Solidity Documentation

Versão 0.4.18

Ethereum

12 fev, 2018

Conteúdo

1 Links úteis **3**

2 Integraçoes Disponíveis para Solidity **5**

3 Ferramentas Solidity **7**

4 Interpretador e Dicionários de terceiros para Solidity **9**

5 Documentação da Linguagem **11**

6 Conteúdo **13**

6.1 Introdução aos Smart Contracts . . . 13
6.2 Instalando Solidity . . . 20
6.3 Solidity by Example . . . 24
6.4 Solidity em Profundidade . . . 34
6.5 Considerações de Segurança . . . 113
6.6 Usando o compilador . . . 117
6.7 Especificação da interface binária do aplicativo . . . 122
6.8 Style Guide . . . 129
6.9 Common Patterns . . . 142
6.10 Lista de Bugs Conhecidos . . . 147
6.11 Contributing . . . 151
6.12 Frequently Asked Questions . . . 152

Solidity é uma linguagem de programação de alto nível, orientada a contratos, com a síntese parecida com a de JavaScript e desenhada para ser executada na Máquina Virtual Ethereum (EVM).

Solidity é estatisticamente tipada, suporta herança, bibliotecas e tipos complexos definidos pelo usuário entre outras características.

Como você verá, é possível criar contratos para votação, vaquinhas (crowdfunding), leilões às cegas, carteiras multi-assinadas e mais.

Nota: A melhor maneira de experimentar com Solidity é utilizando Remix (pode demorar para carregar, seja paciente).

CAPÍTULO 1

Links úteis

- Ethereum
- Lista de mudanças
- Story Backlog
- Código Fonte
- Ethereum Stackexchange
- Gitter Chat

CAPÍTULO 2

Integraçoes Disponíveis para Solidity

- **Remix** IDE baseada em Browser com compilador integrado e ambiente de tempo de execução Solidity sem componentes de servidor.
- **IntelliJ IDEA plugin** Plugin Solidity para IntelliJ IDEA (e todas as outras IDEs JetBrains)
- **Extensão para Visual Studio** Plugin Solidity para Microsoft Visual Studio que inclui o compilador Solidity.
- **Pacote para SublimeText — síntese da linguagem Solidity** Marcação de síntese Solidity para o editor de texto SublimeText.
- **Etheratom** Plugin para o editor Atom que conta com marcação de síntese, compilação e ambiente de tempo de execução (Compatível com backend node & VM).
- **Atom Solidity Linter** Plugin para o editor Atom para Solidity linting.
- **Atom Solium Linter** Solidty linter configurável para Atom utilizando Solium como base.
- **Solium** Lint de linha de comando para Solidity que segue estritamente as regras descritas no Guia de Estilos para Solidity.
- **Extensão para Visual Studio Code** Plugin de Solidity para o Microsoft Visual Studio Code que inclui marcação de síntese e o compilador Solidity.
- **Emacs Solidity** Plugin para o editor Emacs que provê marcação de síntese e aviso de erros de compilação.
- **Vim Solidity** Plugin para o editor Vim que provê marcação de síntese.
- **Vim Syntastic** Plugin para o editor Vim aviso de erros de compilação.

Descontinuado:

- **Mix IDE** IDE baseada em para desenhar, debugar e testar smart contracts escritos em Solidity.
- **Ethereum Studio** IDE Web especializada que também provê acesso à linha de comando completa do ambiente Ethereum.

CAPÍTULO 3

Ferramentas Solidity

- **Dapp** Ferramenta de build, gerenciador de pacotes e assistente de publicação para Solidity.
- **Solidity REPL** Experimente Solidity instantaneamente através da linha de comando Solidity.
- **solgraph** Ferramenta para visualizar o fluxo de controle e mostrar potenciais falhas de segurança no seu contrato inteligente Solidity.
- **evmdis** EVM Disassembler que realiza análise estática no código para garantir um maior nível de abstração em comparação com operações EVM puras.
- **Doxity** Gerador de Documentação para Solidity.

CAPÍTULO 4

Interpretador e Dicionários de terceiros para Solidity

- **solidity-parser** Interpretador de Solidity para JavaScript
- **Solidity Grammar for ANTLR 4** Dicionário de Solidity para o interpretador ANTLR 4.

CAPÍTULO 5

Documentação da Linguagem

Nas próximas páginas vamos ver um *contrato inteligente simples* escrito em Solidity seguido de conceitos básicos sobre *blockchains* e a *Máquina Virtual Ethereum*.

A próxima sessão vai explicar várias *funcionalidades* do Solidity através de *exemplos de contratos úteis* Lembre-se que você pode testar os contratos no seu browser!

A última e mais extensa seção vai cobrir todos os aspectos do Solidity profundamente.

Se você ainda tiver dúvidas você pode procurar ou perguntar no site do Ethereum Stackexchange ou acessar nosso canal gitter. Ideias para melhorar o Solidity ou essa documentação são sempre bem vindas!

Veja também a versão em Russo ( ).

CAPÍTULO 6

Conteúdo

Palavras Chave, Página de Busca

6.1 Introdução aos Smart Contracts

6.1.1 Um simples Smart Contract

Deixe-nos começar com o exemplo mais básico. Sem problemas se você não entender tudo neste momento; Iremos entrar em maiores detalhes depois.

Armazenamento

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

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

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

A primeira linha simplesmente nos diz que o código fonte foi escrito para a versão 0.4.0 do Solidity ou mais recente e não quebra sua funcionalidade (até, mas não incluindo a versão 0.5.0). Isso é para garantir que o contrato não se comporte de forma diferente com uma nova versão do compilador. A palavra chave `pragma` é chamada desta maneira, porque, no geral pragmas são instruções para o compilador sobre como tratar o código fonte (por exemplo *pragma once* <*https://en.wikipedia.org/wiki/Pragma_once*>).

Um contrato, no conceito do Solidity, é um conjunto de códigos (*functions*) e dados (*state*), que residem em um específico endereço na rede blockchain Ethereum. A linha `uint storedData;` declara uma variável de estado chamada `storedData` do tipo `uint` (integer não assinado de 256 bits). Você pode pensar nisso como um único slot em um banco de dados que pode ser consultado e alterado chamando funções do código que gerencia o banco de dados. No caso do Ethereum, ele é sempre o contrato proprietário. E neste caso, as funções `set` e `get` podem ser usadas para modificar ou recuperar o valor da variável.

Para acessar uma variável de estado, você não precisa do prefixo `this.`, como é comum em outras linguagens.

Este contrato ainda não faz muita coisa (devido à infra-estrutura construida pelo Ethereum), além de permitir que qualquer um possa armazenar um simples número que é acessível por qualquer um no mundo sem uma maneira (viável) de impedir você de publicar este número. Naturalmente, qualquer um pode somente chamar novamente a função `set`, com um valor diferente e sobrescrever seu número original, mas o número permanecerá armazenado no histórico do blockchain. Mais tarde, iremos ver como você pode impor restrições de acesso, de maneira que somente você possa alterar este número.

Nota: Todos os identificadores (nomes de contratos, nomes de funções e nomes de varáveis) estão restritos à tabela de caracteres ASCII. É possível armazenar dados codificados UTF-8 em variáveis de string.

Aviso: Seja cuidadoso com o uso do texto Unicode, pois caracteres similares (ou mesmo idênticos) podem ter pontos de código diferentes e, como tal, serão codificados como uma matriz de bytes diferente.

Exemplo de Sub Moeda

O contrato seguinte irá implementar a forma mais simples de uma criptomoeda. É possível gerar moedas do nada, mas somente a pessoa que criou o contrato terá a capacidade de fazer isto (é trivial implementar um esquema de emissão diferente). Além disso, qualquer um pode enviar moedas para outros, sem necessidade de se registrar com usuário e senha. Tudo o que você precisa é de um par de chaves Ethereum.

```
pragma solidity ^0.4.0;

contract Coin {
    // The keyword "public" makes those variables
    // readable from outside.
    address public minter;
    mapping (address => uint) public balances;

    // Events allow light clients to react on
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    function Coin() {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) {
```

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

Este contrato introduz alguns novos conceitos, Passaremos por eles passo à passo.

A linha `address public minter;` declara uma variável de estado do tipo endereço (address), que é publicamente acessível. O tipo `address` é um valor de 160 bits que não permite nenhuma operação aritmética. É adequada para armazenar endereços de contratos ou pares de chaves pertencentes à entidades externas. A palavra-chave `public` automaticamente gera uma função que lhe permite acessar o valor atual do estado da variável. Sem esta palavra-chave, outros contratos podem não ter acesso à variável. A função irá aparecer dessa maneira:

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

Naturalmente, adicionando-se a função, exatamente como está, não irá funcionar porque nós devemos ter a função e uma variável de estado com o mesmo nome, mas felizmente, você pegou a iedia - o compilador irá entender para você.

A próxima linha, `mapping (address => uint) public balances;`, também cria uma variável públca de estado, mas é um tipo de dado mais complexo. O tipo mapa endereça integers não assinadas. Mappings podem ser vistas como hash tables

Praticamente inicializado de tal forma que cada chave possível existe e é mapeado para um valor que representação em bytes é toda em zeros. Esta analogia não vai muito longe, porém, como não é possível obter uma lista de todas as chaves de um mapeamento nem uma lista de todos os valores. Tenha em mente (ou melhor, mantenha uma lista ou use um tipo de dados mais avançado) o que você adicionou no mapeamento ou use em um contexto onde isto não é necessário, como este. A função getter *getter function* criada pela palavra chave `public` é um pouco mais complexa neste caso. Parece aproximadamente como o seguinte:

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

Como você pode ver, você pode usar esta função para facilmente pesquisar o saldo de uma conta simples.

A linha `event Sent(address from, address to, uint amount);` declara um auto-nomeado «event» que é eliminado na última linha da função `send`. Interfaces de usuário (assim como uma aplicação de servidor, naturalmente) pode escutar estes eventos sendo eliminados no blockchain sem muito custo. Assim que é eliminado, o listener irá receber o argumento `from`, `to` e `amount`, que torna fácil rastrear transações. Para escutar estes eventos, você pode usar

```
   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));
       }
   })


Perceba como a função ``balances``, gerada automaticamente, é chamada a partir
do interface do usuário.
```

A função especial `Coin` é o contrutor que é executado durante a criação do contrato e não pode ser chamada depois. Ela armazena permanentemente o endereço da pessoa que criou o contrato; `msg` (junto com `tx` e `block`) é uma variável global mágica que contém algumas propriedades que permitem acessar o blockchain. `msg.sender` é sempre o endereço onde a função corrente (externa) é originada.

Finalmente, a função que era realmente encerrar com o contrato e pode ser chamada pelos usuários e contratos são `mint` and `send`. Se `mint` é chamada por alguém que não a conta que criou o contrato, nada irá acontecer. Por outro lado, `send` pode ser usado por qualquer um (que já tenha alguma dessa moeda) para enviar moedas para alguém. Perceba que se você usar este contrato para enviar moedas para um endereço, você não irá ver nada quando você olhar para este endereço através de um explorador blockchain, porque o fato de você ter enviado as moedas e os saldos alterados só são armazenados no armazenamento de dados deste Contrato de moeda particular. Com o uso de eventos, é relativamente fácil criar um «blockchain explorer» que rastreia transações e saldos da sua nova moeda.

6.1.2 Princípios do Blockchain

Blockchain como um conceito não é muito difícil de entender por programadores. A razão é que a maioria das complicações (mining, hashing, elliptic-curve cryptography, peer-to-peer networks, etc.) é justamente para prover um conjunto de características e promessas. Uma vez que você aceitas estas características como certas, você não precisa se preocupar com a tecnologia adjacente - ou você precisa saber como a Amazon´s AWZ trabalham internamente para usá-las?

Transações

O blockchain é uma base de dados transacional globalmente distribuída. Isso significa que qualquer um pode ler as entradas do banco de dados, somente por participar da rede. Se você deseja mudar alguma coisa no banco de dados, você tem que criar uma transação que precisa ser aceita por todos os demais (participantes).

A palava transação implica que a mudança que você quer fazer (assumindo que você queira mudar dois valores ao mesmo tempo) não é feita nem aplicada completamente. Além disso, enquanto sua transação é aplicada ao banco de dados, nenhuma outra transação pode alterá-la.

Por exemplo, imagine uma tabela que lista os saldos de todas as contas em um moeda eletrônica. Se uma transferência de uma conta para outra for solicitada, a natureza transacional do banco de dados garante que, se o valor for subtraído de uma conta, é sempre adicionado à outra conta. Se, por qualquer motivo, não for possível adicionar o valor à conta destino, a conta de origem também não será modificada.

Por exemplo, imagine uma tabela que lista os saldos de todas as contas de uma moeda eletrônica. Se uma transferência de uma conta para outra for solicitada, a natureza transacional do banco de dados garante que, se o valor for subtraído de uma conta, é sempre adicionado à outra conta. Se, por qualquer motivo, não for possível adicionar o valor à conta destino, a conta de origem também não será modificada.

Além disso, uma transação sempre é criptograficamente assinada pelo remetente (seu criador). Isso torna direto proteger o acesso a modificações específicas da base de dados. No exemplo da moeda eletrônica, uma verificação simples garante que apenas a pessoa que possua as chaves da conta possa transferir dinheiro com ela.

Blocks

Um dos principais obstáculos a superar é o que, em termos Bitcoin, é chamado de «ataque de dupla despesa»: O que acontece se ocorrer duas transações na rede que querem esvaziar uma conta, um chamado conflito?

A resposta abstrata à isso é que você não precisa se preocupar. Uma ordem das transações será selecionada para você, as transações serão empacotadas no que é chamado de «bloco» (block) e então eles serão executados e distribuídos entre todos os nós participantes. Se duas transações se contradizem, o que acaba sendo a segunda transação será rejeitada e não se tornará parte do bloco.

Esses blocos formam uma seqüência linear no tempo e é aí que deriva o termo «bloco em cadeia» (blockchain). Os blocos são adicionados à cadeia em intervalos bastante regulares - para o Ethereum é aproximadamente a cada 17 segundos.

Como parte do «mecanismo de seleção de pedidos» (que é chamado de (mining) «mineração»), pode acontecer que os blocos são revertidos de tempos em tempos, mas apenas na «ponta» da corrente. Quanto mais blocos são adicionados no topo, menos provável é. Então, pode ser que suas transações sejam revertidas e até mesmo removidas do blockchain, mas quanto mais você aguardar, menor a probabilidade.

6.1.3 A Máquina Virtual Ethereum (EVM)

Visão Geral

A Máquina Virtual Ethereum (Ethereum Virtual Machine - EVM) é o ambiente em tempo de execução para contratos inteligentes (smart contracts) no Ethereum. Não é apenas sandbox, mas realmente isolado completamente, o que significa que o código está sendo executado pelo EVM não tem acesso à rede, sistema de arquivos ou outros processos. Os contratos inteligentes (smart contracts) têm acesso limitado à outros contratos inteligentes.

Contas (Accounts)

Existem dois tipos de contas (accounts) no Ethereum que compartilham o mesmo espaço de endereço: **External accounts** (contas externas) que são controladas por pares de chaves público-privadas (isto é, por humanos) e **contract accounts** (contas contratuais) que são controladas pelo código armazenado junto com a conta.

O endereço de uma conta externa é determinado a partir da chave pública enquanto o endereço de um contrato é determinado no momento em que o contrato é criado (é derivado do endereço do criador e do número das transações enviadas a partir desse endereço, o chamado «nonce»).

Independentemente ou não do código de armazenamento das contas, os dois tipos são tratados igualmente pelo EVM.

Cada conta possui um valor-chave persistente mapeado de 256-bits chamado **storage** de 256-bits

Além disso, cada conta tem **balance** (saldo) em Ether (em «Wei» para ser exato) que pode ser modificado enviando transações que incluem Ether.

Transações (Transactions)

Cada transação (transaction) é uma mensagem que é enviada de uma conta para outra conta (que pode ser a mesma ou uma conta especial zero-account, veja abaixo). Pode incluir dados binários (sua carga útil) e Ether.

Se a conta alvo contiver um código, esse código é executado e a carga útil é fornecida como dados de entrada.

Se a conta alvo for do tipo conta zero (a conta com o endereço `0`), a transação cria um novo contrato **new contract**.

Como já mencionado, o endereço desse contrato não é o endereço zero, mas um endereço derivado do remetente e o número de transações enviadas (o «nonce»). A carga útil de tal transação de criação de contrato é considerada como sendo Bytecode EVM e executado. A saída desta execução é permanentemente armazenada como o código do contrato. Isso significa que, para criar um contrato, você não envia o código real do contrato, mas, de fato, o código que retorna esse código.

Gas

Após a criação, cada transação é cobrada com uma certa quantidade de **gas**, cujo objetivo é limitar a quantidade de trabalho que é necessária para executar a transação e para pagar esta execução. Enquanto o EVM executa o transação, o gás é gradualmente usado de acordo com regras específicas.

O **gas price** é um valor definido pelo criador da transação, quem tem que pagar o o resultado de `gas_price * gas` antes da conta de envio. Se algum gás for deixado após a execução, ele será reembolsado da mesma maneira.

Se o gás for esgotado em qualquer ponto (isto é, o saldo disponível de Gas fica negativo), é desencadeada uma exceção em gas (out-of-gas), que reverte todas as modificações feito para o estado no quadro de chamada atual.

Armazenamento, Memória e Pilha

Cada conta tem uma área de memória persistente chamada **storage**. Storage é um valor-chave que mapeia palavras de 256-bits para palavras de 256-bits. Não é possível enumerar o armazenamento dentro de um contrato e é comparativamente caro para ler e ainda mais, para modificar armazenamento. Um contrato não pode ler nem escrever em qualquer armazenamento separado por conta própria.

A segunda área de memória é chamada de **memory**, de que um contrato obtém uma instância recém autorizada para cada chamada de mensagem. A memória é linear e pode ser endereçada no nível do byte, mas as leituras são limitadas a uma largura de 256 bits, enquanto escrever pode ser de 8 bits ou 256 bits de largura. A memória é expandida por uma palavra (256 bits), enquanto acessando (quer lendo ou escrevendo) uma palavra de memória anteriormente intacta (ou seja, qualquer deslocamento dentro de uma palavra). No momento da expansão, o custo em Gas deve ser pago. A memória é mais cara a medida que cresce (em escala quadrática).

O EVM não é uma register machine mas uma stack machine, portanto todas os comandos são realizados em uma área chamada **stack**. Tem o tamanho máximo de 1024 elementos e contém palavras de 256 bits. O acesso à pilha (stack) é limitado ao topo (top end) na seguinte maneira: É possível copiar um dos os 16 elementos superiores para o topo da pilha ou trocar o elemento superior com um dos 16 elementos abaixo. Todas as outras operações levam os dois primeiros (ou uma, ou mais, dependendo de a operação) elementos da pilha e empurre o resultado para a pilha. Naturalmente é possível mover elementos de pilha para armazenamento ou memória, mas não é possível acessar elementos arbitrários mais profundos na pilha sem primeiro remover o topo da pilha.

Lista de Instruções

A lista de instruções do EVM é mantido no mínimo para evitar implementações incorretas que podem causar problemas de consenso. Todas as instruções operam no tipo básico de dados, palavra de 256-bits. As operações mais usuais de aritmética, bit, lógica e comparação estão presentes. Desvios condicionais e não-condicionais são possíveis. Além disso, contratos podem acessar propriedades relevantes do bloco atual como seu número e carimbo de tempo (timestamp).

Chamadas por Mensagem

Contratos podem chamar outros contratos ou enviar Ether para contas não-contrato através de chamadas de mensagens. As chamadas de mensagens são semelhantes para as transações, na medida em que eles têm uma origem, um destino, carga útil de dados, dados de Ether, Gas e informações de retorno. Na verdade, cada transação consiste em uma chamada de mensagem de nível superior que, por sua vez, pode criar mais chamadas de mensagens.

Um contrato pode decidir quanto do **gas** restante deve ser enviado com a chamada de mensagem interna e quanto ele deseja reter. Se ocorrer uma exceção out-of-gas (sem gas) na chamada interna (ou qualquer outra exceção), isso será sinalizado por um valor de erro colocado na pilha. Neste caso, somente o gas enviado junto com a chamada é consumido. No Solidity, o contrato de chamada causa uma exceção manual por padrão em tais situações, de modo que as exceções «expandam» a pilha de chamadas.

Como já dissemos, o contrato chamado (que pode ser o mesmo que o chamador) receberá uma instância de memória recém autorizada e terá acesso à chamada carga útil - que será fornecida em uma área separada chamada **calldata**. Depois de terminar a execução, pode retornar dados que serão armazenados em uma localização na memória do chamador pré-alocada pelo chamador.

As chamadas são limitadas (**limited**) para uma profundidade de 1024, o que significa que para operações mais complexas, loops devem ser preferidos em relação a chamadas recursivas.

Delegatecall / Callcode and Libraries

Existe uma variante especial de chamada de mensagem, chamada **delegatecall** que é identica à chamada de mensagem, exceto pelo fato que o código no endereço de destino é executado no contexto do contrato chamado e `msg.sender` e `msg.value` não alteram seus valores.

Isto significa que um contrato pode dinamicamente carregar código de diferentes endereços em tempo de execução. Armazenamento (Storage), Endereço Atual (current address) e saldo (Balance) podem ainda se referir ao contrato chamador, apenas o código é retirado do endereço chamado.

Isto torna possível a implementação da característica de «library» no Solidity: Códigos reusáveis de «library» podem ser aplicados ao armazenamento (storage) dos contratos, normalmente para implementar estrutura de dsos complexas.

Logs

É possível armazenar dados em uma estrutura de dados especialmente indexada que mapeia até o nível do bloco. Esta característica chamada **logs** é usada pelo Solidity para implementar eventos (**events**). Os contratos não podem acessar os dados de log depois de terem sido criados, mas eles pode ser acessado de forma eficiente, fora da cadeia de blocos. Desde que algumas partes dos dados do log são armazenados no bloom filters, é possível pesquisar por estes dados de uma maneira eficiente e criptograficamente segura, sendo assim, outros participantes (clientes) da rede que não baixarem o blockchain todo («light clients»), podem, asinda, assim, encontrar estes logs.

Create

Contratos podem ainda criar outros contratos usando um opcode especial (em geral, eles não chamam o zero address). A única diferença entre esses **create calls** e uma chamada de mensagem normal é que a carga útil (payload) é executada e o resultado é armazenado como code e o chamador/criador recebe o endereço deste novo contrato na pilha (stack).

Self-destruct

A única maneira de um código ser removido do blockchain é quando o contrato que ele endereça realizar a operação `selfdestruct`. O Ether remanescente, armazenado neste endereço, é enviado para um destino designado e então o armazenamento e o código é removido.

Aviso: mesmo quando o código do contrato não contém uma chamada para `selfdestruct`, ele pode ainda realizar esta operação chamando `delegatecall` or `callcode`.

Nota: A poda de contratos antigos pode ou não ser implementada pelos Clientes Ethereum. Além disso, os nós de arquivo podem optar por manter o armazenamento do contrato e código indefinidamente.

Nota: Atualmente **external accounts** não podem ser removidas do estado.

6.2 Instalando Solidity

6.2.1 Versionamento

As versões do Solidity seguem a semantic versioning e em adição aos releases, **nightly development builds** são sempre disponibilizadas. As montagens noturnas não tem garantia de funcionamento e apesar dos melhores esforços, elas podem conter mudanças não documentadas ou quebradas. Nós recomendamos usar a última versão. Os pacotes de instação abaixo irão usar a última versão.

6.2.2 Remix

Se você deseja usar o Solicity para small contracts, você pode tentar usar o Remix que não necessita instalação. Se você deseja usar sem conexão com a Internet, você pode ir para https://github.com/ethereum/browser-solidity/tree/gh-pages e baixar o arquivo zipado (.ZIP file), conforme explicado nesta página.

6.2.3 npm / Node.js

Este é provavelmente o meio mais conveniente e portável de instalar o Solidity localmente. Uma bibiloteca de JavaScript independente da plataforma é provida através da compilação da fonte C++ dentro do JavaScript usando Emscripten. Pode ser usado nos projetos diretamente (como o Remix). Por gentileza, consulte o diretório solc-js para maiores instruções.

Ele também contém uma ferramenta de linha de comando chamada *solcjs*, que pode ser instalada via nmp:

```
npm install -g solc
```

Nota: As opções da linha de comando *solcjs* não são compatíveis com *solc* e ferramentas (como o Geth) esperando que o comportamento de *solc* não irão funcionar com *solcjs*.

6.2.4 Docker

Nós fornecemos builders atualizadas para o date docker para o compilador. O repositório `stable` contém versões liberadas enquanto o repositório `nightly` contém mudanças potencialmente instáveis no segmento de desenvolvimento.

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

Atualmente, a imagem Docker contém somente o compilador executável, então você terá algum trabalho adicional para linkar a fonte e os diretórios de saída.

6.2.5 Binary Packages

Os pacotes binários do olicity estão disponíveis em solidity/releases.

Nós também temos PPAs para Ubuntu. Para a última versão estável.

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

Se você quiser a versão de desenvolvimento mais avançada:

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

Nós também estamos liberando um snap package,, que é instalável em todas as supported Linux distros. Para instalar a última versão estável do solc:

```
sudo snap install solc
```

Ou se voc~e quiser ajudar a testar o solc instável, com as versões mais recentes do ramo de desenvolvimento:

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

Arch Linux também tem pacotes, embora limitados à mais recente versão de desenvolvimento:

```
pacman -S solidity
```

Para Homebres, até o momento, estão faltando os pre-built bottles, seguindo uma migração de Jenkis para TravisCI, mas para Homebrew ainda deverá funcionar como um meio de construir direto da fonte. Iremos re-adicionar brevemente os pre-built bottles.

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

Se você quiser uma versão específica doSolicity, você pode instalar a fórmula Homebrew diretamente do Github.

Veja em solidity.rb commits on Github.

Siga os links do histórico até ter um link de arquivo bruto de um compromisso específico de ‘‘ solidity.rb‘‘.

Instale-o usando `brew`:

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

Gentoo Linux também fornecer um pacote solidity que pode ser instalado usando `emerge`:

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

6.2.6 Construindo à partir do Fonte

Clone o repositório

Para clonar o código fonte, execute o seguinte comando:

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

Se você deseja ajudar à desenvolver o Solidity, você deve derivar o Solidity e adicionar sua derivação pessoal como um segundo remote:

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

O Solicity tem sub-módulos git. Assegure-se de que eles foram carregados adequadamente:

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

Pré-requisitos - MacOS

Para o macOS, assegure-se de que você tenha a última versão do Xcode installed.

Isto contém o Clang C++ compiler, the Xcode IDE e outras ferramentas Apple de desenvolvimento que são requeridas para construir aplicações C++ no OS X. Se você estiver instalado Xcode pela primeira vez, or simplemente instalado uma nova versão, você terá que concordar com a licença de uso antes de poder fazer compilações de linhas de comando:

```
sudo xcodebuild -license accept
```

Nossas compilações do OS X exigem que você install the Homebrew, gerenciador de pacotes para instalar dependências externas. Eis como desinstalar o Homebrew, caso você queira iniciar novamente do início uninstall Homebrew,

Pré-Requisitos - Windows

Você irá necessitar instalar as seguintes dependências para montar a versão do Solicity no Windows:

Software	Notas
Git for Windows	Ferramenta para linha de comando para recuperação dos fontes a partir do GitHub.
CMake	Gerador de Arquivos de compilação entre plataformas
Visual Studio 2015	Compilador C++ e ambiente de desenolvimento

External Dependencies Dependências Externas ——————

Nós agora temos um script «botão único» que instala todas as dependências externas requeridas em macOS, Windows e numerosas Distros Linux. Isto é usado para ser um processo manual multi-passos, mas agora em uma linha.

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

Ou, no Windows:

```
scripts\install_deps.bat
```

Command-Line Build

O projeto Solidity usa CMake para configurar a compilação. Building Solidity é bastante semelhante ao Linux, MacOS e outros Unices:

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

ou ainda mais fácil:

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

Ou ainda para o Windows:

```
mkdir build
cd build
cmake -G "Visual Studio 14 2015 Win64" ..
```

Este último conjunto de instruções deve resultar na criação de **solidity.sln** nesse diretório de compilação. Clicando duas vezes nesse arquivo deve resultar na ativação do Visual Studio. Sugerimos construir uma configuração **RelWithDebugInfo**, mas todos os outros irão funcionar.

Alternativamente, você pode construir para o Windows na linha de comando, dessa maneira:

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

6.2.7 Opções CMake

Se você está interessado quais opções CMake estão disponíveis, execute `cmake .. -LH`.

6.2.8 A sequência de versão em detalhes

A versão de string do Solidity contém quatro partes:

- O número de versão
- Tag de pré-lançamento, normalmente marcado para `develop.YYYY.MM.DD` ou `nightly.YYYY.MM.DD`
- Commit no formato `commit.GITHASH`
- A plataforma tem um número arbitrário de itns, contendo detalhes sobre a plataforma e o compilador.

Se existirem modificações locais, o commit será pós-fixado com `.mod`.

Essas partes são combinadas conforme exigido pela Semver, onde a marca de pré-lançamento da Solidity é igual ao pré-lançamento do Semver e o Commit do Soliddity e a plataforma combinadas compõem os metadados de construção do Semver.

Um exemplo de release:`0.4.8+commit.60cc1668.Emscripten.clang`.

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

6.2.9 Informação importante sobre versionamento

Depois que um lançamento é feito, o nível da versão do patch é superado, porque assumimos que apenas as mudanças de nível de patch seguem. Quando as alterações são mescladas, a versão deve ser superada de acordo com severidades da mudança. Finalmente, uma versão sempre é feita com a versão da construção noturna atual, mas sem o especificador ‘‘ prerelease‘‘.

Exemplo:

0. the 0.4.0 release is made
1. nightly build has a version of 0.4.1 from now on
2. non-breaking changes are introduced - no change in version

3. a breaking change is introduced - version is bumped to 0.5.0
4. the 0.5.0 release is made

Este comportamento funciona bem com a *version pragma*.

6.3 Solidity by Example

6.3.1 Votando

O contrato seguinte é bastante complexo, mas dmonstra muitas das características do Solidity. Ele implementa um contrato de votação. Naturalmente, o problema principal do voto eletrônico é como atribuir direitos de voto a pessoa e como prevenir manipulações. Não serão resolvidos todos os problemas aqui, mas ao menos mostraremos como delegar direito de voto, pode ser feito de tal forma a que a contagem seja **automática e completamente transparente** ao mesmo tempo.

A idéia é criar um contrato por cédula, fornecendo um nome curto para cada opção. Em seguida, o criador do contrato que serve como presidente dará o direito de votar a cada endereço individualmente.

As pessoas por trás dos endereços podem então escolher entre votar elas mesmas ou delegar o voto a outra pessoa de confiança.

No final do tempo de voto, `winningProposal()` retornará a proposta com maior número de votos.

```
pragma solidity ^0.4.11;

    /// @title Votação com delegação.
contract Ballot {
    // Aqui é declarado um novo tipo complexo que será
            // usado pelas variáveis mais tarde.
            // Represntará um votante único.

    struct Voter {
        uint weight; // peso é acumulado por delegação // weight is accumulated by␣
↪delegation
        bool voted;  // se for verdadeiro, aquela pessoa já votou // if true, that␣
↪person already voted
        address delegate; // pessoa a quem será delegado // person delegated to
        uint vote;   // índice do voto proposto // index of the voted proposal
    }

    // Este é um tipo de proposta única

            struct Proposal {
        bytes32 name;   // nome curto (até 32 bytes) // short name (up to 32 bytes)
        uint voteCount; // número de votos acumulados // number of accumulated votes
    }

            address public chairperson;

    // Aqui é declarada a variável de estado que
    // armazena uma estrutura de "Votante" para cada possível endereço.

    mapping(address => Voter) public voters;

            // Uma estrutura de "Proposta" tipo array dinamicamente dimensionada.
```

```
    Proposal[] public proposals;

    /// Criar uma nova cédula para escolher uma das "proposalNames".

            function Ballot(bytes32[] proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // Para cada nome de proposta, criar um novo
        // objeto proposta e adicione este objeto ao
        // fim do array.

        for (uint i = 0; i < proposalNames.length; i++) {

            // "Proposal({...})" cria um objeto temporário
            // e "proposal.push(...)" adiciona este objeto
            // ao fim das "propostas"

            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Dar ao "votante"  o direito de voto nesta cédula.
    // Pode somente ser chamada pelo "Presidente".

    function giveRightToVote(address voter) {

        // Se o argumento de "require" for determinado como "false",
        // é terminado e todas alterações são revertidas assim
        // como o saldo de Ether retorna ao valor antes da operação.
        // É normalmente uma boa ideia usar isto se funções são
        // chamadas incorretamente. Mas fique atento, isso pode
        // também consumir todo o "gas" disponível.
        // (está planejado para ser mudado no futuro).

        require((msg.sender == chairperson) && !voters[voter].voted && (voters[voter].
↪weight == 0));
        voters[voter].weight = 1;
    }

            /// Delegar seu voto ao votante "to".

    function delegate(address to) {
                    // atribuir referência

        Voter storage sender = voters[msg.sender];
        require(!sender.voted);

                    // Auto-delegação não é permitida.

        require(to != msg.sender);

                    // Encaminhar a atribuição desde que "to" também seja atribuido.

        // Em geral, estes tipos de loops são muito perigosos,
```

```
                // porque se demorarem muito tempo executando, podem
                // causar necessidade de mais "gas" do que é disponível
                // para o bloco.
                // Neste caso, a atribuição não será executada, mas,
                // em outras situações, estes loops podem causar o
                // "travamento" completo do contrato.

    while (voters[to].delegate != address(0)) {
        to = voters[to].delegate;

        // Se encontramos um loop na atribuição, não permitido.

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

    // Desde que "sender" é uma referência, este
                // modifica "voters[msg.sender].voted"

                sender.voted = true;
    sender.delegate = to;
    Voter storage delegate = voters[to];
    if (delegate.voted) {
        // Se o atribuido já votou,
                        // some diretamente no número de votos.

                    proposals[delegate.vote].voteCount += sender.weight;
    } else {
        // Se o atribuido ainda não votou,
                        // some ao seu peso.

                    delegate.weight += sender.weight;
    }
}

/// Dê o seu voto (incluindo votos atribuidos a você)
        /// a proposta "proposals[proposal].name".

        function vote(uint proposal) {
    Voter storage sender = voters[msg.sender];
    require(!sender.voted);
    sender.voted = true;
    sender.vote = proposal;

                // Se "proposal" está fora do range do array,
                // será rejeitada automaticamente e todas as
                // alterações revertidas.

    proposals[proposal].voteCount += sender.weight;
}

/// @dev calcula a proposta vencedora levando todos
        /// os votos prévios em consideração.

        function winningProposal() constant
        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;
            }
        }
    }

    /// Chama a função "winningProposal()" para selecionar
            /// o indíce do vencedor contido no array de propostas e
            /// então retorna o nome do vencedor.

            function winnerName() constant
            returns (bytes32 winnerName)
    {
        winnerName = proposals[winningProposal()].name;
    }
}
```

Possíveis Melhorias

Atualmente, são necessárias muitas transações para atribuir os direitos de votar para todos os participantes. Você pode pensar em uma maneira melhor?

6.3.2 Leilão cego

Nesta seção, mostraremos quão fácil é criar um contrato de leilão completamente cego no Ethereum. Começaremos com um leilão aberto onde todos podem ver os lances efetuados e, em seguida, estender esse contrato em um leilão cego onde não é possível ver o lance real até que o período de lances tenha finalizado.

Leilão aberto Simples

A idéia geral do contrato de leilão simples a seguir é que todos podem enviar suas propostas durante o período de licitação. Os lances já incluem envio de dinheiro / ethers, a fim de vincular os licitantes ao seu lance. Se o lance mais alto for superado, o melhor lance anterior recebe seu dinheiro de volta. Após o final do período de licitação, o contrato deve ser chamado manualmente para o beneficiário receber seu dinheiro - contratos não podem ativar-se.

```
pragma solidity ^0.4.11;

contract SimpleAuction {
    // Parâmetros do leilão. Tempos são dados em "Unix timestamps" absolutos␣
↪(segundos desde 01-01-1970)
    // ou períodos de tempo em segundos.

    address public beneficiary;
    uint public auctionStart;
    uint public biddingTime;

    // Estado do leilão corrente.

    address public highestBidder;
    uint public highestBid;
```

```
    // Permitidas retiradas de propostas prévias
    mapping(address => uint) pendingReturns;

    // Colocar em "true" no final, desabilitando qualquer mudança
    bool ended;

    // Eventos que serão ativados com as mudanças.

    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // A seguir vem o assim chamado "natspec comment"
    // reconhecível por três barras.
    // Será mostrado quando o usuário é requisitado para
    // confirmar a transação.

    /// Criar um leilão simples com "_biddingTime" segundos,
    /// período de proposta em nome do endereço de beneficiário
    /// "_beneficiary".
    function SimpleAuction(
        uint _biddingTime,
        address _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingTime = _biddingTime;
    }

    /// Proposta no leilão com o valor enviado
    /// junto com esta transação.
    /// O valor somente será devolvido se a pro-
    /// posta não for vencedora.

    function bid() payable {
        // Não necessita de argumentos, toda
        // informação faz já parte da transação.
        // A "keyword payable" é requerida para
        // a função estar habilitada a receber Ethers.

        // Reverte a chamada se o período de proposta
        // for encerrado.
        require(now <= (auctionStart + biddingTime));

        // Se a proposta não for a mais alta, enviar o
        // dinheiro de volta

        require(msg.value > highestBid);

        if (highestBidder != 0) {
            // Restituir o dinheiro simplesmente udando
            // " highestBidder.send(highestBid)" é um risco de
            // segurança porque poderia ter executado um contrato
            // não confiável.
            // É sempre mais seguro deixar os destinatários das restiuições
            // resgatar seus valores por eles mesmos.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
```

```
        highestBid = msg.value;
        HighestBidIncreased(msg.sender, msg.value);
    }

    /// Restituir uma proposta que foi superada.
    function withdraw() returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // É importante colocar esta variável em zero para que o destinatário
            // possa chamar esta função novamente como parte da chamada recebida
            // antes de "send" retornar.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // Não necessário chamar aqui, somente dar um reset no valor devido.
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// Fim do leilão e envio da proposta mais alta
    /// para o beneficiário.
    function auctionEnd() {
        // É uma boa diretriz estruturar as função que interagem
        // com outros contratos (isto é, elas chamam funções para enviar Ethers)
        // em três fases:
        // 1. verificar condições;
        // 2. realizar ações (condições potenciais de mudança);
        // 3. interagir com outros contratos.
        // Se essas fases forem misturadas, o outro contrato pode
        // chamar de volta dentro do corrente contrato e modificar o estado ou
        // efeito de causa (pagamento de ethers) a ser realizado multiplas vezes.
        // Se as funções chamadas internamente incluiem interação com contratos
        // externos, eles também tem que considerar interações com estes.
        // 1. Condições

        require(now >= (auctionStart + biddingTime)); // leilão não encerrado ainda
        require(!ended); // função já foi chamada

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

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

Leilão Cego

O leilão aberto anterior é estendido para um leilão cego na sequência. A vantagem de um leilão cego é que não há pressão no tempo para o final do período de licitação. Criando um leilão cego em uma plataforma de computação transparente pode soar como uma contradição, mas a criptografia vem ao resgate.

Durante o ** período de licitação**, um licitante não envia seu lance de verdade, mas apenas uma versão hash

do mesmo. Como atualmente é considerado praticamente impossível para encontrar dois valores (suficientemente longos) cujos hash sejão iguais, o licitante é associado ao lance por esse meio. Após o final do período de licitação, os concorrentes têm que revelar suas propostas: eles enviam seus valores não criptografados e o contrato verifica se o valor de hash é o mesmo que o fornecido durante o período de licitação.

Outro desafio é como fazer o leilão ** vinculativo e cego ** ao mesmo tempo: a única maneira de impedir que o licitante apenas não envie o dinheiro depois que ele ganhou o leilão é fazer com que ele o envie juntamente com a oferta. Como as transferências de valor não podem ser cegas na rede Ethereum, qualquer um pode ver o valor.

O contrato a seguir resolve esse problema aceitando qualquer valor maior do que o mais alto lance. Uma vez que, claro, isso só pode ser verificado durante a fase de revelação, algumas propostas podem ser ** inválidas ** e isso pode ser proposital (até mesmo fornece um flag explícito para colocar lances inválidos com transferências de alto valor): Participantes podem confundir a concorrência colocando vários lances altos ou baixos inválidos.

```
pragma solidity ^0.4.11;

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

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

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

    address public highestBidder;
    uint public highestBid;

    // Retiradas permitidas por negociações prévias

    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Modificadores são um meio conveniente para validas
            /// entradas nas funções. "onlyBefore" é aplicado ao negócio
            /// abaixo:
            /// O novo corpo da função é o modificador do corpo onde "_"
            /// é substituido pelo corpo antigo da função.

    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }

    function BlindAuction(
        uint _biddingTime,
        uint _revealTime,
        address _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }
```

```
            /// Colocar uma negociação "cega" com `_blindedBid` = keccak256(value,
    /// fake, secret).
            /// Os ethers remetidos somente são devolvido se a negociação
↪/// for corretamente revelada na fase de revelação de propostas. A negociação
↪/// é valida se o valor enviado junto com a negociação é ao menos "value" e
            /// fake não é verdadeiro.
↪/// Configurar fake para "true" (verdadeiro) e enviar nao exatamente o valor são
↪/// maneiras de esconder a oferta real mas ainda fazer o depósito requerido. O mesmo
            /// endereço pode colocar multiplas ofertas.

    function bid(bytes32 _blindedBid)
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

            /// Revelar as ofertas "cegas". Você ira restituir para todos
            /// ofertas inválidas corretamente cegas e para todas as ofertas
            /// exceto para a mais alta de todas.

    function reveal(
        uint[] _values,
        bool[] _fake,
        bytes32[] _secret
    )
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            var bid = bids[msg.sender][i];
            var (value, fake, secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bid.blindedBid != keccak256(value, fake, secret)) {
                // Oferta não foi realmente revelada.
                                    // Não faça o depósito de restituição.

                continue;
            }
            refund += bid.deposit;
            if (!fake && bid.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Torne impossível para o remetente reinvindicar
```

```
                            // o mesmo depósito.

            bid.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }
            // Esta é uma função "interna" que significa que só
            // pode ser chamada pelo próprio contrato (ou por contra-
            // tos derivados)

    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != 0) {
            // Resituir o maior ofertante.

                            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

            /// Retirar uma oferta que foi superada.

    function withdraw() {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
                            // É importante colocar em zero para que o receptor
                            // possa chamar essa função novamente como parte do␣
↪recebimento
                            // da chamada antes de "send" retornar (veja o comentário␣
↪acima sobre condições -->
                            // efeitos --> interações).


            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    /// Fim do leilão e envio da maior proposta
            /// para o beneficiário.

    function auctionEnd()
        onlyAfter(revealEnd)
    {
        require(!ended);
        AuctionEnded(highestBidder, highestBid);
        ended = true;

                    // Enviaremos todo o dinheiro existente, porque
                    // algumas das restiuições podem ter falhado.
```

```
        beneficiary.transfer(this.balance);
    }
}
```

6.3.3 Compra Remota Segura

```
pragma solidity ^0.4.11;

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

    // Garantir que 'msg.value' é um número par.
    // Divisão será truncada se for um número impar.
    // Verificar via multiplicação que não é um número impar.

    function Purchase() payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value);
    }

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

    modifier onlyBuyer() {
        require(msg.sender == buyer);
        _;
    }

    modifier onlySeller() {
        require(msg.sender == seller);
        _;
    }

    modifier inState(State _state) {
        require(state == _state);
        _;
    }

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

    /// Abortar a compra e reinvindicar os ether.
    /// Pode somente ser chamado pelo vendedor antes
    /// do contrato ser travado.

    function abort()
        onlySeller
        inState(State.Created)
```

```
    {
        Aborted();
        state = State.Inactive;
        seller.transfer(this.balance);
    }

    /// Confirme a compra como comprador.
    /// Transação tem que incluir '2 * valor' ether.
    /// Os ether ficarão presos até a função confirmReceived
    /// for chamada.

    function confirmPurchase()
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// Confirmar que você (o comprador) recebeu o item.
    /// Isto irá liberar os ether presos.

    function confirmReceived()
        onlyBuyer
        inState(State.Locked)
    {
        ItemReceived();
        // É importante mudar o estado primeiro porque
        // de outra forma, o contrato chamado usando 'send'
        // abaixo pode chamar novamente aqui.

        state = State.Inactive;

        // NOTA: Isto efetivamente permite o comprador e o vendedor
        // bloquear a restituição - a retirada padrão deve ser usada.

        buyer.transfer(value);
        seller.transfer(this.balance);
    }çp~´
}
```

6.3.4 Micropayment Channel

To be written.

6.4 Solidity em Profundidade

Esta seção irá lhe providenciar tudo o que você precisa saber sobre o Solidity. Se algo estiver faltando aqui, por favor nos contate em Gitter ou faça um pedido em Github.

6.4.1 Layout of a Solidity Source File

Source files can contain an arbitrary number of contract definitions, include directives and pragma directives.

Version Pragma

Source files can (and should) be annotated with a so-called version pragma to reject being compiled with future compiler versions that might introduce incompatible changes. We try to keep such changes to an absolute minimum and especially introduce changes in a way that changes in semantics will also require changes in the syntax, but this is of course not always possible. Because of that, it is always a good idea to read through the changelog at least for releases that contain breaking changes, those releases will always have versions of the form `0.x.0` or `x.0.0`.

The version pragma is used as follows:

```
pragma solidity ^0.4.0;
```

Such a source file will not compile with a compiler earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0 (this second condition is added by using `^`). The idea behind this is that there will be no breaking changes until version `0.5.0`, so we can always be sure that our code will compile the way we intended it to. We do not fix the exact version of the compiler, so that bugfix releases are still possible.

It is possible to specify much more complex rules for the compiler version, the expression follows those used by npm.

Importing other Source Files

Syntax and Semantics

Solidity supports import statements that are very similar to those available in JavaScript (from ES6 on), although Solidity does not know the concept of a «default export».

At a global level, you can use import statements of the following form:

```
import "filename";
```

This statement imports all global symbols from «filename» (and symbols imported there) into the current global scope (different than in ES6 but backwards-compatible for Solidity).

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

...creates a new global symbol `symbolName` whose members are all the global symbols from `"filename"`.

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

...creates new global symbols `alias` and `symbol2` which reference `symbol1` and `symbol2` from `"filename"`, respectively.

Another syntax is not part of ES6, but probably convenient:

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

which is equivalent to `import * as symbolName from "filename";`.

Paths

In the above, `filename` is always treated as a path with `/` as directory separator, `.` as the current and `..` as the parent directory. When `.` or `..` is followed by a character except `/`, it is not considered as the current or the parent directory. All path names are treated as absolute paths unless they start with the current `.` or the parent directory `..`.

To import a file `x` from the same directory as the current file, use `import "./x" as x;`. If you use `import "x" as x;` instead, a different file could be referenced (in a global «include directory»).

It depends on the compiler (see below) how to actually resolve the paths. In general, the directory hierarchy does not need to strictly map onto your local filesystem, it can also map to resources discovered via e.g. ipfs, http or git.

Use in Actual Compilers

When the compiler is invoked, it is not only possible to specify how to discover the first element of a path, but it is possible to specify path prefix remappings so that e.g. `github.com/ethereum/dapp-bin/library` is remapped to `/usr/local/dapp-bin/library` and the compiler will read the files from there. If multiple remappings can be applied, the one with the longest key is tried first. This allows for a «fallback-remapping» with e.g. `""` maps to `"/usr/local/include/solidity"`. Furthermore, these remappings can depend on the context, which allows you to configure packages to import e.g. different versions of a library of the same name.

solc:

For solc (the commandline compiler), these remappings are provided as `context:prefix=target` arguments, where both the `context:` and the `=target` parts are optional (where target defaults to prefix in that case). All remapping values that are regular files are compiled (including their dependencies). This mechanism is completely backwards-compatible (as long as no filename contains = or :) and thus not a breaking change. All imports in files in or below the directory `context` that import a file that starts with `prefix` are redirected by replacing `prefix` by `target`.

So as an example, if you clone `github.com/ethereum/dapp-bin/` locally to `/usr/local/dapp-bin`, you can use the following in your source file:

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

and then run the compiler as

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

As a more complex example, suppose you rely on some module that uses a very old version of dapp-bin. That old version of dapp-bin is checked out at `/usr/local/dapp-bin_old`, then you can use

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

so that all imports in `module2` point to the old version but imports in `module1` get the new version.

Note that solc only allows you to include files from certain directories: They have to be in the directory (or subdirectory) of one of the explicitly specified source files or in the directory (or subdirectory) of a remapping target. If you want to allow direct absolute includes, just add the remapping `=/`.

If there are multiple remappings that lead to a valid file, the remapping with the longest common prefix is chosen.

Remix:

Remix provides an automatic remapping for github and will also automatically retrieve the file over the network: You can import the iterable mapping by e.g. `import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;`.

Other source code providers may be added in the future.

Comments

Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Additionally, there is another type of comment called a natspec comment, for which the documentation is not yet written. They are written with a triple slash (`///`) or a double asterisk block(`/** ... */`) and they should be used directly above function declarations or statements. You can use Doxygen-style tags inside these comments to document functions, annotate conditions for formal verification, and provide a **confirmation text** which is shown to users when they attempt to invoke a function.

In the following example we document the title of the contract, the explanation for the two input parameters and two returned values.

```
pragma solidity ^0.4.0;

/** @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) returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

6.4.2 Estrutura de um contrato

Contratos em Solidity são similares a classes em linguagens orientadas a objetos. Cada contrato pode conter declarações de *Variáveis de Estado*, *Funções*, *Modificadores de Função*, *Eventos*, *Tipos de Estrutura* and *Tipos de Enum*. Além disso, contratos podem herdar de outros contratos.

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of *Variáveis de Estado*, *Funções*, *Modificadores de Função*, *Eventos*, *Tipos de Estrutura* and *Tipos de Enum*. Furthermore, contracts can inherit from other contracts.

Variáveis de Estado

Variáveis de estado são valores armazenados permanentemente na memória de contrato.

```
pragma solidity ^0.4.0;

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

Veja a seção *Types* para tipos válidos de variáveis de estado e *Visibility and Getters* para possíveis escolhad de visibilidade.

Funções

Funções são unidades executáveis de código dentro de um contrato.

```
pragma solidity ^0.4.0;

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

Function Calls podem acontecer internamente ou externamente e tem diferentes níveis de visibilidade (*Visibility and Getters*) para outros contratos.

Modificadores de Função

Modificadores de função podem ser usados para alterar a semântica das funções de forma declarativa (veja *Function Modifiers* na seção contratos).

```
pragma solidity ^0.4.11;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modificador
        require(msg.sender == seller);
        _;
    }

    function abort() onlySeller { // Uso do Modificador
        // ...
    }
}
```

Eventos

Eventos são interfaces de conveniência com facilidades de log na EVM.

```
pragma solidity ^0.4.0;

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

```
    function bid() payable {
        // ...
        HighestBidIncreased(msg.sender, msg.value); // Evento Trigger
    }
}
```

Consulte: ref: *eventos* na seção de contratos para obter informações sobre como eventos são declarados e podem ser usados dentro de uma dapp.

Tipos de Estrutura

Estruturas são tipos personalizados que podem agrupar diversas variáveis (ver : ref: *structs* na seção de tipos).

```
pragma solidity ^0.4.0;

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

Tipos de Enum

Enums podem ser usados para criar tipos personalizados com um conjunto finito de valores (veja : ref: *enums* na seção de tipos).

```
pragma solidity ^0.4.0;

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

6.4.3 Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified (or at least known - see *Type Deduction* below) at compile-time. Solidity provides several elementary types which can be combined to form complex types.

In addition, types can interact with each other in expressions containing operators. For a quick reference of the various operators, see *Order of Precedence of Operators*.

Value Types

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

Booleans

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

Operators:

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

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

Integers

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

Operators:

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

Division always truncates (it is just compiled to the `DIV` opcode of the EVM), but it does not truncate if both operators are *literals* (or literal expressions).

Division by zero and modulus with zero throws a runtime exception.

The result of a shift operation is the type of the left operand. The expression `x << y` is equivalent to `x * 2**y`, and `x >> y` is equivalent to `x / 2**y`. This means that shifting negative numbers sign extends. Shifting by a negative amount throws a runtime exception.

Aviso: The results produced by shift right of negative values of signed integer types is different from those produced by other programming languages. In Solidity, shift right maps to division so the shifted negative values are going to be rounded towards zero (truncated). In other programming languages the shift right of negative values works like division with rounding down (towards negative infinity).

Fixed Point Numbers

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

`fixed` / `ufixed`: Signed and unsigned fixed point number of various sizes. Keywords `ufixedMxN` and `fixedMxN`, where `M` represent the number of bits taken by the type and `N` represent how many decimal points are

available. `M` must be divisible by 8 and goes from 8 to 256 bits. `N` must be between 0 and 80, inclusive. `ufixed` and `fixed` are aliases for `ufixed128x19` and `fixed128x19`, respectively.

Operators:

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

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

Address

`address`: Holds a 20 byte value (size of an Ethereum address). Address types also have members and serve as a base for all contracts.

Operators:

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

Members of Addresses

- `balance` and `transfer`

For a quick reference, see *Address Related*.

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

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

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

- `send`

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

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

- `call`, `callcode` and `delegatecall`

Furthermore, to interface with contracts that do not adhere to the ABI, the function `call` is provided which takes an arbitrary number of arguments of any type. These arguments are padded to 32 bytes and concatenated. One exception is the case where the first argument is encoded to exactly four bytes. In this case, it is not padded to allow the use of function signatures here.

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
nameReg.call("register", "MyName");
nameReg.call(bytes4(keccak256("fun(uint256)")), a);
```

`call` returns a boolean indicating whether the invoked function terminated (`true`) or caused an EVM exception (`false`). It is not possible to access the actual data returned (for this we would need to know the encoding and size in advance).

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

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

Similarly, the supplied Ether value can be controlled too:

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

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

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

Nota: It is not yet possible to use the gas or value modifiers on overloaded functions.

A workaround is to introduce a special case for gas and value and just re-check whether they are present at the point of overload resolution.

In a similar way, the function `delegatecall` can be used: the difference is that only the code of the given address is used, all other aspects (storage, balance, ...) are taken from the current contract. The purpose of `delegatecall` is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for delegatecall to be used. Prior to homestead, only a limited variant called `callcode` was available that did not provide access to the original `msg.sender` and `msg.value` values.

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

The `.gas()` option is available on all three methods, while the `.value()` option is not supported for `delegatecall`.

Nota: All contracts inherit the members of address, so it is possible to query the balance of the current contract using `this.balance`.

Nota: The use of `callcode` is discouraged and will be removed in the future.

Aviso: All these functions are low-level functions and should be used with care. Specifically, any unknown contract might be malicious and if you call it, you hand over control to that contract which could in turn call back into your contract, so be prepared for changes to your state variables when the call returns.

Fixed-size byte arrays

`bytes1`, `bytes2`, `bytes3`, ..., `bytes32`. `byte` is an alias for `bytes1`.

Operators:

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

The shifting operator works with any integer type as right operand (but will return the type of the left operand), which denotes the number of bits to shift by. Shifting by a negative amount will cause a runtime exception.

Members:

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

Nota: It is possible to use an array of bytes as `byte[]`, but it is wasting a lot of space, 31 bytes every element, to be exact, when passing in calls. It is better to use `bytes`.

Dynamically-sized byte array

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

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

As a rule of thumb, use `bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of `bytes1` to `bytes32` because they are much cheaper.

Address Literals

Hexadecimal literals that pass the address checksum test, for example `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` are of `address` type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce a warning and are treated as regular rational number literals.

Rational and Integer Literals

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

Decimal fraction literals are formed by a `.` with at least one number on one side. Examples include `1.`, `.1` and `1.3`.

Scientific notation is also supported, where the base can have fractions, while the exponent cannot. Examples include `2e10`, `-2e10`, `2e-10`, `2.5e1`.

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

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

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

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

Aviso: Division on integer literals used to truncate in earlier versions, but it will now convert into a rational number, i.e. `5 / 2` is not equal to `2`, but to `2.5`.

Nota: Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions. Even though we know that the value of the expression assigned to `b` in the following example evaluates to an integer, the partial expression `2.5 + a` does not type check so the code does not compile

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

String Literals

String literals are written with either double or single-quotes (`"foo"` or `'bar'`). They do not imply trailing zeroes as in C; `"foo"` represents three bytes not four. As with integer literals, their type can vary, but they are implicitly convertible to `bytes1`, ..., `bytes32`, if they fit, to `bytes` and to `string`.

String literals support escape characters, such as `\n`, `\xNN` and `\uNNNN`. `\xNN` takes a hex value and inserts the appropriate byte, while `\uNNNN` takes a Unicode codepoint and inserts an UTF-8 sequence.

Hexadecimal Literals

Hexademical Literals are prefixed with the keyword `hex` and are enclosed in double or single-quotes (`hex"001122FF"`). Their content must be a hexadecimal string and their value will be the binary representation of those values.

Hexademical Literals behave like String Literals and have the same convertibility restrictions.

Enums

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversions check the value ranges at runtime and a failure causes an exception. Enums needs at least one member.

```
pragma solidity ^0.4.0;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
```

```
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

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

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity. The integer type used is just
    // large enough to hold all enum values, i.e. if you have more values,
    // `uint16` will be used and so on.
    function getChoice() returns (ActionChoices) {
        return choice;
    }

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

Function Types

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

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

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

Function types are notated as follows:

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

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

By default, function types are internal, so the `internal` keyword can be omitted. In contrast, contract functions themselves are public by default, only when used as the name of a type, the default is internal.

There are two ways to access a function in the current contract: Either directly by its name, `f`, or using `this.f`. The former will result in an internal function, the latter in an external function.

If a function type variable is not initialized, calling it will result in an exception. The same happens if you call a function after using `delete` on it.

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

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

Example that shows how to use internal function types:

```
pragma solidity ^0.4.5;

library ArrayUtils {
  // internal functions can be used in internal library functions because
  // they will be part of the same code context
  function map(uint[] memory self, function (uint) returns (uint) f)
    internal
    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) returns (uint) f
  )
    internal
    returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
      r = f(r, self[i]);
    }
  }
  function range(uint length) internal 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) returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
  }
  function square(uint x) internal returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal returns (uint) {
    return x + y;
  }
}
```

Another example that uses external function types:

```
pragma solidity ^0.4.11;

contract Oracle {
  struct Request {
    bytes data;
    function(bytes memory) external callback;
  }
  Request[] requests;
  event NewRequest(uint);
```

```
  function query(bytes data, function(bytes memory) external callback) {
    requests.push(Request(data, callback));
    NewRequest(requests.length - 1);
  }
  function reply(uint requestID, bytes response) {
    // Here goes the check that the reply comes from a trusted source
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant oracle = Oracle(0x1234567); // known contract
  function buySomething() {
    oracle.query("USD", this.oracleResponse);
  }
  function oracleResponse(bytes response) {
    require(msg.sender == address(oracle));
    // Use the data
  }
}
```

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

Reference Types

Complex types, i.e. types which do not always fit into 256 bits have to be handled more carefully than the value-types we have already seen. Since copying them can be quite expensive, we have to think about whether we want them to be stored in **memory** (which is not persisting) or **storage** (where the state variables are held).

Data location

Every complex type, i.e. *arrays* and *structs*, has an additional annotation, the «data location», about whether it is stored in memory or in storage. Depending on the context, there is always a default, but it can be overridden by appending either `storage` or `memory` to the type. The default for function parameters (including return parameters) is `memory`, the default for local variables is `storage` and the location is forced to `storage` for state variables (obviously).

There is also a third data location, `calldata`, which is a non-modifiable, non-persistent area where function arguments are stored. Function parameters (not return parameters) of external functions are forced to `calldata` and behave mostly like `memory`.

Data locations are important because they change how assignments behave: assignments between storage and memory and also to a state variable (even from other state variables) always create an independent copy. Assignments to local storage variables only assign a reference though, and this reference always points to the state variable even if the latter is changed in the meantime. On the other hand, assignments from a memory stored reference type to another memory-stored reference type do not create a copy.

```
pragma solidity ^0.4.0;

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

    // the data location of memoryArray is memory
```

```
    function f(uint[] memoryArray) {
        x = memoryArray; // works, copies the whole array to storage
        var y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) {}
}
```

Summary

Forced data location:

- parameters (not return) of external functions: calldata
- state variables: storage

Default data location:

- parameters (also return) of functions: memory
- all other local variables: storage

Arrays

Arrays can have a compile-time fixed size or they can be dynamic. For storage arrays, the element type can be arbitrary (i.e. also other arrays, mappings or structs). For memory arrays, it cannot be a mapping and has to be an ABI type if it is an argument of a publicly-visible function.

An array of fixed size `k` and element type `T` is written as `T[k]`, an array of dynamic size as `T[]`. As an example, an array of 5 dynamic arrays of `uint` is `uint[][5]` (note that the notation is reversed when compared to some other languages). To access the second uint in the third dynamic array, you use `x[2][1]` (indices are zero-based and access works in the opposite way of the declaration, i.e. `x[2]` shaves off one level in the type from the right).

Variables of type `bytes` and `string` are special arrays. A `bytes` is similar to `byte[]`, but it is packed tightly in calldata. `string` is equal to `bytes` but does not allow length or index access (for now).

So `bytes` should always be preferred over `byte[]` because it is cheaper.

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

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

Allocating Memory Arrays

Creating arrays with variable length in memory can be done using the `new` keyword. As opposed to storage arrays, it is **not** possible to resize memory arrays by assigning to the `.length` member.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint len) {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        // Here we have a.length == 7 and b.length == len
        a[6] = 8;
    }
}
```

Array Literals / Inline Arrays

Array literals are arrays that are written as an expression and are not assigned to a variable right away.

```
pragma solidity ^0.4.0;

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

The type of an array literal is a memory array of fixed size whose base type is the common type of the given elements. The type of `[1, 2, 3]` is `uint8[3] memory`, because the type of each of these constants is `uint8`. Because of that, it was necessary to convert the first element in the example above to `uint`. Note that currently, fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible:

```
// This will not compile.

pragma solidity ^0.4.0;

contract C {
    function f() {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] x = [uint(1), 3, 4];
    }
}
```

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

Members

length: Arrays have a `length` member to hold their number of elements. Dynamic arrays can be resized in storage (not in memory) by changing the `.length` member. This does not happen automatically when attempting to

access elements outside the current length. The size of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

push: Dynamic storage arrays and `bytes` (not `string`) have a member function called `push` that can be used to append an element at the end of the array. The function returns the new length.

Aviso: It is not yet possible to use arrays of arrays in external functions.

Aviso: Due to limitations of the EVM, it is not possible to return dynamic content from external function calls. The function `f` in `contract C { function f() returns (uint[]) { ... } }` will return something if called from web3.js, but not if called from Solidity.

The only workaround for now is to use large statically-sized arrays.

```
pragma solidity ^0.4.0;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of length two).
    bool[2][] m_pairsOfFlags;
    // newPairs is stored in memory - the default for function arguments

    function setAllFlagPairs(bool[2][] newPairs) {
        // assignment to a storage array replaces the complete array
        m_pairsOfFlags = newPairs;
    }

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

    function changeFlagArraySize(uint newSize) {
        // if the new size is smaller, removed array elements will be cleared
        m_pairsOfFlags.length = newSize;
    }

    function clear() {
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags.length = 0;
    }

    bytes m_byteData;

    function byteArrays(bytes data) {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = 8;
```

```
        delete m_byteData[2];
    }

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

    function createMemoryArray(uint size) returns (bytes) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(i);
        return b;
    }
}
```

Structs

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

```
pragma solidity ^0.4.11;

contract CrowdFunding {
    // Defines a new type with two fields.
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

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

    function newCampaign(address beneficiary, uint goal) returns (uint campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // Creates new struct and saves in storage. We leave out the mapping type.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

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

```
    function checkGoalReached(uint campaignID) 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;
    }
}
```

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

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

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

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

Mappings

Mapping types are declared as `mapping(_KeyType => _ValueType)`. Here `_KeyType` can be almost any type except for a mapping, a dynamically sized array, a contract, an enum and a struct. `_ValueType` can actually be any type, including mappings.

Mappings can be seen as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type's *default value*. The similarity ends here, though: The key data is not actually stored in a mapping, only its `keccak256` hash used to look up the value.

Because of this, mappings do not have a length or a concept of a key or value being «set».

Mappings are only allowed for state variables (or as storage reference types in internal functions).

It is possible to mark mappings `public` and have Solidity create a *getter*. The `_KeyType` will become a required parameter for the getter and it will return `_ValueType`.

The `_ValueType` can be a mapping too. The getter will have one parameter for each `_KeyType`, recursively.

```
pragma solidity ^0.4.0;

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

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

contract MappingUser {
    function f() returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
```

```
        return m.balances(this);
    }
}
```

Nota: Mappings are not iterable, but it is possible to implement a data structure on top of them. For an example, see iterable mapping.

Operators Involving LValues

If `a` is an LValue (i.e. a variable or something that can be assigned to), the following operators are available as shorthands:

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `a |=`, `&=` and `^=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1` / `a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change.

delete

`delete a` assigns the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`, but it can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements reset. For structs, it assigns a struct with all members reset.

`delete` has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings. However, individual keys and what they map to can be deleted.

It is important to note that `delete a` really behaves like an assignment to `a`, i.e. it stores a new object in `a`.

```
pragma solidity ^0.4.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x which still holds a copy
        uint[] y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[] is a␣
↪complex object, also
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as assignments to local␣
↪variables
        // referencing storage objects can only be made from existing storage objects.
    }
}
```

Conversions between Elementary Types

Implicit Conversions

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands to the type of the other (the same is true for assignments). In general, an implicit conversion between value-types is possible if it makes sense semantically and no information is lost: `uint8` is convertible to `uint16` and `int128` to `int256`, but `int8` is not convertible to `uint256` (because `uint256` cannot hold e.g. `-1`). Furthermore, unsigned integers can be converted to bytes of the same or larger size, but not vice-versa. Any type that can be converted to `uint160` can also be converted to `address`.

Explicit Conversions

If the compiler does not allow implicit conversion but you know what you are doing, an explicit type conversion is sometimes possible. Note that this may give you some unexpected behaviour so be sure to test to ensure that the result is what you want! Take the following example where you are converting a negative `int8` to a `uint`:

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

At the end of this code snippet, `x` will have the value `0xfffff..fd` (64 hex characters), which is -3 in the two's complement representation of 256 bits.

If a type is explicitly converted to a smaller type, higher-order bits are cut off:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

Type Deduction

For convenience, it is not always necessary to explicitly specify the type of a variable, the compiler automatically infers it from the type of the first expression that is assigned to the variable:

```
uint24 x = 0x123;
var y = x;
```

Here, the type of `y` will be `uint24`. Using `var` is not possible for function parameters or return parameters.

Aviso: The type is only deduced from the first assignment, so the loop in the following snippet is infinite, as `i` will have the type `uint8` and any value of this type is smaller than `2000`. `for (var i = 0; i < 2000; i++) { ... }`

6.4.4 Units and Globally Available Variables

Ether Units

A literal number can take a suffix of `wei`, `finney`, `szabo` or `ether` to convert between the subdenominations of Ether, where Ether currency numbers without a postfix are assumed to be Wei, e.g. `2 ether == 2000 finney` evaluates to `true`.

Time Units

Suffixes like `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` after literal numbers can be used to convert between units of time where seconds are the base unit and units are considered naively in the following way:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`
- `1 years == 365 days`

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds. Due to the fact that leap seconds cannot be predicted, an exact calendar library has to be updated by an external oracle.

These suffixes cannot be applied to variables. If you want to interpret some input variable in e.g. days, you can do it in the following way:

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

Special Variables and Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain.

Block and Transaction Properties

- `block.blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks excluding current
- `block.coinbase` (`address`): 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 as seconds since unix epoch
- `msg.data` (`bytes`): complete calldata
- `msg.gas` (`uint`): remaining gas
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `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`): sender of the transaction (full call chain)

Nota: The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every **external** function call. This includes calls to library functions.

Nota: Do not rely on `block.timestamp`, `now` and `block.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.

Nota: If you want to implement access restrictions in library functions using `msg.sender`, you have to manually supply the value of `msg.sender` as an argument.

Nota: 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.

Error Handling

`assert(bool condition)`: throws if the condition is not met - to be used for internal errors.

`require(bool condition)`: throws if the condition is not met - to be used for errors in inputs or external components.

`revert()`: abort execution and revert state changes

Mathematical and Cryptographic Functions

`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`.

`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`.

`keccak256(...) returns (bytes32)`: compute the Ethereum-SHA-3 (Keccak-256) hash of the (tightly packed) arguments

`sha256(...) returns (bytes32)`: compute the SHA-256 hash of the (tightly packed) arguments

`sha3(...) returns (bytes32)`: alias to `keccak256`

`ripemd160(...) returns (bytes20)`: compute RIPEMD-160 hash of the (tightly packed) arguments

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recover the address associated with the public key from elliptic curve signature or return zero on error (example usage)

In the above, «tightly packed» means that the arguments are concatenated without padding. This means that the following are all identical:

```
keccak256("ab", "c")
keccak256("abc")
keccak256(0x616263)
keccak256(6382179)
keccak256(97, 98, 99)
```

If padding is needed, explicit type conversions can be used: `keccak256("\x00\x12")` is the same as `keccak256(uint16(0x12))`.

Note that constants will be packed using the minimum number of bytes required to store them. This means that, for example, `keccak256(0) == keccak256(uint8(0))` and `keccak256(0x12345678) == keccak256(uint32(0x12345678))`.

It might be that you run into Out-of-Gas for `sha256`, `ripemd160` or `ecrecover` on a *private blockchain*. The reason for this is that those are implemented as so-called precompiled contracts and these contracts only really exist after they received the first message (although their contract code is hardcoded). Messages to non-existing contracts are more expensive and thus the execution runs into an Out-of-Gas error. A workaround for this problem is to first send e.g. 1 Wei to each of the contracts before you use them in your actual contracts. This is not an issue on the official or test net.

Address Related

`<address>.balance(uint256)`: balance of the *Address* in Wei

`<address>.transfer(uint256 amount)`: send given amount of Wei to *Address*, throws on failure

`<address>.send(uint256 amount) returns (bool)`: send given amount of Wei to *Address*, returns `false` on failure

`<address>.call(...) returns (bool)`: issue low-level `CALL`, returns `false` on failure

`<address>.callcode(...) returns (bool)`: issue low-level `CALLCODE`, returns `false` on failure

`<address>.delegatecall(...) returns (bool)`: issue low-level `DELEGATECALL`, returns `false` on failure

For more information, see the section on *Address*.

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

Nota: The use of `callcode` is discouraged and will be removed in the future.

Contract Related

`this` (current contract's type): the current contract, explicitly convertible to *Address*

`selfdestruct(address recipient)`: destroy the current contract, sending its funds to the given *Address*

suicide(address recipient): alias to `selfdestruct`

Furthermore, all functions of the current contract are callable directly including the current function.

6.4.5 Expressions and Control Structures

Input Parameters and Output Parameters

As in Javascript, functions may take parameters as input; unlike in Javascript and C, they may also return arbitrary number of parameters as output.

Input Parameters

The input parameters are declared the same way as variables are. As an exception, unused parameters can omit the variable name. For example, suppose we want our contract to accept one kind of external calls with two integers, we would write something like:

```
pragma solidity ^0.4.0;

contract Simple {
    function taker(uint _a, uint _b) {
        // do something with _a and _b.
    }
}
```

Output Parameters

The output parameters can be declared with the same syntax after the `returns` keyword. For example, suppose we wished to return two results: the sum and the product of the two given integers, then we would write:

```
pragma solidity ^0.4.0;

contract Simple {
    function arithmetics(uint _a, uint _b) returns (uint o_sum, uint o_product) {
        o_sum = _a + _b;
        o_product = _a * _b;
    }
}
```

The names of output parameters can be omitted. The output values can also be specified using `return` statements. The `return` statements are also capable of returning multiple values, see *Returning Multiple Values*. Return parameters are initialized to zero; if they are not explicitly set, they stay to be zero.

Input parameters and output parameters can be used as expressions in the function body. There, they are also usable in the left-hand side of assignment.

Control Structures

Most of the control structures from JavaScript are available in Solidity except for `switch` and `goto`. So there is: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, `? :`, with the usual semantics known from C or JavaScript.

Parentheses can *not* be omitted for conditionals, but curly brances can be omitted around single-statement bodies.

Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so `if (1) { ... }` is *not* valid Solidity.

Returning Multiple Values

When a function has multiple output parameters, `return (v0, v1, ..., vn)` can return multiple values. The number of components must be the same as the number of output parameters.

Function Calls

Internal Function Calls

Functions of the current contract can be called directly («internally»), also recursively, as seen in this nonsensical example:

```
pragma solidity ^0.4.0;

contract C {
    function g(uint a) returns (uint ret) { return f(); }
    function f() returns (uint ret) { return g(7) + f(); }
}
```

These function calls are translated into simple jumps inside the EVM. This has the effect that the current memory is not cleared, i.e. passing memory references to internally-called functions is very efficient. Only functions of the same contract can be called internally.

External Function Calls

The expressions `this.g(8);` and `c.g(2);` (where `c` is a contract instance) are also valid function calls, but this time, the function will be called «externally», via a message call and not directly via jumps. Please note that function calls on `this` cannot be used in the constructor, as the actual contract has not been created yet.

Functions of other contracts have to be called externally. For an external call, all function arguments have to be copied to memory.

When calling functions of other contracts, the amount of Wei sent with the call and the gas can be specified with special options `.value()` and `.gas()`, respectively:

```
pragma solidity ^0.4.0;

contract InfoFeed {
    function info() payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(address addr) { feed = InfoFeed(addr); }
    function callFeed() { feed.info.value(10).gas(800)(); }
}
```

The modifier `payable` has to be used for `info`, because otherwise, the *.value()* option would not be available.

Note that the expression `InfoFeed(addr)` performs an explicit type conversion stating that «we know that the type of the contract at the given address is `InfoFeed`» and this does not execute a constructor. Explicit type conversions have to be handled with extreme caution. Never call a function on a contract where you are not sure about its type.

We could also have used `function setFeed(InfoFeed _feed) { feed = _feed; }` directly. Be careful about the fact that `feed.info.value(10).gas(800)` only (locally) sets the value and amount of gas sent with the function call and only the parentheses at the end perform the actual call.

Function calls cause exceptions if the called contract does not exist (in the sense that the account does not contain code) or if the called contract itself throws an exception or goes out of gas.

Aviso: Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.

Named Calls and Anonymous Function Parameters

Function call arguments can also be given by name, in any order, if they are enclosed in `{ }` as can be seen in the following example. The argument list has to coincide by name with the list of parameters from the function declaration, but can be in arbitrary order.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint key, uint value) {
        // ...
    }

    function g() {
        // named arguments
        f({value: 2, key: 3});
    }
}
```

Omitted Function Parameter Names

The names of unused parameters (especially return parameters) can be omitted. Those names will still be present on the stack, but they are inaccessible.

```
pragma solidity ^0.4.0;

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

Creating Contracts via `new`

A contract can create a new contract using the `new` keyword. The full code of the contract being created has to be known in advance, so recursive creation-dependencies are not possible.

```
pragma solidity ^0.4.0;

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

contract C {
    D d = new D(4); // will be executed as part of C's constructor

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

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

As seen in the example, it is possible to forward Ether to the creation using the `.value()` option, but it is not possible to limit the amount of gas. If the creation fails (due to out-of-stack, not enough balance or other problems), an exception is thrown.

Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done. See *Order of Precedence of Operators* for more information.

Assignment

Destructuring Assignments and Returning Multiple Values

Solidity internally allows tuple types, i.e. a list of objects of potentially different types whose size is a constant at compile-time. Those tuples can be used to return multiple values at the same time and also assign them to multiple variables (or LValues in general) at the same time:

```
pragma solidity ^0.4.0;

contract C {
    uint[] data;

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

```
    function g() {
        // Declares and assigns the variables. Specifying the type explicitly is not␣
↪possible.
        var (x, b, y) = f();
        // Assigns to a pre-existing variable.
        (x, y) = (2, 7);
        // Common trick to swap values -- does not work for non-value storage types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
        // If the tuple ends in an empty component,
        // the rest of the values are discarded.
        (data.length,) = f(); // Sets the length to 7
        // The same can be done on the left side.
        (,data[3]) = f(); // Sets data[3] to 2
        // Components can only be left out at the left-hand-side of assignments, with
        // one exception:
        (x,) = (1,);
        // (1,) is the only way to specify a 1-component tuple, because (1) is
        // equivalent to 1.
    }
}
```

Complications for Arrays and Structs

The semantics of assignment are a bit more complicated for non-value types like arrays and structs. Assigning *to* a state variable always creates an independent copy. On the other hand, assigning to a local variable creates an independent copy only for elementary types, i.e. static types that fit into 32 bytes. If structs or arrays (including `bytes` and `string`) are assigned from a state variable to a local variable, the local variable holds a reference to the original state variable. A second assignment to the local variable does not modify the state but only changes the reference. Assignments to members (or elements) of the local variable *do* change the state.

Scoping and Declarations

A variable which is declared will have an initial default value whose byte-representation is all zeros. The «default values» of variables are the typical «zero-state» of whatever the type is. For example, the default value for a `bool` is `false`. The default value for the `uint` or `int` types is `0`. For statically-sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. Finally, for dynamically-sized arrays, `bytes` and `string`, the default value is an empty array or string.

A variable declared anywhere within a function will be in scope for the *entire function*, regardless of where it is declared. This happens because Solidity inherits its scoping rules from JavaScript. This is in contrast to many languages where variables are only scoped where they are declared until the end of the semantic block. As a result, the following code is illegal and cause the compiler to throw an error, `Identifier already declared`:

```
// This will not compile

pragma solidity ^0.4.0;

contract ScopingErrors {
    function scoping() {
        uint i = 0;

        while (i++ < 1) {
            uint same1 = 0;
```

```
        }

        while (i++ < 2) {
            uint same1 = 0;// Illegal, second declaration of same1
        }
    }

    function minimalScoping() {
        {
            uint same2 = 0;
        }

        {
            uint same2 = 0;// Illegal, second declaration of same2
        }
    }

    function forLoopScoping() {
        for (uint same3 = 0; same3 < 1; same3++) {
        }

        for (uint same3 = 0; same3 < 1; same3++) {// Illegal, second declaration of␣
↪same3
        }
    }
}
```

In addition to this, if a variable is declared, it will be initialized at the beginning of the function to its default value. As a result, the following code is legal, despite being poorly written:

```
function foo() returns (uint) {
    // baz is implicitly initialized as 0
    uint bar = 5;
    if (true) {
        bar += baz;
    } else {
        uint baz = 10;// never executes
    }
    return bar;// returns 5
}
```

Error handling: Assert, Require, Revert and Exceptions

Solidity uses state-reverting exceptions to handle errors. Such an exception will undo all changes made to the state in the current call (and all its sub-calls) and also flag an error to the caller. The convenience functions `assert` and `require` can be used to check for conditions and throw an exception if the condition is not met. The `assert` function should only be used to test for internal errors, and to check invariants. The `require` function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts. If used properly, analysis tools can evaluate your contract to identify the conditions and function calls which will reach a failing `assert`. Properly functioning code should never reach a failing assert statement; if this happens there is a bug in your contract which you should fix.

There are two other ways to trigger exceptions: The `revert` function can be used to flag an error and revert the current call. In the future it might be possible to also include details about the error in a call to `revert`. The `throw` keyword can also be used as an alternative to `revert()`.

Nota: From version 0.4.13 the `throw` keyword is deprecated and will be phased out in the future.

When exceptions happen in a sub-call, they «bubble up» (i.e. exceptions are rethrown) automatically. Exceptions to this rule are `send` and the low-level functions `call`, `delegatecall` and `callcode` – those return `false` in case of an exception instead of «bubbling up».

Aviso: The low-level `call`, `delegatecall` and `callcode` will return success if the calling account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

Catching exceptions is not yet possible.

In the following example, you can see how `require` can be used to easily check conditions on inputs and how `assert` can be used for internal error checking:

```
pragma solidity ^0.4.0;

contract Sharer {
    function sendHalf(address addr) payable returns (uint balance) {
        require(msg.value % 2 == 0); // Only allow even numbers
        uint balanceBeforeTransfer = this.balance;
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(this.balance == balanceBeforeTransfer - msg.value / 2);
        return this.balance;
    }
}
```

An `assert`-style exception is generated in the following situations:

1. If you access an array at a too large or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).
2. If you access a fixed-length `bytesN` at a too large or negative index.
3. If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).
4. If you shift by a negative amount.
5. If you convert a value too big or negative into an enum type.
6. If you call a zero-initialized variable of internal function type.
7. If you call `assert` with an argument that evaluates to false.

A `require`-style exception is generated in the following situations:

1. Calling `throw`.
2. Calling `require` with an argument that evaluates to `false`.
3. If you call a function via a message call but it does not finish properly (i.e. it runs out of gas, has no matching function, or throws an exception itself), except when a low level operation `call`, `send`, `delegatecall` or `callcode` is used. The low level operations never throw exceptions but indicate failures by returning `false`.
4. If you create a contract using the `new` keyword but the contract creation does not finish properly (see above for the definition of «not finish properly»).
5. If you perform an external function call targeting a contract that contains no code.

6. If your contract receives Ether via a public function without `payable` modifier (including the constructor and the fallback function).
7. If your contract receives Ether via a public getter function.
8. If a `.transfer()` fails.

Internally, Solidity performs a revert operation (instruction `0xfd`) for a `require`-style exception and executes an invalid operation (instruction `0xfe`) to throw an `assert`-style exception. In both cases, this causes the EVM to revert all changes made to the state. The reason for reverting is that there is no safe way to continue execution, because an expected effect did not occur. Because we want to retain the atomicity of transactions, the safest thing to do is to revert all changes and make the whole transaction (or at least call) without effect. Note that `assert`-style exceptions consume all gas available to the call, while `require`-style exceptions will not consume any gas starting from the Metropolis release.

6.4.6 Contracts

Contracts in Solidity are similar to classes in object-oriented languages. They contain persistent data in state variables and functions that can modify these variables. Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables are inaccessible.

Creating Contracts

Contracts can be created «from outside» or from Solidity contracts. When a contract is created, its constructor (a function with the same name as the contract) is executed once.

A constructor is optional. Only one constructor is allowed, and this means overloading is not supported.

From `web3.js`, i.e. the JavaScript API, this is done as follows:

```
// Need to specify some source including contract name for the data param below
var source = "contract CONTRACT_NAME { function CONTRACT_NAME(uint a, uint b) {} }";

// The json abi array generated by the compiler
var abiArray = [
    {
        "inputs":[
            {"name":"x","type":"uint256"},
            {"name":"y","type":"uint256"}
        ],
        "type":"constructor"
    },
    {
        "constant":true,
        "inputs":[],
        "name":"x",
        "outputs":[{"name":"","type":"bytes32"}],
        "type":"function"
    }
];

var MyContract_ = web3.eth.contract(source);
MyContract = web3.eth.contract(MyContract_.CONTRACT_NAME.info.abiDefinition);
// deploy new contract
var contractInstance = MyContract.new(
    10,
    11,
```

```
    {from: myAccount, gas: 1000000}
);
```

Internally, constructor arguments are passed 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 ^0.4.0;

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

    function transfer(address newOwner) {
        // 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 here stops
        // immediately.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

contract TokenCreator {
```

```
    function createToken(bytes32 name)
       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) {
        // Again, the external type of "tokenAddress" is
        // simply "address".
        tokenAddress.changeName(name);
    }

    function isTokenTransferOK(
        address currentOwner,
        address newOwner
    ) returns (bool ok) {
        // Check some arbitrary condition.
        address tokenAddress = msg.sender;
        return (keccak256(newOwner) & 0xff) == (bytes20(tokenAddress) & 0xff);
    }
}
```

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 can be specified as being `external`, `public`, `internal` or `private`, where the default is `public`. For state variables, `external` is not possible and the default is `internal`.

`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.

Nota: Everything that is inside a contract is visible to all external observers. 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 ^0.4.0;
```

```
contract C {
    function f(uint a) private 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`.

```
// This will not compile

pragma solidity ^0.4.0;

contract C {
    uint private data;

    function f(uint a) private returns(uint b) { return a + 1; }
    function setData(uint a) { data = a; }
    function getData() public returns(uint) { return data; }
    function compute(uint a, uint b) internal returns (uint) { return a+b; }
}


contract D {
    function readData() {
        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() {
        C c = new C();
        uint val = compute(3, 5);  // acces to internal member (from derivated 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`. The initialization of state variables can be done at declaration.

```
pragma solidity ^0.4.0;

contract C {
    uint public data = 42;
}


contract Caller {
```

```
    C c = new C();
    function f() {
        uint local = c.data();
    }
}
```

The getter functions have external visibility. If the symbol is accessed internally (i.e. without `this.`), it is evaluated as a state variable. If it is accessed externally (i.e. with `this.`), it is evaluated as a function.

```
pragma solidity ^0.4.0;

contract C {
    uint public data;
    function x() {
        data = 3; // internal access
        uint val = this.data(); // external access
    }
}
```

The next example is a bit more complex:

```
pragma solidity ^0.4.0;

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

It will generate a function of the following form:

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

Note that the mapping in the struct is omitted because there is no good way to provide the key for the mapping.

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 ^0.4.11;

contract owned {
    function owned() { owner = msg.sender; }
    address 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);
        _;
    }
}


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() onlyOwner {
        selfdestruct(owner);
    }
}


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;

    function Register(uint initialPrice) { price = initialPrice; }

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

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

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(!locked);
        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() noReentrancy returns (uint) {
        require(msg.sender.call());
        return 7;
    }
}
```

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

Aviso: 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`, `this.balance` or `block.number`) or execution data (`msg.gas`) or make calls to external contracts are 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;

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

View Functions

Functions can be declared `view` in which case they promise not to modify the state.

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 ^0.4.16;

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

Nota: `constant` is an alias to `view`.

Nota: Getter methods are marked `view`.

Aviso: The compiler does not enforce yet that a `view` method is not modifying state.

Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state.

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 `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 ^0.4.16;

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

Aviso: The compiler does not enforce yet that a `pure` method is not reading from the state.

Fallback Function

A contract can have exactly one unnamed function. This function cannot have arguments and cannot return anything. 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 such a context, there is usually very little gas available to the function call (to be precise, 2300 gas), so it is important to make fallback functions as cheap as possible.

In particular, the following operations will consume more gas than the stipend provided to a fallback function:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

Please ensure you test your fallback function thoroughly to ensure the execution cost is less than 2300 gas before deploying a contract.

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

Aviso: 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 fallback function.

Aviso: 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 `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 ^0.4.0;

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() { x = 1; }
    uint x;
}


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


contract Caller {
    function callTest(Test test) {
        test.call(0xabcdef01); // hash does not exist
        // results in test.x becoming == 1.

        // The following will not compile, but even
        // if someone sends ether to that contract,
        // the transaction will fail and reject the
        // Ether.
        //test.send(2 ether);
    }
}
```

Events

Events allow the convenient usage of the EVM logging facilities, which in turn can be used to «call» JavaScript callbacks in the user interface of a dapp, which listen for these events.

Events are inheritable members of contracts. When they are called, 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 and will be incorporated into the blockchain and stay there as long as a block is accessible (forever as of Frontier and Homestead, but this might change with Serenity). Log and event data is not accessible from within contracts (not even from the contract that created them).

SPV proofs for logs are possible, so if an external entity supplies a contract with such a proof, it can check that the log actually exists inside the blockchain. But be aware that block headers have to be supplied because the contract can only see the last 256 block hashes.

Up to three parameters can receive the attribute `indexed` which will cause the respective arguments to be searched for: It is possible to filter for specific values of indexed arguments in the user interface.

If arrays (including `string` and `bytes`) are used as indexed arguments, the Keccak-256 hash of it is stored as topic instead.

The hash of the signature of the event is one of the topics except if you declared the event with `anonymous` specifier. This means that it is not possible to filter for specific anonymous events by name.

All non-indexed arguments will be stored in the data part of the log.

Nota: Indexed arguments will not be stored themselves. You can only search for the values, but it is impossible to retrieve the values themselves.

```
pragma solidity ^0.4.0;
```

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

    function deposit(bytes32 _id) payable {
        // Any call to this function (even deeply nested) can
        // be detected from the JavaScript API by filtering
        // for `Deposit` to be called.
        Deposit(msg.sender, _id, msg.value);
    }
}
```

The use in the JavaScript API would be 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 will contain various information
    // including the argumets 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);
});
```

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

```
log3(
    msg.value,
    0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20,
    msg.sender,
    _id
);
```

where the long hexadecimal number is equal to `keccak256("Deposit(address,hash256,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 multiple 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.

Details are given in the following example.

```
pragma solidity ^0.4.0;

contract owned {
    function owned() { owner = msg.sender; }
    address 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() {
        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.
contract Config {
    function lookup(uint id) returns (address adr);
}


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


// 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 {
    function named(bytes32 name) {
        Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Functions can be overridden by another function with the same name and
    // the same number/types of inputs.  If the overriding function has different
    // types of output parameters, that causes an error.
    // Both local and message-based function calls take these overrides
    // into account.
    function kill() {
        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) {
      if (msg.sender == owner) info = newInfo;
   }

   function get() constant 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 ^0.4.0;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}

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

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


contract Base2 is mortal {
```

```
    function kill() { /* 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 ^0.4.0;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}

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


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


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


contract Final is Base2, Base1 {
}
```

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

Arguments for Base Constructors

Derived contracts need to provide all arguments needed for the base constructors. This can be done in two ways:

```
pragma solidity ^0.4.0;

contract Base {
    uint x;
    function Base(uint _x) { x = _x; }
}


contract Derived is Base(7) {
```

```
    function Derived(uint _y) Base(_y * _y) {
    }
}
```

One way is directly in the inheritance list (`is Base(7)`). The other is in the way a modifier would be invoked as part of the header 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. If, as in this silly example, both places are used, the modifier-style argument takes precedence.

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity follows the path of Python and uses «C3 Linearization» to force a specific order in the 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. In the following code, Solidity will give the error «Linearization of inheritance graph impossible».

```
// This will not compile

pragma solidity ^0.4.0;

contract X {}
contract A is X {}
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.

A simple rule to remember is to specify the base classes in the order from «most base-like» to «most derived».

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

Contract functions can lack an implementation as in the following example (note that the function declaration header is terminated by `;`):

```
pragma solidity ^0.4.0;

contract Feline {
    function utterance() 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;

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

contract Cat is Feline {
    function utterance() 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.

Interfaces

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

1. Cannot inherit other contracts or interfaces.
2. Cannot define constructor.
3. Cannot define variables.
4. Cannot define structs.
5. Cannot define enums.

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;

interface Token {
    function transfer(address recipient, uint amount);
}
```

Contracts can inherit interfaces as they would inherit other contracts.

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).

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 memory types 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 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 be sure to check out *using for* for a more advanced example to implement a set).

```
pragma solidity ^0.4.11;

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)
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }

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

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


contract C {
    Set.Data knownValues;

    function register(uint value) {
        // 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, take care 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 memory types and internal functions in libraries in order to implement custom types without the overhead of external function calls:

```
pragma solidity ^0.4.0;

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

    function fromUint(uint x) internal returns (bigint r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint _a, bigint _b) internal returns (bigint 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);
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint _a, uint _limb) internal returns (uint) {
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }

    function max(uint a, uint b) private returns (uint) {
        return a > b ? a : b;
    }
}


contract C {
    using BigInt for BigInt.bigint;

    function f() {
        var x = BigInt.fromUint(7);
```

```
        var y = BigInt.fromUint(uint(-1));
        var z = x.add(y);
    }
}
```

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 *Usando o Comando em linha do Compilador* 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.

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.)

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, even those where the type of the first parameter does not match the type of the object, are attached. 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 for the current scope, which is limited to a contract for now but will be lifted to the global scope later, so that by including a module, 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 ^0.4.11;

// 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)
      returns (bool)
  {
      if (self.flags[value])
        return false; // already there
      self.flags[value] = true;
      return true;
  }

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

```
  }

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


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

    function register(uint value) {
        // 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 ^0.4.0;

library Search {
    function indexOf(uint[] storage self, uint value) 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) {
        data.push(value);
    }

    function replace(uint _old, uint _new) {
        // 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.

6.4.7 Solidity Assembly

O Solidity define uma linguagem de assembly que também pode ser utilizada sem o Solidity. Essa linguagem de assembly também pode ser utilizada como «inline assembly», ou seja, pode-se utilizar o assembly em linha dentro do código fonte do Solidity. Vamos começar descrevendo como utilizar o assembly dentro do código e como isso se difere da utilização do assembly autônomo e então vamos ver as especificações do código assembly.

Nota: TODO: Escrever sobre como as regras de escopo do assembly em linha são um pouco diferentes e as complicações que surgem quando, por exemplo, utilizamos funções internas ou bibliotecas. Além disso, escrever sobre os símbolos definidos pelo compilador.

Inline Assembly

Para um controle mais fino, especialmente para aprimorar a linguagem com a escrita de bibliotecas, é possível intercalar declarações Solidity com assembly em linha em uma linguagem parecida com a utilizada pela máquina virtual. Graças ao fato da EVM ser uma maquina de empilhamento, muitas vezes é difícil endereçar o slot correto da pilha e prover argumentos para os opcodes no ponto exato na pilha. O assembly em linha do Solidity tenta facilitar isso e outros problemas decorrentes da escrita de assembly manualmente com as seguintes ferramentas:

- opcodes estilo funcional: `mul(1, add(2, 3))` ao invés de `push1 3 push1 2 add push1 1 mul`
- variáveis locais de assembly: `let x := add(2, 3) let y := mload(0x40) x := add(x, y)`
- acesso à variáveis externas: `function f(uint x) { assembly { x := sub(x, 1) } }`
- rótulos: `let x := 10 repeat: x := sub(x, 1) jumpi(repeat, eq(x, 0))`
- loops: `for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }`
- declarações switch: `switch x case 0 { y := mul(x, 2) } default { y := 0 }`
- chamadas de funções: `function f(x) -> y { switch x case 0 { y := 1 } default { y := mul(x, f(sub(x, 1))) } }`

Agora vamos descrever a linguagem assembly em linha de forma detalhada

Aviso: O assembly em linha é uma maneira de acessar a Máquina Virtual Ethereum em baixo nível. Isso descarta vários aspectos de segurança importantes do Solidity.

Exemplo

O exemplo a seguir fornece códigos de bibliotecas que acessam o código de outro contrato e o carregam na variável `bytes`. Isso seria impossível utilizando somente «Solidity puro» e a ideia é que bibliotecas assembly sejam utilizadas para aprimorar a linguagem.

```
pragma solidity ^0.4.0;

library GetCode {
    function at(address _addr) returns (bytes o_code) {
        assembly {
            // recupera o tamanho do código, isso precisa de assembly
            let size := extcodesize(_addr)
            // aloca uma saída para o vetor de bytes - isso poderia ser feito sem
↪utilizar assembly
```

```
            // utilizando o código: o_code = new bytes(size)
            o_code := mload(0x40)
            // nova "memory end" incluindo preenchimento
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // guarda o comprimento na memória
            mstore(o_code, size)
            // recupera o código, isso precisa de assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

O assembly em linha também pode ser beneficial para casos em que o otimizador falha em produzir código eficiente. Por favor, tenha em mente que assembly é muito mais difícil de escrever pois o compilador não realiza checagens, então é necessário utilizar somente para tarefas complexas e se você realmente sabe o que está fazendo.

```
pragma solidity ^0.4.12;

library VectorSum {
    // Essa função é menos eficiente pois o otimizador falha em remover
    // as checagens de limites no acesso ao vetor.
    function sumSolidity(uint[] _data) returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // Nós sabemos que só queremos acessar o vetor dentro de seus limites , então␣
↪podemos
    // ignorar a checagem. 0x20 precisa ser adicionado ao vetor pois a primeira␣
↪posição contém
    // o comprimento do vetor.
    function sumAsm(uint[] _data) 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))))
            }
        }
    }

    // Mesmo código que o anterior, mas executa todo o código dentro do assembly em␣
↪linha.
    function sumPureAsm(uint[] _data) returns (uint o_sum) {
        assembly {
           // Carrega o comprimento (primeiros 32 bytes)
           let len := mload(_data)

           // Ignora o campo de comprimento.
           //
           // Mantém uma variável para que seja incrementada localmente.
           //
           // NOTA: incrementing _data resultará em uma
           //       variável _data inutilizável após esse bloco de assembly
           let data := add(_data, 0x20)

           // Itera enquanto o limite não for atingido.
           for
               { let end := add(data, len) }
               lt(data, end)
```

```
                { data := add(data, 0x20) }
            {
                o_sum := add(o_sum, mload(data))
            }
        }
    }
}
```

Síntese

O assembly analisa comentários, literais e identificadores exatamente como o Solidity, assim você pode utilizar os comentários `//` e `/* */`. o assembly em linha é marcado por `assembly { ... }` e dentro das chaves podem ser utilizados (veja as próximas seções para mais detalhes)

- literais, ex. `0x123`, `42` ou `"abc"` (textos de até 32 caracteres)
- opcodes (em «estilo de instrução»), ex. `mload sload dup1 sstore`, veja abaixo para uma lista
- opcodes em estilo funcional, ex. `add(1, mlod(0))`
- rótulos, ex. `name:`
- declaração de variáveis, ex. `let x := 7`, `let x := add(y, 3)` ou `let x` (valor inicial de (0) é atribuído)
- identificadores (rótulos ou variáveis locais de assembly ou externas se utilizadas no assembly em linha), ex. `jump(name)`, `3 x add`
- atribuições (em «estilo de instrução»), ex. `3 =: x`
- atribuições em estilo funcional, ex. `x := add(y, 3)`
- blocos onde variáveis locais tem escopo interno, ex. `{ let x := 3 { let y := add(x, 1) } }`

Opcodes

Esse documento não tenta ser uma descrição completa da Máquina Virtual Ethereum, mas a lista a seguir pode ser utilizada como referência aos seus opcodes.

Se um opcode recebe argumentos (sempre do topo da pilha), eles são dados em parênteses. Note que a ordem dos argumentos pode ser vista invertida em estilos não funcionais (explicado abaixo). Opcodes marcados com – não inserem um item na pilha, os marcados com `*` são especiais e todas as outras inserem exatamente um item na pilha.

No código, `mem[a...b)` simboliza os bytes da memória iniciando na posição `a` até (excluindo) a posição `b` e `storage[p]` simboliza a armazenagem de conteúdo na posição `p`. Os opcodes `pushi` e `jumpdest` não podem ser utilizados diretamente.

No dicionário, opcodes são representados como identificadores pré-definidos.

stop	-	para a execução, idêntico à return(0,0)
add(x, y)		x + y
sub(x, y)		x - y
mul(x, y)		x * y
div(x, y)		x / y
sdiv(x, y)		x / y, para números assinados em complemento de 2

Table 6.1 – continued

mod(x, y)		x % y
smod(x, y)		x % y, para números assinados em complemento de 2
exp(x, y)		x elevado a y
not(x)		~x, todos os bits de x são negados
lt(x, y)		1 se x < y, senão 0
gt(x, y)		1 se x > y, senão 0
slt(x, y)		1 se x < y, senão 0, para números assinados em complemento de 2
sgt(x, y)		1 se x > y, senão 0, para números assinados em complemento de 2
eq(x, y)		1 se x == y, senão 0
iszero(x)		1 se x == 0, senão 0
and(x, y)		bitwise and de x e y
or(x, y)		bitwise or de x e y
xor(x, y)		bitwise xor de x e y
byte(n, x)		enésimo byte de x, onde o byte mais significante é o byte 0
addmod(x, y, m)		(x + y) % m com precisão aritmética arbitrária
mulmod(x, y, m)		(x * y) % m com precisão aritmética arbitrária
signextend(i, x)		extensão de sinal do(i*8+7)th bit à partir do menos significante
keccak256(p, n)		keccak(mem[p...(p+n)))
sha3(p, n)		keccak(mem[p...(p+n)))
jump(label)	-	pula para o rótulo / posição do código
jumpi(label, cond)	-	pula para o rótulo se cond é nonzero
pc		posição atual no código
pop(x)	-	remove o elemento adicionado por x
dup1 ... dup16		copia a enésima posição (à partir do topo) da pilha para o topo
swap1 ... swap16	*	troca o topmost (primeira posição da pilha) e a enésima posição da pilha
mload(p)		mem[p..(p+32))
mstore(p, v)	-	mem[p..(p+32)) := v
mstore8(p, v)	-	mem[p] := v & 0xff - modifica somente um único byte
sload(p)		storage[p]
sstore(p, v)	-	storage[p] := v
msize		tamanho da memória, ex. maior índice de memória acessada
gas		gas disponível para execução
address		endereço do contrato atual / contexto de execução
balance(a)		saldo em wei do endereço a
caller		disparador da chamada (exceto delegatecall)
callvalue		wei enviado com a chamada
calldataload(p)		dados da chamada iniciando na posição p (32 bytes)
calldatasize		tamanho da chamada em bytes
calldatacopy(t, f, s)	-	copia s bytes dos dados na posição f para mem na posição t
codesize		tamanho do código do contrato atual / contexto de execução
codecopy(t, f, s)	-	copia s bytes do codigo na posição f para mem na posição t
extcodesize(a)		tamanho do código no endereço a
extcodecopy(a, t, f, s)	-	como codecopy(t, f, s) mas utiliza o código do endereço a
returndatasize		tamanho do último returndata
returndatacopy(t, f, s)	-	copia s bytes de returndata na posição f para mem na posição t
create(v, p, s)		cria um novo contrato com o código de mem[p..(p+s)), envia v wei e retorna o endereç
create2(v, n, p, s)		cria um novo contrato com o código de mem[p..(p+s)) no endereço keccak256(<addre
call(g, a, v, in, insize, out, outsize)		chama um contrato no endereço a com os dados em mem[in..(in+insize)) provendo g g
callcode(g, a, v, in, insize, out, outsize)		idêntico ao *call* mas só utiliza o código de a, além de manter o contexto de execução d
delegatecall(g, a, in, insize, out, outsize)		idêntico ao *callcode* mas também mantém `caller` e `callvalue`

Table 6.1 – continued

staticcall(g, a, in, insize, out, outsize)		idêntico ao *call(g, a, 0, in, insize, out, outsize)* mas não permite modificação de estado
return(p, s)	-	finaliza a execução, retorna os dados em mem[p..(p+s))
revert(p, s)	-	finaliza a execução, reverte as mudanças de estado e retorna os dados em mem[p..(p+s
selfdestruct(a)	-	finaliza a execução, destrói o contrato atual e envia o saldo para o contrato a
invalid	-	finaliza a execução com instrução inválida
log0(p, s)	-	loga sem tópicos os dados em mem[p..(p+s))
log1(p, s, t1)	-	loga com os tópico t1 os dados em mem[p..(p+s))
log2(p, s, t1, t2)	-	loga com os tópicos t1 e t2 os dados em mem[p..(p+s))
log3(p, s, t1, t2, t3)	-	loga com os tópicos t1, t2 e t3 os dados em mem[p..(p+s))
log4(p, s, t1, t2, t3, t4)	-	loga com os tópicos t1, t2, t3 e t4 os dados em mem[p..(p+s))
origin		quem enviou a transação
gasprice		preço do gas da transação
blockhash(b)		hash do bloco b - somente para os últimos 256 blocos excluindo o bloco atual
coinbase		beneficiário da mineração
timestamp		timestamp do bloco atual em segundos desde a época (1970-01-01T00:00:00Z)
number		número do bloco atual
difficulty		dificuldade do bloco atual
gaslimit		limite de gas do bloco atual

Literais

Você pode utilizar constantes inteiras (integer) digitando-as em decimal ou hexadecimal e a instrução apropriada `PUSHi` vai ser automaticamente gerada. O trecho a seguir cria um código para adicionar 2 e 3 resultando em 5 e então computa o bitwise and com o texto «abc». Textos são guardados alinhados à esquerda e não podem conter mais de 32 bytes.

```
assembly { 2 3 add "abc" and }
```

Estilo Funcional

Você pode digitar opcode após opcode da mesma maneira que eles ficarão em bytecode. Por exemplo adicionar `3` ao conteúdo da memória na posição `0x80` seria

```
3 0x80 mload add 0x80 mstore
```

Como é difícil ver quais são os argumentos atuais para certos opcodes, o aseembly em linha do Solidity provê uma notação de «estilo funcional» onde o mesmo código seria escrito como

```
mstore(0x80, add(mload(0x80), 3))
```

As expressões em estilo funcional não podem utilizar estilo instrucional internamente, ex. `1 2 mstore(0x80, add)` não é um assembly válido, o correto seria escrever como `mstore(0x80, add(2, 1))`. Para opcodes que não recebem argumentos, os parênteses podem ser omitidos.

Note que a ordem dos argumentos é invertida em estilo funcional, o oposto do estilo instrucional. Se você utiliza estilo funcional, o primeiro argumento vai ficar no topo da pilha.

Acesso à Variaveis Externas e Funções

Vaiáveis de Solidity e outros identificadores podem ser acessados simplesmente utilizando seu nome. Para variáveis de memória, isso insere seu endereço e não seu valor na pilha. Variáveis de armazenamento são diferentes: Valores no armazenamento podem não ocupar um espaço de armazenamento completo na memória, por isso seu «endereço» é composto de um espaço na memória e um byte-offset dentro do espaço. Para recuperar o conteúdo da variável `x`, voce deve utilizar `x_slot` e para recuperar o byte-offset você deve utilizar `x_offset`.

Em atribuições (veja abaixo), nós podemos utilizar variáveis locais do Solidity para atribuir um valor.

Funções externas ao assembly em linha também pode ser acessado: O assembly irá inserir seu rótulo de entrada (com a resolução da função virtual aplicada). A semântica para chamada em solidity é: Functions external to inline assembly can also be accesse

- o consumidor chama return label, arg1, arg2, . . . , argn
- a chamada retorna com ret1, ret2, . . . , retm

Esse recurso ainda é um pouco moroso para usar, porque a pilha essencialmente muda durante a chamada, e desta forma referências para variáveis locais estarão erradas.

```
pragma solidity ^0.4.11;

contract C {
    uint b;
    function f(uint x) returns (uint r) {
        assembly {
            r := mul(x, sload(b_slot)) // ignore o offset, sabemos que é 0
        }
    }
}
```

Rótulos

Outro problema com assembly de EVM é que `jump` e `jumpi` utilizam endereços absolutos que podem mudar facilmente. Assembly em linha do solidity provê rótulos para facilitar o uso de pulos (jumps). Note que rótulos são um recurso de baixo nível e que é possível escrever assemblies eficientes sem rótulos, utilizando apenas funções de assembly, loops e instruções switch (veja abaixo). O código a seguir computa um elemento na série Fibonacci.

```
{
    let n := calldataload(4)
    let a := 1
    let b := a
loop:
    jumpi(loopend, eq(n, 0))
    a add swap1
    n := sub(n, 1)
    jump(loop)
loopend:
    mstore(0, a)
    return(0, 0x20)
}
```

Perceba que o acesso automático às variáveis da pilha só funciona se o assembler sabe a altura atual da pilha. Isso não funciona se a origem do salto e o destino possuem diferentes alturas de pilha. Ainda é possível utilizar tais saltos, mas você não deve acessar nenhuma variável da pilha (incluindo variáveis de assembly) neste caso.

Além disso, o analisador da altura de pilha analisa o código opcode por opcode (e não de acordo com o fluxo de controle), então nesse caso, o assembler terá a impressão incorreta sobre a altura da pilha no rótulo `two`:

```
{
    let x := 8
    jump(two)
    one:
        // Aqui a altura da pilha é 2 (porque nós inserimos x e 7),
        // mas o assembler pensa que é 1 por que ele lê
        // do topo pra baixo.
        // Acessar a variável da pilha x aqui vai resultar em erro.
        x := 9
        jump(three)
    two:
        7 // insira algo na pilha
        jump(one)
    three:
}
```

Esse problema pode ser resolvido ajustando manualmente a altura da pilha para o assembler - você pode prover uma altura de pilha delta que é adicionada à altura da pilha antes do rótulo. Note que que você não precisa se preocupar com isso se utilizar loops e funções de nivel de assembly.

O código a seguir é um exemplo de como isso pode ser feito em casos extremos.

```
{
    let x := 8
    jump(two)
    0 // Esse código é inacessível mas ajusta a altura da pilha corretamente
    one:
        x := 9 // Agora x pode ser acessado.
        jump(three)
        pop // Correção negativa similar.
    two:
        7 // insere algo na pilha
        jump(one)
    three:
    pop // Temos que remover o valor inserido manualmente novamente.
}
```

Declarando Variáveis Locais de Assembly

Você pode usar a palavra `let` para declarar variáveis visíveis somente no assembly em linha e somente no bloco `{...}```atual. A instrução ``let` vai criar um novo espaço na pilha reservado para a variável, que será automaticamente removido quando o bloco chegar ao final. Você precisa prover um valor inicial para a variável, que pode ser apenas `0`, mas também pode ser alguma expressão de estilo funcional complexo.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint x) returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
                b := y
```

```
            } // y é "desalocado" aqui
            b := add(b, v)
        } // v é "desalocado" aqui
    }
}
```

Atribuições

Atribuições são possíveis para variáveis locais do assembly e para variáveis locais de função. Fique atento pois quando voce atribui valores que apontam para memória ou armazenamento à uma variável, você vai alterar apenas o ponteiro e não os dados.

Existem dois tipos de atribuições: de estilo funcional e de estilo instrucional. Para atribuições de estilo funcional (`variable := value`), voce precisa fornecer um valor em expressão funcional que resulte em exatamente um valor de pilha. Para atribuições de estilo instrucional (`=: variable`), o valor é o mesmo do topo da pilha. Em ambos os casos, os «dois pontos» apontam para o nome da variável. A atribuiçao é realizada substituindo o valor da variável na pilha pelo novo valor.

```
{
    let v := 0 // atribuição de estilo funcional realizada como parte da declaração
↪da variável
    let g := add(v, 2)
    sload(10)
    =: v // atribuição dem estilo instrucional, colcoa o valor de sload(10) em v
}
```

Switch

Você pode utilizar uma declaração de switch como uma versão básica de «if/else». Ela pega o valor da expressão e compara à várias constantes. O código da constante correspondente é selecionado. Diferente do comportamento visto em outras linguagens de programação, o fluxo de controle não continua para o próximo caso após ser executado. Pode existir um recurso padrão chamado de `default` que é executado caso todos os testes falhem.

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

A lista de casos não reuqer chaves, mas o corpo do caso sim.

Loops

O assembly suporta um loop estilo for simples. Loops for possuem um cabeçalho contendo uma parte de inicialização, uma condição e uma parte pós iteração. A condição deve ser uma expressão de estilo funcional enquanto as outras

duas são blocos. Se a parte de inicialização declara qualquer variável, o escopo dessas variáveis é extendido para o corpo da função (incluindo a condição e a parte pós iteração).

O exemplo a seguir computa a soma de uma área na memória.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

Loops for podem também ser escritos de uma maneira para que se comportem como loops while: Basta deixar a parte de inicialização e pós iteração vazias.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {      // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

Funções

O código assembly permite a definição de funções de baixo nível. Essas funções recebem seus argumentos (e retornam um PC) da pilha e também inserem seus resultados na pilha. Chamar uma função é bem parecido com executar um opcode de estilo funcional.

Funções podem ser definidaas em qualquer local do código e é visível no bloco onde foram declaradas. Dentro da função você não pode acessar variáveis locais definidas fora da função. Não existe `return` explícito.

Se você chamar uma função que retorna múltiplos valores, você deve atribuí-los À um tuple utilizando `a, b := f(x)` ou `let a, b := f(x)`.

O exemplo a seguir implementa a função potência utilizando raiz quadrada e multiplicação.

```
{
    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) }
        }
    }
}
```

Coisas a Evitar

Assembly em linha pode parecer uma linguagem de alto nível, mas é uma linguagem extramente de baixo nível. Chamadas de funções, loops e switches são convertidos simplesmente reescrevendo regras e, depois disso, a única coisa que o assembler faz par você é rearranjar opcodes de estilo funcional, administrando rótulos de jump, contando

a altura da pilha para acesso à variáveis e removendo espaços de pilha para variáveis locais de assembly ao final dos blocos. Especialmente para os dois últimos dois casos, é importante saber que o assembler só conta a altura da pilha do topo para baixo, não necessariamente seguindo o fluxo de controle. Além disso, operações como swap só vão modificar o conteúdo da pilha e não o local das variáveis.

Convenções em Solidity

Em contraste ao assembly EVM, o Solidity sabe os tipos que são menores que 256 bits, ex. `uint24`. Para ser mais eficientes, a maior parte das operações aritmética as trata como números de 256-bit e os bits de ordem maior só são limpos quando necessário, ex. pouco antes de serem escritos na memória ou de comparações serem efetuadas. Isso faz com que se você acessar esse tipo de variável de dentro do assembly em linha, você deve limpar manualmente os bits de ordem maior primeiro.

O solidity gerencia a memória de uma maneira simples: Existe um «ponteiro de memória livre» na posição `0x40` da memória. Se você quiser alocar memória basta utilizar a memória daquele ponto e realizar as alterações necessárias.

Elementos em vetores de memória no Solidity sempre ocupa múltiplos de 32 bytes (sim, até mesmo para `byte[]`, mas não para `bytes` e `string`). Vetores multi dimensionais de memória são ponteiros para vetores de memória. O comprimento de um vetor dinâmico é armazenado na primeira posição do vetor, depois disso os elemntos são armazenados normalmente.

Aviso: Vetores de memória estáticamente dimensionados não possuem um espaço para o comprimento, mas isso vai ser alterado em breve para melhor convertibilidade entre vetores estáticos e dinâmicos. Por favor, não confie nesta conversão.

Assembly Autônomo

A linguagem assembly descrita como assembly em linha também pode ser utilizada de forma autônoma. Na verdade o plano é que ela seja utilizada como linguagem intermediária para o compilador Solidity. Desta forma, ela tenta atingir diversos objetivos:

1. Programas escritos na linguagem devem ser legíveis, mesmo que o código tenha sido gerado por um compilador Solidity.
2. A tradução do assembly para o bytecode deve conter a menor quantidade de «surpresas» possível.
3. O controle de fluxo deve ser fácil de detectar para ajudar na verificação formal e otimização.

Para atingir o primeiro e último objetivos, o assembly fornece conceitos de alto nível como loops `for`, declarações de `switch` e chamadas de função. Deve ser possível escrever programas assembly que não fazem uso explícito das declarações `SWAP`, `DUP`, `JUMP` e `JUMPI`, pois os dois primeiros ofuscam o fluxo de data e os dois últimos ofuscam o fluxo de controle. Além disso, declarações funcionais na forma `mul(add(x, y), 7)` são preferidas ao invés de opcode puro como `7 y x add mul` pois na primeira forma é muito mais fácil ver qual operador é utilizado para cada opcode.

O segundo objetivo é atingido introduzindo a fase de desaçucarização que remove somente os níveis mais altos de conceitos de uma maneira padrão que ainda permite a inspeção do código assembly de baixo nível gerado pelo assembler. A única operação não local feita pelo assembler é a pesquisa de nomes para identificadores definidos pelo usuário (funções, variáveis, ...), que seguem regras simples de escopo e limpeza de variáveis locais da pilha.

Âmbito: Um identificador que é declarado (rótulo, variável, função, assembly) só é visível no bloco onde foi declarado (incluindo blocos aninhados detro do bloco corrente). Não é permitido acessar variáveis locais entre fronteiras de funções, mesmo que elas estejam no mesmo escopo. Shadowing não é permitido. Variáveis locais não podem ser acessadas antes de serem declaradas, mas rótulos, funções e assemblies podem. Assemblies são blocos especiais que

por ex. retornem código de tempo de execução ou criem contratos. Nenhum identificador de um assembly externo é visível em um sub-assembly.

Se o fluxo de controle passar do fim de um bloco, instruções de pop são inseridas para corresponder ao número de variáveis locais declaradas no bloco. Sempre que uma variável local é referenciada, o gerador de código precisa saber a posição relativa na pilha e consequentemente precisa manter o registro da então altura da pilha. Já que todas as variáveis locais são removidas ao final do bloco, a altura do bloco deverá ser a mesma. Se este não for o caso, um alerta é disparado.

Por que utilizamos conceitos de alto nível como `switch`, `for` e funções:

Com a utilização de `switch`, `for` e funções, é possível escrever códigos complexos sem a necessidade de utilizar `jump` ou `jumpi` manualmente. Isso facilita a análize do controle de fluxo, o que melhora a verificação formal e otimização.

Além disso, se saltos (jump) manuais fossem permitidos, computar a altura da pilha seria muito complicado. A posição de todas as variáveis na pilha precisam ser conhecidas, senão nenhuma das referências às variáveis locais nem a remoção automática de variáveis locais iriam funcionar. O mecanismo de desaçucarização insere operações corretamente em blocos inatingíveis para ajustar a altura do bloco adequadamente em casos de saltos (jump) que não possuem um controle de fluxo contínuo.

Exemplo:

Vamos ver um exemplo de compilção de Solidity para um assembly desaçucarizado. Considere o bytecode de tempo de execução do seguinte programa Solidity:

```
pragma solidity ^0.4.0;

contract C {
  function f(uint x) returns (uint y) {
    y = 1;
    for (uint i = 0; i < x; i++)
      y = 2 * y;
  }
}
```

O assembly a seguir será gerado:

```
{
  mstore(0x40, 0x60) // guarda o "ponteiro de memória livre"
  // 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) }
  // alocação de memória
  function $allocate(size) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, size))
  }
  // a função do contrato
  function f(x) -> y {
    y := 1
    for { let i := 0 } lt(i, x) { i := add(i, 1) } {
      y := mul(2, y)
    }
```

```
  }
}
```

Após a desaçucarização esse é o resultado:

```
{
  mstore(0x40, 0x60)
  {
    let $0 := div(calldataload(0), exp(2, 226))
    jumpi($case1, eq($0, 0xb3de648b))
    jump($caseDefault)
    $case1:
    {
      // a chamada da função - colocamos o rótulo de retorno e argumentos na pilha
      $ret1 calldataload(4) jump(f)
      // Esse código é inacessível. Opcodes são adicionados para espelhar o
      // o efeito da função na altura da pilha: argumentos
      // são removidos e valores de retorno inseridos.
      pop pop
      let r := 0
      $ret1: // o ponto real de retorno
      $ret2 0x20 jump($allocate)
      pop pop let ret := 0
      $ret2:
      mstore(ret, r)
      return(ret, 0x20)
      // apesar de ser inútil, o salto (jump) é automaticamente inserido,
      // já que o processo de desaçucarização é uma operação puramente sintática
      // que não analisa o o fluxo de controle
      jump($endswitch)
    }
    $caseDefault:
    {
      revert(0, 0)
      jump($endswitch)
    }
    $endswitch:
  }
  jump($afterFunction)
  allocate:
  {
    // saltamos o código inacessível que introduz os argumentos da função
    jump($start)
    let $retpos := 0 let size := 0
    $start:
    // variáveis de saída vivem no mesmo escopo que os argumentos e estão
    // atualmente alocados.
    let pos := 0
    {
      pos := mload(0x40)
      mstore(0x40, add(pos, size))
    }
    // Esse código substitui os argumentos pelos valores de retorno e salta (jump) de␣
↪volta.
    swap1 pop swap1 jump
    // Novamente código inacessível corrige a altura da pilha.
    0 0
  }
```

```
  f:
  {
    jump($start)
    let $retpos := 0 let x := 0
    $start:
    let y := 0
    {
      let i := 0
      $for_begin:
      jumpi($for_end, iszero(lt(i, x)))
      {
        y := mul(2, y)
      }
      $for_continue:
      { i := add(i, 1) }
      jump($for_begin)
      $for_end:
    } // Aqui, uma instrução pop será inserida para i
    swap1 pop swap1 jump
    0 0
  }
  $afterFunction:
  stop
}
```

O assembly acontece em quatro estágios:

1. Análise
2. desaçucarização (remove switch, for e funções)
3. Gera o fluxo de opcodes
4. Geração de bytecode

Vamos especificar os passos de um a três de uma maneira pseudo formal. Especificações mais formais virão em seguida.

Análise / Dicionário

As tarefas do analisador são as seguintes:

- Transforma o fluxo de bytes em um fluxo de tokens, descarta comentários estilo C++ (um comentário especial existe para referência de origem, mas não será explicado aqui).
- Transforma o fluxo de tokens em um AST de acordo com o dicionário abaixo
- Registra identificadores com o bloco onde estão definidos (anotação no nó AST) e anota à partir de qual ponto as variáveis podem ser acessadas.

O lexer do assembly é definido pelo próprio Solidity.

Espaço em branco é utilizado para delimitar tokens e consiste nos caracteres Espaço, Tab e nova linha. Comentários no padrão JavaScript/C++ são interpretados como espaço em branco.

Grammar:

```
AssemblyBlock = '{' AssemblyItem* '}'
AssemblyItem =
    Identifier |
```

```
    AssemblyBlock |
    FunctionalAssemblyExpression |
    AssemblyLocalDefinition |
    FunctionalAssemblyAssignment |
    AssemblyAssignment |
    LabelDefinition |
    AssemblySwitch |
    AssemblyFunctionDefinition |
    AssemblyFor |
    'break' | 'continue' |
    SubAssembly | 'dataSize' '(' Identifier ')' |
    LinkerSymbol |
    'errorLabel' | 'bytecodeSize' |
    NumberLiteral | StringLiteral | HexLiteral
Identifier = [a-zA-Z_$] [a-zA-Z_0-9]*
FunctionalAssemblyExpression = Identifier '(' ( AssemblyItem ( ',' AssemblyItem )* )?
↪')'
AssemblyLocalDefinition = 'let' IdentifierOrList ':=' FunctionalAssemblyExpression
FunctionalAssemblyAssignment = IdentifierOrList ':=' FunctionalAssemblyExpression
IdentifierOrList = Identifier | '(' IdentifierList ')'
IdentifierList = Identifier ( ',' Identifier)*
AssemblyAssignment = '=:' Identifier
LabelDefinition = Identifier ':'
AssemblySwitch = 'switch' FunctionalAssemblyExpression AssemblyCase*
    ( 'default' AssemblyBlock )?
AssemblyCase = 'case' FunctionalAssemblyExpression AssemblyBlock
AssemblyFunctionDefinition = 'function' Identifier '(' IdentifierList? ')'
    ( '->' '(' IdentifierList ')' )? AssemblyBlock
AssemblyFor = 'for' ( AssemblyBlock | FunctionalAssemblyExpression)
    FunctionalAssemblyExpression ( AssemblyBlock | FunctionalAssemblyExpression)␣
↪AssemblyBlock
SubAssembly = 'assembly' Identifier AssemblyBlock
LinkerSymbol = 'linkerSymbol' '(' StringLiteral ')'
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]+
```

Desaçucarização

Uma transformação AST remove conceitos de for, switch e funções. O resultado ainda é analisável pelo mesmo analisador, mas não utilizará certos conceitos. Se jumpdests são adicionados e são apenas saltados para e não continuados em, informação sobre a pilha é adicionada, ao menos que variáveis locais de fora do escopo não sejam acessadas e que a altura da pilha seja a mesma que na última instrução.

Pseudocode:

```
desugar item: AST -> AST =
match item {
AssemblyFunctionDefinition('function' name '(' arg1, ..., argn ')' '->' ( '(' ret1, ..
↪., retm ')' body) ->
  <name>:
  {
    jump($<name>_start)
    let $retPC := 0 let argn := 0 ... let arg1 := 0
```

```
    $<name>_start:
    let ret1 := 0 ... let retm := 0
    { desugar(body) }
    swap and pop items so that only ret1, ... retm, $retPC are left on the stack
    jump
    0 (1 + n times) to compensate removal of arg1, ..., argn and $retPC
  }
AssemblyFor('for' { init } condition post body) ->
  {
    init // cannot be its own block because we want variable scope to extend into the␣
↪body
    // find I such that there are no labels $forI_*
    $forI_begin:
    jumpi($forI_end, iszero(condition))
    { body }
    $forI_continue:
    { post }
    jump($forI_begin)
    $forI_end:
  }
'break' ->
  {
    // find nearest enclosing scope with label $forI_end
    pop all local variables that are defined at the current point
    but not at $forI_end
    jump($forI_end)
    0 (as many as variables were removed above)
  }
'continue' ->
  {
    // find nearest enclosing scope with label $forI_continue
    pop all local variables that are defined at the current point
    but not at $forI_continue
    jump($forI_continue)
    0 (as many as variables were removed above)
  }
AssemblySwitch(switch condition cases ( default: defaultBlock )? ) ->
  {
    // find I such that there is no $switchI* label or variable
    let $switchI_value := condition
    for each of cases match {
      case val: -> jumpi($switchI_caseJ, eq($switchI_value, val))
    }
    if default block present: ->
      { defaultBlock jump($switchI_end) }
    for each of cases match {
      case val: { body } -> $switchI_caseJ: { body jump($switchI_end) }
    }
    $switchI_end:
  }
FunctionalAssemblyExpression( identifier(arg1, arg2, ..., argn) ) ->
  {
    if identifier is function <name> with n args and m ret values ->
      {
        // find I such that $funcallI_* does not exist
        $funcallI_return argn  ... arg2 arg1 jump(<name>)
        pop (n + 1 times)
        if the current context is `let (id1, ..., idm) := f(...)` ->
```

```
          let id1 := 0 ... let idm := 0
          $funcallI_return:
        else ->
          0 (m times)
          $funcallI_return:
          turn the functional expression that leads to the function call
          into a statement stream
      }
    else -> desugar(children of node)
  }
default node ->
  desugar(children of node)
}
```

Geração de Fluxo de Opcode

Durante a geração do fluxo de opcode mantemos registro da altura da pilha em um contador para que seja possível acessar as variáveis à partir de seus nomes. A altura da pilha é modificada com cada opcode que modifica a pilha e com cada rótulo que não é anotado com um ajuste de pilha. Toda vez que uma nova variável local é introduzida ela é registrada com a altura da pilha. Se uma variável é acessada (seja para cópia do valor ou para atribuição), a instrução aprorpiada `DUP` ou `SWAP` é selecionada dependendo da diferença entre a altura atual da pilha e a altura da pilha no momento em que a variável foi introduzida.

Pseudocode:

```
codegen item: AST -> opcode_stream =
match item {
AssemblyBlock({ items }) ->
  join(codegen(item) for item in items)
  if last generated opcode has continuing control flow:
    POP for all local variables registered at the block (including variables
    introduced by labels)
    warn if the stack height at this point is not the same as at the start of the␣
↪block
Identifier(id) ->
  lookup id in the syntactic stack of blocks
  match type of id
    Local Variable ->
      DUPi where i = 1 + stack_height - stack_height_of_identifier(id)
    Label ->
      // referência  que deve ser resolvida durante a geração de bytecode
      PUSH<bytecode position of label>
    SubAssembly ->
      PUSH<bytecode position of subassembly data>
FunctionalAssemblyExpression(id ( arguments ) ) ->
  join(codegen(arg) for arg in arguments.reversed())
  id (which has to be an opcode, might be a function name later)
AssemblyLocalDefinition(let (id1, ..., idn) := expr) ->
  register identifiers id1, ..., idn as locals in current block at current stack␣
↪height
  codegen(expr) - assert that expr returns n items to the stack
FunctionalAssemblyAssignment((id1, ..., idn) := expr) ->
  lookup id1, ..., idn in the syntactic stack of blocks, assert that they are␣
↪variables
  codegen(expr)
  for j = n, ..., i:
```

```
  SWAPi where i = 1 + stack_height - stack_height_of_identifier(idj)
  POP
AssemblyAssignment(=: id) ->
  look up id in the syntactic stack of blocks, assert that it is a variable
  SWAPi where i = 1 + stack_height - stack_height_of_identifier(id)
  POP
LabelDefinition(name:) ->
  JUMPDEST
NumberLiteral(num) ->
  PUSH<num interpreted as decimal and right-aligned>
HexLiteral(lit) ->
  PUSH32<lit interpreted as hex and left-aligned>
StringLiteral(lit) ->
  PUSH32<lit utf-8 encoded and left-aligned>
SubAssembly(assembly <name> block) ->
  append codegen(block) at the end of the code
dataSize(<name>) ->
  assert that <name> is a subassembly ->
  PUSH32<size of code generated from subassembly <name>>
linkerSymbol(<lit>) ->
  PUSH32<zeros> and append position to linker table
}
```

6.4.8 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).

Aviso: 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.

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 an (unfilled) slot in storage at some position `p` according to the above rule (or by recursively applying this rule for mappings to mappings or arrays of arrays). For a dynamic array, this slot stores the number of elements in the array (byte arrays and strings are an exception here, see below). For a mapping, 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)`.

`bytes` and `string` store their data in the same slot where also the length is stored if they are short. 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`. If it is longer, the main slot stores `length * 2 + 1` and the data is stored as usual in `keccak256(slot)`.

So for the following contract snippet:

```
pragma solidity ^0.4.0;

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`.

Layout in Memory

Solidity reserves three 256-bit slots:

- 0 - 64: scratch space for hashing methods
- 64 - 96: currently allocated memory size (aka. free memory pointer)

Scratch space can be used between statements (ie. within inline assembly).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

Aviso: 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 lifecycle, 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 be zeroed out.

Layout of Call Data

When a Solidity contract is deployed and when it is called from an account, the input data is assumed to be in the format in *the ABI specification*. The ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

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

```
var 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 indentifiers 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.

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::2;;`

Contract Metadata

The Solidity compiler automatically generates a JSON file, the contract metadata, that contains information about the current contract. It can be used to query the compiler version, the sources used, the ABI and NatSpec documentation in order to more safely interact with the contract and to 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.

Of course, you have to publish the metadata file to Swarm (or some other service) so that others can access it. The file can be output by using `solc --metadata` and the file will be called `ContractName_meta.json`. It will contain 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 of course also 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:
  {
    "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: [ ... ],
  }
}
```

Nota: Note the ABI definition above has no fixed order. It can change with compiler versions.

Nota: Since the bytecode of the resulting contract contains the metadata hash, any change to the metadata will result in a change of the bytecode. Furthermore, since the metadata includes a hash of all the sources used, a single whitespace change in any of the source codes will result in a different metadata, and subsequently a different bytecode.

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.

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) 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, Mist can use the userdoc to display a confirmation message to the user whenever they interact with the contract.

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.

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*.
- If your contract has a function called `send` but you want to use the built-in send-function, use `address(contractVariable).send(amount)`.
- Initialise storage structs with a single assignment: `x = MyStruct({a: 1, b: 2});`

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 plus and 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

- `block.blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks
- `block.coinbase` (`address`): 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
- `msg.data` (`bytes`): complete calldata
- `msg.gas` (`uint`): remaining gas
- `msg.sender` (`address`): 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`): 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)
- `revert()`: abort execution and revert state changes
- `keccak256(...) returns (bytes32)`: compute the Ethereum-SHA-3 (Keccak-256) hash of the (tightly packed) arguments
- `sha3(...) returns (bytes32)`: an alias to `keccak256`
- `sha256(...) returns (bytes32)`: compute the SHA-256 hash of the (tightly packed) arguments
- `ripemd160(...) returns (bytes20)`: compute the RIPEMD-160 hash of the (tightly packed) arguments
- `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`
- `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`
- `this` (current contract's type): the current contract, explicitly convertible to `address`
- `super`: the contract one level higher in the inheritance hierarchy
- `selfdestruct(address recipient)`: destroy the current contract, sending its funds to the given address
- `suicide(address recipieint)`: an alias to `selfdestruct`
- `<address>.balance` (`uint256`): balance of the *Address* in Wei

- `<address>.send(uint256 amount) returns (bool)`: send given amount of Wei to *Address*, returns `false` on failure
- `<address>.transfer(uint256 amount)`: send given amount of Wei to *Address*, throws on failure

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 - this is not enforced yet.
- `view` for functions: Disallows modification of state - this is not enforced yet.
- `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.
- `constant` for functions: Same as `view`.
- `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`, `case`, `catch`, `default`, `final`, `in`, `inline`, `let`, `match`, `null`, `of`, `relocatable`, `static`, `switch`, `try`, `type`, `typeof`.

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 )* ')' )?

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 IndexedParameterList 'anonymous'? ';'

EnumValue = Identifier
EnumDefinition = 'enum' Identifier '{' EnumValue? (',' EnumValue)* '}'

IndexedParameterList = '(' ( TypeName 'indexed'? Identifier? (',' TypeName 'indexed'?␣
↪Identifier?)* )? ')'
ParameterList =        '(' ( TypeName            Identifier? (',' TypeName          ␣
↪Identifier?)* )? ')'
TypeNameList =         '(' ( TypeName (',' TypeName )* )? ')'

// semantic restriction: mappings and structs (recursively) containing mappings
// are not allowed in argument lists
VariableDeclaration = TypeName StorageLocation? Identifier

TypeName = ElementaryTypeName
         | UserDefinedTypeName
         | Mapping
         | ArrayTypeName
         | FunctionTypeName

UserDefinedTypeName = Identifier ( '.' Identifier )*

Mapping = 'mapping' '(' ElementaryTypeName '=>' TypeName ')'
ArrayTypeName = TypeName '[' Expression? ']'
FunctionTypeName = 'function' TypeNameList ( 'internal' | 'external' |␣
↪StateMutability )*
                   ( 'returns' TypeNameList )?
StorageLocation = 'memory' | 'storage'
StateMutability = 'pure' | 'constant' | 'view' | 'payable'

Block = '{' Statement* '}'
Statement = IfStatement | WhileStatement | ForStatement | Block |␣
↪InlineAssemblyStatement |
            ( DoWhileStatement | PlaceholderStatement | Continue | Break | Return |
              Throw | SimpleStatement ) ';'
```

```
ExpressionStatement = Expression
IfStatement = 'if' '(' Expression ')' Statement ( 'else' Statement )?
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'
VariableDefinition = ('var' IdentifierList | VariableDeclaration) ( '=' Expression )?
IdentifierList = '(' ( Identifier? ',' )* Identifier? ')'

// 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
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]+

TupleExpression = '(' ( Expression ( ',' Expression )*  )? ')'
                | '[' ( Expression ( ',' Expression )*  )? ']'

ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | 'var'
                   | 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' DecimalNumber 'x' DecimalNumber )

Uixed = 'ufixed' | ( 'ufixed' DecimalNumber 'x' DecimalNumber )

InlineAssemblyBlock = '{' AssemblyItem* '}'

AssemblyItem = Identifier | FunctionalAssemblyExpression | InlineAssemblyBlock |␣
↪AssemblyLocalBinding | AssemblyAssignment | AssemblyLabel | NumberLiteral |␣
↪StringLiteral | HexLiteral
AssemblyLocalBinding = 'let' Identifier ':=' FunctionalAssemblyExpression
AssemblyAssignment = ( Identifier ':=' FunctionalAssemblyExpression ) | ( '=:'␣
↪Identifier )
AssemblyLabel = Identifier ':'
FunctionalAssemblyExpression = Identifier '(' AssemblyItem? ( ',' AssemblyItem )* ')'
```

6.5 Considerações de Segurança

Embora geralmente seja bastante fácil criar software que funcione como o esperado, é muito mais difícil verificar que ninguém possa usá-lo de uma forma **não** antecipada.

No Solidity, isso é ainda mais importante porque você pode usar contratos inteligentes (smart contracts) para manipular tokens ou, possivelmente, coisas ainda mais valiosas. Além disso, cada execução de um contrato inteligente acontece em público e, adicionalmente, o código-fonte está frequentemente disponível.

É claro que você sempre tem que considerar o quanto está em jogo: Você pode comparar um smart contract com um serviço da Web aberto para o público (e, portanto, também para atores mal-intencionados) e talvez até fonte aberta. Se você apenas armazena sua lista de compras nesse serviço da Web, talvez você não tenha que se preocupar muito, mas se você gerencia sua conta bancária usando esse serviço web, você deve ser mais cuidadoso.

Esta seção listará algumas armadilhas e recomendações gerais de segurança, mas pode, é claro, nunca estar completa. Além disso, tenha em mente que, mesmo que o seu código do contrato inteligente seja livre de erros, o compilador ou a própria plataforma pode tem um bug. Uma lista de alguns erros conhecidos de segurança pública do compilador pode ser encontrado no *list of known bugs*, que também é legível por máquina.

Note que existe um programa de recompensa de erros (bugs) que abrange o gerador de código do compilador de Solidity.

Como sempre, com documentação open source, ajude-nos a expandir esta seção Especialmente, com alguns exemplos que não causem problemas!

6.5.1 Armadilhas (Pitfalls)

Informações Privativas e Aleatoriedade

Tudo o que você usa em Smart Contracts está visível ao público, mesmo variáveis locais (local variables) e variáveis de estado (state variables) marcadas como privativas `private`

Usar números aleatórios em contratos inteligentes é bastante complicado se você não quiser mineiros habilitados para trapacear.

Re-Entrancy

Qualquer interação de um contrato (A) com outro contrato (B) e qualquer transferência de Ether, entrega o controle a esse contrato (B). Isso permite que (B) chamar de volta para (A) antes que esta interação seja concluída. Para dar um exemplo, O código a seguir contém um erro (é apenas um trecho e não um contrato completo):

```
pragma solidity ^0.4.0;

// 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() {
        if (msg.sender.send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

O problema não é muito grave aqui por causa do gás limitado como parte de `send`, mas ainda assim expõe uma fraqueza: transferência de Ether sempre inclui a execução do código, para que o destinatário possa ser um contrato

que chama de volta para `withdraw`. Isso permitiria obter reembolsos múltiplos e basicamente, recupera todo o Ether no contrato.

Para evitar re-entrancy, você pode usar o padrão Checks-Effects-Interactions como descrito abaixo:

```
pragma solidity ^0.4.11;

contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() {
        var share = shares[msg.sender];
        shares[msg.sender] = 0;
        msg.sender.transfer(share);
    }
}
```

Perceba que re-entrancy não somente um efeito de transferência de Ether, mas de qualquer função chamada em outro contrato. Além disso, você pode ter situações de multi-contratos dentro da conta. Um contrato chamado pode modificar o estado do outro contrato de que você depende.

Gas Limit e Loops

Loops que não tenham um número fixo de interações, por exemplo, loops que dependem de valores armazenados, tem que ser usados com muito cuidado: Devido ao limite de Gas por bloco, as transações podem consumir somente uma certa quantidade de Gas. Explicitamente ou apenas devido a operação normal, o número de interações em um loop pode crescer além do limite de gás de bloqueio, que pode causar o completo contrato para ser parado em um determinado ponto. Isso pode não se aplicar às funções `constant` que são executadas apenas para ler dados da cadeia de blocos. Ainda assim, tais funções podem ser chamadas por outros contratos como parte das operações na cadeia e paralisar aqueles.

Seja pormenorizado sobre tais casos na documentação dos seus contratos.

Mandando e Recebendo Ether

- Nem os contratos nem «external accounts» atualmente podem impedir que alguém lhes envie Ether. Os contratos podem reagir e rejeitar uma transferência regular, mas existem formas para mover Ether sem criar uma chamada de mensagem. Uma maneira é simplesmente «mine to» (minerar para) o endereço do contrato e a segunda maneira é usar `selfdestruct(x)`.
- Se um contrato receber Ether (sem uma função chamada), a função de retorno (fallback function) é executada. Se não tiver uma função de retorno, o Ether será rejeitado (jogando uma exceção). Durante a execução da função de retorno, o contrato só pode confiar sobre o «gas stipend» (gasto de gas) (2300 gas) que estava disponível naquele momento. Este gasto não é suficiente para acessar o armazenamento de qualquer maneira. Para ter certeza de que seu contrato pode receber Ether dessa maneira, verifique os requisitos de gas da função de retorno (por exemplo, na seção «details» em Remix).
- Existe uma maneira de encaminhar mais gas para o contrato de recebimento usando a função `addr.call.value(x)()`. Isto é essencialmente o mesmo que a função `addr.transfer(x)`, só que encaminhando o gas restante e abre a capacidade de destinatário para executar ações mais caras (e apenas retorna um código de falha e não propaga automaticamente o erro). Isso pode incluir chamar de volta no contrato de envio ou outras mudanças de estado em que você talvez não tenha pensado. Assim, permite uma grande flexibilidade para usuários honestos, mas também para atores maliciosos.
- Se você quiser enviar Ether usando `address.transfer`, existem certos detalhes para se dar conta:

1. Se o destinatário for um contrato, ele faz com que sua função de retorno seja executada, o que pode, por sua vez, chamar de volta o contrato de envio.
2. O envio de Ether pode falhar devido à profundidade da chamada acima de 1024. Como o chamador (caller) está no controle total da profundidade da chamada, ele (caller) pode forçar a transferência para provocar a falhar; Tire essa possibilidade em consideração ou use `send` e certifique-se sempre de verificar seu valor de retorno. Melhor ainda, escreva seu contrato usando um padrão em que o destinatário pode retirar Ether em vez disso.
3. O envio de Ether pode falhar porque a execução do contrato do destinatário requer mais do que a quantidade atribuída de gás (`require`, `assert`, `revert`, `throw`` ou porque a operação é muito cara) - "runs out of gas" (OOG). Se você usar ``transfer` ou `send` com uma verificação de valor de retorno, isso pode fornecer um meio do destinatário bloquear o progresso no contrato de envio. Mais uma vez, a melhor prática aqui é usar *«withdraw» pattern instead of a «send» pattern*.

Callstack Depth

Chamadas de funções externas podem falhar a qualquer tempo porque excedem a profundidade de chamadas de 1024 pilhas. Em tais situações, o Solidity gera uma exceção. Atores maliciosos podem forçar uma chamada de pilha para este alto valor antes de interagir com seu contrato.

Perceba que `.send()` não não **not** lança uma exceção se a pilha de chamadas for sendo empobrecida, mas sim retorna `false` nesse caso. As funções de baixo nível `.call()`, `.callcode()` e `.delegatecall()` comportam-se da mesma maneira.

tx.origin

Nunca use tx.origin para autorização. Vamos supor que você tenha um contrato de wallet conforme segue:

```
pragma solidity ^0.4.11;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;

    function TxUserWallet() {
        owner = msg.sender;
    }

    function transferTo(address dest, uint amount) {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Agora, alguém o engana para enviar Ether para o endereço desta carteira de ataque:

```
pragma solidity ^0.4.11;

interface TxUserWallet {
    function transferTo(address dest, uint amount);
}

contract TxAttackWallet {
    address owner;
```

```
    function TxAttackWallet() {
        owner = msg.sender;
    }

    function() {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Se a sua carteira tivesse verificado `msg.sender` para obter autorização, obteria o endereço da carteira de ataque, em vez do endereço do proprietário. Mas, ao verificar *tx.origin*', obtém o endereço original que iniciou a transação, que ainda é o endereço do proprietário. A carteira de ataque drena instantaneamente todos os seus fundos.

Detalhes Menores

- Em `for (var i = 0; i < arrayName.length; i++) { ... }`, o tipo de `i` será `uint8`, porque este é o menor tipo que é necessário para manter o valor `0`. Se a matriz tiver mais de 255 elementos, o loop não será encerrado.
- A palavra-chave `constant` para funções atualmente não é aplicada pelo compilador. Além disso, não é aplicada pelo EVM, então uma função de contrato que «reivindica» para ser uma constante pode ainda causar alterações no estado.
- Tipos que não ocupam os 32 bytes completos podem conter «bits sujos de ordem superior» (dirty higher order bits). Isto é especialmente importante se você acessar `msg.data` - isso representa um risco de maleabilidade: Você pode criar transações que chamam a função `f(uint8 x)` com um argumento de byte bruto de `0xff000001` e com `0x00000001`. Ambos são alimentados ao contrato e ambos irão parecer que o número `1` até `x` está em causa, mas `msg.data` irá parecer diferente, então, se você usar `keccak256(msg.data)` para qualquer coisa, você obterá resultados diferentes.

6.5.2 Recomendações

Restringir o total de Ether

Restringir o total de Ether (ou outros tokens) que pode ser armazenado em um smart contract. Se o seu código fonte, o compilador ou a plataforma tiver um erro (bug), estes fundos poderão ser perdidos. Se você quiser limitar suas perdas, limite o total de Ether.

Mantenha Pequeno e Modular

Mantenha seu contrato pequeno e facilmente compreensível. Demonstre funcionalidades não relacionadas em outros contratos ou bibliotecas. Recomendações gerais sobre a qualidade do código fonte naturalmente se aplicam: Limite o total de variáveis locais, o comprimento das funções e assim por diante; Documente suas funções para que outros podem ver quais eram suas intenções e se é diferente do que o código executa.

Use o padrão Checks-Effects-Interactions

A maioria das funções irá executar primeiro algumas verificações (quem chamou a função, quais são os argumentos, se foi enviado Ether suficiente, se a pessoa tem tokens, etc.). Essas verificações devem ser feitas primeiro.

Como o segundo passo, se todas as verificações passaram, os efeitos nas variáveis de estado do contrato atual devem ser feitos. Interação com outros contratos devem ser o último passo em qualquer função.

Contratos iniciais retardam alguns efeitos e aguardaram a função externa chamada para retornar em um estado sem erro. Este é muitas vezes um grave erro devido ao problema de re-entrada (re-entrancy) explicado acima.

Observe também que as chamadas para contratos conhecidos podem, por sua vez, causar chamadas para contratos desconhecidos, por isso provavelmente é melhor simplesmente aplicar esse padrão.

Incluir um Modo de Falha Segura (Fail-Fase Mode)

Ao tornar seu sistema totalmente descentralizado, você removerá qualquer intermediário, Pode ser uma boa ideia, especialmente para o novo código, incluir algum tipo de mecanismo de segurança:

Você pode adicionar uma função no seu contrato inteligente que executa algumas auto-verificações como «Algum Ether vazou?» (Has any Ether leaked?), «A soma dos tokens é igual ao saldo do contrato?» («Is the sum of the tokens equal to the balance of the contract?») e situações similares.

Tenha em mente que você não pode usar muito gas para isso, por isso pensar fora da cadeia (off-chain) pode ser necessário aqui.

Se a auto-verificação falhar, o contrato muda automaticamente para algum tipo do modo «failsafe», que, por exemplo, desabilita a maioria dos recursos, entrega o controle para uma terceira parte fixa e confiável ou simplesmente converte o contrato em um simples contrato «devolva meu dinheiro» (give me back my money).

6.5.3 Verificação Formal

Usando a verificação formal, é possível realizar uma prova matemática automatizada de que o seu código fonte cumpre uma determinada especificação formal. A especificação ainda é formal (assim como o código fonte), mas geralmente muito mais simples.

Observe que a verificação formal em si só pode ajudá-lo a entender a diferença entre o que você fez (a especificação) e como você fez isso (a implementação real). Você ainda precisa verificar se a especificação é o que você queria e que não perdeu os efeitos não intencionais.

6.6 Usando o compilador

6.6.1 Usando o Comando em linha do Compilador

Um dos objetivos de construção do repositório Solidity é `solc`, o compilador de linha de comando do Solidity. Usar `solc --help` fornecerá uma explicação de todas as opções. O compilador pode produzir várias saídas, que vão desde binários simples e montagem em uma árvore de sintaxe abstrata (árvore de análise) até estimativas do uso de gás.

Se você quiser compilar apenas um único arquivo, você pode executá-lo como `solc --bin sourceFile.sol` e ele irá imprimir o binário. Antes de implantar seu contrato, ative o otimizador ao compilar usando `solc --optimize --bin sourceFile.sol`. Se você quiser obter algumas variantes mais avançadas de saída do `solc`, provavelmente é melhor contar para que ele saia tudo para separar arquivos usando `solc -o outputDirectory --bin --ast --asm sourceFile.sol`.

O compilador de linha de comando lê automaticamente arquivos importados do sistema de arquivos, mas também é possível fornecer redirecionamentos de caminho usando `prefix=path` da seguinte maneira:

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

Isso essencialmente instrui o compilador a procurar qualquer coisa começando com `github.com/ethereum/dapp-bin/` em `/usr/local/lib/dapp-bin` e se não encontrar o arquivo lá, ele verá `/usr/local/lib/fallback` (o prefixo vazio sempre irá corresponder). `solc` não lerá arquivos do sistema de arquivos que ficam fora de os destinos de remapeamento e fora dos diretórios onde os arquivos fontes explicitamente residem, então coisas como `import "/etc/passwd";` só funcionam se você adicionar `=/` como um remapeamento.

Se houver várias correspondências devido a remapeamentos, é selecionado aquele com o prefixo comum mais longo.

Por motivos de segurança, o compilador tem restrições sobre quais diretórios ele pode acessar. Os caminhos (e seus subdiretórios) dos arquivos de origem especificados na linha de comando e os caminhos definidos pelos remakes são permitidos para instruções de importação, mas tudo o resto é rejeitado. Caminhos adicionais (e seus subdiretórios) podem ser permitidos através do parâmetro `--allow-paths /sample/path,/another/sample/path`.

Se seus contratos usam *libraries*, você notará que o bytecode contém substrings do formulário `__LibraryName_____`. Você pode usar `solc` como um vinculador, o que significa que ele irá inserir os endereços da biblioteca para você nesses pontos:

Ou adicione `--libraries "Math:0x1234567890123456789 Heap:0xabcdef0123456"` para o seu comando fornecer um endereço para cada biblioteca ou armazenar a string em um arquivo (uma biblioteca por linha) e executar `solc` usando `--libraries fileName`.

Se `solc` é chamado com a opção `--link`, todos os arquivos de entrada são interpretados como binários não vinculados (codificados em hexadecimal) no `__LibraryName_____`-formato acima e estão ligados no local (se a entrada é lida a partir de stdin, está escrito para stdout). Todas as opções, exceto `--libraries`, são ignoradas (incluindo `-o`) neste caso.

Se `solc` é chamado com a opção `--standard-json`, espera uma entrada JSON (conforme explicado abaixo) na entrada padrão e retorna uma saída JSON na saída padrão.

6.6.2 Entrada e saída do compilador Descrição JSON

Esses formatos JSON são usados pela API do compilador, bem como estão disponíveis através de `solc`. Estes estão sujeitos a alterações, alguns campos são opcionais (como observado), mas é destinado a apenas fazer alterações compatíveis com versões anteriores.

A API do compilador espera uma entrada formatada JSON e produz o resultado da compilação em uma saída formatada JSON.

Os comentários não são, naturalmente, permitidos e são utilizados aqui apenas para fins explicativos.

Descrição da Entrada

```
{
  // Required: Source code language, such as "Solidity", "serpent", "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.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "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 (enabled defaults to false)
    optimizer: {
      enabled: true,
      runs: 500
    },
    // 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,
    // 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.
    //
    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.opcodes" ]
      },
      // Enable the source map output of every single contract.
      "*": {
        "*": [ "evm.sourceMap" ]
      },
      // Enable the legacy AST output of every single file.
      "*": {
        "": [ "legacyAST" ]
      }
    }
  }
}
```

Descrição da Saída

```
{
  // Optional: not present if no errors/warnings were encountered
  errors: [
    {
      // Optional: Location within the source file.
      sourceLocation: {
        file: "sourceFile.sol",
        start: 0,
        end: 100
      },
      // Mandatory: Error type, such as "TypeError", "InternalCompilerError",
↪"Exception", etc
      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.
          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: ""
        }
      }
    }
  }
}
```

6.7 Especificação da interface binária do aplicativo

6.7.1 Design Básico

A interface binária do aplicativo é a maneira padrão de interagir com os contratos no ecossistema Ethereum, ambos de fora do blockchain e para a interação contrato-contrato. Os dados são codificados de acordo com seu tipo, conforme descrito nesta especificação. A codificação não é auto-descritiva e, portanto, requer um esquema para decodificar.

Assumimos que as funções de interface de um contrato são fortemente digitadas, conhecidas no momento da compilação e estáticas. Nenhum mecanismo de introspecção será fornecido. Assumimos que todos os contratos terão as definições de interface de quaisquer contratos que eles chamem disponíveis em tempo de compilação.

Esta especificação não aborda contratos cuja interface é dinâmica ou conhecida apenas em tempo de execução. Se esses casos se tornarem importantes, eles podem ser tratados adequadamente como instalações construídas no ecossistema Ethereum.

6.7.2 Função Selector

Os primeiros quatro bytes dos dados de chamada para uma chamada de função especificam a função a ser chamada. É o primeiro (esquerda, alta-ordem em big-endian) quatro bytes do Keccak (SHA-3) hash da assinatura da função. A assinatura é definida como a expressão canônica do protótipo básico, em geral, o nome da função com a lista entre parênteses dos tipos de parâmetros. Os tipos de parâmetros são divididos por uma única vírgula - não são utilizados espaços.

6.7.3 Codificação de Argumento

A partir do quinto byte, seguem os argumentos codificados. Esta codificação também é usada em outros lugares, p.ex. os valores de retorno e também os argumentos de eventos são codificados da mesma maneira, sem os quatro bytes especificando a função.

6.7.4 Tipos

existem os seguintes tipos elementares:

- `uint<M>`: tipo integer não assinado de `M` bits, `0 < M <= 256`, `M % 8 == 0`. normalmente `uint32`, `uint8`, `uint256`.
- `int<M>`: o complemento de dois complementos de tipo inteiro de `M` bits, `0 < M <= 256`, `M % 8 == 0`.
- `address`: equivalente ao `uint160`, exceto pela interpretação assumida e digitação de idioma.
- `uint`, `int`: sinônimos para `uint256`, `int256` respectivamente (não deve ser usado para computar o seletor de funções).
- `bool`: equivalente para `uint8` restrito para o valor 0 e 1
- `fixed<M>x<N>`: Número decimal de ponto fixo assinado de `M` bits, `8 <= M <= 256`, `M % 8 ==0`, e `0 < N <= 80`, que denota o valor `v` como `v / (10 ** N)`.
- `ufixed<M>x<N>`: variante não assinada de `fixed<M>x<N>`.
- `fixed`, `ufixed`: sinônimos para `fixed128x19`, `ufixed128x19` respectivamente (não deve ser usado para computar o seletor de funções).
- `bytes<M>`: tipo binário de `M` bytes, `0 < M <= 32`.
- `function`: equivalente para `bytes24`: um endereço, seguido por um seletor de função

Existe o seguinte tipo de matriz (tamanho fixo):

- `<type>[M]`: uma matriz de comprimento fixo do tipo de comprimento fixo fornecido.

Existem os seguintes tipos de tamanho não fixo:

- `bytes`: Seqüência de bytes de tamanho dinâmico.
- `string`: Cadeia unicode de tamanho dinâmico assumida como codificada em UTF-8.
- `<type>[]`: uma matriz de comprimento variável do tipo de comprimento fixo fornecido.

Os tipos podem ser combinados para estruturas anônimas, encerrando um número finito não negativo deles entre parênteses, separados por vírgulas:

- `(T1,T2,...,Tn)`: estrutura anônima (tupla ordenada) constituída pelos tipos `T1`, ..., `Tn`, `n >= 0`

É possível formar estruturas de estruturas, arrays de estruturas e assim por diante.

6.7.5 Especificação Formal da Codificação

Nós iremos agora especificar formalmente a codificação, de modo que ele terá o seguinte propriedades, que são especialmente úteis se alguns argumentos forem arrays aninhados:

Propriedades:

1. O número de leituras necessárias para acessar um valor é, no máximo, a profundidade do valor dentro da estrutura da matriz do argumento, ou seja, são necessárias quatro leituras para recuperar `a_i[k][l][r]`.. Em uma versão anterior do ABI, o número de leituras escalonadas linearmente com o número total de parâmetros dinâmicos no pior dos casos.
2. Os dados de uma variável ou elemento de matriz não são entrelaçados com outros dados e são relocáveis, isto é, ele usa apenas «endereços» relativos

Nós distinguimos tipos estáticos e dinâmicos. Os tipos estáticos são codificados no local e os tipos dinâmicos são codificados em um local alocado separadamente após o bloco corrente.

Definição: Os seguintes tipos são chamados «dinâmicos» (dynamic): * `bytes` * `string` * `T[]` para qualquer `T` * `T[k]` para qualquer dinâmico `T` e qualquer `k > 0` * `(T1,...,Tk)` se qualquer `Ti` é dinâmico para `1 <= i <= k`

Todos os outros tipos são chamados «estáticos» (static)

Definição: `len(a)` é o n´mero de bytes em uma string binária‘‘a‘‘. O tipo de `len(a)` é assumido para ser `uint256`.

Definimos `enc`, a codificação real, como um mapeamento de valores dos tipos ABI para cadeias binárias, como que `len(enc(X))` depende do valor de `X` se e somente se o tipo de `X` for dinâmico.

Definição: Para qualquer valor ABI `X`, nós recursivamente definimos `enc(X)`, dependendo do tipo de `X` sendo

- `(T1,...,Tk)` para `k >= 0` e quaisquer tipos `T1`, ..., `Tk`

  `enc(X) = head(X(1)) ... head(X(k-1)) tail(X(0)) ... tail(X(k-1))`

  onde `X(i)` é o componente `ith` do valor, e `head` e `tail` são definidos para `Ti` sendo um tipo estático como

  `head(X(i)) = enc(X(i))` e `tail(X(i)) = ""` (a string vazia)

  e como

  `head(X(i)) = enc(len(head(X(0)) ... head(X(k-1)) tail(X(0)) ... tail(X(i-1)))) tail(X(i)) = enc(X(i))`

  caso contrário, em geral se `Ti` é um tipo dinâmico.

  Observe que no caso dinâmico, `head(X(i))` está bem definido, pois os comprimentos das partes principais apenas dependem dos tipos e não dos valores. Seu valor é o deslocamento do começo do `tail(X(i))` relativo ao começo de `enc(X)`.

- `T[k]` para qualquer `T` e `k`:

  `enc(X) = enc((X[0], ..., X[k-1]))`

  isto é codificado como se fosse uma estrutura anônima com elementos `k` do mesmo tipo.

- `T[]` onde `X` tem `k` elementos (`k` é assumido para ser do tipo `uint256`):

  `enc(X) = enc(k) enc([X[1], ..., X[k]])`

  isto é codificado como se fosse uma matriz de tamanho estático `k`, prefixado com o número de elementos.

- `bytes`, de tamanho `k` (que é assumido ser do tipo `uint256`):

`enc(X) = enc(k) pad_right(X)`, isto é, o número de bytes é codificado como um `uint256` seguido pelo valor real de `X` como uma seqüência de bytes, seguida por o número mínimo de zero-bytes tal que `len(enc(X))` é um múltiplo de 32.

- `string`:

`enc(X) = enc(enc_utf8(X))`, ou seja, `X` é codificado em utf-8 e esse valor é interpretado como de tipo `bytes` e codificado ainda mais. Observe que o comprimento usado nesta codificação subseqüente é o número de bytes da seqüência codificada utf-8, não o número de caracteres.

- `uint<M>`: `enc(X)` é a codificação big-endian de `X`, preenchida no lado de ordem superior (esquerda) com zero-bytes, de modo que o comprimento é um múltiplo de 32 bytes.
- `address`: como no caso `uint160`
- `int<M>`: `enc(X)` é o complemento de código de dois big-endian `X`, preenchido na mais alta ordem (esquerda) com `0xff` para `X` negativo e com zero bytes para ‘‘ X “positivo, de modo que o comprimento seja um múltiplo de 32 bytes.
- `bool`: como no caso `uint8`, onde `1` é usado para `true` e `0` para `false`
- `fixed<M>x<N>`: `enc(X)` é `enc(X * 10**N)` onde `X * 10**N` é interprestado como `int256`.
- `fixed`: como no caso `fixed128x19`
- `ufixed<M>x<N>`: `enc(X)` é `enc(X * 10**N)` onde `X * 10**N` é interpretado como `uint256`.
- `ufixed`: como no caso `ufixed128x19`
- `bytes<M>`: `enc(X)` é a sequência de bytes em `X` preenchido com zero-bytes para um comprimento de 32.

perceba que para qualquer *X*‘, `len(enc(X))` é um múltiplo de 32.

6.7.6 Função Seletor e Codificação de Argumento

Em suma, uma chamada para a função `f` com os parâmetros `a_1, ..., a_n` está codificada como

```
function_selector(f) enc((a_1, ..., a_n))
```

e os valores de retorno `v_1, ..., v_k` de `f` são codificados como

```
enc((v_1, ..., v_k))
```

ou seja, os valores são combinados em uma estrutura anônima e codificados.

6.7.7 Exemplos

Dado o contrato:

```
pragma solidity ^0.4.0;

contract Foo {
  function bar(bytes3[2] xy) {}
  function baz(uint32 x, bool y) returns (bool r) { r = x > 32 || y; }
  function sam(bytes name, bool z, uint[] data) {}
}
```

Assim, para o nosso exemplo `Foo` se queríamos chamar `baz` com os parâmetros `69` e `true`, passaríamos 68 bytes no total, que podem ser divididos em:

- `0xcdcd77c0`: o Method ID. Isto é derivado como os primeiros 4 bytes do hash Keccak do formato ASCII da assinatura `baz(uint32,bool)`.

- `0x0000000000000000000000000000000000000000000000000000000000000045`: o primeiro parâmetro, um uint32 de valor `69` preenchido para 32 bytes
- `0x0000000000000000000000000000000000000000000000000000000000000001`: o segundo parâmetro - booleano `true`, preenchido para 32 bytes

No total:

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000
```

Isso irá retornar um único `bool`. Se, por exemplo, devesse retornar `false`, sua saída seria a matriz de byte `0x0000000000000000000000000000000000000000000000000000000000000000`, um único bool.

Se quiséssemos chamar `bar` com o argumento `["abc", "def"]`, passaríamos 68 bytes totais, divididos em:

- `0xfce353f6`: o Method ID. Isto é derivado da assinatura `bar(bytes3[2])`.
- `0x6162630000000000000000000000000000000000000000000000000000000000`: a primeira parte do primeiro parâmetro, o valor `bytes3` para `"abc"` (alinhado à esquerda).
- `0x6465660000000000000000000000000000000000000000000000000000000000`: a segunda parte do primeiro parâmetro, o valor `bytes3` para `"def"` (alinhado á esquerda).

No total:

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000006465660000000000000000000000
```

Se quisermos chamar `sam` com os argumentos `"dave"`, `true` e `[1,2,3]`, passaríamos 292 bytes totais, divididos em:

- `0xa5643bf2`: o Method ID. É derivado da assinatura‘‘sam(bytes,bool,uint256[])‘‘. Perceba que `uint` é substituída por sua representação canônica‘‘uint256‘‘.
- `0x0000000000000000000000000000000000000000000000000000000000000060`: a localização da parte de dados do primeiro parâmetro (tipo dinâmico), medida em bytes a partir do início do bloco de argumentos. Nesse caso, `0x60`.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: o segundo parâmetro = booleano de true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0`: a localização da parte de dados do terceiro parâmetro (tipo dinâmico), medida em bytes. Nesse caso, `0xa0`.
- `0x0000000000000000000000000000000000000000000000000000000000000004`: a parte de dados do primeiro argumento, ele começa com o comprimento da matriz de bytes em elementos, neste caso, 4.
- `0x6461766500000000000000000000000000000000000000000000000000000000`: o conteúdo do primeiro argumento: a codificação UTF-8 (igual a ASCII neste caso) de `"dave"`, preenchida à direita para 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: a parte de dados do terceiro argumento, ele começa com o comprimento da matriz em elementos, neste caso, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: A primeira entrada do terceiro parâmetro.
- `0x0000000000000000000000000000000000000000000000000000000000000002`: A segunda entrada do terceiro parâmetro.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: A terceira entrada do terceiro parâmetro.

No total:

```
0xa5643bf20000000000000000000000000000000000000000000000000000000000000060000000000000000000000000000000
```

6.7.8 Uso dos Tipos Dinâmicos

Uma chamada para uma função com a assinatura `f(uint,uint32[],bytes10,bytes)` com valores `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` está codificado da seguinte maneira:

Nós pegamos os primeiros quatro bytes de `sha3("f(uint256,uint32[],bytes10,bytes)")`, isto é `0x8be65246`

Então codificamos as partes principais de todos os quatro argumentos. Para os tipos estáticos `uint256` e `bytes10`, estes são diretamente os valores que queremos passar, enquanto que para os tipos dinâmicos `uint32 [] `` e ``bytes`, usamos o deslocamento em bytes para o início de sua área de dados, medida a partir do início da codificação do valor (ou seja, não contando os quatro primeiros bytes contendo o hash da assinatura da função). Esses são:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` preenchido para 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset para o início da parte de dados do segundo parâmetro, 4*32 bytes, exatamente o tamanho da parte principal)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` preenchido para 32 bytes à direita)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset para o início da parte de dados do quarto parâmetro = offset para o início da parte de dados do primeiro parâmetro dinâmico + o tamanho da parte de dados do primeiro parâmetro dinâmico = 4*32 + 3*32 (veja abaixo))

Após isto, a parte de dados do primeiro argumento dinâmico `[0x456, 0x789]` conforme segue:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (número de elementos do array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (primeiro elemento)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (segundo elemento)

Finalmente, nós codificamos a parte de dados do segundo argumento dinâmico `"Hello, world!"`:

- `0x000000000000000000000000000000000000000000000000000000000000000d` (número de elementos (bytes neste caso): 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` (`"Hello, world!"` preenchido para 32 bytes à direita)

Tudo junto, a codificação é (nova linha após o seletor de função e cada 32 bytes para maior clareza):

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

6.7.9 Eventos

Os eventos são uma abstração do protocolo Ethereum para logging/event-watching. As entradas do registro fornecem o endereço do contrato, uma série de até quatro tópicos e alguns dados binários de comprimento arbitrário. Os eventos aproveitam a função ABI existente para interpretar isso (juntamente com uma especificação de interface) como uma estrutura corretamente digitada.

Dado um nome de evento e uma série de parâmetros de eventos, os dividimos em duas subséries: as que estão indexadas e as que não são. Aqueles que são indexados, que podem numerar até 3, são usados ao lado do hash Keccak da assinatura do evento para formar os tópicos da entrada de log. Aqueles que, como não indexados, formam a matriz de bytes do evento.

De fato, uma entrada de log usando este ABI é descrito como:

- `address`: o endereço do contrato (intrinsicamente fornecido pelo Ethereum);
- `topics[0]: keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` é uma função que simplesmente retorna o tipo canônico de um determinado argumento, em geral para `uint indexed foo`, ele retornaria `uint256`). Se o evento for declarado como «anônimo», o `topics [0]` não é gerado;
- `topics[n]`: `EVENT_INDEXED_ARGS[n - 1]` (`EVENT_INDEXED_ARGS` é uma série de `EVENT_ARGS` que são indexados);
- `data`: `abi_serialise(EVENT_NON_INDEXED_ARGS)` (`EVENT_NON_INDEXED_ARGS` é uma série de `EVENT_ARGS` que não são indexados, `abi_serialise` é a função de serialização ABI usado para retornar uma série de valores digitados de uma função conforme descrito acima.

6.7.10 JSON

O formato JSON para a interface de um contrato é fornecido por uma série de funções e/ou descrições de eventos. Uma descrição de função é um objeto JSON com os campos:

- `type`: `"function"`, `"constructor"`, ou `"fallback"` (a *unnamed «default» function*);
- `name`: o nome da função;
- `inputs`: uma tabela de objetos, cada um contendo: * `name`: o nome do parâmetro; * `type`: o tipo canônico do parâmetro.
- `outputs`: uma tabela de objetos similar aos `inputs`, podendo ser omitido se a função não retornar algo;
- `payable`: `true` se a função aceitar ether, por default para `false`;
- `stateMutability`: uma string com um dos seguintes valores: `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` e `payable` (o mesmo que `payable` acima).
- `constant`: `true` se a função é ambos `pure` ou `view`

`type` pode ser omitido, defaulting to `"function"`.

As funções Constructor e fallback nunca tem `name` ou `outputs`. A função de retorno também não possui `inputs`.

O envio de Ether não-zero para função não pagável será executada. Não faça isso.

Uma descrição do evento é um objeto JSON com campos bastante semelhantes:

- `type`: sempre `"event"`
- `name`: o nome do evento;

- `inputs`: uma tabela de objetos, cada um contendo: * `name`: o nome do parâmetro; * `type`: o tipo canônico do parâmetro. * `indexed`: `true` se o campo for parte dos tópicos do log, `false` se um dos segmentos de dados do log.
- `anonymous`: `true` se o evento foi declarado como `anonymous`.

Por examplo,

```
pragma solidity ^0.4.0;

contract Test {
  function Test(){ b = 0x1234567890123456789012345678901234567890123456789012; }
  event Event(uint indexed a, bytes32 b)
  event Event2(uint indexed a, bytes32 b)
  function foo(uint a) { Event(a, b); }
  bytes32 b;
}
```

pode resultar no 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": []
}]
```

6.8 Style Guide

6.8.1 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 judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

6.8.2 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:

```
contract A {
    ...
}


contract B {
    ...
}


contract C {
    ...
}
```

No:

```
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:

```
contract A {
    function spam();
    function ham();
}


contract B is A {
    function spam() {
        ...
    }

    function ham() {
        ...
    }
}
```

No:

```
contract A {
    function spam() {
        ...
    }
    function ham() {
        ...
    }
}
```

Source File Encoding

UTF-8 or ASCII encoding is preferred.

Imports

Import statements should always be placed at the top of the file.

Yes:

```
import "owned";


contract A {
    ...
}


contract B is owned {
    ...
}
```

No:

```
contract A {
    ...
}
```

```
import "owned";


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 `constant` functions last.

Yes:

```
contract A {
    function A() {
        ...
    }

    function() {
        ...
    }

    // External functions
    // ...

    // External functions that are constant
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

No:

```
contract A {

    // External functions
    // ...
```

```
    // Private functions
    // ...

    // Public functions
    // ...

    function A() {
        ...
    }

    function() {
        ...
    }

    // 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() { spam(); }
```

Immediately before a comma, semicolon:

Yes:

```
function spam(uint i, Coin coin);
```

No:

```
function spam(uint i , Coin coin) ;
```

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() {
    ...
}
```

No:

```
function () {
    ...
}
```

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:

```
contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
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 preceeded by a single space.

Yes:

```
function increment(uint x) returns (uint) {
    return x + 1;
}

function increment(uint x) public onlyowner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) returns (uint)
{
    return x + 1;
}

function increment(uint x) returns (uint){
    return x + 1;
}

function increment(uint x) returns (uint) {
    return x + 1;
    }

function increment(uint x) returns (uint) {
    return x + 1;}
```

The visibility modifiers 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
) {
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to it's 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();
}
```

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:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
    }
}
```

No:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4)
    {
        // do something with param5
    }
}

contract A is B, C, D {
```

```
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
        // do something with param5
    }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible:

```
function shortFunction() { 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;
```

6.8.3 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 supercede 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`

Nota: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

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.

Events

Events should be named using the CapWords style.

Function Names

Functions should use mixedCase.

Function Arguments

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 Variables

Use mixedCase.

Constants

Constants should be named with all capital letters with underscores separating words. (for example:`MAX_BLOCKS`)

Modifiers

Use mixedCase.

Avoiding 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

6.9 Common Patterns

6.9.1 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 `send` call, this is not recommended as it introduces a potential security risk. You may read more about this on the *Considerações de Segurança* page.

This 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 ^0.4.11;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    function WithdrawalContract() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() payable returns (bool) {
        if (msg.value > mostSent) {
            pendingWithdrawals[richest] += msg.value;
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }

    function withdraw() {
        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);
    }
}
```

This is as opposed to the more intuitive sending pattern:

```
pragma solidity ^0.4.11;

contract SendContract {
    address public richest;
    uint public mostSent;

    function SendContract() payable {
        richest = msg.sender;
        mostSent = msg.value;
```

```
    }

    function becomeRichest() 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 conssuming more than the 2300 gas stipend). 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.

6.9.2 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 page is about.

The use of **function modifiers** makes these restrictions highly readable.

```
pragma solidity ^0.4.11;

contract AccessRestriction {
    // These will be assigned at the construction
    // phase, where `msg.sender` is the account
    // 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);
        // 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)
        onlyBy(owner)
    {
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        require(now >= _time);
        _;
    }

    /// Erase ownership information.
    /// May only be called 6 weeks after
    /// the contract has been created.
    function disown()
        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) {
        require(msg.value >= _amount);
        _;
        if (msg.value > _amount)
            msg.sender.send(msg.value - _amount);
    }

    function forceOwnerChange(address _newOwner)
        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.

6.9.3 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.

Nota: 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.

Nota: 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 ^0.4.11;

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 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()
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // We will not implement that here
    }

    function reveal()
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // This modifier goes to the next stage
    // after the function is done.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

6.10 Lista de Bugs Conhecidos

Abaixo, você consegue encontrar em um formato JSON uma lista dos bugs relevantes para segurança, encontrados no compilador do Solidity. O arquivo em si, é hospedado em um repositório no GitHub. A lista se estende até a versão 0.3.0, bugs conhecidos que estão presentes somente nas versões precedentes a esta não estão listados.

Existe outro arquivo chamado bugs_by_version.json, que pode ser utilizado para checar quais bugs afetam uma versão específica do compilador.

Ferramentas de verificação de fontes de contratos e outras ferramentas que interajam com contratos devem consultar esta lista de acordo com os seguintes critérios:

- É meio suspeito que um contrato foi compilado com uma versão nightly do compilador ao invés de uma versão de release. Esta lista não cobre versões unreleased ou nightly.
- Também é meio suspeito que um contrato foi compilado com uma versão que não seja a mais recente no momento da criação. Para contratos criados a partir de outros contratos, você deve seguir a corrente de criação até a transação e usar a data desta transação como data de criação.
- É altamente suspeito que um contrato foi compilado com um compilador que contém um bug conhecido e o contrato foi criado quando uma nova versão de compilador contendo a correção foi lançada.

O arquivo JSON de bugs conhecidos abaixo, é um array de objetos, um para cada bug, com as seguintes chaves:

name Nome único dado a um bug

summary Descrição curta do bug

description Descrição detalhada do bug

link URL de um website com informações mais detalhadas, opcional

introduced A primeira versão publicada do compilador que possuía o bug, optional

fixed A primeira versão publicada do compilador que já não possuía mais o bug

publish A data em que o bug se tornou conhecido publicamente, opcional

severity Gravidade do bug, low (baixa), medium (média), high (alta). Leva em consideração a facilidade de ser encontrado em testes de contrato, probabilidade de ocorrência e danos potenciais por exploits.

conditions Condições que precisam ser atendidas para disparar o bug. Atualmente, esse objeto pode conter um valor boolean `optimizer`, que significa que o optimizer (otimizador) precisa ser ligado para ativar o bug. Se nenhuma condição for dada, suponha que o erro está presente.

```
[
    {
        "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.",
        "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.",
        "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",
        "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"
    }
]
```

6.11 Contributing

Help is always appreciated!

To get started, you can try *Construindo à partir do Fonte* 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.

6.11.1 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.

6.11.2 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.

Additionally, if you are writing a new feature, please ensure you write appropriate Boost test cases and place them under `test/`.

However, if you are making a larger change, please consult with the Gitter channel, first.

Finally, please make sure you respect the coding standards 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!

6.11.3 Running the compiler tests

Solidity includes different types of tests. They are included in the application called `soltest`. Some of them require the `cpp-ethereum` client in testing mode.

To run a subset of the tests that do not require `cpp-ethereum`, use `./build/test/soltest -- --no-ipc`.

For all other tests, you need to install cpp-ethereum and run it in testing mode: `eth --test -d /tmp/testeth`.

Then you run the actual tests: `./build/test/soltest -- --ipcpath /tmp/testeth/geth.ipc`.

To run a subset of tests, filters can be used: `soltest -t TestSuite/TestName -- --ipcpath /tmp/testeth/geth.ipc`, where `TestName` can be a wildcard `*`.

Alternatively, there is a testing script at `scripts/test.sh` which executes all tests and runs `cpp-ethereum` automatically if it is in the path (but does not download it).

Travis CI even runs some additional tests (including `solc-js` and testing third party Solidity frameworks) that require compiling the Emscripten target.

6.11.4 Whiskers

Whiskers is a 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 *Inline Assembly* (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.

6.12 Frequently Asked Questions

This list was originally compiled by fivedogit.

6.12.1 Basic Questions

Example contracts

There are some contract examples by fivedogit and there should be a test contract for every single feature of Solidity.

Create and publish the most basic contract possible

A quite simple contract is the greeter

Is it possible to do something on a specific block number? (e.g. publish a contract or execute a transaction)

Transactions are not guaranteed to happen on the next block or any future specific block, since it is up to the miners to include transactions and not up to the submitter of the transaction. This applies to function calls/transactions and contract creation transactions.

If you want to schedule future calls of your contract, you can use the alarm clock.

What is the transaction «payload»?

This is just the bytecode «data» sent along with the request.

Is there a decompiler available?

There is no decompiler to Solidity. This is in principle possible to some degree, but for example variable names will be lost and great effort will be necessary to make it look similar to the original source code.

Bytecode can be decompiled to opcodes, a service that is provided by several blockchain explorers.

Contracts on the blockchain should have their original source code published if they are to be used by third parties.

Create a contract that can be killed and return funds

First, a word of warning: Killing contracts sounds like a good idea, because «cleaning up» is always good, but as seen above, it does not really clean up. Furthermore, if Ether is sent to removed contracts, the Ether will be forever lost.

If you want to deactivate your contracts, it is preferable to **disable** them by changing some internal state which causes all functions to throw. This will make it impossible to use the contract and ether sent to the contract will be returned automatically.

Now to answering the question: Inside a constructor, `msg.sender` is the creator. Save it. Then `selfdestruct(creator);` to kill and return funds.

example

Note that if you `import "mortal"` at the top of your contracts and declare `contract SomeContract is mortal { ...` and compile with a compiler that already has it (which includes Remix), then `kill()` is taken care of for you. Once a contract is «mortal», then you can `contractname.kill.sendTransaction({from:eth.coinbase})`, just the same as my examples.

Store Ether in a contract

The trick is to create the contract with `{from:someaddress, value: web3.toWei(3,"ether")...}`

See endowment_retriever.sol.

Use a non-constant function (req `sendTransaction`) to increment a variable in a contract

See value_incrementer.sol.

Get a contract to return its funds to you (not using `selfdestruct(...)`).

This example demonstrates how to send funds from a contract to an address.

See endowment_retriever.

Can you return an array or a `string` from a solidity function call?

Yes. See array_receiver_and_returner.sol.

What is problematic, though, is returning any variably-sized data (e.g. a variably-sized array like `uint[]`) from a fuction **called from within Solidity**. This is a limitation of the EVM and will be solved with the next protocol update.

Returning variably-sized data as part of an external transaction or call is fine.

Is it possible to in-line initialize an array like so: `string[] myarray = ["a", "b"];`

Yes. However it should be noted that this currently only works with statically sized memory arrays. You can even create an inline memory array in the return statement. Pretty cool, huh?

Example:

```
pragma solidity ^0.4.0;

contract C {
    function f() returns (uint8[5]) {
        string[4] memory adaArr = ["This", "is", "an", "array"];
        return ([1, 2, 3, 4, 5]);
    }
}
```

Can a contract function return a `struct`?

Yes, but only in `internal` function calls.

If I return an `enum`, I only get integer values in web3.js. How to get the named values?

Enums are not supported by the ABI, they are just supported by Solidity. You have to do the mapping yourself for now, we might provide some help later.

Can state variables be initialized in-line?

Yes, this is possible for all types (even for structs). However, for arrays it should be noted that you must declare them as static memory arrays.

Examples:

```
pragma solidity ^0.4.0;

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();
}
```

How do structs work?

See struct_and_for_loop_tester.sol.

How do for loops work?

Very similar to JavaScript. There is one point to watch out for, though:

If you use `for (var i = 0; i < a.length; i ++) { a[i] = i; }`, then the type of `i` will be inferred only from `0`, whose type is `uint8`. This means that if `a` has more than `255` elements, your loop will not terminate because `i` can only hold values up to `255`.

Better use `for (uint i = 0; i < a.length...`

See struct_and_for_loop_tester.sol.

What are some examples of basic string manipulation (`substring`, `indexOf`, `charAt`, etc)?

There are some string utility functions at stringUtils.sol which will be extended in the future. In addition, Arachnid has written solidity-stringutils.

For now, if you want to modify a string (even when you only want to know its length), you should always convert it to a `bytes` first:

```
pragma solidity ^0.4.0;

contract C {
    string s;

    function append(byte c) {
        bytes(s).push(c);
    }

    function set(uint i, byte c) {
        bytes(s)[i] = c;
    }
}
```

Can I concatenate two strings?

You have to do it manually for now.

Why is the low-level function `.call()` less favorable than instantiating a contract with a variable (`ContractB b;`) and executing its functions (`b.doSomething();`)?

If you use actual functions, the compiler will tell you if the types or your arguments do not match, if the function does not exist or is not visible and it will do the packing of the arguments for you.

See ping.sol and pong.sol.

Is unused gas automatically refunded?

Yes and it is immediate, i.e. done as part of the transaction.

When returning a value of say `uint` type, is it possible to return an `undefined` or «null»-like value?

This is not possible, because all types use up the full value range.

You have the option to `throw` on error, which will also revert the whole transaction, which might be a good idea if you ran into an unexpected situation.

If you do not want to throw, you can return a pair:

```
pragma solidity ^0.4.0;

contract C {
    uint[] counters;

    function getCounter(uint index)
        returns (uint counter, bool error) {
            if (index >= counters.length)
                return (0, true);
            else
                return (counters[index], false);
    }

    function checkCounter(uint index) {
        var (counter, error) = getCounter(index);
        if (error) {
            // ...
        } else {
            // ...
        }
    }
}
```

Are comments included with deployed contracts and do they increase deployment gas?

No, everything that is not needed for execution is removed during compilation. This includes, among others, comments, variable names and type names.

What happens if you send ether along with a function call to a contract?

It gets added to the total balance of the contract, just like when you send ether when creating a contract. You can only send ether along to a function that has the `payable` modifier, otherwise an exception is thrown.

Is it possible to get a tx receipt for a transaction executed contract-to-contract?

No, a function call from one contract to another does not create its own transaction, you have to look in the overall transaction. This is also the reason why several block explorer do not show Ether sent between contracts correctly.

What is the `memory` keyword? What does it do?

The Ethereum Virtual Machine has three areas where it can store items.

The first is «storage», where all the contract state variables reside. Every contract has its own storage and it is persistent between function calls and quite expensive to use.

The second is «memory», this is used to hold temporary values. It is erased between (external) function calls and is cheaper to use.

The third one is the stack, which is used to hold small local variables. It is almost free to use, but can only hold a limited amount of values.

For almost all types, you cannot specify where they should be stored, because they are copied everytime they are used.

The types where the so-called storage location is important are structs and arrays. If you e.g. pass such variables in function calls, their data is not copied if it can stay in memory or stay in storage. This means that you can modify their content in the called function and these modifications will still be visible in the caller.

There are defaults for the storage location depending on which type of variable it concerns:

- state variables are always in storage
- function arguments are always in memory
- local variables always reference storage

Example:

```
pragma solidity ^0.4.0;

contract C {
    uint[] data1;
    uint[] data2;

    function appendOne() {
        append(data1);
    }

    function appendTwo() {
        append(data2);
    }

    function append(uint[] storage d) internal {
        d.push(1);
    }
}
```

The function `append` can work both on `data1` and `data2` and its modifications will be stored permanently. If you remove the `storage` keyword, the default is to use `memory` for function arguments. This has the effect that at the point where `append(data1)` or `append(data2)` is called, an independent copy of the state variable is created in memory and `append` operates on this copy (which does not support `.push` - but that is another issue). The modifications to this independent copy do not carry back to `data1` or `data2`.

A common mistake is to declare a local variable and assume that it will be created in memory, although it will be created in storage:

```
/// THIS CONTRACT CONTAINS AN ERROR

pragma solidity ^0.4.0;

contract C {
    uint someVariable;
    uint[] data;

    function f() {
        uint[] x;
        x.push(2);
```

```
        data = x;
    }
}
```

The type of the local variable `x` is `uint[] storage`, but since storage is not dynamically allocated, it has to be assigned from a state variable before it can be used. So no space in storage will be allocated for `x`, but instead it functions only as an alias for a pre-existing variable in storage.

What will happen is that the compiler interprets `x` as a storage pointer and will make it point to the storage slot `0` by default. This has the effect that `someVariable` (which resides at storage slot `0`) is modified by `x.push(2)`.

The correct way to do this is the following:

```
pragma solidity ^0.4.0;

contract C {
    uint someVariable;
    uint[] data;

    function f() {
        uint[] x = data;
        x.push(2);
    }
}
```

6.12.2 Advanced Questions

How do you get a random number in a contract? (Implement a self-returning gambling contract.)

Getting randomness right is often the crucial part in a crypto project and most failures result from bad random number generators.

If you do not want it to be safe, you build something similar to the coin flipper but otherwise, rather use a contract that supplies randomness, like the RANDAO.

Get return value from non-constant function from another contract

The key point is that the calling contract needs to know about the function it intends to call.

See ping.sol and pong.sol.

Get contract to do something when it is first mined

Use the constructor. Anything inside it will be executed when the contract is first mined.

See replicator.sol.

How do you create 2-dimensional arrays?

See 2D_array.sol.

Note that filling a 10x10 square of `uint8` + contract creation took more than `800,000` gas at the time of this writing. 17x17 took `2,000,000` gas. With the limit at 3.14 million... well, there's a pretty low ceiling for what you can create right now.

Note that merely «creating» the array is free, the costs are in filling it.

Note2: Optimizing storage access can pull the gas costs down considerably, because 32 `uint8` values can be stored in a single slot. The problem is that these optimizations currently do not work across loops and also have a problem with bounds checking. You might get much better results in the future, though.

What happens to a `struct`'s mapping when copying over a `struct`?

This is a very interesting question. Suppose that we have a contract field set up like such:

```
struct user {
    mapping(string => address) usedContracts;
}

function somefunction {
   user user1;
   user1.usedContracts["Hello"] = "World";
   user user2 = user1;
}
```

In this case, the mapping of the struct being copied over into the userList is ignored as there is no «list of mapped keys». Therefore it is not possible to find out which values should be copied over.

How do I initialize a contract with only a specific amount of wei?

Currently the approach is a little ugly, but there is little that can be done to improve it. In the case of a `contract A` calling a new instance of `contract B`, parentheses have to be used around `new B` because `B.value` would refer to a member of `B` called `value`. You will need to make sure that you have both contracts aware of each other's presence and that `contract B` has a `payable` constructor. In this example:

```
pragma solidity ^0.4.0;

contract B {
    function B() payable {}
}

contract A {
    address child;

    function test() {
        child = (new B).value(10)(); //construct a new B with 10 wei
    }
}
```

Can a contract function accept a two-dimensional array?

This is not yet implemented for external calls and dynamic arrays - you can only use one level of dynamic arrays.

What is the relationship between `bytes32` and `string`? Why is it that `bytes32 somevar = "stringliteral";` works and what does the saved 32-byte hex value mean?

The type `bytes32` can hold 32 (raw) bytes. In the assignment `bytes32 samevar = "stringliteral";`, the string literal is interpreted in its raw byte form and if you inspect `somevar` and see a 32-byte hex value, this is just `"stringliteral"` in hex.

The type `bytes` is similar, only that it can change its length.

Finally, `string` is basically identical to `bytes` only that it is assumed to hold the UTF-8 encoding of a real string. Since `string` stores the data in UTF-8 encoding it is quite expensive to compute the number of characters in the string (the encoding of some characters takes more than a single byte). Because of that, `string s; s.length` is not yet supported and not even index access `s[2]`. But if you want to access the low-level byte encoding of the string, you can use `bytes(s).length` and `bytes(s)[2]` which will result in the number of bytes in the UTF-8 encoding of the string (not the number of characters) and the second byte (not character) of the UTF-8 encoded string, respectively.

Can a contract pass an array (static size) or string or `bytes` (dynamic size) to another contract?

Sure. Take care that if you cross the memory / storage boundary, independent copies will be created:

```
pragma solidity ^0.4.0;

contract C {
    uint[20] x;

    function f() {
        g(x);
        h(x);
    }

    function g(uint[20] y) internal {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

The call to `g(x)` will not have an effect on `x` because it needs to create an independent copy of the storage value in memory (the default storage location is memory). On the other hand, `h(x)` successfully modifies `x` because only a reference and not a copy is passed.

Sometimes, when I try to change the length of an array with ex: `arrayname.length = 7;` I get a compiler error `Value must be an lvalue`. Why?

You can resize a dynamic array in storage (i.e. an array declared at the contract level) with `arrayname.length = <some new length>;`. If you get the «lvalue» error, you are probably doing one of two things wrong.

1. You might be trying to resize an array in «memory», or
2. You might be trying to resize a non-dynamic array.

```
int8[] memory memArr;         // Case 1
memArr.length++;              // illegal
int8[5] storageArr;           // Case 2
somearray.length++;           // legal
int8[5] storage storageArr2; // Explicit case 2
somearray2.length++;          // legal
```

Important note: In Solidity, array dimensions are declared backwards from the way you might be used to declaring them in C or Java, but they are access as in C or Java.

For example, `int8[][5] somearray;` are 5 dynamic `int8` arrays.

The reason for this is that `T[5]` is always an array of 5 `T`'s, no matter whether `T` itself is an array or not (this is not the case in C or Java).

Is it possible to return an array of strings (`string[]`) from a Solidity function?

Not yet, as this requires two levels of dynamic arrays (`string` is a dynamic array itself).

If you issue a call for an array, it is possible to retrieve the whole array? Or must you write a helper function for that?

The automatic *getter function* for a public state variable of array type only returns individual elements. If you want to return the complete array, you have to manually write a function to do that.

What could have happened if an account has storage value(s) but no code? Example: http://test.ether.camp/account/5f740b3a43fbb99724ce93a879805f4dc89178b5

The last thing a constructor does is returning the code of the contract. The gas costs for this depend on the length of the code and it might be that the supplied gas is not enough. This situation is the only one where an «out of gas» exception does not revert changes to the state, i.e. in this case the initialisation of the state variables.

https://github.com/ethereum/wiki/wiki/Subtleties

After a successful CREATE operation's sub-execution, if the operation returns x, 5 * len(x) gas is subtracted from the remaining gas before the contract is created. If the remaining gas is less than 5 * len(x), then no gas is subtracted, the code of the created contract becomes the empty string, but this is not treated as an exceptional condition - no reverts happen.

What does the following strange check do in the Custom Token contract?

```
require((balanceOf[_to] + _value) >= balanceOf[_to]);
```

Integers in Solidity (and most other machine-related programming languages) are restricted to a certain range. For `uint256`, this is `0` up to `2**256 - 1`. If the result of some operation on those numbers does not fit inside this range, it is truncated. These truncations can have serious consequences, so code like the one above is necessary to avoid certain attacks.

More Questions?

If you have more questions or your question is not answered here, please talk to us on gitter or file an issue.

Índice

A

abi, 122
abstract contract, **79**
access
    restricting, 143
account, **17**
addmod, 56, 108
address, 17, 41, 43
anonymous, 109
application binary interface, 122
array, 47, **48**
    allocating, **48**
    length, **49**
    literals, **49**
    push, **49**
asm, **85**
assembly, **85**
assert, 56, **63**, 108
assignment, 53, **61**
    destructuring, **61**
auction
    blind, 27
    open, 27

B

balance, 17, 41, 57, 108
ballot, 24
base
    constructor, **78**
base class, **76**
blind auction, 27
block, **16**, 55, 108
    number, 55, 108
    timestamp, 55, 108
bool, **39**
break, 58
Bugs, 146
byte array, 42
bytes, 44
bytes32, 42

C

C3 linearization, **79**
call, 41, 57
callcode, 19, 41, 57, 80
cast, **53**
coding style, 129
coin, 16
coinbase, 55, 108
commandline compiler, **117**
comment, **37**
common subexpression elimination, 103
compiler
    commandline, 117
constant, **71**, 109
constant propagation, 103
constructor, 65
    arguments, 66
continue, 58
contract, 37, **65**
    abstract, **79**
    base, **76**
    creation, **65**
    interface, **80**
contract creation, 19
contracts
    creating, 60
cryptography, 56, 108

D

data, 55, 108
days, 54
declarations, 62
default value, 62
delegatecall, 19, 41, 57, 80
delete, **53**
deriving, **76**
difficulty, 55, 108

do/while, 58

E

ecrecover, 56, 108
else, 58
enum, 37, 44
escrow, 33
ether, 54
ethereum virtual machine, **17**
event, 15, 37, **74**
evm, **17**
evmasm, **85**
exception, **63**
external, 67, 109

F

fallback function, **73**
false, **39**
finney, 54
fixed, **40**
fixed point number, **40**
for, 58
function, 37
    call, 18, **59**
    external, 59
    fallback, 73
    getter, **68**
    internal, 59
    modifier, 37, **69**, 143, 145
function type, **45**

G

gas, **17**, 55, 108
gas price, **17**, 55, 108
getter
    function, **68**
goto, 58

H

hours, 54

I

if, 58
import, **35**
indexed, 109
inheritance, **76**
    multiple, **79**
inline
    arrays, **49**
installing, **19**
instruction, **18**
int, **40**
integer, **40**
interface contract, **80**
internal, 67, 109

K

keccak256, 56, 108

L

length, 49
library, 19, **80**, 83
linearization, **79**
linker, **117**
literal, 43, 44
    address, 43
    rational, 43
    string, 44
location, 47
log, 19, **75**
lvalue, 53

M

mapping, 15, **52**, 101
memory, **18**, 47
message call, **18**
minutes, 54
modifiers, 109
msg, 55, 108
mulmod, 56, 108

N

natspec, 37
new, 48, **60**
now, 55, 108
number, 55, 108

O

open auction, 27
optimizer, 103
origin, 55, 108

P

parameter, **58**
    input, 58
    output, 58
payable, 109
pragma, **35**
precedence, 107
private, 67, 109
public, 67, 109
purchase, 33
pure, 109
push, 49

R

reference type, **47**

remote purchase, 33
require, 56, **63**, 108
return, 58
revert, 56, **63**, 108
ripemd160, 56, 108

S

scoping, **62**
seconds, 54
selfdestruct, 19, 57, 108
send, 41, 57, 108
sender, 55, 108
set, 81
sha256, 56, 108
solc, **117**
source file, 35
source mappings, 104
stack, **18**
state machine, 144
state variable, 37, 101
storage, 17, **18**, 47, 101
string, 44
struct, 37, 47, **51**
style, 129
submoeda, **14**
super, 108
switch, 58
szabo, 54

T

this, 57, 108
throw, **63**
time, 54
timestamp, 55, 108
transaction, 16, **17**
transfer, 41, 57
true, **39**
type, 39
    conversion, **53**
    deduction, **54**
    function, **45**
    reference, **47**
    struct, **51**
    value, **39**

U

ufixed, **40**
uint, **40**
using for, 81, **83**

V

value, 55, 108
value type, **39**
var, **54**
version, 35
view, 109
visibility, **67**, 109
voting, 24

W

weeks, 54
wei, 54
while, 58
withdrawal, 142

Y

years, 54