | ImportDirective | ContractDefinition)*

// Pragma actually parses anything up to the trailing ';' to be fully forward-
↪compatible.
PragmaDirective = 'pragma' Identifier ([^;]+) ';'

ImportDirective = 'import' StringLiteral ('as' Identifier)? ';'
        | 'import' ('*' | Identifier) ('as' Identifier)? 'from' StringLiteral ';'
        | 'import' '{' Identifier ('as' Identifier)? ( ',' Identifier ('as'␣
↪Identifier)? )* '}' 'from' StringLiteral ';'

ContractDefinition = ( 'contract' | 'library' | 'interface' ) Identifier
                     ( 'is' InheritanceSpecifier (',' InheritanceSpecifier )* )?
                     '{' ContractPart* '}'

ContractPart = StateVariableDeclaration | UsingForDeclaration
             | StructDefinition | ModifierDefinition | FunctionDefinition |␣
↪EventDefinition | EnumDefinition

InheritanceSpecifier = UserDefinedTypeName ( '(' Expression ( ',' Expression )* ')' )?
```

(suite sur la page suivante)

(suite de la page précédente)

```
StateVariableDeclaration = TypeName ( 'public' | 'internal' | 'private' | 'constant'␣
↪)* Identifier ('=' Expression)? ';'
UsingForDeclaration = 'using' Identifier 'for' ('*' | TypeName) ';'
StructDefinition = 'struct' Identifier '{'
                    ( VariableDeclaration ';' (VariableDeclaration ';')* ) '}'

ModifierDefinition = 'modifier' Identifier ParameterList? Block
ModifierInvocation = Identifier ( '(' ExpressionList? ')' )?

FunctionDefinition = 'function' Identifier? ParameterList
                    ( ModifierInvocation | StateMutability | 'external' | 'public' |
↪'internal' | 'private' )*
                    ( 'returns' ParameterList )? ( ';' | Block )
EventDefinition = 'event' Identifier EventParameterList 'anonymous'? ';'

EnumValue = Identifier
EnumDefinition = 'enum' Identifier '{' EnumValue? (',' EnumValue)* '}'

ParameterList = '(' ( Parameter (',' Parameter)* )? ')'
Parameter = TypeName StorageLocation? Identifier?

EventParameterList = '(' ( EventParameter (',' EventParameter )* )? ')'
EventParameter = TypeName 'indexed'? Identifier?

FunctionTypeParameterList = '(' ( FunctionTypeParameter (',' FunctionTypeParameter )*␣
↪)? ')'
FunctionTypeParameter = TypeName StorageLocation?

// semantic restriction: mappings and structs (recursively) containing mappings
// are not allowed in argument lists
VariableDeclaration = TypeName StorageLocation? Identifier

TypeName = ElementaryTypeName
         | UserDefinedTypeName
         | Mapping
         | ArrayTypeName
         | FunctionTypeName
         | ( 'address' 'payable' )

UserDefinedTypeName = Identifier ( '.' Identifier )*

Mapping = 'mapping' '(' ElementaryTypeName '=>' TypeName ')'
ArrayTypeName = TypeName '[' Expression? ']'
FunctionTypeName = 'function' FunctionTypeParameterList ( 'internal' | 'external' |␣
↪StateMutability )*
                   ( 'returns' FunctionTypeParameterList )?
StorageLocation = 'memory' | 'storage' | 'calldata'
StateMutability = 'pure' | 'view' | 'payable'

Block = '{' Statement* '}'
Statement = IfStatement | WhileStatement | ForStatement | Block |␣
↪InlineAssemblyStatement |
            ( DoWhileStatement | PlaceholderStatement | Continue | Break | Return |
              Throw | EmitStatement | SimpleStatement ) ';'

ExpressionStatement = Expression
IfStatement = 'if' '(' Expression ')' Statement ( 'else' Statement )?
```

(suite sur la page suivante)

(suite de la page précédente)

```
WhileStatement = 'while' '(' Expression ')' Statement
PlaceholderStatement = '_'
SimpleStatement = VariableDefinition | ExpressionStatement
ForStatement = 'for' '(' (SimpleStatement)? ';' (Expression)? ';'␣
↪(ExpressionStatement)? ')' Statement
InlineAssemblyStatement = 'assembly' StringLiteral? InlineAssemblyBlock
DoWhileStatement = 'do' Statement 'while' '(' Expression ')'
Continue = 'continue'
Break = 'break'
Return = 'return' Expression?
Throw = 'throw'
EmitStatement = 'emit' FunctionCall
VariableDefinition = (VariableDeclaration | '(' VariableDeclaration? (','␣
↪VariableDeclaration? )* ')' ) ( '=' Expression )?

// Precedence by order (see github.com/ethereum/solidity/pull/732)
Expression
  = Expression ('++' | '--')
  | NewExpression
  | IndexAccess
  | MemberAccess
  | FunctionCall
  | '(' Expression ')'
  | ('!' | '~' | 'delete' | '++' | '--' | '+' | '-') Expression
  | Expression '**' Expression
  | Expression ('*' | '/' | '%') Expression
  | Expression ('+' | '-') Expression
  | Expression ('<<' | '>>') Expression
  | Expression '&' Expression
  | Expression '^' Expression
  | Expression '|' Expression
  | Expression ('<' | '>' | '<=' | '>=') Expression
  | Expression ('==' | '!=') Expression
  | Expression '&&' Expression
  | Expression '||' Expression
  | Expression '?' Expression ':' Expression
  | Expression ('=' | '|=' | '^=' | '&=' | '<<=' | '>>=' | '+=' | '-=' | '*=' | '/=' ␣
↪| '%=') Expression
  | PrimaryExpression

PrimaryExpression = BooleanLiteral
                  | NumberLiteral
                  | HexLiteral
                  | StringLiteral
                  | TupleExpression
                  | Identifier
                  | ElementaryTypeNameExpression

ExpressionList = Expression ( ',' Expression )*
NameValueList = Identifier ':' Expression ( ',' Identifier ':' Expression )*

FunctionCall = Expression '(' FunctionCallArguments ')'
FunctionCallArguments = '{' NameValueList? '}'
                      | ExpressionList?

NewExpression = 'new' TypeName
MemberAccess = Expression '.' Identifier
```

(suite sur la page suivante)

(suite de la page précédente)

```
IndexAccess = Expression '[' Expression? ']'

BooleanLiteral = 'true' | 'false'
NumberLiteral = ( HexNumber | DecimalNumber ) (' ' NumberUnit)?
NumberUnit = 'wei' | 'szabo' | 'finney' | 'ether'
           | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years'
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*

HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+ ( '.' [0-9]* )? ( [eE] [0-9]+ )?

TupleExpression = '(' ( Expression? ( ',' Expression? )*  )? ')'
                | '[' ( Expression  ( ',' Expression  )*  )? ']'

ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | Int | Uint | Byte | Fixed |␣
↪Ufixed

Int = 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' |
↪'int64' | 'int72' | 'int80' | 'int88' | 'int96' | 'int104' | 'int112' | 'int120' |
↪'int128' | 'int136' | 'int144' | 'int152' | 'int160' | 'int168' | 'int176' | 'int184
↪' | 'int192' | 'int200' | 'int208' | 'int216' | 'int224' | 'int232' | 'int240' |
↪'int248' | 'int256'

Uint = 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' |
↪'uint56' | 'uint64' | 'uint72' | 'uint80' | 'uint88' | 'uint96' | 'uint104' |
↪'uint112' | 'uint120' | 'uint128' | 'uint136' | 'uint144' | 'uint152' | 'uint160' |
↪'uint168' | 'uint176' | 'uint184' | 'uint192' | 'uint200' | 'uint208' | 'uint216' |
↪'uint224' | 'uint232' | 'uint240' | 'uint248' | 'uint256'

Byte = 'byte' | 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' |
↪'bytes6' | 'bytes7' | 'bytes8' | 'bytes9' | 'bytes10' | 'bytes11' | 'bytes12' |
↪'bytes13' | 'bytes14' | 'bytes15' | 'bytes16' | 'bytes17' | 'bytes18' | 'bytes19' |
↪'bytes20' | 'bytes21' | 'bytes22' | 'bytes23' | 'bytes24' | 'bytes25' | 'bytes26' |
↪'bytes27' | 'bytes28' | 'bytes29' | 'bytes30' | 'bytes31' | 'bytes32'

Fixed = 'fixed' | ( 'fixed' [0-9]+ 'x' [0-9]+ )

Ufixed = 'ufixed' | ( 'ufixed' [0-9]+ 'x' [0-9]+ )

InlineAssemblyBlock = '{' AssemblyItem* '}'

AssemblyItem = Identifier | FunctionalAssemblyExpression | InlineAssemblyBlock |␣
↪AssemblyVariableDeclaration | AssemblyAssignment | AssemblyLabel | NumberLiteral |␣
↪StringLiteral | HexLiteral
AssemblyExpression = Identifier | FunctionalAssemblyExpression | NumberLiteral |␣
↪StringLiteral | HexLiteral
AssemblyVariableDeclaration = 'let' Identifier ':=' AssemblyExpression
AssemblyAssignment = ( Identifier ':=' AssemblyExpression ) | ( '=:' Identifier )
AssemblyLabel = Identifier ':'
FunctionalAssemblyExpression = Identifier '(' AssemblyItem? ( ',' AssemblyItem )* ')'
```

4.4.9 Solidity v0.5.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.5.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

Note : Contracts compiled with Solidity v0.5.0 can still interface with contracts and even libraries compiled with older versions without recompiling or redeploying them. Changing the interfaces to include data locations and visibility and mutability specifiers suffices.

Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- Signed right shift now uses proper arithmetic shift, i.e. rounding towards negative infinity, instead of rounding towards zero. Signed and unsigned shift will have dedicated opcodes in Constantinople, and are emulated by Solidity for the moment.
- The `continue` statement in a `do...while` loop now jumps to the condition, which is the common behavior in such cases. It used to jump to the loop body. Thus, if the condition is false, the loop terminates.
- The functions `.call()`, `.delegatecall()` and `.staticcall()` do not pad anymore when given a single `bytes` parameter.
- Pure and view functions are now called using the opcode `STATICCALL` instead of `CALL` if the EVM version is Byzantium or later. This disallows state changes on the EVM level.
- The ABI encoder now properly pads byte arrays and strings from calldata (`msg.data` and external function parameters) when used in external function calls and in `abi.encode`. For unpadded encoding, use `abi.encodePacked`.
- The ABI decoder reverts in the beginning of functions and in `abi.decode()` if passed calldata is too short or points out of bounds. Note that dirty higher order bits are still simply ignored.
- Forward all available gas with external function calls starting from Tangerine Whistle.

Semantic and Syntactic Changes

This section highlights changes that affect syntax and semantics.

- The functions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` and `ripemd160()` now accept only a single `bytes` argument. Moreover, the argument is not padded. This was changed to make more explicit and clear how the arguments are concatenated. Change every `.call()` (and family) to a `.call("")` and every `.call(signature, a, b, c)` to use `.call(abi.encodeWithSignature(signature, a, b, c))` (the last one only works for value types). Change every `keccak256(a, b, c)` to `keccak256(abi.encodePacked(a, b, c))`. Even though it is not a breaking change, it is suggested that developers change `x.call(bytes4(keccak256("f(uint256)"), a, b)` to `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.
- Functions `.call()`, `.delegatecall()` and `.staticcall()` now return `(bool, bytes memory)` to provide access to the return data. Change `bool success = otherContract.call("f")` to `(bool success, bytes memory data) = otherContract.call("f")`.
- Solidity now implements C99-style scoping rules for function local variables, that is, variables can only be used after they have been declared and only in the same or nested scopes. Variables declared in the initialization block of a `for` loop are valid at any point inside the loop.

Explicitness Requirements

This section lists changes where the code now needs to be more explicit. For most of the topics the compiler will provide suggestions.

- Explicit function visibility is now mandatory. Add `public` to every function and constructor, and `external` to every fallback or interface function that does not specify its visibility already.
- Explicit data location for all variables of struct, array or mapping types is now mandatory. This is also applied to function parameters and return variables. For example, change `uint[] x = m_x` to `uint[] storage x = m_x`, and `function f(uint[][] x)` to `function f(uint[][] memory x)` where `memory` is the data location and might be replaced by `storage` or `calldata` accordingly. Note that `external` functions require parameters with a data location of `calldata`.
- Contract types do not include `address` members anymore in order to separate the namespaces. Therefore, it is now necessary to explicitly convert values of contract type to addresses before using an `address` member. Example : if `c` is a contract, change `c.transfer(...)` to `address(c).transfer(...)`, and `c.balance` to `address(c).balance`.
- The `address` type was split into `address` and `address payable`, where only `address payable` provides the `transfer` function. An `address payable` can be directly converted to an `address`, but the other way around is not allowed. Converting `address` to `address payable` is possible via conversion through `uint160`. If `c` is a contract, `address(c)` results in `address payable` only if `c` has a payable fallback function. If you use the *withdraw pattern*, you most likely do not have to change your code because `transfer` is only used on `msg.sender` instead of stored addresses and `msg.sender` is an `address payable`.
- Conversions between `bytesX` and `uintY` of different size are now disallowed due to `bytesX` padding on the right and `uintY` padding on the left which may cause unexpected conversion results. The size must now be adjusted within the type before the conversion. For example, you can convert a `bytes4` (4 bytes) to a `uint64` (8 bytes) by first converting the `bytes4` variable to `bytes8` and then to `uint64`. You get the opposite padding when converting through `uint32`.
- Using `msg.value` in non-payable functions (or introducing it via a modifier) is disallowed as a security feature. Turn the function into `payable` or create a new internal function for the program logic that uses `msg.value`.
- For clarity reasons, the command line interface now requires `-` if the standard input is used as source.

Deprecated Elements

This section lists changes that deprecate prior features or syntax. Note that many of these changes were already enabled in the experimental mode `v0.5.0`.

Command Line and JSON Interfaces

- The command line option `--formal` (used to generate Why3 output for further formal verification) was deprecated and is now removed. A new formal verification module, the SMTChecker, is enabled via `pragma experimental SMTChecker;`.
- The command line option `--julia` was renamed to `--yul` due to the renaming of the intermediate language `Julia` to `Yul`.
- The `--clone-bin` and `--combined-json clone-bin` command line options were removed.
- Remappings with empty prefix are disallowed.
- The JSON AST fields `constant` and `payable` were removed. The information is now present in the `stateMutability` field.
- The JSON AST field `isConstructor` of the `FunctionDefinition` node was replaced by a field called `kind` which can have the value `"constructor"`, `"fallback"` or `"function"`.

Constructors

— Constructors must now be defined using the `constructor` keyword.
— Calling base constructors without parentheses is now disallowed.
— Specifying base constructor arguments multiple times in the same inheritance hierarchy is now disallowed.
— Calling a constructor with arguments but with wrong argument count is now disallowed. If you only want to specify an inheritance relation without giving arguments, do not provide parentheses at all.

Functions

— Function `callcode` is now disallowed (in favor of `delegatecall`). It is still possible to use it via inline assembly.
— `suicide` is now disallowed (in favor of `selfdestruct`).
— `sha3` is now disallowed (in favor of `keccak256`).
— `throw` is now disallowed (in favor of `revert`, `require` and `assert`).

Conversions

— Explicit and implicit conversions from decimal literals to `bytesXX` types is now disallowed.
— Explicit and implicit conversions from hex literals to `bytesXX` types of different size is now disallowed.

Literals and Suffixes

— The unit denomination `years` is now disallowed due to complications and confusions about leap years.
— Trailing dots that are not followed by a number are now disallowed.
— Combining hex numbers with unit denominations (e.g. `0x1e wei`) is now disallowed.
— The prefix `0X` for hex numbers is disallowed, only `0x` is possible.

Variables

— Declaring empty structs is now disallowed for clarity.
— The `var` keyword is now disallowed to favor explicitness.
— Assignments between tuples with different number of components is now disallowed.
— Values for constants that are not compile-time constants are disallowed.
— Multi-variable declarations with mismatching number of values are now disallowed.
— Uninitialized storage variables are now disallowed.
— Empty tuple components are now disallowed.
— Detecting cyclic dependencies in variables and structs is limited in recursion to 256.
— Fixed-size arrays with a length of zero are now disallowed.

Syntax

— Using `constant` as function state mutability modifier is now disallowed.
— Boolean expressions cannot use arithmetic operations.
— The unary + operator is now disallowed.
— Literals cannot anymore be used with `abi.encodePacked` without prior conversion to an explicit type.
— Empty return statements for functions with one or more return values are now disallowed.
— The « loose assembly » syntax is now disallowed entirely, that is, jump labels, jumps and non-functional instructions cannot be used anymore. Use the new `while`, `switch` and `if` constructs instead.

— Functions without implementation cannot use modifiers anymore.
— Function types with named return values are now disallowed.
— Single statement variable declarations inside if/while/for bodies that are not blocks are now disallowed.
— New keywords : `calldata` and `constructor`.
— New reserved keywords : `alias`, `apply`, `auto`, `copyof`, `define`, `immutable`, `implements`, `macro`, `mutable`, `override`, `partial`, `promise`, `reference`, `sealed`, `sizeof`, `supports`, `typedef` and `unchecked`.

Example

The following example shows a contract and its updated version for Solidity v0.5.0 with some of the changes listed in this section.

Old version :

```
// This will not compile
pragma solidity ^0.4.25;

contract OtherContract {
   uint x;
   function f(uint y) external {
      x = y;
   }
   function() payable external {}
}

contract Old {
   OtherContract other;
   uint myNumber;

   // Function mutability not provided, not an error.
   function someInteger() internal returns (uint) { return 2; }

   // Function visibility not provided, not an error.
   // Function mutability not provided, not an error.
   function f(uint x) returns (bytes) {
      // Var is fine in this version.
      var z = someInteger();
      x += z;
      // Throw is fine in this version.
      if (x > 100)
         throw;
      bytes 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }
        return b;
    }

    // No need for an explicit data location for 'arr'
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
        otherContract.transfer(1 ether);

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the first 4 bytes of x will be lost. This might lead to
        // unexpected behavior since bytesX are right padded.
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

New version :

```
pragma solidity >0.4.99 <0.6.0;

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Function mutability must be specified.
    function someInteger() internal pure returns (uint) { return 2; }

    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
        // The type must now be explicitly given.
        uint z = someInteger();
        x += z;
        // Throw is now disallowed.
        require(x > 100);
        int y = -3 >> 1;
        // y == -2 (correct)
        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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    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
        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));
    }
}
```

4.5 Security Considerations

While it is usually quite easy to build software that works as expected, it is much harder to check that nobody can use it in a way that was **not** anticipated.

In Solidity, this is even more important because you can use smart contracts to handle tokens or, possibly, even more valuable things. Furthermore, every execution of a smart contract happens in public and, in addition to that, the source code is often available.

Of course you always have to consider how much is at stake : You can compare a smart contract with a web service that is open to the public (and thus, also to malicious actors) and perhaps even open source. If you only store your grocery list on that web service, you might not have to take too much care, but if you manage your bank account using that web service, you should be more careful.

This section will list some pitfalls and general security recommendations but can, of course, never be complete. Also, keep in mind that even if your smart contract code is bug-free, the compiler or the platform itself might have a bug. A list of some publicly known security-relevant bugs of the compiler can be found in the *list of known bugs*, which is also machine-readable. Note that there is a bug bounty program that covers the code generator of the Solidity compiler.

As always, with open source documentation, please help us extend this section (especially, some examples would not hurt) !

4.5.1 Pitfalls

Private Information and Randomness

Everything you use in a smart contract is publicly visible, even local variables and state variables marked `private`.

Using random numbers in smart contracts is quite tricky if you do not want miners to be able to cheat.

Re-Entrancy

Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed. To give an example, the following code contains a bug (it is just a snippet and not a complete contract) :

```
pragma solidity >=0.4.0 <0.6.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        if (msg.sender.send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

The problem is not too serious here because of the limited gas as part of `send`, but it still exposes a weakness : Ether transfer can always include code execution, so the recipient could be a contract that calls back into `withdraw`. This would let it get multiple refunds and basically retrieve all the Ether in the contract. In particular, the following contract will allow an attacker to refund multiple times as it uses `call` which forwards all remaining gas by default :

```
pragma solidity >=0.4.0 <0.6.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        (bool success,) = msg.sender.call.value(shares[msg.sender])("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

To avoid re-entrancy, you can use the Checks-Effects-Interactions pattern as outlined further below :

```
pragma solidity >=0.4.11 <0.6.0;

contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        msg.sender.transfer(share);
    }
}
```

Note that re-entrancy is not only an effect of Ether transfer but of any function call on another contract. Furthermore, you also have to take multi-contract situations into account. A called contract could modify the state of another contract you depend on.

Gas Limit and Loops

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully : Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. This may not apply to `view` functions that are only executed to read data from the blockchain. Still, such functions may be called by other contracts as part of on-chain operations and stall those. Please be explicit about such cases in the documentation of your contracts.

Sending and Receiving Ether

— Neither contracts nor « external accounts » are currently able to prevent that someone sends them Ether. Contracts can react on and reject a regular transfer, but there are ways to move Ether without creating a message call. One way is to simply « mine to » the contract address and the second way is using `selfdestruct(x)`.
— If a contract receives Ether (without a function being called), the fallback function is executed. If it does not have a fallback function, the Ether will be rejected (by throwing an exception). During the execution of the fallback function, the contract can only rely on the « gas stipend » it is passed (2300 gas) being available to it at that time. This stipend is not enough to modify storage (do not take this for granted though, the stipend might change with future hard forks). To be sure that your contract can receive Ether in that way, check the gas requirements of the fallback function (for example in the « details » section in Remix).
— There is a way to forward more gas to the receiving contract using `addr.call.value(x)("")`. This is essentially the same as `addr.transfer(x)`, only that it forwards all remaining gas and opens up the ability for the recipient to perform more expensive actions (and it returns a failure code instead of automatically propagating the error). This might include calling back into the sending contract or other state changes you might not have thought of. So it allows for great flexibility for honest users but also for malicious actors.
— If you want to send Ether using `address.transfer`, there are certain details to be aware of :

1. If the recipient is a contract, it causes its fallback function to be executed which can, in turn, call back the sending contract.
2. Sending Ether can fail due to the call depth going above 1024. Since the caller is in total control of the call depth, they can force the transfer to fail ; take this possibility into account or use `send` and make sure to always check its return value. Better yet, write your contract using a pattern where the recipient can withdraw Ether instead.
3. Sending Ether can also fail because the execution of the recipient contract requires more than the allotted amount of gas (explicitly by using `require`, `assert`, `revert`, `throw` or because the operation is just too expensive) - it « runs out of gas » (OOG). If you use `transfer` or `send` with a return value check, this

might provide a means for the recipient to block progress in the sending contract. Again, the best practice here is to use a *« withdraw » pattern instead of a « send » pattern*.

Callstack Depth

External function calls can fail any time because they exceed the maximum call stack of 1024. In such situations, Solidity throws an exception. Malicious actors might be able to force the call stack to a high value before they interact with your contract.

Note that `.send()` does **not** throw an exception if the call stack is depleted but rather returns `false` in that case. The low-level functions `.call()`, `.callcode()`, `.delegatecall()` and `.staticcall()` behave in the same way.

tx.origin

Never use tx.origin for authorization. Let's say you have a wallet contract like this :

```
pragma solidity >0.4.99 <0.6.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint amount) public {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Now someone tricks you into sending ether to the address of this attack wallet :

```
pragma solidity >0.4.99 <0.6.0;

interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() public {
        owner = msg.sender;
    }

    function() external {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

If your wallet had checked `msg.sender` for authorization, it would get the address of the attack wallet, instead of the owner address. But by checking `tx.origin`, it gets the original address that kicked off the transaction, which is still the owner address. The attack wallet instantly drains all your funds.

Two's Complement / Underflows / Overflows

As in many programming languages, Solidity's integer types are not actually integers. They resemble integers when the values are small, but behave differently if the numbers are larger. For example, the following is true : `uint8(255) + uint8(1) == 0`. This situation is called an *overflow*. It occurs when an operation is performed that requires a fixed size variable to store a number (or piece of data) that is outside the range of the variable's data type. An *underflow* is the converse situation : `uint8(0) - uint8(1) == 255`.

In general, read about the limits of two's complement representation, which even has some more special edge cases for signed numbers.

Try to use `require` to limit the size of inputs to a reasonable range and use the *SMT checker* to find potential overflows, or use a library like *SafeMath<https ://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol>* if you want all overflows to cause a revert.

Minor Details

— Types that do not occupy the full 32 bytes might contain « dirty higher order bits ». This is especially important if you access `msg.data` - it poses a malleability risk : You can craft transactions that call a function `f(uint8 x)` with a raw byte argument of `0xff000001` and with `0x00000001`. Both are fed to the contract and both will look like the number `1` as far as `x` is concerned, but `msg.data` will be different, so if you use `keccak256(msg.data)` for anything, you will get different results.

4.5.2 Recommendations

Take Warnings Seriously

If the compiler warns you about something, you should better change it. Even if you do not think that this particular warning has security implications, there might be another issue buried beneath it. Any compiler warning we issue can be silenced by slight changes to the code.

Always use the latest version of the compiler to be notified about all recently introduced warnings.

Restrict the Amount of Ether

Restrict the amount of Ether (or other tokens) that can be stored in a smart contract. If your source code, the compiler or the platform has a bug, these funds may be lost. If you want to limit your loss, limit the amount of Ether.

Keep it Small and Modular

Keep your contracts small and easily understandable. Single out unrelated functionality in other contracts or into libraries. General recommendations about source code quality of course apply : Limit the amount of local variables, the length of functions and so on. Document your functions so that others can see what your intention was and whether it is different than what the code does.

Use the Checks-Effects-Interactions Pattern

Most functions will first perform some checks (who called the function, are the arguments in range, did they send enough Ether, does the person have tokens, etc.). These checks should be done first.

As the second step, if all checks passed, effects to the state variables of the current contract should be made. Interaction with other contracts should be the very last step in any function.

Early contracts delayed some effects and waited for external function calls to return in a non-error state. This is often a serious mistake because of the re-entrancy problem explained above.

Note that, also, calls to known contracts might in turn cause calls to unknown contracts, so it is probably better to just always apply this pattern.

Include a Fail-Safe Mode

While making your system fully decentralised will remove any intermediary, it might be a good idea, especially for new code, to include some kind of fail-safe mechanism :

You can add a function in your smart contract that performs some self-checks like « Has any Ether leaked ? », « Is the sum of the tokens equal to the balance of the contract ? » or similar things. Keep in mind that you cannot use too much gas for that, so help through off-chain computations might be needed there.

If the self-check fails, the contract automatically switches into some kind of « failsafe » mode, which, for example, disables most of the features, hands over control to a fixed and trusted third party or just converts the contract into a simple « give me back my money » contract.

Ask for Peer Review

The more people examine a piece of code, the more issues are found. Asking people to review your code also helps as a cross-check to find out whether your code is easy to understand - a very important criterion for good smart contracts.

4.5.3 Formal Verification

Using formal verification, it is possible to perform an automated mathematical proof that your source code fulfills a certain formal specification. The specification is still formal (just as the source code), but usually much simpler.

Note that formal verification itself can only help you understand the difference between what you did (the specification) and how you did it (the actual implementation). You still need to check whether the specification is what you wanted and that you did not miss any unintended effects of it.

4.6 Using the compiler

4.6.1 Using the Commandline Compiler

Note : This section does not apply to *solcjs*, not even if it is used in commandline mode.

One of the build targets of the Solidity repository is `solc`, the solidity commandline compiler. Using `solc --help` provides you with an explanation of all options. The compiler can produce various outputs, ranging from simple binaries and assembly over an abstract syntax tree (parse tree) to estimations of gas usage. If you only want to compile a single file, you run it as `solc --bin sourceFile.sol` and it will print the binary. If you want to get some of the more advanced output variants of `solc`, it is probably better to tell it to output everything to separate files using `solc -o outputDirectory --bin --ast --asm sourceFile.sol`.

Before you deploy your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime. If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--runs=1`. If you expect many transactions and do not care for higher deployment cost and output size, set `--runs` to a high number.

The commandline compiler will automatically read imported files from the filesystem, but it is also possible to provide path redirects using `prefix=path` in the following way :

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

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`. `solc` will not read files from the filesystem that lie outside of the remapping targets and outside of the directories where explicitly specified source files reside, so things like `import "/etc/passwd";` only work if you add `/=/` as a remapping.

An empty remapping prefix is not allowed.

If there are multiple matches due to remappings, the one with the longest common prefix is selected.

For security reasons the compiler has restrictions what directories it can access. Paths (and their subdirectories) of source files specified on the commandline and paths defined by remappings are allowed for import statements, but everything else is rejected. Additional paths (and their subdirectories) can be allowed via the `--allow-paths /sample/path,/another/sample/path` switch.

If your contracts use *libraries*, you will notice that the bytecode contains substrings of the form `__LibraryName_____`. You can use `solc` as a linker meaning that it will insert the library addresses for you at those points :

Either add `--libraries "Math:0x1234567890123456789 Heap:0xabcdef0123456"` to your command to provide an address for each library or store the string in a file (one library per line) and run `solc` using `--libraries fileName`.

If `solc` is called with the option `--link`, all input files are interpreted to be unlinked binaries (hex-encoded) in the `__LibraryName_____`-format given above and are linked in-place (if the input is read from stdin, it is written to stdout). All options except `--libraries` are ignored (including `-o`) in this case.

If `solc` is called with the option `--standard-json`, it will expect a JSON input (as explained below) on the standard input, and return a JSON output on the standard output. This is the recommended interface for more complex and especially automated uses.

4.6.2 Setting the EVM version to target

When you compile your contract code you can specify the Ethereum virtual machine version to compile for to avoid particular features or behaviours.

Avertissement : Compiling for the wrong EVM version can result in wrong, strange and failing behaviour. Please ensure, especially if running a private chain, that you use matching EVM versions.

On the command line, you can select the EVM version as follows :

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

In the *standard JSON interface*, use the `"evmVersion"` key in the `"settings"` field :

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

Target options

Below is a list of target EVM versions and the compiler-relevant changes introduced at each version. Backward compatibility is not guaranteed between each version.

- `homestead` (oldest version)
- **`tangerineWhistle`**
  - gas cost for access to other accounts increased, relevant for gas estimation and the optimizer.
  - all gas sent by default for external calls, previously a certain amount had to be retained.
- **`spuriousDragon`**
  - gas cost for the `exp` opcode increased, relevant for gas estimation and the optimizer.
- **`byzantium` (default)**
  - opcodes `returndatacopy`, `returndatasize` and `staticcall` are available in assembly.
  - the `staticcall` opcode is used when calling non-library view or pure functions, which prevents the functions from modifying state at the EVM level, i.e., even applies when you use invalid type conversions.
  - it is possible to access dynamic data returned from function calls.
  - `revert` opcode introduced, which means that `revert()` will not waste gas.
- **`constantinople` (still in progress)**
  - opcodes `shl`, `shr` and `sar` are available in assembly.
  - shifting operators use shifting opcodes and thus need less gas.

4.6.3 Compiler Input and Output JSON Description

The recommended way to interface with the Solidity compiler especially for more complex and automated setups is the so-called JSON-input-output interface. The same interface is provided by all distributions of the compiler.

The fields are generally subject to change, some are optional (as noted), but we try to only make backwards compatible changes.

The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output.

The following subsections describe the format through an example. Comments are of course not permitted and used here only for explanatory purposes.

Input Description

```
{
  // Required: Source code language, such as "Solidity", "Vyper", "lll", "assembly",␣
↪etc.
  language: "Solidity",
  // Required
  sources:
  {
    // The keys here are the "global" names of the source files,
    // imports can use other files via remappings (see below).
    "myFile.sol":
    {
      // Optional: keccak256 hash of the source file
      // It is used to verify the retrieved content if imported via URLs.
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): URL(s) to the source file.
      // URL(s) should be imported in this order and the result checked against the
      // keccak256 hash (if available). If the hash doesn't match or none of the
      // URL(s) result in success, an error should be raised.
```

(suite sur la page suivante)

(suite de la page précédente)

```
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        // If files are used, their directories should be added to the command line␣
↪via
        // `--allow-paths <path>`.
        "file:///tmp/path/to/file.sol"
      ]
    },
    "mortal":
    {
      // Optional: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "urls" is used): literal contents of the source file
      "content": "contract mortal is owned { function kill() { if (msg.sender ==␣
↪owner) selfdestruct(owner); } }"
    }
  },
  // Optional
  settings:
  {
    // Optional: Sorted list of remappings
    remappings: [ ":g/dir" ],
    // Optional: Optimizer settings
    optimizer: {
      // disabled by default
      enabled: true,
      // Optimize for how many times you intend to run the code.
      // Lower values will optimize more for initial deployment cost, higher values␣
↪will optimize more for high-frequency usage.
      runs: 200
    },
    evmVersion: "byzantium", // Version of the EVM to compile for. Affects type␣
↪checking and code generation. Can be homestead, tangerineWhistle, spuriousDragon,␣
↪byzantium or constantinople
    // Metadata settings (optional)
    metadata: {
      // Use only literal content and not URLs (false by default)
      useLiteralContent: true
    },
    // Addresses of the libraries. If not all libraries are given here, it can result␣
↪in unlinked objects whose output data is different.
    libraries: {
      // The top level key is the the name of the source file where the library is␣
↪used.
      // If remappings are used, this source file should match the global path after␣
↪remappings were applied.
      // If this key is an empty string, that refers to a global level.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    }
    // The following can be used to select desired outputs.
    // If this field is omitted, then the compiler loads and does type checking, but␣
↪will not generate any outputs apart from errors.
    // The first level key is the file name and the second is the contract name,␣
↪where empty contract name refers to the file itself,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // while the star refers to all of the contracts.
    //
    // The available output types are as follows:
    //   abi - ABI
    //   ast - AST of all source files
    //   legacyAST - legacy AST of all source files
    //   devdoc - Developer documentation (natspec)
    //   userdoc - User documentation (natspec)
    //   metadata - Metadata
    //   ir - New assembly format before desugaring
    //   evm.assembly - New assembly format after desugaring
    //   evm.legacyAssembly - Old-style assembly format in JSON
    //   evm.bytecode.object - Bytecode object
    //   evm.bytecode.opcodes - Opcodes list
    //   evm.bytecode.sourceMap - Source mapping (useful for debugging)
    //   evm.bytecode.linkReferences - Link references (if unlinked object)
    //   evm.deployedBytecode* - Deployed bytecode (has the same options as evm.
↪bytecode)
    //   evm.methodIdentifiers - The list of function hashes
    //   evm.gasEstimates - Function gas estimates
    //   ewasm.wast - eWASM S-expressions format (not supported atm)
    //   ewasm.wasm - eWASM binary format (not supported atm)
    //
    // Note that using a using `evm`, `evm.bytecode`, `ewasm`, etc. will select every
    // target part of that output. Additionally, `*` can be used as a wildcard to␣
↪request everything.
    //
    outputSelection: {
      // Enable the metadata and bytecode outputs of every single contract.
      "*": {
        "*": [ "metadata", "evm.bytecode" ]
      },
      // Enable the abi and opcodes output of MyContract defined in file def.
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      },
      // Enable the source map output of every single contract.
      "*": {
        "*": [ "evm.bytecode.sourceMap" ]
      },
      // Enable the legacy AST output of every single file.
      "*": {
        "": [ "legacyAST" ]
      }
    }
  }
}
```

Output Description

```
{
  // Optional: not present if no errors/warnings were encountered
  errors: [
    {
      // Optional: Location within the source file.
```

(suite sur la page suivante)

(suite de la page précédente)

```
      sourceLocation: {
        file: "sourceFile.sol",
        start: 0,
        end: 100
      ],
      // Mandatory: Error type, such as "TypeError", "InternalCompilerError",
↪"Exception", etc.
      // See below for complete list of types.
      type: "TypeError",
      // Mandatory: Component where the error originated, such as "general", "ewasm",␣
↪etc.
      component: "general",
      // Mandatory ("error" or "warning")
      severity: "error",
      // Mandatory
      message: "Invalid keyword"
      // Optional: the message formatted with source location
      formattedMessage: "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // This contains the file-level outputs. In can be limited/filtered by the␣
↪outputSelection settings.
  sources: {
    "sourceFile.sol": {
      // Identifier (used in source maps)
      id: 1,
      // The AST object
      ast: {},
      // The legacy AST object
      legacyAST: {}
    }
  },
  // This contains the contract-level outputs. It can be limited/filtered by the␣
↪outputSelection settings.
  contracts: {
    "sourceFile.sol": {
      // If the language used has no contract names, this field should equal to an␣
↪empty string.
      "ContractName": {
        // The Ethereum Contract ABI. If empty, it is represented as an empty array.
        // See https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI
        abi: [],
        // See the Metadata Output documentation (serialised JSON string)
        metadata: "{...}",
        // User documentation (natspec)
        userdoc: {},
        // Developer documentation (natspec)
        devdoc: {},
        // Intermediate representation (string)
        ir: "",
        // EVM-related outputs
        evm: {
          // Assembly (string)
          assembly: "",
          // Old-style assembly (object)
          legacyAssembly: {},
          // Bytecode and related details.
```

(suite sur la page suivante)

(suite de la page précédente)

```
          bytecode: {
            // The bytecode as a hex string.
            object: "00fe",
            // Opcodes list (string)
            opcodes: "",
            // The source mapping as a string. See the source mapping definition.
            sourceMap: "",
            // If given, this is an unlinked object.
            linkReferences: {
              "libraryFile.sol": {
                // Byte offsets into the bytecode. Linking replaces the 20 bytes␣
↪located there.
                "Library1": [
                  { start: 0, length: 20 },
                  { start: 200, length: 20 }
                ]
              }
            }
          },
          // The same layout as above.
          deployedBytecode: { },
          // The list of function hashes
          methodIdentifiers: {
            "delegate(address)": "5c19a95c"
          },
          // Function gas estimates
          gasEstimates: {
            creation: {
              codeDepositCost: "420000",
              executionCost: "infinite",
              totalCost: "infinite"
            },
            external: {
              "delegate(address)": "25000"
            },
            internal: {
              "heavyLifting()": "infinite"
            }
          }
        },
        // eWASM related outputs
        ewasm: {
          // S-expressions format
          wast: "",
          // Binary format (hex string)
          wasm: ""
        }
      }
    }
  }
}
```

Error types

1. `JSONError` : JSON input doesn't conform to the required format, e.g. input is not a JSON object, the language is not supported, etc.
2. `IOError` : IO and import processing errors, such as unresolvable URL or hash mismatch in supplied sources.
3. `ParserError` : Source code doesn't conform to the language rules.
4. `DocstringParsingError` : The NatSpec tags in the comment block cannot be parsed.
5. `SyntaxError` : Syntactical error, such as `continue` is used outside of a `for` loop.
6. `DeclarationError` : Invalid, unresolvable or clashing identifier names. e.g. `Identifier not found`
7. `TypeError` : Error within the type system, such as invalid type conversions, invalid assignments, etc.
8. `UnimplementedFeatureError` : Feature is not supported by the compiler, but is expected to be supported in future versions.
9. `InternalCompilerError` : Internal bug triggered in the compiler - this should be reported as an issue.
10. `Exception` : Unknown failure during compilation - this should be reported as an issue.
11. `CompilerError` : Invalid use of the compiler stack - this should be reported as an issue.
12. `FatalError` : Fatal error not processed correctly - this should be reported as an issue.
13. `Warning` : A warning, which didn't stop the compilation, but should be addressed if possible.

4.7 Contract Metadata

The Solidity compiler automatically generates a JSON file, the contract metadata, that contains information about the current contract. You can use this file to query the compiler version, the sources used, the ABI and NatSpec documentation to more safely interact with the contract and verify its source code.

The compiler appends a Swarm hash of the metadata file to the end of the bytecode (for details, see below) of each contract, so that you can retrieve the file in an authenticated way without having to resort to a centralized data provider.

You have to publish the metadata file to Swarm (or another service) so that others can access it. You create the file by using the `solc --metadata` command that generates a file called `ContractName_meta.json`. It contains Swarm references to the source code, so you have to upload all source files and the metadata file.

The metadata file has the following format. The example below is presented in a human-readable way. Properly formatted metadata should use quotes correctly, reduce whitespace to a minimum and sort the keys of all objects to arrive at a unique formatting. Comments are not permitted and used here only for explanatory purposes.

```
{
  // Required: The version of the metadata format
  version: "1",
  // Required: Source code language, basically selects a "sub-version"
  // of the specification
  language: "Solidity",
  // Required: Details about the compiler, contents are specific
  // to the language.
  compiler: {
    // Required for Solidity: Version of the compiler
    version: "0.4.6+commit.2dabbdf0.Emscripten.clang",
    // Optional: Hash of the compiler binary which produced this output
    keccak256: "0x123..."
  },
  // Required: Compilation source files/source units, keys are file names
  sources:
```

(suite sur la page suivante)

(suite de la page précédente)

```
  {
    "myFile.sol": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): Sorted URL(s)
      // to the source file, protocol is more or less arbitrary, but a
      // Swarm URL is recommended
      "urls": [ "bzzr://56ab..." ]
    },
    "mortal": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "url" is used): literal contents of the source file
      "content": "contract mortal is owned { function kill() { if (msg.sender ==
↪owner) selfdestruct(owner); } }"
    }
  },
  // Required: Compiler settings
  settings:
  {
    // Required for Solidity: Sorted list of remappings
    remappings: [ ":g/dir" ],
    // Optional: Optimizer settings (enabled defaults to false)
    optimizer: {
      enabled: true,
      runs: 500
    },
    // Required for Solidity: File and name of the contract or library this
    // metadata is created for.
    compilationTarget: {
      "myFile.sol": "MyContract"
    },
    // Required for Solidity: Addresses for libraries used
    libraries: {
      "MyLib": "0x123123..."
    }
  },
  // Required: Generated information about the contract.
  output:
  {
    // Required: ABI definition of the contract
    abi: [ ... ],
    // Required: NatSpec user documentation of the contract
    userdoc: [ ... ],
    // Required: NatSpec developer documentation of the contract
    devdoc: [ ... ],
  }
}
```

Avertissement : Since the bytecode of the resulting contract contains the metadata hash, any change to the metadata results in a change of the bytecode. This includes changes to a filename or path, and since the metadata includes a hash of all the sources used, a single whitespace change results in different metadata, and different bytecode.

Note : Note the ABI definition above has no fixed order. It can change with compiler versions.

4.7.1 Encoding of the Metadata Hash in the Bytecode

Because we might support other ways to retrieve the metadata file in the future, the mapping `{"bzzr0": <Swarm hash>}` is stored CBOR-encoded. Since the beginning of that encoding is not easy to find, its length is added in a two-byte big-endian encoding. The current version of the Solidity compiler thus adds the following to the end of the deployed bytecode :

```
0xa1 0x65 'b' 'z' 'z' 'r' '0' 0x58 0x20 <32 bytes swarm hash> 0x00 0x29
```

So in order to retrieve the data, the end of the deployed bytecode can be checked to match that pattern and use the Swarm hash to retrieve the file.

Note : The compiler currently uses the « swarm version 0 » hash of the metadata, but this might change in the future, so do not rely on this sequence to start with `0xa1 0x65 'b' 'z' 'z' 'r' '0'`. We might also add additional data to this CBOR structure, so the best option is to use a proper CBOR parser.

4.7.2 Usage for Automatic Interface Generation and NatSpec

The metadata is used in the following way : A component that wants to interact with a contract (e.g. Mist or any wallet) retrieves the code of the contract, from that the Swarm hash of a file which is then retrieved. That file is JSON-decoded into a structure like above.

The component can then use the ABI to automatically generate a rudimentary user interface for the contract.

Furthermore, the wallet can use the NatSpec user documentation to display a confirmation message to the user whenever they interact with the contract, together with requesting authorization for the transaction signature.

Additional information about Ethereum Natural Specification (NatSpec) can be found here.

4.7.3 Usage for Source Code Verification

In order to verify the compilation, sources can be retrieved from Swarm via the link in the metadata file. The compiler of the correct version (which is checked to be part of the « official » compilers) is invoked on that input with the specified settings. The resulting bytecode is compared to the data of the creation transaction or `CREATE` opcode data. This automatically verifies the metadata since its hash is part of the bytecode. Excess data corresponds to the constructor input data, which should be decoded according to the interface and presented to the user.

4.8 Contract ABI Specification

4.8.1 Basic Design

The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data is encoded according to its type, as described in this specification. The encoding is not self describing and thus requires a schema in order to decode.

We assume the interface functions of a contract are strongly typed, known at compilation time and static. We assume that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time.

4.8.2 Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 (SHA-3) hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.

Note : The return type of a function is not part of this signature. In *Solidity's function overloading* return types are not considered. The reason is to keep function call resolution context-independent. The *JSON description of the ABI* however contains both inputs and outputs.

4.8.3 Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

4.8.4 Types

The following elementary types exist :

- `uint<M>` : unsigned integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`. e.g. `uint32`, `uint8`, `uint256`.
- `int<M>` : two's complement signed integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`.
- `address` : equivalent to `uint160`, except for the assumed interpretation and language typing. For computing the function selector, `address` is used.
- `uint`, `int` : synonyms for `uint256`, `int256` respectively. For computing the function selector, `uint256` and `int256` have to be used.
- `bool` : equivalent to `uint8` restricted to the values 0 and 1. For computing the function selector, `bool` is used.
- `fixed<M>x<N>` : signed fixed-point decimal number of `M` bits, `8 <= M <= 256`, `M % 8 ==0`, and `0 < N <= 80`, which denotes the value `v` as `v / (10 ** N)`.
- `ufixed<M>x<N>` : unsigned variant of `fixed<M>x<N>`.
- `fixed`, `ufixed` : synonyms for `fixed128x18`, `ufixed128x18` respectively. For computing the function selector, `fixed128x18` and `ufixed128x18` have to be used.
- `bytes<M>` : binary type of `M` bytes, `0 < M <= 32`.
- `function` : an address (20 bytes) followed by a function selector (4 bytes). Encoded identical to `bytes24`.

The following (fixed-size) array type exists :

- `<type>[M]` : a fixed-length array of `M` elements, `M >= 0`, of the given type.

The following non-fixed-size types exist :

- `bytes` : dynamic sized byte sequence.
- `string` : dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]` : a variable-length array of elements of the given type.

Types can be combined to a tuple by enclosing them inside parentheses, separated by commas :

- `(T1,T2,...,Tn)` : tuple consisting of the types `T1`, ..., `Tn`, `n >= 0`

It is possible to form tuples of tuples, arrays of tuples and so on. It is also possible to form zero-tuples (where `n == 0`).

Mapping Solidity to ABI types

Solidity supports all the types presented above with the same names with the exception of tuples. On the other hand, some Solidity types are not supported by the ABI. The following table shows on the left column Solidity types that are not part of the ABI, and on the right column the ABI types that represent them.

Solidity	ABI
address payable	`address`
contract	`address`
enum	smallest `uint` type that is large enough to hold all values For example, an `enum` of 255 values or less is mapped to `uint8` and an `enum` of 256 values is mapped to `uint16`.
struct	`tuple`

4.8.5 Design Criteria for the Encoding

The encoding is designed to have the following properties, which are especially useful if some arguments are nested arrays :

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]`. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
2. The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative « addresses ».

4.8.6 Formal Specification of the Encoding

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition : The following types are called « dynamic » :
— `bytes`
— `string`
— `T[]` for any `T`
— `T[k]` for any dynamic `T` and any `k >= 0`
— `(T1,...,Tk)` if `Ti` is dynamic for some `1 <= i <= k`
All other types are called « static ».

Definition : `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

Definition : For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being
— `(T1,...,Tk)` for `k >= 0` and any types `T1`, ..., `Tk`
`enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`
where `X = (X(1), ..., X(k))` and `head` and `tail` are defined for `Ti` being a static type as

`head(X(i)) = enc(X(i))` and `tail(X(i)) = ""` (the empty string)

and as

```
head(X(i)) = enc(len(head(X(1)) ... head(X(k)) tail(X(1)) ...
tail(X(i-1)) )) tail(X(i)) = enc(X(i))
```

otherwise, i.e. if `Ti` is a dynamic type.
Note that in the dynamic case, `head(X(i))` is well-defined since the lengths of the head parts only depend on the types and not the values. Its value is the offset of the beginning of `tail(X(i))` relative to the start of `enc(X)`.

- `T[k]` for any `T` and `k`:
  `enc(X) = enc((X[0], ..., X[k-1]))`
  i.e. it is encoded as if it were a tuple with `k` elements of the same type.
- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):
  `enc(X) = enc(k) enc([X[0], ..., X[k-1]])`
  i.e. it is encoded as if it were an array of static size `k`, prefixed with the number of elements.
- `bytes`, of length `k` (which is assumed to be of type `uint256`):
  `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.
- `string`:
  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is utf-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the utf-8 encoded string, not its number of characters.
- `uint<M>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is 32 bytes.
- `address`: as in the `uint160` case
- `int<M>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-order (left) side with `0xff` for negative `X` and with zero bytes for positive `X` such that the length is 32 bytes.
- `bool`: as in the `uint8` case, where `1` is used for `true` and `0` for `false`
- `fixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `int256`.
- `fixed`: as in the `fixed128x18` case
- `ufixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `uint256`.
- `ufixed`: as in the `ufixed128x18` case
- `bytes<M>`: `enc(X)` is the sequence of bytes in `X` padded with trailing zero-bytes to a length of 32 bytes.

Note that for any `X`, `len(enc(X))` is a multiple of 32.

4.8.7 Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

```
function_selector(f) enc((a_1, ..., a_n))
```

and the return values `v_1, ..., v_k` of `f` are encoded as

```
enc((v_1, ..., v_k))
```

i.e. the values are combined into a tuple and encoded.

4.8.8 Examples

Given the contract :

```
pragma solidity >=0.4.16 <0.6.0;

contract Foo {
  function bar(bytes3[2] memory) public pure {}
  function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
  function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

Thus for our `Foo` example if we wanted to call `baz` with the parameters `69` and `true`, we would pass 68 bytes total, which can be broken down into :

- `0xcdcd77c0` : the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature `baz(uint32,bool)`.
- `0x0000000000000000000000000000000000000000000000000000000000000045` : the first parameter, a uint32 value `69` padded to 32 bytes
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter - boolean `true`, padded to 32 bytes

In total :

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000045000000000000000000000000000000
```

It returns a single `bool`. If, for example, it were to return `false`, its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000`, a single bool.

If we wanted to call `bar` with the argument `["abc", "def"]`, we would pass 68 bytes total, broken down into :

- `0xfce353f6` : the Method ID. This is derived from the signature `bar(bytes3[2])`.
- `0x6162630000000000000000000000000000000000000000000000000000000000` : the first part of the first parameter, a `bytes3` value `"abc"` (left-aligned).
- `0x6465660000000000000000000000000000000000000000000000000000000000` : the second part of the first parameter, a `bytes3` value `"def"` (left-aligned).

In total :

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000646566000000000000000000000000
```

If we wanted to call `sam` with the arguments `"dave"`, `true` and `[1,2,3]`, we would pass 292 bytes total, broken down into :

- `0xa5643bf2` : the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])`. Note that `uint` is replaced with its canonical representation `uint256`.
- `0x0000000000000000000000000000000000000000000000000000000000000060` : the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60`.
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter : boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0` : the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xa0`.
- `0x0000000000000000000000000000000000000000000000000000000000000004` : the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.
- `0x6461766500000000000000000000000000000000000000000000000000000000` : the contents of the first argument : the UTF-8 (equal to ASCII in this case) encoding of `"dave"`, padded on the right to 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003` : the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the first entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000002` : the second entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000003` : the third entry of the third parameter.

In total :

```
0xa5643bf2000000000000000000000000000000000000000000000000000000000000006000000000000000000000000000000
```

4.8.9 Use of Dynamic Types

A call to a function with the signature `f(uint,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way :

We take the first four bytes of `sha3("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are :

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32 + 3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows :

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"` :

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case) : 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (newline after function selector and each 32-bytes for clarity) :

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Let us apply the same principle to encode the data for a function with a signature `g(uint[][],string[])` with values `([[1, 2], [3]], ["one", "two", "three"])` but start from the most atomic parts of the encoding :

First we encode the length and data of the first embedded dynamic array `[1, 2]` of the first root array `[[1, 2], [3]]` :

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first array, 2 ; the elements themselves are `1` and `2`)
- `0x0000000000000000000000000000000000000000000000000000000000000001` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000002` (second element)

Then we encode the length and data of the second embedded dynamic array `[3]` of the first root array `[[1, 2], [3]]` :

- `0x0000000000000000000000000000000000000000000000000000000000000001` (number of elements in the second array, 1 ; the element is `3`)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (first element)

Then we need to find the offsets `a` and `b` for their respective dynamic arrays `[1, 2]` and `[3]`. To calculate the offsets we can take a look at the encoded data of the first root array `[[1, 2], [3]]` enumerating each line in the encoding :

```
0 - a                                                                  - offset of [1,␣
↪2]
1 - b                                                                  - offset of [3]
2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1,␣
↪2]
3 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
4 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
5 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
6 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
```

Offset `a` points to the start of the content of the array `[1, 2]` which is line 2 (64 bytes); thus `a = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `b` points to the start of the content of the array `[3]` which is line 5 (160 bytes); thus `b = 0x00000000000000000000000000000000000000000000000000000000000000a0`.

Then we encode the embedded strings of the second root array :

- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of characters in word `"one"`)
- `0x6f6e650000000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"one"`)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of characters in word `"two"`)
- `0x74776f0000000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"two"`)
- `0x0000000000000000000000000000000000000000000000000000000000000005` (number of characters in word `"three"`)
- `0x7468726565000000000000000000000000000000000000000000000000000000` (utf8 representation of word `"three"`)

In parallel to the first root array, since strings are dynamic elements we need to find their offsets `c`, `d` and `e` :

```
0 - c                                                                  - offset for "one
↪"
1 - d                                                                  - offset for "two
↪"
2 - e                                                                  - offset for
↪"three"
3 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of
↪"one"
5 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of
↪"two"
7 - 0000000000000000000000000000000000000000000000000000000000000005 - count for
↪"three"
8 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of
↪"three"
```

Offset `c` points to the start of the content of the string `"one"` which is line 3 (96 bytes); thus `c = 0x0000000000000000000000000000000000000000000000000000000000000060`.

Offset `d` points to the start of the content of the string `"two"` which is line 5 (160 bytes); thus `d = 0x00000000000000000000000000000000000000000000000000000000000000a0`.

Offset `e` points to the start of the content of the string `"three"` which is line 7 (224 bytes); thus `e = 0x00000000000000000000000000000000000000000000000000000000000000e0`.

Note that the encodings of the embedded elements of the root arrays are not dependent on each other and have the same encodings for a function with a signature `g(string[],uint[][])`.

Then we encode the length of the first root array :

— `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first root array, 2 ; the elements themselves are `[1, 2]` and `[3]`)

Then we encode the length of the second root array :

— `0x0000000000000000000000000000000000000000000000000000000000000003` (number of strings in the second root array, 3 ; the strings themselves are `"one"`, `"two"` and `"three"`)

Finally we find the offsets `f` and `g` for their respective root dynamic arrays `[[1, 2], [3]]` and `["one", "two", "three"]`, and assemble parts in the correct order :

```
0x2289b18c                                                          - function␣
↪signature
 0 - f                                                              - offset of [[1,
↪ 2], [3]]
 1 - g                                                              - offset of [
↪"one", "two", "three"]
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [[1,
↪ 2], [3]]
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - offset of [1,␣
↪2]
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset of [3]
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1,␣
↪2]
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
10 - 0000000000000000000000000000000000000000000000000000000000000003 - count for [
↪"one", "two", "three"]
11 - 0000000000000000000000000000000000000000000000000000000000000060 - offset for
↪"one"
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset for
↪"two"
13 - 00000000000000000000000000000000000000000000000000000000000000e0 - offset for
↪"three"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one
↪"
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of
↪"one"
16 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two
↪"
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of
↪"two"
18 - 0000000000000000000000000000000000000000000000000000000000000005 - count for
↪"three"
19 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of
↪"three"
```

Offset `f` points to the start of the content of the array `[[1, 2], [3]]` which is line 2 (64 bytes) ; thus `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `g` points to the start of the content of the array `["one", "two", "three"]` which is line 10 (320 bytes) ; thus `g = 0x0000000000000000000000000000000000000000000000000000000000000140`.

4.8.10 Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series : those which are indexed and those which are not. Those which are indexed, which may number up to 3, are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as :

— `address` : the address of the contract (intrinsically provided by Ethereum) ;
— `topics[0]` : `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo`, it would return `uint256`). If the event is declared as `anonymous` the `topics[0]` is not generated ;
— `topics[n]` : `EVENT_INDEXED_ARGS[n - 1]` (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed) ;
— `data` : `abi_serialise(EVENT_NON_INDEXED_ARGS)` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_serialise` is the ABI serialisation function used for returning a series of typed values from a function, as described above).

For all fixed-length Solidity types, the `EVENT_INDEXED_ARGS` array contains the 32-byte encoded value directly. However, for *types of dynamic length*, which include `string`, `bytes`, and arrays, `EVENT_INDEXED_ARGS` will contain the *Keccak hash* of the packed encoded value (see *Non-standard Packed Mode*), rather than the encoded value directly. This allows applications to efficiently query for values of dynamic-length types (by setting the hash of the encoded value as the topic), but leaves applications unable to decode indexed values they have not queried for. For dynamic-length types, application developers face a trade-off between fast search for predetermined values (if the argument is indexed) and legibility of arbitrary values (which requires that the arguments not be indexed). Developers may overcome this tradeoff and achieve both efficient search and arbitrary legibility by defining events with two arguments — one indexed, one not — intended to hold the same value.

4.8.11 JSON

The JSON format for a contract's interface is given by an array of function and/or event descriptions. A function description is a JSON object with the fields :

— `type` : `"function"`, `"constructor"`, or `"fallback"` (the *unnamed « default » function*) ;
— `name` : the name of the function ;
— `inputs` : an array of objects, each of which contains :
  — `name` : the name of the parameter ;
  — `type` : the canonical type of the parameter (more below).
  — `components` : used for tuple types (more below).
— `outputs` : an array of objects similar to `inputs`, can be omitted if function doesn't return anything ;
— `stateMutability` : a string with one of the following values : `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` (function does not accept Ether) and `payable` (function accepts Ether) ;
— `payable` : `true` if function accepts Ether, `false` otherwise ;
— `constant` : `true` if function is either `pure` or `view`, `false` otherwise.

`type` can be omitted, defaulting to `"function"`, likewise `payable` and `constant` can be omitted, both defaulting to `false`.

Constructor and fallback function never have `name` or `outputs`. Fallback function doesn't have `inputs` either.

Avertissement : The fields `constant` and `payable` are deprecated and will be removed in the future. Instead, the `stateMutability` field can be used to determine the same properties.

Note : Sending non-zero Ether to non-payable function will revert the transaction.

An event description is a JSON object with fairly similar fields :

- `type` : always `"event"`
- `name` : the name of the event ;
- `inputs` : an array of objects, each of which contains :
  - `name` : the name of the parameter ;
  - `type` : the canonical type of the parameter (more below).
  - `components` : used for tuple types (more below).
  - `indexed` : `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous` : `true` if the event was declared as `anonymous`.

For example,

```
pragma solidity >0.4.99 <0.6.0;

contract Test {
  constructor() public { b = hex"12345678901234567890123456789012"; }
  event Event(uint indexed a, bytes32 b);
  event Event2(uint indexed a, bytes32 b);
  function foo(uint a) public { emit Event(a, b); }
  bytes32 b;
}
```

would result in the JSON :

```
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Handling tuple types

Despite that names are intentionally not part of the ABI encoding they do make a lot of sense to be included in the JSON to enable displaying it to the end user. The structure is nested in the following way :

An object with members `name`, `type` and potentially `components` describes a typed variable. The canonical type is determined until a tuple type is reached and the string description up to that point is stored in `type` prefix with the word `tuple`, i.e. it will be `tuple` followed by a sequence of `[]` and `[k]` with integers `k`. The components of the tuple are then stored in the member `components`, which is of array type and has the same structure as the top-level object except that `indexed` is not allowed there.

As an example, the code

```
pragma solidity >=0.4.19 <0.6.0;
pragma experimental ABIEncoderV2;

contract Test {
  struct S { uint a; uint[] b; T[] c; }
  struct T { uint x; uint y; }
  function f(S memory s, T memory t, uint a) public;
  function g() public returns (S memory s, T memory t, uint a);
}
```

would result in the JSON :

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
              }
            ]
          }
        ]
      },
```

(suite sur la page suivante)

(suite de la page précédente)

```
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

4.8.12 Non-standard Packed Mode

Through `abi.encodePacked()`, Solidity supports a non-standard packed mode where :

— types shorter than 32 bytes are neither zero padded nor sign extended and

— dynamic types are encoded in-place and without the length.

As an example encoding `int8, bytes1, uint16, string` with values `-1, 0x42, 0x2424, "Hello, world!"` results in :

```
0xff42242448656c6c6f2c20776f726c6421
  ^^                                  int8(-1)
    ^^                                bytes1(0x42)
      ^^^^                            uint16(0x2424)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") without a length field
```

More specifically, each statically-sized type takes as many bytes as its range has and dynamically-sized types like `string`, `bytes` or `uint[]` are encoded without their length field. This means that the encoding is ambiguous as soon as there are two dynamically-sized elements.

If padding is needed, explicit type conversions can be used : `abi.encodePacked(uint16(0x12)) == hex"0012"`.

Since packed encoding is not used when calling functions, there is no special support for prepending a function selector.

4.9 Yul

Yul (previously also called JULIA or IULIA) is an intermediate language that can compile to various different backends (EVM 1.0, EVM 1.5 and eWASM are planned). Because of that, it is designed to be a usable common denominator of all three platforms. It can already be used for « inline assembly » inside Solidity and future versions of the Solidity

compiler will even use Yul as intermediate language. It should also be easy to build high-level optimizer stages for Yul.

Note : Note that the flavour used for « inline assembly » does not have types (everything is `u256`) and the built-in functions are identical to the EVM opcodes. Please resort to the inline assembly documentation for details.

The core components of Yul are functions, blocks, variables, literals, for-loops, if-statements, switch-statements, expressions and assignments to variables.

Yul is typed, both variables and literals must specify the type with postfix notation. The supported types are `bool`, `u8`, `s8`, `u32`, `s32`, `u64`, `s64`, `u128`, `s128`, `u256` and `s256`.

Yul in itself does not even provide operators. If the EVM is targeted, opcodes will be available as built-in functions, but they can be reimplemented if the backend changes. For a list of mandatory built-in functions, see the section below.

The following example program assumes that the EVM opcodes `mul`, `div` and `mod` are available either natively or as functions and computes exponentiation.

```
{
    function power(base:u256, exponent:u256) -> result:u256
    {
        switch exponent
        case 0:u256 { result := 1:u256 }
        case 1:u256 { result := base }
        default:
        {
            result := power(mul(base, base), div(exponent, 2:u256))
            switch mod(exponent, 2:u256)
                case 1:u256 { result := mul(base, result) }
        }
    }
}
```

It is also possible to implement the same function using a for-loop instead of with recursion. Here, we need the EVM opcodes `lt` (less-than) and `add` to be available.

```
{
    function power(base:u256, exponent:u256) -> result:u256
    {
        result := 1:u256
        for { let i := 0:u256 } lt(i, exponent) { i := add(i, 1:u256) }
        {
            result := mul(result, base)
        }
    }
}
```

4.9.1 Specification of Yul

This chapter describes Yul code. It is usually placed inside a Yul object, which is described in the following chapter.

Grammar :

```
Block = '{' Statement* '}'
Statement =
    Block |
```

(suite sur la page suivante)

(suite de la page précédente)

```
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue
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'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier | BuiltinTypeName
BuiltinTypeName = 'bool' | [us] ( '8' | '32' | '64' | '128' | '256' )
TypedIdentifierList = Identifier ':' TypeName ( ',' Identifier ':' TypeName )*
Literal =
    (NumberLiteral | StringLiteral | HexLiteral | TrueLiteral | FalseLiteral) ':'␣
↪TypeName
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

Restrictions on the Grammar

Switches must have at least one case (including the default case). If all possible values of the expression is covered, the default case should not be allowed (i.e. a switch with a `bool` expression and having both a true and false case should not allow a default case).

Every expression evaluates to zero or more values. Identifiers and Literals evaluate to exactly one value and function calls evaluate to a number of values equal to the number of return values of the function called.

In variable declarations and assignments, the right-hand-side expression (if present) has to evaluate to a number of values equal to the number of variables on the left-hand-side. This is the only situation where an expression evaluating to more than one value is allowed.

Expressions that are also statements (i.e. at the block level) have to evaluate to zero values.

In all other situations, expressions have to evaluate to exactly one value.

The `continue` and `break` statements can only be used inside loop bodies and have to be in the same function as the loop (or both have to be at the top level). The condition part of the for-loop has to evaluate to exactly one value.

Literals cannot be larger than the their type. The largest type defined is 256-bit wide.

Scoping Rules

Scopes in Yul are tied to Blocks (exceptions are functions and the for loop as explained below) and all declarations (`FunctionDefinition`, `VariableDeclaration`) introduce new identifiers into these scopes.

Identifiers are visible in the block they are defined in (including all sub-nodes and sub-blocks). As an exception, identifiers defined in the « init » part of the for-loop (the first block) are visible in all other parts of the for-loop (but not outside of the loop). Identifiers declared in the other parts of the for loop respect the regular syntatical scoping rules. The parameters and return parameters of functions are visible in the function body and their names cannot overlap.

Variables can only be referenced after their declaration. In particular, variables cannot be referenced in the right hand side of their own variable declaration. Functions can be referenced already before their declaration (if they are visible).

Shadowing is disallowed, i.e. you cannot declare an identifier at a point where another identifier with the same name is also visible, even if it is not accessible.

Inside functions, it is not possible to access a variable that was declared outside of that function.

Formal Specification

We formally specify Yul by providing an evaluation function E overloaded on the various nodes of the AST. Any functions can have side effects, so E takes two state objects and the AST node and returns two new state objects and a variable number of other values. The two state objects are the global state object (which in the context of the EVM is the memory, storage and state of the blockchain) and the local state object (the state of local variables, i.e. a segment of the stack in the EVM). If the AST node is a statement, E returns the two state objects and a « mode », which is used for the `break` and `continue` statements. If the AST node is an expression, E returns the two state objects and as many values as the expression evaluates to.

The exact nature of the global state is unspecified for this high level description. The local state `L` is a mapping of identifiers `i` to values `v`, denoted as `L[i] = v`.

For an identifier `v`, let `$v` be the name of the identifier.

We will use a destructuring notation for the AST nodes.

```
E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    let L2 be a restriction of L1 to the identifiers of L
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
```

(suite sur la page suivante)

(suite de la page précédente)

```
            E(G1, L1, St2, ..., Stn)
        otherwise
            G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var1, ..., varn := rhs>: VariableDeclaration) =
    E(G, L, <var1, ..., varn := rhs>: Assignment)
E(G, L, <let var1, ..., varn>: VariableDeclaration) =
    let L1 be a copy of L where L1[$vari] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var1, ..., varn := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$vari] = vi for i = 1, ..., n
    G, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L1, mode = E(G, L, i1, ..., in)
        // mode has to be regular due to the syntactic restrictions
        let G2, L2, mode = E(G1, L1, for {} condition post body)
        // mode has to be regular due to the syntactic restrictions
        let L3 be the restriction of L2 to only variables of L
        G2, L3, regular
    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
            else:
                G3, L3, mode = E(G2, L2, post)
                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, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // Evaluate literals, context doesn't matter
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    if there exists smallest i such that vi = v:
        E(G0, L0, sti)
    else:
        E(G0, L0, st')
```

(suite sur la page suivante)

(suite de la page précédente)

```
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: HexLiteral) = G, L, hexString(l),
    where hexString decodes l from hex and left-aligns it into 32 bytes
E(G, L, l: StringLiteral) = G, L, utf8EncodeLeftAligned(l),
    where utf8EncodeLeftAligned performs a utf8 encoding of l
    and aligns it left into 32 bytes
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal decoding function
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal decoding function
```

Type Conversion Functions

Yul has no support for implicit type conversion and therefore functions exist to provide explicit conversion. When converting a larger type to a shorter type a runtime exception can occur in case of an overflow.

Truncating conversions are supported between the following types :

- `bool`
- `u32`
- `u64`
- `u256`
- `s256`

For each of these a type conversion function exists having the prototype in the form of `<input_type>to<output_type>(x:<input_type>) -> y:<output_type>`, such as `u32tobool(x:u32) -> y:bool`, `u256tou32(x:u256) -> y:u32` or `s256tou256(x:s256) -> y:u256`.

Note : `u32tobool(x:u32) -> y:bool` can be implemented as `y := not(iszerou256(x))` and `booltou32(x:bool) -> y:u32` can be implemented as `switch x case true:bool { y := 1:u32 } case false:bool { y := 0:u32 }`

Low-level Functions

The following functions must be available :

Logic	
not(x :bool) -> z :bool	logical not
and(x :bool, y :bool) -> z :bool	logical and

or(x :bool, y :bool) -> z :bool	logical or
xor(x :bool, y :bool) -> z :bool	xor
Arithmetic	
addu256(x :u256, y :u256) -> z :u256	x + y
subu256(x :u256, y :u256) -> z :u256	x - y
mulu256(x :u256, y :u256) -> z :u256	x * y
divu256(x :u256, y :u256) -> z :u256	x / y
divs256(x :s256, y :s256) -> z :s256	x / y, for signed numbers in two's co
modu256(x :u256, y :u256) -> z :u256	x % y
mods256(x :s256, y :s256) -> z :s256	x % y, for signed numbers in two's c
signextendu256(i :u256, x :u256) -> z :u256	sign extend from (i*8+7)th bit counti
expu256(x :u256, y :u256) -> z :u256	x to the power of y
addmodu256(x :u256, y :u256, m :u256) -> z :u256	(x + y) % m with arbitrary precision
mulmodu256(x :u256, y :u256, m :u256) -> z :u256	(x * y) % m with arbitrary precision
ltu256(x :u256, y :u256) -> z :bool	true if x < y, false otherwise
gtu256(x :u256, y :u256) -> z :bool	true if x > y, false otherwise
sltu256(x :s256, y :s256) -> z :bool	true if x < y, false otherwise (for sign
sgtu256(x :s256, y :s256) -> z :bool	true if x > y, false otherwise (for sign
equ256(x :u256, y :u256) -> z :bool	true if x == y, false otherwise
iszerou256(x :u256) -> z :bool	true if x == 0, false otherwise
notu256(x :u256) -> z :u256	~x, every bit of x is negated
andu256(x :u256, y :u256) -> z :u256	bitwise and of x and y
oru256(x :u256, y :u256) -> z :u256	bitwise or of x and y
xoru256(x :u256, y :u256) -> z :u256	bitwise xor of x and y
shlu256(x :u256, y :u256) -> z :u256	logical left shift of x by y
shru256(x :u256, y :u256) -> z :u256	logical right shift of x by y
saru256(x :u256, y :u256) -> z :u256	arithmetic right shift of x by y
byte(n :u256, x :u256) -> v :u256	nth byte of x, where the most signific
Memory and storage	
mload(p :u256) -> v :u256	mem[p..(p+32))
mstore(p :u256, v :u256)	mem[p..(p+32)) := v
mstore8(p :u256, v :u256)	mem[p] := v & 0xff - only modifies a
sload(p :u256) -> v :u256	storage[p]
sstore(p :u256, v :u256)	storage[p] := v
msize() -> size :u256	size of memory, i.e. largest accessed
Execution control	
create(v :u256, p :u256, n :u256)	create new contract with code mem[p
create2(v :u256, p :u256, n :u256, s :u256)	create new contract with code mem[p
call(g :u256, a :u256, v :u256, in :u256, insize :u256, out :u256, outsize :u256) -> r :u256	call contract at address a with input m
callcode(g :u256, a :u256, v :u256, in :u256, insize :u256, out :u256, outsize :u256) -> r :u256	identical to `call` but only use the co
delegatecall(g :u256, a :u256, in :u256, insize :u256, out :u256, outsize :u256) -> r :u256	identical to `callcode`, but also kee
abort()	abort (equals to invalid instruction or
return(p :u256, s :u256)	end execution, return data mem[p..(p
revert(p :u256, s :u256)	end execution, revert state changes, r
selfdestruct(a :u256)	end execution, destroy current contra
log0(p :u256, s :u256)	log without topics and data mem[p..(
log1(p :u256, s :u256, t1 :u256)	log with topic t1 and data mem[p..(p+
log2(p :u256, s :u256, t1 :u256, t2 :u256)	log with topics t1, t2 and data mem[p
log3(p :u256, s :u256, t1 :u256, t2 :u256, t3 :u256)	log with topics t, t2, t3 and data mem
log4(p :u256, s :u256, t1 :u256, t2 :u256, t3 :u256, t4 :u256)	log with topics t1, t2, t3, t4 and data

State queries	
blockcoinbase() -> address :u256	current mining beneficiary
blockdifficulty() -> difficulty :u256	difficulty of the current block
blockgaslimit() -> limit :u256	block gas limit of the current block
blockhash(b :u256) -> hash :u256	hash of block nr b - only for last 256
blocknumber() -> block :u256	current block number
blocktimestamp() -> timestamp :u256	timestamp of the current block in sec
txorigin() -> address :u256	transaction sender
txgasprice() -> price :u256	gas price of the transaction
gasleft() -> gas :u256	gas still available to execution
balance(a :u256) -> v :u256	wei balance at address a
this() -> address :u256	address of the current contract / exec
caller() -> address :u256	call sender (excluding delegatecall)
callvalue() -> v :u256	wei sent together with the current cal
calldataload(p :u256) -> v :u256	call data starting from position p (32
calldatasize() -> v :u256	size of call data in bytes
calldatacopy(t :u256, f :u256, s :u256)	copy s bytes from calldata at position
codesize() -> size :u256	size of the code of the current contra
codecopy(t :u256, f :u256, s :u256)	copy s bytes from code at position f t
extcodesize(a :u256) -> size :u256	size of the code at address a
extcodecopy(a :u256, t :u256, f :u256, s :u256)	like codecopy(t, f, s) but take code at
extcodehash(a :u256)	code hash of address a
Others	
discard(unused :bool)	discard value
discardu256(unused :u256)	discard value
splitu256tou64(x :u256) -> (x1 :u64, x2 :u64, x3 :u64, x4 :u64)	split u256 to four u64's
combineu64tou256(x1 :u64, x2 :u64, x3 :u64, x4 :u64) -> (x :u256)	combine four u64's into a single u25
keccak256(p :u256, s :u256) -> v :u256	keccak(mem[p...(p+s)))

Backends

Backends or targets are the translators from Yul to a specific bytecode. Each of the backends can expose functions prefixed with the name of the backend. We reserve `evm_` and `ewasm_` prefixes for the two proposed backends.

Backend : EVM

The EVM target will have all the underlying EVM opcodes exposed with the *evm_* prefix.

Backend : « EVM 1.5 »

TBD

Backend : eWASM

TBD

4.9.2 Specification of Yul Object

Grammar :

```
TopLevelObject = 'object' '{' Code? ( Object | Data )* '}'
Object = 'object' StringLiteral '{' Code? ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral HexLiteral
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
```

Above, `Block` refers to `Block` in the Yul code grammar explained in the previous chapter.

An example Yul Object is shown below :

```
// Code consists of a single object. A single "code" node is the code of the object.
// Every (other) named object or data section is serialized and made accessible to
↪the special built-in functions datacopy / dataoffset / datasize
object {
    code {
        let size = datasize("runtime")
        let offset = allocate(size)
        // This will turn into a memory->memory copy for eWASM and a codecopy for
↪EVM
        datacopy(dataoffset("runtime"), offset, size)
        // this is a constructor and the runtime code is returned
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "runtime" {
        code {
            // runtime code

            let size = datasize("Contract2")
            let offset = allocate(size)
            // This will turn into a memory->memory copy for eWASM and a codecopy
↪for EVM
            datacopy(dataoffset("Contract2"), offset, size)
            // constructor parameter is a single number 0x1234
            mstore(add(offset, size), 0x1234)
            create(offset, add(size, 32))
        }

        // Embedded object. Use case is that the outside is a factory contract, and
↪Contract2 is the code to be created by the factory
        object "Contract2" {
            code {
                // code here ...
            }

            object "runtime" {
                code {
                    // code here ...
                }
             }
```

(suite sur la page suivante)

(suite de la page précédente)

```
                data "Table1" hex"4123"
            }
        }
}
```

4.10 Style Guide

4.10.1 Introduction

This guide is intended to provide coding conventions for writing solidity code. This guide should be thought of as an evolving document that will change over time as useful conventions are found and old conventions are rendered obsolete.

Many projects will implement their own style guides. In the event of conflicts, project specific style guides take precedence.

The structure and many of the recommendations within this style guide were taken from python's pep8 style guide.

The goal of this guide is *not* to be the right way or the best way to write solidity code. The goal of this guide is *consistency*. A quote from python's pep8 captures this concept well.

> A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important. But most importantly : know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgement. Look at other examples and decide what looks best. And don't hesitate to ask !

4.10.2 Code Layout

Indentation

Use 4 spaces per indentation level.

Tabs or Spaces

Spaces are the preferred indentation method.

Mixing tabs and spaces should be avoided.

Blank Lines

Surround top level declarations in solidity source with two blank lines.

Yes :

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
    // ...
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract B {
    // ...
}


contract C {
    // ...
}
```

No :

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Within a contract surround function declarations with a single blank line.

Blank lines may be omitted between groups of related one-liners (such as stub functions for an abstract contract)

Yes :

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
    function spam() public pure;
    function ham() public pure;
}


contract B is A {
    function spam() public pure {
        // ...
    }

    function ham() public pure {
        // ...
    }
}
```

No :

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
    function spam() public pure {
        // ...
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    function ham() public pure {
        // ...
    }
}
```

Maximum Line Length

Keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code.

Wrapped lines should conform to the following guidelines.

1. The first argument should not be attached to the opening parenthesis.
2. One, and only one, indent should be used.
3. Each argument should fall on its own line.
4. The terminating element, `);`, should be placed on the final line by itself.

Function Calls

Yes :

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

No :

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Assignment Statements

Yes :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

No :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(argument1,
                                                                   argument2,
                                                                   argument3,
                                                                   argument4);
```

Event Definitions and Event Emitters

Yes :

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

No :

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Source File Encoding

UTF-8 or ASCII encoding is preferred.

Imports

Import statements should always be placed at the top of the file.

Yes :

```
pragma solidity >=0.4.0 <0.6.0;

import "./Owned.sol";

contract A {
    // ...
}

contract B is Owned {
    // ...
}
```

No :

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
    // ...
}


import "./Owned.sol";


contract B is Owned {
    // ...
}
```

Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered :

- constructor
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the `view` and `pure` functions last.

Yes :

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
    constructor() public {
        // ...
    }

    function() external {
        // ...
    }

    // External functions
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // ...

    // External functions that are view
    // ...

    // External functions that are pure
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

No :

```
pragma solidity >=0.4.0 <0.6.0;

contract A {

    // External functions
    // ...

    function() external {
        // ...
    }

    // Private functions
    // ...

    // Public functions
    // ...

    constructor() public {
        // ...
    }

    // Internal functions
    // ...
}
```

Whitespace in Expressions

Avoid extraneous whitespace in the following situations :

Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.

Yes :

```
spam(ham[1], Coin({name: "ham"}));
```

No :

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception :

```
function singleLine() public { spam(); }
```

Immediately before a comma, semicolon :

Yes :

```
function spam(uint i, Coin coin) public;
```

No :

```
function spam(uint i , Coin coin) public ;
```

More than one space around an assignment or other operator to align with another :

Yes :

```
x = 1;
y = 2;
long_variable = 3;
```

No :

```
x             = 1;
y             = 2;
long_variable = 3;
```

Don't include a whitespace in the fallback function :

Yes :

```
function() external {
    ...
}
```

No :

```
function () external {
    ...
}
```

Control Structures

The braces denoting the body of a contract, library, functions and structs should :
- open on the same line as the declaration
- close on their own line at the same indentation level as the beginning of the declaration.
- The opening brace should be proceeded by a single space.

Yes :

```
pragma solidity >=0.4.0 <0.6.0;

contract Coin {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No :

```
pragma solidity >=0.4.0 <0.6.0;

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

The same recommendations apply to the control structures `if`, `else`, `while`, and `for`.

Additionally there should be a single space between the control structures `if`, `while`, and `for` and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.

Yes :

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No :

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

For control structures whose body contains a single statement, omitting the braces is ok *if* the statement is contained on a single line.

Yes :

```
if (x < 10)
    x += 1;
```

No :

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

For `if` blocks which have an `else` or `else if` clause, the `else` should be placed on the same line as the `if`'s closing brace. This is an exception compared to the rules of other block-like structures.

Yes :

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

No :

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.

The closing brace should be at the same indentation level as the function declaration.

The opening brace should be preceded by a single space.

Yes :

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyowner returns (uint) {
    return x + 1;
}
```

No :

```
function increment(uint x) public pure returns (uint)
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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;}
```

You should explicitly label the visibility of all functions, including constructors.

Yes :

```
function explicitlyPublic(uint val) public {
    doSomething();
}
```

No :

```
function implicitlyPublic(uint val) {
    doSomething();
}
```

The visibility modifier for a function should come before any custom modifiers.

Yes :

```
function kill() public onlyowner {
    selfdestruct(owner);
}
```

No :

```
function kill() onlyowner public {
    selfdestruct(owner);
}
```

For long function declarations, it is recommended to drop each argument onto it's own line at the same indentation level as the function body. The closing parenthesis and opening bracket should be placed on their own line as well at the same indentation level as the function declaration.

Yes :

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    doSomething();
}
```

No :

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to its own line.

Yes :

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z,
)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}
```

No :

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyowner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyowner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address) {
    doSomething();
}
```

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the *Maximum Line Length* section.

Yes :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

No :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
```

(suite sur la page suivante)

(suite de la page précédente)

```
            uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Yes :

```
pragma solidity >=0.4.0 <0.6.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) public {
    }
}
contract C {
    constructor(uint, uint) public {
    }
}
contract D {
    constructor(uint) public {
    }
}

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)
        public
    {
        // do something with param5
        x = param5;
    }
}
```

No :

```
pragma solidity >=0.4.0 <0.6.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) public {
    }
}
contract C {
    constructor(uint, uint) public {
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
contract D {
    constructor(uint) public {
    }
}

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)
    public
    {
        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)
        public {
        x = param5;
    }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible :

```
function shortFunction() public { doSomething(); }
```

These guidelines for function declarations are intended to improve readability. Authors should use their best judgement as this guide does not try to cover all possible permutations for function declarations.

Mappings

TODO

Variable Declarations

Declarations of array variables should not have a space between the type and the brackets.

Yes :

```
uint[] x;
```

No :

```
uint [] x;
```

Other Recommendations

— Strings should be quoted with double-quotes instead of single-quotes.

Yes :

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No :

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

— Surround operators with a single space on either side.

Yes :

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No :

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

— Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence. This is meant to allow for improved readability for complex statement. You should always use the same amount of whitespace on either side of an operator :

Yes :

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No :

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

4.10.3 Order of Layout

Layout contract elements in the following order :

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Inside each contract, library or interface, use the following order :

1. Type declarations
2. State variables
3. Events
4. Functions

Note : It might be clearer to declare types close to their use in events or state variables.

4.10.4 Naming Conventions

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant *meta* information that would otherwise not be immediately available.

The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Lastly, consistency within a codebase should always supersede any conventions outlined in this document.

Naming Styles

To avoid confusion, the following names will be used to refer to different naming styles.

— `b` (single lowercase letter)
— `B` (single uppercase letter)
— `lowercase`
— `lower_case_with_underscores`
— `UPPERCASE`
— `UPPER_CASE_WITH_UNDERSCORES`
— `CapitalizedWords` (or CapWords)
— `mixedCase` (differs from CapitalizedWords by initial lowercase character !)
— `Capitalized_Words_With_Underscores`

Note : When using initialisms in CapWords, capitalize all the letters of the initialisms. Thus HTTPServerError is better than HttpServerError. When using initialisms is mixedCase, capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name. Thus xmlHTTPRequest is better than XMLHTTPRequest.

Names to Avoid

— `l` - Lowercase letter el
— `O` - Uppercase letter oh
— `I` - Uppercase letter eye

Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.

Contract and Library Names

— Contracts and libraries should be named using the CapWords style. Examples : `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
— Contract and library names should also match their filenames.
— If a contract file includes multiple contracts and/or libraries, then the filename should match the *core contract*. This is not recommended however if it can be avoided.

As shown in the example below, if the contract name is *Congress* and the library name is *Owned*, then their associated filenames should be *Congress.sol* and *Owned.sol*.

Yes :

```
pragma solidity >=0.4.0 <0.6.0;

// Owned.sol
contract Owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}

// Congress.sol
import "./Owned.sol";

contract Congress is Owned, TokenRecipient {
    //...
}
```

No :

```
pragma solidity >=0.4.0 <0.6.0;

// owned.sol
contract owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}

// Congress.sol
import "./owned.sol";

contract Congress is owned, tokenRecipient {
```

(suite sur la page suivante)

(suite de la page précédente)

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

Struct Names

Structs should be named using the CapWords style. Examples : `MyCoin`, `Position`, `PositionXY`.

Event Names

Events should be named using the CapWords style. Examples : `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

Function Names

Functions other than constructors should use mixedCase. Examples : `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Function Argument Names

Function arguments should use mixedCase. Examples : `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

When writing library functions that operate on a custom struct, the struct should be the first argument and should always be named `self`.

Local and State Variable Names

Use mixedCase. Examples : `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constants

Constants should be named with all capital letters with underscores separating words. Examples : `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Modifier Names

Use mixedCase. Examples : `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enums

Enums, in the style of simple type declarations, should be named using the CapWords style. Examples : `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Avoiding Naming Collisions

— `single_trailing_underscore_`

This convention is suggested when the desired name collides with that of a built-in or otherwise reserved name.

General Recommendations

TODO

4.11 Common Patterns

4.11.1 Withdrawal from Contracts

The recommended method of sending funds after an effect is using the withdrawal pattern. Although the most intuitive method of sending Ether, as a result of an effect, is a direct `transfer` call, this is not recommended as it introduces a potential security risk. You may read more about this on the *Security Considerations* page.

The following is an example of the withdrawal pattern in practice in a contract where the goal is to send the most money to the contract in order to become the « richest », inspired by King of the Ether.

In the following contract, if you are usurped as the richest, you will receive the funds of the person who has gone on to become the new richest.

```
pragma solidity >0.4.99 <0.6.0;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable returns (bool) {
        if (msg.value > mostSent) {
            pendingWithdrawals[richest] += msg.value;
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        msg.sender.transfer(amount);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

This is as opposed to the more intuitive sending pattern :

```
pragma solidity >0.4.99 <0.6.0;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable returns (bool) {
        if (msg.value > mostSent) {
            // This line can cause problems (explained below).
            richest.transfer(msg.value);
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }
}
```

Notice that, in this example, an attacker could trap the contract into an unusable state by causing `richest` to be the address of a contract that has a fallback function which fails (e.g. by using `revert()` or by just consuming more than the 2300 gas stipend transferred to them). That way, whenever `transfer` is called to deliver funds to the « poisoned » contract, it will fail and thus also `becomeRichest` will fail, with the contract being stuck forever.

In contrast, if you use the « withdraw » pattern from the first example, the attacker can only cause his or her own withdraw to fail and not the rest of the contract's workings.

4.11.2 Restricting Access

Restricting access is a common pattern for contracts. Note that you can never restrict any human or computer from reading the content of your transactions or your contract's state. You can make it a bit harder by using encryption, but if your contract is supposed to read the data, so will everyone else.

You can restrict read access to your contract's state by **other contracts**. That is actually the default unless you declare make your state variables `public`.

Furthermore, you can restrict who can make modifications to your contract's state or call your contract's functions and this is what this section is about.

The use of **function modifiers** makes these restrictions highly readable.

```
pragma solidity >=0.4.22 <0.6.0;

contract AccessRestriction {
    // These will be assigned at the construction
    // phase, where `msg.sender` is the account
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // creating this contract.
    address public owner = msg.sender;
    uint public creationTime = now;

    // Modifiers can be used to change
    // the body of a function.
    // If this modifier is used, it will
    // prepend a check that only passes
    // if the function is called from
    // a certain address.
    modifier onlyBy(address _account)
    {
        require(
            msg.sender == _account,
            "Sender not authorized."
        );
        // Do not forget the "_;"! It will
        // be replaced by the actual function
        // body when the modifier is used.
        _;
    }

    /// Make `_newOwner` the new owner of this
    /// contract.
    function changeOwner(address _newOwner)
        public
        onlyBy(owner)
    {
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        require(
            now >= _time,
            "Function called too early."
        );
        _;
    }

    /// Erase ownership information.
    /// May only be called 6 weeks after
    /// the contract has been created.
    function disown()
        public
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // This modifier requires a certain
    // fee being associated with a function call.
    // If the caller sent too much, he or she is
    // refunded, but only after the function body.
    // This was dangerous before Solidity version 0.4.0,
    // where it was possible to skip the part after `_;`.
    modifier costs(uint _amount) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        require(
            msg.value >= _amount,
            "Not enough Ether provided."
        );
        _;
        if (msg.value > _amount)
            msg.sender.transfer(msg.value - _amount);
    }

    function forceOwnerChange(address _newOwner)
        public
        payable
        costs(200 ether)
    {
        owner = _newOwner;
        // just some example condition
        if (uint(owner) & 0 == 1)
            // This did not refund for Solidity
            // before version 0.4.0.
            return;
        // refund overpaid fees
    }
}
```

A more specialised way in which access to function calls can be restricted will be discussed in the next example.

4.11.3 State Machine

Contracts often act as a state machine, which means that they have certain **stages** in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage (especially if the contract models **interaction**). It is also common that some stages are automatically reached at a certain point in **time**.

An example for this is a blind auction contract which starts in the stage « accepting blinded bids », then transitions to « revealing bids » which is ended by « determine auction outcome ».

Function modifiers can be used in this situation to model the states and guard against incorrect usage of the contract.

Example

In the following example, the modifier `atStage` ensures that the function can only be called at a certain stage.

Automatic timed transitions are handled by the modifier `timeTransitions`, which should be used for all functions.

Note : Modifier Order Matters. If atStage is combined with timedTransitions, make sure that you mention it after the latter, so that the new stage is taken into account.

Finally, the modifier `transitionNext` can be used to automatically go to the next stage when the function finishes.

Note : Modifier May be Skipped. This only applies to Solidity before version 0.4.0 : Since modifiers are applied by simply replacing code and not by using a function call, the code in the transitionNext modifier can be skipped if the

function itself uses return. If you want to do that, make sure to call nextStage manually from those functions. Starting with version 0.4.0, modifier code will run even if the function explicitly returns.

```
pragma solidity >=0.4.22 <0.6.0;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }

    // This is the current stage.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(
            stage == _stage,
            "Function cannot be called at this time."
        );
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Perform timed transitions. Be sure to mention
    // this modifier first, otherwise the guards
    // will not take the new stage into account.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    now >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                now >= creationTime + 12 days)
            nextStage();
        // The other stages transition by transaction
        _;
    }

    // Order of the modifiers matters here!
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // We will not implement that here
    }

    function reveal()
        public
```

(suite sur la page suivante)

(suite de la page précédente)

```
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // This modifier goes to the next stage
    // after the function is done.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

4.12 List of Known Bugs

Below, you can find a JSON-formatted list of some of the known security-relevant bugs in the Solidity compiler. The file itself is hosted in the Github repository. The list stretches back as far as version 0.3.0, bugs known to be present only in versions preceding that are not listed.

There is another file called bugs_by_version.json, which can be used to check which bugs affect a specific version of the compiler.

Contract source verification tools and also other tools interacting with contracts should consult this list according to the following criteria :

— It is mildly suspicious if a contract was compiled with a nightly compiler version instead of a released version. This list does not keep track of unreleased or nightly versions.
— It is also mildly suspicious if a contract was compiled with a version that was not the most recent at the time the contract was created. For contracts created from other contracts, you have to follow the creation chain back to a transaction and use the date of that transaction as creation date.
— It is highly suspicious if a contract was compiled with a compiler that contains a known bug and the contract was created at a time where a newer compiler version containing a fix was already released.

The JSON file of known bugs below is an array of objects, one for each bug, with the following keys :

name Unique name given to the bug

summary Short description of the bug

description Detailed description of the bug

link URL of a website with more detailed information, optional

introduced The first published compiler version that contained the bug, optional

fixed The first published compiler version that did not contain the bug anymore

publish The date at which the bug became known publicly, optional

severity Severity of the bug : very low, low, medium, high. Takes into account discoverability in contract tests, likelihood of occurrence and potential damage by exploits.

conditions Conditions that have to be met to trigger the bug. Currently, this is an object that can contain a boolean value `optimizer`, which means that the optimizer has to be switched on to enable the bug. If no conditions are given, assume that the bug is present.

check This field contains different checks that report whether the smart contract contains the bug or not. The first type of check are Javascript regular expressions that are to be matched against the source code (« source-regex ») if the bug is present. If there is no match, then the bug is very likely not present. If there is a match, the bug might be present. For improved accuracy, the checks should be applied to the source code after stripping comments. The second type of check are patterns to be checked on the compact AST of the Solidity program (« ast-compact-json-path »). The specified search query is a JsonPath expression. If at least one path of the Solidity AST matches the query, the bug is likely present.

```
[
    {
        "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.",
        "fixed": "0.4.25",
        "severity": "medium/high",
        "check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
    },
    {
        "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.",
        "introduced": "0.4.17",
        "fixed": "0.4.25",
        "severity": "very low",
        "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')]..
↪[?(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
↪'struct'))]"}
    },
    {
        "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.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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][^\\]]*\\][^{;]*[;{]"}
    },
    {
        "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"
    },
    {
        "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"
    },
    {
        "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 successuful.",
        "introduced": "0.3.0",
        "fixed": "0.4.15",
        "severity": "low"
    },
    {
        "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"
    },
    {
        "name": "SkipEmptyStringLiteral",
        "summary": "If \"\" is used in a function call, the following function␣
↪arguments will not be correctly passed to the function.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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"
    },
    {
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode␣
↪by routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.ethereum.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "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"
    },
    {
        "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
        }
    },
    {
        "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.",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "link": "https://blog.ethereum.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "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
        }
    },
    {
        "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"
    },
    {
        "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"
    },
    {
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite␣
↪loop and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this␣
↪length is zero, code was generated that did not terminate and thus consumed all gas.
↪",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "name": "OptimizerClearStateOnCodePathJoin",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "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
        }
    },
    {
        "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"
    },
    {
        "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"
    },
    {
        "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"
    }
]
```

4.13 Contributing

Help is always appreciated !

To get started, you can try *Compilation à partir des sources* in order to familiarize yourself with the components of Solidity and the build process. Also, it may be useful to become well-versed at writing smart-contracts in Solidity.

In particular, we need help in the following areas :

— Improving the documentation
— Responding to questions from other users on StackExchange and the Solidity Gitter
— Fixing and responding to Solidity's GitHub issues, especially those tagged as up-for-grabs which are meant as introductory issues for external contributors.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project - in the issues, pull requests, or Gitter channels - you agree to abide by its terms.

4.13.1 How to Report Issues

To report an issue, please use the GitHub issues tracker. When reporting issues, please mention the following details :

— Which version of Solidity you are using
— What was the source code (if applicable)
— Which platform are you running on
— How to reproduce the issue
— What was the result of the issue
— What the expected behaviour is

Reducing the source code that caused the issue to a bare minimum is always very helpful and sometimes even clarifies a misunderstanding.

4.13.2 Workflow for Pull Requests

In order to contribute, please fork off of the `develop` branch and make your changes there. Your commit messages should detail *why* you made your change in addition to *what* you did (unless it is a tiny change).

If you need to pull in any changes from `develop` after making your fork (for example, to resolve potential merge conflicts), please avoid using `git merge` and instead, `git rebase` your branch. This will help us review your change more easily.

Additionally, if you are writing a new feature, please ensure you add appropriate test cases under `test/` (see below).

However, if you are making a larger change, please consult with the Solidity Development Gitter channel (different from the one mentioned above, this one is focused on compiler and language development instead of language use) first.

New features and bugfixes should be added to the `Changelog.md` file : please follow the style of previous entries, when applicable.

Finally, please make sure you respect the coding style for this project. Also, even though we do CI testing, please test your code and ensure that it builds locally before submitting a pull request.

Thank you for your help !

4.13.3 Running the compiler tests

There is a script at `scripts/tests.sh` which executes most of the tests and runs `aleth` automatically if it is in the path, but does not download it, so it most likely will not work right away. Please read on for the details.

Solidity includes different types of tests. Most of them are bundled in the application called `soltest`. Some of them require the `aleth` client in testing mode, some others require `libz3` to be installed.

To run a basic set of tests that neither require `aleth` nor `libz3`, run `./scripts/soltest.sh --no-ipc --no-smt`. This script will run `build/test/soltest` internally.

The option `--no-smt` disables the tests that require `libz3` and `--no-ipc` disables those that require `aleth`.

If you want to run the ipc tests (those test the semantics of the generated code), you need to install aleth and run it in testing mode : `aleth --test -d /tmp/testeth` (make sure to rename it).

Then you run the actual tests : `./scripts/soltest.sh --ipcpath /tmp/testeth/geth.ipc`.

To run a subset of tests, filters can be used : `./scripts/soltest.sh -t TestSuite/TestName --ipcpath /tmp/testeth/geth.ipc`, where `TestName` can be a wildcard `*`.

The script `scripts/tests.sh` also runs commandline tests and compilation tests in addition to those found in `soltest`.

The CI even runs some additional tests (including `solc-js` and testing third party Solidity frameworks) that require compiling the Emscripten target.

Note : Some versions of `aleth` cannot be used for testing. We suggest using the same version that is used by the Solidity continuous integration tests. Currently the CI uses `d661ac4fec0aeffbedcdc195f67f5ded0c798278` of `aleth`.

Writing and running syntax tests

Syntax tests check that the compiler generates the correct error messages for invalid code and properly accepts valid code. They are stored in individual files inside `tests/libsolidity/syntaxTests`. These files must contain annotations, stating the expected result(s) of the respective test. The test suite will compile and check them against the given expectations.

Example : `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

A syntax test must contain at least the contract under test itself, followed by the separator `// ----`. The following comments are used to describe the expected compiler errors or warnings. The number range denotes the location in the source where the error occurred. In case the contract should compile without any errors or warning, the section after the separator has to be empty and the separator can be left out completely.

In the above example, the state variable `variable` was declared twice, which is not allowed. This will result in a `DeclarationError` stating that the identifier was already declared.

The tool that is being used for those tests is called `isoltest` and can be found under `./test/tools/`. It is an interactive tool which allows editing of failing contracts using your preferred text editor. Let's try to break this test by removing the second declaration of `variable` :

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

Running `./test/isoltest` again will result in a test failure :

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        contract test {
            uint256 variable;
        }

    Expected result:
        DeclarationError: (36-52): Identifier already declared.
    Obtained result:
        Success
```

`isoltest` prints the expected result next to the obtained result, but also provides a way to change edit / update / skip the current contract or to even quit. It offers several options for failing tests :

- edit : `isoltest` tries to open the contract in an editor so you can adjust it. It either uses the editor given on the command line (as `isoltest --editor /path/to/editor`), in the environment variable `EDITOR` or just `/usr/bin/editor` (in this order).
- update : Updates the contract under test. This either removes the annotation which contains the exception not met or adds missing expectations. The test will then be run again.
- skip : Skips the execution of this particular test.
- quit : Quits `isoltest`.

Automatically updating the test above will change it to

```
contract test {
    uint256 variable;
}
// ----
```

and re-run the test. It will now pass again :

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

Note : Please choose a name for the contract file that explains what it tests, e.g. `double_variable_declaration.sol`. Do not put more than one contract into a single file, unless you are testing inheritance or cross-contract calls. Each file should test one aspect of your new feature.

4.13.4 Running the Fuzzer via AFL

Fuzzing is a technique that runs programs on more or less random inputs to find exceptional execution states (segmentation faults, exceptions, etc). Modern fuzzers are clever and do a directed search inside the input. We have a specialized binary called `solfuzzer` which takes source code as input and fails whenever it encounters an internal compiler error, segmentation fault or similar, but does not fail if e.g. the code contains an error. This way, internal problems in the compiler can be found by fuzzing tools.

We mainly use AFL for fuzzing. You need to download and install AFL packages from your repos (afl, afl-clang) or build them manually. Next, build Solidity (or just the `solfuzzer` binary) with AFL as your compiler :

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

At this stage you should be able to see a message similar to the following :

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

If the instrumentation messages did not appear, try switching the cmake flags pointing to AFL's clang binaries :

```
# if previously failed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Othwerise, upon execution the fuzzer will halt with an error saying binary is not instrumented :

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6920
```

Next, you need some example source files. This will make it much easier for the fuzzer to find errors. You can either copy some files from the syntax tests or extract test files from the documentation or the other tests :

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↪SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs docs
```

The AFL documentation states that the corpus (the initial input files) should not be too large. The files themselves should not be larger than 1 kB and there should be at most one input file per functionality, so better start with a small number of input files. There is also a tool called `afl-cmin` that can trim input files that result in similar behaviour of the binary.

Now run the fuzzer (the `-m` extends the size of memory to 60 MB) :

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

The fuzzer will create source files that lead to failures in `/tmp/fuzzer_reports`. Often it finds many similar source files that produce the same error. You can use the tool `scripts/uniqueErrors.sh` to filter out the unique errors.

4.13.5 Whiskers

Whiskers is a string templating system similar to Mustache. It is used by the compiler in various places to aid readability, and thus maintainability and verifiability, of the code.

The syntax comes with a substantial difference to Mustache : the template markers `{{` and `}}` are replaced by < and > in order to aid parsing and avoid conflicts with *Assembleur en ligne (inline)* (The symbols < and > are invalid in inline assembly, while `{` and `}` are used to delimit blocks). Another limitation is that lists are only resolved one depth and they will not recurse. This may change in the future.

A rough specification is the following :

Any occurrence of `<name>` is replaced by the string-value of the supplied variable `name` without any escaping and without iterated replacements. An area can be delimited by `<#name>...</name>`. It is replaced by as many concatenations of its contents as there were sets of variables supplied to the template system, each time replacing any `<inner>` items by their respective value. Top-level variables can also be used inside such areas.

4.14 FAQ - Questions Fréquentes

Cette liste a été compilée par fivedogit.

4.14.1 Questions Basiques

Qu'est-ce que le « payload » d'une transaction ?

C'est la charge utile (binaire) envoyée avec la requête.

Créer un contrat qui peut être « tué » et retourner ses fonds

Avertissement : Tuer les contrats semble être une bonne idée, parce que « nettoyer » est toujours bon, mais comme on l'a vu précédement, il ne nettoie pas vraiment. En outre, si l'Ether est envoyé à des contrats tués, l'Ether sera perdu à jamais.

Si vous souhaitez tuer vos contrats, l'idéal est de les **désactiver** en changeant un état interne qui empêche le lancement de ses fonctions. Cela rendra impossible l'utilisation du contrat et l'éther envoyé au contrat sera remboursé automatiquement.

Répondons maintenant à la question : Dans un constructeur, `msg.sender` est le nom de l'expéditeur créateur. Stocker-le. Puis `selfdestruct(createur);` pour tuer et rendre les fonds.

Voir cet exemple.

Notez que si vous faites un `import "mortal"` en haut e vos contrats et déclarez `contract SomeContract is mortal { ...` et compilez par un compilateur qui l'inclus (comme Remix), alors la fonction `kill()` est gérée pour vous. Une fois qu'un contrat est « mortal », vous pouvez `contractname.kill.sendTransaction({from:eth.coinbase})`, pile poil comme dans l'exemple.

Peut-on retourner une `array` ou `string` dans un appel de fonctions en Solidity ?

Oui, voir array_receiver_and_returner.sol.

Est-il possible d'initialiser un tableau à la déclaration tel que : `string[] myarray = ["a", "b"];` ?

Oui. Cependant il devrait être noté que ça ne marche actuellement qu'avec les tableaux de taille statique. Vous pouvez même générer un tableau à la volée dans la ligne de retour.

Exemple :

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    function f() public pure returns (uint8[5] memory) {
        string[4] memory adaArr = ["This", "is", "an", "array"];
        adaArr[0] = "That";
        return [1, 2, 3, 4, 5];
    }
}
```

Un contrat peut-il retourner une `struct` ?

Oui, mais seulement dans un appel de fonction `internal` ou si `pragma experimental "ABIEncoderV2";` est utilisé.

Si je retourne un `enum`, je ne reçois que les `integer` avec web3.js. Comment avoir les noms associés ?

Les Enums ne sont pas encore supportés par l'ABI, juste par Solidity. Vous devrez faire la correspondance vous même, mais nous fournirons probablement des outils plus tard.

Les variables d'état peuvent-elles être initialisées à la déclaration ?

Oui, possible pour tous les types (même les structs). Cependant, là encore, un tableau devra être de taille statique pour ce faire.

Examples :

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    struct S {
        uint a;
        uint b;
    }

    S public x = S(1, 2);
    string name = "Ada";
    string[4] adaArr = ["This", "is", "an", "array"];
}

contract D {
    C c = new C();
}
```

Comment marchent les `struct` ?

Regardez struct_and_for_loop_tester.sol.

Comment marche la boucle `for` ?

Très similaire au JS, comme dans l'exemple ci-dessous :

```
for (uint i = 0; i < a.length; i ++) { a[i] = i; }
```

Regardez struct_and_for_loop_tester.sol.

Quelles sont les exemples de fonctions de manipulation de `string` (`substring`, `indexOf`, `charAt`, etc) ?

Il existe quelques fonctions de manipulation de strings comme stringUtils.sol qui seront étendues dans le futur. En addition, Arachnid a écrit solidity-stringutils.

Pour l'instant, si vous voulez modifier une string (même seulement pour connaitre a taille), vous devriez y convertir en `bytes` (représentation octale) d'abord :

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    string s;

    function append(byte c) public {
        bytes(s).push(c);
    }

    function set(uint i, byte c) public {
        bytes(s)[i] = c;
    }
}
```

Puis-je concaténer 2 strings ?

Oui, vous pouvez utiliser `abi.encodePacked` :

```
pragma solidity >=0.4.0 <0.6.0;

library ConcatHelper {
    function concat(bytes memory a, bytes memory b)
             internal pure returns (bytes memory) {
        return abi.encodePacked(a, b);
    }
}
```

Pourquoi la foncttion bas-niveau `.call()` est moins recommendable que d'instancier un contrat dans une variable (`ContractB b;`) puis d'exécuter ses fonctions (`b.doSomething();`) ?

Si vous utilisez des fonctions, le compilateur vous dira si les types ou vos arguments ne correspondent pas, si la fonction n'existe pas ou n'est pas visible et il encodera les arguments pour vous.

Regardez ping.sol et pong.sol.

En retournant par exemple un `uint`, est-il possible de retourner `undefined` , « null » ou une valeur similaire ?

Cela n'est pas possible, car tous les types utilisent toute la plage de valeurs binaires possibles.

Vous avez la possibilité de `throw` en cas d'erreur, ce qui annulera également l'ensemble de la transaction et pourrait être une bonne idée si vous avez rencontré une situation inattendue.

Si vous ne voulez pas annuler, vous pouvez retourner une seconde valeur :

```
pragma solidity >0.4.23 <0.6.0;

contract C {
    uint[] counters;

    function getCounter(uint index)
        public
        view
        returns (uint counter, bool error) {
            if (index >= counters.length)
                return (0, true);
            else
                return (counters[index], false);
    }

    function checkCounter(uint index) public view {
        (uint counter, bool error) = getCounter(index);
        if (error) {
            // Gère l'erreur
        } else {
            // Fait quelque chose avec counter.
            require(counter > 7, "Invalid counter value");
        }
    }
}
```

Les commentaires sont-ils déployés avec le contrat et/ou augmentent t'ils le coût du déploiement (gas) ?

Non, tout ce qui n'est pas nécessaire à l'exécution est retiré à la compilation. Ça inclut, entre autres, les commentaires, noms de variables et noms de types.

Que se passe t'il si j'envoie des Ether lors de l'appel de fonction à un contrat ?

Le montant s'ajoute à la `balance` du contrat, tout comme l'envoi d'Ether à la création. Vous ne pouvez envoyer une transaction comprenant de l'Ether qu'à une fonction ayant le modifieur `payable`, sinon une exception interromp l'exécution.

Est-il possible d'avoir un reçu de transaction pour une transaction contrat à contrat ?

Non, un appel de fonction d'un contrat à un autre ne crée pas sa propre transaction, vous devez regarder dans la transaction initiatrice. C'est aussi la raison pour laquelle plusieurs explorateurs de blocs n'affichent pas correctement l'Ether envoyé entre les contrats.

4.14.2 Questions Avancées

Comment obtenir un nombre aléatoire dans un contrat ? (implémenter un contrat de jeu de hasard automatisé)

Obtenir de l'aléatoire correctement est souvent la partie cruciale d'un projet de crypto et la plupart des échecs résultent de mauvais générateurs de nombres aléatoires.

Si vous ne voulez pas qu'il soit sûr, vous construisez quelque chose de similaire au coin flipper mais sinon, utilisez plutôt un contrat qui fournit un l'aléatoire, comme le RANDAO.

Obtenir la valeur de retour d'une fonction non constante d'un autre contrat

Le point principal est que le contrat appelant doit connaître la fonction qu'il a l'intention d'appeler.

Regardez ping.sol et pong.sol.

Comment créer des tableaux à 2 dimensions ?

Regardez 2D_array.sol.

Notez que remplir un carré 10x10 de `uint8` + création de contrat a pris plus de `800,000` gas au moment d'écrire ces lignes. 17x17 aura pris « 2 000 000 000 » de gas. La limite étant fixée à 3,14 millions. . . . eh bien, il y a un plafond assez bas pour ce que vous pouvez créer correctement maintenant.

Notez que simplement « créer » le tableau est gratuit, les coûts sont dans son remplissage.

Note2 : L'optimisation de l'accès au stockage peut réduire considérablement les coûts du gas, car 32 valeurs `uint8` peuvent être stockées dans un seul emplacement. Le problème est que ces optimisations ne fonctionnent mal avec les boucles et ont également un problème avec la vérification des limites (bound-checking). Vous obtiendrez de bien meilleurs résultats de ce côté là dans le futur, normalement.

Qu'arrive t'il à un mapping de `struct``s quand il est copié dans une ``struct` ?

C'est une question très intéressante. Supposons que nous ayons un environnement de contrat configuré comme tel :

```
struct User {
    mapping(string => string) comments;
}

function somefunction public {
   User user1;
   user1.comments["Hello"] = "World";
   User user2 = user1;
}
```

Dans ce cas, le mappage de la structure copiée dans `user2` est ignoré car il n'y a pas de « liste des clés mappées ». Il n'est donc pas possible de savoir quelles valeurs doivent être copiées.

Comment initialiser un contrat avec un montant spécifique de wei ?

Actuellement, l'approche est un peu sale, mais il n'y a pas grand-chose à faire pour l'améliorer. Dans le cas d'un `contract A` appelant une nouvelle instance du `contract B`, les parenthèses doivent être utilisées autour du `new B` parce que `B.value` renvoie à un membre de `B` appelé `value`. Vous devrez vous assurer que les deux contrats sont conscients l'un de l'autre et que « contract B » a un constructor `payable`. Dans cet exemple :

```
pragma solidity >0.4.99 <0.6.0;

contract B {
    constructor() public payable {}
}

contract A {
    B child;

    function test() public {
        child = (new B).value(10)(); // construit un nouveau B avec 10 wei
    }
}
```

Une fonction de contrat peut-elle prendre en entrée un tableau à 2 dimensions ?

Si vous voulez passer des tableaux bidimensionnels entre des fonctions non internes, vous avez très probablement besoin d'utiliser `pragma experimental "ABIEncoderV2";`.

Quelle est la relation entre `bytes32` et `string` ? Comment se fait-il que `bytes32 somevar = "stringliteral";` fonctionne et que signifie la valeur hexadécimale de 32 octets stockée ?

Le type `bytes32` peut contenir 32 octets (bruts). Dans l'affectation `bytes32 somevar = "stringliteral";`, le texte de la `string` est interprété dans sa forme d'octets bruts et si vous consultez `somevar` et voyez une valeur hexa sur 32 octets, c'est juste `"stringliteral` en hexa.

Le type « bytes » est similaire, mais peut changer sa longueur.

Enfin, `string` est fondamentalement identique à `bytes` seulement qu'il est supposé contenir l'encodage UTF-8 d'une chaîne de caractères valide. Puisque `string` stocke les données en encodage UTF-8, il est assez coûteux de calculer le nombre de caractères dans la chaîne (l'encodage de certains caractères prennant plus d'un octet). Pour cette raison, `string s ; s.length` n'est pas encore supporté ni même l'accès par index `s[2]`. Mais si vous voulez accéder à l'encodage d'octets de bas niveau de la chaîne, vous pouvez utiliser `bytes(s).length` et `bytes(s)[2]` ce qui aura pour résultat le nombre d'octets dans le codage UTF-8 de la chaîne (pas le nombre de caractères) et le second octet (pas forcément caractère) de la chaîne encodée UTF-8, respectivement.

Un contrat peut-il passer un tableau (taille statique) ou une chaîne de caractères ou encore un `bytes` (taille dynamique) à un autre contrat ?

Bien sûr. Veillez à ce que si vous franchissez la limite mémoire / stockage, des copies indépendantes soient créées. :

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

L'appel à `g(x)``n'aura pas d'effet sur ``x` car il doit créer une copie indépendante de la valeur de stockage en mémoire. Par contre, `h(x)` modifie `x` avec succès parce que seule une référence et non une copie est transmise.

Parfois, quand j'essaie de changer la longueur d'un tableau avec par exemple `arrayname.length = 7;`, j'obtiens une erreur de compilation `Value must be an lvalue`. Pourquoi ?

Vous pouvez redimensionner un tableau dynamique en storage (c'est-à-dire un tableau déclaré au niveau du contrat) avec `arrayname.length = <une nouvelle longueur>;`. Si vous obtenez l'erreur « lvalue », vous faites probablement l'une des deux choses suivantes.

1. Vous essayez peut-être de redimensionner un tableau en « memory », ou
2. Vous essayez peut-être de redimensionner un tableau non dynamique.

```
pragma solidity >=0.4.18 <0.6.0;

// Ceci ne compile pas
contract C {
    int8[] dynamicStorageArray;
    int8[5] fixedStorageArray;

    function f() public {
        int8[] memory memArr;        // Cas 1
        memArr.length++;             // illégal

        int8[5] storage storageArr = fixedStorageArray;   // Cas 2
        storageArr.length++;                             // illégal

        int8[] storage storageArr2 = dynamicStorageArray;
        storageArr2.length++;                       // légal


    }
}
```

Note : En Solidity, les dimensions des tableaux sont déclarées à l'envers par rapport à la façon dont vous pourriez être habitué à les déclarer en C ou Java, mais elles sont accessibles comme en C ou en Java. Par exemple, `int8[][5] somearray;` sont 5 tableaux dynamiques de `int8`. La raison en est que `T[5]` est toujours un tableau de 5 `T`, peu importe si `T` lui-même est un tableau ou non (ce n'est pas le cas en C ou Java).

Est-il possible de retourner un tableau de chaînes de caractères (`string[]`) à partir d'une fonction Solidity ?

Uniquement lorsque `pragma experimental "ABIEncoderV2";` est utilisé.

Que fait l'étrange vérification suivante dans le contrat Custom Token ?

```
require((balanceOf[_to] + _valeur) >= balanceOf[_to]) ;
```

Les entiers dans Solidity (et la plupart des autres langages de programmation bas-niveau) sont limités à une certaine plage. Pour `uint256`, il s'agit de `0` jusqu'à `2**256 - 1`. Si le résultat d'une opération quelconque sur ces nombres ne correspond pas à cette plage, il est tronqué. Ces troncatures peuvent avoir de graves conséquences, donc un code comme celui ci est nécessaire pour éviter certaines attaques.

Pourquoi les conversions explicites entre les `bytes` de taille fixe et les types `int` échouent-elles ?

Depuis la version 0.5.0, les conversions explicites entre les tableaux d'octets de taille fixe et les entiers ne sont autorisées que si les deux types ont la même taille. Cela permet d'éviter les comportements inattendus lors de la troncation ou du bourrage. De telles conversions sont encore possibles, mais des conversions intermédiaires explicites sont nécessaires pour rendre visible la convention de troncature et de bourrage souhaitée. Voir *Conversions entre les types élémentaires* pour une explication complète et des exemples.

Pourquoi les nombres littéraux (dans une `string`) ne peuvent-ils pas être convertis en types `bytes` de taille fixe ?

Depuis la version 0.5.0, seuls les nombres hexadécimaux peuvent être convertis en bytes de taille fixe et uniquement si le nombre de chiffres hexadécimaux correspond à la taille du type. Voir types-conversion-litterals pour une explication complète et des exemples.

Autres questions ?

Si vous avez d'autres questions ou si vous ne trouvez pas la réponse à la votre ici, n'hésitez pas à nous contacter, en anglais, sur gitter ou à nous faire parvenir un problème sur github.

Index

R

S

T

U

V

W

Y