Solidity Documentation

Yayım 0.8.15

Ethereum

06 Nis 2023

Temeller

Uyarı: You are reading a community translation of the Solidity documentation. The Solidity team can give no guarantees on the quality and accuracy of the translations provided. The English reference version is and will remain the only officially supported version by the Solidity team and will always be the most accurate and most up-to-date one. When in doubt, please always refer to the English (original) documentation.

Solidity akıllı sözleşmelerin (smart contracts) uygulanması için geliştirilen nesne yönelimli, üst düzey bir programlama dilidir. Akıllı sözleşmeler Ethereum ağı içindeki hesapların hareketlerini ve davranışlarını yöneten programlardır.

Solidity Ethereum Sanal Makinası (ESM) (Ethereum Virtual Machine) hedeflenerek dizayn edilmiş bir curly-bracket dilidir. C++, Python ve JavaScript gibi dillerden ilham alınarak oluşturulmuştur. Solidity'nin başka hangi dillerden ilham aldığı hakkındaki detaylı bilgiyi *ilham alınan diller* bölümünde bulabilirsiniz.

Solidity statik olarak yazılmış olmasının yanı sıra, kütüphaneleri, kullanıcı tanımlı karmaşık türleri ve kalıtımsallığı destekler.

Solidity'le kullanıcılar için oylama, crowdfunding, blind auctions ve çoklu-imza cüzdanları gibi kullanımlara yönelik akıllı sözleşmeler oluşturabilirsiniz.

Sözleşmelerin gönderimini yaparken, en son yayınlanan Solidity sürümünü kullanmalısınız. İstisnai durumlar dışında, yalnızca son sürüm güvenlik düzeltmeleri güncellemelerini alır. Ayrıca, önemli değişikliklerinin yanı sıra yeni özellikler düzenli olarak tanıtılmaktadır. Bu hızlı değişimleri belirtmek için bir 0.y.z sürüm numarası kullanıyoruz.

İpucu: Solidity kısa bir süre önce birçok yenilik ve önemli değişiklikler getiren 0.8.x sürümünü yayınladı. Değişiklikleri mutlaka okuyun *tam liste*.

Solidity'yi veya bu dokümantasyonu geliştirmek için fikirlere her zaman açığız, Daha fazla ayrıntı için *katkıda bulunanlar rehberi* sayfamızı okuyun.

Uyarı: Bu belgeyi, sol alt köşedeki sürümler menüsüne tıklayarak ve tercih edilen indirme biçimini seçerek PDF, HTML veya Epub olarak indirebilirsiniz.

BÖLÜM 1

Hadi Başlayalım

1. Akıllı Sözleşmelerin Temellerini Anlama

Eğer akıllı sözleşmeler kavramında yeniyseniz "Akıllı Sözleşmelere Giriş" bölümünü araştırarak başlamanızı öneririz. Bu bölüm aşağıdakileri kapsar:

- Solidity ile yazılmış *Basit bir akıllı sözleşme örneği*.
- *Blockchain Temelleri*.
- *Ethereum Sanal Makinası (Ethereum Virtual Machine)*.

2. Solidity ile Tanışın

Temel bilgilere alıştıktan sonra, *"Örneklerle Solidity"* bölümünü okumanızı öneririz. Ve ayrıca "Dil Tanımları" bölümünü inceleyerek dilin temel kavramlarını anlayabilirsiniz..

3. Solidity Derleyicisini İndirme

Solidity derleyicisini indirmenin birçok yolu vardır, tercih edeceğiniz yola göre *indirme sayfası* 'da bulunan adımları izleyin.

İpucu: Remix IDE ile birlikte kod örneklerini doğrudan tarayıcınızda deneyebilirsiniz. Remix, Solidity'yi yerel olarak yüklemenize gerek kalmadan Solidity akıllı sözleşmelerini yazmanıza, dağıtmanıza ve yönetmenize olanak tanıyan web tarayıcısı tabanlı bir IDE'dir.

Uyarı: İnsanlar kodlama yaparken, hataları olabilir. Akıllı sözleşmelerinizi yazarken belirlenmiş en iyi yazılım geliştirme uygulamalarını izlemelisiniz. Buna kod incelemesi, kodunuzu test etme, denetimler ve correctness proofs dahildir. Akıllı sözleşme kullanıcıları bazen kod konusunda yazarlarından daha emin olabilirler, blockchain ve akıllı sözleşmelerin dikkat edilmesi gereken kendine özgü sorunları vardır, bu nedenle üretim kodu(production code) üzerinde çalışmadan önce *Güvenlikle İlgili Değerlendirmeler* bölümünü okuduğunuzdan emin olun.

4. Daha Fazla Bilgi Edinin

Ethereum ağı üzerinde merkeziyetsiz uygulamalar oluşturma hakkında daha fazla bilgi edinmek istiyorsanız, Ethereum Geliştirici Kaynakları size Ethereum ile ilgili daha fazla genel dokümantasyon, çok çeşitli öğreticiler, araçlar ve framework'ler(Yazılım iskeleti) konusunda yardımcı olabilir.

Eğer herhangi bir sorunuz varsa, Ethereum StackExchange, veya Gitter kanalımıza sorabilirsiniz.

BÖLÜM 2

Çeviriler

Topluluk'tan bazı gönüllüler bu belgeyi farklı dillere çevirmemize yardımcı oluyor. Bu sebeple çevirilerin farklı derecelerde bütünlük ve güncelliğe sahip olduğunu unutmayın. İngilizce versiyonunu referans olarak alın.

Sol alt köşedeki açılır menüye tıklayarak ve tercih ettiğiniz dili seçerek diller arasında geçiş yapabilirsiniz.

- Fransızce
- Endonezya Dili
- Farsça
- Japonca
- Korece
- Çince

Not: Kısa süre önce topluluk çalışmalarını kolaylaştırmak ve düzene koymak için yeni bir GitHub organizasyonu ve çeviri için bir iş akışı(workflow) kurduk. Yeni bir dile nasıl başlayacağınız veya var olan çevirilere nasıl katkıda bulunacağınız hakkında bilgi için lütfen çeviri kılavuzuna bakın.

BÖLÜM 3

İçindekiler

Anahtar Kelime Dizini, Arama Sayfası

3.1 Akıllı Sözleşmelere Giriş

3.1.1 Basit Bir Akıllı Sözleşme

Bir değişkenin değerini atayan ve bunu diğer sözleşmelerin erişimine sunan temel bir örnekle başlayalım. Şu an her şeyi anlamadıysanız sorun değil, birazdan daha fazla ayrıntıya gireceğiz.

Depolama

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

contract SimpleStorage {
    uint storedData;

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

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

İlk satır size kaynak kodunun GPL 3.0 sürümü altında lisanslanmış olduğunu söyler. Kaynak kodu yayınlamanın standart olduğu bir ortamda makine tarafından okunabilen lisans belirleyicileri önemlidir.

Bir sonraki satır, kaynak kodun Solidity 0.4.16'dan başlayarak 0.9.0'a kadar (0.9.0 hariç) olan sürümler için yazıldığını belirtir. Bu, sözleşmenin farklı sonuçlar verebileceği yeni bir derleyici sürümü ile derlenemez olmasını sağlamak içindir. *Pragmalar*, derleyiciler için kaynak kodun nasıl ele alınacağına ilişkin ortak talimatlardır (ör. pragma once).

Solidity kapsamında olan bir sözleşme, Ethereum blok zinciri ağında belirli bir adreste bulunan kod (*fonksiyonlar*) ve veri (*durum*) bütünüdür. `uint storeData;` satırı, `uint` türünde (*256* bitlik bir *u*nsigned (pozitif) *int*eger) `storedData` adlı bir durum değişkeni tanımlar . Bunu, veritabanını yöneten kodun fonksiyonlarını çağırarak sorgulayabileceğiniz ve değiştirebileceğiniz, veritabanındaki bir bilgi olarak düşünebilirsiniz. Ve bu örnektede, "set" ve "get" fonksiyonları değişkenin değerini değiştirmek veya çağırmak için tanımlanmıştır.

Mevcut sözleşmenizde bulunan bir durum değişkenine erişmek için genellikle `this.` önekini eklemezsiniz, doğrudan adı üzerinden erişirsiniz. Diğer bazı dillerin aksine, bu öneki atlamak sadece kodun görünüşünü iyileştirmek için değildir. Bu düzenleme değişkene erişmek için de tamamen farklı sonuçlar doğurabilir, fakat bu konuya daha sonra detaylıca değineceğiz.

Bu sözleşme, (Ethereum temel yapısı nedeniyle) herhangi birinin, tanımladığınız bu değişkenin (yayınlamanızı engelleyecek (uygulanabilir) bir yol olmadan) dünyadaki herkes tarafından erişilebilmesi için saklamaktan başka pek bir işe yaramıyor. Herhangi biri `set` fonksiyonunu farklı bir değer tanımlamak için tekrar çağırabilir ve değişkeninizin üzerine yazdırabilir, fakat bu değiştirilen değişkenin kayıtları blok zincirinin geçmişinde saklanmaya devam eder. İlerleyen zamanlarda, değişkeni yalnızca sizin değiştirebilmeniz için nasıl erişim kısıtlamalarını koyabileceğinizi göreceksiniz.

Uyarı: Unicode metni kullanırken dikkatli olunması gerekir, çünkü benzer görünümlü (hatta aynı) karakterler farklı kod işlevlerine sahip olabilir ve farklı bir bayt dizisi olarak kodlanabilirler.

Not: Sözleşmenizin tüm tanımlayıcı değerleri (sözleşme isimleri, fonksiyon isimleri ve değişken isimleri) ASCII karakter seti ile sınırlıdır. UTF-8 ile kodlanmış verileri string değişkenlerinde saklamak mümkündür.

Alt Para Birimi Örneği

Aşağıdaki sözleşme, bir kripto para biriminin en basit biçiminin bir örneğidir. Bu sözleşme, yalnızca sözleşme sahibinin (oluşturucusunun) yeni paralar oluşturmasına izin verir (farklı para oluşturma planları ayarlamak mümkündür). Herkes kullanıcı adı ve parolayla kayıt olmadan birbirine para gönderebilir. Tüm bunlar için tek ihtiyacınız olan şey sadece Ethereum anahtar çiftidir.

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

contract Coin {
    // "public" anahtar kelimesi, değişkenleri
    // diğer sözleşmeler tarafından erişilebilir kılar
    address public minter;
    mapping (address => uint) public balances;

    // Event'ler istemcilerin sözleşme üzerinde yaptığınız
    // değişikliklere tepki vermelerini sağlar
    event Sent(address from, address to, uint amount);

    // Constructor kodu sadece sözleşme
    // oluşturulduğunda çalışır
    constructor() {
        minter = msg.sender;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }

    // Yeni oluşturulan bir miktar parayı adrese gönderir
    // Yalnızca sözleşme yaratıcısı tarafından çağrılabilir
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Error'ler bir işlemin neden başarısız olduğu hakkında
    // bilgi almanızı sağlar. Fonksiyonu çağıran kişiye
    // bilgilendirme amacıyla bir sonuç döndürürler.
    error InsufficientBalance(uint requested, uint available);

    // Fonksiyonu çağıran kişinin var olan paralarından
    // alıcı adrese para gönderir.
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

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

Bu sözleşmede bazı yeni kavramlar tanıtılıyor, hadi hepsini teker teker inceleyelim.

`address public minter;` satırı *address* türündeki bir durum değişkenini tanımlıyor. `address` değişken türü, herhangi bir aritmetik işlemin uygulanmasına izin vermeyen 160 bitlik bir değerdir. Sözleşmelerin adreslerini veya *harici hesaplar*'a ait bir anahtar çiftinin teki olan public key hash'ini saklamak için uygundur.

`public` anahtar sözcüğü otomatik olarak durum değişkeninin mevcut değerine sözleşme dışından da erişmenizi sağlayan bir fonksiyonu oluşturur. Bu anahtar kelime olmadan, diğer sözleşmelerin bu değişkene erişme yolu yoktur. Derleyici tarafından oluşturulan fonksiyonun kodu aşağıdakine eşdeğerdir (şimdilik `external` ve `view` i göz ardı edin):

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

Yukarıdaki gibi bir fonksiyonu koda kendiniz de ekleyebilirsiniz, fakat aynı isimde olan bir fonksiyon ve durum değişkeniniz olur. Bunu yapmanıza gerek yoktur, bu işi derleyici sizin yerinize halleder.

Diğer satır olan `mapping (address => uint) public balances;` de bir public durum değişkeni oluşturuyor, fakat bu değişken biraz daha karmaşık bir veri yapısına sahip. Burada bulunan ref:*mapping <mapping-types>* türü adresleri *unsigned integers* ile eşliyor.

Mapping'ler, sanal bir şekilde tanımlanıp değer atanan hash tabloları olarak görülebilir. Bu yapıda mümkün olan her anahtar değeri tanımlandığı andan itibaren bulunur ve bu anahtarların eşlendiği değer (byte gösterminde) sıfırdır. Ancak, bir mapping'in ne tüm anahtarlarının ne de tüm değerlerinin bir listesini elde etmek mümkün değildir. Bunun için mapping'e eklediğiniz değerleri kaydedin veya buna gerek duyulmayacak bir durumda kullanın. Hatta daha da iyisi bir liste tutun ya da daha uygun bir veri türünü kullanmayı deneyin.

`public` anahtar kelimesi ile oluşturulmuş aşağıda bulunan *çağırıcı fonksiyon*, mapping örneğine göre biraz daha kar-

maşık bir yapıya sahiptir:

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

Bu fonksiyonu tek bir hesabın bakiyesini sorgulamak için kullanabilirsiniz.

`event Sent(address from, address to, uint amount);` satırı `send` fonksiyonunun son satırında yayılan (emit) bir *"olay (event)"* bildirir. Web uygulamaları gibi Ethereum istemcileri, blok zincirinde yayılan (emit) bu olaylardan (event) fazla maliyet olmadan veri alabilir. Event yayılır yayılmaz, veri alıcısı `from`, `to` ve `amount` argümanlarını alır, bu da alım satım işlemlerinin takip edilmesini mümkün kılar.

Bu olayı(event) dinlemek amacıyla, `Coin` sözleşme nesnesini oluşturmak için web3.js kütüphanesini kullanan aşağıdaki JavaScript kodunu kullanabilirsiniz. Ve herhangi bir kullanıcı arayüzü (user interface), otomatik olarak oluşturulan `balances` fonksiyonunu yukarıdan sizin için çağırır:

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

constructor fonksiyonu, sözleşmenin oluşturulması sırasında çalıştırılan ve daha sonra çağırılamayan özel bir fonksiyondur. Bu örnekte ise constructor fonksiyonu sözleşmeyi oluşturan kişinin adresini kalıcı olarak depoluyor. `msg` değişkeni (`tx` ve `block` ile birlikte), blok zincirine erişim izini veren özellikleri olan *özel bir global değişken* dir. `msg.sender` her zaman varsayılan fonksiyonu (external) çağıran kişinin adresini döndürür.

Sözleşmeyi oluşturan ve hem kullanıcıların hemde sözleşmelerin çağırabileceği fonksiyonlar `mint` ve `send` dir.

`mint` fonksiyonu yeni oluşturulan bir miktar parayı başka bir adrese gönderir. ref:*require <assert-and-require>* fonksiyon çağrısı, karşılanmadığı takdirde tüm değişiklikleri geri döndüren koşulları tanımlar. Bu örnekte, `require(msg.sender == minter);` yalnızca sözleşme yaratıcısının `mint` fonksiyonunu çağırabilmesini sağlar. Genel olarak, sözleşme yaratıcısı istediği kadar para basabilir, fakat belirili bir noktadan sonra bu durum "owerflow" adı verilen bir olaya yol açacaktır. Varsayılan *Checked arithmetic* nedeniyle, `balances[receiver] += amount;` ifadesi taşarsa, yani `balances[receiver] + amount` ifadesi `uint` maksimum değerinden (`2**256 - 1`) büyükse işlemin geri döndürüleceğini unutmayın. Bu, `send` fonksiyonundaki `balances[receiver] += amount;` ifadesi için de geçerlidir.

Hatalar, bir koşulun veya işlemin neden başarısız olduğu hakkında fonksiyonu çağıran kişiye daha fazla bilgi sağlamanıza olanak tanır. Hatalar *revert ifadesi* ile birlikte kullanılır. `revert` ifadesi, `require` fonksiyonuna benzer bir şekilde tüm değişiklikleri koşulsuz olarak iptal eder ve geri alır, ancak aynı zamanda bir hatanın daha kolay hata ayıklanabilmesi veya tepki verilebilmesi için hatanın adını ve çağıran kişiye (ve nihayetinde ön uç uygulamaya veya blok gezginine) sağlanacak ek verileri sağlamanıza olanak tanır.

`send` fonksiyonu, herhangi biri tarafından (hali hazırda bir miktar paraya sahip olan) başka birine para göndermek için kullanılabilir. Gönderen kişinin göndermek için yeterli bakiyesi yoksa, `if` koşulu doğru (true) olarak değerlendirilir. Sonuç olarak `revert` fonksiyonu, `InsufficientBalance` (Yetersiz bakiye) hatasını kullanarak göndericiye hata ayrıntılarını sağlarken işlemin başarısız olmasına neden olacaktır.

Not: Bu sözleşmeyi bir adrese para (coin) göndermek için kullanırsanız, bir blok zinciri gezgininde (explorer) o adrese baktığınızda hiçbir şey göremezsiniz, çünkü para (coin) gönderdiğiniz kayıt ve değişen bakiyeler yalnızca bu coin sözleşmesinin veri deposunda saklanır. Event'leri kullanarak, yeni coin'inizin işlemlerini ve bakiyelerini izleyen bir

"blok zinciri gezgini (explorer)" oluşturabilirsiniz, ancak coin sahiplerinin adreslerini değil, coin'in sözleşme adresini incelemeniz gerekir.

3.1.2 Blok Zinciri Temelleri

Bir kavram olarak blok zincirleri anlamak programcılar için çok zor değildir. Bunun nedeni, komplikasyonların (madencilik (mining), hashing, elliptic-curve cryptography, peer-to-peer networks, etc.) çoğunun sadece platform için belirli bir dizi özellik ve vaat sağlamak için orada olmasıdır. Bu özellikleri olduğu gibi kabul ettiğinizde, altta yatan teknoloji hakkında endişelenmenize gerek kalmaz - yoksa Amazon'un AWS'sini kullanmak için dahili olarak nasıl çalıştığını bilmek zorunda mısınız?

İşlemler (Transactions)

Blok zinciri, küresel olarak paylaşılan, işlemsel bir veritabanıdır. Bu, herkesin yalnızca ağa katılarak veritabanındaki girdileri okuyabileceği anlamına gelir. Veritabanındaki bir şeyi değiştirmek istiyorsanız, diğerleri tarafından kabul edilmesi gereken bir "işlem" oluşturmanız gerekir. İşlem kelimesi, yapmak istediğiniz değişikliğin (aynı anda iki değeri değiştirmek istediğinizi varsayın) ya hiç yapılmadığını ya da tamamen uygulanmasını ifade eder. Ayrıca, işleminiz veritabanına uygulanırken başka hiçbir işlem onu değiştiremez.

Örnek olarak, elektronik para birimindeki tüm hesapların bakiyelerini listeleyen bir tablo hayal düşünün. Bir hesaptan diğerine transfer talep edilirse, veri tabanının işlemsel yapısı, tutar bir hesaptan çıkarılırsa, her zaman diğer hesaba eklenmesini sağlar. Herhangi bir nedenden dolayı tutarın hedef hesaba eklenmesi mümkün değilse, kaynak hesaptaki bakiye de değiştirilmez.

Ayrıca, bir işlem her zaman gönderen (yaratıcı) tarafından şifreli olarak imzalanır. Bu, veritabanındaki belirli değişikliklere erişimi korumayı kolaylaştırır. Kripto para birimi örneğinde, basit bir kontrol, yalnızca anahtarları hesaba katan bir kişinin hesaptan para aktarabilmesini sağlar.

Bloklar

Üstesinden gelinmesi gereken en büyük engellerden biri (Bitcoin açısından) "çifte harcama saldırısı" olarak adlandırılan bir olaydır: Ağda bir cüzdanı boşaltmak isteyen eşzamanlı iki işlem varsa ne olur? İşlemlerden sadece biri geçerli olabilir, tipik olarak önce kabul edilmiş olanı. Sorun, "ilk" in eşler arası ağda (peer-to-peer network) nesnel bir terim olmamasıdır.

Özetle tüm bunları düşünmenize gerk yoktur. İşlemlerin global olarak kabul edilen bir sırası sizin için seçilecek ve çatışma çözülecektir. İşlemler "blok" adı verilen bir yapıda bir araya getirilecek ve daha sonra yürütülerek tüm katılımcı düğümler arasında dağıtılacaktır. Eğer iki işlem birbiriyle çelişirse, ikinci olan işlem reddedilecek ve bloğun bir parçası olmayacaktır.

Bu bloklar zaman içinde doğrusal bir dizi oluşturur ve "blok zinciri" kelimesi de zaten buradan türemiştir. Bloklar zincire oldukça düzenli aralıklarla eklenir - Ethereum için bu süre kabaca her 17 saniye birdir.

"Sıra seçim mekanizmasının" ("madencilik" olarak adlandırılır) bir parçası olarak zaman zaman bloklar geri alınabilir, ancak bu sadece zincirin en "ucunda" gerçekleşir. Belirli bir bloğun üzerine ne kadar çok blok eklenirse, bu bloğun geri döndürülme olasılığı o kadar azalır. Yani işlemleriniz geri alınabilir ve hatta blok zincirinden kaldırılabilir, ancak ne kadar uzun süre beklerseniz, bu olasılık o kadar azalacaktır.

Not: İşlemlerin bir sonraki bloğa veya gelecekteki herhangi bir bloğa dahil edileceği garanti edilmez, çünkü işlemin hangi bloğa dahil edileceğini belirlemek, işlemi gönderen kişiye değil madencilere bağlıdır.

Sözleşmenizin gelecekteki çağrılarını planlamak istiyorsanız, bir akıllı sözleşme otomasyon aracı veya bir oracle hizmeti kullanabilirsiniz.

3.1.3 Ethereum Sanal Makinası

Genel Bakış

Ethereum Sanal Makinesi veya ESM, Ethereum'daki akıllı sözleşmeler için çalışma ortamıdır. Bu alan yalnızca korumalı bir alan değil, aynı zamanda tamamen yalıtılmış bir alandır; yani ESM içinde çalışan kodun ağa, dosya sistemine ya da diğer süreçlere erişimi yoktur. Akıllı sözleşmelerin diğer akıllı sözleşmelere erişimi bile sınırlıdır.

Hesaplar

Ethereum'da aynı adres alanını paylaşan iki tür hesap vardır: Public anahtar çiftleri (yani insanlar) tarafından kontrol edilen **harici hesaplar** ve hesapla birlikte depolanan kod tarafından kontrol edilen **sözleşme hesapları**.

Harici bir hesabın adresi açık (public) anahtardan belirlenirken, bir sözleşmenin adresi sözleşmenin oluşturulduğu anda belirlenir ("nonce" olarak adlandırılan yaratıcı adres ve bu adresten gönderilen işlem sayısından türetilir).

Hesabın kod depolayıp depolamadığına bakılmaksızın, iki tür ESM tarafından eşit olarak değerlendirilir.

Her hesabın, 256-bit sözcükleri **storage** adı verilen 256-bit sözcüklere eşleyen kalıcı bir anahtar-değer deposu vardır.

Ayrıca, her hesabın Ether cinsinden bir **bakiyesi** vardır (tam olarak "Wei" cinsinden, `1 ether` `10**18 wei` dir) ve bu Ether içeren işlemler gönderilerek değiştirilebilir.

İşlemler

İşlem, bir hesaptan diğerine gönderilen bir mesajdır (aynı veya boş olabilir, aşağıya bakınız). İkili verileri ("yük" olarak adlandırılır) ve Ether içerebilir.

Hedef hesap kod içeriyorsa, bu kod çalıştırılır ve sonucunda elde erilen veri yükü girdi olarak kabul edilir.

Hedef hesap ayarlanmamışsa (işlemin alıcısı yoksa veya alıcı `null` olarak ayarlanmışsa), işlem **yeni bir sözleşme** oluşturur. Daha önce de belirtildiği gibi, bu sözleşmenin adresi sıfır adres değil, göndericiden ve gönderilen işlem sayısından ("nonce") türetilen bir adrestir. Böyle bir sözleşme oluşturma işleminin yükü ESM bytecode'u olarak alınır ve çalıştırılır. Bu uygulamanın çıktı verileri kalıcı olarak sözleşmenin kodu olarak saklanır. Bu, bir sözleşme oluşturmak için sözleşmenin gerçek kodunu değil, aslında yürütüldüğünde bu kodu döndüren kodu gönderdiğiniz anlamına gelir.

Not: Bir sözleşme oluşturulurken, kodu hala boştur. Bu nedenle, constructor fonksiyonu çalışmayı bitirene kadar yapım aşamasındaki sözleşmeyi geri çağırmamalısınız.

Gas

Oluşturulduktan sonra, her işlem, işlemin kaynağı (`tx.origin`) tarafından ödenmesi gereken belirli bir **gas** miktarı ile ücretlendirilir. ESM işlemi gerçekleştirirken, gas belirli kurallara göre kademeli olarak tüketilir. Gas herhangi bir noktada tükenirse (yani negatif olursa), yürütmeyi sona erdiren ve mevcut çağrı çerçevesinde durumunda yapılan tüm değişiklikleri geri alan bir out-of-gas (gas bitti) istisnası tetiklenir.

Bu mekanizma, ESM'in çalışma süresinin tasarruflu bir şekilde kullanılmasını teşvik eder ve aynı zamanda ESM yürütücülerinin (yani madencilerin / stakerların) çalışmalarını telafi eder. Her blok maksimum miktarda gaza sahip olduğundan, bir bloğu doğrulamak için gereken iş miktarını da sınırlanmış olur.

Gas ücreti, işlemin yaratıcısı tarafından yani gönderen hesabından `gaz_ücreti * gaz` miktarında ödemek zorunda olduğu bir değerdir. Uygulamadan sonra bir miktar gaz kalırsa, bu miktar işlemi çalıştıran kişiye iade edilir. Değişikliği geri döndüren bir istisna olması durumunda, kullanılmış gas'ın iadesi yapılmaz.

ESM yürütücüleri bir işlemi ağa dahil edip etmemeyi seçebildiğinden, işlem gönderenler düşük bir gas fiyatı belirleyerek sistemi kötüye kullanamazlar.

Depolama, Bellek ve Yığın

Ethereum Sanal Makinesi'nin veri depolayabileceği üç alan vardır: storage (depolama), memory (bellek) ve stack (yığın).

Her hesap, fonksiyon çağrıları ve işlemler arasında kalıcı olan **storage** adlı bir veri alanına sahiptir. Depolama, 256 bit kelimeleri 256 bit kelimelerle eşleyen bir anahtar/değer deposudur. Bir sözleşmenin içinden depolamayı belirtmek mümkün değildir, depolamayı okumak da maliyetlidir ancak depolamayı başlatmak ve değiştirmek daha da maliyetlidir. Bu maliyet nedeniyle, kalıcı depolama alanında depoladığınız verinin miktarını sözleşmenin çalışması için gereken en azami miktara indirmelisiniz. Ayrıca türetilmiş hesaplamalar, önbelleğe alma ve toplamalar gibi verileri sözleşmenin dışında depolamalısınız. Bir sözleşme, kendi depolama alanı dışında herhangi bir depolama alanını ne okuyabilir ne de bu alandaki verileri değiştirebilir.

İkincisi ise, **memory** (bellek) olarak adlandırılan ve bir sözleşmenin her ileti çağrısı için yeniden oluşturulmuş bir örneğini alan bir veri alanıdır. Bellek doğrusaldır ve bayt düzeyinde adreslenebilir, ancak okumalar 256 bit genişlikle sınırlıyken, yazmalar 8 bit veya 256 bit genişliğinde olabilir. Daha önceden dokunulmamış bir bellek kelimesine (yani bir kelime içindeki herhangi bir ofsete) erişirken (okurken veya yazarken) bellek bir kelime (256 bit) kadar genişletilir. Bu genişletilme sırasında gas maliyeti ödenmelidir. Bellek büyüdükçe daha maliyetli olmaya başlıyacaktır (söz konusu artış maliyetin karesi olarak artmaya devam edecektir).

ESM, kayıt makinesi değil yığın makinesi olduğundan tüm hesaplamalar **stack** (yığın) adı verilen bir veri alanında gerçekleştirilir. Bu alan maksimum 1024 eleman boyutuna sahiptir ve 256 bitlik kelimeler içerir. Yığına erişim aşağıdaki şekilde üst uçla sınırlıdır: En üstteki 16 elemandan birini yığının en üstüne kopyalamak veya en üstteki elemanı altındaki 16 elemandan biriyle değiştirmek mümkündür. Diğer tüm işlemler yığından en üstteki iki (veya işleme bağlı olarak bir veya daha fazla) elemanı alır ve sonucu yığının üzerine iter. Elbette yığına daha derin erişim sağlamak için yığın elemanlarını depolama alanına veya belleğe taşımak mümkündür, ancak önce yığının üst kısmını çıkarmadan yığının daha derinlerindeki rastgele elemanlara erişmek mümkün değildir.

Yönerge Seti

ESM'nin komut seti, uzlaşma sorunlarına neden olabilecek yanlış veya tutarsız uygulamalardan kaçınmak için minimum düzeyde tutulmuştur. Tüm komutlar temel veri tipi olan 256 bitlik kelimeler veya bellek dilimleri (veya diğer bayt dizileri) üzerinde çalışır. Her zamanki aritmetik, bit, mantıksal ve karşılaştırma işlemleri mevcuttur. Koşullu ve koşulsuz atlamalar mümkündür. Ayrıca, sözleşmeler mevcut bloğun numarası ve zaman bilgisi gibi ilgili özelliklerine erişebilir.

Tam bir liste için lütfen satır içi montaj belgelerinin bir parçası olarak *işlem kodu (opcode) listeleri* belgesine bakın.

Mesaj Çağırıları

Sözleşmeler, mesaj çağrıları aracılığıyla diğer sözleşmeleri çağırabilir veya sözleşme dışı hesaplara Ether gönderebilir. Mesaj çağrıları, bir kaynak, bir hedef, veri yükü, Ether, gas ve geri dönüş verilerine sahip olmaları bakımından işlemlere benzerler. Aslında, her işlem üst düzey bir mesaj çağrısından oluşur ve bu da başka mesaj çağrıları oluşturabilir.

Bir sözleşme, kalan **gas'ın** ne kadarının iç mesaj çağrısı ile gönderilmesi gerektiğine ve ne kadarını tutmak istediğine karar verebilir. İç çağrıda yetersiz-gas dışında bir istisna meydana gelirse (veya başka bir istisna), bu durum yığına yerleştirilen bir hata değeri ile bildirilir. Bu durumda, sadece çağrı ile birlikte gönderilen gas miktarı kullanılır. Solidity dilinde, bu gibi istisnaların oluşması varsayılan olarak manuel başka zincirleme istisnalar da yaratmaya meyilli olduğundan totalde yığınını "kabarcıklandıran" durum olarak nitelendirilir.

Daha önce de belirtildiği gibi, çağrılan sözleşme (arayan ile aynı olabilir) belleğin yeni temizlenmiş bir örneğini alır ve **calldata** adı verilen ayrı bir alanda sağlanacak olan çağrı yüküne (payload) erişebilir. Yürütmeyi tamamladıktan sonra, arayanın belleğinde arayan tarafından önceden ayrılmış bir konumda saklanacak olan verileri döndürebilir. Tüm bu çağrılar tamamen eşzamanlıdır.

Çağrılar, 1024 bitlik alanla ile sınırlıdır; bu, daha karmaşık işlemler için tekrarlamalı çağrılar yerine döngüler tercih edileceği anlamına gelir. Ayrıca, bir mesaj çağrısında gazın sadece 63 / 64'ü iletilebilir; bu, pratikte 1000 bit'ten daha az bir alan sınırlamasına neden olur.

Delegatecall / Çağrı Kodu ve Kütüphaneler

Bir mesaj çağrısı ile temelde aynı anlama gelen **delegatecall**, hedef adresteki kodun arama sözleşmesi bağlamında (yani adresinde) yürütülmesi ve `msg.sender` ve `msg.value` değerlerinin değiştirilememesi gibi özellikleri ile mesaj çağrısının özel bir çeşidi olarak kabul edilir.

Bu, bir sözleşmenin çalışma zamanında farklı bir adresten dinamik olarak kod yükleyebileceği anlamına gelir. Depolama, geçerli adres ve bakiye hala çağıran sözleşmeye atıfta bulunurken, yalnızca kod çağrılan adresten aktarılır.

Karmaşık bir veri yapısını uygulamak için bir sözleşmenin depolama alanına uygulanabilen ve yeniden kullanılabilen bir kütüphane kodu örnek olarak verilebilir.

Kayıtlar (Logs)

Verileri, tamamen blok seviyesine kadar haritalayan özel olarak indekslenmiş bir veri yapısında depolamak mümkündür. **Kayıtlar** (log) olarak adlandırılan bu özellik, Solidity tarafından *event'lerin* uygulanmasını için kullanılır. Sözleşmeler, oluşturulduktan sonra kayıt verilerine erişemez, ancak bunlara blok zincirinin dışından etkin bir şekilde erişilebilir. Kayıt edilen verilerinin bir kısmı bloom filtrelerinde depolandığından, bu verileri verimli ve kriptografik olarak güvenli bir şekilde aramak mümkündür, böylece tüm zinciri indirmek zorunda kalmayan ağ elemanları(peer) ("hafif istemciler" olarak adlandırılır) yine de bu günlükleri bulabilir.

Create

Sözleşmeler, özel bir opcode kullanarak başka sözleşmeler bile oluşturabilir (bunu, hedef adresi boş bırakarak yaparlar). Bu arama çağrıları ve normal mesaj çağrıları arasındaki tek fark, açığa çıkan veri yükünün yürütülmesi ve sonucun kod olarak saklanarak arayan tarafın(yaratıcının) yığındaki yeni sözleşmenin adresini almasıdır.

Devre Dışı Bırakma ve Kendini İmha

Blok zincirinden bir kodu kaldırmanın tek yolu, söz konusu adresteki bir sözleşmenin selfdestruct işlemini gerçekleştirmesidir. Bu adreste depolanan kalan Ether belirlenen bir hedefe gönderilir ve ardından depolama ve kod durumdan kaldırılır. Teoride sözleşmeyi kaldırmak iyi bir fikir gibi görünse de, biri kaldırılan sözleşmelere Ether gönderirse, Ether sonsuza dek kaybolacağından potansiyel olarak tehlikelidir.

Uyarı: Bir sözleşme `selfdestruct` ile kaldırılsa bile, hala blok zinciri geçmişinin bir parçasıdır ve muhtemelen çoğu Ethereum node`u tarafından saklanmaktadır. Yani `selfdestruct` kullanmak sabit diskten veri silmekle aynı şey değildir.

Not: Bir sözleşmenin kodu `selfdestruct` çağrısı içermese bile, `delegatecall` veya `callcode` kullanarak bu işlemi gerçekleştirebilir.

Sözleşmelerinizi devre dışı bırakmak istiyorsanız, bunun yerine tüm fonksiyonların geri alınmasına neden olan bazı iç durumları değiştirerek bunları devre dışı bırakmalısınız. Bu, Ether'i derhal iade ettiğinden sözleşmeyi kullanmayı imkansız kılar.

Önceden Derlenmiş Sözleşmeler (Precompiled Contracts)

Özel olan bir dizi küçük sözleşme adresi vardır: `1` ile (`8` dahil) `8` arasındaki adres aralığı, diğer sözleşmeler gibi çağrılabilen "önceden derlenmiş sözleşmeler" içerir, ancak davranışları (ve gaz tüketimleri) bu adreste saklanan ESM kodu tarafından tanımlanmaz (kod içermezler), bunun yerine ESM kendi yürütme ortamında yürütülür.

Farklı ESM uyumlu zincirler, önceden derlenmiş farklı bir sözleşme seti kullanabilir. Gelecekte Ethereum ana zincirine önceden derlenmiş yeni sözleşmelerin eklenmesi de mümkün olabilir, ancak mantıklı olarak bunların her zaman `1` ile `0xffff` (dahil) aralığında olmasını beklemelisiniz.

3.2 Solidity Derleyicisini Yükleme

3.2.1 Sürüm

Solidity sürümleri Semantic Sürümlemeyi takip eder. Ek olarak, ana sürüm 0'a (yani 0.x.y) sahip yama düzeyindeki sürümler, kırılma değişiklikleri(breaking changes) içermeyecektir. Bu, 0.x.y sürümü ile derlenen kodun z > y olduğu durumlarda 0.x.z ile derlenmesinin umulabileceği anlamına gelir.

Sürümlere ek olarak, geliştiricilerin gelecek özellikleri denemelerini ve erken geri bildirim sağlamalarını kolaylaştırmak amacıyla **gece geliştirme yapıları** (Nightly Development Builds diye de bilinir) sağlıyoruz. Bununla birlikte, nightly yapılar genellikle çok kararlı olsalar da, geliştirme kolundaki (branch) en yeni kodları içerdiklerini ve her zaman çalışacaklarının garanti edilmediğini unutmayın. Tüm emeklerimize karşın, hala gerçek sürümün bir parçası olmayacak belgelenmemiş ve/veya arızalı değişiklikler içerebilirler. Bunlar üretim amaçlı kullanım için uygun değillerdir.

Sözleşmeleri derleyip yüklerken Solidity'nin yayınlanan en son sürümünü kullanmalısınız. Bunun nedeni, kırılma değişikliklerinin yanı sıra yeni özelliklerin tanıtılması ve eski sürümlerdeki hataların düzenli olarak düzeltilmesinden kaynaklanmaktadır. Bu hızlı sürüm değişikliklerini belirtmek için şu anda 0.x sürüm numarası kullanıyoruz.

3.2.2 Remix

Solidity'i hızlı bir şekilde öğrenmek ve küçük akıllı sözleşmeler geliştirmek için Remix'i kullanmanızı tavsiye ediyoruz.

Remix'i online bir şekilde kullanabilirsiniz, bunun için herhangi bir şey indirip kurmanıza gerek yoktur. Remix'i internet bağlantısı olmadan da kullanmak istiyorsanız, https://github.com/ethereum/remix-live/tree/gh-pages adresine gidip sayfada açıklandığı gibi `.zip` dosyasını indirebilirsiniz. Remix, birden fazla Solidity sürümü yüklemenize gerek kalmadan gece yapılarını da test etmek için uygun bir seçenektir.

Bu sayfada bulunan diğer seçenekler de komut satırı için Solidity derleyicisini bilgisayarınıza nasıl kuracağınızı detaylı bir şekilde anlatmaktadır. Eğer daha büyük bir sözleşme üzerinde çalışıyorsanız veya daha fazla derleme seçeneğine ihtiyacınız varsa lütfen bir komut satırı derleyicisi seçin.

3.2.3 npm / Node.js

Solidity derleyicisi olan `solcjs` programını kurmanın kullanışlı ve taşınabilir bir yolu için `npm` programını kullanabilirsiniz. *solcjs* programı, bu sayfanın ilerleyen kısımlarında açıklanacak olan derleyiciye erişim yollarından daha az özelliğe sahiptir. `solc`.ref:*commandline-compiler`(komut satırı derleyicisi) dokümantasyonu tam özellikli derleyici olan ``solc`* kullandığınızı varsayar. `solcjs` kullanımı için oluşturulan belgeler kendi deposu içinde bulunmaktadır.

Not: solc-js projesi, Emscripten kullanılarak oluşturulan C++ *solc* projesinden türetilmiştir, bu da her ikisinin de aynı derleyici kaynak kodunu kullandığı anlamına gelir. Aynı zamanda *solc-js* doğrudan JavaScript projelerinde (Remix gibi) kullanılabilmektedir. Talimatlar için lütfen solc-js deposuna göz atın.

```
npm install -g solc
```

Not: Komut satırında çalışabilen kod``solcjs`` olarak adlandırılmıştır (Komut satırına "solcjs" yazarak çalıştırabilirsiniz).

`solcjs` komut satırı seçenekleri `solc` ile uyumlu değildir. Aynı zamanda çalışmak için `solc` komutuna ihtiyaç duyan araçlar (örneğin `geth` gibi) `solcjs` ile çalışmayacaktır.

3.2.4 Docker

Solidity yapılarında bulunan Docker imajları, `ethereum` kuruluşundaki `solc` imajlarını da kullanarak elde edilebilir. Yayınlanan en son sürüm için `stable` etiketini ve geliştirme kolundaki (branch) sağlam olmayabilecek stabil olmayan değişiklikler için `nightly` etiketini kullanabilirsiniz.

Docker imajı derleyicinin yürütülebilir dosyasını çalıştırır, bu sayede tüm değişkenleri derleyiciye iletebilirsiniz. Örneğin, aşağıdaki komut `solc` imajının (elinizde mevcut değilse) kararlı bir sürümünü çeker ve `--help` parametresini ileterek yeni bir konteynerde çalıştırır.

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

Etikette derleme sürümlerini de belirtebilirsiniz, örneğin 0.5.4 sürümü için:

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

Docker imajını kullanarak Solidity dosyalarını ana makinede derlemek istiyorsanız, girdi ve çıktı için yerel bir klasör bağladıktan sonra derlenecek olan sözleşmeyi belirtin. Örnek vermek gerekirse:

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

Ayrıca spesifik bir JSON arayüzünü de kullanabilirsiniz (Hardhat,Truffle gibi derleyiciyi araçlarıyla birlikte kullanırken tavsiye edilir). Bu arayüzü kullanırken, JSON girdisi bağımsız olduğu sürece herhangi bir dizini bağlamak gerekli değildir (yani *içeri aktarılan(import) geri çağrısı (callback)* tarafından yüklenmesi gereken herhangi bir harici dosyaya referans göstermez).

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

3.2.5 Linux Paketleri

Solidity'nin binary paketleri solidity/releases adresinde mevcuttur.

Ayrıca Ubuntu için PPA'larımız da bulunmaktadır, aşağıdaki komutları kullanarak en son kararlı sürümü edinebilirsiniz:

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

Gece sürümü de bu komutlar kullanılarak kurulabilir:

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

Ayrıca, bazı Linux dağıtımları kendi paketlerini sağlamaktadırlar. Fakat bu paketlerin bakımı doğrudan bizim tarafımızdan yapılmamaktadır. Bu paketler genellikle ilgili paket sorumluları tarafından güncel tutulmaktadır.

Örnek vermek gerekirse, Arch Linux en son geliştirme sürümü için paketlere sahiptir:

```
pacman -S solidity
```

Ayrıca bir snap paketi vardır, ancak **şu anda bakımı yapılmamaktadır**. Bu paket desteklenen tüm Linux dağıtımlarına yüklenebilir. Solc'un en son çıkan kararlı sürümünü yüklemek için:

```
sudo snap install solc
```

Solidity'nin en son değişiklikleri içeren son çıkan geliştirme sürümünün test edilmesine yardımcı olmak istiyorsanız, lütfen aşağıdaki komutları kullanın:

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

Not: `solc` snap`i katı bir sınırlama sistemine sahiptir. Bu snap paketleri için uygulanabilecek en güvenli moddur, tabi bu modda yalnızca `/home` ve `/media` dizinlerinizdeki dosyalara erişmek gibi sınırlamalarla birlikte gelmektedir. Daha fazla bilgi için lütfen Sıkı Snap Sınırlaması Sistemini Açıklamak bölümüne gidin.

3.2.6 macOS Paketleri

Solidity derleyicisini, kaynaktan oluşturulmuş bir sürüm olarak Homebrew aracılığıyla dağıtıyoruz. Önceden oluşturulmuş olan "bottles"lar(binary paketleri) şu anda desteklenmemektedir.

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

Solidity'nin en son 0.4.x / 0.5.x sürümünü yüklemek için sırasıyla `brew install solidity@4` ve `brew install solidity@5` de kullanabilirsiniz.

Solidity'nin belirli bir sürümüne ihtiyacınız varsa, doğrudan Github'dan bir Homebrew "formula"sını (Formula, paket tanımı için kullanılan bir ifadedir) yükleyebilirsiniz.

Github'daki solidity.rb "commit"lerini görüntüleyin.

İstediğiniz bir sürümün commit hash'ini kopyalayabilir ve kendi makinenizde kontrol edebilirsiniz.

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

Bunu `brew` kullanarak yükleyin:

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

3.2.7 Statik Binaryler

Desteklenen tüm platformlar için geçmiş ve güncel derleyici sürümlerinin statik yapılarını içeren bir depoyu solc-bin adresinde tutuyoruz. Bu adreste aynı zamanda nightly yapıları da bulabilirsiniz.

Bu depo, son kullanıcıların ikili dosya sistemlerini kullanıma hazır hale getirmeleri için hızlı ve kolay bir yol olmasının yanı sıra üçüncü taraf araçlarla da dost olmayı (kolay bir şekilde etkileşimde bulunmayı) amaçlamaktadır:

- https://binaries.soliditylang.org adresine yansıtılan bu içerik herhangi bir kimlik doğrulama, hız sınırlaması veya git kullanma ihtiyacı olmadan HTTPS üzerinden kolayca indirilebilir.
- İçerik, tarayıcıda çalışan araçlar tarafından doğrudan yüklenebilmesi için doğru *Content-Type* başlıklarıyla ve serbest CORS yapılandırmasıyla sunulur.
- Binaryler için herhangi bir kurulum veya paketten çıkarma işlemi gerekmez (gerekli DLL'lerle birlikte gelen eski Windows yapıları hariç).
- Biz yüksek düzeyde geriye dönük uyumluluk için çabalamaktayız. Dosyalar eklendikten sonra, eski konumunda bulunan bir kısayol bağlantısı veya yönlendirme sağlanmadan kaldırılmaz veya taşınmaz. Ayrıca bu dosyalar hiçbir zaman değiştirilmez ve her zaman orijinal sağlama toplamı ile eşleşmelidirler. Buradaki tek istisna, olduğu gibi bırakıldığında yarardan çok zarar verme potansiyeli olan bozuk veya kullanılamaz dosyalar için geçerlidir.
- Dosyalar hem HTTP hem de HTTPS protokolleri üzerinden sunulur. Dosya listesini güvenli bir şekilde aldığınız (git, HTTPS, IPFS aracılığıyla veya yerel olarak önbelleğe aldığınız) ve indirdikten sonra ikili sayı sistemi dosyalarının hash'lerini doğruladığınız sürece, ikili dosyalar için HTTPS protokolünü kullanmanız gerekmez.

Aynı ikili sayı sistemi dosyaları genellikle Github üzerindeki Solidity sürüm sayfası nda bulunmaktadır. Aradaki fark, Github sürüm sayfasındaki eski sürümleri genellikle güncellemiyor olmamızdır. Bu, adlandırma kuralı değişirse onları

yeniden adlandırmadığımız ve yayınlandığı sırada desteklenmeyen platformlar için derlemeler eklemediğimiz anlamına gelir. Bu sadece `solc-bin` içinde gerçekleşir.

`solc-bin` deposu, her biri tek bir platformu temsil eden birkaç üst düzey dizin içerir. Her biri mevcut ikili sayı sistemi dosyalarını listeleyen bir `list.json` dosyası içerir. Örneğin `emscripten-wasm32/list.json` dosyasında bulunan 0.7.4 sürümü hakkındaki bilgileri aşağıda bulabilirsiniz:

```
{
  "path": "solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js",
  "version": "0.7.4",
  "build": "commit.3f05b770",
  "longVersion": "0.7.4+commit.3f05b770",
  "keccak256": "0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3",
  "sha256": "0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2",
  "urls": [
    "bzzr://16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1",
    "dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS"
  ]
}
```

Bu şu anlama gelmektedir:

- Binary dosyasını aynı dizinde solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js adı altında bulabilirsiniz. Dosyanın bir kısayol bağlantısı olabileceğini ve dosyayı indirmek için eğer git kullanmıyorsanız veya dosya sisteminiz kısayol bağlantılarını desteklemiyorsa bu dosyayı kendiniz çözümlemeniz gerekebileceğini unutmayın.
- Binary dosyası ayrıca https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js adresine de yansıtılır. Bu durumda git kullanımı gerekli değildir ve kısayol bağlantıları ya dosyanın bir kopyasını sunarak ya da bir HTTP yönlendirmesi döndürerek dosyanın şeffaf bir şekilde çözümlenmesini sağlar.
- Dosya ayrıca IPFS üzerinde QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS adresinde de mevcuttur.
- Dosya, gelecekte Swarm'da bulunan 16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1 adresinde mevcut olabilir.
- Binary'nin bütünlüğünü keccak256 hash değerini `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279` ile karşılaştırarak da doğrulayabilirsiniz. Hash, komut satırında sha3sum tarafından sağlanan `keccak256sum` yardımcı programı veya JavaScript'te ethereumjs-util'de bulunan keccak256() fonksiyonu kullanılarak da hesaplanabilir.
- Binary'nin bütünlüğünü sha256 hash değerini `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55a` ile karşılaştırarak da doğrulayabilirsiniz.

Uyarı: Güçlü bir şekilde geriye dönük uyumluluk gereksinimi sebebiyle depo bazı eski öğeler içerir, ancak yeni araçlar yazarken bunları kullanmaktan kaçınmalısınız:

- En iyi performansı istiyorsanız `bin/` yerine `emscripten-wasm32/` son çare (fallback) (`emscripten-asmjs/` geri dönüşü ile) kullanın. Biz 0.6.1 sürümüne kadar sadece asm.js ikili sayı sistemi dosyalarını sağlamıştık. 0.6.2`den itibaren çok daha iyi performans sağlayan *WebAssembly derlemeleri`_ne geçtik. Eski sürümleri wasm için yeniden oluşturduk ancak orijinal asm.js dosyaları ``bin/`* içinde kaldı. Çünkü isim çakışmalarını önlemek amacıyla yenilerinin ayrı bir dizine yerleştirilmesi gerekiyordu.
- Bir wasm veya asm.js ikili sayı sistemi dosyasını indirdiğinizden emin olmak istiyorsanız `bin/` ve `wasm/` dizinleri yerine `emscripten-asmjs/` ve `emscripten-wasm32/` dizinlerini kullanın.

- `list.js` ve `list.txt` yerine `list.json` kullanın. JSON liste formatı eskilerde bulunan tüm bilgileri ve daha fazlasını içerir.
- https://solc-bin.ethereum.org yerine https://binaries.soliditylang.org kullanın. İşleri basit tutmak için derleyiciyle ilgili neredeyse her şeyi yeni `soliditylang.org` alan adı altına taşıdık ve bu durum `solc-bin` için de geçerlidir. Yeni alan adı önerilse de, eski alan adı hala tam olarak desteklenmekte ve aynı konuma işaret etmesi garanti edilmektedir.

Uyarı: Binary dosyaları https://ethereum.github.io/solc-bin/ adresinde de mevcuttur, fakat bu sayfanın güncellenmesi 0.7.2 sürümünün yayınlanmasından hemen sonra durdurulmuştur. Aynı zamanda bu adres herhangi bir platform için yeni sürümler veya nightly yapılar almayacak ve emscripten olmayan yapılar da dahil olmak üzere yeni dizin yapısını sunmayacaktır.

Eğer hala bu adresi kullanıyorsanız, lütfen bunun yerine https://binaries.soliditylang.org adresine kullanmaya devam edin. Bu, temeldeki barındırma hizmeti(hosting) üzerinde şeffaf bir şekilde değişiklik yapmamıza ve kesintiyi en aza indirmemize olanak tanır. Herhangi bir kontrole sahip olmadığımız `ethereum.github.io` alan adının aksine, `binaries.soliditylang.org` alan adının uzun vadede aynı URL yapısını koruyacağını ve çalışacağını garanti ediyoruz.

3.2.8 Kaynağından Kurulum

Ön Koşullar - Tüm İşletim Sistemleri

Aşağıda Solidity'nin tüm geliştirmeleri için bağımlılıklar verilmiştir:

Yazılım	Notlar
CMake (sürüm 3.13+)	Platformlar arası derleme dosyası oluşturucusu.
Boost (Windows 'ta 1.77+ sürümü, aksi takdirde 1.65+)	C++ kütüphaneleri.
Git	Kaynak kodu almak için komut satırı aracı.
z3 (sürüm 4.8+, Opsiyonel)	SMT denetleyicisi ile kullanım için.
cvc4 (Opsiyonel)	SMT denetleyicisi ile kullanım için.

Not: Solidity'nin 0.5.10'dan önceki sürümleri Boost'un 1.70+ olan sürümlerine doğru bir şekilde bağlanamayabilir. Olası bir geçici çözüm, solidity'yi yapılandırmak için cmake komutunu çalıştırmadan önce `<Boost yükleme yolu>/lib/cmake/Boost-1.70.0` adını geçici olarak yeniden adlandırmaktır.

0.5.10'dan başlayarak Boost 1.70+ kadar olan sürümlerle bağlantı kurmak(linking) manuel müdahale olmadan çalışmalıdır.

Not: Varsayılan derleme yapılandırması belirli bir Z3 sürümü (kodun en son güncellendiği zamandaki en son sürüm) gerektirir. Z3 sürümleri arasında yapılan değişiklikler genellikle biraz farklı (ancak yine de geçerli olan) sonuçların döndürülmesine neden olur. SMT testlerimiz bu farklılıkları hesaba katmaz ve muhtemelen yazıldıkları sürümden farklı olan bir sürümde başarısız olacaklardır. Bu, farklı bir sürüm kullanan bir derlemenin hatalı olduğu anlamına gelmez. CMake'e `-DSTRICT_Z3_VERSION=OFF` seçeneğini iletirseniz, yukarıdaki tabloda verilen gereksinimi karşılayan herhangi bir sürümle derleme yapabilirsiniz. Ancak bunu yaparsanız, SMT testlerini atlamak için lütfen `scripts/tests.sh` dosyasına `--no-smt` seçeneğini de eklemeyi unutmayın.

Minimum Derleyici Sürümleri

Aşağıdaki C++ derleyicileri ve minimum sürümleri Solidity kod tabanını derleyebilir:

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

Ön Koşullar - macOS

macOS derlemeleri için, Xcode`un en son sürümünün yüklü olduğundan emin olun. Bu, Clang C++ derleyicisi, Xcode IDE ve OS X üzerinde C++ uygulamaları oluşturmak için gerekli olan diğer Apple geliştirme araçlarını içerir. Xcode'u ilk kez yüklüyorsanız veya yeni bir sürüm yüklediyseniz, komut satırı derlemeleri yapmadan önce lisansı kabul etmeniz gerekecektir:

```
sudo xcodebuild -license accept
```

OS X derleme betiğimiz, harici bağımlılıkları yüklemek için Homebrew paket yöneticisini kullanır. Eğer sıfırdan başlamak isterseniz, Homebrew <https://docs.brew.sh/FAQ#how-do-i-uninstall-homebrew>`_'i nasıl kaldıracağınız aşağıda açıklanmıştır.

Ön Koşullar - Windows

Solidity'nin Windows derlemeleri için aşağıdaki bağımlılıkları yüklemeniz gerekir:

Yazılım	Notlar
Visual Studio 2019 Build Tools	C++ derleyicisi
Visual Studio 2019 (Opsiyonel)	C++ derleyicisi ve geliştirme ortamı
Boost (sürüm 1.77+)	C++ kütüphaneleri.

Eğer zaten bir IDE'niz varsa ve yalnızca derleyici ve kütüphanelere ihtiyaç duyuyorsanız, Visual Studio 2019 Build Tools'u yükleyebilirsiniz.

Visual Studio 2019 hem IDE hem de gerekli derleyici ve kütüphaneleri sağlar. Dolayısıyla, bir IDE'niz yoksa ve Solidity geliştirmeyi tercih ediyorsanız, Visual Studio 2019 her şeyi kolayca kurmanız için iyi bir tercih olabilir.

Visual Studio 2019 Build Tools veya Visual Studio 2019'da yüklenmesi gereken bileşenlerin listesi aşağıda verilmiştir:

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

Gerekli tüm harici bağımlılıkları yüklemek için kullanabileceğiniz bir yardımcı betiğimiz var:

```
scripts\install_deps.ps1
```

Bu `boost` ve `cmake`'i `deps` alt dizinine yükleyecektir.

Depoyu Klonlamak

Kaynak kodunu klonlamak için aşağıdaki komutu çalıştırın:

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

Solidity'nin geliştirilmesine yardımcı olmak istiyorsanız, Solidity'yi çatallamalı(fork) ve kişisel çatalınızı(fork) ikinci bir remote olarak eklemelisiniz:

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

Not: Bu yöntem, örneğin böyle bir derleyici tarafından üretilen her bayt kodunda bir bayrağın ayarlanmasına yol açan bir ön sürüm derlemesiyle sonuçlanacaktır. Yayınlanmış bir Solidity derleyicisini yeniden derlemek istiyorsanız, lütfen github sürüm sayfasındaki kaynak tarball'u kullanın:

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

(github tarafından sağlanan "Kaynak kodu" değil).

Komut Satırı Kullanarak Derlemek

Derlemeden önce Harici Bağımlılıkları(yukarıda bulunan) yüklediğinizden emin olun.

Solidity projesi derlemeyi yapılandırmak için CMake kullanır. Tekrarlanan derlemeleri hızlandırmak için ccache yüklemek isteyebilirsiniz. CMake bunu otomatik olarak alacaktır. Solidity'yi derlemek Linux, macOS ve diğer Unix'lerde de oldukça benzerdir:

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

veya Linux ve macOS'ta daha da kolay çalıştırabilirsiniz:

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

Uyarı: BSD derlemeleri çalışmalıdır, fakat Solidity ekibi tarafından test edilmemiştir.

Ve Windows İçin:

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

Eğer `scripts\install_deps.ps1` tarafından yüklenen boost sürümünü kullanmak isterseniz, `-DBoost_DIR="deps\boost\lib\cmake\Boost-*"` ve `-DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded` seçeneklerini `cmake` çağrısına argüman olarak iletmeniz gerekecektir.

Bunun sonucunda bu yapı dizininde **solidity.sln** dosyası oluşturulmalıdır. Ayrıca bu dosyaya çift tıklandığında Visual Studio nun açılması gerekir. Biz **Yayın** yapılandırmasını oluşturmanızı öneririz, ancak diğerleri de çalışır.

Alternatif olarak, Windows için komut satırında aşağıdaki gibi bir derleme de yapabilirsiniz:

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

3.2.9 CMake Ayarları

CMake ayarlarının ne olduğunu merak ediyorsanız `cmake .. -LH` komutunu çalıştırın.

SMT Çözücüleri

Solidity, SMT çözücülerine karşı derlenebilir ve sistemde bulunurlarsa default(varsayılan) olarak bunu yapacaklardır. Her çözücü bir *cmake* seçeneği ile devre dışı bırakılabilir.

Not: Bazı durumlarda bu, derleme hataları için potansiyel olarak geçici bir çözüm de olabilir.

Yapı klasörünün içinde bunları devre dışı bırakabilirsiniz, çünkü varsayılan olarak etkin durumdadırlar:

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

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

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

3.2.10 Sürüm Dizgisi (String) Detayları

Solidity sürüm dizgisi dört bölümden oluşur:

- Sürüm numarası
- Sürüm öncesi etiketi (genellikle develop.YYYY.MM.DD veya night..YYYY.MM.DD olarak ayarlanır)
- `commit.GITHASH` biçiminde ilgili commit
- Platform ve derleyici ile ilgili ayrıntıları içeren, rasgele sayıda öğeye sahip platform

Yerel değişiklikler varsa commit'in sonuna `.mod` diye eklenir.

Tüm değişiklikler, Semver'in gerektirdiği şekilde, Solidity yayınlanma öncesi sürümün Semver yayınlanma öncesi sürümüne eşit olduğu ve Solidity'de bir işlem yapıldığında Semver'deki meta verilerinin de değiştiği bir şekilde gerçekleşir.

Bir yayın örneği: `0.4.8+commit.60cc1668.Emscripten.clang`.

Bir ön yayın örneği: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

3.2.11 Sürümleme Hakkında Önemli Bilgi

Bir sürüm yapıldıktan sonra, yama sürüm seviyesi yükseltilir, çünkü sadece yama seviyesindeki değişikliklerin takip edildiğini varsayıyoruz. Değişiklikler birleştirildiğinde (merge) , SemVer'e ve değişikliğin ciddiyetine göre sürüm yükseltilmelidir. Son olarak, bir sürüm her zaman mevcut nightly derlemenin sürümüyle, ancak `prerelease` belirteci olmadan yapılır.

Örnek:

1. 0.4.0 sürümü çıktı.
2. Nightly yapı şu andan itibaren 0.4.1 sürümüne sahiptir.
3. İşleyişi bozmayan değişikliler tanıtıldı –> sürümde değişiklik yok.
4. İşleyişi bozan değişiklikler tanıtıldı –> version 0.5.0'a yükseltildi.
5. 0.5.0 sürümü çıktı.

Bu davranış *version pragma* ile iyi çalışır.

3.3 Solidity by Example

3.3.1 Oylama

Birazdan göreceğiniz sözleşme biraz karışık ancak Solidity'nin bir çok özelliğini görebilirsiniz. Göreceğiniz kontrat bir oylama kontratıdır. Elektronik oy kullanmada asıl problem oy hakkının doğru kişilere nasıl verildiği ve manipülasyonun nasıl engelleneceğidir. Bütün problemleri burada çözmeyeceğiz ama en azından yetkilendirilmiş kişilerle hem otomatik hem de şeffaf olarak nasıl oylama yapılacağını göstereceğiz.

Fikrimiz oy sandığı başına bir kontrat oluşturup her seçenek için kısa isimler vermek. Sonrasında kontratın yaratıcısı, aynı zamanda seçim başkanı oluyor, her cüzdana tek tek oy hakkı verecek.

Sonrasında cüzdan sahipleri kendilerine ya da güvendikleri bir kişiye oy verebilirler.

Oylamanın süresi dolduğunda, `winningProposal()` en yüksek oyu almış teklifi geri döndürecek.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Yetkili Oylama
contract Ballot {
    // Bu sonrasında değişken olarak
    // kullanılmak için oluşturulmuş kompleks bir tür
    // Tek bir seçmeni temsil eder
    struct Voter {
        uint weight; // oyun seçimdeki etki ağırlığı
        bool voted;  // true ise oy kullanılmıştır
        address delegate; // yetkilendirilecek kişinin adresi
        uint vote;   // oy verilmiş proposalın index numarası
    }

    // Tekli teklif türü
    struct Proposal {
        bytes32 name;   // kısa ismi (32 bayta kadar)
        uint voteCount; // toplam oy miktarı
    }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    address public chairperson; // seçim başkanının adresi

    // Her adres için `Voter` (Oy kullanan kişi)
    // structına mapping (eşleştirme) değişkeni
    mapping(address => Voter) public voters;

    // `Proposal` structlarından oluşan bir dinamik dizi (dynamic array).
    Proposal[] public proposals;

    /// `proposalNames`lerden birini seçmek için bir oy sandığı oluşturur.
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // Her teklif ismi için bir teklif objesi oluşturup
        // dizinin (array) sonuna ekle
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` geçici bir Proposal (Teklif)
            // objesi oluşturur ve `proposals.push(...)`
            // objeyi `proposals` dizisinin sonuna ekler.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // `voter`a bu sandıkta oy kullanma yetkisi ver.
    // `chairperson` bu fonksiyonu çağırabilir.
    function giveRightToVote(address voter) external {
        // Eğer `require`ın ilk argümanı `false`
        // gelirse işlem iptal olur ve Ether
        // harcamaları eski haline gelir
        // Eskiden bu durumda bütün gas harcanırdı
        // ama artık harcanmıyor.
        // Çoğu zaman fonksiyonun doğru çağrılıp
        // çağrılmadığını anlamak için `require`
        // kullanılırsa iyi olur
        // İkinci argüman olarak neyin hatalı olduğunu
        // açıklayan bir yazı girilebilir.
        require(
            msg.sender == chairperson,
            "Sadece chairperson yetki verebilir."
        );
        require(
            !voters[voter].voted,
            "Kişi zaten oy kullandı."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    /// Delege `to` ata.
    function delegate(address to) external {
        // referans atar
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Oy verme yetkin yok");
        require(!sender.voted, "Zaten oy kullandın.");

        require(to != msg.sender, "Kendini temsilci gösteremezsin.");

        // Delege atamasını `to` da delege atandıysa
        // aktarır
        // Genelde bu tür döngüler oldukça tehlikelidir,
        // çünkü eğer çok fazla çalışırlarsa bloktaki
        // kullanılabilir gas'ten daha fazlasına ihtiyaç duyabilir.
        // Bu durumda, delege atama çalışmayacaktır
        // ama başka durumlarda bu tür döngüler
        // kontratın tamamiyle kitlenmesine sebep olabilir.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // Delege atamada bir döngü bulduk, bunu istemiyoruz
            require(to != msg.sender, "Delege atamada döngü bulundu.");
        }

        Voter storage delegate_ = voters[to];

        // Oy kullanan kişi oy kullanamayan kişileri delege gösteremez.
        require(delegate_.weight >= 1);

        // `sender` bir referans olduğundan
        // `voters[msg.sender]` değişir.
        sender.voted = true;
        sender.delegate = to;

        if (delegate_.voted) {
            // Eğer delege zaten oylandıysa
            // otomatik olarak oylara eklenir
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // Eğer delege oylanmadıysa
            // ağırlığına eklenir.
            delegate_.weight += sender.weight;
        }
    }

    /// Oy kullan (sana atanmış oylar da dahil)
    /// teklif ismine `proposals[proposal].name`.
    function vote(uint proposal) external {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Oy kullanma yetkisi yok");
        require(!sender.voted, "Already voted.");
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        sender.voted = true;
        sender.vote = proposal;

        // Eğer `proposal` dizinin içinde yoksa,
        // otomatik olarak bütün değişiklikler
        // eski haline döner
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Bütün oyları hesaplayarak kazanan
    /// teklifi hesaplar
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Kazanan teklifin indeks numarasını bulmak için
    // winningProposal() fonksiyonunu çağırır ardından
    // kazanan teklifin adını döndürür.
    function winnerName() external view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

Olası İyileştirmeler

Şu an tüm katılımcılara yetki vermek için çok sayıda işlem gerçekleştirilmesi gerekiyor. Daha iyi bir yöntem düşünebiliyor musunuz?

3.3.2 Gizli İhale

Bu bölümde Ethereum'da tamamiyle gizli bir ihale kontratı oluşturmanın ne kadar kolay olduğunu göstereceğiz. Önce herkesin başkalarının tekliflerini görebildiği açık bir ihale kontratı oluşturup sonrasında ondan teklif süresi dolana kadar kimsenin başkasının teklifini göremediği gizli bir ihale kontratı oluşturacağız.

Basit Açık İhale

Aşağıdaki basit ihale kontratındaki fikir herkes teklif sürecinde teklfilerini gönderebilcek. Teklifler yanında teklif verenlerin tekliflerine sadık kalmaları için teklifte belirtilen parayı da içerecek. Eğer en yüksek teklif geçilirse önceki en yüksek teklifi veren kişi parasını geri alacak. Teklif süreci bittiğinde kontratlar kendi kendilerine çalışamadıklarından hak sahibi parasını almak için kontratı manuel olarak çağırılmalıdır.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract SimpleAuction {
    // İhalenin parametreleri. Süreler unix zaman damgası
    // (1970-01-01'den itibaren saniyeler) ya da saniye
    // cinsinden ne kadar süreceği.
    address payable public beneficiary;
    uint public auctionEndTime;

    // İhalenin şu an ki durumu
    address public highestBidder;
    uint public highestBid;

    // Önceki tekliflerden para çekmeye izin verilenler
    mapping(address => uint) pendingReturns;

    // En son `true`ya çevir, herhangi bir değişiklik yapılmasını engeller
    // varsayılan olarak `false` tanımlanır.
    bool ended;

    // Değişikliklerde yayınlanacak Event'ler
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Başarısızları açıklayan hatalar

    // Üçlü eğik çizgiler natspec yorumları olarak
    // adlandırılır. Kullanıcıya bir işlemi onaylayacağı
    // zaman ya da bir hatada gösterilir.

    /// İhale bitti.
    error AuctionAlreadyEnded();
    /// Eşit ya da daha yüksek bir teklif var.
    error BidNotHighEnough(uint highestBid);
    /// Teklif henüz bitmedi.
    error AuctionNotYetEnded();
    /// auctionEnd fonksiyonu zaten çağrıldı.
    error AuctionEndAlreadyCalled();
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    /// Hak sahibi adına `beneficiaryAddress`
    /// `biddingTime` daki süre ile bir ihale başlatır.
    constructor(
        uint biddingTime,
        address payable beneficiaryAddress
    ) {
        beneficiary = beneficiaryAddress;
        auctionEndTime = block.timestamp + biddingTime;
    }

    /// İşlemle birlikte teklifteki para da
    /// gönderilir. Ödeme sadece teklif kazanmazsa
    /// iade edilir.
    function bid() external payable {
        // Herhangi bir argümana gerek yok,
        // bütün bilgi zaten işlemin parçası.
        // payable anahtar kelimesi fonksiyonun
        // Ether alabilmesi için zorunlu.

        // teklif süreci bittiyse çağrıyı
        // geri çevir.
        if (block.timestamp > auctionEndTime)
            revert AuctionAlreadyEnded();
        // Teklif daha yüksek değilse,
        // parayı geri gönderin (revert ifadesi
        // parayı almış olması da dahil olmak
        // üzere bu fonksiyon yürütmesindeki tüm
        // değişiklikleri geri alacaktır).
        if (msg.value <= highestBid)
            revert BidNotHighEnough(highestBid);

        if (highestBid != 0) {
            // Basit bir şekilde highestBidder.send(highestBid)'i
            // kullanarak para göndermek bir güvenlik riski oluşturuyor
            // çünkü güvenilmez bir kontratı (içinde fallback fonksiyonu
            // içeren) çalıştırabilir. Her zaman katılımcıların paralarını
            // kendilerinin çekmeleri daha güvenilirdir.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Geçilmiş bir teklifin parasını geri çek.
    function withdraw() external returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // Bu değeri sıfıra eşitlemek önemli çünkü alıcı bu fonksiyonu
            // `send` tamamlanmadan tekrar çağırırsa (reentrancy) alması gerekenden
            // daha fazla para çekebilir.
            pendingReturns[msg.sender] = 0;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            // msg.sender `address payable` türünde değil ve `send()`
            // fonksiyonunda çağrılabilmesi `payable(msg.sender)` ile
            // `address payable` a dönüştürülmesi gerekiyor.
            if (!payable(msg.sender).send(amount)) {
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// İhaleyi bitir ve en yüksek teklifi
    /// hak sahibine gönder.
    function auctionEnd() external {
        // Diğer kontratlar ile etkileşime giren (fonksiyon çağıran ya da
        // Ether gönderen) fonksiyonları üç parçada şekilldenirmek güzel bir yöntem.
        // Şu parçalar
        // 1. koşul kontrolleri
        // 2. eylem gerçekleştirenler (koşulları değiştirebilirler)
        // 3. başka kontratlarlar etkileşime girenler
        // Eüer bu fazlar karışırsa, diğer kontrat bu kontratı çağırıp
        // durumları değiştirebilir ya da olayların (ether ödemesi gibi)
        // birkaç kere gerçekleşmesine sebep olabilir.
        // Eğer içeriden çağırılan fonksiyonlar başka kontratlarla etkileşime
        // giriyorsa o fonksiyonlar da başka fonksiyonlarla etkileşenler olarak
        // değerlendirilmeli

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

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

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

Gizli İhale

Aşağıda yukarıdaki açık ihalenin kapalı ihaleye dönüştürülmüş halini bulabilirsiniz. Gizli ihalenin avantajı ihale sürecinin sonunda doğru bir zaman baskısı oluşturmaması. Saydam bir işlem platformunda gizli ihale oluşturmak çelişkili olsa da kriptografi burada yardımımıza koşuyor.

Teklif süreci boyunca, teklif veren kişi aslında gerçekten teklif yapmıyor, sadece hashlenmiş bir halini gönderiyor. Şu an hash değerleri eşit olan iki değer (yeterince uzun) bulmak pratik olarak imkansız olduğundan, teklif veren kişi bu şekilde teklif oluşturmuş olur. Teklif süreci bittikten sonra teklif veren kişiler tekliflerini açıklamalı, girdikleri şifrelenmemiş değerin hashlenmiş hali ile önceden girdikleri hash ile aynı olmalıdır.

Başka bir zorluk da **gizlilik ve bağlayıcılığı** aynı anda sağlamak. Teklif veren kişinin kazandıktan sonra teklifinden vazgeçmemesinin tek yolu teklif ile birlikte parayı da yollaması ancak transferler Ethereum'da gizlenemediğinden herhangi bir kişi miktarı görebilir.

Aşağıdaki kotnrat bu sorunu teklif ile birlikte herhangi bir miktar paranın birlikte gönderilmesiyle çözüyor. Miktar ile teklifin eşitliği sadece açıklama fazında ortaya anlaşılabildiği için bazı teklifleri **geçersiz** olabilir, ve teklif veren kişiler bunu kasıtlı olarak kullanabilir (hatta bu durum daha fazla gizlilik sağlıyor) Teklif veren kişiler kasıtlı olarak bir kaç yüksek ve düşük geçersiz teklifler oluşturarak kafa karıştırabilirler.

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

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

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

    address public highestBidder;
    uint public highestBid;

    // Önceki tekliflerden para çekmeye izin verilenler
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    // Başarısızları açıklayan hatalar

    /// Fonksiyon erken çağırıldı.
    /// `time` de tekrar deneyin.
    error TooEarly(uint time);
    /// Fonksyion geç çağırıldı.
    /// `time` dan sonra çağırılamaz.
    error TooLate(uint time);
    /// auctionEnd fonksyionu zaten çağırıldı.
    error AuctionEndAlreadyCalled();

    // Modifierlar fonksiyon girdilerini kontrol etmenin
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    // kolay bir yöntemidir. `onlyBefore` modifierı aşağıdaki
    // `bid` e uygulandı:
    // Yeni fonksyionun gövde kısmı modifierın gövde kısmı oluyor.
    // Sadece `_` eski fonksiyonun gövdesiyle değişiyor..
    modifier onlyBefore(uint time) {
        if (block.timestamp >= time) revert TooLate(time);
        _;
    }
    modifier onlyAfter(uint time) {
        if (block.timestamp <= time) revert TooEarly(time);
        _;
    }

    constructor(
        uint biddingTime,
        uint revealTime,
        address payable beneficiaryAddress
    ) {
        beneficiary = beneficiaryAddress;
        biddingEnd = block.timestamp + biddingTime;
        revealEnd = biddingEnd + revealTime;
    }
    /// `blindedBid` = keccak256(abi.encodePacked(value, fake, secret))
    /// ile gizli bir teklif ver. Gönderilen ether sadece teklif doğru
    /// bir şekilde açıklandıysa geri alınabilir. Teklif eğer "value"daki
    /// değer ile en az gönderilen Ether kadar ya da "fake" değeri `false`
    /// ise geçerlidir.  Bir miktar Ether yatırılması  gereksenede
    /// "fake" değerini `true` yapmak ve "value" değerinden
    /// farklı miktarda Ether göndermek gerçek teklifi gizlemenin yöntemleridir.
    /// Aynı adres birden fazla kez para yatırabilir.
    function bid(bytes32 blindedBid)
        external
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: blindedBid,
            deposit: msg.value
        }));
    }

    /// Gizli teklifini açıkla. Tüm teklifler arasında en yüksek olan hariç
    /// doğru şekilde açıklanmış tüm tekliflerin parasını iade alabilirsin.
    function reveal(
        uint[] calldata values,
        bool[] calldata fakes,
        bytes32[] calldata secrets
    )
        external
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (values[i], fakes[i], secrets[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,␣
↪secret))) {
                // Teklif açıklanmadı
                // Yatırılan parayı iade etme
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Göndericinin gönderdiği parayı tekrar geri almasını
            // imkansız hale getir.
            bidToCheck.blindedBid = bytes32(0);
        }
        payable(msg.sender).transfer(refund);
    }

    /// Fazladan para yatırılmış bir teklifi geri çek.
    function withdraw() external {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // Bu değeri sıfıra eşitlemek önemli çünkü alıcı bu fonksiyonu
            // `send` tamamlanmadan tekrar çağırırsa (reentrancy) alması gerekenden
            // daha fazla para çekebilir. (yukarıdaki şartlar -> etkiler -> etkileşimler
            // hakkındaki bilgilendirmeye bakabilirsiniz)
            pendingReturns[msg.sender] = 0;

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

    /// İhaleyi bitir ve en yüsek teklifi
    /// hak sahibine gönder.
    function auctionEnd()
        external
        onlyAfter(revealEnd)
    {
        if (ended) revert AuctionEndAlreadyCalled();
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }

    // "internal" (içsel) bir fonksiyon yani sadece kontratın
    // kendisi (ya da bu kontrattan çıkan (derive edilen) kontratlar)
    // bunu çağırabilir.
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Önceki en yüksek teklifin parasını iade et.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }
}
```

3.3.3 Güvenli Uzaktan Alışveriş

Uzaktan bir mal satın almak birden fazla tarafın birbirine güvenmesini gerektirir. En basit durumda bir satıcı bir de alıcı olur. Alıcı ürünü satıcıdan almak ister, satıcı da karılığında parayı (ya da eş değeri bir şeyi) almak. Burada problemli kısım kargolama: Kesin olarak malın alıcıya ulaştığından emin olmanın yolu yok.

Bu problemmi çözmenin birden fazla yolu var ama hepsinin bir şekilde bir eksiği oluyor. Aşağıdaki örnekte, iki taraf da kontrata malın değerinin iki katını yatırırlar. Yatırma gerçekleştiği anda alıcı onaylayana kadar iki tafaında parası içeride kitli kalır. Alıcı satın almayı onayladığında malın değeri (yatırdığının yarısı) karşı tarafa geçer ve satıcı malın üç katını (yatırdığı iki kat ve alıcının yatırdığı malın değeri) geri çeker. Bu sistemin arkaplanındaki fikir iki tarafında problemi çözmeleri için gönüllü olmaları yoksa ikisinin parası da içeride sonsuza kadar kitli kalacak

Bu kontrat tabi ki bu problemi çözmüyor ama makine benzeri yapıları sözleşmede nasıl kullanabileceğinize dair genel bir bakış sağlıyor.

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

    enum State { Created, Locked, Release, Inactive }
    // state değişkeni varsayılan olarak ilk üyedir,  `State.created`
    State public state;

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    /// Bu fonksiyonu sadece alıcı çağırabilir
    error OnlyBuyer();
    /// BU fonksyionu sadece satıcı çağırabilir.
    error OnlySeller();
    /// Bu fonksiyon şu an çağırılamaz.
    error InvalidState();
    /// Girilen değer çift olmalı.
    error ValueNotEven();

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

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

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

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

    // `msg.value` in çift olduğundan emin ol.
    // Eğer tek sayı ise bölme kırpılmış bir sonuç olacak.
    // Çarpma ile tek sayı olmadığını kontrol et.
    constructor() payable {
        seller = payable(msg.sender);
        value = msg.value / 2;
        if ((2 * value) != msg.value)
            revert ValueNotEven();
    }

    /// Satın almayı iptal et ve etheri geri al.
    /// Sadece satıcı tarafından kontrat kitlenmeden
    /// önce çağırılabilir.
    function abort()
        external
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        // Burada transfer'i direkt olarak kullanıyoruz.
        // Tekrar giriş (reentrancy) saldırılarına karşı güvenli
        // çünkü fonksiyondaki son çağrı (call) ve durumu (state)
        // zaten değiştirdik.
        seller.transfer(address(this).balance);
    }

    /// Alıcı olarak satın almayı onayla.
    /// İşlem `2 * value` kadar ether içermeli.
    /// Ether confirmReceived fonksiyonu çağırılana
    /// kadar kitli kalacak.
    function confirmPurchase()
        external
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = payable(msg.sender);
        state = State.Locked;
    }

    /// Malı teslim aldığını onayla (alıcı)
    /// Kitli etheri serbest bırakacak.
    function confirmReceived()
        external
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // Durumu (state) önceden değiştirmek oldukça önemli
        // yoksa aşağıdaki `send` i kontratlar burada tekrar
        // bu fonksiyonu çağırabilir. (tekrar giriş saldırısı - reentrancy attack)
        state = State.Release;

        buyer.transfer(value);
    }

    /// Bu fonksiyon satıcıya iade eder
    /// (satıcının kitli parasını geri öder)
    function refundSeller()
        external
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // Durumu (state) önceden değiştirmek oldukça önemli
        // yoksa aşağıdaki `send` i kontratlar burada tekrar
        // bu fonksiyonu çağırabilir. (tekrar giriş saldırısı - reentrancy attack)
        state = State.Inactive;

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }
}
```

3.3.4 Mikro Ödeme Kanalı

Bu bölümde bir ödeme kanalı örneğinin nasıl yapılacağını öğreneceğiz. Bu sistem belli kişiler arasındaki Ether transferini güvenli, anında ve işlem masrafsız gerçekleştirmek için kriptografik imzaları kullanacak. Bu örnek için, imzaların nasıl imzalandığını ve doğrulandığını anlamamız gerekiyor.

İmza Oluşturma ve Doğrulama

Mesela Alice Bob'a bir miktar Ether göndermek istiyor. Başka bir deyişle Alice gönderici Bob ise alıcı.

Alice'in Bob'a, çek yazmaya benzer bir şekilde, sadece kriptografik olarak imzalanmış off-chain (zincir dışı) bir mesaj göndermesi yeterli.

Alice ve Bob bu imzaları ödemeleri yetkilendirmek için kullanabilirler ve bu Ethereum üzerindeki akıllı kontratlar ile mümkün olan bir şey. Alice Ethere göndermek için basit bir akıllı kontrat yazacak ancak bu fonksiyonu kendi çağırmak yerine Bob'un çağırmasını isteyecek böylece Bob işlem masraflarını da ödemiş olacak.

Kontrat aşağıdaki şekilde ilerleyecek.

1. Alice içine ödeme için gerekli Ether'i de ekleyerek `ReceiverPays` kontratını yayınlayacak.
2. Alice ödemeyi gizli anahtarı ile imzalayarak yetkilendirecek.
3. Alice kriptografik olarak imzalanmış mesajı Bob'a gönderecek. Mesajın gizli tutulmasına (daha sonra açıklanacak) gerek yok ve mesaj herhangi bir yöntem ile gönderilebilir.
4. Bob imzalanmış mesajı akıllı kontrata girerek ödemesini alabilir, akullı kontrat mesajın gerçekliğini doğrular ve ödemeyi serbest bırakır.

İmza oluşturma

Alice'in ödemeyi imzlamak için Ethereum ağı ile etkileşime girmesine gerek yok, bu işlem tamamiyle çevrimdışı gerçekleştirilebilir. Bu eğitimde, mesajları tarayıcıda web3.js ve MetaMask kullanarak ve getirdiği güvenlik kazançları için EIP-712 gösterilen metot ile imzalayacağız.

```
/// En başta "Hash"leme işleri daha kolay bir hale getirir
var hash = web3.utils.sha3("imzalanacak mesaj");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log(
↪"İmzalandı"); });
```

Not: `web3.eth.personal.sign` mesajın uzunluğunu imzalanmış bilginin başına ekler. İlk olarak "hash"lediğimiz için, mesaj her zaman 32 bayt uzunluğunda olacak ve dolayısıyla bu uzunluk ön eki her zaman aynı olacak.

Ne İmzalanacak

Ödeme gerçekleştiren bir kontrat için, imzalanmış bir mesaj aşağıdakiler içermeli:

1. Alıcının adresi.
2. Transfer edilecek miktar.
3. Tekrarlama saldırılarına karşı önlem

Tekrarlama saldırısı, imzalanmış bir mesajın tekrar yetkilendirme için kullanılmasıdır. Tekrarlama saldırılarını önlemek için Ethereum işlemlerinden kullanan bir cüzdandan yapılan işlem sayısını, nonce, kullanan tekniği kullanacğız. Akıllı kontrat bir *nonce* un bir kaç kez kullanılıp kullanılmadığını kontrol edecek.

Başka bir tekrarlamma saldırısı açığı ödemeyi gönderen kişi `ReceiverPays` akıllı kontratını yayınlayıp sonrasında yok edip sonra tekrar yayınladığında oluşur. Bunun sebebi tekrar yayınlanan kontrat önceki kontratta kullanılan *nonce* ları bilemediğinden saldırgan eski mesajları tekrar kullanabilir.

Alice buna karşı korunmak için kontratın adresini de mesajın içerisine ekleyebilir. Böylece sadece kontrat'ın adresini içeren mesajlar onaylanır. Bu örneği bu bölümün sonundaki tam kontratın `claimPayment()` fonksiyonundaki ilk iki satırda görebilirsiniz.

Argümanları Paketleme

Şimdi imzalanmış mesajımızda nelerin olacağına karar verdiğimize göre mesajı mesajı oluşturup, hashleyip, imzalamaya hazırız. Basit olsun diye verileri art arda bağlayacağız. ethereumjs-abi kütüphanesi bize `soliditySHA3` adında Solidity'deki `abi.encodePacked` ile enkode edilmiş argümanlara `keccak256` fonksiyonu uygulanması ile aynı işlevi gören bir fonksiyon sağlıyor. Aşağıda `ReceiverPays` için düzgün bir imza sağlayan JavaScript fonksiyonunu görebilirsiniz.

```
// recipient alıcı adres,
// amount, wei cinsinden, ne kadar gönderilmesi gerektiği
// nonce, tekrarlama saldırılarını önlemek için eşsiz bir sayı
// contractAddress, kontratlar arası tekrarlama saldırısını engellemek için kontrat␣
↪adresi
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + abi.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

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

Solidity'de İmzalayanı Bulma

Genelde ECDSA imzaları iki parametreden oluşur, `r` ve `s`. Ethereum'daki imzalar `v` denilen üçüncü bir parametre daha içerir. `v` parametresi ile mesajı imzalamak için kullanılmış cüzdanın gizli anahtarı doğrulanabiliyirsiniz. Solidity *ecrecover* fonksiyonunu gömülü olarak sağlamaktadır. Bu fonksiyon mesajla birlikte `r`, `s` ve `v` parametrelerini de alır ve mesajı imzalamak için kullanılmış adresi verir.

İmza Parametrelerini Çıkartma

web3.js ile oluşturulmuş imzalar `r`, `s` ve `v`'in birleştirilmesi ile oluşturulur, yani ilk adım bu parametreleri ayırmak. Bunu kullanıcı tarafında da yapabilirsiniz ancak parametre ayırma işleminin akıllı kontratın içinde olması akıllı kontrata üç parametre yerine sadece bir parametre göndermemizi sağlar. Bir bayt dizisini (byte array) bileşenlerine ayırmak biraz karışık dolayısıyla bu işlemi `splitSignature` fonksiyonunda yapmak için *inline assembly* kullanacağız. (Bu bölümün sonundaki tam kontrattaki üçüncü fonksiyon.)

Mesaj Hashini Hesaplama

Akıllı kontratın tam olarak hangi parametrelerin izalandığını bilmesi gerekiyor çünkü kontratın imzzayı doğrulamak için mesajı parametrelerinden tekrar oluşturması lazım. `claimPayment` fonksiyonundaki `prefixed` ve `recoverSigner` fonksiyonları bu işlemi gerçekleştiriyor.

Tam Kontrat

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

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

        // istemcide imzalanmış mesajı tekrar oluşturur.
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce,␣
↪this)));

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

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

    /// sözleşmeyi yok eder ve kalan parayı geri alır
    function shutdown() external {
        require(msg.sender == owner);
        selfdestruct(payable(msg.sender));
    }

    /// imza methodları
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

        assembly {
            // uzunluk önekinden sonraki ilk 32 bayt.
            r := mload(add(sig, 32))
            // ikinci 32 bayt
            s := mload(add(sig, 64))
            // son bayt (gelecek 32 baytın son baytı)
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

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

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

    /// eth_sign'i kopyalayan önüne eklenmiş hash oluşturur.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Basit Bir Ödeme Kanalı Yazmak

Alice şimdi ödeme basit ama tam işlevsel bir ödeme kanalı oluşturacak. Ödeme kanalları anında ve masrafsız tekrarlayan Ether transferleri gerçekleştirmek için kriptografik imzaları kullanırlar.

Ödeme Kanalı Nedir?

Ödeme kanalları katılımcıların herhangi bir işlem gerçekleştirmeden tekrarlayan Ether transferleri gerçekleştirmelerini sağlar. Bu sayesede ödemeyle ilgili gecikme ve masraflardan kurtulabilirsiniz. Şimdi iki kişi (Alice ve Bob) arasında tek yönlü bir ödeme kanalı nasıl oluşturul onu göreceğiz. Böyle bir sistemi 3 adımda oluşturabiliriz. Bunlar:

1. Alice ödeme kanalına Ether yükler böylece ödeme kanali "açık" hale gelir.
2. Alice ne kadar Ether'in ödenmesi gerektiğini bir mesajda belirtir. Bu adım her ödemede tekrar gerçekleştirilir.
3. Bob Ether ödemesini alıp kalanı geri göndererek ödeme kanalını kapatır.

Not: Sadece 1. ve 3. adımlar Ethereum işlemi gerektiriyor. 2. adımda gönderici kriptografik olarak imzalanmış mesajı alıcıya zincir dışı (off-chain) bir şekilde (mesela e-posta) gönderebilir. Kısaca herhangi bir sayıda transfer için 2 Ethereum işlemi gerekiyor.

Bob kesinlikle parasını alacak çünkü Ether bir akıllı kontratta tutuluyor ve geçerli bir imzalı mesaj ile akıllı kontratlar her zaman işlemi gerçekleştirir. Akıllı kontrat ayrıca zaman aşımını da zorunlu tutar, yani alıcı parası almazsa Alice eninde sonunda parasını geri alabilir. Zaman aşımının süresine katılımcılar kendi karar verir. İnternet kafedeki kullanım süresi gibi kısa süreli bir işlem için, ödeme kanalı süreli bir şekilde oluşturulabilir. Diğer bir yandan, bir çalışana saatlik maaşını ödemek gibi tekrarlayan bir ödeme için ödeme kanalı bir kaç ay ya da yıl açık kalabilir.

Ödeme Kanalını Açma

Ödeme kanalını açmak için Alice içine gerekli Ether'i ekleyip ve alıcının kim olduğunu girerek akıllı kontratı yayınlar. Bu işlemi bölümün sonundaki kontratta `SimplePaymentChannel` fonksiyonu gerçekleştirir.

Ödeme Gerçekleştirme

Alice ödemeyi Bob'a imzalanmış mesajı göndererek yapar. Bu adım tamammiyle Etherum ağının dışında gerçekeleşir. Mesaj gönderici tarafında kriptografik olarak imzalanır ve direkt olarak alıcıya gönderilir.

Her mesaj aşağıdaki bilgileri içerir:

- Akıllı kontratın adresi, kontratlar arası tekrarlama saldırılarını önlemek için.
- Alıcıya borçlu olunan Ether miktarı.

Ödeme kanalı bütün transferler gerçekleştikten sonra sadece bir kez kapanır. Bundan dolayı sadece bir mesajın ödemesi gerçekleşir. Bu yüzden her mesaj küçük ödemeler yerine toplam gönderilmesi gereken Ether miktarını içerir. Alıcı doğal olarak en yüksek miktarı alabilmek için en güncel mesajın ödemesini alır. Artık akıllı kontrat sadece bir mesaj okuduğunderstan artık işlem sayısını (nonce) mesaja eklemeye gerek yok ancak akıllı kontratın adresine mesajın başka bir ödeme kanalında kullanılmaması için hala ihtiyaç var.

Aşağıda önceki bölümdeki mesajın kriptografik imzalanmasını sağlayan JavaScript kodunun düzenlenmiş bir halini bulabilirsiniz.

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.solidity.SHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

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

// contractAddress, kontratlar arası tekrarlama saldırısını engellemek için kontrat␣
↪adresi
// amount, wei cinsinden, ne kadar gönderilmesi gerektiği

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

Ödeme Kanalını Kapatma

Bob ödemesini almaya hazır olduğunda ödeme kanalını da `close` fonksiyonunu çağırarak kapatmanın vakti de gelmiş demektir. Kanal kapatıldığında alıcı kendine borçlu olunan Ether miktarını alır ve kalan miktarı Alice'e geri göndererek kontratı yok eder. Bob sadece Alice tarafında imzalanmış bir mesaj ile kanalı kapatabilir.

Akıllı kontratın göndericiden gelen geçerli bir mesajı doğrulaması gerekir. Bu doğrulama süreci alıcının kullandığı süreç ile aynıdır. Solidity fonksiyonlarından `isValidSignature` ve `recoverSigner` (`ReceiverPays` kontratından aldık) önceki bölümdeki JavaScript hallerindekiyle aynı şekilde çalışır.

Sadece ödeme kanalının alıcısı `close` fonksiyonunu çağırabilir. Alıcı da doğal olarak en yüksek miktarı taşığı için en güncel mesajı gönderir. Eğer gönderici bu mesajı çağırabiliyor olsaydı daha düşük bir miktar içeren bir mesaj ile çağırıp, ödemeleri gerekenden daha düşük bir para göndererek hile yapabilirlerdi.

Fonksiyon verilen parametreler ile imzalanmış mesajı doğrular. Eğer her şey uygunsa, alıcıya kendi payına düşen Ether miktarı gönderilir ve göndericiye kalan miktar `selfdestruct` ile gönderilir. Tam kontratta `close` fonksiyonunu görebilirsiniz.

Kanalın Zaman Aşımına Uğraması

Bob istediği zaman ödeme kanalını kapatabilir anca kapatmazsa Alice'in bir şekilde parasını geri alması gerekiyor. Bunun için kontrata bir *zaman aşımı* süresi girilir. Süre dolduğunda, Alice `claimTimeout` fonksiyonunu çağırarak içerideki parasını geri alabilir. `claimTimeout` fonksyionunu tam kontratta görebilirsiniz.

Bu fonksiyon çağırıldıktan sonra Bob artık sistemden Ether alamaz dolayısıyla Bob'un zaman aşımına uğramadan parasını alması oldukça önemli.

Tam Kontrat

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract SimplePaymentChannel {
    address payable public sender;      // göndericinin adresi.
    address payable public recipient;   // alıcının adresi.
    uint256 public expiration;  // kapanmaması durumunda zaman aşımı süresi.

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

    /// alıcı, göndericinin imzalı mesajı ile istediği zaman kanalı kapatabilir.
    /// alıcı alacaklısı olduğu miktarı alıp
    /// kalanı göndericiye geri gönderir.
    function close(uint256 amount, bytes memory signature) external {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }

    /// gönderici zaman aşımı süresini istediği zaman arttırabilir
    function extend(uint256 newExpiration) external {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// Eğer süre alıcı kanalı kapatmadan dolarsa
    /// Ether göndericiye geri döner
    function claimTimeout() external {
        require(block.timestamp >= expiration);
        selfdestruct(sender);
    }

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

        // imzanın göndericiden geldiğini kontrol et
        return recoverSigner(message, signature) == sender;
    }

    /// Aşağıdaki tüm konksyionlar 'imza oluşturma ve doğrulama'
    /// bölümünden alındı.

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

        assembly {
            // uzunluk önekinden sonraki ilk 32 bayt.
            r := mload(add(sig, 32))
            // ikinci 32 bayt
            s := mload(add(sig, 64))
            // son bayt (gelecek 32 baytın son baytı)
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

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

    /// eth_sign'i kopyalayan önüne eklenmiş hash oluşturur.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Not: `splitSignature` fonksiyonu bütün güvenlik önlemlerini almıyor. Gerçek bir uygulamada openzeppelin'in versionu gibi daha iyi test edilmiş bir kütüphane kullanılmalı.

Ödemeleri Doğrulama

Önceki bölümlerdekinin aksine, ödeme kanalındaki mesajlar anında alınmamakta. Alıcı mesajların takibini yapıp zamanı geldiğinde ödeme kanalını kapatır. Yani bu durumda alıcının mesajları kendisinin doğrulaması oldukça önemli. Yoksa alıcının ödemesini kesin alacağının bir garantisi yok.

Alıcı her mesajı aşağıdaki işlemler ile doğrulamalı:

1. Mesajdaki kontrat adresinin ödeme kanalı ile aynı olduğunu kontrol et
2. Yeni toplam miktarın beklenen miktar ile aynı olduğunu kontrol et
3. Yeni toplam miktarın kontrattakinden fazla olmadığını kontrol et
4. Mesajın ödeme kanalının göndericisinden geldiğini kontrol et.

Bu doğrulamayı yazmak için ethereumjs-util kütüphanesini kullanacağız. Son adım için bir çok farklı yol var ve biz JavaScript kullanacağuz. Aşağıdaki kod `constructPaymentMessage` fonksiyonunu yukarıdaki imzalama **JavaScript kodundan** ödünç alıyor:

```
// Bu eth_sign JSON-RPC metodunun ön ekleme özelliğini taklit eder.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

3.3.5 Modüler Kontratlar

Kontratları oluştururken modüler bir yaklaşım izlemek kodların karışıklığını azaltıp, okunabilirliğini arttırır. Bu durumda hataların ve açıkların daha kolay bir şekilde bulunmasını sağlar. Eğer her modülün nasıl davranacağını izole bir şekilde tanımlar ve kontrol ederseniz, sadece bütün kontratta olup biten yerine o kontratlar arasındaki ilişkileri inceleyebilirsiniz. Aşağıdaki örnekte kontrat adresler arasında gönderilenin beklenen şekilde olup olmadığını görmek için `Balances` *library* kütüphanesinin `move` metodunu kullanır. `Balances` kütüphanesinin asla nefatif bir bakiye çıkarmadığı ya da bütün bakiyelerin toplamından overflow yaratmayacağı kolaylıkla doğrulanabilir ve bu durum kontratın yaşam süresi boyunca değişmez.

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

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

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

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

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

    }

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        return true;
    }

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

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

3.4 Solidity Kaynak Dosyasının Düzeni

Kaynak dosyalar, istenilen sayıda *contract definitions*, *import* ,:ref:*pragma<pragma>* ve *using for* yönergeleri ile *struct*,:ref:*enum<enums>*, *function*, *error* ve *constant variable* tanımları içerebilir.

3.4.1 SPDX Lisans Tanımlayıcısı

Kaynak kodlarının erişilebilir olması, akıllı sözleşmeleri daha güvenilir hale getirebilir. Kaynak kodunun erişilebilir hale getirilmesi her zaman telif hakkı ile ilgili yasal sorunlara yol açtığından, Solidity derleyicisi, makine tarafından okunabilen SPDX lisans tanımlayıcılarının kullanılmasını teşvik eder:

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

Derleyici, lisansın, SPDX'in izin verdiği liste kapsamında olduğunu doğrulamaz ancak sağlanan dizeyi *bytecode metadata* içine dahil eder.

Bir lisans belirtmek istemiyorsanız veya kaynak kodu açık kaynak değilse, lütfen `UNLICENSED` özel değerini kullanın. `UNLICENSED` (kullanıma izin verilmez, SPDX lisans listesinde bulunmaz) değerinin, `UNLICENSE` (herkese tüm hakları verir) değerinden farklı olduğunu unutmayın. Solidity, ` npm önerisine <https://docs.npmjs.com/cli/v7/configuring-npm/package-json#license>`_ uyar.

Bu açıklamayı eklemeniz, elbette ki sizi, her kaynak dosyada belirli bir lisans adından veya telif hakkının orijinal sahibinden bahsetme zorunluluğu gibi, lisans konusuyla ilgili diğer yükümlülüklerden muaf tutmaz. Derleyici, açıklamayı, dosya düzeyinde dosyanın herhangi bir yerinde algılayabilir ancak dosyanın üst kısmına eklenmesi önerilir.

SPDX lisans tanımlayıcılarının nasıl kullanılacağı hakkında daha fazla bilgi SPDX web sitesinde bulunabilir.

3.4.2 Pragmalar

`Pragma` anahtar sözcüğü, belirli derleyici özelliklerini veya kontrollerini etkinleştirmek için kullanılır. Bir pragma yönergesi, her zaman bir kaynak dosya için yereldir, bu nedenle pragmayı tüm projenizde etkinleştirmek istiyorsanız tüm dosyalarınıza eklemeniz gerekir. Başka bir dosyayı *içe aktarırsanız* o dosyadaki pragma, içe aktarılan dosyaya otomatik olarak _*uygulanmaz*._ .. index:: ! pragma, version

Sürüm Pragması

Uyumsuz değişiklikler getirebilecek gelecekteki derleyici sürümleriyle derlemeyi önlemek için kaynak dosyalarına bir sürüm pragması eklenebilir (ve eklenmelidir). Bunları mutlak minimumda tutmaya ve anlambilimdeki değişikliklerin sözdiziminde de değişiklik gerektireceği şekilde tanıtmaya çalışıyoruz, ancak bu her zaman mümkün olmayabilir. Bu nedenle, en azından işleyişi bozan değişiklikler içeren sürümler için değişiklik günlüğünü okumak her zaman iyi bir fikirdir. Bu sürümler her zaman `0.x.0` veya `x.0.0` biçiminde versiyonlara sahiptir.

Sürüm pragması aşağıdaki gibi kullanılır: `pragma solidity ^0.5.2;`

Yukarıdaki satırı içeren bir kaynak dosyası, 0.5.2'den eski sürümlü bir derleyiciyle derleme yapmadığı gibi, 0.6.0'dan yeni sürümlü bir derleyicide de çalışmaz (bu ikinci koşul `^` kullanılarak eklenir). `0.6.0` sürümüne kadar işleyişi bozan bir değişiklik olmayacağından, kodunuzun amaçladığınız şekilde derleme yaptığından emin olabilirsiniz. Derleyicinin tam sürümü sabit olmadığından hata düzeltme sürümlerinin kullanılması da mümkün olacaktır.

Derleyici sürümü için daha karmaşık kurallar belirlemek mümkündür, bunlar npm tarafından kullanılan sözdizimin aynısına uyar.

Not: Sürüm pragmasının kullanılması, derleyicinin sürümünü _*değiştirmez*._ Derleyicinin özelliklerini etkinleştirme veya devre dışı bırakma işlevine de sahip _*değildir*._ Yalnızca, derleyiciye kendi sürümünün, pragmanın gerektirdiği sürüm ile uyumlu olup olmadığını kontrol etmesi için yönerge verir. Sürümler uyumlu değilse derleyici hata verir.

ABI Kodlayıcı Pragması

`pragma abicoder v1` veya `pragma abicoder v2` kullanarak ABI kodlayıcı ile kod çözücü iki uygulama arasında seçim yapabilirsiniz.

Yeni ABI kodlayıcı (v2) keyfi olarak iç içe geçmiş dizileri ve yapıları kodlama(encode) ve kod çözme(decode) yapabilir . Daha az optimal kod üretebilir ve eski kodlayıcı kadar test edilmemiştir, ancak Solidity 0.6.0'dan itibaren deneysel olmayan olarak kabul edilir. Yine de `pragma abicoder v2;` kullanarak açıkça etkinleştirmeniz gerekir. Solidity 0.8.0'dan itibaren varsayılan olarak etkinleştirileceğinden, `pragma abicoder v1;` kullanarak eski kodlayıcıyı seçme seçeneği vardır.

Yeni kodlayıcı tarafından desteklenen türler, eskisi tarafından desteklenenlerin katı bir üst kümesidir. Bunu kullanan sözleşmeler, kullanmayanlarla sınırlama olmadan etkileşime girebilir. Bunun tersi ancak, `abicoder v2` dışı sözleşme, yalnızca yeni kodlayıcı tarafından desteklenen kod çözme türlerini gerektirecek çağrılarda bulunmaya çalışmadığı sürece mümkündür. Aksi halde, derleyici bu çağrıları tespit ederek hata verebilir. Sözleşmeniz için `abicoder v2` yi etkinleştirmeniz hatanın ortadan kalkması için yeterlidir.

Not: Bu pragma, en nihayetinde kodun nerede sonlandığına bakılmaksızın, etkinleştirildiği dosyada tanımlanan tüm kodlar için geçerlidir. Yani, kaynak dosyası ABI coder v1 ile derlenmek üzere seçilen bir sözleşme, başka bir sözleşmeden kalıt alarak, yeni kodlayıcıyı kullanan kod içermeye devam edebilir. Bu, yeni türlerin, external fonksiyon imzalarında değil, yalnızca dahili olarak kullanılması halinde mümkündür.

Not: Solidity 0.7.4'e kadar, `pragma experimental ABIEncoderV2` kullanarak ABI kodlayıcı v2'yi seçmek mümkündü, ancak varsayılan olduğu için kodlayıcı v1'i açık bir şekilde seçmek mümkün değildi.

Deneysel Pragma

İkinci pragma deneysel pragmadır. Derleyicinin veya dilin henüz varsayılan olarak etkinleştirilmemiş özelliklerini etkinleştirmek için kullanılabilir. Şu anda, aşağıdaki deneysel pragmalar desteklenmektedir:

ABIEncoderV2

ABI kodlayıcı v2 artık deneysel kabul edilmediğinden Solidity 0.7.4 sonrasında `pragma abicoder v2` aracılığıyla seçilebilir (lütfen yukarıya bakın).

SMTChecker

Bu bileşeni, Solidity derleyicisi oluşturulduğunda etkinleştirmek gerektiği için tüm Solidity binary'lerinde mevcut değildir. *build yönergeleri* bu seçeneğin nasıl etkinleştirileceğini açıklar. Çoğu sürümde Ubuntu PPA sürümleri için etkinleştirilmiş olsa da Docker görüntüleri, Windows binary'leri veya statik olarak oluşturulmuş Linux binary'leri için etkin değildir. Yerel olarak yüklenmiş bir SMT çözücünüz varsa ve solc-js'yi node üzerinden (tarayıcı üzerinden değil) çalıştırıyorsanız smtCallback kullanarak solc-js için etkinleştirebilirsiniz.

Eğer `pragma experimental SMTChecker;` kullanırsanız bir SMT çözücü sorgulatarak ek:ref:*güvenlik uyarıları<formal_verification>* alırsınız. Bileşen henüz Solidity dilinin tüm özelliklerini desteklememekte ve muhtemelen çok sayıda uyarı vermektedir. Desteklenmeyen özellikleri bildirmesi durumunda, analiz tamamen sağlıklı olmayabilir.

3.4.3 Diğer Kaynak Dosyalarını İçe Aktarma

Sözdizimi ve Anlambilim

Solidity, kodunuzu modüler hale getirmenize yardımcı olmak için Javascript'te mevcut olanlara (ES6'dan sonrası) benzer import ifadelerini destekler. Ancak, Solidity varsayılan export kavramını desteklemez.

Genel düzeyde, aşağıdaki formdaki içe aktarma deyimlerini kullanabilirsiniz:

```
import "filename";
```

`filename` kısmı *import path* olarak adlandırılır. Bu deyim, "filename "deki tüm global sembolleri (ve orada içe aktarılan sembolleri) geçerli global kapsama içe aktarır (ES6'dakinden farklıdır, ancak Solidity için geriye dönük olarak uyumludur). Bu formun kullanılması tavsiye edilmez, çünkü isim alanını tahmin edilemeyecek şekilde kirletir. "filename" içine yeni üst düzey öğeler eklerseniz, bunlar otomatik olarak "filename "den bu şekilde içe aktarılan tüm dosyalarda görünür. Belirli sembolleri açık bir şekilde içe aktarmak daha iyidir.

Aşağıdaki örnek, üyeleri `"filename"` içindeki tüm global semboller olan yeni bir global sembol `symbolName` oluşturur:

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

bu da tüm global sembollerin `symbolName.symbol` biçiminde kullanılabilir olmasıyla sonuçlanır.

Bu sözdiziminin ES6'nın bir parçası olmayan, ancak muhtemelen yararlı olan bir çeşidi: .. code-block:: solidity

import “filename” as symbolName;

bu da `import * as symbolName from "filename";` ile eşdeğerdir.

Bir adlandırma çakışması varsa içe aktarma sırasında sembolleri yeniden adlandırabilirsiniz. Örneğin, aşağıdaki kod sırasıyla `"filename"` içinden `symbol1` ve `symbol2` yi referans veren yeni global semboller `alias` ve `symbol2` oluşturur. .. code-block:: solidity

import {symbol1 as alias, symbol2} from “filename”;

İçe Aktarma Yolları

Tüm platformlarda tekrarlanabilir derlemeleri destekleyebilmek için Solidity derleyicisinin kaynak dosyalarının depolandığı dosya sisteminin ayrıntılarını soyutlaması gerekir. Bu nedenle içe aktarma yolları doğrudan ana dosya sistemindeki dosyalara başvurmaz. Bunun yerine derleyici, her kaynak birime opak ve yapılandırılmamış bir tanımlayıcı olan benzersiz bir *kaynak birim adı* atanan dahili bir veritabanı (*sanal dosya sistemi* veya kısaca *VFS*) tutar. İçe aktarma ifadesinde belirtilen içe aktarma yolu, bir kaynak birim adına çevrilir ve veritabanında ilgili kaynak birimini bulmak için kullanılır.

Standart JSON API'sini kullanarak, derleyici girdisinin bir parçası olarak tüm kaynak dosyaların adlarını ve içeriğini doğrudan sağlamak mümkündür. Bu durumda kaynak birim adları gerçekten keyfi olabilir. Ancak, derleyicinin kaynak kodu otomatik olarak bulmasını ve VFS'ye yüklemesini istiyorsanız, kaynak birim adlarınızın bir *import callback* i mümkün kılacak şekilde yapılandırılması gerekir. Komut satırı derleyicisini kullanırken varsayılan import callback yalnızca kaynak kodun bir ana bilgisayar dosya sisteminden yüklenmesini destekler; yani kaynak birim adları, yollar olmalıdır.``` Bazı ortamlar daha çok yönlü olan özel callback'ler sağlar. Örneğin Remix IDE, HTTP, IPFS ve Swarm URL'lerinden dosya içe aktarmanıza veya doğrudan NPM kayıt defterindeki paketlere başvurmanıza olanak tanıyan bir tane sağlar. Derleyici tarafından kullanılan sanal dosya sistemi ve yol çözümleme mantığının tam bir açıklaması için bkz *Path Resolution*.

3.4.4 Yorumlar

Tek satırlı yorumlar (`//`) ve çok satırlı yorumlar (`/*...*/`) mümkündür.

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

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

Not: Tek satırlık bir yorum UTF-8 kodlamasında herhangi bir unicode satır sonlandırıcısı (LF, VF, FF, CR, NEL, LS veya PS) ile sonlandırılır. Sonlandırıcı, yorumdan sonra hala kaynak kodun bir parçasıdır, bu nedenle bir ASCII sembolü değilse (bunlar NEL, LS ve PS'dir), bir ayrıştırıcı hatasına yol açacaktır. Ayrıca, NatSpec yorumu adı verilen başka bir yorum türü daha vardır, *stil kılavuzu* içinde ayrıntılı olarak açıklanmıştır. Bunlar üçlü eğik çizgi (`///`) veya çift yıldız bloğu (`/** ... */`) ile yazılır ve doğrudan fonksiyon bildirimlerinin veya deyimlerinin üzerinde kullanılmalıdır.

3.5 Bir Sözleşmenin Yapısı

Solidity'deki sözleşmeler, nesne yönelimli dillerdeki sınıflara benzer. Her kontrat içerisinde şu beyanları bulundurabilir: *Durum Değişkenleri*, *Fonksiyonlar*, *Fonksiyon Değiştiriciler (Modifier'lar)*, *Olaylar (Event)*, *Hatalar*, *Yapı (Struct) Tipleri* ve *Enum Tipleri*. Ayrıca, sözleşmeler bilgileri diğer sözleşmelerden kalıt alabilir.

Aynı zamanda *libraries* ve *interfaces* adı verilen özel sözleşme türleri de vardır.

contracts ile ilgili bölüm, bu bölümden daha fazla ayrıntı içerdiğinden hızlı bir bakış açısı elde etmek adına faydalıdır.

3.5.1 Durum Değişkenleri

Durum değişkenleri, değerleri sözleşmenin deposunda kalıcı olarak saklanan değişkenlerdir.

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

contract SimpleStorage {
    uint storedData; // Durum değişkeni
    // ...
}
```

Geçerli durum değişkeni tiplerini görmek için *Türler* bölümüne ve görünürlük hakkındaki olası seçenekler için *Görünürlük ve Getter Fonksiyonlar* bölümüne bakabilirsiniz.

3.5.2 Fonksiyonlar

Fonksiyonlar, yürütülebilir kod birimleridir. Fonksiyonlar genellikle bir sözleşme içinde tanımlanabilecekleri gibi sözleşmelerin dışında da tanımlanabilirler.

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

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

// Helper fonksiyonu sözleşmenin dışında tanımlanmıştır
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

Fonksiyon Çağrıları dahili veya harici olarak gerçekleşebilir ve diğer sözleşmelere göre farklı *visibility* seviyelerine sahiptir. *Functions* parametre ve değişkenleri birbiri arasında geçirmek için *parameters and return variables* kabul eder.

3.5.3 Fonksiyon Değiştiriciler (Modifier'lar)

Fonksiyon değiştiriciler fonksiyonların semantiğini bildirimsel bir şekilde değiştirmek için kullanılabilir. (sözleşmeler bölümündeki *Fonksiyon Modifier'ları* kısmına bakın).

Aşırı yükleme (Overloading), yani aynı değiştirici adını farklı parametrelerle kullanma durumu mümkün değildir.

Fonksiyonlar gibi, değiştiriciler de *overridden* olabilir.

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

contract Purchase {
    address public seller;

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

    function abort() public view onlySeller { // Değiştirici kullanımı
        // ...
    }
}
```

3.5.4 Olaylar (Event)

Olaylar, EVM için yapılacak olan kayıt işlemlerine kolaylık sağlayan arayüzlerdir.

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

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

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

Olayların nasıl bildirildiği ve bir dapp içinden nasıl kullanılabileceği hakkında bilgi almak için sözleşmeler bölümündeki *Eventler* e bakabilirsiniz.

3.5.5 Hatalar

Hatalar, kodunuzdaki hatalı durumlar için açıklayıcı adlar ve veriler tanımlamanıza olanak sunar. Hatalar *revert statements* içerisinde kullanılabilir. String tanımlamaları ile karşılaştırıldığında, hatalar çok daha zahmetsizdir ve ek verileri kodlamanıza olanak tanır. Hatayı kullanıcıya açıklamak için NatSpec'i kullanabilirsiniz.

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

/// Transfer için yeterli para yok. `requested` kadar miktar istendi,
/// ancak sadece `available` miktarda var.
error NotEnoughFunds(uint requested, uint available);

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

Daha fazla bilgi için sözleşmeler bölümündeki *Hata ve Geri Alma Durumları* a bakın.

3.5.6 Yapı (Struct) Tipleri

Yapılar, birkaç değişkeni grup halinde bir arada bulunduran özel tanımlı türlerdir (tipler bölümündeki *Yapılar* kısmına bakın).

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

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

3.5.7 Enum Tipleri

Enum'lar 'sabit değerlerden' oluşan ve sınırlı sayıda setler halinde oluşturabileceğiniz özel tipler oluşturmanızı sağlar (tipler bölümündeki *Numaralandırmalar (Enums)* kısmına bakın).

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

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

3.6 Türler

Solidity statik olarak yazılmış bir dildir, bu, her bir değişkenin türünün (durum ve yerel) belirtilmesi gerektiği anlamına gelir. Solidity, karmaşık türler oluşturmak için bir araya getirilen birkaç temel tür sağlar.

Ayrıca, operatör içeren ifadelerde türler birbirleriyle etkileşime girebilirler. Çeşitli operatörlere göz atmak için, *Operatörlerin Öncelik Sırası*.

Solidity'de "tanımsız" veya "boş" değerler kavramı yoktur, yeni bildirilen değişkenlerin türüne bağlı olarak her zaman *varsayılan bir değeri* vardır. Beklenmeyen değerler ile uğraşırken, tüm işlemi geri almak için bir *geri alma fonksiyonu* kullanmalı ya da sonucu işaret eden ikinci bir `bool` değerine sahip bir veri demeti döndürmelisiniz.

3.6.1 Değer Türleri

Aşağıdaki türlere de değer türleri denir, çünkü bu türlerin değişkenleri her zaman değere göre iletilir, yani fonksiyon argümanları olarak veya atamalarda kullanıldıklarında her zaman kopyalanırlar.

Booleans

`bool`: Olası değerler `true` ve `false` sabitleridir.

Operatörler:

- `!` (Mantıksal olumsuzlama)
- `&&` (Mantıksal bağlaç, "ve")
- `||` (Mantıksal ayrılma, "veya")
- `==` (Eşitlik)
- `!=` (Eşitsizlik)

`||` ve `&&` operatörleri ortak kısa devre kurallarını uygular. Bunun anlamı `f(x) || g(y)`, eğer `f(x)` `true` (doğru) olarak değerlendirilirse, `g(y)` yan etkileri olsa bile değerlendirilmeyecektir.

Tamsayılar

`int` / `uint`: Çeşitli boyutlarda işaretli ve işaretsiz tam sayılar. `8` (8'den 256 bit'e kadar işaretsiz) ve `uint8` ila `uint256` adımlarında `uint8` ile `uint256` arasındaki anahtar kelimeler. `uint` ve `int` sırasıyla `uint256` ve `int256` için takma adlardır.

Operatörler:

- Karşılaştırmalar: <=, <, ==, !=, >=, > (`bool` olarak değerlendir)
- Bit operatörleri: &, |, ^ (bit düzeyinde özel veya), ~ (bitsel olumsuzlama)
- Değiştirme (Shift) operatörleri: << (sol shift), >> (sağ shift)
- Aritmetik operatörler: +, -, tekli - (sadece imzalı tamsayılar için), *, /, % (mod alma operatörü), ** (ül alma operatörü)

Bir tamsayı türü olan `X` için, tür tarafından gösterilebilen minimum ve maksimum değere erişmek için `type(X).min` ve `type(X).max` ı kullanabilirsiniz.

Uyarı: Solidity'deki tamsayılar belirli bir aralıkla sınırlıdır. Örneğin, `uint32` ile bu `0``dan ``2**32 - 1``e kadardır. Bu türlerde aritmetiğin gerçekleştirildiği iki mod vardır: "wrapping" veya "unchecked" mod ve "checked" mod. Varsayılan olarak, aritmetik her zaman "checked" durumundadır, yani bir işlemin sonucu türün değer aralığının dışına çıkarsa, çağrı bir :ref:`başarısız onaylama<asset-and-require>` aracılığıyla geri döndürülür. ``unchecked { ... }` kullanarak "unchecked" moda geçebilirsiniz. Daha fazla ayrıntı *unchecked* ile ilgili bölümde bulunabilir.

Karşılaştırmalar

Bir karşılaştırmanın değeri, tamsayı değeri karşılaştırılarak elde edilen değerdir.

Bit işlemleri

Bit işlemleri, sayının ikisinin tümleyen gösterimi üzerinde gerçekleştirilir. Bu, örneğin `~int256(0) == int256(-1)` anlamına gelir.

Shifts

Bir kaydırma işleminin sonucu, sol işlenenin türüne sahiptir ve sonucu türle eşleşecek şekilde kısaltır. Doğru işlenen imzasız türde olmalıdır, imzalı bir türle kaydırmaya çalışmak derleme hatası üretecektir.

Vardiyalar, aşağıdaki şekilde ikinin kuvvetleriyle çarpma kullanılarak "simüle edilebilir". Sol işlenenin türünün kesilmesinin her zaman sonunda gerçekleştirildiğini, ancak açıkça belirtilmediğini unutmayın.

- `x << y`, `x * 2**y` matematiksel ifadesine eşdeğerdir.
- `x >> y`, `x / 2**y` matematiksel ifadesine eşdeğerdir, negatif sonsuza yuvarlanır.

Uyarı: `0.5.0` sürümünden önce, negatif `x` için bir sağa kaydırma `x >> y` sıfıra yuvarlanmış `x / 2**y` matematiksel ifadesine eşdeğerdi, yani sağa kaydırmalar, aşağı yuvarlama (negatif sonsuza doğru) yerine yukarı (sıfıra doğru) yuvarlama olarak kullanılır.

Not: Aritmetik işlemlerde olduğu gibi kaydırma işlemleri için de taşma kontrolleri yapılmaz. Bunun yerine, sonuç her zaman kesilir.

Toplama, Çıkarma ve Çarpma

Toplama, çıkarma ve çarpma, taşma ve alttan akışa ilişkin iki farklı mod ile olağan semantiklere sahiptir:

Varsayılan olarak, tüm aritmetik yetersiz veya taşma açısından kontrol edilir, ancak bu, *unchecked blok* kullanılarak devre dışı bırakılabilir, bu da sarma aritmetiğiyle sonuçlanır. Daha fazla ayrıntı o bölümde bulunabilir.

`-x` ifadesi, `(T(0) - x)` ile eşdeğerdir; burada `T`, `x``in türüdür. Yalnızca imzalı türlere uygulanabilir. ``x` negatifse `-x` in değeri pozitif olabilir. İkisinin tamamlayıcı temsilinden kaynaklanan başka bir uyarı daha var:

`int x = type(int).min;` varsa, `-x` pozitif aralığa uymaz. `unchecked { assert(-x == x); }` çalışır ve işaretli modda kullanıldığında `-x` ifadesi başarısız bir onaylamaya neden olur.

Bölme

Bir işlemin sonucunun türü her zaman işlenenlerden birinin türü olduğundan, tamsayılarda bölme her zaman bir tamsayı ile sonuçlanır. Solidity'de bölme sıfıra doğru yuvarlanır. Bu, `int256(-5) / int256(2) == int256(-2)` anlamına gelir.

Buna karşılık, *değişmezler (literals)* üzerinde bölmenin keyfi kesinliğin kesirli değerleriyle sonuçlandığını unutmayın.

Not: Sıfıra bölme bir *panik hatasına* neden olur. Bu kontrol, `unckecked { ... }` ile devre dışı **bırakılamaz**.

Not: `type(int).min / (-1)` ifadesi, bölmenin taşmaya neden olduğu tek durumdur. Kontrollü aritmetik modda, bu başarısız bir onaylamaya neden olurken, sarma modunda değer `type(int).min` olacaktır.

Mod Alma

Mod alma işlemi `a % n`, `a` işleneninin `n` işlenenine bölünmesinden sonra kalan `r``yi verir, burada ``q = int(a / n)` ve `r = a - (n * q)`. Bu, mod alma işleminin sol işleneni (veya sıfır) ile aynı işaretle sonuçlandığı ve `a % n == -(-a % n)``nin negatif ``a` için geçerli olduğu anlamına gelir:

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

Not: Sıfırlı mod alma işlemi *Panik hatasına* neden oluyor. Bu kontrol, `unckecked { ... }` ile devre dışı **bırakılamaz**.

Üs Alma

Üs, yalnızca üsteki işaretsiz türler için kullanılabilir. Elde edilen bir üs türü her zaman tabanın türüne eşittir. Lütfen sonucu tutacak ve olası onaylama hatalarına veya sarma davranışına hazırlanacak kadar büyük olmasına dikkat edin.

Not:

İşaretli (checked) modda, üs alma yalnızca küçük tabanlar için nispeten ucuz `exp` işlem kodunu kullanır.
`x**3` durumları için `x*x*x` ifadesi daha ucuz olabilir. Her durumda, gaz maliyeti testleri ve optimize edicinin kullanılması tavsiye edilir.

Not: `0**0``ın EVM tarafından ``1` olarak tanımlandığını unutmayın.

Sabit Nokta Sayıları

> **Uyarı:** Sabit nokta sayıları henüz Solidity tarafından tam olarak desteklenmemektedir. Bildirilebilirler, ancak atanamazlar veya atanamazlar.

`fixed` / `ufixed`: Çeşitli boyutlarda imzalı ve imzasız sabit nokta sayısı. Anahtar sözcükler `ufixedMxN` ve `fixedMxN`, burada `M` türün aldığı bit sayısını ve `N` kaç ondalık noktanın mevcut olduğunu gösterir. `M` 8'e bölünebilir olmalı ve 8'den 256 bit'e kadar gider. `N` 0 ile 80 arasında olmalıdır. `ufixed` ve `fixed` sırasıyla `ufixed128x18` ve `fixed128x18` için takma adlardır.

Operatörler:

- Karşılaştırma: <=, <, ==, !=, >=, > (`bool` olarak değerlendir)
- Aritmetik operatörler: +, -, tekil -, *, /, % (mod alma)

Not: Kayan nokta (birçok dilde `float` ve `double`, daha doğrusu IEEE 754 sayıları) ile sabit nokta sayıları arasındaki temel fark, tamsayı ve kesirli kısım için kullanılan bit sayısının (birçok dilde ondalık nokta) birincisinde esnektir, ikincisinde ise kesin olarak tanımlanmıştır. Genel olarak, kayan noktada neredeyse tüm alan sayıyı temsil etmek için kullanılırken, ondalık noktanın nerede olduğunu yalnızca az sayıda bit tanımlar.

Adresler

Adres türü, büyük ölçüde aynı olan iki şekilde gelir:

- `address`: 20 baytlık bir değer tutar (bir Ethereum adresinin boyutu).
- `address payable`: `address` ile aynıdır, ek olarak `transfer` ve `send` bulundurur.

Bu ayrımın arkasındaki fikir, `address payable` in, Ether gönderebileceğiniz bir adres olduğu, ancak Ether'i düz bir `address` e göndermemeniz gerektiğidir, örneğin akıllı bir sözleşme olabileceği için. Ether'i kabul etmek için oluşturulmamıştır.

Tür dönüşümleri:

`address payable``den ``address``e örtülü dönüşümlere izin verilirken, ``address``den ``address payable``a dönüşümler ``payable(<address>)` üzerinden açık olmalıdır.

`uint160`, tamsayı değişmezleri, `bytes20` ve sözleşme türleri için `address` e ve adresten açık dönüşümlere izin verilir.

Yalnızca `address` ve sözleşme türündeki ifadeler, açık dönüştürme `payable(...)` aracılığıyla `address payable` türüne dönüştürülebilir. Sözleşme türü için, bu dönüştürmeye yalnızca sözleşme Ether alabiliyorsa, yani sözleşmenin bir *alma* veya ödenebilir yedek fonksiyonu varsa izin verilir. `payable(0)` ın geçerli olduğunu ve bu kuralın bir istisnası olduğunu unutmayın.

Not: `address` türünde bir değişkene ihtiyacınız varsa ve buna Ether göndermeyi planlıyorsanız, bu gereksinimi görünür kılmak için türünü `address payable` olarak bildirin. Ayrıca, bu ayrımı veya dönüşümü mümkün olduğunca erken yapmaya çalışın.

Operatörler:

- <=, <, ==, !=, >= ve >

Uyarı:

Daha büyük bir bayt boyutu kullanan bir türü bir `address``e, örneğin ``bytes32``ye dönüştürürseniz, ``address` kısaltılır. Dönüştürme belirsizliğini azaltmak için sürüm 0.4.24 ve derleyici kuvvetinin daha yüksek sürümü, dönüştürmede kesmeyi açık hale getirirsiniz.

Örneğin, `0x1111222233334444555566667777888899999AAAABBBBCCCCDDDDEEEEFFFFCCC` 32 bayt değerini alın.

`address(uint160(bytes20(b)))` kullanabilirsiniz, bu da `0x1111222233334444555566667777888899999aAaa` ile sonuçlanır,

veya `0x777788889999AaAAbBbbCccccddDdeeeEfFFfCcCc` ile sonuçlanan `address(uint160(uint256(b)))` i kullanabilirsiniz.

Not: `address` ve `address payable` arasındaki ayrım, 0.5.0 sürümüyle tanıtıldı. Ayrıca bu versiyondan başlayarak, sözleşmeler adres türünden türetilmez, ancak yine de bir alma veya ödeme geri dönüş fonksiyonu varsa, açıkça `address` e veya `address payable` a dönüştürülebilir.

Adres Üyeleri

Adreslerin tüm üyelerine hızlıca göz atmak için, bkz.:ref:*address_related*.

- `balance` and `transfer`

Bir adresin bakiyesini `balance` özelliğini kullanarak sorgulamak ve `transfer` fonksiyonunu kullanarak Ether'i (wei birimi cinsinden) bir ödenecek adrese göndermek mümkündür:

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

Mevcut sözleşmenin bakiyesi yeterince büyük değilse veya Ether transferi alıcı hesap tarafından reddedilirse `transfer` fonksiyonu başarısız olur. `transfer` fonksiyonu başarısızlık üzerine geri döner.

Not: `x` bir sözleşme (kontrat) adresiyse, kodu (daha spesifik olarak: varsa *Receive Ether Fonksiyonu* veya varsa *Fallback Fonksiyonu* yürütülür. `transfer` çağrısı ile birlikte (bu, EVM'nin bir özelliğidir ve engellenemez). Bu yürütmenin gazı

biterse veya herhangi bir şekilde başarısız olursa, Ether transferi geri alınacak ve mevcut sözleşme bir istisna dışında durdurulacaktır.

- `send`

Gönder, `transfer``in alt düzey karşılığıdır. Yürütme (execution) başarısız olursa, mevcut sözleşme bir istisna dışında durmaz, ancak ``send`, `false` döndürür.

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`.

Uyarı:

`send` kullanmanın bazı tehlikeleri vardır:
Çağrı yığını derinliği 1024 ise aktarım başarısız olur (bu her zaman arayan tarafından zorlanabilir) ve ayrıca alıcının gazı biterse de başarısız olur. Bu nedenle, güvenli Ether transferleri yapmak için her zaman `send` in dönüş değerini kontrol edin, `transfer` i kullanın veya daha iyisi: alıcının parayı çektiği bir kalıp kullanın.

- `call`, `delegatecall` ve `staticcall`

ABI'ye uymayan sözleşmelerle arayüz oluşturmak veya kodlama üzerinde daha doğrudan kontrol sağlamak için `call`, `delegatecall` ve `staticcall` fonksiyonları sağlanmıştır. Hepsi tek bir `bytes memory` parametresi alır ve başarı koşulunu (`bool` olarak) ve döndürülen verileri (`bytes memory`) döndürür. Yapılandırılmış verileri kodlamak için `abi.encode`, `abi.encodePacked`, `abi.encodeWithSelector` ve `abi.encodeWithSignature` fonksiyonları kullanılabilir.

Örnek:

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

Uyarı: Tüm bu fonksiyonlar alt düzey fonksiyonlarıdır ve dikkatli kullanılmalıdır. Spesifik olarak, bilinmeyen herhangi bir sözleşme kötü niyetli olabilir ve onu çağırırsanız, kontrolü o sözleşmeye devredersiniz ve bu da sözleşmenize geri çağrı yapabilir, bu nedenle arama geri döndüğünde durum değişkenlerinizdeki değişikliklere hazır olun. Diğer sözleşmelerle etkileşime girmenin normal yolu, bir sözleşme nesnesi (`x.f()`) üzerindeki bir fonksiyonu çağırmaktır.

Not: Solidity'nin önceki sürümleri, bu fonksiyonların rastgele argümanlar almasına izin veriyordu ve ayrıca `bytes4` türündeki ilk argümanı farklı şekilde ele alıyorlardı. Bu uç durumlar 0.5.0 sürümünde kaldırılmıştır.

Verilen gazı `gas` değiştiricisi ile ayarlamak mümkündür:

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

Benzer şekilde, sağlanan Ether değeri de kontrol edilebilir:

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

Son olarak, bu değiştiriciler birleştirilebilir. Onların sırası önemli değil:

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

Benzer şekilde, `delegatecall` fonksiyonu kullanılabilir: fark, yalnızca verilen adresin kodunun kullanılması, diğer tüm yönlerin (depolama, bakiye, . . .) mevcut sözleşmeden alınmasıdır. `delegatecall` un amacı, başka bir sözleşmede saklanan kütüphane kodunu kullanmaktır. Kullanıcı, her iki sözleşmedeki depolama düzeninin, kullanılacak temsilci çağrısı için uygun olduğundan emin olmalıdır.

Not: Homestead'den önce, orijinal `msg.sender` ve `msg.value` değerlerine erişim sağlamayan `callcode` adlı yalnızca sınırlı bir değişken mevcuttu. Bu fonksiyon 0.5.0 sürümünde kaldırılmıştır.

Bizans'tan (Byzantium) beri `staticcall` da kullanılabilir. Bu temelde `call` ile aynıdır, ancak çağrılan fonksiyon durumu herhangi bir şekilde değiştirirse geri döner.

Her üç fonksiyon, `call`, `delegatecall` ve `staticcall` çok düşük düzeyli fonksiyonlardır ve Solidity'nin tür güvenliğini bozdukları için yalnızca *son çare* olarak kullanılmalıdır.

`Gas` seçeneği her üç yöntemde de mevcuttur, `value` seçeneği ise yalnızca `call` da mevcuttur.

Not: Durumun okunması veya yazılmasından bağımsız olarak akıllı sözleşme kodunuzdaki sabit kodlanmış gaz değerlerine güvenmekten kaçınmak en iyisidir, çünkü bunun birçok tuzağı olabilir. Ayrıca, gelecekte gaza erişim değişebilir.

- `code` and `codehash`

Herhangi bir akıllı sözleşme için dağıtılan kodu sorgulayabilirsiniz. EVM bayt kodunu boş olabilecek bir `bytes memory` olarak almak için `.code` kullanın. `.codehash` kullanın, bu kodun Keccak-256 karmasını alın (`bytes32` olarak). `addr.codehash``in ``keccak256(addr.code)` kullanmaktan daha ucuz olduğunu unutmayın.

Not: Tüm sözleşmeler `address` türüne dönüştürülebilir, bu nedenle `address(this).balance` kullanılarak mevcut sözleşmenin bakiyesini sorgulamak mümkündür.

Sözleşme Türleri

Her *sözleşme* kendi türünü tanımlar. Sözleşmeleri dolaylı olarak miras aldıkları sözleşmelere dönüştürebilirsiniz. Sözleşmeler açıkça `address` türüne dönüştürülebilir.

`address payable` türüne ve `address payable` türünden açık dönüştürme, yalnızca sözleşme türünün bir alacak veya ödenebilir yedek fonksiyonu varsa mümkündür. Dönüştürme hala `address(x)` kullanılarak gerçekleştirilir. Sözleşme türünün bir alma veya ödenebilir yedek fonksiyonu yoksa, `address payable``a dönüştürme ``payable(address(x))` kullanılarak yapılabilir.

Adres türü ile ilgili bölümde daha fazla bilgi bulabilirsiniz.

Not: 0.5.0 sürümünden önce, sözleşmeler doğrudan adres türünden türetilir, ve `address` ve `address payable` arasında bir ayrım yoktu.

Sözleşme tipinde (`MyContract c`) yerel bir değişken bildirirseniz, o sözleşmedeki fonksiyonları çağırabilirsiniz. Aynı sözleşme türünden bir yerden atamaya özen gösterin.

Ayrıca sözleşmeleri somutlaştırabilirsiniz (bu, sözleşmelerin yeni oluşturuldukları anlamına gelir). Daha fazla ayrıntıyı *'Contracts via new'* bölümünde bulabilirsiniz.

Bir sözleşmenin veri temsili, `address` türününkiyle aynıdır ve bu tür aynı zamanda *ABI* içinde kullanılır.

Sözleşmeler hiçbir operatörü desteklemez.

Sözleşme türlerinin üyeleri, `public` olarak işaretlenen tüm durum değişkenleri dahil olmak üzere sözleşmenin harici fonksiyonlarıdır.

Bir `C` sözleşmesi için, sözleşmeyle ilgili *tür bilgisine* erişmek için `type(C)` yi kullanabilirsiniz.

Sabit Boyutlu Bayt Dizileri

`bytes1`, `bytes2`, `bytes3`, ..., `bytes32` değer türleri 1'den 32'ye kadar bir bayt dizisini tutar.

Operatörler:

- Karşılaştırmalar: <=, <, ==, !=, >=, > (`bool` olarak değerlendir)
- Bit operatörleri: &, |, ^ (bit düzeyinde özel veya), ~ (bitsel olumsuzlama)
- Shift operatörleri: << (sol shift), >> (sağ shift)
- Dizin erişimi: `x`, `bytesI` türündeyse, `0 <= k < I` için `x[k]`, `k` ıncı baytı (salt okunur) döndürür.

Kaydırma operatörü, sağ işlenen olarak işaretsiz tamsayı türüyle çalışır (ancak sol işlenenin türünü döndürür), bu, kaydırılacak bit sayısını belirtir. İmzalı bir türe göre kaydırma, bir derleme hatası üretecektir.

Üyeler:

- `.length`, bayt dizisinin sabit uzunluğunu verir (salt okunur).

Not: `bytes1[]` türü bir bayt dizisidir, ancak doldurma kuralları nedeniyle her öğe için (depolama dışında) 31 baytlık alan harcar. Bunun yerine `bytes` türünü kullanmak daha iyidir.

Not: 0.8.0 sürümünden önce, `byte`, `bytes1` için bir takma addı.

Dinamik Olarak Boyutlandırılmış Bayt Dizisi

`bytes`:
Dinamik olarak boyutlandırılmış bayt dizisi, bkz. *Diziler*. Bir değer türü değil!

`string`:
Dinamik olarak boyutlandırılmış UTF-8 kodlu dize, bkz.:ref:*arrays*. Bir değer türü değil!

Adres Değişmezleri

Adres sağlama toplamı (checksum) testini geçen onaltılık sabit değerler, örneğin `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF`, `address` türündedir.

39 ila 41 basamak uzunluğunda olan ve sağlama toplamı (checksum) testini geçmeyen onaltılık değişmez değerler bir hata üretir. Hatayı kaldırmak için başa (tamsayı türleri için) veya sona(bytesNN türleri için) sıfırlar ekleyebilirsiniz.

Not: Karışık büyük/küçük harfli adres sağlama toplamı biçimi, EIP-55 içinde tanımlanır.

Rasyonel ve Tamsayı Değişmezleri

Tamsayı değişmezleri, 0-9 aralığında bir basamak dizisinden oluşturulur. Ondalık sayılar olarak yorumlanırlar. Örneğin, `69` altmış dokuz anlamına gelir. Solidity'de sekizlik değişmez değerler yoktur ve baştaki sıfırlar geçersizdir.

Ondalık kesirli değişmezler, `.`'nın ardından en az bir sayı yerleştirilmesi ile oluşturulur. Örnekler arasında `.1` ve `1.3` bulunur (`1.` geçersizdir).

Mantisin kesirli olabileceği ancak üssün bir tamsayı olması gereken `2e10` şeklindeki bilimsel gösterim de desteklenmektedir. `MeE` değişmez değeri, `M * 10**E` ile eşdeğerdir. Örnekler arasında `2e10`, `-2e10`, `2e-10`, `2.5e1` yer alır.

Okunabilirliğe yardımcı olmak için sayısal bir hazır bilginin basamaklarını ayırmak için alt çizgiler kullanılabilir. Örneğin, ondalık (decimal) `123_000`, onaltılık (hexadecimal) `0x2eff_abde`, bilimsel ondalık gösterim `1_2e345_678` hepsi geçerlidir. Alt çizgiye yalnızca iki basamak arasında izin verilir ve yalnızca bir ardışık alt çizgiye izin verilir. Alt çizgi içeren bir sayı değişmezine ek bir anlamsal anlam eklenmez, alt çizgiler yoksayılır.

Sayı değişmezi ifadeleri, sabit olmayan bir türe dönüştürülene kadar (yani, bunları bir sayı değişmezi ifadesi (boolean değişmezleri gibi) dışında herhangi bir şeyle birlikte kullanarak veya açık dönüştürme yoluyla) isteğe bağlı kesinliği korur. Bu, hesaplamaların taşmadığı ve bölmelerin sayı değişmez ifadelerinde kesilmediği anlamına gelir.

Örneğin, `(2**800 + 1) - 2**800`, ara sonuçlar makine kelime boyutuna bile sığmasa da `1` sabitiyle sonuçlanır (`uint8` türünden). Ayrıca, `.5 * 8`, `4` tamsayısıyla sonuçlanır (arada tamsayı olmayanlar kullanılmasına rağmen).

Uyarı: Çoğu operatör, değişmez değerlere uygulandığında değişmez bir ifade üretirken, bu kalıbı takip etmeyen bazı operatörler vardır:

- Üçlü operatör (`... ? ... : ...`),
- Dizi alt simgesi (subscript) (`<array>[<index>]`).

`255 + (true ? 1 : 0)` veya `255 + [1, 2, 3][0]` gibi ifadelerin doğrudan 256 değişmezini kullanmaya eşdeğer olmasını bekleyebilirsiniz, ancak aslında bunlar `uint8` türünde hesaplanır ve taşabilir.

Tamsayılara uygulanabilen herhangi bir operatör, işlenenler tamsayı olduğu sürece sayı değişmez ifadelerine de uygulanabilir. İkisinden herhangi biri kesirliyse, bit işlemlerine izin verilmez ve üs kesirliyse üs almaya izin verilmez (çünkü bu rasyonel olmayan bir sayıya neden olabilir).

Sol (veya taban) işlenen olarak değişmez sayılar ve sağ (üs) işlenen olarak tamsayı türleri ile kaydırmalar ve üs alma, her zaman "uint256" (negatif olmayan değişmezler için) veya sağ (üs) işlenenin türünden bağımsız olarak "int256" (negatif değişmezler için) içinde gerçekleştirilir.

Uyarı: 0.4.0 sürümünden önce Solidity'de tamsayı değişmezleri üzerinde bölme kullanılırdı, ancak şimdi rasyonel bir sayıya dönüştürülür, yani `5 / 2`, `2` ye eşit değil, `2.5` e eşittir .

Not: Solidity, her rasyonel sayı için bir sayı değişmez (literal) tipine sahiptir. Tamsayı değişmezleri ve rasyonel sayı değişmezleri, sayı değişmez türlerine aittir. Ayrıca, tüm sayı değişmez ifadeleri (yani yalnızca sayı değişmezlerini ve işleçlerini içeren ifadeler) sayı değişmez türlerine aittir. Dolayısıyla, `1 + 2` ve `2 + 1` sayı değişmez ifadelerinin her ikisi de üç rasyonel sayı için aynı sayı değişmez türüne aittir.

Not: Sayı değişmez ifadeleri, değişmez olmayan ifadelerle birlikte kullanılır kullanılmaz, değişmez bir türe dönüştürülür. Türlerden bağımsız olarak, aşağıdaki `b``ye atanan ifadenin değeri bir tamsayı olarak değerlendirilir. "a", "uint128" türünde olduğundan, "2.5 + a" ifadesinin uygun bir türe`

sahip olması gerekir. ``2.5 ve uint128 tipi için ortak bir tip olmadığı için Solidity derleyicisi bu kodu kabul etmez.

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

Dize Değişmezleri ve Türleri

Dize değişmezleri ya çift ya da tek tırnak ("foo" veya 'bar') ile yazılır ve ayrıca uzun dizelerle uğraşırken yardımcı olabilecek şekilde birden çok ardışık parçaya bölünebilirler ("foo" "bar", "foobar" ile eşdeğerdir). C'deki gibi sondaki sıfırları ima etmezler; "foo" dört değil, üç baytı temsil eder. Tamsayı değişmezlerinde olduğu gibi, türleri değişebilir, ancak sığarlarsa "bytes1", ..., "bytes32"ye örtük olarak "bytes" ve "string"e dönüştürülebilirler.

Örneğin, bytes32 samevar = "stringliteral" ile dize değişmezi, bir bytes32 türüne atandığında ham bayt biçiminde yorumlanır.

Dize değişmezleri yalnızca yazdırılabilir ASCII karakterleri içerebilir; bu, 0x20 .. 0x7E arasındaki ve dahil olan karakterler anlamına gelir.

Ayrıca, dize değişmezleri aşağıdaki kaçış karakterlerini de destekler:

- \<newline> (gerçek bir yeni satırdan kaçar)
- \\ (ters eğik çizgi)
- \' (tek alıntı)
- \" (çift alıntı)
- \n (Yeni satır)
- \r (satırbaşı)
- \t (etiket)
- \xNN (hex kaçış, aşağıya bakınız)
- \uNNNN (unicode kaçış, aşağıya bakınız)

\xNN bir onaltılık değer alıp uygun baytı eklerken, \uNNNN bir Unicode kod noktası alır ve bir UTF-8 dizisi ekler.

Not: 0.8.0 sürümüne kadar üç ek kaçış dizisi vardı: \b, \f ve \v. Diğer dillerde yaygın olarak bulunurlar, ancak pratikte nadiren ihtiyaç duyulur. Bunlara ihtiyacınız varsa, yine de diğer ASCII karakterleri gibi, sırasıyla \x08, \x0c ve \x0b gibi onaltılık çıkışlar yoluyla eklenebilirler.

Aşağıdaki örnekteki dizenin uzunluğu on bayttır. Yeni satır baytı ile başlar, ardından çift tırnak, tek tırnak, ters eğik çizgi ve ardından (ayırıcı olmadan) abcdef karakter dizisi gelir.

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

Yeni satır olmayan herhangi bir Unicode satır sonlandırıcı (yani LF, VF, FF, CR, NEL, LS, PS) dize değişmezini sonlandırdığı kabul edilir. Yeni satır, yalnızca önünde bir \ yoksa dize değişmezini sonlandırır.

Unicode Değişmezler

Normal dize değişmezleri yalnızca ASCII içerebilirken, Unicode değişmezleri `unicode` – anahtar kelimesiyle önek – herhangi bir geçerli UTF-8 dizisi içerebilir. Ayrıca, normal dize değişmezleri ile aynı kaçış dizilerini de desteklerler.

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

Onaltılık (Hexadecimal) Değişmezler

Onaltılık değişmezlerin önüne `hex` anahtar kelimesi getirilir ve çift veya tek tırnak içine alınır (`hex"001122FF"` , `hex'0011_22_FF'`). İçerikleri, isteğe bağlı olarak bayt sınırları arasında ayırıcı olarak tek bir alt çizgi kullanabilen onaltılık basamaklar olmalıdır. Değişmez değerin değeri, onaltılık dizinin ikili gösterimi olacaktır.

Boşlukla ayrılmış birden çok onaltılık sabit değer, tek bir sabit değerde birleştirilir: `hex"00112233" hex"44556677"` , `hex"0011223344556677"` ye eşittir

Onaltılık değişmez değerler *string değişmezleri* gibi davranır ve aynı dönüştürülebilirlik kısıtlamalarına sahiptir.

Numaralandırmalar (Enums)

Numaralandırmalar, Solidity'de kullanıcı tanımlı bir tür oluşturmanın bir yoludur. Tüm tamsayı türlerine açıkça dönüştürülebilirler, ancak örtük dönüştürmeye izin verilmez. Tamsayıdan yapılan açık dönüştürme, çalışma zamanında değerin numaralandırma aralığı içinde olup olmadığını kontrol eder ve aksi takdirde bir *Panik hatası* oluşmasına neden olur. Numaralandırmalar en az bir üye gerektirir ve bildirildiğinde varsayılan değeri ilk üyedir. Numaralandırmaların 256'dan fazla üyesi olamaz.

Veri gösterimi, C'deki numaralandırmalarla aynıdır: Seçenekler, `0` dan başlayan müteakip işaretsiz tamsayı değerleriyle temsil edilir.

`type(NameOfEnum).min` ve `type(NameOfEnum).max` kullanarak verilen numaralandırmanın en küçük ve sırasıyla en büyük değerini alabilirsiniz.

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

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

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

    // Enum türleri ABI'nin bir parçası olmadığından, Solidity'nin dışındaki tüm konular␣
↪için "getChoice" imzası otomatik olarak "getChoice() returns (uint8)" olarak␣
↪değiştirilecektir.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

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

Not: Numaralandırmalar, sözleşme veya kitaplık tanımlarının dışında dosya düzeyinde de bildirilebilir.

Kullanıcı Tanımlı Değer Türleri

Kullanıcı tanımlı bir değer türü, bir temel değer türü üzerinde sıfır maliyetli bir soyutlama oluşturmaya izin verir. Bu, takma ada benzer, ancak daha katı tür gereksinimleri vardır.

Kullanıcı tanımlı bir değer türü, `type C is V` kullanılarak tanımlanır; burada `C` yeni tanıtılan türün adıdır ve `V` yerleşik bir değer türü olmalıdır ("altta yatan tip"/ "underlying type"). `C.wrap` fonksiyonu, temeldeki türden özel türe dönüştürmek için kullanılır. Benzer şekilde, özel türden temel türe dönüştürmek için `C.unwrap` fonksiyonu kullanılır.

`C` türünün herhangi bir işleci veya bağlı üye fonksiyonu yoktur. Özellikle, == operatörü bile tanımlanmamıştır. Diğer türlere ve diğer türlerden açık ve örtük dönüştürmelere izin verilmez.

Bu türlerin değerlerinin veri temsili, temeldeki türden devralınır ve temel alınan tür de ABI'da kullanılır.

Aşağıdaki örnek, 18 ondalık basamaklı bir ondalık sabit nokta türünü ve tür üzerinde aritmetik işlemler yapmak için bir minimum kitaplığı temsil eden özel bir `UFixed256x18` türünü gösterir.

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

// Kullanıcı tanımlı bir değer türü kullanarak 18 ondalık, 256 bit genişliğinde sabit␣
↪nokta türünü temsil eder.
type UFixed256x18 is uint256;

/// UFixed256x18 üzerinde sabit nokta işlemleri yapmak için minimal bir kütüphane.
library FixedMath {
    uint constant multiplier = 10**18;

    ///İki UFixed256x18 sayısı ekler. uint256'da kontrol edilen aritmetiği temel alarak␣
↪taşma durumunda geri döner.
    function add(UFixed256x18 a, UFixed256x18 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) + UFixed256x18.unwrap(b));
    }
    /// UFixed256x18 ve uint256'yı çarpar. uint256'da kontrol edilen aritmetiği temel␣
↪alarak taşma durumunda geri döner.
    function mul(UFixed256x18 a, uint256 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
    }
    /// UFixed256x18 numarasının zeminini alın.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    /// "a"yı geçmeyen en büyük tamsayıyı döndürür.
    function floor(UFixed256x18 a) internal pure returns (uint256) {
        return UFixed256x18.unwrap(a) / multiplier;
    }
    /// Bir uint256'yı aynı değerde bir UFixed256x18'e dönüştürür.
    /// Tamsayı çok büyükse geri döner.
    function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(a * multiplier);
    }
}
```

`UFixed256x18.wrap` ve `FixedMath.toUFixed256x18` öğelerinin nasıl aynı imzaya sahip olduğuna, ancak çok farklı iki işlem gerçekleştirdiğine dikkat edin: `UFixed256x18.wrap` işlevi, girişle aynı veri temsiline sahip bir `UFixed256x18` döndürürken, `toUFixed256x18`, aynı sayısal değere sahip bir `UFixed256x18` döndürür.

Fonksiyon Tipleri

Fonksiyon türleri, kullanulan fonksiyonların türleridir. Fonksiyon tipinin değişkenleri fonksiyonlardan atanabilir ve fonksiyon tipinin fonksiyon parametreleri fonksiyon çağrılarına fonksiyon geçirmek ve fonksiyon çağrılarından fonksiyon döndürmek için kullanılabilir. Fonksiyon türleri iki şekilde gelir - *dahili* ve *harici* fonksiyonlar:

Dahili fonksiyonlar, yalnızca geçerli sözleşmenin içinde (daha spesifik olarak, dahili kitaplık fonksiyonları ve devralınan fonksiyonları da içeren geçerli kod biriminin içinde) çağrılabilir çünkü bunlar geçerli sözleşmenin bağlamı dışında yürütülemezler. Dahili bir fonkaiyonu çağırmak, tıpkı mevcut sözleşmenin bir fonksiyonunu dahili olarak çağırırken olduğu gibi, giriş etiketine atlanarak gerçekleştirilir.

Harici fonksiyonlar bir adres ve bir işlev imzasından oluşur ve bunlar iletilebilir ve harici fonksiyon çağrılarından döndürülebilir.

Fonksiyon türleri aşağıdaki gibi not edilir:

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

Parametre türlerinin aksine, dönüş türleri boş olamaz - fonksiyonun türünün hiçbir şey döndürmemesi gerekiyorsa, `returns (<return types>)` bölümünün tamamı atlanmalıdır.

Varsayılan olarak, fonksiyon türleri dahilidir, bu nedenle `internal` anahtar sözcüğü atlanabilir. Bunun yalnızca fonksiyon türleri için geçerli olduğunu unutmayın. Sözleşmelerde tanımlanan fonksiyonlar için görünürlük açıkça belirtilmelidir, varsayılan değer yoktur.

Dönüşümler:

`A` fonksiyon türü, yalnızca ve yalnızca parametre türleri aynıysa, dönüş türleri aynıysa, dahili/harici özellikleri aynıysa ve `A` öğesinin durum değişkenliği aynıysa, dolaylı olarak `B` işlev türüne dönüştürülebilir. `A`, `B` durum değişkenliğinden daha kısıtlayıcıdır. Özellikle:

- `pure` fonksiyonlar, `view` ve `non-payable` fonksiyonlara dönüştürülebilir
- `view` fonksiyonları `non-payable` fonksiyonlara dönüştürülebilir
- `payable` fonksiyonlar `non-payable` fonksiyonlara dönüştürülebilir

Fonksiyon türleri arasında başka hiçbir dönüşüm mümkün değildir.

`payable` ve `non-payable` fonksiyonlarla alakalı kural biraz kafa karıştırıcı olabilir, ancak özünde, bir fonksiyon `payable` ise, bu aynı zamanda sıfır Ether ödemesini de kabul ettiği anlamına gelir, yani bu fonksiyon atrıca

non-payable``dır. Öte yandan, bir ``non-payable fonksiyon kendisine gönderilen Ether'i reddedecektir, bu nedenle non-payable fonksiyonlar payable fonksiyonlara dönüştürülemez.

Bir fonksiyon türü değişkeni başlatılmazsa, onu çağırmak bir *Panik hatası* ile sonuçlanır. Aynısı, bir fonksiyon üzerinde delete kullandıktan sonra çağırırsanız da olur.

Harici fonksiyon türleri, Solidity bağlamı dışında kullanılırsa, adres ve ardından fonksiyon tanımlayıcısını birlikte tek bir bytes24 türünde kodlayan function türü olarak kabul edilirler.

Mevcut sözleşmenin genel (public) fonksiyonlarının hem dahili hem de harici (external) bir fonksiyon olarak kullanılabileceğini unutmayın. f yi dahili bir fonksiyon olarak kullanmak için f yi kullanın, harici biçimini kullanmak istiyorsanız this.f yi kullanın.

Dahili tipte bir fonksiyon, nerede tanımlandığına bakılmaksızın dahili fonksiyon tipindeki bir değişkene atanabilir. Bu, hem sözleşmelerin hem de kütüphanelerin özel, dahili ve genel fonksiyonlarını ve ayrıca ücretsiz fonksiyonlarını içerir. harici fonksiyon türleri ise yalnızca genel (public) ve harici (external) sözleşme fonksiyonlarıyla uyumludur. Kitaplıklar, bir delegatecall gerektirdikleri ve *seçicileri için farklı bir ABI kuralı* kullandıkları için hariç tutulur. Arayüzlerde bildirilen fonksiyonların tanımları yoktur, bu nedenle onlara işaret etmek de bir anlam ifade etmez.

Üyeler:

Harici (veya genel) fonksiyonlar aşağıdaki üyelere sahiptir:

- .address fonksiyonun sözleşmesinin adresini döndürür.
- .selector, *BI işlev seçicisini* döndürür

Not: Harici (veya genel) fonksiyonlar, .gas(uint) ve .value(uint) ek üyelerine sahiptiler. Bunlar Solidity 0.6.2'de tartışmaya açıldı ve Solidity 0.7.0'da kaldırıldı. Bunun yerine, bir fonksiyona gönderilen gaz miktarını veya wei miktarını belirtmek için sırasıyla {gas: ...} ve {value: ...} kullanın. Daha fazla bilgi için bkz. *External Function Calls* .

Üyelerin nasıl kullanılacağını gösteren örnek:

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

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

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

Dahili fonksiyon türlerinin nasıl kullanılacağını gösteren örnek:

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

library ArrayUtils {
    // aynı kod bağlamının parçası olacakları için dahili fonksiyonlar dahili kütüphane␣
↪fonksiyonlarında kullanılabilir
    function map(uint[] memory self, function (uint) pure returns (uint) f)
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

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

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


contract Pyramid {
    using ArrayUtils for *;

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

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

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

Harici işlev türlerini kullanan başka bir örnek:

```
// SPDX-License-Identifier: GPL-3.0
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
pragma solidity >=0.4.22 <0.9.0;


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

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

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

    function reply(uint requestID, uint response) public {
        // Cevabın güvenilir bir kaynaktan gelip gelmediği kontrol edilir
        requests[requestID].callback(response);
    }
}


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

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

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

Not: Lambda veya satır içi işlevler planlanmıştır ancak henüz desteklenmemektedir.

3.6.2 Referans Türleri

Referans türünün değerleri, birden çok farklı adla değiştirilebilir. Bunu, bir değer türü değişkeni kullanıldığında bağımsız bir kopya aldığınız değer türleriyle karşılaştırın. Bu nedenle referans türleri, değer türlerinden daha dikkatli ele alınmalıdır. Şu anda referans türleri yapılar, diziler ve eşlemelerden oluşmaktadır. Bir referans türü kullanıyorsanız, her zaman türün depolandığı veri alanını açıkça sağlamanız gerekir: `memory` (ömrü, harici bir fonksiyon çağrısıyla sınırlıdır), `storage` (durum değişkenlerinin ömrünün, bir sözleşmenin ömrüyle sınırlı olduğu durumlarda saklanır) veya `calldata` (fonksiyon argümanlarını içeren özel veri konumu).

Veri konumunu değiştiren bir atama veya tür dönüştürme işlemi her zaman otomatik bir kopyalama işlemine neden olurken, aynı veri konumu içindeki atamalar yalnızca bazı durumlarda depolama türleri için kopyalanır.

Veri Konumu

Her referans türünün, nerede depolandığı hakkında "veri konumu" olan ek bir açıklaması vardır. Üç veri konumu vardır: `memory`, `storage` ve `calldata`. Çağrı verileri (calldata), fonksiyon bağımsız değişkenlerinin depolandığı ve çoğunlukla bellek gibi davrandığı, değiştirilemeyen, kalıcı olmayan bir alandır.

Not: Yapabiliyorsanız, veri konumu olarak `calldata` kullanmayı deneyin, çünkü bu kopyaları önler ve ayrıca verilerin değiştirilememesini sağlar. "calldata" veri konumuna sahip diziler ve yapılar da fonksiyonlarla döndürülebilir, ancak bu türlerin atanması mümkün değildir.

Not: 0.6.9 sürümünden önce, referans türü argümanlar için veri konumu, harici fonksiyonlarda `calldata` , genel fonksiyonlarda `memory` ve dahili ve özel fonksiyonlarda `memory` veya `storage` ile sınırlıydı. . Artık `memory` e ve `calldata` ya, görünürlüklerinden bağımsız olarak tüm fonksiyonlarda izin verilir.

Not: 0.5.0 sürümünden önce, veri konumu atlanabilir ve değişkenin türüne, fonksiyon türüne vb. bağlı olarak varsayılan olarak farklı konumlara atanırdı, ancak tüm karmaşık türler şimdi açık bir veri konumu vermelidir.

Veri Konumu ve Atama Davranışı

Veri konumları yalnızca verilerin kalıcılığı için değil, aynı zamanda atamaların anlamı için de önemlidir:

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

- `storage` ve `memory` (veya `calldata`) arasındaki atamalar her zaman bağımsız bir kopya oluşturur.
- `memory``den ``memory``ye (bellekten belleğe) yapılan atamalar yalnızca referans oluşturur. Bu, bir bellek değişkeninde (``memory`) yapılan değişikliklerin aynı verilere atıfta bulunan diğer tüm bellek değişkenlerinde de görülebileceği anlamına gelir.
- `storage` dan (depolamadan), **local** (yerel) depolama değişkenine yapılan atamalar da yalnızca bir referans atar.
- Diğer tüm atamalar `storage` a her zaman kopyalanır. Bu duruma örnek olarak, yerel değişkenin kendisi yalnızca bir başvuru olsa bile, durum değişkenlerine veya depolama yapısı türünün yerel değişkenlerinin üyelerine atamalar verilebilir.

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
contract C {
    // x'in veri konumu depolamadır.
    // Bu, veri konumunun atlanabileceği tek yerdir.
    uint[] x;

    // memoryArray öğesinin veri konumu bellektir.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // çalışır ve tüm diziyi depoya kopyalar
        uint[] storage y = x; // çalışır ve bir işaretçi atar. y'nin veri konumu␣
↪depolamadır
        y[7]; // 8. öğeyi döndürür
        y.pop(); // x'i y ile değiştirir
        delete x; // diziyi temizler, ayrıca y'yi değiştirir
        // Aşağıdakiler çalışmıyor; depolamada yeni bir geçici adsız dizi oluşturması␣
↪gerekir, ancak depolama "statik olarak" tahsis edilir: /
        // y = memoryArray;
        // İşaretçiyi "sıfırlayacağı" için bu da işe yaramaz, ancak işaret edebileceği␣
↪mantıklı bir konum yoktur.
        // delete y;
        g(x); // g'yi çağırır, x'e bir referans verir
        h(x); // h'yi çağırır ve bellekte bağımsız, geçici bir kopya oluşturur
    }

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

Diziler

Diziler, derleme zamanında sabit bir boyuta sahip olabilir veya dinamik bir boyuta sahip olabilir.

Sabit boyutlu bir dizinin türü `k` ve öğe türü `T`, `T[k]` olarak yazılır ve dinamik boyut dizisi `T[]` olarak yazılır.

Örneğin, `uint` in 5 dinamik dizisinden oluşan bir dizi `uint[][5]` olarak yazılır. Notasyon, diğer bazı dillere kıyasla tersine çevrilir. Solidity'de, `X[3]` her zaman `X` türünde üç öğe içeren bir dizidir, `X` in kendisi bir dizi olsa bile. C gibi diğer dillerde durum böyle değildir.

Endeksler sıfır tabanlıdır ve erişim bildirimin tersi yönündedir.

Örneğin, bir `uint[][5] memory x` değişkeniniz varsa, `x[2][6]` kullanarak üçüncü dinamik dizi içerisindeki yedinci `uint`'e erişirsiniz ve üçüncü dinamik diziye erişmek için `x[2]` kullanırsınız. Yine, aynı zamanda bir dizi de olabilen bir `T` türü için bir `T[5] a` diziniz varsa, o zaman `a[2]` her zaman `T` tipine sahiptir.

Dizi öğeleri, eşleme veya yapı dahil olmak üzere herhangi bir türde olabilir. Türler için genel kısıtlamalar geçerlidir, çünkü eşlemeler yalnızca "depolama" veri konumunda depolanabilir ve genel olarak görülebilen fonksiyonlar *ABI types* olan parametrelere ihtiyaç duyar.

Durum değişkeni dizilerini `public` olarak işaretlemek ve Solidity'nin bir *alıcı* oluşturmasını sağlamak mümkündür. Sayısal dizin, alıcı için gerekli bir parametre haline gelir.

Sonunu aşan bir diziye erişmek, başarısız bir onaylamaya neden olur. `.push()` ve `.push(value)` yöntemleri dizinin sonuna yeni bir öğe eklemek için kullanılabilir; burada `.push()` sıfır başlatılmış bir öğe ekler ve ona bir referans döndürür.

Diziler olarak `bytes` ve `string`

`bytes` ve `string` türündeki değişkenler özel dizilerdir. `bytes` türü `bytes1[]` ile benzerdir, ancak çağrı verileri ve bellekte sıkıca paketlenmiştir. `string`, `bytes` değerine eşittir ancak uzunluk veya dizin erişimine izin vermez.

Solidity'nin string işleme fonksiyonları yoktur, ancak üçüncü taraf string kitaplıkları vardır. Ayrıca, `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` kullanarak iki dizgiyi keccak256-hash ile karşılaştırabilir ve `string.concat(s1, s2)` kullanarak iki dizgiyi birleştirebilirsiniz.

`bytes1[]` yerine `bytes` kullanmalısınız çünkü daha ucuzdur, çünkü `memory``de ``bytes1[]` kullanmak, öğeler arasında 31 dolgu bayt ekler. `storage`"da, sıkı paketleme nedeniyle dolgu bulunmadığına dikkat edin, bkz. *bayt ve string*. Genel bir kural olarak, rastgele uzunluktaki ham bayt verileri için `bytes` ve rastgele uzunluktaki string (UTF-8) verileri için `string` kullanın. Uzunluğu belirli bir bayt sayısıyla sınırlayabiliyorsanız, her zaman `bytes1` ile `bytes32` arasındaki değer türlerinden birini kullanın çünkü bunlar çok daha ucuzdur.

Not: `s` stringinin bayt temsiline erişmek istiyorsanız, `bytes(s).length` / `bytes(s)[7] = 'x';` yapısını kullanın. Tek tek karakterlere değil, UTF-8 temsilinin düşük seviyeli baytlarına eriştiğinizi unutmayın.

`bytes.concat` ve `string.concat` Fonksiyonları

`string.concat` kullanarak rastgele sayıda `string` değerini birleştirebilirsiniz. Fonksiyon, bağımsız değişkenlerin içeriğini doldurmadan içeren tek bir `string memory` dizisi döndürür. Örtülü olarak `string` e dönüştürülemeyen diğer türlerin parametrelerini kullanmak istiyorsanız, önce bunları `string` e dönüştürmeniz gerekir.

Benzer şekilde, `bytes.concat` fonksiyonu, rastgele sayıda `bytes` veya `bytes1 ... bytes32` değerlerini birleştirebilir. Fonksiyon, bağımsız değişkenlerin içeriğini doldurmadan içeren tek bir `bytes memory` dizisi döndürür. String parametreleri veya örtük olarak `bytes` a dönüştürülemeyen diğer türleri kullanmak istiyorsanız, önce bunları `bytes` veya `bytes1` /.../ `bytes32` ye dönüştürmeniz gerekir.

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

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

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

`string.concat` ı veya `bytes.concat` ı, argüman olmadan çağırırsanız, boş bir dizi döndürürler.

Bellek Dizilerini Ayırma

Dinamik uzunluktaki bellek dizileri `new` operatörü kullanılarak oluşturulabilir. Depolama dizilerinin aksine, bellek dizilerini yeniden boyutlandırmak **değildir** (ör. `.push` üye fonksiyonları kullanılamaz). Gereken boyutu önceden hesaplamanız veya yeni bir bellek dizisi oluşturmanız ve her öğeyi kopyalamanız gerekir.

Solidity'deki tüm değişkenler gibi, yeni tahsis edilen dizilerin öğeleri her zaman *varsayılan değer* ile başlatılır.

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

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

Dizi İfadeleri

Bir dizi ifadesi, köşeli parantezler (`[...]`) içine alınmış bir veya daha fazla ifadenin virgülle ayrılmış bir listesidir. Örneğin `[1, a, f(3)]`. Dizi ifadesinin türü şu şekilde belirlenir:

Her zaman uzunluğu ifade sayısı olan statik olarak boyutlandırılmış bir bellek dizisidir.

Dizinin temel türü, diğer tüm ifadelerin dolaylı olarak kendisine dönüştürülebileceği şekilde listedeki ilk ifadenin türüdür. Bu mümkün değilse bir tür hatasıdır.

Tüm öğelerin dönüştürülebileceği bir türün olması yeterli değildir. Öğelerden birinin bu türden olması gerekir.

Aşağıdaki örnekte, `[1, 2, 3]` türü `uint8[3] memory` dir, çünkü bu sabitlerin her birinin türü `uint8` dir. Sonucun `uint[3] memory` türünde olmasını istiyorsanız, ilk öğeyi `uint` e dönüştürmeniz gerekir.

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

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

Birinci ifadenin türü `uint8` iken ikincinin türü `int8` olduğundan ve bunlar örtük olarak birbirine dönüştürülemediğinden `[1, -1]` dizisi ifadesi geçersizdir. Çalışması için örneğin `[int8(1), -1]` kullanabilirsiniz.

Farklı türdeki sabit boyutlu bellek dizileri birbirine dönüştürülemediğinden (temel türler yapabilse bile), iki boyutlu dizi ifadelerini kullanmak istiyorsanız, her zaman ortak bir temel türü açıkça belirtmeniz gerekir:

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

contract C {
    function f() public pure returns (uint24[2][4] memory) {
        uint24[2][4] memory x = [[uint24(0x1), 1], [0xffffff, 2], [uint24(0xff), 3],␣
↪[uint24(0xffff), 4]];
        // Aşağıdakiler çalışmaz, çünkü bazı iç diziler doğru tipte değildir.
        // uint[2][4] memory x = [[0x1, 1], [0xffffff, 2], [0xff, 3], [0xffff, 4]];
        return x;
    }
}
```

Sabit boyutlu bellek dizileri, dinamik olarak boyutlandırılmış bellek dizilerine atanamaz, yani aşağıdakiler mümkün değildir:

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

// Bu derleme gerçekleşmeyecek.
contract C {
    function f() public {
        // Sonraki satır bir tür hatası oluşturur çünkü uint[3] belleği, uint[]␣
↪belleğine dönüştürülemez.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

İleride bu kısıtlamanın kaldırılması planlanıyor ancak dizilerin ABI'dan geçirilme şekli nedeniyle bazı komplikasyonlar yaratıyor.

Dinamik olarak boyutlandırılmış dizileri başlatmak istiyorsanız, tek tek öğeleri atamanız gerekir:

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

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

Dizi Üyeleri

length:
Diziler, eleman sayısını içeren bir `length` (uzunluk) üyesine sahiptir.Bellek dizilerinin uzunluğu, oluşturulduktan sonra sabittir (ancak dinamiktir, yani çalışma zamanı parametrelerine bağlı olabilir).

push():
Dinamik depolama dizileri ve `bytes` (`string` değil), dizinin sonuna sıfır başlatılmış bir öğe eklemek için kullanabileceğiniz `push()` adlı üye fonksiyonuna sahiptir. Öğeye bir başvuru döndürür, böylece `x.push().t = 2` veya `x.push() = b` gibi kullanılabilir.

push(x):
Dinamik depolama dizileri ve `bytes` (`string` değil), dizinin sonuna belirli bir öğeyi eklemek için kullanabileceğiniz `push(x)` adlı bir üye fonksiyonuna sahiptir. Fonksiyon hiçbir şey döndürmez.

pop():
Dinamik depolama dizileri ve `bytes` (`string` değil), dizinin sonundan bir öğeyi kaldırmak için kullanabileceğiniz `pop()` adlı bir üye fonksiyonuna sahiptir. Bu ayrıca kaldırılan öğede örtük olarak *delete* öğesini çağırır. Fonksiyon hiçbir şey döndürmez.

Not: `pop()` kullanarak uzunluk azaltılırken kaldırılan öğenin "boyutuna" bağlı olarak bir ücreti varken, bir depolama dizisinin uzunluğunu `push()` çağırarak artırmanın sabit gaz maliyetleri vardır çünkü başlarken depolama sıfırdır. Kaldırılan öğe bir diziyse, çok maliyetli olabilir, çünkü *delete* çağrılmasına benzer şekilde kaldırılan öğelerin açıkça temizlenmesini içerir.

Not: Dizi dizilerini harici (genel yerine) fonksiyonlarda kullanmak için ABI kodlayıcı v2'yi etkinleştirmeniz gerekir.

Not: "Byzantium" öncesi EVM sürümlerinde fonksiyon çağrılarından dönen dinamik dizilere erişim mümkün değildi. Dinamik diziler döndüren fonksiyonları çağırırsanız, Byzantium moduna ayarlanmış bir EVM kullandığınızdan emin olun.

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

contract ArrayContract {
    uint[2**20] aLotOfIntegers;
    // Aşağıdakilerin bir çift dinamik dizi değil, dinamik bir çift dizisi (yani, iki␣
↪uzunluktaki sabit boyutlu diziler) olduğuna dikkat edin.
    // Bu nedenle, T[], T'nin kendisi bir dizi olsa bile, her zaman dinamik bir T␣
↪dizisidir.
    // Tüm durum değişkenleri için veri konumu depolamadır.
    bool[2][] pairsOfFlags;

    // newPairs bellekte saklanır - tek olasılık
    // açık (public) sözleşme fonksiyonları argümanları için
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // bir depolama dizisine atama, "``newPairs``in bir kopyasını gerçekleştirir ve␣
↪``pairsOfFlags`` dizisinin tamamının yerini alır.
        pairsOfFlags = newPairs;
    }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

    function f(uint[] memory c) public {
        // ``g`` içindeki ``s`` referansını saklar
        StructType storage g = s;
        // ayrıca ``s.moreInfo``yu da değiştirir.
        g.moreInfo = 2;
        // ``g.contents`` yerel bir değişken değil, yerel bir değişkenin üyesi olduğu
↪için bir kopya atar.
        g.contents = c;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // var olmayan bir dizine erişim bir istisna atar
        pairsOfFlags[index][0] = flagA;
        pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // bir dizinin uzunluğunu değiştirmenin tek yolu push ve pop kullanmaktır
        if (newSize < pairsOfFlags.length) {
            while (pairsOfFlags.length > newSize)
                pairsOfFlags.pop();
        } else if (newSize > pairsOfFlags.length) {
            while (pairsOfFlags.length < newSize)
                pairsOfFlags.push();
        }
    }

    function clear() public {
        // bunlar dizileri tamamen temizler
        delete pairsOfFlags;
        delete aLotOfIntegers;
        // identical effect here
        pairsOfFlags = new bool[2][](0);
    }

    bytes byteData;

    function byteArrays(bytes memory data) public {
        // bayt dizileri ("bayts"), dolgu olmadan depolandıkları için farklıdır, ancak
↪"uint8[]" ile aynı şekilde ele alınabilirler.
        byteData = data;
        for (uint i = 0; i < 7; i++)
            byteData.push();
        byteData[3] = 0x08;
        delete byteData[2];
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }

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

    function createMemoryArray(uint size) public pure returns (bytes memory) {
        // Dinamik bellek dizileri `new` kullanılarak oluşturulur:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        // Satır içi diziler her zaman statik olarak boyutlandırılmıştır ve yalnızca␣
↪değişmez değerler kullanıyorsanız, en az bir tür sağlamanız gerekir.
        arrayOfPairs[0] = [uint(1), 2];

        // Dinamik bir bayt dizisi oluşturun:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = bytes1(uint8(i));
        return b;
    }
}
```

Depolama Dizisi Öğelerine Sarkan Referanslar

Depolama dizileriyle çalışırken, sarkan referanslardan kaçınmaya özen göstermeniz gerekir. Sarkan referans, artık var olmayan veya referans güncellenmeden taşınmış bir şeye işaret eden bir referanstır. Örneğin, bir dizi öğesine bir başvuruyu yerel bir değişkende saklarsanız ve ardından içeren diziden `.pop()` depolarsanız, sarkan bir başvuru oluşabilir:

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

contract C {
    uint[][] s;

    function f() public {
        // s öğesinin son dizi öğesine bir işaretçi depolar.
        uint[] storage ptr = s[s.length - 1];
        // s öğesinin son dizi öğesini kaldırır.
        s.pop();
        // Artık dizi içinde olmayan dizi öğesine yazar.
        ptr.push(0x42);
        // Şimdi ``s`` öğesine yeni bir öğe eklemek boş bir dizi eklemez, ancak öğe␣
↪olarak ``0x42`` olan 1 uzunluğunda bir diziyle sonuçlanır.
        s.push();
        assert(s[s.length - 1][0] == 0x42);
    }
}
```

`ptr.push(0x42)` içindeki yazma, `ptr``nin artık geçerli bir ``s` öğesini ifade etmemesine rağmen **dönmeyecek**. Derleyici kullanılmayan depolamanın her zaman sıfırlandığını varsaydığından, sonraki bir `s.push()`, depo-

lamaya açıkça sıfır yazmaz, bu nedenle `push()``dan sonraki ``s``nin son öğesi ``1` uzunluğa sahip ve ilk öğesi olarak `0x42` içeriyor.

Solidity'nin, depolamadaki değer türlerine referansların bildirilmesine izin vermediğini unutmayın. Bu tür açık sarkan başvurular, iç içe geçmiş başvuru türleriyle sınırlıdır. Ancak, tanımlama grubu atamalarında karmaşık ifadeler kullanılırken geçici olarak sarkan referanslar da oluşabilir:

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

contract C {
    uint[] s;
    uint[] t;
    constructor() {
        // Bazı başlangıç değerlerini depolama dizilerine aktarın.
        s.push(0x07);
        t.push(0x03);
    }

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

    function f() public returns (uint[] memory) {
        // Aşağıdakiler ilk önce ``s.push()`` öğesini dizin 1'deki yeni bir öğeye yapılan␣
↪bir başvuruya göre değerlendirecektir.
        // Daha sonra, ``g`` çağrısı bu yeni öğeyi açar ve en soldaki demet öğesinin␣
↪sarkan bir referans haline gelmesine neden olur.
        // Atama hala devam ediyor ve ``s`` veri alanının dışına yazacak.
        (s.push(), g()[0]) = (0x42, 0x17);
        // Daha sonra ``s``ye basılması (push edilmesi/pushlanması), önceki ifade␣
↪tarafından yazılan değeri ortaya çıkaracaktır, yani bu fonksiyonun sonunda "s"nin son␣
↪elemanı "0x42" değerine sahip olacaktır.
        s.push();
        return s;
    }
}
```

Her ifade için depolamaya yalnızca bir kez atama yapmak ve atamanın sol tarafında karmaşık ifadelerden kaçınmak her zaman daha güvenlidir.

Bir bayt dizisindeki bir `.push()`, *depolamada kısa düzenden uzun düzene* geçebileceğinden, `bytes` dizilerinin öğelerine yapılan başvurularla uğraşırken özellikle dikkatli olmanız gerekir.

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

// Bu bir uyarı bildirir
contract C {
    bytes x = "0123456789012345678901234567890";

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }
}
```

Burada, ilk `x.push()` değerlendirildiğinde, `x` hala kısa düzende saklanır, bu nedenle `x.push()`, `x``in ilk depolama yuvasındaki bir öğeye bir referans döndürür. Ancak, ikinci ``x.push()` bayt dizisini büyük düzene geçirir. Şimdi `x.push()` öğesinin atıfta bulunduğu öğe dizinin veri alanındayken, başvuru hala uzunluk alanının bir parçası olan orijinal konumunu işaret eder ve atama, `x` dizisinin uzunluğunu etkin bir şekilde bozar.

Güvende olmak için, tek bir atama sırasında bayt dizilerini yalnızca en fazla bir öğeyle büyütün ve aynı ifadede diziye aynı anda dizin erişimi yapmayın.

Yukarıda, derleyicinin geçerli sürümündeki sarkan depolama referanslarının davranışı açıklanırken, sarkan referanslara sahip herhangi bir kodun *tanımsız davranışa* sahip olduğu düşünülmelidir. Özellikle bu, derleyicinin gelecekteki herhangi bir sürümünün, sarkan referanslar içeren kodun davranışını değiştirebileceği anlamına gelir.

Kodunuzda sarkan referanslardan kaçındığınızdan emin olun!

Dizi Dilimleri

Dizi dilimleri, bir dizinin bitişik kısmındaki bir görünümdür. `x[start:end]` olarak yazılırlar, burada `start` ve `end` , uint256 türüyle sonuçlanan (veya dolaylı olarak ona dönüştürülebilir) ifadelerdir. Dilimin ilk öğesi `x[start]` ve son öğesi `x[end - 1]` dir.

`start`, `end``den büyükse veya ``end`, dizinin uzunluğundan büyükse, bir istisna atılır.

Hem `start` hem de `end` isteğe bağlıdır: `start` varsayılanları `0` ve `end` varsayılanları dizinin uzunluğudur.

Dizi dilimlerinin herhangi bir üyesi yoktur. Altta yatan türdeki dizilere örtük olarak dönüştürülebilirler ve dizin erişimini desteklerler. Dizin erişimi, temel alınan dizide mutlak değil, dilimin başlangıcına göredir.

Dizi dilimlerinin bir tür adı yoktur, yani hiçbir değişken tür olarak dizi dilimlerine sahip olamaz, yalnızca ara ifadelerde bulunurlar.

Not: Şu anda dizi dilimleri yalnızca çağrı verisi dizileri için uygulanmaktadır.

Dizi dilimleri, fonksiyon parametrelerinde iletilen ikincil verilerin ABI kodunu çözmek için kullanışlıdır:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.5 <0.9.0;
contract Proxy {
    /// @dev, proxy (vekil) tarafından yönetilen alıcı (client) sözleşmesinin adresi,␣
↪yani bu sözleşme
    address client;

    constructor(address client_) {
        client = client_;
    }

    /// Adres bağımsız değişkeninde temel doğrulama yaptıktan sonra istemci tarafından␣
↪uygulanan "setOwner(address)" çağrısını yönlendirin.
    function forward(bytes calldata payload) external {
        bytes4 sig = bytes4(payload[:4]);
        // Kesme davranışı nedeniyle, bytes4(payload) aynı şekilde çalışır.
        // bytes4 sig = bytes4(payload);
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        if (sig == bytes4(keccak256("setOwner(address)"))) {
            address owner = abi.decode(payload[4:], (address));
            require(owner != address(0), "Address of owner cannot be zero.");
        }
        (bool status,) = client.delegatecall(payload);
        require(status, "Forwarded call failed.");
    }
}
```

Yapılar

Solidity, aşağıdaki örnekte gösterildiği gibi, yapılar biçiminde yeni türleri tanımlamanın bir yolunu sağlar:

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

// İki alanlı yeni bir tür tanımlar.
// Bir sözleşmenin dışında bir yapı bildirmek, birden fazla sözleşme tarafından␣
↪paylaşılmasına izin verir.
// Burada, bu gerçekten gerekli değil.
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
    // Yapılar, sözleşmelerin içinde de tanımlanabilir, bu da onları yalnızca orada ve␣
↪türetilmiş sözleşmelerde görünür kılar.
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

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

    function newCampaign(address payable beneficiary, uint goal) public returns (uint␣
↪campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // "campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)" kullanamayız␣
↪çünkü sağ taraf bir eşleme içeren bir bellek yapısı "Campaign" oluşturur.
        Campaign storage c = campaigns[campaignID];
        c.beneficiary = beneficiary;
        c.fundingGoal = goal;
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        // Verilen değerlerle başlatılan yeni bir geçici bellek yapısı oluşturur ve bunu␣
↪depoya kopyalar.
        // Başlatmak için Funder(msg.sender, msg.value) öğesini de kullanabileceğinizi␣
↪unutmayın.
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

Sözleşme, bir kitle fonlaması sözleşmesinin tam işlevselliğini sağlamaz, ancak yapıları anlamak için gerekli temel kavramları içerir. Yapı türleri eşlemeler ve diziler içinde kullanılabilir ve kendileri eşlemeler ve diziler içerebilir.

Bir yapının kendi türünden bir üye içermesi mümkün değildir, ancak yapının kendisi bir eşleme üyesinin değer türü olabilir veya kendi türünde dinamik olarak boyutlandırılmış bir dizi içerebilir. Yapının boyutunun sonlu olması gerektiğinden bu kısıtlama gereklidir.

Tüm fonksiyonlarda, veri konumu `storage` olan yerel bir değişkene bir yapı türünün nasıl atandığına dikkat edin. Bu, yapıyı kopyalamaz, ancak yalnızca bir referansı saklar, böylece yerel değişkenin üyelerine yapılan atamalar aslında duruma yazılır.

Not: Solidity 0.7.0'a kadar, yalnızca depolama türlerinin üyelerini (ör. eşlemeler) içeren bellek yapılarına izin veriliyordu ve `campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)` gibi atamalar işe yarıyordu ve bunları sessizce atlıyordu.

3.6.3 Eşleme Türleri

Eşleme türleri `mapping(KeyType => ValueType)` sözdizimi yapısını kullanır ve eşleme türünün değişlenleri, `mapping(KeyType => ValueType) VariableName` sözdizimi kullanılarak bildirilir.

`KeyType`, herhangi bir yerleşik değer türü, `bytes`, `string`, herhangi bir sözleşme ya da numaralandırma türü olabilir. Eşlemeler, yapılar veya dizi türleri gibi diğer kullanıcı tanımlı veya karmaşık türlere izin verilmez. `ValueType`, eşlemeleri, dizileri ve yapıları içeren herhangi bir tür olabilir.

Eşlemeleri, olası her anahtarın var olduğu ve bir türün *varsayılan değeri* olan bayt temsilinin tamamı sıfır olan bir değere eşlendiği şekilde sanal olarak başlatılan karma tablolar olarak düşünebilirsiniz. Benzerlik burada sona eriyor, anahtar veriler bir eşlemede saklanmıyor, değeri aramak için yalnızca `keccak256` karma değeri kullanılıyor.

Bu nedenle, eşlemelerin bir uzunluğu veya ayarlanan bir anahtar veya değer kavramı yoktur ve bu nedenle atanan anahtarlarla ilgili ek bilgi olmadan silinemezler (bkz. *Mappingleri Temizleme*).

Eşlemeler yalnızca `storage` veri konumuna sahip olabilir ve bu nedenle, fonksiyonlardaki depolama referans türleri olarak veya kitaplık fonksiyonları için parametreler olarak durum değişkenleri için izin verilir. Bunlar, genel olarak

görülebilen sözleşme fonksiyonlarının parametreleri veya dönüş parametreleri olarak kullanılamazlar. Bu kısıtlamalar, eşlemeler içeren diziler ve yapılar için de geçerlidir.

Eşleme türündeki durum değişkenlerini `public` olarak işaretleyebilirsiniz ve Solidity sizin için bir *alıcı* oluşturur. `KeyType`, alıcı için bir parametre olur. `ValueType` bir değer türü veya yapıysa, alıcı `ValueType` değerini döndürür. `ValueType` bir dizi veya eşleme ise, alıcının her bir `KeyType` için yinelemeli olarak bir parametresi vardır.

Aşağıdaki örnekte, `MappingExample` sözleşmesi, anahtar türü bir `address` olan genel bir `balances` eşlemesini ve bir Ethereum adresini işaretsiz bir tamsayı değerine eşleyen bir `uint` değer türünü tanımlar. `uint` bir değer türü olduğundan, alıcı, belirtilen adreste değeri döndüren `MappingUser` sözleşmesinde görebileceğiniz türle eşleşen bir değer döndürür.

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

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

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

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

Aşağıdaki örnek, bir ERC20 tokenin basitleştirilmiş bir versiyonudur. `_allowances`, başka bir eşleme türü içindeki eşleme türüne bir örnektir.

Aşağıdaki örnekte, başka birinin hesabınızdan çekmesine izin verilen tutarı kaydetmek için `_allowances` kullanılmıştır.

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

contract MappingExample {

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

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

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

    function transferFrom(address sender, address recipient, uint256 amount) public␣
↪returns (bool) {
        require(_allowances[sender][msg.sender] >= amount, "ERC20: Allowance not high␣
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪enough.");
        _allowances[sender][msg.sender] -= amount;
        _transfer(sender, recipient, amount);
        return true;
    }

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

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

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

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

Yinelenebilir Eşlemeler

Eşlemeleri yineleyemezsiniz, yani anahtarlarını numaralandıramazsınız. Yine de bunların üzerine bir veri yapısı uygulamak ve bunun üzerinde yineleme yapmak mümkündür. Örneğin, aşağıdaki kod, `User` sözleşmesinin daha sonra veri eklediği bir `IterableMapping` kitaplığı uygular ve `sum` fonksiyonu tüm değerleri toplamak için yinelenir.

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

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

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

type Iterator is uint;

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool␣
↪replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            return true;
        else {
            keyIndex = self.keys.length;
            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

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

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

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

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

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

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

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
// Nasıl kullanılır
contract User {
    // Sadece verilerimizi tutan bir yapı.
    itmap data;
    // Veri türüne kitaplık fonksiyonlarını uygulayın.
    using IterableMapping for itmap;

    // Bir şeyleri ekle
    function insert(uint k, uint v) public returns (uint size) {
        // Bu, IterableMapping.insert(data, k, v)'yi çağırır.
        data.insert(k, v);
        // Yapının üyelerine hala erişebiliriz,
        // ama bunlarla uğraşmamaya özen göstermeliyiz.
        return data.size;
    }

    // Depolanan tüm verilerin toplamını hesaplar.
    function sum() public view returns (uint s) {
        for (
            Iterator i = data.iterateStart();
            data.iterateValid(i);
            i = data.iterateNext(i)
        ) {
            (, uint value) = data.iterateGet(i);
            s += value;
        }
    }
}
```

3.6.4 Operatörler

Aritmetik operatörler ve bit operatörleri, iki işlenen aynı türe sahip olmasa bile uygulanabilir. Örneğin, `y = x + z` yi hesaplayabilirsiniz, burada `x` bir `uint8` dir ve `z` nin türü `uint32` dir. Bu durumlarda, işlemin hesaplandığı türü (taşma durumunda bu önemlidir) ve operatörün sonucunun türünü belirlemek için aşağıdaki mekanizma kullanılacaktır:

1. **Sağ işlenenin türü dolaylı olarak sol işlenenin türüne dönüştürülebiliyorsa,**
   sol işlenenin türünü kullanın.
2. **Sol işlenenin türü dolaylı olarak sağ işlenenin türüne dönüştürülebiliyorsa,**
   sağ işlenenin türünü kullanın,
3. İki seçenek de uygulanamıyorsa işleme izin verilmez.

İşlenenlerden birinin *gerçek sayı* olması durumunda, ilk önce değeri tutabilen en küçük tür olan "mobil türe" dönüştürülür (aynı bit genişliğindeki işaretsiz türler, işaretli türlerden "daha küçük" olarak kabul edilir) .

Her ikisi de gerçek sayıysa, işlem keyfi bir kesinlikle hesaplanır.

Operatörün sonuç türü, sonucun her zaman `bool` olduğu karşılaştırma operatörleri dışında, işlemin gerçekleştirildiği türle aynıdır.

** (üs alma), << ve >> operatörleri, işlem ve sonuç için sol işlenenin türünü kullanır.

Üçlü Operatör

Üçlü operatör, <expression> ? <trueExpression> : <falseExpression> formunda bulunan ifadelerin açıklanmasında kullanılır. Ana <expression> değerlendirmesinin sonucuna bağlı olarak verilen son iki ifadeden birini değerlendirir. <expression> "doğru" olarak değerlendirilirse, <trueExpression> olarak sayılır, aksi takdirde <falseExpression> olarak sayılır.

Üçlü operatörün sonucu, tüm işlenenleri rasyonel sayı değişmezleri olsa bile, bir rasyonel sayı türüne sahip değildir. Sonuç türü, iki işlenenin türlerinden yukarıdakiyle aynı şekilde belirlenir, gerekirse ilk önce mobil türlerine dönüştürülür.

Sonuç olarak, `255 + (true ? 1 : 0)` işlemi, aritmetik taşma nedeniyle geri döndürülecektir (revert edilecektir). Bunun nedeni, `(true ? 1 : 0)` ifadesinin `uint8` türünde olmasıdır.Bu, eklemenin `uint8` içinde gerçekleştirilmesini zorunlu kılıyor ve 256'nın bu tür için izin verilen aralığı aşıyor.

Diğer bir sonuç da, `1.5 + 1.5` gibi bir ifadenin geçerli olduğu, ancak `1.5 + (true ? 1.5 : 2.5)` olmadığıdır. Bunun nedeni, birincisinin sınırsız kesinlikle değerlendirilen rasyonel bir ifade olması ve yalnızca nihai değerinin önemli olmasıdır. İkincisi, şu anda izin verilmeyen bir kesirli rasyonel sayının bir tam sayıya dönüştürülmesini içerir.

Bileşik Operatörler ve Artırma/Azaltma Operatörleri

`a` bir LValue ise (yani bir değişken veya atanabilecek bir şey), aşağıdaki operatörler kısayol olarak kullanılabilir:

`a += e`, `a = a + e` ile eşdeğerdir. `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`, `<<=` ve `>>=` buna göre tanımlanır. `a++` ve `a--`, `a += 1 / a -= 1` ile eşdeğerdir, ancak ifadenin kendisi hala önceki `a` değerine sahiptir. Buna karşılık, `--a` ve `++a`, `a` üzerinde aynı etkiye sahiptir ancak değişiklikten sonra değeri döndürür.

silmek

`delete a`, türün başlangıç değerini a``ya atar. Yani, tamsayılar için ``a = 0` ile eşdeğerdir, ancak sıfır uzunlukta dinamik bir dizi veya tüm öğeleri başlangıç değerlerine ayarlanmış aynı uzunlukta statik bir dizi atadığı dizilerde de kullanılabilir.

`delete a[x]`, dizinin `x` dizinindeki öğeyi siler ve diğer tüm öğelere ve dizinin uzunluğuna dokunmadan bırakır. Bu özellikle dizide bir boşluk bırakıldığı anlamına gelir. Öğeleri kaldırmayı planlıyorsanız, *eşleme* yapmak muhtemelen daha iyi bir seçimdir.

Yapılar (structs) için, tüm üyelerin sıfırlandığı bir yapı atar. Başka bir deyişle, `a` nın `delete a` dan sonraki değeri, `a` nın atama olmadan bildirilmesiyle aynıdır:

`delete` fonksiyonunun eşlemeler üzerinde hiçbir etkisi yoktur (çünkü eşlemelerin anahtarları rastgele olabilir ve genellikle bilinmez). Bu nedenle, bir yapıyı silerseniz, eşleme olmayan tüm üyeleri sıfırlar ve eşleme olmadıkça üyelere geri döner. Ancak, bireysel anahtarlar ve eşledikleri şey silinebilir: `a` bir eşleme ise, `delete a[x]` , `x` de depolanan değeri siler.

`delete a` nın gerçekten `a` ya atanmış gibi davrandığını, yani `a` da yeni bir nesne depoladığını unutmamak önemlidir. Bu ayrım, `a` referans değişkeni olduğunda görünür: Daha önce atıfta bulunduğu değeri değil, yalnızca `a` nın kendisini sıfırlayacaktır.

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

contract DeleteExample {
    uint data;
    uint[] dataArray;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    function f() public {
        uint x = data;
        delete x; // x'i 0'a ayarlar, verileri etkilemez
        delete data; // verileri 0'a ayarlar, x'i etkilemez
        uint[] storage y = dataArray;
        delete dataArray; // bu, dataArray.length değerini sıfıra ayarlar, ancak uint[]␣
↪karmaşık bir nesne olduğundan,
        // depolama nesnesinin diğer adı olan y da etkilenir.
        // Öte yandan: "delete y" geçerli değildir, çünkü depolama nesnelerine başvuran␣
↪yerel değişkenlere atamalar yalnızca mevcut depolama nesnelerinden yapılabilir.
        assert(y.length == 0);
    }
}
```

Operatörlerin Öncelik Sırası

Aşağıdaki tablo, değerlendirme sırasına göre listelenen operatörler için öncelik sırasını belirtir.

Öncelik	Tanım	Operatör
1	Son ek ile tırma ve azaltma	++, --
	Yeni ifade	new <typename>
	Dizi elamanı görüntüleme	<array>[<index>]
	Üye erişimi	<object>.<member>
	Fonksiyon çağırımı	<func>(<args...>)
	Parantezler	(<statement>)
2	Ön ek ile artırma ve azaltma	++, --
	Tekli çıkarma	-
	Tekli işlemler	delete
	Mantıksal 'DEĞİL'	!
	Bitsel 'DEĞİL'	~
3	Üs alma	**
4	Çarpma, bölme ve mod alma	*, /, %
5	Ekleme ve çıkarma	+, -
6	Bitsel değiştirme operatörleri	<<, >>
7	Bitsel 'VE'	&
8	Bitsel 'Özel veya'	^
9	Bitsel 'YA DA'	|
10	Eşitsizlik operatörleri	<, >, <=, >=
11	Eşitlik operatörleri	==, !=
12	Mantıksal 'VE'	&&
13	Mantıksal 'YA DA'	||
14	Üçlü operatör	<conditional> ? <if-true> : <if-false>
	Atama operatörleri	=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
15	Virgül operatörü	,

3.6.5 Temel Türler Arası Dönüşümler

Örtülü Dönüşümler

Örtülü tür dönüşümü, argümanları fonksiyonlara iletme ya da operatör atamaları sırasında, derleyici tarafından otomatik olarak uygulanır. Genel olarak, bilgi kaybı yoksa ve anlamsal açıdan bir sorun yoksa, değer türleri arasında örtülü bir dönüşüm mümkündür.

Örneğin, `uint8` türü, `uint16` türüne ve `int128` türü, `int256` türüne dönüştürülebilirken, `int8` türü `uint256` türüne dönüştürülemez çünkü `uint256`, `-1` gibi değerleri tutamaz.

Bir operatör birbirinden farklı türlere uygulanırsa, derleyici işlenenlerden birini örtük olarak diğerinin türüne dönüştürmeye çalışır (aynısı atamalar için de geçerlidir). Bu, işlemlerin her zaman işlenenlerden birinin türünde gerçekleştirildiği anlamına gelir.

Hangi örtük dönüşümlerin mümkün olduğu hakkında daha fazla ayrıntı için, lütfen türlerle ilgili bölümlere bakın.

Aşağıdaki örnekte, toplamanın işlenenleri olarak `y` ve `z`, aynı türe sahip değildir, fakat `uint8` örtük olarak `uint16` türüne dönüştürülebilirken bunun tersi mümkün değildir. Bu sebeple, `uint16` türünde bir dönüştürme yapılmadan önce `y` türü, `z` türüne dönüştürülür. `y + z` ifadesinden elde edilen tür, `uint16` dır. Toplama işleminin sonucu `uint32` türünde bir değişkene atandığı için, toplama işleminden sonra yeniden örtük dönüştürme gerçekleşir.

```
uint8 y;
uint16 z;
uint32 x = y + z;
```

Açık Dönüşümler

Derleyici örtük dönüştürmeye izin vermiyorsa ancak bir dönüştürmenin işe yarayacağından eminseniz, bazen açık bir tür dönüştürme mümkündür. Bu, beklenmeyen davranışlara neden olabilir ve derleyicinin bazı güvenlik özelliklerini atlamanıza izin verir, bu nedenle sonucun istediğiniz ve beklediğiniz gibi olduğunu test ettiğinizden emin olun!

Negatif değere sahip bir `int` değişkenini, `uint` değişkenine dönüştüren aşağıdaki örneği ele alalım:

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

Bu kod bloğunun sonunda `x`, `0xfffff..fd` (64 adet onaltılık karaker) değerine sahip olacaktır, bu, iki'nin 256 bitlik tümleyen (two's complement) temsili olan -3'tür.

Bir tam sayı, kendisinden daha küçük bir türe açık şekilde dönüştürülürse, daha yüksek dereceli bitler kesilir.

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b, 0x5678 olacaktır
```

Bir tam sayı, kendisinden daha büyük bir türe açık şekilde dönüştürülürse, elde edilen ortak tümleyenin solu yani daha yüksek dereceli ucu doldurulur. Dönüşümün sonucu orijinal tam sayıya eşit olacaktır:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b, 0x00001234 olacaktır
assert(a == b);
```

Sabit boyutlu bayt dizisi türleri, dönüşümler sırasında farklı davranır. Bireysel bayt dizileri olarak düşünülebilirler ve daha küçük bir türe dönüştürmek diziyi kesecektir:

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b, 0x12 olacaktır
```

Sabit boyutlu bir bayt dizisi türü, daha büyük bir türe açıkça dönüştürülürse, elde edilen ortak tümleyen sağ tarafta doldurulur. Sabit bir dizindeki bayt dizisine erişmek, dönüştürmeden önce ve sonra aynı değerle sonuçlanır (dizin hala aralıktaysa):

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

Tamsayılar ve sabit boyutlu bayt dizileri, kesme veya doldurma sırasında farklı davrandığından, tamsayılar ve sabit boyutlu bayt dizileri arasındaki açık dönüştürmelere yalnızca, her ikisi de aynı boyuta sahipse izin verilir. Farklı boyuttaki tamsayılar ve sabit boyutlu bayt dizileri arasında dönüştürmek istiyorsanız, istenen kesme ve doldurma kurallarını açık hale getiren ara dönüşümleri kullanmanız gerekir:

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

`bytes` dizileri ve `bytes` çağrı verisi (calldata) dilimleri, sabit bayt türlerine(`bytes1`/.../`bytes32`) açıkça dönüştürülebilir. Dizinin hedef sabit bayt türünden daha uzun olması durumunda, sonunda kesme gerçekleşir. Dizi hedef türden daha kısaysa, sonunda sıfırlarla doldurulur.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;

contract C {
    bytes s = "abcdefgh";
    function f(bytes calldata c, bytes memory m) public view returns (bytes16, bytes3) {
        require(c.length == 16, "");
        bytes16 b = bytes16(m);  // 'm'in uzunluğu 16'dan büyükse, kesme gerçekleşecektir
        b = bytes16(s);  // sağa genişletilir, sonuç "abcdefgh\0\0\0\0\0\0\0\0" olacaktır
        bytes3 b1 = bytes3(s); // kesilir, b1, "abc"ye eşittir
        b = bytes16(c[:8]);  // sıfırlar ile genişletilir
        return (b, b1);
    }
}
```

3.6.6 İfadeler (Literals) ve Temel Türler Arasındaki Dönüşümler

Tamsayı Türleri

Ondalık ve onaltılık sayı ifadeleri, onu kesmeden temsil edecek kadar büyük herhangi bir tamsayı türüne örtük olarak dönüştürülebilir:

```
uint8 a = 12; // uygun
uint32 b = 1234; // uygun
uint16 c = 0x123456; // hatalı, çünkü 0x3456 olacak şekilde kesilmek zorundadır
```

Not: 0.8.0 sürümünden önce, herhangi bir ondalık veya onaltılık sayı ifadeleri bir tamsayı türüne açıkça dönüştürülebilirdi. 0.8.0'dan itibaren, bu tür açık dönüştürmeler, örtülü dönüştürmeler kadar katıdır, yani, yalnızca ifade elde edilen aralığa uyuyorsa bunlara izin verilir.

Sabit Boyutlu Bayt Dizileri

Ondalık sayı ifadeleri örtük olarak sabit boyutlu bayt dizilerine dönüştürülemez. Onaltılık sayı ifadeleri olabilir, ancak yalnızca onaltılık basamak sayısı bayt türünün boyutuna tam olarak uyuyorsa. Bir istisna olarak, sıfır değerine sahip hem ondalık hem de onaltılık ifadeler herhangi bir sabit boyutlu bayt türüne dönüştürülebilir:

```
bytes2 a = 54321; // izin verilmez
bytes2 b = 0x12; //  izin verilmez
bytes2 c = 0x123; // izin verilmez
bytes2 d = 0x1234; // uygun
bytes2 e = 0x0012; // uygun
bytes4 f = 0; // uygun
bytes4 g = 0x0; // uygun
```

String ifadeleri ve onaltılı string ifadeleri, karakter sayıları bayt türünün boyutuyla eşleşiyorsa, örtük olarak sabit boyutlu bayt dizilerine dönüştürülebilir:

```
bytes2 a = hex"1234"; // uygun
bytes2 b = "xy"; // uygun
bytes2 c = hex"12"; // izin verilmez
bytes2 d = hex"123"; // izin verilmez
bytes2 e = "x"; // izin verilmez
bytes2 f = "xyz"; // izin verilmez
```

Adresler

> *Adres Değişmezleri* bölümünde açıklandığı gibi, sağlama toplamı (checksum) testini geçen doğru boyuttaki onaltılık ifadeler `address` türündedir. Başka hiçbir ifade `address` türüne örtük olarak dönüştürülemez.

Not: 0.8.0 sürümünden önce `address` veya `address payable`'a herhangi bir tamsayı türünden (herhangi bir boyutta, imzalı veya imzasız) açıkça dönüştürülebilmekteydi. 0.8.0 ile birlikte yalnızce `uint160`'dan dönüştürmeye izin verilmektedir.

3.7 Birimler ve Global Olarak Kullanılabilir Değişkenler

3.7.1 Ether Birimleri

Bir değişmez sayı, Ether'in bir alt para birimini belirtmek için `wei`, `gwei` veya `ether` son ekini alabilir; burada son eki olmayan Ether sayılarının Wei olduğu varsayılır.

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

Alt isim ekinin("e") tek etkisi, onluk bir kuvvetle çarpmadır. .. note:: 0.7.0 sürümünde "finney" ve "szabo" adları kaldırılmıştır.

3.7.2 Zaman Birimleri

Gerçek sayılardan sonra gelen `saniye`, `dakika`, `saat`, `gün` ve `hafta` gibi son ekler, saniyelerin temel birim olduğu zaman birimlerini belirtmek için kullanılabilir. Ve ayrıca birimler aşağıdaki şekilde olduğu gibi basitçe ele alınır:

- `1 == 1 saniye`
- `1 dakika == 60 saniye`
- `1 saat == 60 dakika`
- `1 gün == 24 saat`
- `1 hafta == 7 gün`

Bu birimleri kullanarak takvim hesaplamaları yapıyorsanız dikkatli olun, çünkü her yıl 365 güne eşit değildir ve "artık saniyeler" nedeniyle her gün bile 24 saat değildir. "Artık saniyelerin" tahmin edilememesi nedeniyle, tam bir takvim kütüphanesinin harici bir oracle tarafından güncellenmesi gerekir.

Not: Yukarıdaki nedenlerden dolayı `years` soneki 0.5.0 sürümünde kaldırılmıştır.

Bu son ekler değişkenlere uygulanamaz. Örneğin, bir fonksiyon parametresini gün olarak yorumlamak istiyorsanız, aşağıdaki şekilde yapabilirsiniz:

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

3.7.3 Özel Değişkenler ve Fonksiyonlar

Global ad alanında her zaman var olan özel değişkenler ve işlevler vardır ve bunlar çoğunlukla blok zinciri hakkında bilgi sağlamak için kullanılır. Veya bunlara ek olarak genel kullanım amaçlı yardımcı fonksiyonlar da bulunmaktadır.

Blok ve İşlem Özellikleri

- `blockhash(uint blockNumber) returns (bytes32)`: `blocknumber` en son 256 bloktan biri olduğunda verilen bloğun hash`ini döndürür, aksi takdirde sıfır döndürür
- `block.basefee` (`uint`): mevcut bloğun baz ücreti (EIP-3198 ve EIP-1559)
- `block.chainid` (`uint`): mevcut bloğun zincir kimliği
- `block.coinbase` (`address payable`): mevcut blok madencisinin adresi
- `block.difficulty` (`uint`): mevcut blok zorluğu

- `block.gaslimit` (`uint`): mevcut blok gas sınırı
- `block.number` (`uint`): mevcut blok numarası
- `block.timestamp` (`uint`): unix döneminden bu yana saniye biçimindeki mevcut blok zaman bilgisi
- `gasleft() returns (uint256)`: kalan gas
- `msg.data` (`bytes calldata`): bütün calldata
- `msg.sender` (`address`): mesajın göndericisi (mevcut çağırma için)
- `msg.sig` (`bytes4`): calldata'nın ilk 4 byte değeri (yani fonksiyon tanımlayıcısı)
- `msg.value` (`uint`): mesaj ile birlikte gönderilen wei miktarı
- `tx.gasprice` (`uint`): işlemin gas fiyatı
- `tx.origin` (`address`): işlemin göndericisi (tam çağrı zinciri)

Not: `msg.sender` ve `msg.value` dahil olmak üzere `msg` öğesinin tüm üyelerinin değerleri her **harici** işlev çağrısı için değişebilir. Buna kütüphane fonksiyonlarına yapılan çağrılar da dahildir.

Not: Sözleşmeler, bir bloğa dahil edilen bir işlem bağlamında değil de zincir dışı değerlendirildiğinde, "block.*" ve "tx.*" ifadelerinin herhangi bir belirli blok veya işlemden gelen değerleri ifade ettiğini varsaymamalısınız. Bu değerler, sözleşmeyi yürüten ESM uygulaması tarafından sağlanır ve isteğe bağlı olabilir.

Not: Ne yaptığınızı bilmiyorsanız, rasgelelik kaynağı olarak `block.timestamp` veya `blockhash`'e güvenmeyin.

Hem zaman bilgisi hem de blok hash'i madenciler tarafından bir dereceye kadar etkilenebilir. Madencilik topluluğunda bulunan kötü aktörler, örneğin seçilen bir hash üzerinde bir kumarhane ödeme fonksiyonu çalıştırabilir ve herhangi bir para almazlarsa farklı bir hash'i çözmeyi yeniden deneyebilirler.

Mevcut blok zaman bilgisi, son bloğun zaman bilgisinden kesinlikle daha büyük olmalıdır. Ancak kabul edilebilecek tek garanti zaman bilgisi, standart zincirdeki iki ardışık bloğun zaman bilgileri arasında bir yerde olmasıdır.

Not: Ölçeklenebilirlik nedeniyle blok hash'leri tüm bloklar için mevcut değildir. Yalnızca en son 256 bloğun hash'lerine erişebilirsiniz, bunun dışındaki tüm değerler sıfır olacaktır.

Not: Daha önce `blockhash` işlevi `block.blockhash` olarak biliniyordu, bu işlev 0.4.22 sürümünde kullanımdan kaldırılmış ve 0.5.0 sürümünde tamamen kaldırılmıştır.

Not: Daha önce `gasleft` işlevi `msg.gas` olarak biliniyordu, bu işlev 0.4.21 sürümünde kullanımdan kaldırılmış ve 0.5.0 sürümünde tamamen kaldırılmıştır.

Not: 0.7.0 sürümünde `now` takma adı (`block.timestamp` için) kaldırıldı.

ABI Şifreleme ve Şifreyi Çözme Fonksiyonları

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI verilen verinin şifresini çözerken, tipler ikinci argüman olarak parantez içinde verilir. Örneğin: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: ABI verilen argümanları şifreler
- `abi.encodePacked(...) returns (bytes memory)`: Verilen argümanların *paketlenmiş şifreleme* işlemini gerçekleştirir. Paketli şifrelemenin belirsiz olabileceğine dikkat edin!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: ABI, verilen bağımsız değişkenleri ikinciden başlayarak şifreler ve verilen dört baytlık seçicinin önüne ekler.
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Şuna eşdeğerdir `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)`: ABI, `functionPointer` çağrısını veri grupları içinde bulunan argümanlarla şifreler. Tam bir tür denetimi gerçekleştirerek türlerin fonksiyon imzasıyla eşleşmesini sağlar. Sonuç `abi.encodeWithSelector(functionPointer.selector, (...))` değerine eşittir

---

Not: Bu şifreleme fonksiyonları, harici bir fonksiyonu çağırmadan harici fonksiyon çağrıları için veri oluşturmak amacıyla kullanılabilir. Ayrıca, `keccak256(abi.encodePacked(a, b))` yapılandırılmış verilerin hashini hesaplamanın bir yoludur (ancak farklı fonksiyon parametre türleri kullanarak bir "hash çakışması" oluşturmanın mümkün olduğunu unutmayın).

Şifreleme ile ilgili ayrıntılar için *ABI* ve *sıkıca paketlenmiş şifreleme* hakkındaki belgelere bakabilirsiniz.

Byte Üyeleri

- `bytes.concat(...) returns (bytes memory)`: *Değişken sayıda bayt ve bytes1, …, bytes32 argümanlarını bir bayt dizisine birleştirir*

String Üyeleri

- `string.concat(...) returns (string memory)`: *Değişken sayıda string argümanını tek bir string dizisinde birleştirir*

Hata İşleme

Hata işleme ve hangi fonksiyonun ne zaman kullanılacağı hakkında daha fazla bilgi için *assert ve require* bölümüne bakın.

`assert(bool condition)`
: Panik hatasına ve dolayısıyla koşul karşılanmazsa durum değişikliğinin tersine dönmesine neden olur - dahili hatalar için kullanılır.

`require(bool condition)`
: koşul karşılanmazsa geri döner - girişlerdeki veya harici bileşenlerdeki hatalar için kullanılır.

`require(bool condition, string memory message)`
: koşul karşılanmazsa geri döner - girişlerdeki veya harici bileşenlerdeki hatalar için kullanılır. Ayrıca bir hata mesajı da sağlar.

revert()
yürütmeyi iptal eder ve durum değişikliklerini geri alır

revert(string memory reason)
açıklayıcı bir string sağlayarak yürütmeyi iptal eder ve durum değişikliklerini geri alır

Matematiksel ve Kriptografik Fonksiyonlar

addmod(uint x, uint y, uint k) returns (uint)
toplama işleminin isteğe bağlı kesinlikte gerçekleştirildiği ve `2**256``da kapsamadığı ``(x + y) % k` değerini hesaplar. Sürüm 0.5.0'den başlayarak "k!= 0" olduğunu iddia eder.

mulmod(uint x, uint y, uint k) returns (uint)
çarpmanın isteğe bağlı kesinlikte gerçekleştirildiği ve `2**256` değerinde kapsamadığı `(x * y) % k` değerini hesaplar. Sürüm 0.5.0'dan başlayarak `k != 0` olduğunu iddia eder.

keccak256(bytes memory) returns (bytes32)
girdinin Keccak-256 hash'ini hesaplar

Not: Eskiden `keccak256` için `sha3` adında bir takma ad vardı, ancak bu ad 0.5.0 sürümünde kaldırıldı.

---

sha256(bytes memory) returns (bytes32)
girdinin SHA-256 hash'ini hesaplar

ripemd160(bytes memory) returns (bytes20)
girdinin RIPEMD-160 hash'ini hesaplar

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
eliptik eğri imzasından açık anahtarla ilişkili adresi kurtarır veya hata durumunda sıfır döndürür. Fonksiyon parametreleri imzanın ECDSA değerlerine karşılık gelir:

- `r` = imzanın ilk 32 byte'ı
- `s` = imzanın ikinci 32 byte'ı
- `v` = imzanın son 1 byte'ı

`ecrecover` yalnızca bir `address` döndürür, `address payable` döndürmez. Kurtarılan adrese para aktarmanız gerekirse, dönüştürme için *address payable* bölümüne bakabilirsiniz.

Daha fazla ayrıntı için örnek kullanım bölümünü okuyun.

Uyarı: Eğer `ecrecover` kullanıyorsanız, geçerli bir imzanın ilgili özel anahtarın (private key) bilinmesini gerektirmeden farklı bir geçerli imzaya dönüştürülebileceğini unutmayın. Homestead hard fork'unda bu sorun _transaction_ signatures için düzeltildi (bkz. EIP-2), ancak ecrecover fonksiyonu değişmeden kaldı.

İmzaların benzersiz olmasını istemediğiniz veya bunları öğeleri tanımlamak için kullanmadığınız sürece bu genellikle bir sorun değildir. OpenZeppelin, bu sorun olmadan `ecrecover` için bir wrapper olarak kullanabileceğiniz bir ECDSA yardımcı kütüphanesine sahiptir.

Not: Bir *özel blok zincirinde* `sha256`, `ripemd160` veya `ecrecover` çalıştırırken, Out-of-Gas (Bitmiş Gas) ile karşılaşabilirsiniz. Bunun nedeni, bu fonksiyonların "önceden derlenmiş sözleşmeler" olarak uygulanması ve yalnızca ilk mesajı aldıktan sonra gerçekten var olmalarıdır (sözleşme kodları sabit kodlanmış olsa da). Mevcut olmayan sözleşmelere gönderilen mesajlar daha pahalıdır ve bu nedenle yürütme sırasında Out-of-Gas (Bitmiş Gas) hatasıyla karşılaşa-

bilir. Bu sorun için geçici bir çözüm, gerçek sözleşmelerinizde kullanmadan önce her bir sözleşmeye Wei (örneğin 1) göndermektir. Bu sorun, ana veya test ağında bir geçerli değildir.

Adres Tipleri Üyeleri

<address>.balance (uint256)
Wei biçimindeki *Adresler* bakiyesi

<address>.code (bytes memory)
ref:*address* adresindeki kod (boş olabilir)

<address>.codehash (bytes32)
ref:*address* kod hash'i

<address payable>.transfer(uint256 amount)
verilen Wei miktarını *Adresler* 'ine gönderir, başarısız olması durumunda geri döner, 2300 gas ücreti iletir, ayarlanabilir değildir

<address payable>.send(uint256 amount) returns (bool)
verilen Wei miktarını *Adresler* 'ine gönderir, başarısız olması durumunda `false` döndürür, 2300 gas ücreti iletir, ayarlanabilir değildir

<address>.call(bytes memory) returns (bool, bytes memory)
verilen yük ile düşük seviyeli `CALL` yayınlar, başarı koşulu ve dönüş verisi döndürür, mevcut tüm gas'ı iletir, ayarlanabilirdir

<address>.delegatecall(bytes memory) returns (bool, bytes memory)
verilen yük ile düşük seviyeli `DELEGATECALL` yayınlar, başarı koşulu ve dönüş verisi döndürür, mevcut tüm gazı iletir, ayarlanabilirdir

<address>.staticcall(bytes memory) returns (bool, bytes memory)
verilen yük ile düşük seviyeli `STATICCALL` yayınlar, başarı koşulunu ve dönüş verilerini döndürür, mevcut tüm gazı iletir, ayarlanabilirdir

Daha fazla bilgi için *Adresler* ile ilgili bölüme bakın.

Uyarı: Başka bir sözleşme fonksiyonunu çalıştırırken mümkün olduğunca `.call()` kullanmaktan kaçınmalısınız, çünkü bu tür denetimi, fonksiyon varlığı denetimini ve argüman paketlemeyi atlar.

Uyarı: `send` kullanmanın bazı tehlikeleri vardır: Çağrı yığını derinliği 1024 ise transfer başarısız olur (bu her zaman çağıran kişi tarafından zorlanabilir) ve ayrıca alıcının gas'ı biterse de başarısız olur. Bu nedenle, güvenli Ether transferleri yapmak için, her zaman `send` dönüş değerini kontrol edin, `transfer` kullanın veya daha da iyisi: Alıcının parayı çektiği bir model kullanın.

Uyarı: ESM'nin mevcut olmayan bir sözleşmeye yapılan bir çağrının her zaman başarılı olacağını düşünmesi nedeniyle, Solidity harici çağrılar gerçekleştirirken `extcodesize` işlem kodunu kullanarak ekstra bir kontrol yapar. Bu, çağrılmak üzere olan sözleşmenin ya gerçekten var olmasını (kod içermesini) ya da bir istisnanın ortaya çıkmasını sağlar.

Sözleşme örnekleri yerine adresler üzerinde çalışan düşük seviyeli çağrılar (yani `.call()`, `.delegatecall()`, `.staticcall()`, `.send()` ve `.transfer()`) Bu kontrolü **içermezler**, bu da onları gas açısından daha ucuz ama aynı zamanda daha az güvenli hale getirir.

Not: 0.5.0 sürümünden önce, Solidity adres üyelerine bir sözleşme örneği tarafından erişilmesine izin veriyordu, örnek vermek gerekirse `this.balance`. Bu fonksiyon artık yasaklanmıştır ve adrese yönelik olarak açık bir dönüşüm yapılmalıdır: `address(this).balance`.

Not: Durum değişkenlerine düşük seviyeli bir "delegatecall" yoluyla erişiliyorsa eğer, çağrılan sözleşmenin çağıran sözleşme tarafından depolama değişkenlerine adıyla doğru şekilde erişebilmesi için iki sözleşmenin depolama düzeninin aynı hizada olması gerekir. Üst düzey kütüphanelerde olduğu gibi depolama işaretçilerinin(pointer) fonksiyon argümanları olarak aktarılması durumunda bu durum elbette geçerli değildir.

Not: 0.5.0 sürümünden önce, `.call`, `.delegatecall` ve `.staticcall` yalnızca başarı koşulunu döndürüyordu, dönüş verisini döndürmüyordu.

Not: 0.5.0 sürümünden önce, `delegatecall` ile benzer ancak biraz farklı anlamlara sahip `callcode` adlı bir üye de bulunmaktaydı.

Sözleşme İle İlgili

`this` (mevcut sözleşmenin türü)
mevcut sözleşme, açıkça *Adresler*'ine dönüştürülebilir

`selfdestruct(ödenebilir alıcı adresi)`
Mevcut sözleşmeyi yok eder, fonlarını verilen *Adresler* e gönderir ve yürütür. `selfdestruct`'ın ESM'den miras kalan bazı özelliklere sahip olduğunu unutmayın:

- alıcı sözleşmenin alma(receive) fonksiyonu yürütülmez.
- sözleşme sadece işlemin sonunda gerçekten yok edilir ve `revert` bu yok edilme işlemini "geri alabilir".

Ayrıca, geçerli sözleşmenin tüm fonksiyonları, geçerli fonksiyon da dahil olmak üzere doğrudan çağrılabilir.

Not: 0.5.0 sürümünden önce, `selfdestruct` ile aynı semantiğe sahip `suicide` adlı bir fonksiyon bulunmaktaydı.

Type Bilgileri

`type(X)` ifadesi `X` türü hakkında bilgi almak için kullanılabilir. Şu anda, bu özellik için sınırlı bir destek bulunmaktadır (`X` bir sözleşme veya tamsayı türü olabilir), ancak gelecekte genişletilebilir.

Aşağıdaki özellikler bir sözleşme tipi(type) `C` için kullanılabilir:

`type(C).name`
Sözleşmenin ismi.

`type(C).creationCode`
Sözleşmenin oluşturma bayt kodunu içeren bellek bayt dizisi. Bu, özellikle `create2` işlem kodu kullanılarak özel oluşturma rutinleri oluşturmak için satır içi derlemede kullanılabilir. Bu özelliğe sözleşmenin kendisinden veya türetilmiş herhangi bir sözleşmeden **erişilemez**. Bytecode'un çağrı bölgesisin bytecode'una dahil edilmesine neden olur ve bu nedenle bunun gibi döngüsel referanslar mümkün değildir.

type(C).runtimeCode
: Sözleşmenin çalışma zamanı bayt kodunu içeren bellek bayt dizisi. Bu, genellikle C yapıcısı tarafından dağıtılan koddur. Eğer C``nin inline assembly kullanan bir kurucusu varsa, bu gerçekte dağıtılan bytecode'dan farklı olabilir. Ayrıca, kütüphanelerin normal çağrılara karşı koruma sağlamak için dağıtım sırasında çalışma zamanı bayt kodlarını değiştirdiklerini unutmayın. Bu özellik için de ``.creationCode ile aynı kısıtlamalar geçerlidir.

Yukarıdaki özelliklere ek olarak, bir arayüz tipi I için aşağıdaki özellikler kullanılabilir:

type(I).interfaceId:
: Verilen I arayüzünün EIP-165 <https://eips.ethereum.org/EIPS/eip-165>`_ arayüz tanımlayıcısını içeren bir ``bytes4 değeri. Bu tanımlayıcı, miras alınan tüm fonksiyonlar hariç olmak üzere, arayüzün kendi içinde tanımlanan tüm fonksiyon seçicilerinin XOR 'u olarak tanımlanır.

Aşağıdaki özellikler T tamsayı(integer) türü için kullanılabilir:

type(T).min
: T tipi tarafından temsil edilebilen en küçük değer.

type(T).max
: T tipi tarafından temsil edilebilen en büyük değer.

3.7.4 Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future:

after, alias, apply, auto, byte, case, copyof, default, define, final, implements, in, inline, let, macro, match, mutable, null, of, partial, promise, reference, relocatable, sealed, sizeof, static, supports, switch, typedef, typeof, var.

3.8 İfadeler ve Kontrol Yapıları

3.8.1 Kontrol Yapıları

Kıvrımlı parantez dilleri olarak (C, C++, Java, ya da C# gibi) bilinen kontrol yapılarının çoğu Solidity'de mevcuttur:

C veya JavaScript'te kullanılan standart semantik ile if, else, while, do, for, break, continue, return yapıları vardır.

Solidity ayrıca try/catch-ifadeleri biçiminde hata yakalamayı da destekler, ancak yalnızca *harici fonksiyon çağrıları* ve sözleşme oluşturma çağrıları için geçerlidir. Hatalar *revert ifadesi* kullanılarak oluştutulabilir.

Parantezler koşul ifadelerinde gözardı *edilemez*, ancak tek-durumlu gövdelerin etrafında küme parantezleri gözardı edilebilir.

C ve JavaScript'te olduğu gibi bool olmayan türlerden bool türlerine tür dönüşümü olmadığını unutmayın, yani if (1) { ... } ifadesi Solidity için geçerli *değildir*.

3.8.2 Fonksiyon Çağrıları

Dahili Fonksiyon Çağrıları

Mevcut sözleşmenin fonksiyonları, bu örnekte görüldüğü gibi, doğrudan ("dahili") olarak, aynı zamanda yinelemeli olarak çağrılabilir:

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

// Bu bir uyarı bildirir
contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

Bu fonksiyon çağrıları ESM (Ethereum Sanal Makinası) içinde basit geçişlere dönüştürülür. Bu, mevcut belleğin silinmemesini sağlar, yani dahili olarak çağrılan fonksiyonlara bellek referanslarını iletmek çok verimlidir. Yalnızca aynı sözleşme örneğinin fonksiyonları dahili olarak çağrılabilir.

Yine de aşırı özyinelemeden kaçınmalısınız, çünkü her dahili fonksiyon çağrısı en az bir yığın yuvası kullanır ve yalnızca 1024 yuva mevcuttur.

Harici Fonksiyon Çağrıları

Fonksiyonlar ayrıca `this.g(8);` ve `c.g(2);` notasyonu kullanılarak da çağrılabilir, burada `c` bir sözleşme örneğidir ve `g`, `c` ye ait bir fonksiyondur. `g` fonksiyonunu her iki şekilde çağırmak, yani doğrudan atlamalar yoluyla değil, bir mesaj kullanılarak çağrılması "harici" çağrı olarak adlandırılır. Lütfen `this` üzerindeki fonksiyon çağrılarının constructorda kullanılamayacağını unutmayın, çünkü asıl sözleşme henüz oluşturulmamıştır.

Diğer sözleşmelerin fonksiyonları harici olarak çağrılmalıdır. Harici bir çağrı için, tüm fonksiyon argümanlarının belleğe kopyalanması gerekir.

Not: Bir sözleşmeden diğerine yapılan fonksiyon çağrısı kendi işlemini oluşturmaz, Bu çağrı, genel işlemin parçası olan bir mesaj çağrısıdır.

Diğer sözleşmelerin fonksiyonlarını çağırırken, çağrı ile gönderilen Wei veya gaz miktarını, `{value: 10, gas: 10000}` şeklinde, belirleyebilirsiniz. İşlem kodlarının(opcodes) gaz maliyetleri gelecekte değişebileceğinden, gaz değerlerini açıkça belirtmenin önerilmediğini unutmayın. Sözleşmeye gönderdiğiniz herhangi bir Wei, o sözleşmenin toplam bakiyesine eklenir:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

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

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

`info` fonksiyonu ile `payable` modifier'ını kullanmanız gerekir, çünkü aksi takdirde, `value` seçeneği kullanılamaz.

Uyarı: `feed.info{value: 10, gas: 800}` ifadesinin sadece yerel olarak fonksiyon çağrısı ile gönderilen `value` ve `gas` miktarını ayarladığına, ve sondaki parantezlerin asıl çağrıyı yaptığına dikkat edin. Yani `feed.info{value: 10, gas: 800}` ifadesi fonksiyonu çağırmaz, `value` ve `gas` ayarları kaybolur, yalnızca `feed.info{value: 10, gas: 800}()` fonksiyon çağrısını gerçekleştirir.

ESM'nin var olmayan bir sözleşmeye yapılan bir çağrıyı her zaman başarılı olarak kabul etmesi nedeniyle, Solidity çağrılacak olan sözleşmenin gerçekten var olup olmadığını (kod içermesini) kontrol etmek için `extcodesize` işlem kodunu kullanır, eğer sözleşme yoksa hata verir. Çağrıdan sonra dönüş verilerinin kodu çözülecekse bu aşama atlanır ve böylece ABI kod çözücüsü mevcut olmayan bir sözleşme durumunu yakalar.

Bu kontrolün, sözleşme örnekleri yerine adresler üzerinde çalışan *düşük seviyeli çağrılar* olması durumunda gerçekleştirilmediğini unutmayın.

Not: *önceden derlenmiş sözleşmeler* de, üst düzey çağrılar kullanırken dikkatli olun, çünkü derleyici, kodu çalıştırsalar ve veri döndürebilseler bile bunları mevcut saymaz.

Fonksiyon çağrıları, çağrılan sözleşmenin kendisi bir hata döndürürse veya gazı tükenirse, hatalara da neden olur.

Uyarı: Başka bir sözleşme ile herhangi bir etkileşim, özellikle sözleşmenin kaynak kodu önceden bilinmiyorsa, potansiyel bir tehlike oluşturur. Mevcut sözleşme, kontrolü, çağrılan sözleşmeye devreder ve bu, potansiyel olarak hemen hemen her şeyi yapabilir. Çağrılan sözleşme bilinen bir ana sözleşmeden miras kalsa bile, miras sözleşmesinin yalnızca doğru bir arayüze sahip olması gerekir. Ancak sözleşmenin uygulanması tamamen keyfi olabilir ve bu nedenle tehlike oluşturabilir. Ayrıca, ilk çağrı sisteminizin diğer sözleşmelerine çağrı yapması veya hatta çağrı yapan sözleşmeye geri dönmesi ihtimaline karşı hazırlıklı olun. Bu, çağrılan sözleşmenin,fonksiyonları aracılığıyla çağrı yapan sözleşmesinin durum değişkenlerini değiştirebileceği anlamına gelir. Fonksiyonlarınızı, örneğin sözleşmenizdeki durum değişkenlerinde yapılan herhangi bir değişiklikten sonra harici fonksiyonlara yapılan çağrıların gerçekleşeceği şekilde yazın, böylece sözleşmeniz yeniden giriş istismarına karşı savunmasız kalmaz.

Not: Solidity 0.6.2'den önce, değeri ve gazı belirtmenin önerilen yolu `f.value(x).gas(g)()` kullanmaktı. Bu, Solidity 0.6.2'de kullanımdan kaldırıldı ve Solidity 0.7.0'dan beri kullanımı artık mümkün değil.

Adlandırılmış Çağrılar ve Anonim Fonksiyon Parametreleri

Aşağıdaki örnekte görüldüğü gibi, `{ }` içine alınmışlarsa, fonksiyon çağrısı argümanları herhangi bir sırayla ve tercihe bağlı isimle adlandırılabilir. Argüman listesi, fonksiyon bildirimindeki parametre listesiyle ve adıyla örtüşmelidir, ancak isteğe bağlı olarak sıralanabilir.

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

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }

}
```

Dikkate Alınmayan Fonksiyon Parametre Adları

Kullanılmayan parametrelerin adları (özellikle dönüş parametreleri) atlanabilir. Bu parametreler yığında bulunmaya devam eder, fakat erişilemezler.

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

contract C {
    // parametre için atlanan ad
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

3.8.3 new Yoluyla Sözleşmeler Oluşturma

Bir sözleşme, new anahtar sözcüğünü kullanarak başka sözleşmeler oluşturabilir. Oluşturulan sözleşmenin tam kodu, oluşturulan sözleşme derlendiğinde bilinmelidir, bu nedenle özyinelemeli oluşturma bağımlılıkları imkansızdır.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // C'nin constructor'ının bir parçası olarak yürütülecek

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

    function createAndEndowD(uint arg, uint amount) public payable {
        // Oluşturulmasıyla beraber ether gönder
        D newD = new D{value: amount}(arg);
        newD.x();
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }
}
```

Örnekte görüldüğü gibi, `value` seçeneği kullanılarak bir `D` örneği oluştururken Ether göndermek mümkündür, ancak gaz miktarını sınırlamak mümkün değildir. Oluşturma başarısız olursa (yığın olmaması, yeterli bakiye olmaması veya diğer problemler nedeniyle), bir hata döndürülür.

Saltlı sözleşme kreasyonları / create2

Bir sözleşme oluştururken, sözleşmenin adresi, oluşturulan sözleşmenin adresinden ve her sözleşmede artan bir sayaçtan hesaplanır.

`salt` (bir bytes32 değeri) seçeneğini belirlerseniz, sözleşme kreasyonu yeni sözleşmenin adresini bulmak için farklı bir mekanizma kullanır:

Oluşturan sözleşmenin adresinden adresi, verilen salt değeri, oluşturulan sözleşmenin (kreasyon) bayt kodu ve constructor argümanlarını hesaplayacaktır.

Özellikle sayaç ("nonce") kullanılmaz. Bu, sözleşmeler oluştururken daha fazla esneklik sağlar: Sözleşme oluşturulmadan önce yeni sözleşmenin adresini öğrenebilirsiniz. Üstelik, bu adrese güvenebilirsiniz ayrıca sözleşmelerin oluşturulması durumunda başka sözleşmeler oluşturur.

Buradaki ana kullanım durumu, sadece bir anlaşmazlık varsa yaratılması gereken, zincir dışı etkileşimler için yargıç görevi gören sözleşmelerdir.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        // Bu karmaşık ifade size sadece adresin
        // nasıl önceden hesaplanabileceğini söyler. O sadece örnekleme için oradadır.
        // İhtiyacınız olan aslında sadece ``new D{salt: salt}(arg)``.
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                abi.encode(arg)
            ))
        )))));

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

Uyarı: Saltlı kreasyonların alışılmadık bazı özellikleri vardır. Bir sözleşme, yok edildikten sonra aynı adreste yeniden oluşturulabilir. Yine de, bu yeni oluşturulan sözleşmenin kreasyon bayt kodu aynı olmasına rağmen (bu bir gerekliliktir çünkü aksi takdirde adres değişir) farklı bir dağıtılmış bayt koduna sahip olması bile mümkündür. Bunun nedeni, constructorın iki kreasyon arasında değişmiş olabilecek dış durumu sorgulayabilmesi ve depolanmadan önce bunu deploy edilmiş bayt koduna dahil edebilmesidir.

3.8.4 İfadelerin Değerlendirme Sırası

İfadelerin değerlendirme sırası belirtilmez (daha resmi olarak, ifade ğacındaki bir düğümün çocuklarının değerlendirildiği sıra belirtilmez, ancak elbette düğümün kendisinden önce değerlendirilirler.). Bu sadece ifadelerin sırayla yürütülmesini garanti eder ve boolean ifadeler için kısa devre yapılır.

3.8.5 Atama

Atamaları Yok Etme ve Birden Çok Değer Döndürme

Solidity dahili olarak tuple türlerine, yani derleme zamanında sayısı sabit olan potansiyel olarak farklı türdeki nesnelerin bir listesine izin verir. Bu tuple'lar aynı anda birden çok değer döndürmek için kullanılabilir. Bunlar daha sonra yeni bildirilen değişkenlere veya önceden var olan değişkenlere (veya genel olarak LDeğerlere) atanabilir.

Tuple'lar, Solidity'de uygun tipler değildir, sadece sözdizimsel ifade grupları oluşturmak için kullanılabilirler.

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

contract C {
    uint index;

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

    function g() public {
        // Type ile bildirilen ve döndürülen tupledan atanan değişkenler,
        // tüm öğelerin belirtilmesi gerekmez (ancak sayı eşleşmelidir).
        (uint x, , uint y) = f();
        // Değerleri değiştirmek için yaygın bir numara -- değer olmayan depolama␣
↪türleri için çalışmaz.
        (x, y) = (y, x);
        // Bileşenler dışarıda bırakılabilir (ayrıca değişken bildirimleri için).
        (index, , ) = f(); // index'i  7'ye ayarlar.
    }
}
```

Değişken bildirimlerini ve bildirim dışı atamaları karıştırmak mümkün değildir, yani bu geçerli değildir: `(x, uint y) = (1, 2);`

Not: 0.5.0 sürümünden önce, ya solda ya da sağda doldurulan (hangisi boşsa) daha küçük boyutlu tuplelara atamak mümkündü. Buna artık izin verilmiyor, bu nedenle her iki tarafın da aynı sayıda bileşene sahip olması gerekiyor.

Uyarı: Referans türleri söz konusu olduğunda, Aynı anda birden fazla değişkene atama yaparken dikkatli olun, çünkü bu, beklenmedik kopyalama davranışına yol açabilir.

Diziler ve Structlar için Komplikasyonlar

Atamaların anlamı, diziler ve structlar gibi değer olmayan türler için daha karmaşıktır. `bytelar` ve `string` dahil, ayrıntılar için *Data location and assignment behaviour* bölümüne bakın.

Aşağıdaki örnekte, `g(x)` çağrısının `x` üzerinde hiçbir etkisi yoktur. Çünkü bellekteki depolama değerinin bağımsız bir kopyasını oluşturur. Ancak, `h(x)` `x` öğesini başarıyla değiştirir çünkü bir kopya değil, yalnızca bir referans iletilir.

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

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

3.8.6 Kapsam Belirleme ve Beyanlar

Bildirilen bir değişken, bayt temsilinin tümü sıfır olan bir başlangıç varsayılan değerine sahip olacaktır. Değişkenlerin "varsayılan değerleri", türü ne olursa olsun tipik "sıfır durumu"dur. Örneğin, bir `bool` için varsayılan değer `false` tur. `uint` veya `int` türleri için varsayılan değer `0` dır. Statik olarak boyutlandırılmış diziler ve `bytes1` den `bytes32``ye kadar olan her bir öğe, kendi türüne karşılık gelen varsayılan değere göre başlatılacaktır. Dinamik olarak boyutlandırılmış diziler ve ``bytelar` ve `string` için, varsayılan değer boş bir dizi veya dizedir. `enum` türü için varsayılan değer kendisinin ilk üyesidir.

Solidity'de kapsam belirleme, C99'un yaygın kapsam belirleme kurallarına uyar (ve diğer birçok dil): Değişkenler, bildirimlerinden hemen sonraki noktadan bildirimi içeren en küçük `{ }` bloğunun sonuna kadar görülebilir. Bu kuralın bir istisnası olarak, for döngüsünün başlatma parçasında tanımlanan değişkenler yalnızca for döngüsünün sonuna kadar görünür.

Parametre benzeri değişkenler (fonksiyon parametreleri, modifier parametreleri, catch parametreleri, ...) bir fonksiyon için fonksiyon/modifier'ın gövdesi ve modifier parametresi ve bir catch parametresi için catch bloğunu takip eden kod bloğunun içinde görünürdür.

Değişkenler ve diğer öğeler bir kod bloğunun dışında bildirilir, örneğin fonksiyonlar, sözleşmeler, kullanıcı tanımlı türler, vb. beyan edilmeden önce bile görülebilir. Bu, durum değişkenlerini bildirilmeden önce kullanabileceğiniz ve fonksiyonları yinelemeli olarak çağırabileceğiniz anlamına gelir.

Sonuç olarak, aşağıdaki örnekler uyarı olmadan derlenecektir, çünkü iki değişken aynı ada ancak ayrık kapsamlara sahiptir.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

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

C99 kapsam belirleme kurallarına özel bir örnek olarak, aşağıdakilere dikkat edin, `x` e yapılan ilk atama aslında iç değişkeni değil dış değişkeni atayacaktır. Her durumda, gölgelenen dış değişken hakkında bir uyarı alacaksınız.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// Bu bir uyarı bildirir
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // bu dış değişkene atayacaktır
            uint x;
        }
        return x; // x değeri 2'dir
    }
}
```

Uyarı: 0.5.0 sürümünden önce Solidity, JavaScript ile aynı kapsam kurallarını takip ediyordu; yani, bir fonksiyon içinde herhangi bir yerde bildirilen bir değişken, nerede bildirildiğine bakılmaksızın tüm fonksiyonun kapsamında olurdu. Aşağıdaki örnek, derleme için kullanılan ancak 0.5.0 sürümünden başlayarak bir hataya yol açan bir kod parçacığını göstermektedir.

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

3.8.7 Checked veya Unchecked Matematiksel İşlemler

Bir overflow veya underflow durumu, aritmetik bir işlemin sınırsız bir tamsayı üzerinde işlem yaptığında sonuç türünün aralığı normalin dışında kalması durumudur.

Solidity 0.8.0'dan önce, aritmetik işlemler, ek kontroller getiren kitaplıkların yaygın kullanımına yol açan underflow veya overflow durumunu her zaman kapsardı.

Solidity 0.8.0'dan bu yana, tüm aritmetik işlemler varsayılan olarak overflow ve underflow'u engeller, böylece bu, kitaplıkların kullanımını gereksiz hale getirir.

Önceki davranışı elde etmek için bir `unchecked` bloğu kullanılabilir:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // Bu çıkarma underflow'a sebep olur.
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // Bu çıkarma underflow'u engeller.
        return a - b;
    }
}
```

`f(2, 3)` çağrısı `2**256-1` döndürürken, `g(2, 3)` başarısız bir assertion'a neden olur.

`unchecked` blok, bir blok içinde her yerde kullanılabilir, ama bir bloğu değiştirmek için. Ayrıca nested olamaz.

Ayar yalnızca sözdizimsel olarak bloğun içindeki ifadeleri etkiler. Bir `unchecked` blok içinden çağrılan fonksiyonlar özelliği miras almaz.

Not: Belirsizliği önlemek için, bir `unchecked` blok içinde `_;` kullanamazsınız.

Aşağıdaki operatörler, overflow veya underflow durumunda başarısız bir assertion'a neden olur ve unchecked bir blok içinde kullanılırsa hatasız kalır:

`++`, `--`, `+`, binary `-`, unary `-`, `*`, `/`, `%`, `**`

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

Uyarı: Unchecked blok kullanılarak sıfıra bölme veya mod alma işleminin kontrolünü devre dışı bırakmak mümkün değildir.

Not: Bitsel operatörler overflow veya underflow kontrolleri yapmaz. Bu, özellikle bitsel değiştirmeler (`<<`, `>>`, `<<=`, `>>=`) kullanılırken tamsayı bölme ve 2'nin kuvvetiyle çarpma işleminde görülebilir, Örneğin `type(uint256).max * 8` revert edilse bile `type(uint256).max << 3` revert edilmez.

Not: `int x = type(int).min; -x;` içindeki ikinci ifade overflow ile sonuçlanacak. çünkü negatif aralık, pozitif aralıktan bir değer daha fazla tutabilir.

Açık tür dönüşümleri her zaman sekteye uğrar ve integer'dan enum türüne dönüştürme dışında asla başarısız bir assertion'a neden olmaz.

3.8.8 Hata İşleme: Assert, Require, Revert ve Exceptionlar

Solidity, hataları işlemek için durumu geri döndüren exceptionlar kullanır. Böyle bir exception, geçerli çağrıdaki (ve tüm alt çağrılarındaki) durumda yapılan tüm değişiklikleri geri alır ve çağırana bir hata bildirir.

Bir alt aramada exceptionlar meydana geldiğinde, `try/catch` ifadesiyle yakalanmadıkları sürece otomatik olarak yeniden atılırlar. Bu kuralın istisnaları `send` ve düşük seviyeli fonksiyonlar `call`, `delegatecall` ve `staticcall`: bir exception olması durumunda onu yeniden atmak yerine `false` değerini ilk dönüş değerleri olarak döndürürler.

Uyarı: Düşük seviyeli fonksiyonlar `call`, `delegatecall` ve `staticcall` ESM tasarımının bir parçası olarak, çağrılan hesap mevcut değilse, ilk dönüş değeri olarak `true` döndürür. Gerekirse çağrıdan önce hesabın varlığı kontrol edilmelidir.

Exceptionlar çağırana *hata örnekleri* şeklinde geri iletilen hata verilerini içerebilir. `Error(string)` ve `Panic(uint256)` yerleşik hataları aşağıda açıklandığı gibi özel işlevler tarafından kullanılır. aşağıda açıklandığı gibi özel fonksiyonlar tarafından kullanılır. `Panic` hatasız kodda olmaması gereken hatalar için kullanılırken `Error` "normal" hata koşulları için kullanılır.

`assert` ile Panic ve `require` ile Hata

`assert` ve `require` uygunluk fonksiyonları şu amaçlarla kullanılabilir: koşulları kontrol etmek ve koşul karşılanmazsa bir hata fırlatmak.

`assert` fonksiyonu `Panic(uint256)` türünde bir hata oluşturur. Aynı hata, derleyici tarafından aşağıda listelendiği gibi belirli durumlarda oluşturulur.

Assert yalnızca dahili hataları test etmek ve değişken olmayanları kontrol etmek için kullanılmalıdır. Düzgün çalışan kod, geçersiz harici girişte bile asla Panik oluşturmamalıdır. Eğer bu olursa, o zaman sözleşmenizde düzeltmeniz gereken bir hata olur. Dil analiz araçları, koşulları ve Paniğe neden olacak fonksiyon çağrıları belirlemek için sözleşmenizi değerlendirebilir.

Aşağıdaki durumlarda bir Panik exception'ı oluşturulur. Hata verileriyle birlikte verilen hata kodu, panik türünü belirtir.

1. 0x00: Jenerik derleyici yerleştirilmiş panikler için kullanılır.
2. 0x01: Yanlış olarak değerlendirilen bir argümanla `assert` ü çağırırsanız.
3. 0x11: Bir aritmetik işlem, `unchecked { ... }` bir bloğun dışında underflow veya overflow ie sonuçlanırsa.
4. 0x12; Sıfıra bölerseniz veya mod alma işlemi yaparsanız (ör. `5 / 0` ya da `23 % 0`).
5. 0x21: Çok büyük veya negatif bir değeri bir enum türüne dönüştürürseniz.
6. 0x22: Yanlış kodlanmış bir depolama bayt dizisine erişirseniz.
7. 0x31: Boş bir dizide `.pop()` çağırırsanız.
8. 0x32: Bir diziye, `bytesN``ye veya sınır dışı veya negatif bir dizindeki bir dizi dilimine erişirseniz (ör. ``x[i]` i, `i >= x.length` veya `i < 0` olduğunda).
9. 0x41: Çok fazla bellek ayırırsanız veya çok büyük bir dizi oluşturursanız.
10. 0x51: Dahili fonksiyon türünün hiç başlatılmamış bir değişkenini çağırırsanız.

The `require` fonksiyonu, ya herhangi bir veri içermeyen bir hata ya da `Error(string)` türünde bir hata oluşturur. Yürütme zamanına kadar tespit edilemeyen geçerli koşulları sağlamak için kullanılmalıdır. Bu, çağrılardan harici sözleşmelere yapılan girdiler veya dönüş değerleri üzerindeki koşulları içerir.

Not: Şu anda özel hataları `require` ile birlikte kullanmak mümkün değildir. Lütfen bunun yerine `if (!condition) revert CustomError();` kullanın.

Bir `Error(string)` exceptionı (veya veri içermeyen bir exception) derleyici tarafından aşağıdaki durumlarda oluşturulur:

1. `x``in ``false` olarak değerlendirildiği durumlarda `require(x)` çağrılır.
2. `revert()` veya `revert("description")` kullanırsanız.
3. Kod içermeyen bir sözleşmeyi hedefleyen harici bir fonksiyon çağrısı gerçekleştirirseniz.
4. Sözleşmeniz Ether'i `payable` modifier (constructor ve geri dönüş fonksiyonu dahil) içermeyen public bir fonksiyon aracılığıyla alırsa
5. Sözleşmeniz Ether'i bir public getter fonksiyonu aracılığıyla alıyorsa.

Aşağıdaki durumlarda, harici çağrılardan gelen hata verileri (varsa) iletilir. Bu, bir *Error* veya *Panic* (veya başka ne verildiyse)'e neden olabileceği anlamına gelir:

1. Bir `.transfer()` başarısız olursa.
2. Mesaj çağrısı yoluyla bir fonksiyonu çağırırsanız ancak işlem uygun şekilde tamamlanmazsa (ör. gazının bitmesi, eşleşen bir foksiyonunun olmaması, veya bir exception atması), düşük seviyeli bir işlem `call`, `send`, `delegatecall`, `callcode` ya da `staticcall` kullanılması bunun dışındadır. Düşük seviyeli işlemler hiçbir zaman exception oluşturmaz, ancak `false` döndürerek hataları belirtir.
3. `new` anahtar sözcüğünü kullanarak bir sözleşme oluşturursanız ancak sözleşme oluşturma *düzgün bitmezse*.

İsteğe bağlı olarak `require` için bir string mesajı sağlayabilirsiniz, ancak `assert` için değil.

Not: `require` için bir dize argümanı sağlamazsanız, hata seçiciyi içermeden bile boş hata verileriyle geri dönecektir.

Aşağıdaki örnek, girdilerdeki koşulları kontrol etmek için `require` ve dahili hata kontrolü için `assert` i nasıl kullanabileceğinizi gösterir.

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

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Çift değerler isteniyor");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // Transfer, başarısızlık durumunda bir exception oluşturduğundan ve
        // buradan geri çağırma yapamadığı için, hala paranın yarısına sahip olmak için
        //bir yol olmaması gerekiyor
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

Dahili olarak, Solidity bir geri alma işlemi gerçekleştirir (talimat `0xfd`). Bu, ESM'nin duruma yapılan tüm değişiklikleri geri almasına neden olur. Geri dönüş nedeni yürütmeye devam etmenin güvenli bir yolunun olmamasıdır, çünkü beklenen etki gerçekleşmez. İşlemlerin atomikliğini korumak istediğimiz için, en güvenli eylem, tüm değişiklikleri geri almak ve etkisi olmadan tüm işlemi (veya en azından çağrıyı) yapmaktır.

Her iki durumda da çağıran bu tür hatalara `try/catch` kullanarak tepki verebilir, ancak çağrılandaki değişiklikler her zaman geri alınır.

Not: Solidity 0.8.0'dan önce, panik exceptionları, çağrı için mevcut tüm gazı tüketen `geçersiz` işlem kodu kullanırdı. `require` ifadesini kullanan exceptionlar Metropolis'in piyasaya sürülmesinden önce tüm gazı tüketmek için kullanırlardı.

---

revert

`revert` ifadesi ve `revert` fonksiyonu kullanılarak doğrudan bir geri alma tetiklenebilir.

`revert` ifadesi, parantez olmadan doğrudan argüman olarak özel bir hata alır:

revert CustomError(arg1, arg2);

Geriye dönük uyumluluk nedenleriyle, parantez kullanan `revert()` fonksiyonu da vardır. ve bir string kabul eder:

revert(); revert("açıklama");

Hata verileri çağırana geri gönderilir ve orada yakalanabilir. `revert()` kullanmak, herhangi bir hata verisi olmadan geri döndürmeye neden olurken, `revert("açıklama")`, bir `Error(string)` hatası yaratacaktır.

Özel bir hata örneği kullanmak, genellikle bir metin açıklamasından çok daha ucuz olacaktır, çünkü sadece 4 baytta kodlanmış olan hatayı tanımlamak için hatanın adını kullanabilirsiniz. Herhangi bir maliyete maruz kalmadan NatSpec aracılığıyla daha uzun bir açıklama sağlanabilir.

Aşağıdaki örnek, `revert` ve eşdeğer `require` ile birlikte bir hata metni ve özel bir hata örneğinin nasıl kullanılacağını gösterir:

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

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Yeterli Eter sağlanmadı.");
        // Bunu yapmanın alternatif yolu:
        require(
            amount <= msg.value / 2 ether,
            "Yeterli Eter sağlanmadı."
        );
        // Satın alma işlemini gerçekleştirin.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }
}
```

`revert` ve `require` argümanlarının yan etkileri olmadığı sürece örneğin argümanlar sadece stringse `if (! condition) revert(...);` ve `require(condition, ...);` eşdeğerdir.

Not: `require` fonksiyonu diğer herhangi bir fonksiyon gibi değerlendirilir. Bu, fonksiyonun kendisi yürütülmeden önce tüm argümanların değerlendirildiği anlamına gelir. Özellikle, `require(condition, f())` içinde, `f` fonksiyonu `condition` doğru olduğunda bile yürütülür.

Elde edilen string, fonksiyonuna yapılan bir çağrıymış gibi *abi-encoded* şeklindedir. Yukarıdaki örnekte, `revert("Yeterli Eter sağlanmadı..");` hata dönüş verisi olarak aşağıdaki hexadecimal değeri döndürür:

```
0x08c379a0                                                         // Error(string)␣
↪için fonksiyon seçici
0x0000000000000000000000000000000000000000000000000000000000000020 // veri ofseti
0x000000000000000000000000000000000000000000000000000000000000001a // String uzunluğu
0x4e6f7420656e6f756768204574686572207072f6696465642e0000000000000000 // String verisi
```

Elde edilen mesaj, çağıran tarafından aşağıda gösterildiği gibi `try/catch` kullanılarak alınabilir.

Not: Eskiden 0.4.13 sürümünde kullanımdan kaldırılan ve 0.5.0 sürümünde kaldırılan `revert()` ile aynı semantiğe sahip `throw` adında bir anahtar kelime vardı.

---

try/ catch

Harici aramadaki bir hata, aşağıdaki gibi bir try/catch ifadesi kullanılarak yakalanabilir:

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

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // 10'dan fazla hata varsa mekanizmayı
        // kalıcı olarak devre dışı bırakır.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // Bu, getData içinde
            // revertün çağrılması durumunda
            // ve bir stringin sağlanması durumunda yürütülür.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            // Bu panik durumunda yürütülür,
            // ör. sıfıra bölme veya taşma gibi ciddi bir hata
            // ya da overflow varsa. Hata kodu
            // hatanın türünü belirlemek için kullanılır.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // Bu revert() kullanıldığında yürütülür.
            errorCount++;
            return (0, false);
        }
    }
}
```

`try` anahtar sözcüğünü, harici bir fonksiyon çağrısını veya bir sözleşme oluşturmayı temsil eden bir ifade takip etmelidir (`new ContractName()`). İfade içindeki hatalar yakalanmaz (örneğin fonksiyon çağrılarını içeren karmaşık bir ifade ise), harici fonksiyonun içinde sadece bir geri dönüş oluşur. Aşağıdaki `returns` kısmı (isteğe bağlıdır) harici arama tarafından döndürülen türlerle eşleşen dönüş değişkenlerini bildirir. Hata olmaması durumunda,bu değişkenler atanır ve sözleşmenin yürütülmesi ilk başarı bloğu içinde devam eder. Başarı bloğunun sonuna ulaşılırsa, `catch` bloklarından sonra yürütme devam eder.

Solidity, duruma bağlı olarak farklı türde catch bloklarını destekler. hata türü:

- `catch Error(string memory reason) { ... }`: Bu catch yan tümcesi eğer hata `revert("reasonString")` ya da `require(false, "reasonString")` nedeniyle oluyorsa (veya böyle bir istisnaya neden olan dahili bir hata) çalıştırılır.
- `catch Panic(uint errorCode) { ... }`: If the error was caused by a panic, i.e. by a failing `assert`, division by zero, invalid array access, arithmetic overflow and others, this catch clause will be run.
- `catch (bytes memory lowLevelData) { ... }`: Bu yan tümce hata imzası başka bir maddeyle eşleşmezse, hata mesajının kodu çözülürken bir hata oluştuysa veya exception dışında hiçbir hata verisi sağlanmadıysa yürütülür. Bildirilen değişken, bu durumda düşük seviyeli hata verilerine erişim sağlar.
- `catch { ... }`: Hata verileriyle ilgilenmiyorsanız, `catch { ... }` (tek catch maddesi olarak bile) önceki madde yerine sadece kullanabilirsiniz.

Gelecekte başka türdeki hata verilerinin de desteklemesi planlanmaktadır. `Error` ve `Panic` stringleri şu anda olduğu gibi ayrıştırılıyor ve tanımlayıcı olarak kabul edilmiyor.

Tüm hata durumlarını yakalamak için, en azından `catch { ...}` veya `catch (bytes memory lowLevelData) { ... }` yan tümcesine sahip olmalısınız.

`returns` ve `catch` yan tümcesinde belirtilen değişkenler yalnızca takip eden blok kapsamındadır.

Not: `catch Error(string memory reason)` kodunun çözülmesi sırasında bir hata varsa ve düşük seviyeli bir catch cümlesi varsa, bu hata orada yakalanır.

Not: Yürütme bir catch bloğuna ulaşırsa, harici çağrının durum değiştiren etkilerine geri dönülür. Yürütme başarı bloğuna ulaşırsa, etkiler geri alınmaz. Etkiler geri alındıysa, yürütme ya bir catch bloğunda devam eder ya da try/catch ifadesinin yürütülmesi kendisine geri döner (örneğin, yukarıda belirtildiği gibi kod çözme hataları veya düşük seviyeli bir yakalama maddesi sağlamama nedeniyle).

Not: Başarısız bir çağrının arkasındaki sebep çok çeşitli olabilir. Hata mesajının doğrudan çağrılan sözleşmeden geldiğini varsaymayın: Hata, çağrı zincirinde daha derinlerde meydana gelmiş olabilir ve çağrılan sözleşme onu iletmiş olabilir. Ayrıca, kasıtlı bir hata durumu değil, gazın bitmesi durumundan kaynaklanabilir: Çağıran, bir aramada her zaman gazın en az 1/64'ünü elinde tutar ve böylece çağrılan sözleşmenin gazı bitse bile, çağıranın hala biraz gazı vardır.

3.9 Akıllı Sözleşmeler

Solidity'deki akıllı sözleşmeler nesne yönelimli programlama dillerine benzerdir. State değişkenlerinde kalıcı data içerirler ve fonksiyonlar bu değişkenlerin değerini değiştirebilir. Başka bir akıllı sözleşmedeki fonksiyonu çağırmak bir EVM fonksiyon çağrısı gerçekleştirir ve burada çağıran akıllı sözleşmenin state değişkenlerine erişilemez. Akıllı sözleşmede herhangi bir şeyin yaşanmasını istiyorsanız o akıllı sözleşmenin herhangi bir fonksiyonunu çağırmanız gerekir. Çünkü Ethereum'da "cron" konsepti yoktur, yani akıllı sözleşmeler kendi başlarına bir şeyler yapamaz. Dışarıdan tetiklenmeleri gerekir.

3.9.1 Akıllı Sözleşme Oluşturma

Akıllı sözleşmeler iki şekilde oluşturulabilir; "dışarıdan" bir Ethereum transactionı ile veya Solidity kullanarak direkt başka bir akıllı sözleşme içerisinde.

Remix gibi IDE'ler oluşturma aşamasını kullanıcı arayüzü kullanarak kolayca gerçekleştirmenize yardımcı olur.

Programlama ile akıllı sözleşme oluşturmanın bir yolu ise web3.js gibi bir JavaScript API kullanımıdır. Akıllı sözleşme oluşturmaya yarayan web3.eth.Contract isimli bir methodu vardır.

Bir akıllı sözleşme oluşturulduğu zaman *constructor* isimli bir fonksiyon sadece bir kere olmak üzere çalıştırılır. Bundan sonra bu fonksiyona erişim mümkün değildir.

Constructor kullanmak zorunlu değildir. Bir akıllı sözleşmede sadece bir adet constructor olabilir ve overloading yapılması mümkün değildir.

Constructor çalıştırıldıktan sonra akıllı sözleşme kodunun son hali blok zincirinde saklanır. Bu kod bütün public ve external fonksiyonları içerir. Deploy edilen kod constructor fonksiyonu ve sadece constructor içerisinde çağrılan internal fonksiyonları içermez.

Özünde constructor parametreleri *ABI encoded* olarak akıllı sözleşme kodunun sonuna eklenir (bytecode halinin), ama eğer `web3.js` kullanıyorsanız bunu umursamanıza gerek yok. Çünkü o sizin için bu işlemleri gerçekleştiriyor.

Eğer bir akıllı sözleşme başka bir akıllı sözleşme oluşturmak istiyorsa, oluşturmak istediği akıllı sözleşmenin kaynak kodunu (ve binary halini) bilmelidir. Bu demektir ki döngüsel olarak akıllı sözleşme oluşturmak mümkün değildir.

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


contract OwnedToken {
    // `TokenCreator` aşağıda belirtilmiş bir akıllı sözleşme tipidir.
    // Yeni bir akıllı sözleşme oluşturmak için kullanılmadığı sürece
    // referans etmekte sorun yoktur.
    TokenCreator creator;
    address owner;
    bytes32 name;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    // Burası constructor fonksiyonumuz. Burada
    // belirtilen isim ve akıllı sözleşmeyi oluşturan adres
    // akıllı sözleşmede kaydedilir.
    constructor(bytes32 name_) {
        // State değişkenlerine isimleri kullanılarak
        // erişilir. `this.owner` şeklinde bir kullanım
        // ile değil. Fonksiyonlara direkt olarak kendi
        // isimlerini kullanarak veya `this.f` şeklinde
        // bir kullanım ile erişebiliriz. Ancak ikinci
        // şekildeki kullanım external olarak (dışarıdan)
        // bir görüş sağlar. Özellikle constructorlarda,
        // fonksiyonlara external olarak erişmemelisiniz.
        // Çünkü o fonksiyonlar henüz oluşturulmadı, yani
        // erişilebilir değil.
        // Daha fazlası için bir sonraki bölüme bakabilirsiniz.
        owner = msg.sender;

        // Burada `address` tipinden `TokenCreator` tipine
        // bir explicit (açık) dönüşüm sağlarız ve bu fonksiyonu
        // çağıran akıllı sözleşmenin bir `TokenCreator` olduğunu varsayarız.
        // Bunu doğrulamanın gerçek bir yöntemi bulunmamakta.
        // Bu işlem yeni bir akıllı sözleşme oluşturmaz.
        creator = TokenCreator(msg.sender);
        name = name_;
    }

    function changeName(bytes32 newName) public {
        // Sadece `creator` `name` değişkenini değiştirebilir.
        // Akıllı sözleşmenin adresini explicit bir dönüşüm ile
        // elde edebilir ve karşılaştırmamızı yapabiliriz.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // Sadece şu anki `owner` token transferi gerçekleştirebilir.
        if (msg.sender != owner) return;

        // `creator` adresindeki akıllı sözleşmenin bir fonksiyonunu
        // kullanarak, işlemin gerçekleştirilebilirliğini
        // kontrol edebilir. Eğer bu işlem hata verirse
        // (örneğin, out-of-gas (gazın tükenmesi)),
        // işlem burada son bulur.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}


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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        public
        returns (OwnedToken tokenAddress)
    {
        // Yeni bir `Token` akıllı sözleşmeyi oluşturur ve adresini return eder.
        // JavaScript tarafında return tipi `address` tipidir.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
        // `tokenAddress` isimli parametrenin tipi
        // `address` tipindendir.
        tokenAddress.changeName(name);
    }

    // Bir transferin gerçekleşip gerçekleşmeyeceğini belirler
    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // Keyfi bir koşul ile işlemin gerçekleşip gerçekleşmeyeceğini
        // belirler ve sonucu return eder.
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

3.9.2 Görünürlük ve Getter Fonksiyonlar

Durum Değişkenlerinde Görünürlük

public
: Public durum değişkenleri internallerden sadece bir açıdan farklıdır, o da derleyicinin direkt olarak bir *getter fonksiyon* oluşturmasıdır. Bu şekilde diğer akıllı sözleşmeler bu değerlere erişebilir. Aynı fonksiyon içerisinde external erişim sağlandığında (örneğin, `this.x`) getter fonksiyonu çağrılırken, internal erişimde (örneğin, `x`) değer direkt olarak storage'den alınır. Setter fonksiyonlar derleyici tarafından tanımlanmaz. Bu yüzden siz kendiniz bir setter fonksiyon eklemediyseniz diğer fonksiyonlar bu değişkeni değiştiremez.

internal
: Internal durum değişkenleri sadece tanımlandıkları akıllı sözleşmeler ve o akıllı sözleşmelerden türetilen (inherited) akıllı sözleşmeler tarafından erişebilir durumdadır. External erişim mümkün değildir. Bütün durum değişkenlerinin default hali internal'dir.

private
: Private durum değişkenlerine sadece tanımlandıkları akıllı sözleşmeden erişim mümkündür. Internal'den farklı olarak türetilen akıllı sözleşmelerden de erişilemez.

> **Uyarı:** Bir şeyi `private` veya `internal` yapmak sadece diğer akıllı sözleşmelerin o bilgiye erişimini veya değiştirilmesini engeller. Ama bu bilgiler blok zinciri dışından erişilebilir durumdadır.

Fonksiyonlarda Görünürlük

Solidity iki tip fonksiyon çağrısı bilir: gerçek bir EVM mesaj çağrısı yapan external'lar ve bu çağrıyı yapmayan internal'lar. Ayrıca internal fonksiyonlar türetilen fonksiyonlardan erişilemez hale de getirilebilir. Bu da ortaya dört çeşit fonksiyon görünürlüğü çıkarır.

external
: External fonksiyonlar akıllı sözleşme interface'inin bir parçasıdır, bu da demektir ki diğer akıllı sözleşmeler ve transactionlar tarafından çağrılabilirler. Bir `f` external fonksiyonu internal olarak çağrılamaz (yani, `f()` işe yaramaz, ama `this.f()` çalışır).

public
: Public fonksiyonlar akıllı sözleşme interface'inin bir parçasıdır ve internal olarak veya mesaj çağrıları ile kullanılabilirler.

internal
: Internal fonksiyonlar sadece tanımlandıkları akıllı sözleşmeden veya o akıllı sözleşmeden türetilen akıllı sözleşmeler tarafından erişilebilir. External olarak erişim mümkün değildir. ABI kullanılarak external olarak erişilemese bile mapping ve storage referanslarını parametre olarak alabilirler.

private
: Private fonksiyonlar internaller gibidir ama bunlara türetilen fonksiyonlardan da erişim mümkün değildir.

> **Uyarı:** Bir şeyi `private` veya `internal` yapmak sadece diğer akıllı sözleşmelerin o bilgiye erişimini veya değiştirilmesini engeller. Ama bu bilgiler blok zinciri dışından erişilebilir durumdadır.

Görünürlük parametreleri durum değişkenleri için değişkenin tipinden sonra yazılırken fonksiyonlarda parametreler ve return tanımının arasına yazılır.

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

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

Aşağıdaki örnekte, `D`, `c.getData()` çağrısı yapabilir ve `data` değerini elde eder, ama `f` fonksiyonunu çağıramaz. `E` akıllı sözleşmeleri ise `C` akıllı sözleşmesinden türetildiği için `compute` fonksiyonunu çağırabilir.

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

contract C {
    uint private data;

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

// Bu akıllı sözleşme derlenemez, hata verir
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // hata: `f` görünür değil
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // hata: `compute` görünür değil
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // internal fonksiyona türetilen fonksiyon sayesinde␣
↪erişim sağlanabilir
    }
}
```

Getter Fonksiyonlar

Derleyici bütün **public** durum değişkenleri için getter fonksiyonu oluşturur. Örneğin aşağıdaki akıllı sözleşme için, derleyici `data` adında bir fonksiyon üretir. Bu fonksiyon hiçbir parametre almaz ve `uint` tipinde bir değişken return eder. Return edilen değer ise `data` değişkeninde saklanan değerdir. Durum değişkenleri tanımlandıkları yerde initialize edilebilir (initialize, bir değişkenin ilk defa tanımlanması olarak çevrilebilir).

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

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

Getter fonksiyonların görünürlüğü external'dir. Eğer internal olarak erişim sağlandıysa (`this.` olmadan), bu bir durum değişkenine erişim anlamına gelir. Eğer external olarak erişildiyse (`this.` kullanarak), bu getter fonksiyonuna erişim anlamına gelir.

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

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // internal erişim
        return this.data(); // external erişim
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }
}
```

Eğer bir `public` görünürlüğe sahip dizi tipinden bir durum değişkenine sahipseniz, getter fonksiyonunu kullanarak sadece tek bir elemana erişim sağlayabilirsiniz. Bu mekanizma tüm diziyi return ederken oluşan yüksek gaz ücretlerinden sıyrılmak için kurulmuştur. Hangi elemanın return edileceğini belirtmek için parametreleri kullanabilirsiniz (örneğin, `myArray(0)`). Eğer bütün diziyi tek bir fonksiyon ile elde etmeniz gerekiyorsa, bunun için aşağıdaki gibi bir fonksiyon yazmanız gerekir.

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

contract arrayExample {
    // public durum değişkeni
    uint[] public myArray;

    // Derleyici tarafından tanımlanan getter fonksiyonu
    /*
    function myArray(uint i) public view returns (uint) {
        return myArray[i];
    }
    */

    // Bütün array'i return eden fonksiyon
    function getArray() public view returns (uint[] memory) {
        return myArray;
    }
}
```

Artık tüm dizine erişmek için her bir aramayı tek bir öğeye döndüren `myArray(i)` yerine `getArray()` kullanabilirsiniz.

Sıradaki örnek biraz daha karmaşık.

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

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

Derleyici bize aşağıdaki gibi bir getter fonksiyonu oluşturur. Struct'daki mapping'ler ve diziler (byte dizileri istisnadır) gözardı edilmiştir. Çünkü getter fonksiyonlarında onların spesifik bir elemanına uygun bir şekilde erişim mümkün değildir.

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

3.9.3 Fonksiyon Modifier'ları

Modifier'lar fonksiyonların tanımlandığı şekillerinden farklı davranmalarını sağlamak için kullanılabilir. Örneğin, bir fonksiyonun çalıştırılmasından hemen önce bir koşulun kontrolünü gerçekleştirebilirsiniz.

Modifier'lar akıllı sözleşmelerin türetilebilen özelliklerindendir. Bu yüzden türetilmiş bir akıllı sözleşme bir modifier'ı eğer `virtual` olarak belirtilmişse onu override edebilir. Daha fazla bilgi için *Modifier Overriding* kısmına bakabilirsiniz.

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

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

    // Bu akıllı sözleşme sadece bir tane modifier tanımlar ve onu da kullanmıyor.
    // Tanımlanan modifier türetilen fonksiyonda kullanılacaktır.
    // Fonksiyon içerisindeki kodların kullanılacağı yer
    // `_;` şeklinde modifier içerisinde belirtilir.
    // Yani `_;` gördüğünüz yerde o modifier'ın kullanıldığı fonksiyonun
    // içerisindeki kodlar yazılmış gibi düşünebilirsiniz.
    // Bu modifier eklendiği fonksiyonu sadece akıllı sözleşmeyi oluşturan
    // kişinin çağırmasını sağlar. Diğer erişimlerde ise işlemi revert eder.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract destructible is owned {
    // Bu akıllı sözleşme türetildiği `owned` fonksiyonundaki
    // `onlyOwner` modifier'ını `destroy` fonksiyonuna ekler.
    // Böylece `destroy` fonksiyonunu sadece `owner` çağırabilir.
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    // Modifier'lar parametre alabilir:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

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

    constructor(uint initialPrice) { price = initialPrice; }

    // Buradaki `payable` sözcüğü de oldukça önemlidir.
    // Eğer bu fonksiyon `payable` olmazsa kendisine gelen bütün
    // etherleri reddeder.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

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

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

    /// Bu fonksiyon bir mutex ile korunmaktadır.
    /// Yani, bu akıllı sözleşme re-entrancy çağrılarına karşı zaafiyetli değildir.
    /// `return 7` fonksiyonun bittiğini belirtse de henüz modifier'ımızın işi bitmedi.
    /// `locked = false;` satırı return ifademizden sonra çalışır.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

Eğer `C` akıllı sözleşmesindeki `m` modifier'ına erişmek istiyorsanız, `C.m` şeklinde erişebilirsiniz. Modifier'lar sadece tanımlandıkları akıllı sözleşmede veya türetilen bir akıllı sözleşmede kullanılabilir. Modifier'lar kütüphanelerde de tanımlanabilir. Ancak kullanımları o kütüphanenin fonksiyonlarıyla kısıtlıdır. Yani tanımlandıkları kütüphane dışında kullanılamazlar.

Bir fonksiyona birden fazla modifier tanımlanabilir. Bunu gerçekleştirmek için her bir modifier isminden sonra bir boşluk bırakılmalıdır. Modifier'lar tanımlandıkları sıraya göre çalışacaktır.

Modifier'lar eklendikleri fonksiyonların parametrelerine veya return değerlerine kendi başlarına erişemezler. Eğer bir parametreyi bir modifier'da kullanmak istiyorsanız, o modifier'ı eklediğiniz yerde parametreyi de vermelisiniz. Fonksiyon çağırma yapısına benzer bir şekilde kullanılırlar.

Modifier'daki veya fonksiyon'daki return işlemi sadece o yazıldığı modifier'dan veya fonksiyon'dan çıkmaya yarar. Program akışı _ işaretinin olduğu yerden çalışmaya devam eder.

Uyarı: Daha önceki Solidity versiyonlarında modifier'a sahip fonksiyonlarda `return` ifadesi farklı bir şekilde davranış sergiler.

Açık bir şekilde `return;` ifadesinin yer aldığı bir modifier, fonksiyonun return edeceği değerle alakalı değildir. Modifier'lar fonksiyon içerisindeki kodları hiç çalıştırmamayı da tercih edebilirler. Bu durumda return değerleri *default değerlerine* eşitlenebilir. Böylelikle, fonksiyonun hiç bir kodu yokmuş gibi bir davranış sergilenir.

_ sembolü bir modifier'da birden fazla kez kullanılabilir. Her bir kullanım, fonksiyon içerisindeki kodla değiştirilecektir. Yani, _ gördüğünüz her yerde, eklenen fonksiyonun kodlarının bulunduğunu düşünebilirsiniz.

Modifier'lar parametre alabildiği için, bir fonksiyondaki bütün parametreler istenilen modifier'a gönderilebilir. Modifier'da tanımlanan semboller, fonksiyonlarda görülemez (override ile değiştirilebilir).

3.9.4 Constant ve Immutable State Değişkenleri

State değişkenleri `constant` veya `immutable` olarak tanımlanabilir. Her iki durumda da akıllı sözleşme kurulduktan sonra (constructor çalıştıktan sonra) bu tür değişkenler değiştirilemez. `constant` compile-time'da (kodun içerisinde) tanımlı olması gerekirken, `immutable` değişkenler constructor içerisinde de tanımlanabilir.

`constant` değişkenleri akıllı sözleşmelerin dışarısında (dosya seviyesinde) da tanımlayabiliriz.

Derleyici bu tür değişkenler için storage'de slot ayırmaz. Çünkü bu değişkenlerin kullanıldığı her yer, belirlenmiş değerle değiştirilir.

Normal state değişkenleriyle karşılaştırıldığında, constant ve immutable değişkenler çok daha az gaz harcar. Constant değişkenlerde, kullanıldıkları her yere karşılığında verilen değer kopyalanıp yapıştırılır. Bu, lokal optimizasyon olarak kullanılır. Immutable değişkenlerde ise, akıllı sözleşmenin kurulum anında (construction time) karşılık gelen değeri belirlenir ve kullanıldığı her yere kopyalanıp yapıştırılır. Bu değerler 32 byte'dan daha az yer kaplasa bile 32 byte'lık bir alanda muhafaza edilir. Bu sebepten ötürü, bazı durumlarda constant değerler kullanmak, immutable değerleri kullanmaktan daha ucuz olabilir.

Şu anda constant ve immutable bütün tipler için uygulanamamaktadır. Desteklenen tipler *strings* (sadece constant) ve *değer tipleridir*.

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

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    constructor(uint decimals_, address ref) {
        decimals = decimals_;
        // Immutable tanımlamalarında blok zincirinden veri de okunabilir.
        maxBalance = ref.balance;
    }

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

Constant

`constant` değişkenlerin değerleri derleme anında (compile-time) sabit olmalı ve değişkenin tanımlandığı konumda belirtilmelidir. Herhangi bir storage'e erişim, blok zinciri verisi (örneğin, `block.timestamp`, `address(this).balance` veya `block.number`) veya çalıştırma verisi (`msg.value` veya `gasleft()`) veya başka akıllı sözleşmelere yapılan external çağrılara izin verilmez. Kullanılacak memory'i belirleme konusunda yan etki oluşturacak tanımalamalara izin verilirken, başka memory objeleri üzerinde yan etki oluşturan tanımlamalara izin verilmez. Built-in fonksiyonlarından `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` ve `mulmod` fonksiyonlarının kullanımına izin verilmiştir (`keccak256` başka akıllı sözleşmeleri çağırsa da, bir istisnadır).

Memory belirleyicisi üzerinde yan etkiye izin verilmesinin sebebi, karmaşık yapılarında kurulabilinmesi gereksinimidir (örneğin, lookup-table). Bu özellikler henüz tamamen kullanılabilir değildir.

Immutable

`immutable` olarak tanımlanan değişkenler `constant` olarak tanımlananlara göre biraz daha az kısıtlanmıştır: Immutable değişkenler akıllı sözleşmenin constructor fonksiyonunda keyfi bir değere atanabilir. Sadece bir kere tanımlanabilirler ve tanımlandıktan sonra istenilen anda sahip oldukları değer okunabilir.

Derleyici tarafından oluşturulmuş akıllı sözleşmenin creation code'u, runtime code'u return etmeden önce bütün immutable referanslarını tanımlanan değerle değiştirir. Bu yüzden immutable değişken kullandığınız bir akıllı sözleşme için, compiler'ın oluşturduğu runtime code ile blok zincirinde saklanan runtime code'u karşılaştırdığınızda farklı sonuçlar alırsınız.

Not: Tanımlandıkları satırda direkt olarak değerleri atanan immutable değişkenler akıllı sözleşmenin constructor fonksiyonu çalıştıktan sonra initialize edilmiş olarak düşünülür. Bu demek oluyor ki başka bir immutable değişkenin değerini kullanan bir immutable değişkenin değerini direkt olarak atayamazsınız. Bunu ancak constructor içerisinde yapabilirsiniz.

Bu state değişkenlerini ilk defa tanımlama sırasının farklı bir şekilde yorumlanmasını engellemek amacıyla konulmuş bir önleyicidir, özellikle de türetme (inheritance) konusunda.

3.9.5 Fonksiyonlar

Fonksiyonlar akıllı sözleşmelerin içerisinde veya dışarısında tanımlanabilir.

Akıllı sözleşmelerin dışarısında tanımlanan fonksiyonlara "özgür fonksiyonlar" denir ve her zaman `internal` *görünürlüktedirler*. Kodları, o fonksiyonları kullanan her bir akıllı sözleşmeye eklenir, tıpkı internal kütüphane fonksiyonları gibi.

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

function sum(uint[] memory arr) pure returns (uint s) {
    for (uint i = 0; i < arr.length; i++)
        s += arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory arr) public {
        // Burada özgür bir fonksiyon internal olarak çağrılıyor.
        // Derleyici `sum` fonksiyonunu bu akıllı sözleşmenin kodları arasına ekleyecek.
        uint s = sum(arr);
        require(s >= 10);
        found = true;
    }
}
```

Not: Akıllı sözleşme dışında tanımlanan bir fonksiyon her zaman o akıllı sözleşmenin içeriği ile birlikte çalıştırılırlar. Hâlâ diğer akıllı sözleşmeleri çağırabilir, onlara Ether gönderebilir ve kendilerini çağıran akıllı sözleşmeleri yok edebilirler. Akıllı sözleşme içerisinde tanımlanan bir fonksiyon ile özgür bir fonksiyonun arasındaki en temel farklar özgür fonksiyonların `this` değişkenine erişimi olmaması, ve de kendi alanlarında (scope) bulunmayan storage değişkenlerine ve fonksiyonlara direkt erişime sahip olmamalarıdır.

Fonksiyon Parametreleri ve Return Parametreleri

Fonksiyonlar tipi belirtilmiş parametreler alabilir ve diğer birçok programlama dilinin aksine keyfi sayıda değişkeni return edebilirler.

Fonksiyon Parametreleri

Fonksiyon parametreleri değişkenlerle aynı şekilde tanımlanırlar. Ayrıca kullanılmayan parametreler gözardı edilebilirler.

Örneğin, eğer akıllı sözleşmenizdeki bir fonksiyonun iki adet integer değişkeni parametre olarak almasını isterseniz, aşağıdaki gibi bir yapı kullanabilirsiniz:

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

contract Simple {
    uint sum;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    function taker(uint a, uint b) public {
        sum = a + b;
    }
}
```

Fonksiyon parametreleri herhangi bir lokal değişken olarak kullanılaiblir ve ayrıca lokal değişkenlere atanabilirler.

Not: Bir *external fonksiyon* çok boyutlu bir diziyi parametre olarak alamazlar. Bu özelliği eğer ABI coder v2'yi kaynak kodunuzda `pragma abicoder v2;` bu şekilde aktifleştirdiyseniz kullanabilirsiniz.

Bir *internal fonksiyon* o özelliği aktifleştirmeden de çok boyutlu bir diziyi parametre olarak alabilir.

Return Değişkenleri

Fonksiyon return değişkenleri aynı şekilde `returns` sözcüğünden sonra tanımlanır.

Örneğin, iki adet sonucu return etmek istediğinizi düşünün: fonksiyon parametresi olarak verilmiş iki adet integer'ın toplamı ve çarpımı. Şu şekilde bir kod işinizi görecektir:

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

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        sum = a + b;
        product = a * b;
    }
}
```

Return değişkenlerinin tipleri gözardı edilebilirler. Return değişkenleri herhangi bir lokal değişken olarak kullanılabilirler. Bu değişkenler direkt olarak *default değerine* eşitlenir ve değiştirilene kadar bu değere eşit olurlar.

İsterseniz yukarıdaki gibi açık bir şekilde return değişkenlerinin değerlerini verebilir veya aşağıdaki gibi direkt olarak `return` ifadesini kullanabilirsiniz (ister tek, isterseniz de *çoklu return*):

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

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        return (a + b, a * b);
    }
}
```

Eğer fonksiyondan çıkmak için erkenden `return` kullanmanak istiyorsanız, bütün return değişkenlerini vermeniz gerekir.

Not: Bazı tipleri internal olmayan fonksiyonlardan return edemezsiniz, örneğin, çok boyutlu dinamik boyutlu diziler ve structlar. Eğer ABI coder v2'yi `pragma abicoder v2;` şeklinde kodunuza eklerseniz daha fazla tip kullanılabilir olacaktır, ancak `mapping` tipi hâlâ bir akıllı sözleşme içerisinde sınırlıdır ve onları transfer edemezsiniz.

Çoklu Değer Return Etme

Bir fonksiyonda birden fazla değişkeni return etmek istiyorsanız `return (v0, v1, ..., vn)` şeklinde bir ifade kullanabilirsiniz. Return değişkeni sayısı ve tipleri, bir *implicit dönüşümden* sonra belirtilen değerlerle eşleşmelidir.

State Değişkenliği

View Fonksiyonlar

`view` ile tanımlanan fonksiyonlar state'te herhangi bir değişikliği yapamaz, sadece state'deki değerleri okuyabilirler.

Not: Eğer derleyicinin EVM target kısmı Byzantium veya daha yenisi (default) ise `view` fonksiyonlar çağrıldığında `STATICCALL` opcode'u kullanılır ve bu opcode state'i değişmemeye zorlar. Kütüphanelerdeki `view` fonksiyonlarında ise `DELEGATECALL` kullanılır. Çünkü `DELEGATECALL` ve `STATICCALL` opcode'larından kombine edilmiş bir opcode bulunmamaktadır. Bu demek oluyor ki `view` fonksiyonlar state değişikliğini önlemek için run-time kontrollerine sahip değildirler. Bunun kötü bir güvenlik etkisi olmamalıdır. Çünkü kütüphane kodu genellikle derlenirken bilinir ve statik kontrol edici (static checker) compile-time kontrollerini gerçekleştirir.

Aşağıdaki ifadeler state değişikliğini temsil eder:

1. State değişkenlerine yazmak.
2. *Event yayınlama*.
3. *Başka akıllı sözleşmeler oluşturma*.
4. `selfdestruct` kullanmak.
5. Ether göndermek.
6. `view` veya `pure` olarak belirtilmeyen bir fonksiyon çağırmak.
7. Low-level çağrılar kullanmak.
8. Belirli opcode'ları kullanan inline assembly kullanmak.

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

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

Not: Versiyon 0.5.0 öncesinde fonksiyonlarda `constant` sözcüğü şu anki `view` için kullanılırdı, ancak artık kullanılmıyor.

Not: Getter fonksiyonlar otomatik olarak `view` görünürlüğüne sahip olur.

Not: Versiyon 0.5.0 öncesinde derleyici `view` için `STATICCALL` opcode'unu kullanmazdı. Bu, `view` fonksiyonlarda yanlış explicit tip dönüşümlerini kullanarak state değişikliği yapılmasına izin verdi. `STATICCALL` opcode'unu `view` fonksiyonlar için kullanarak EVM seviyesinde state değişikliklerinin yapılmasının önüne geçildi.

Pure Fonksiyonlar

Fonksiyonlar `pure` olarak tanımlanabilir ve bu şekilde tanımlanan fonksiyonlar state'i okuyamaz ve değişiklik yapamaz. Pure fonksiyonlar içerisinde `immutable` değişkenler okuyabilir durumdadır.

Not: Eğer derleyicinin EVM target kısmı Byzantium veya daha yeni (default) ise, `STATICCALL` opcode'u kullanılır. Bu opcode state'in okunmadığına dair garanti vermez ama en azından değiştirilmediğine dair bir garanti verir.

Yukarıda state'i değiştiren ifadeleri açıklamışken, state'i okuduğu düşünülen ifadeleri de aşağıda bulabilirsiniz:

1. State değişkenlerini okumak.
2. `address(this).balance` veya `<address>.balance` değişkenlerine erişmek.
3. `block`, `tx` veya `msg` değişkenlerinin herhangi bir üyesine erişmek (`msg.sig` ve `msg.data` istisnadır).
4. `pure` olmayan herhangi bir fonksiyonu çağırmak.
5. Belirli opcode'ları kullanan inline assembly kullanmak.

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

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

Pure fonksiyonlar `revert()` ve `require()` ifadelerini kullanarak *hata oluşması* durumunda potansiyel state değişikliğini engelleyebilirler.

State değişikliğini revert etmek bir "state değişikliği" olarak düşünülmez.

Bir state değişikliğini revert etmek bir "state değişikliği" olarak kabul edilmez, çünkü yalnızca daha önce kodda `view` veya `pure` kısıtlamaya sahip olmayan state'de yapılan değişiklikler revert edilir ve bu kodun `revert`'i yakalama ve aktarmama seçeneği vardır.

Bu davranış `STATICCALL` için de geçerlidir.

Uyarı: EVM seviyesinde fonksiyonların state'den okuma yapmasını engellemek mümkün değildir, sadece yazma engellenebilir (yani, EVM seviyesinde sadece `view` zorunlu kılınabilir, `pure` kılınamaz).

Not: Versiyon 0.5.0 öncesinde derleyici `pure` için `STATICCALL` opcode'unu kullanmazdı. Bu, `pure` fonksiyonlarda yanlış explicit tip dönüşümlerini kullanarak state değişikliği yapılmasına izin verdi. `STATICCALL` opcode'unu `pure` fonksiyonlar için kullanarak EVM seviyesinde state değişikliklerinin yapılmasının önüne geçildi.

Not: Versiyon 0.4.17 öncesinde derleyici `pure` fonksiyonların state'i okuması durumunda hata vermezdi. Bu, sözleşme türleri arasında geçersiz açık dönüşümler yaparak atlatılabilen ve bir tür denetim olan derleme zamanı yüzünden kaynaklanmaktaydı. Çünkü derleyici, sözleşme türünün durum değiştirme işlemleri yapmadığını doğrulayabilir, fakat çalışma zamanında çağrılacak olan sözleşmenin gerçekten bu türden olup olmadığını kontrol edemez.

Özel Fonksiyonlar

Receive Ether Fonksiyonu

Bİr akıllı sözleşme sadece bir adet `receive` fonksiyonuna sahip olabilir. Bu fonksiyon şu şekilde tanımlanır: `receive() external payable { ... }` (function sözcüğü olmadan). Bu fonksiyon parametre alamaz, hiçbir şey return edemez, görünürlüğü `external` olmalı ve ayrıca `payable` olarak tanımlanmalıdır. Bir receive fonksiyonu virtual olabilir, override edilebilir ve modifier'lara sahip olabilir.

Receive fonksiyonu akıllı sözleşmemize gelen boş bir calldata'sı bulunan çağrılarda çalıştırılır. Bu fonksiyon, akıllı sözleşmemize direkt Ether transferi gerçekleştirildiğinde (`.send()` veya `.transfer()` kullanılarak) çalıştırılır. Eğer bu fonksiyon tanımlı değil ama payable bir *fallback fonksiyon* tanımlı ise, direkt Ether transferlerinde bu fallback fonksiyonu çalıştırılır. Eğer akıllı sözleşme ne bir receive fonksiyonu, ne de bir payable fallback fonksiyonu tanımlamamışsa, akıllı sözleşmemiz direkt Ether transflerlerini kabul edemez, kendisine ether gönderildiğinde bir hata verir.

En kötü durumda `receive` fonksiyonu 2300 adet gazın mevcut olduğunu varsayabilir (örneğin `send` veya `transfer` kullanımında), geriye ise sadece log işlemleri gibi basit işlemler için gaz kalır. Aşağıdaki işlemler 2300 gazdan daha fazlasını harcar:

- Storage'e yazmak
- Akıllı sözleşme oluşturmak
- Yüksek miktarda gaz harcayan bir external fonksiyonun çağrılması
- Ether gönderimi

Uyarı: Bir akıllı sözleşmede direkt olarak Ether gönderirken (bir fonksiyon çağrısı olmadan, yani gönderenin `send` veya `transfer` kullandığı durumda) eğer akıllı sözleşme bir receive fonksiyonu veya bir payable fallback fonksiyonu tanımlamamışsa, bir hata oluşur ve Etherler gönderene iade edilir (bu durum Solidity 0.4.0 öncesinde farklıydı). Eğer akıllı sözleşmenizin direkt Ether transferlerini kabul etmesini istiyorsanız, bir receive fonksiyonu tanımlayın (Ether kabulu için payable fallback fonksiyonunun kullanımını tavsiye etmiyoruz, çünkü fallback fonksiyonu interface karışıklığı yaşandığında kullanıcıya hata vermeyecektir).

Uyarı: Bir akıllı sözleşme receive fonksiyonu olmadan da Ether kabul edebilir; *coinbase transaction* (diğer adıyla *miner block reward*) veya `selfdestruct` kullanılırken hedef adres olarak verilmesi halinde akıllı sözleşme Etherleri kabul etmek zorundadır.

Bir akıllı sözleşme bu gibi durumlardaki Ether transferlerine herhangi bir tepki veremez ve dolayısıyla bunları reddedemez. Bu EVM'in tasarım tercihlerinden biridir ve Solidity bunu es geçemez.

Bu ayrıca demek oluyor ki `address(this).balance` değişkenindeki değer sizin kendi hesaplamanızla (örneğin, receive fonksiyonunda her gelen miktarı hesaplamanız halinde) farklı olabilir.

Aşağıdaki Sink akıllı sözleşmesi `receive` kullanımına bir örnektir.

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

// Bu akıllı sözleşmeye gönderilen Etherleri geri almanın hiçbir
// yolu yoktur.
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Fallback Fonksiyonu

Bir akıllı sözleşme sadece bir adet `fallback` fonksiyonuna sahip olabilir. Bu fonksiyon şu iki şekilde tanımlanabilir: `fallback () external [payable]` veya `fallback (bytes calldata input) external [payable] returns (bytes memory output)` (ikisi de `function` sözcüğü olmadan kullanılıyor). Bu fonksiyon `external` görünürlüğe sahip olmalıdır. Bir fallback fonksiyonu virtual olabilir, override edilebilir ve modifier'lara sahip olabilir.

Fallback fonksiyonu bir çağrıda gönderilen fonksiyon imzasının (function signature) akıllı sözleşmedeki herhangi bir fonksiyon ile eşleşmediği durumda çalıştırılır, yani, eğer kullanıcının çalıştırmak istediği fonksiyon akıllı sözleşmede yoksa, fallback fonksiyonu çalıştırılır. Bir diğer kullanım alanı ise direkt Ether gönderimlerinde eğer akıllı sözleşmede *receive Ether fonksiyonu* yoksa ve fallback fonksiyonumuz `payable` ise, fallback fonksiyonu çalıştırılır.

Eğer yukarıda gösterdiğimiz iki kullanım şeklinden `input` kullanılanı kullanmak isterseniz, `input` akıllı sözleşmeye gönderilen tüm data, `msg.data`, olacaktır. Ayrıca `output` ile de data return edebilir. Return edilen data ABI-encoded olmayacaktır, onun yerine herhangi bir düzenleme olmadan (hatta padding bile olmadan) return edilecektir.

En kötü durumda, eğer bir payable fallback fonksiyonu receive fonksiyonun da yerine kullanıldıysa, sadece 2300 adet gaz ile işlemini tamamlayabilir (*receive Ether fonksiyonu*).

Diğer herhangi bir fonksiyon gibi fallback fonksiyonu da yeterli gaza sahip olduğu sürece çok karmaşık işlemleri yürütebilir.

Uyarı: Bir `payable` fallback fonksiyonu ayrıca direkt Ether transferlerinde de, eğer *receive Ether fonksiyonu* kullanılmadıysa, çalıştırılabilir. Eğer payable fallback fonksiyonuna spesifik bir kullanım için ihtiyacınız yoksa, receive fonksiyonunu kullanmanızı tavsiye ederiz.

Not: Eğer input verisini decode etmek istiyorsanız, ilk dört byte'ı fonksiyon imzası için kullanabilir ve kalan kısmı `abi.decode` kullanarak ABI-encoded veriyi decode edebilirsiniz: `(c, d) = abi.decode(input[4:], (uint256,`

uint256)); Şunu unutmayın ki, bu bir son çaredir. Eğer yapabiliyorsanız daha uygun bir fonksiyon kullanmaya çalışın.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Test {
    uint x;
    // Bu akıllı sözleşmeye gelen bütün mesaj çağrılarını
    // bu fonksiyon karşılar (akıllı sözleşmede başka bir
    // fonksiyon bulunmadığı için).
    // Fonksiyon payable olarak belirtilmediği için
    // Ether gönderimlerinde hata alınacaktır.
    fallback() external { x = 1; }
}

contract TestPayable {
    uint x;
    uint y;
    // Bu akıllı sözleşmeye gelen direkt Ether gönderimleri dışındaki bütün mesajları
    // bu fonksiyon karşılayacaktır (receive dışında başka bir fonksiyon
    // bulunmamakta). Calldatası boş olmayan bütün çağrıları bu fonksiyon
    // karşılar (çağrı ile birlikte Ether gönderilse bile).
    fallback() external payable { x = 1; y = msg.value; }

    // Bu fonksiyon sadece direkt Ether gönderimleri için kullanılır, yani,
    // boş bir calldata ve Ether gönderilen çağrıları bu fonksiyon karşılar.
    receive() external payable { x = 2; y = msg.value; }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // test.x'in == 1 olmasına neden olur.

        // address(test) direkt olarak ``send`` kullanımına izin vermez.
        // ``send`` fonksiyonunu çağırabilmek için bile ``address payable``
        // tipine dönüştürme gerekmektedir.
        address payable testPayable = payable(address(test));

        // Eğer biri burada da olduğu gibi payable fallback fonksiyonu olmayan bir
        // akıllı sözleşmeye ether göndermeye çalışırsa, hata alacaktır.
        // Dolayısıyla burada ``false`` return edilir.
        return testPayable.send(2 ether);
    }

    function callTestPayable(TestPayable test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // test.x == 1 olur ve test.y 0 olur.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        (success,) = address(test).call{value: 1}(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // test.x == 1 olur ve test.y 1 olur.

        // Eğer biri aşağıdaki gibi TestPayable akıllı sözleşmesine Ether gönderirse,␣
↪receive fonksiyonu çalışır.
        // Yukarıda tanımladığımız receive fonksiyonu storage'e yazdığı için 2300'den␣
↪daha fazla
        // gaz harcanmasına sebep olur. O yüzden ``send`` ve ``transfer`` kullanılamaz.
        // Onların yerine low-level call kullanmalıyız.
        (success,) = address(test).call{value: 2 ether}("");
        require(success);
        // test.x'in == 2 ve test.y'nin 2 Ether olmasıyla sonuçlanır.

        return true;
    }
}
```

Fonksiyon Overloading

Bir akıllı sözleşme aynı isimde fakat farklı parametre tiplerine sahip fonksiyonlara sahip olabilir. Bu işlem "overloading" olarak adlandırılır ve ayrıca türetilen fonksiyonlar için de geçerlidir. Aşağıdaki örnek `A` akıllı sözleşmesindeki `f` fonksiyonları ile overloading'i gösterir.

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

contract A {
    function f(uint value) public pure returns (uint out) {
        out = value;
    }

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

Overload edilmiş fonksiyonlar external interface'de de göründüğü için iki fonksiyonun aldığı parametreler external tiplerine göre karşılaştırılır. Yani, örneğin aşağıdaki fonksiyonlardan biri parametre olarak akıllı sözleşme aldığını belirtmiş. Ancak external interface'de bu, bir akıllı sözleşme değil, adres olarak görünür. O yüzden bu akıllı sözleşme compile edilemez.

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

// Compile edilemez
contract A {
    function f(B value) public pure returns (B out) {
        out = value;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }

    function f(address value) public pure returns (address out) {
        out = value;
    }
}

contract B {
}
```

Yukarıdaki iki `f` fonksiyonu da ABI'leri aracılığı ile address tipinden bir parametre kabul ediyor, her ne kadar Solidity içerisinde farklı tipler kabul etseler de.

Overload Ayrıştırma ve Parametre Eşleştirme

Overload edilmiş fonksiyonlar, geçerli kapsamdaki fonksiyon tanımlamalarını fonksiyon çağrısında sağlanan parametrelerle eşleştirerek seçilir. Tüm parametreler implicit olarak beklenen türlere dönüştürülebiliyorsa, fonksiyon overload adayı olarak seçilir. Tam olarak bir aday yoksa, çözümleme başarısız olur.

Not: Overload ayrıştırma için return parametreleri dikkate alınmaz.

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

contract A {
    function f(uint8 val) public pure returns (uint8 out) {
        out = val;
    }

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

`f(50)` çağrısını yaptığımızda bir hata alırız. Bunun sebebi `50` sayısının hem `uint8` hem de `uint256` tipinde de kullanılabilmesidir. Ama eğer `f(256)` çağrısını gerçekleştirirsek `256` sayısı direkt olarak `f(uint256)` bu şekilde tanımlanan fonksiyona gönderilir. Çünkü `256` `uint8` olarak gösterilemez.

3.9.6 Eventler

Solidity eventleri EVM'nin loglama işlevinin üzerine bir soyutlama verir. Uygulamalar Ethereum clientlarının RPC arayüzüne abone olarak bu eventleri dinleyebilirler.

Eventler akıllı sözleşmelerin türetilebilen üyeleridir. Çağrıldıklarında işlemlerin log kısmında - blok zincirindeki özel bir veri yapısı - depolanırlar. Bu eventler çağrıldıkları akıllı sözleşmenin adresi ile özdeşleştirilir ve işlemin bulunduğu blok erişilebilir olduğu sürece bu eventlere de erişilebilir (şu anda bu süre sonsuza kadardır ancak Serenity ile bu değişebilir). Log ve event verisi akıllı sözleşme tarafından erişilebilir değildir (eventi oluşturan akıllı sözleşme için bile bu geçerlidir).

Loglar için bir Merkle proof talep etmek mümkündür, bu nedenle external bir varlık böyle bir kanıtla bir akıllı sözleşme sağlarsa, logun blok zinciri içinde gerçekten var olup olmadığını kontrol edebilir. Sözleşme yalnızca son 256 blok hashini görebildiği için blok başlıkları sağlamanız gerekir.

Logun veri kısmı yerine *"topics"* olarak bilinen özel bir veri yapısına ekleyen en fazla üç parametreye `indexed` özniteliği ekleyebilirsiniz. Bir topic yalnızca tek bir kelimeyi (32 byte) tutabilir, bu nedenle indekslenmiş bir argüman için bir referans tipi kullanırsanız, bunun yerine değerin Keccak-256 hashi topic olarak saklanır.

`indexed` olmadan kullanılan bütün parametreler logun veri kısmına *ABI-encoded* olarak saklanır.

Topicler eventleri aramanıza izin verir, örneğin belirli eventler için bir blok dizisini filtrelerken. Ayrıca eventleri yayınlandıkları akıllı sözleşmede göre de filtreleyebilirsiniz.

Örneğin aşağıdaki kod web3.js'in `subscribe("logs")` methodunu kullanarak logları belirli bir adrese göre filtreleme işlemi yapmıştır:

```
var options = {
    fromBlock: 0,
    address: web3.eth.defaultAccount,
    topics: ["0x0000000000000000000000000000000000000000000000000000000000000000", null,␣
↪null]
};
web3.eth.subscribe('logs', options, function (error, result) {
    if (!error)
        console.log(result);
})
    .on("data", function (log) {
        console.log(log);
    })
    .on("changed", function (log) {
});
```

Eventin imzasının hashi, etkinliği anonim belirteçle bildirmeniz dışında, topiclerden biridir. Bu, belirli anonim eventleri ada göre filtrelemenin mümkün olmadığı, yalnızca akıllı sözleşme adresine göre filtreleyebileceğiniz anlamına gelir. Anonim eventlerin avantajı, deploy etmenin ve çağırmanın daha ucuz olmasıdır. Ayrıca, üç yerine dört indexed değişken bildirmenize olanak tanır.

Not: İşlem logları değişken türünü değil, yalnızca olay verilerini sakladığından, verileri doğru bir şekilde yorumlamak için hangi parametrenin dizine eklendiği ve eventin anonim olup olmadığı dahil olmak üzere olayın türünü bilmeniz gerekir. Özellikle, anonim bir event kullanarak başka bir eventin imzasını "sahte" yapmak mümkündür.

Eventlerin Üyeleri

- `event.selector`: Anonim olmayan eventlerde `bytes32` tipindeki bir değerdir ve eventin imzasının hashini içerir `keccak256`.

Örnek

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

contract ClientReceipt {
    event Deposit(
        address indexed from,
        bytes32 indexed id,
        uint value
    );

    function deposit(bytes32 id) public payable {
        // Eventler `emit` sözcüğü ve sonrasında
        // eventin ismi ve parametreleri (varsa) parantez
        // içerisine konularak yayınlanır.
        // Bu şekildeki herhangi bir çağırma işlemi
        // (iç içe olsa bile) `Deposit` ile filtreleme
        // yaparak JavaScript API tarafından yakalanabilir.
        emit Deposit(msg.sender, id, msg.value);
    }
}
```

JavaScript API kullanımı ise şu şekildedir:

```
var abi = /* derleyici tarafından üretilen ABI */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* adres */);

var depositEvent = clientReceipt.Deposit();

// değişiklikleri izle
depositEvent.watch(function(error, result){
    // sonuç, `Deposit` çağrısına verilen indekslenmemiş
    // argümanları ve topicleri içerir.
    if (!error)
        console.log(result);
});


// veya bir callback fonksiyonu ile direkt olarak dinlemeye başlayabilirsiniz
var depositEvent = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

Yukarıdaki kod şu şekilde bir çıktı verir (trim edilmiş hali ile):

```
{
   "returnValues": {
       "from": "0x1111...FFFFCCCC",
       "id": "0x50...sd5adb20",
       "value": "0x420042"
   },
   "raw": {
       "data": "0x7f...91385",
       "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
   }
}
```

Eventleri Anlamak İçin Ekstra Kaynaklar

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

3.9.7 Hata ve Geri Alma Durumları

Solidity'de hatalar gaz-verimli ve kullanışlı bir şekilde kullanıcılara bir işlemin neden başarısız olduğunu söylemeyi sağlar. Akıllı sözleşmenin içerisinde veya dışarısında tanımlanabilirler (interface ve kütüphaneler de dahil).

Revert ifadesi ile kullanılmalıdır. Bu ifade anlık çağrıda yapılan bütün değişiklikleri geri alır ve işlemi çağıran kişiye bir hata gönderir.

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

/// Transfer için yetersiz bakiye. `required` kadar bakiye
/// olmalıyken, `available` kadar bakiye mevcuttur.
/// @param available, kullanılabilir bakiye.
/// @param required, transfer edilmek istenen miktar.
error InsufficientBalance(uint256 available, uint256 required);

contract TestToken {
    mapping(address => uint) balance;
    function transfer(address to, uint256 amount) public {
        if (amount > balance[msg.sender])
            revert InsufficientBalance({
                available: balance[msg.sender],
                required: amount
            });
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    // ...
}
```

Hatalar overload veya override edilemez ama türetilebilirler. Alanları farklı olduğu sürece aynı hata birden fazla kere tanımlanabilir. Hata örnekleri sadece `revert` ifadesi kullanılarak üretilebilir.

Hata, daha sonra zincir dışı bileşene geri dönmek veya onu *try/catch ifadesiyle*. yakalamak için geri alma işlemiyle işlemi çağırana veri iletir. Bir hatanın yalnızca harici bir aramadan geldiğinde yakalanabileceğini, dahili aramalarda veya aynı işlevin içinde gerçekleşen geri dönüşlerin yakalanamayacağını unutmayın.

Herhangi bir parametre sağlamazsanız, hata yalnızca dört bayt veriye ihtiyaç duyar ve zincirde depolanmayan hatanın ardındaki nedenleri daha fazla açıklamak için *NatSpec'i* yukarıdaki gibi kullanabilirsiniz. Bu, bunu aynı zamanda çok ucuz ve kullanışlı bir hata raporlama özelliği yapar.

Daha spesifik olarak, bir hata örneği, aynı ad ve türdeki bir işleve yapılan bir işlev çağrısıyla aynı şekilde ABI ile kodlanır ve daha sonra geri alma işlem kodunda dönüş verileri olarak kullanılır. Bu, verilerin 4 baytlık bir fonksiyon selector'ünün ve ardından *ABI-encoded* verilerden oluştuğu anlamına gelir. Selector, hata türünün imzasının keccak256 hash'inin ilk dört baytından oluşur.

Not: Bir sözleşmenin aynı adı taşıyan farklı hatalarla veya hatta işlemi çağıran tarafından ayırt edilemeyen farklı yerlerde tanımlanan hatalarla geri dönmesi mümkündür. Dışarıdan, yani ABI için, tanımlandığı sözleşme veya dosya değil, yalnızca hatanın adı önemlidir.

`require(condition, "description");` ifadesi ile `if (!condition) revert Error("description")` ifadesi eğer hata `error Error(string)` bu şekilde tanımlanmışsa, aynı işi yapar. `Error` tipinin bir built-in tipi olduğunu ve kullanıcı tarafından tanımlanamayacağını unutmayın.

Benzer olarak bir `assert` ile tespit edilen bir başarısızlık, yine bir built-in tipi olan `Panic(uint256)` ile geri alınacaktır.

Not: Hata verileri sadece bir başarısızlığı işaret etmek için kullanılmalıdır, kontrol akışı için kullanılmamalıdır. Bunun nedeni, dahili çağrıların geri alınan verilerinin, varsayılan olarak harici çağrılar zinciri boyunca geri yayılmasıdır. Bu, bir iç çağrının, kendisini çağıran sözleşmeden gelmiş gibi görünen verileri "sahte" hale getirebileceği anlamına gelir.

Hataların Üyeleri

- `error.selector`: Hatanın selector'ünü içeren `bytes4` dört baytlık bir değer.

3.9.8 Kalıtım

Solidity polimorfizm dahil birçok kalıtım yöntemini destekler.

Polimorfizm, bir fonksiyon çağrısının (dahili ve harici), kalıtım hiyerarşisinde aynı fonksiyona sahip birden fazla akıllı sözleşmenin olması durumunda, ilk türetilen akıllı sözleşmenin fonksiyonunun çalıştırılmasına verilen isimdir. Bu, `virtual` ve `override` anahtar sözcükleri kullanılarak hiyerarşideki her işlevde açıkça etkinleştirilmelidir. Daha fazla ayrıntı için *Function Overriding'e* bakın.

Kalıtım hiyerarşisinden bir fonksiyonu çağırmak isterseniz; `ContractName.functionName()` bu şekilde çağırabilirsiniz. Veya kalıtım hiyerarşisinde bir üst akıllı sözleşmede bulunan bir fonksiyonu çağırmak isterseniz de; `super.functionName()` kullanabilirsiniz.

Bir akıllı sözleşme başka bir akıllı sözleşmeyi türettiğinde, blockchainde sadece bir adet akıllı sözleşme oluşturulur ve tüm ana akıllı sözleşmelerden gelen kodlar oluşturulan akıllı sözleşmeye eklenir. Bu demek oluyorki ana akıllı sözleşmelerin fonksiyonlarına yapılan bütün internal çağrılar sadece internal fonksiyon çağrılarını kullanırlar (`super.f(..)` sadece JUMP opcode'unu kullanacaktır, mesaj çağrısı yapmayacaktır).

Durum değişkeni gölgeleme bir hata olarak kabul edilir. Bir türetilen akıllı sözleşme sadece ve sadece eğer türettiği akıllı sözleşmelerden hiçbiri `x` isminde bir değişkeni kullanmıyorsa bu isimde bir değişken tanımlayabilir.

Genel kalıtım sistemi Python'a oldukça benzer, özellikle de çoklu kalıtım konusunda, fakat ayrıca bazı *farklılıklar* da bulunmaktadır

Aşağıdaki örnekte detaylar açıklanmıştır.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;


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


// `is` kullanarak başka bir akıllı sözleşmeyi türetebiliriz.
// Türetilen akıllı sözleşmeler private olmayan bütün üyelere
// erişebilir, internal fonksiyonlar ve durum değişkenleri
// dahil. Bunlara external olarak `this` kullanılarak da erişilemez.
contract Destructible is Owned {
    // `virtual` sözcüğü bu fonksiyonun, türetilen
    // akıllı sözleşmelerde değiştirilebileceğini belirtir ("overriding").
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


// Abstract akıllı sözleşmeler sadece derleyiciye interface'i
// bildirmek için kullanılır. Fonksiyonun kodlarının olmadığına
// dikkat edin. Eğer bir akıllı sözleşme bütün fonksiyonlarının içeriğini
// bulundurmazsa, sadece interface olarak da kullanılabilir.
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}


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


// Çoklu türetim de mümkündür. `Owned` akıllı sözleşmesinin
// ayrıca `Destructible` akıllı sözleşmesinin ana akıllı sözleşmelerinden
// biri olduğunu unutmayın. Ancak `Owned` akıllı sözleşmesinin
// sadece bir adet örneği vardır (C++'daki sanal kalıtım gibi).
contract Named is Owned, Destructible {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Fonksiyonlar başka bir fonksiyon tarafından aynı isim ve aynı
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    // sayıda/tipte girdi ile override edilebilir. Eğer override eden
    // fonksiyon farklı sayıda çıktı veriyorsa, bu ortaya bir hata çıkarır.
    // Hem yerel hem de mesaj-tabanlı fonksiyon çağrıları bu override işlemlerini
    // hesaba katar. Eğer bir fonksiyonu override etmek istiyorsanız
    // `override` sözcüğünü kullanmak zorundasınız. Ayrıca fonksiyonunuzun
    // tekrardan override edilebilir olmasını istiyorsanız, tekrardan
    // `virtual` olarak belirlemelisiniz.
    function destroy() public virtual override {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // Override edilmiş bir fonksiyonu spesifik olarak
            // çağırmak mümkündür.
            Destructible.destroy();
        }
    }
}


// Eğer bir constructor parametre alıyorsa, bu
// başlıkta veya değiştirici-çağırma-stili ile
// türetilen akıllı sözleşmesinin constructor'ında
// verilmelidir (aşağıya bakın).
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // Burada sadece `override` yazıyoruz, `virtual` yazmıyoruz.
    // Bu, `PriceFeed` akıllı sözleşmesinden türetilen akıllı sözleşmelerin
    // artık `destroy` fonksiyonunu override edemeyecekleri anlamına geliyor.
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

Yukarıdaki `Destructible.destroy()` fonksiyon çağrımızın bazı problemlere yol açtığını aşağıdaki örnekte görebilirsiniz.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

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

contract Destructible is owned {
    function destroy() public virtual {
        if (msg.sender == owner) selfdestruct(owner);
    }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ Destructible.
↪destroy(); }
}

contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ Destructible.
↪destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { Base2.destroy(); }
}
```

`Final.destroy()` çağrısı `Base2.destroy` fonksiyonunu çağıracak. Çünkü yaptığımız son override'da böyle belirtti. Ancak bu fonksiyon `Base1.destroy` fonksiyonunu bypass eder.

A call to `Final.destroy()` will call `Base2.destroy` because we specify it explicitly in the final override, but this function will bypass `Base1.destroy`. Bunu aşmanın yolu `super` kelimesini kullanmaktır:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

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

contract Destructible is owned {
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ super.destroy(); }
}


contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ super.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { super.destroy(); }
}
```

`Base2` , `super` işlevini çağırırsa, bu işlevi temel sözleşmelerinden birinde çağırmaz. Bunun yerine, son kalıtım grafiğindeki bir sonraki temel sözleşmede bu işlevi çağırır, bu nedenle `Base1.destroy()` u çağırır (son kalıtım dizisinin – en türetilmiş sözleşmeyle başlayarak şöyle olduğuna dikkat edin: Final, Base2, Base1, Destructible, owned). super kullanılırken çağrılan asıl işlev, türü bilinmesine rağmen kullanıldığı sınıf bağlamında bilinmemektedir. Bu, sıradan

sanal yöntem araması için benzerdir.

Fonksiyon Override Etme

Temel fonksiyonlar `virtual` olarak işaretlenmişse, davranışlarını değiştirmek için override edilebilirler. Override eden fonksiyon `override` olarak belirlenmelidir. Override edilen fonksiyonun görünürlüğü `external`'dan `public`'e dönüştürülebilir. Değişebilirlik ise daha fazla kısıtlandırılmış bir yapıya dönüştürülebilir: `nonpayable`, `view` ve `pure` tarafından override edilebilir. `view` ise `pure` tarafından override edilebilir. `payable` bir istisna olarak diğer değişebilirlik türlerine dönüştürülemez.

Aşağıdaki örnek değişebilirliği ve görünürlüğü değiştirmeyi açıklıyor:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base
{
    function foo() virtual external view {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() override public pure {}
}
```

Çoklu kalıtım için, aynı işlevi tanımlayan en çok türetilmiş temel sözleşmeler, `override` anahtar sözcüğünden sonra açıkça belirtilmelidir. Başka bir deyişle, aynı işlevi tanımlayan ve henüz başka bir temel sözleşme tarafından geçersiz kılınmamış tüm temel sözleşmeleri belirtmeniz gerekir (miras grafiği boyunca bir yolda). Ek olarak, bir sözleşme aynı işlevi birden çok (ilgisiz) temelden devralırsa, bunu açıkça geçersiz kılması gerekir:

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

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
    // foo() fonksiyonuna sahip birden fazla temel akıllı sözleşmesi türetir.
    // Bu yüzden override etmek için açıkça belirtmeliyiz.
    function foo() public override(Base1, Base2) {}
}
```

Fonksiyon, ortak bir temel sözleşmede tanımlanmışsa veya ortak bir temel sözleşmede diğer tüm işlevleri zaten override eden benzersiz bir işlev varsa, açık bir override belirteci gerekli değildir.

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

contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// Açıkça override gerekmemektedir.
contract D is B, C {}
```

Daha resmi olarak, imza için tüm override etme yollarının parçası olan bir temel sözleşme varsa, birden çok tabandan devralınan bir fonksiyonu (doğrudan veya dolaylı olarak) override etme gerekli değildir ve (1) bu taban fonksiyonu uygular ve mevcut akıllı sözleşmeden tabana giden hiçbir yol bu imzaya sahip bir fonksiyondan bahsetmez veya (2) bu taban fonksiyonu yerine getirmiyor ve mevcut akıllı sözleşmeden o tabana kadar olan tüm yollarda fonksiyondan en fazla bir kez söz ediliyor.

Bu anlamda, bir imza için override etme yolu, söz konusu akıllı sözleşmede başlayan ve override etmeyen bu imzaya sahip bir işlevden bahseden bir akıllı sözleşmede sona eren miras grafiği boyunca bir yoldur.

Override eden bir fonksiyonu `virtual` olarak işaretlemezseniz, türetilmiş sözleşmeler artık bu fonksiyonun davranışını değiştiremez.

Not: `private` görünürlüğe sahip fonksiyonlar `virtual` olamaz.

Not: Interface dışında olup da kodu olmayan fonksiyonlar `virtual` olarak işaretlenmelidir. Interface içerisindeki bütün fonksiyonlar otomatikmen `virtual` olarak düşünülür.

Not: Solidity 0.8.8 itibari ile bir interface fonksiyonunu override ederken `override` sözcüğünü kullanmanıza gerek kalmıyor, birden fazla temel akıllı sözleşmede tanımlanan fonksiyonlar dışında.

Public durum değişkenleri parametre ve dönüş tipleri uyuştuğu zaman bir external fonksiyonu override edebilir:

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

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

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

Not: Public durum değişkenleri external fonksiyonları override edebilirken, kendileri override edilemez.

Modifier Override Etme

Fonksiyon modifier'ları birbirlerini override edebilirler. Bu aynı *fonksiyon override etmedeki* gibidir (modifierlarda overload etme olmamakla istisnası ile). `virtual` sözcüğü override edilecek modifier'da kullanılmalı ve override eden modifier'da ise `override` sözcüğü kullanılmalıdır.

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

contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

Çoklu kalıtım durumumnda bütün temel akıllı sözleşmeler açıkça override edilme durumunu belirtmelidir.

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

contract Base1
{
    modifier foo() virtual {_;}
}

contract Base2
{
    modifier foo() virtual {_;}
}

contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

Constructor'lar

Constructor isteğe bağlı olarak tanımlanan özel fonksiyonlardan biridir ve `constructor` sözcüğü ile tanımlanır. Bu fonksiyon akıllı sözleşme oluşumu sırasında çalıştırılır ve akıllı sözleşme başlatma kodunuz burada bulunmaktadır.

Constructor kodu çalıştırılmadan önce durum değişkenleri eğer aynı satırda tanımladıysanız gerekli değer atamalarını veya tanımlamadıysanız *default değerlerini* alırlar.

Constructor çalıştırıldıktan sonra kodun son hali blockchain'e yüklenir. Bu işlemin ücreti ise lineer bir şekilde olup kodun uzunluğuna bağımlıdır. Bu kod dışarıdan erişilebilecek ve bir fonksiyon tarafından erişilen bütün fonksiyonları içerir. Constructor kodunu veya sadece constructor tarafından erişilen internal fonksiyonları içermez.

Eğer constructor yoksa, default constructor çalıştırılır `constructor() {}`. Örneğin:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint a_) {
        a = a_;
    }
}

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

Constructor'larda internal parametreleri kullanabilirsiniz (örneğin, storage pointer'ları). Bu durumda akıllı sözleşme *abstract* olarak işaretlenmelidir. Çünkü bu parametrelere dışarıdan geçerli değerler atanamaz, ancak yalnızca türetilmiş sözleşmelerin constructor'ları aracılığıyla atanır.

Uyarı: Versiyon 0.4.22 öncesinde constructor'lar akıllı sözleşme ile aynı isme sahip fonksiyonlar olarak kullanılırdı. Ancak bu yazılış biçiminin Versiyon 0.5.0 sonrasında kullanımına izin verilmemektedir.

Uyarı: Versiyon 0.7.0 öncesinde constructor'ların görünürlüğünü `internal` veya `public` olarak belirtmek zorundaydınız.

Temel Constructor'lar için Argümanlar

Tüm temel akıllı sözleşmelerin constructor'ları, aşağıda açıklanan doğrusallaştırma kurallarına göre çağrılacaktır. Temel akıllı sözleşmelerin argümanları varsa, türetilmiş akıllı sözleşmelerin hepsini belirtmesi gerekir. Bu iki şekilde yapılabilir:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base {
    uint x;
    constructor(uint x_) { x = x_; }
}

// Direkt kalıtım listesinde belirtme...
contract Derived1 is Base(7) {
    constructor() {}
}

// veya "modifier" stilinde belirtme...
contract Derived2 is Base {
    constructor(uint y) Base(y * y) {}
}
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
// veya abstract olarak belirtin...
abstract contract Derived3 is Base {
}

// ve bir sonraki contractın onu başlatmasını sağlayın.
contract DerivedFromDerived is Derived3 {
    constructor() Base(10 + 10) {}
}
```

Bir yol doğrudan kalıtım listesindedir (`is Base(7)`). Diğeri, türetilmiş constructor'ın bir parçası olarak bir modifier'ın çağrılma biçimindedir (`Base(y * y)`). Bunu yapmanın ilk yolu, constructor argümanının sabit olması ve akıllı sözleşmenin davranışını tanımlaması veya tanımlaması durumunda daha uygundur. Temel constructor argümanları türetilmiş akıllı sözleşmenin argümanlarına bağlıysa, ikinci yol kullanılmalıdır. Argümanlar ya kalıtım listesinde ya da türetilmiş constructor'da değiştirici-tarzda verilmelidir. Argümanları her iki yerde de belirtmek bir hatadır.

Türetilmiş bir akıllı sözleşme, temel akıllı sözleşmelerin tüm constructorları için argümanları belirtmiyorsa, özet olarak bildirilmelidir. Bu durumda, ondan başka bir akıllı sözleşme türetildiğinde, diğer akıllı sözleşmenin miras listesi veya constructor'ı, parametreleri belirtilmemiş tüm temel sınıflar için gerekli parametreleri sağlamalıdır (aksi takdirde, diğer akıllı sözleşme da soyut olarak bildirilmelidir). Örneğin, yukarıdaki kod parçacığında, bkz. `Derived3` ve `DerivedFromDerived`.

Çoklu Kalıtım ve Doğrusallaştırma

Çoklu kalıtıma izin veren diller birkaç problemle uğraşmak zorundadır. Bunlardan bir tanesi Elmas Problemi'dir. Solidity Python'a benzer olarak "C3 Linearization" kullanarak directed acyclic graph'da (DAG) spesifik bir sırayı zorlar. Bu, istenen monotonluk özelliği ile sonuçlanır, ancak bazı kalıtım grafiklerine izin vermez. Özellikle `is` yönergesinde temel sınıfların veriliş sırası önemlidir: Doğrudan temel sözleşmeleri "en temele benzeyen"den "en çok türetilene" doğru sıralamalısınız. Bu sıralamanın Python'da kullanılanın tersi olduğuna dikkat edin.

Bunu açıklamanın bir başka basitleştirici yolu, farklı akıllı sözleşmelerde birden çok kez tanımlanan bir fonksiyon çağrıldığında, verilen tabanların sağdan sola (Python'da soldan sağa) derinlemesine ilk olarak aranması ve ilk eşleşmede durdurulmasıdır. . Bir temel akıllı sözleşme zaten aranmışsa, atlanır.

Aşağıdaki kodda Solidity "Linearization of inheritance graph impossible" hatası verecektir.

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

contract X {}
contract A is X {}
// Bu derlenemez
contract C is A, X {}
```

Bunun sebebi `C` akıllı sözleşmesinin `X` akıllı sözleşmesinin `A` akıllı sözleşmesini override etmesini istemesidir (`A, X` sırası ile bunu belirtiyor), ancak `A` akıllı sözleşmesinin kendisi `X` akıllı sözleşmesini override etmeyi talep eder ki bu çözülemeyecek bir çelişkidir.

Benzersiz bir override olmadan birden çok tabandan devralınan bir fonksiyonu açıkça override etmek gerektiğinden, pratikte C3 doğrusallaştırması çok önemli değildir.

Kalıtım doğrusallaştırmasının özellikle önemli olduğu ve belki de o kadar net olmadığı bir alan, miras hiyerarşisinde birden çok constructor olduğu zamandır. Constructor'lar, argümanlarının devralınan akıllı sözleşmenin constructor'ında sağlandığı sıraya bakılmaksızın her zaman doğrusallaştırılmış sırada yürütülür. Örneğin:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base1 {
    constructor() {}
}

contract Base2 {
    constructor() {}
}

// Constructor'lar aşağıdaki sıra ile çalışır:
//  1 - Base1
//  2 - Base2
//  3 - Derived1
contract Derived1 is Base1, Base2 {
    constructor() Base1() Base2() {}
}

// Constructor'lar aşağıdaki sıra ile çalışır:
//  1 - Base2
//  2 - Base1
//  3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() Base2() Base1() {}
}

// Constructors are still executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived3
contract Derived3 is Base2, Base1 {
    constructor() Base1() Base2() {}
}
```

Farklı Türden Aynı İsme Sahip Üyeleri Türetme

Bir akıllı sözleşmede aşağıdaki çiftlerden herhangi birinin miras nedeniyle aynı ada sahip olması bir hatadır:

- bir fonksiyon ve bir modifier
- bir fonksiyon ve bir event
- bir event ve bir modifier

İstisna olarak, bir durum değişkeninin getirici fonksiyonu bir external fonksiyonu override edebilir.

3.9.9 Abstract Akıllı Sözleşmeler

Sözleşmeler, işlevlerinden en az biri uygulanmadığında veya bütün temel sözleme yapıcılar için argüman sağlamadığında abstract olarak işaretlenmelidir. Bu durumlardan herhangi biri geçerli değilse bile bir akıllı sözleşme abstract olarak işaretlenebilir. Örneğin bir akıllı sözleşmenin direkt olarak oluşturulmasını istemediğiniz durumlarda bunu gerçekleştirebilirsiniz. Abstract akıllı sözleşmeler *Interface'ler* oldukça benzerdir ancak interface'ler çok daha kısıtlı bir yapıdadır.

Aşağıdaki örnekte belirtildiği gibi, Abstract akıllı sözleşmeler `abstract` olarak işaretlenerek belirtilir. Aşağıdaki akıllı sözleşmenin abstract olarak tanımlanması gerektiğine dikkat edin. Çünkü `utterance()` fonksiyonu tanımlanıp kodları yazılmamıştır (`{ }` arasında kod bulunmamakta).

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

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

Bu tip abstract akıllı sözleşmeler direkt olarak örneklendirilemez. Bu, abstract sözleşmenin kendisi tanımlanmış tüm işlevleri yerine getiriyorsa da geçerlidir. Abstract bir akıllı sözleşmenin temel sınıf olarak kullanımı aşağıda gösterilmiştir:

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

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

contract Cat is Feline {
    function utterance() public pure override returns (bytes32) { return "miaow"; }
}
```

Bir akıllı sözleşme bir abstract akıllı sözleşmeden türetiliyorsa ve abstract akıllı sözleşmedeki bütün kodu yazılmamış fonksiyonların kodunu yazmıyorsa, o akıllı sözleşme da abstract olarak belirtilmelidir.

Kodu olmayan bir fonksiyonun *Fonksiyon Tipinden* farklı olduğuna dikkat edin, her ne kadar yazılışları oldukça benzer olsa da.

Kodu olmayan bir fonksiyona örnek olarak (fonksiyon tanımlaması):

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

Türü bir fonksiyon türü olan bir değişken bildirimi örneği:

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

Abstract akıllı sözleşmeler, daha iyi genişletilebilirlik ve kendi kendine belgeleme sağlayarak ve Template yöntemi gibi kalıpları kolaylaştırarak ve kod tekrarını ortadan kaldırarak bir akıllı sözleşmenin tanımını uygulamasından ayırır. Abstract akıllı sözleşmeler, bir arabirimdeki yöntemleri tanımlamanın yararlı olduğu şekilde yararlıdır. Abstract akıllı sözleşmenin tasarımcısının "her çocuğum bu yöntemi uygulamalı" demesinin bir yoludur.

Not: Abstract akıllı sözleşmeler kodu yazılmış bir virtual fonksiyonu kodu yazılmamış bir fonksiyon ile override edemezler.

3.9.10 Interface'ler

Interface'ler abstract akıllı sözleşmelere benzerler ama onlardan farklı olarak hiçbir fonksiyonunun kodu yazılamaz. Daha fazla kısıtlama vardır:

- Diğer akıllı sözleşmelerden miras alamazken, diğer interface'lerden alabilirler.
- Interface'deki bütün fonksiyonlar external olmalıdır, akıllı sözleşmede public olsalar dahi.
- Constructor tanımlayamazlar.
- Durum değişkeni tanımlayamazlar.
- Modifier tanımlayamazlar.

Bu kısıtlamalardan bazıları ilerleyen zamanlarda kaldırılabilir.

Interface'ler kabaca akıllı sözleşme ABI'sinin temsil edebileciği ile kısıtlıdır. Bu yüzden ABI ve interface arasındaki dönüşümler bilgi kaybı yaşanmadan gerçekleştirilebilmelidir.

Interface'ler kendi anahtar sözcükleri ile tanımlanırlar:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Akıllı sözleşmeler diğer akıllı sözleşmelerden miras alabildikleri gibi diğer interface'lerden de alabilirler.

Interface'lerdeki bütün fonksiyonlar gizlici `virtual` olarak işaretlenmiş haldedir ve onları override ederken `override` kelimesine gerek yoktur. Bu, otomatik olarak override eden bir fonksiyonun yeniden override edilebileceği anlamına gelmez - bu yalnızca override eden fonksiyon `virtual` olarak işaretlenmişse mümkündür.

Interface'ler diğer interfacelerden miras alabilirler, normal kalıtım kuralında olduğu gibi.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Ebeveny anlamlarının uyumlu olduğunu iddia
    // etmek için test yeniden tanımlanmalıdır.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Interface'lerde tanımlanan tiplere ve diğer akıllı sözleşme benzeri yapılara diğer akıllı sözleşmelerden erişilebilir: `Token.TokenType` veya `Token.Coin`.

3.9.11 Kütüphaneler

Kütüphaneler akıllı sözleşmelere benzerler, ama onların amacı sadece bir kere deploy edilip daha sonrasında ihtiyaç duyulması halinde `DELEGATECALL` ile çağrılmalarıdır (Homestead'a kadar `CALLCODE` kullanılırdı). Bu demek oluyor ki kütüphane fonksiyonları çağrıldığında, onların kodu çağıran akıllı sözleşmenin içeriği ile çalıştırılıyor, mesela `this` sözcüğü çağıran akıllı sözleşmeyi işaret eder ve özellikle storage olarak çağıran akıllı sözleşmenin storage kısmı kullanılır. Bir kütüphane izole edilmiş bir kaynak kodu parçası olduğundan, yalnızca açıkça sağlanmışlarsa çağrı sözleşmesinin durum değişkenlerine erişebilir (aksi takdirde bunları adlandırmanın hiçbir yolu yoktur). Kütüphane fonksiyonları yalnızca durumu değiştirmedikleri takdirde (yani `view` veya `pure` fonksiyonlarsa) doğrudan (yani `DELEGATECALL` kullanılmadan) çağrılabilir, çünkü kütüphanelerin durumsuz olduğu varsayılır. Özellikle, bir kütüphaneyi yok etmek mümkün değildir.

Not: 0.4.20 sürümüne kadar, Solidity'nin tip sistemini atlayarak kütüphaneleri yok etmek mümkündü. Bu sürümden başlayarak, kütüphaneler, durumu değiştiren fonksiyonların doğrudan çağrılmasına izin vermeyen bir *mekanizma* içerir (yani `DELEGATECALL` olmadan).

Kütüphaneler, onları kullanan akıllı sözleşmelerin zımni temel akıllı sözleşmeleri olarak görülebilir. Miras hiyerarşisinde açıkça görünmezler, ancak kütüphane fonksiyonlarına yapılan çağrılar, açık temel akıllı sözleşmelerin fonksiyonlarına yapılan çağrılara benzer (L.f() gibi nitelikli erişim kullanarak). Tabii ki, dahili fonksiyonlara yapılan çağrılar dahili çağrı kuralını kullanır; bu, tüm dahili türlerin iletilebileceği ve bellekte depolanan türlerin kopyalanmadan referans olarak iletileceği anlamına gelir. Bunu EVM'de gerçekleştirmek için, bir akıllı sözleşmeden çağrılan dahili kütüphane fonksiyonlarının ve buradan çağrılan tüm fonksiyonların kodu derleme zamanında çağrı akıllı sözleşmesine dahil edilecek ve bir `DELEGATECALL` yerine normal bir JUMP çağrısı kullanılacaktır.

Not: Public fonksiyonlar söz konusu olduğunda miras analojisi bozulur. L.f() ile bir genel kütüphane fonksiyonunun çağrılması, harici bir çağrıyla sonuçlanır (kesin olarak `DELEGATECALL`). Buna karşılık, A mevcut akıllı sözleşmesinin temel akıllı sözleşmesi olduğunda, A.f() dahili bir çağrıdır.

Aşağıdaki örnek, kütüphanelerin nasıl kullanılacağını gösterir (ancak manuel bir yöntem kullanarak, bir kümeyi uygulamak için daha gelişmiş bir örnek için kullanmayı kontrol ettiğinizden emin olun).

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


// Çağrı akıllı sözleşmesinde verilerini tutmak
// için kullanılacak yeni bir struct veri türü tanımlıyoruz.
struct Data {
    mapping(uint => bool) flags;
}

library Set {
    // İlk parametrenin "depolama referansı" türünde
    // olduğunu ve bu nedenle çağrının bir parçası
    // olarak içeriğinin değil, yalnızca depolama
    // adresinin iletildiğini unutmayın.
    // Bu, kütüphane fonksiyonlarının özel bir özelliğidir.
    // Fonksiyon, o nesnenin bir yöntemi olarak görülebiliyorsa,
    // ilk parametreyi 'self' olarak adlandırmak deyimseldir.
    function insert(Data storage self, uint value)
        public
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        returns (bool)
    {
        if (self.flags[value])
            return false; // zaten orada
        self.flags[value] = true;
        return true;
    }

    function remove(Data storage self, uint value)
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false; // orada değil
        self.flags[value] = false;
        return true;
    }

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


contract C {
    Data knownValues;

    function register(uint value) public {
        // "Instance" geçerli akıllı sözleşme olacağından,
        // kütüphane fonksiyonları kütüphanenin belirli
        // bir örneği olmadan çağrılabilir.
        require(Set.insert(knownValues, value));
    }
    // Bu sözleşmede ayrıca direkt olarak knownValues.flags değişkenine de erişebiliriz.
}
```

Elbette kütüphaneleri kullanmak için bu yolu izlemeniz gerekmez: struct veri türleri tanımlamadan da kullanılabilirler. Fonksiyonlar ayrıca herhangi bir depolama referans parametresi olmadan da çalışırlar ve herhangi bir pozisyonda birden fazla depolama referans parametresine sahip olabilirler.

`Set.contains`, `Set.insert` ve `Set.remove` çağrılarının hepsi harici çağrı olarak derlenir (`DELEGATECALL`). Eğer kütüphaneleri kullanacaksanız gerçekten bir harici fonksiyon çağrısı yaptığınızı unutmayın. `msg.sender`, `msg.value` ve `this` çağrı boyunca kendi değerlerini koruyacaktır (Homestead öncesi `CALLCODE` yüzünden `msg.sender` ve `msg.value` değişiyordu).

Aşağıdaki örnek, harici fonksiyon çağrılarının ek yükü olmadan özel türleri uygulamak için *bellekte depolanan türlerin* ve kütüphanelerdeki dahili fonksiyonların nasıl kullanılacağını gösterir:

```
// SPDX-License-Identifier: GPL-3.0
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
pragma solidity ^0.8.0;

struct bigint {
    uint[] limbs;
}

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

    function add(bigint memory a, bigint memory b) internal pure returns (bigint memory␣
↪r) {
        r.limbs = new uint[](max(a.limbs.length, b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint limbA = limb(a, i);
            uint limbB = limb(b, i);
            unchecked {
                r.limbs[i] = limbA + limbB + carry;

                if (limbA + limbB < limbA || (limbA + limbB == type(uint).max && carry >␣
↪0))
                    carry = 1;
                else
                    carry = 0;
            }
        }
        if (carry > 0) {
            // çok kötü, bir limb eklemeliyiz
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            uint i;
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint memory a, uint index) internal pure returns (uint) {
        return index < a.limbs.length ? a.limbs[index] : 0;
    }

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

contract C {
    using BigInt for bigint;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

Bir kütüphanenin adresini, kütüphane tipini `address` tipine çevirerek, yani `address(LibraryName)` kullanarak elde etmek mümkündür.

Derleyici kütüphanenin konuşlandırılacağı adresi bilmediğinden, derlenmiş onaltılık kod `__$30bbc0abd4d6364515865950d3e0d10953$__` biçiminde yer tutucular içerecektir. Yer tutucu, tam nitelikli kütüphane adının keccak256 hashinin hex kodlamasının 34 karakterlik bir önekidir; bu, örneğin kütüphane `bigint.sol` isimli bir dosyada ve `libraries/` isimli bir dizinde bulunuyorsa şu şekilde gösterilir `libraries/bigint.sol:BigInt`. Bu tür bayt kodu eksiktir ve dağıtılmamalıdır. Yer tutucuların gerçek adreslerle değiştirilmesi gerekir. Bunu, kütüphane derlenirken bunları derleyiciye ileterek veya önceden derlenmiş bir ikili dosyayı güncellemek için bağlayıcıyı kullanarak yapabilirsiniz. Bağlama için komut satırı derleyicisinin nasıl kullanılacağı hakkında bilgi için *Kütüphane Bağlantıları (Library Linking)* konusuna bakın.

Akıllı sözleşmelerle kıyaslandığında, kütüphaneler aşağıdaki şekillerde kısıtlanmışlardır:

- durum değişkenleri olamaz
- miras veremezler veya alamazlar
- Ether kabul edemezler
- yok edilemezler

(Bunlar ilerleyen zamanlarda kaldırılabilirler.)

Function Signatures and Selectors in Libraries

Public veya external kütüphane fonksiyonlarına harici çağrılar mümkün olsa da, bu tür çağrılar için çağrı kuralının Solidity'nin içinde olduğu ve normal *contract ABI* için belirtilenle aynı olmadığı kabul edilir. External kütüphane fonksiyonları, örneğin özyinelemeli yapılar ve depolama işaretçileri gibi external kütüphane fonksiyonlarından daha fazla bağımsız değişken türünü destekler. Bu nedenle, 4 baytlık seçiciyi hesaplamak için kullanılan fonksiyon imzaları, bir internal adlandırma şemasının ardından hesaplanır ve ABI akıllı sözleşmesinde desteklenmeyen türdeki bağımsız değişkenler bir dahili kodlama kullanır.

İmzalardaki türler için aşağıdaki tanımlayıcılar kullanılır:

- Değer tipleri, storage olmayan `string` ve storage olmayan `bytes` tipleri akıllı sözleşme ABI'sinde aynı tanımlayıcıları kullanır.
- Storage olmayan array tipleri de akıllı sözleşme ABI'sindeki genel görüşü kabul eder, yani dinamik arrayler için `<type>[]` ve fixed-size arrayler için `<type>[M]` kullanılır.
- Storage olmayan structlar tam isimleri ile referans edilir, yani `contract C { struct S { ... } }` için `C.S`.
- Storage pointer mappingleri de `mapping(<keyType> => <valueType>) storage` kullanır. Burada `<keyType>` ve `<valueType>` sırasıyla mappingdeki anahtar ve değer tipleridir.
- Diğer storage pointer tipleri de kendi storage olmayan tiplerinin tanımlayıcılarını kullanırlar, ama bir boşluk ile `storage` eklenmiş halleri ile.

Argüman encode'lama da sıradan akıllı sözleşme ABI'si gibidir, storage pointerları hariç, işaret ettikleri storage slotuna atıfta bulunan bir `uint256` değeri olarak kodlanmıştır.

Akıllı sözleşme ABI'sine benzer bir şekilde, selector, imzanın Keccak256-hashinin ilk dört baytından oluşur. Değeri, `.selector` üyesi kullanılarak Solidity'den şu şekilde elde edilebilir:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

Kütüphaneler İçin Çağrı Koruması

Girişte belirtildiği gibi, bir kütüphanenin kodu `DELEGATECALL` veya `CALLCODE` yerine bir `CALL` kullanılarak yürütülürse, bir `view` veya `pure` fonksiyon çağrılmadığı sürece geri dönecektir.

EVM, bir akıllı sözleşmenin `CALL` kullanılarak çağrılıp çağrılmadığını tespit etmek için doğrudan bir yol sağlamaz, ancak bir sözleşme, "nerede" çalıştığını bulmak için `ADDRESS` işlem kodunu kullanabilir. Oluşturulan kod, arama modunu belirlemek için bu adresi yapım sırasında kullanılan adresle karşılaştırır.

Daha spesifik olarak, bir kütüphanenin çalışma zamanı kodu her zaman derleme zamanında 20 bayt sıfır olan bir push komutuyla başlar. Dağıtım kodu çalıştığında, bu sabit bellekte geçerli adresle değiştirilir ve bu değiştirilmiş kod sözleşmede saklanır. Çalışma zamanında, bu, dağıtım zamanı adresinin yığına gönderilecek ilk sabit olmasına neden olur ve dağıtıcı kodu, herhangi bir görünüm olmayan ve saf olmayan işlev için geçerli adresi bu sabitle karşılaştırır.

Bu, bir kitaplık için zincirde depolanan gerçek kodun derleyici tarafından bildirilen koddan farklıdır. `deployedBytecode`.

3.9.12 Using For

`using A for B;` yönergesi, (`A`) fonksiyonlarını herhangi bir türe (`B`) üye fonksiyonlar olarak eklemek için kullanılabilir. Bu fonksiyonlar, çağrıldıkları nesneyi ilk parametreleri olarak alırlar (Python'daki `self` değişkeni gibi).

Dosya seviyesinde veya bir akıllı sözleşme içerisinde, akıllı sözleşme seviyesinde, geçerlidir.

İlk kısım, `A`, aşağıdakilerden biri olabilir:

- dosya seviyesindeki fonksiyonların bir listesi veya kütüphane fonksiyonları (`using {f, g, h, L.t} for uint;`) - sadece o fonksiyonlar eklenecektir.
- kütüphanenin adı (`using L for uint;`) - bütün fonksiyonlar (public ve internallerin hepsi) tipe eklenir.

Dosya seviyesinde, ikinci kısım, `B`, açık bir tip olmalıdır (veri konumu belirtici olmadan). Akıllı sözleşmenin içerisinde, ayrıca şu ifadeyi de kullanabilirsiniz `using L for *;`, böylece `L` kütüphanesinin bütün fonksiyonları *bütün* tiplere eklenmiş olur.

Bir kütüphane belirtirseniz, kütüphanedeki tüm fonksiyonlar, ilk parametrenin türü nesnenin türüyle eşleşmese bile eklenir. Fonksiyonun çağrıldığı noktada tip kontrol edilir ve fonksiyon aşırı yük çözünürlüğü gerçekleştirilir.

Eğer bir fonksiyon listesi kullanırsanız (`using {f, g, h, L.t} for uint;`), ardından gelen tip (`uint`) o kütüphanedeki bütün fonksiyonların ilk parametrelerine gizlice dönüştürülebilir olmalıdır. Bu kontrol, fonksiyonların hiçbiri çağrılmasa bile gerçekleştirilir.

`using A for B;` direktifi, tüm fonksiyonları dahil olmak üzere yalnızca mevcut kapsamda (sözleşme veya mevcut modül/kaynak birim) etkindir ve kullanıldığı sözleşme veya modül dışında hiçbir etkisi yoktur.

Yönerge dosya düzeyinde kullanıldığında ve aynı dosyada dosya düzeyinde tanımlanmış kullanıcı tanımlı bir türe uygulandığında, sonuna `global` sözcüğü eklenebilir. Bu, yalnızca using ifadesinin kapsamında değil, türün kullanılabilir olduğu her yerde (diğer dosyalar dahil) işlevlerin türe eklenmesi etkisine sahip olacaktır.

Kütüphaneler bölümünde yazdığımız bir örneği dosya seviyesindeki fonksiyonlarla yeniden yazalım:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

struct Data { mapping(uint => bool) flags; }
// Şimdi örneğe fonksiyonları ekliyoruz.
// Eklenen fonksiyonlar modül boyuna kullanılabilir.
// Eğer modülü başka bir dosyadan eklerseniz
// using yönergesini orada yeniden kullanmalısınız:
//   import "flags.sol" as Flags;
//   using {Flags.insert, Flags.remove, Flags.contains}
//     for Flags.Data;
using {insert, remove, contains} for Data;

function insert(Data storage self, uint value)
    returns (bool)
{
    if (self.flags[value])
        return false; // 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)
    view
    returns (bool)
{
    return self.flags[value];
}


contract C {
    Data knownValues;

    function register(uint value) public {
        // Burada, Data türündeki tüm değişkenlerin karşılık
        // gelen üye işlevleri vardır. Aşağıdaki işlev çağrısı,
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        // `Set.insert(knownValues, value)` ile aynıdır.
        require(knownValues.insert(value));
    }
}
```

Yerleşik türleri bu şekilde genişletmek de mümkündür. Bu örnekte bir kütüphane kullanacağız.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return type(uint).max;
    }
}
using Search for uint[];

contract C {
    uint[] data;

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

    function replace(uint from, uint to) public {
        // Bu, kütüphane işlev çağrısını gerçekleştirir
        uint index = data.indexOf(from);
        if (index == type(uint).max)
            data.push(to);
        else
            data[index] = to;
    }
}
```

Tüm harici kütüphane çağrılarının gerçek EVM fonksiyon çağrıları olduğunu unutmayın. Bu, bellek veya değer türlerini geçerseniz, `self` değişken durumunda bile bir kopyanın gerçekleştirileceği anlamına gelir. Kopyalama yapılmayacak tek durum, depolama referans değişkenlerinin kullanıldığı veya dahili kütüphane fonksiyonlarının çağrıldığı durumlardır.

3.10 Inline Assembly

Inline assembly ile Solidity ifadelerini Ethereum Sanal Makine'sinin dillerinden birine yakın dile çevirebilirsiniz. Bu size özellikle dili kütüphaneler yazarak geliştiriyorsanız daha detaylı bir kontrol sağlar.

Inline assembly için kullanılan Solidity diline *Yul* deniyor ve dosyalarını kendi bölümünde bulabilirsiniz. BU bölüm sadece inline assembly kodunun etrafındaki Solidity kodları ile nasıl bağlandığını anlatacak.

Uyarı: Inline assembly Ethereum Sanal Makinesi'ne düşük seviyede erişişmin bir yoludur. Bu, Solidity'nin birçok güvenlik özelliklerini ve kontrollerini yok sayar. Yani inline assembly'i sadece gereken yerlerde ve nasıl kullanacağınızdan eminseniz kullanmalısınız.

Bir inline assembly bloğu `assembly { ... }` ile işaretlidir. Süslü parantez içerisindeki kod *Yul* dili içerisinde yer alır.

Bir inline assembly kodu yerel Solidity değişkenlerine aşağıda açıklandığı gibi erişebilir.

Farklı inline assembly blokları aynı yer adlarını paylaşmazlar. Yani farklı bir inline assembly bloğunda tanımlanmış olan bir Yul fonksiyonunu çağırmak ya da bir Yul değişkenine erişmek mümkün değldir.

3.10.1 Örnek

Aşağıdaki örnek başka bir kontrat üzerindeki koda erişimi ve bir `bytes` değişkenine atımını sağlayan kütüphane kodunu verir. Bu "düz Solidity" ile de `<address>.code` kullanarak mümkündür ama buradaki amaç tekrar kullanılabilir assembly kütüphanelerinin bir derleyici(compiler) değişimi olmadan Solidity dilini geliştirebildiğini göstermektir.

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

library GetCode {
    function at(address addr) public view returns (bytes memory code) {
        assembly {
            // kodun boyutunu döndürür, burası için assembly kullanılmalı
            let size := extcodesize(addr)
            // çıkış bit array'ini allocate() eder
            // burası assembly kullanmadan,
            // code = new bytes(size) kullanarak da yapılabilir.
            code := mload(0x40)
            // padding'i içeren yeni "memory end"
            mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // uzunluğu hafızada saklayın
            mstore(code, size)
            // kodun şu anki halini döndürür, burası için assembly kullanılmalı
            extcodecopy(addr, add(code, 0x20), 0, size)
        }
    }
}
```

Inline assembly optimizer verimli kodlar üretemediği zamanlarda da yararlıdır, örneğin:

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

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
library VectorSum {
    // Bu fonksiyon şu anda verimli değil
    // çünkü optimizer array sınır erişim kontrolünü yaparken hata veriyor
    function sumSolidity(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i)
            sum += data[i];
    }

    // Array'e sadece sınırları içerisinde erişebileceğimizi biliyoruz, yani bu kontrolü␣
↪atlayabiliriz.
    // 0x20'nin array'e eklenmesi gerekiyor çünkü array'in ilk slotu array uzunluğunu␣
↪içerir.
    function sumAsm(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i) {
            assembly {
                sum := add(sum, mload(add(add(data, 0x20), mul(i, 0x20))))
            }
        }
    }
    // Yukarıdaki gibi ama tüm kodu inline assembly kullanarak tamamlayın.
    function sumPureAsm(uint[] memory data) public pure returns (uint sum) {
        assembly {
            // uzunluğu yükleyin (önce 32 byte)
            let len := mload(data)

            // Uzunluk alanını atlayın.
            //
            // Geçici bir değişken tutun, böylece yer değiştikçe onu da␣
↪arttırabilirsiniz.
            //
            // NOT: Bu assembly bloktan sonra arttırılan veri kullanılamayacak bir␣
↪değişkene dönüşecek

            let dataElementLocation := add(data, 0x20)

            // Sınıra ulaşana kadar tekrarlayın.
            for
                { let end := add(dataElementLocation, mul(len, 0x20)) }
                lt(dataElementLocation, end)
                { dataElementLocation := add(dataElementLocation, 0x20) }
            {
                sum := add(sum, mload(dataElementLocation))
            }
        }
    }
}
```

3.10.2 Dış(External) değişkenlere, fonksiyonlara ve kütüphanelere erişim

Solidity değişkenlerine ve diğer tanımlayıcılara isimlerini kullanarak erişebilirsiniz.

Bir değer tipinin yerel değişkenleri inline assembly içinde kullanılabilir durumdadır. Bu yerel değişkenler okunabilir de atanabilir de.

Belleği kasteden yerel değişkenler değerin kendisini değil, değerin bellekteki adresini işaret eder. Bu değişkenler aynı zamanda değiştirilebilir de ancak bu sadece bir pointer değişimi olur, veri değişimi olmaz. Bu sebeple Solidity'nin hafıza yönetimini yapmak sizin yükümlülüğünüzdedir. Bkz *Solidity'de Konvansiyonlar*

Benzer şekilde, statik boyutlandırılmış calldata array'leri ya da struct'ları gösteren yerel değişkenler de değerin adresini işaret eder, değerini değil. Bu değişken yeni bir offset'e de atanabilir fakat değişkenin `calldatasize()` çalıştırılması dışında bir yeri işaret edebileceğinin hiçbir garantisi yoktur.

Dış(External) fonksiyon pointer'ları için adres ve fonksiyon seçiyiye `x.address` ve `x.selector` ile erişilebilir. Seçici dört adet right-aligned bitten oluşur. İki değer de atanbilir. Örneğin:

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

contract C {
    // @fun değerini dönmek için yeni bir seçici de adres atayın
    function combineToFunctionPointer(address newAddress, uint newSelector) public pure␣
↪returns (function() external fun) {
        assembly {
            fun.selector := newSelector
            fun.address  := newAddress
        }
    }
}
```

Dinamik calldata array'leri üzerinde, `x.offset` ve `x.length` kullanarak -bit halinde- calldata offset'ine ve uzunluğuna erişebilirsiniz. Her iki ifade aynı zamanda atanabilir de ama statik bir durum için dönecekleri sonucun `calldatasize()` sınırları içerisinde olacağının bir garantisi yoktur.

Yerel depolama değişkenleri ya da durum değişkenleri için tek bir Yul tanımlayıcısı yeterli değildir. Çünkü bu değişkenler her zaman tam bir depolama alanı kaplamazlar. Bu sebeple onların 'adresleri' bir slottan ve o slot içerisindeki bir byte-offset'ten oluşur. `x` değişkeni tarafından işaret edilen slotu çağırmak için `x.slot` , byte-offset'i çağırmak için ise `x.offset` kullanılır. Sadece `x` kullanmak ise hata verecektir.

Bir yerel depolama değişkeninin pointer'ının `.slot` kısmına atama yapılabilir. Bu değişkenler(struct, array, mapping) için `.offset` kısmı ise her zaman sıfırdır. Fakat bir durum değişkeninin `.slot` ve `.offset` kısmına atama yapmak ise mümkün değildir.

Yerel Solidity değişkenleri görevler için hazırdır. Örneğin:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            // Bu senaryoda depolama slotunun offset'ini değelendirmiyoruz.
            // Çünkü sıfır olduğunu biliyoruz.
            r := mul(x, sload(b.slot))
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        }
    }
}
```

Uyarı: Eğer `uint64`, `address` veya `bytes16` gibi 256 bitten daha az yer kaplayan bir değişkene erişmeye çalışıyorsanız bu tipin parçası olmayan bitler hakkında bir varsayımda bulunmayın. Özellikle de o bitleri sıfır kabul etmeyin. Her ihtimale karşı, `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` parçasının önemli olduğu yerlerde düzgün bir şekilde bu verileri temizleyin. Signed tipleri temizlemek için `signextend` kullanabilirsiniz. opcode: `assembly { signextend(<num_bytes_of_x_minus_one>, x) }`

Solidity 0.6.0'dan beri bir inline assembly değişkeninin ismi inline assembly bloğundaki kullanımını karşılamayabilir. (değişken, kontrat ve fonkisyon kullanımları dahil)

Soldity 0.7.0'dan beri inline assembly bloğunun içinde kullanılan değişken ve fonksiyonlar `.` içermeyebilir. Fakat `.` kullanmak inline assembly bloğu dışındaki Solidity değişkenlerine ulaşmak için etkilidir.

3.10.3 Kaçınılacak Şeyler

Inline assembly high-level gözükebilir fakat aslında aşırı derecede low-level'dır. Fonksiyon çağrıları, döngüler, if'ler ve switch'ler basit tekrar yazım kuralları ile çevrilir ve bundan sonra assembler'ın tek yaptığı iş blok sonuna erişildiğinde functional-style opcode'ları tekar ayarlamak, değişken erişimi için stack boyutunu saymak ve assembly içerisindeki değişkenleri için stack slotlarını kaldırmaktır.

3.10.4 Solidity kuralları

Typed Değişkenlerin Değerleri

EVM assembly'nin aksine, Solidity 256 bitten daha küçük tiplere sahiptir (ör: `uint24`). Verimlilik için çoğu aritmetik işlem bazı tiplerin 256 bitten küçük olabileceğini yok sayar ve higher-order bitler gerekliyse (hafızaya yazılmadan hemen önce ya da herhangi bir karşılaştırma yapılmadan önce) temizlenir. Burası şu yüzden önemlidir: Eğer inline assembly içerisinde böyle bir değişkene erişmek istiyorsanız önce higher-order bitleri kendiniz temizlemeniz gerekebilir.

Hafıza Yönetimi

Solidity Belleği şu şekilde yönetir. Hafızada `0x40` konumunda bir "boş bellek pointer"ı bulunur. Eğer belleğe bir şey atamak isterseniz bu pointer'ın işaret ettiği yerden başlayıp güncelleyin. Bu hafızanın daha önce kullanılmadığına dair herhangi bir kanıt bulunmadığı için tamamen sıfır olduğunu da varsayamazsınız. Belleği boşaltacak ya da rahatlatacak herhangi bir hazır kurulu mekanizma yoktur. Aşağıda belleği yukarıda anlatıldığı şekilde kullanabileceğiniz bir assembly kod parçası bulunuyor:

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

Hafızanın ilk 64 biti kısa dönem hafızası için "geçici alan" olarak kullanılabilir. Boş bellek pointer'ından sonraki 32 bit (yani `0x60` tan başlayan alan) ise kalıcı olarak sıfır olmalıdır ve bu alan boş dinamik bellek array'lerinin temel değeri olarak kullanılır. Bunlar ise demektir ki kullanılabilir hafıza `0x80` den başlar ve bu değer ise boş bellek pointer'ının ilk değeridir. Solidity'deki hafıza array'lerinin tamamı 32 bitin katları olacak şekilde yer kaplar.(Bu kural `bytes1[]` için

de geçerlidir fakat `bytes` ve `string` için geçerli değildir.) Çok boyutlu hafıza array'leri ise başka hafıza array'lerine pointer'lardır. Dinamik array'in uzunluğu array'in ilk slotunda saklanır ve diğer slotlara array'in elemanları gelir.

Uyarı: Statik boyutlandırılmış hafıza array'leri herhangi bir uzunluk alanına sahip değildir fakat bu sonradan dinamik ve statik boyutlandırılmış array'ler arasında daha kolay çevrimi sağlamak için eklenmiş olabilir. Yani bu kurala dayanarak ilerlememelisiniz.

Hafıza Güvenliği

Inline assembly kullanmadan; derleyici(compiler), iyi tanımlanmış bir durumda kalmak için her zaman belleğe güvenir. Bu özellikle *Yul IR üzerinden yeni kod oluşturma hattı Yul IR* ile ilgilidir. Bu kod parçası yerel değişkenleri stack üzerinden belleğe atarak stack-too-deep hatasından kaçınmayı sağlar ve eğer bazı kesin varsayımlara uyuyorsa ekstra bellek optimizasyonları uygulayabilir.

Biz her ne kadar Solidity'nin kendi bellek modeline saygı gösterilmesini önersek de Inline assembly belleği uyumsuz bir biçimde kullanmanızı sağlar. Bu nedenle stack değişkenlerini belleğe taşımak ve diğer bellek optimizasyonları, bir bellek işlemi içeren ya da Solidity değişkenlerini belleğe atayan tüm inline assembly bloklarında varsayılan olarak devredışı haldedir.

Fakat bir assembly bloğuna aşağıdaki şekilde özel olarak ek açıklamalar ekleyerek Solidity'nin bellek modeline uyduğunu belirtebilirsiniz:

```
assembly ("memory-safe") {
    ...
}
```

Bellek açısından güvenli bir assembly bloğu sadece aşağıdaki bellek bölümlerine erişebilir: - Sizin tarafınızdan yukarıda anlatıldığı gibi `allocate` benzeri bir mekanizma kullanarak atanmış bir bellek. - Solidity tarafından atanmış bellek, yani sizin referans verdiğiniz bellek array'inin sınırları içerisinde kalan alan. - Yukarıda bahsedilen 0 ile 64 bellek offset'leri arasında kalan geçici alan. - Assembly bloğunun başındaki boş bellek pointer'ının değerinden *sonra* konumlanmış geçici bellek, yani boş bellek pointer'ının güncellememiş hali için ayrılan bellek alanı.

Bunlara ek olarak, eğer bir assembly bloğu bellekteki bir Solidity değişkenine atanırsa bu erişimin yukarıda belirtilen bellek sınırları içerisinde olduğundan emin olmalısınız.

Belirtilen işlemler genellikle optimizer ile ilgili olduğu için assembly bloğu hata verse de verilen kısıtlamalar takip edilmeli. Bir örnek olarak aşağıda verilen assembly kod parçası bellek açısından güvenli değil. Sebebi ise `returndatasize()` fonksiyonunun değeri belirtilen 64 bitlik geçici bellek alanını aşabilir.

```
assembly {
  returndatacopy(0, 0, returndatasize())
  revert(0, returndatasize())
}
```

Fakat aşağıdaki kod ise bellek açısından *güvenli* dir. Çünkü boş bellek pointer'ının gösterdiği yerden sonrası güvenli bir şekilde geçici alan olarak kullanılabilir.

```
assembly ("memory-safe") {
  let p := mload(0x40)
  returndatacopy(p, 0, returndatasize())
  revert(p, returndatasize())
}
```

Unutmayın ki eğer bir atama yoksa boş bellek pointer'ını güncellemenize gerek yoktur ama belleği kullanmaya boş bellek pointer'ının verdiği offset'ten başlayabilirsiniz.

Eğer bellek işlemleri sıfır uzunluğunu kullanıyorsa -geçici alana düşmediği sürece- herhangi bir offset'i de kullanabilirsiniz.

```
assembly ("memory-safe") {
  revert(0, 0)
}
```

Unutmayın ki inline assembly içerisindeki bellek işlemleri bellek için güvenli olmadığı gibi bellekte referans tipinde olan Solidity değişkenlerine olan atamalar da bellek için güvenli olmayabilir. Aşağıdaki örnek bellek için güvenli değildir:

```
bytes memory x;
assembly {
  x := 0x40
}
x[0x20] = 0x42;
```

Belleğe erişim istemeyen işlemlerden oluşan ve bellek üzerindeki Solidity değişkenlerine atama yapmayan inline assembly otomatik olarak bellek için güvenli sayılır ve ekstra olarak belirtilmesine gerek duyulmaz.

Uyarı: Assembly'nin bellek modelini sağladığından emin olmak sizin sorumluluğunuzdadır. Eğer siz bir assembly bloğunu bellek için güvenli olarak tanımlayıp herhangi bir bellek hatası yaparsanız bu **kesinlikle**, doğru olmayan ya da tanımlanmamış bir davranışa sebep olur. Ve bu hata test yaparak kolay bir şekilde bulunamaz.

Eğer Solidity'nin farklı versiyonları ile uyumlu olacak şekilde bir kütüphane oluşturuyorsanız bir assembly bloğunun bellek için güvenli olduğunu özel bir komut ile belirtebilirsiniz:

```
/// @solidity memory-safe-assembly
assembly {
    ...
}
```

Unutmayın ki yorum satırları ile belirtmeyi gelecek bir sürümde kaldıracağız yani eğer geçmiş derleyici(compiler) sürümleri ile uyum konusunda yeterli bilgiye sahip değilseniz dialect string kullanmayı tercih edin.

3.11 Kopya Kağıdı

3.11.1 Operatörlerin Öncelik Sırası

Aşağıdaki tablo, değerlendirme sırasına göre listelenen operatörler için öncelik sırasını belirtir.

Öncelik	Tanım	Operatör
1	Son ek ile tırma ve azaltma	`++`, `--`
	Yeni ifade	`new <typename>`
	Dizi elamanı görüntüleme	`<array>[<index>]`
	Üye erişimi	`<object>.<member>`
	Fonksiyon çağırımı	`<func>(<args...>)`
	Parantezler	`(<statement>)`
2	Ön ek ile artırma ve azaltma	`++`, `--`
	Tekli çıkarma	`-`
	Tekli işlemler	`delete`
	Mantıksal 'DEĞİL'	`!`
	Bitsel 'DEĞİL'	`~`
3	Üs alma	`**`
4	Çarpma, bölme ve mod alma	`*`, `/`, `%`
5	Ekleme ve çıkarma	`+`, `-`
6	Bitsel değiştirme operatörleri	`<<`, `>>`
7	Bitsel 'VE'	`&`
8	Bitsel 'Özel veya'	`^`
9	Bitsel 'YA DA'	`|`
10	Eşitsizlik operatörleri	`<`, `>`, `<=`, `>=`
11	Eşitlik operatörleri	`==`, `!=`
12	Mantıksal 'VE'	`&&`
13	Mantıksal 'YA DA'	`||`
14	Üçlü operatör	`<conditional> ? <if-true> : <if-false>`
	Atama operatörleri	`=`, `|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=`
15	Virgül operatörü	`,`

3.11.2 Global Değişkenler

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI formatında gönderilen verinin ayrıştırılması sırasında, tipler ikinci argüman olarak parantez içinde verilir. Örneğin: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: ABI formatında verileri düzenler
- `abi.encodePacked(...) returns (bytes memory)`: Verilen argümanların *ABI formatında paketlenmiş veri* işlemini gerçekleştirir. Paketli ABI formatındaki verinin belirsiz olabileceğine dikkat edin!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: ABI, verilen bağımsız değişkenleri ikinciden başlayarak ABI olarak formatlar ve verilen dört baytlık seçicinin önüne ekler.
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Şuna eşdeğerdir `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)`: ABI, `functionPointer` çağrısını veri grupları içinde bulunan argümanlarla ABI olarak formatlar. Argümanlardaki tüm veri tipleri, fonksiyonun imzasi ile eşleşmesi kontrol edilir. Sonuç `abi.encodeWithSelector(functionPointer.selector, (...))` değerine eşittir
- `bytes.concat(...) returns (bytes memory)`: *Değişken sayıda bayt ve bytes1, ..., bytes32 argümanlarını bir bayt dizisine birleştirir*
- `string.concat(...) returns (string memory)`: *Değişken sayıda string argümanını tek bir string dizisinde birleştirir*

- `block.basefee` (`uint`): mevcut bloğun baz ücreti (EIP-3198 ve EIP-1559)
- `block.chainid` (`uint`): mevcut bloğun zincir kimliği
- `block.coinbase` (`address payable`): mevcut blok madencisinin adresi
- `block.difficulty` (`uint`): mevcut blok zorluğu
- `block.gaslimit` (`uint`): mevcut blok gas sınırı
- `block.number` (`uint`): mevcut blok numarası
- `block.timestamp` (`uint`): unix bazlı epoch (1/1/1970) gününden bu yana saniye biçiminde formatlanmış mevcut bloğun yaratılış zaman bilgisi
- `gasleft() returns (uint256)`: kalan gas
- `msg.data` (`bytes calldata`): bütün calldata
- `msg.sender` (`address`): mesajın göndericisi (mevcut çağırma için)
- `msg.sig` (`bytes4`): calldata'nın ilk 4 byte değeri (yani fonksiyon tanımlayıcısı)
- `msg.value` (`uint`): mesaj ile birlikte gönderilen wei miktarı
- `tx.gasprice` (`uint`): işlemin gas fiyatı
- `tx.origin` (`address`): işlemin göndericisi (tam çağrı zinciri)
- `assert(bool condition)`: koşul "yanlış" ise yürütmeyi iptal edilir ve durum değişikliklerini geri alır (dahili hata için kullanın)
- `require(bool condition)`: koşul "yanlış" ise yürütmeyi durdurur ve durum değişikliklerini geri alır (harici bileşende hatalı biçimlendirilmiş giriş veya hata için kullanın)
- `require(bool condition, string memory message)`: koşul "yanlış" ise yürütmeyi iptal eder ve durum değişikliklerini geri alır (harici bileşende hatalı biçimlendirilmiş giriş veya hata için kullanın). Ayrıca hata mesajıda verir.
- `revert()`: yürütmeyi iptal eder ve durum değişikliklerini geri alır
- `revert(string memory message)`: açıklayıcı bir string geri döndürerek yürütmeyi iptal eder ve durum değişikliklerini geri alır
- `blockhash(uint blockNumber) returns (bytes32)`: verilen bloğun hash'i - yalnızca en son 256 blok için çalışır
- `keccak256(bytes memory) returns (bytes32)`: girdinin Keccak-256 hash'ini hesaplar
- `sha256(bytes memory) returns (bytes32)`: cgirdinin SHA-256 hash'ini hesaplar
- `ripemd160(bytes memory) returns (bytes20)`: girdinin RIPEMD-160 hash'ini hesaplar
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: eliptik eğri imzasından açık anahtarla ilişkili adresi kurtarır veya hata durumunda sıfır döndürür.
- `addmod(uint x, uint y, uint k) returns (uint)`: toplama işleminin isteğe bağlı kesinlikte gerçekleştirildiği ve `2**256``da kapsamadığı ``(x + y) % k` değerini hesaplar. Sürüm 0.5.0'den başlayarak "k!= 0" olduğunu iddia eder.
- `mulmod(uint x, uint y, uint k) returns (uint)`: çarpmanın isteğe bağlı kesinlikte gerçekleştirildiği ve `2**256` değerinde kapsamadığı `(x * y) % k` değerini hesaplar. Sürüm 0.5.0'dan başlayarak `k != 0` olduğunu iddia eder.
- `this` (mevcut sözleşme tipi): mevcut sözleşme, açıkça "adres" veya "ödenecek adres"e dönüştürülebilir
- `super`: kalıtım(miras) hiyerarşisinde bir seviye daha yüksek sözleşme

- `selfdestruct(address payable recipient)`: mevcut sözleşmeyi imha edin, fonlarını verilen adrese gönderin
- `<address>.balance` (`uint256`): Wei biçimindeki *Adresler* bakiyesi
- `<address>.code` (`bytes memory`): ref:*address* adresindeki kod (boş olabilir)
- `<address>.codehash` (`bytes32`): ref:*address* kod hash'i
- `<address payable>.send(uint256 amount) returns (bool)`: verilen Wei miktarını *Adresler* 'ine gönderir, başarısız olması durumunda `false` döndürür
- `<address payable>.transfer(uint256 amount)`: verilen Wei miktarını *Adresler* 'ine gönderir, başarısız olması durumunda geri döner
- `type(C).name` (`string`): sözleşmenin ismi
- `type(C).creationCode` (`bytes memory`): verilen sözleşmenin bayt kodu oluşturma, bkz. *Type Information*.
- `type(C).runtimeCode` (`bytes memory`): verilen sözleşmenin çalışma zamanı bayt kodu, bkz. *Type Information*.
- `type(I).interfaceId` (`bytes4`): verilen arayüzün EIP-165 arayüz tanımlayıcısını içeren değer, bkz. *Type Information*.
- `type(T).min` (`T`): `T` tipi tarafından temsil edilebilen en küçük değer, bkz. *Type Information*.
- `type(T).max` (`T`): `T` tipi tarafından temsil edilebilen en büyük değer, bkz. *Type Information*.

3.11.3 Fonksiyon Görünürlük Belirteçleri

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

- `public`: harici ve dahili olarak görünür (depolama/durum değişkenleri için bir *alıcı fonksiyon* oluşturur)
- `private`: sadece mevcut sözleşmede görünür
- `external`: yalnızca harici olarak görünür (yalnızca fonksiyonlar için) - yani yalnızca mesajla çağrılabilir (`this.func` aracılığıyla)
- `internal`: sadece dahili olarak görünür

3.11.4 Modifiers

- `pure` fonksiyonlar için: Durumun değiştirilmesine veya erişime izin vermez.
- `view` fonksiyonlar için: Durum değişikliğine izin vermez.
- `payable` fonksiyonlar için: Bir çağrıyla birlikte Ether almalarını sağlar.
- `constant` durum değişkenleri için: Atamaya izin vermez (başlatma dışında), depolama yuvasını işgal etmez.
- `immutable` durum değişkenleri için: Başlatılma sırasında bir defa atamaya izin verir ve daha sonra sabit bir şekilde kalır. Kodda saklanır.
- `anonymous` event'ler için: Event imzasını başlık olarak saklamaz
- `indexed` event parametreleri için: Parametreyi başlık olarak saklar

- `virtual` fonksiyonlar ve modifier'lar için: Türetilmiş sözleşmelerde modifier'ların fonksiyonlarının değiştirilmesine izin verir.
- `override`: Bu fonksiyon, modifier veya genel durum değişkeninin, bir temel sözleşmedeki bir fonksiyonun veya modifier'ın davranışını değiştirdiğini ifade etmektedir.

3.12 Derleyicinin Kullanımı

3.12.1 Komut Satırı Derleyicisinin Kullanımı

Not: Bu bölüm, komut satırı modunda kullanılsa bile *solcjs* için geçerli değildir.

Temel Kullanım

Solidity deposunun(repository) derleme kaynaklarından biri de Solidity komut satırı derleyicisi olan `solc` dur. `solc --help` komutunu kullanmak size tüm seçeneklerin açıklamalarını verir. Derleyici, soyut bir sözdizimi ağacı (parse tree) üzerinde basit binary ve assembly'den gaz kullanımı tahminlerine kadar çeşitli çıktılar üretebilir. Sadece tek bir dosyayı derlemek istiyorsanız, `solc --bin sourceFile.sol` şeklinde çalıştırdığınızda binary dosyayı yazdıracaktır. Eğer `solc``un daha gelişmiş çıktı çeşitlerinden bazılarını elde etmek istiyorsanız, ``solc -o outputDirectory --bin --ast-compact-json --asm sourceFile.sol` kullanarak her öğeyi ayrı dosyalara çıktı olarak vermesini söylemek muhtemelen daha iyi bir seçenek olacaktır.

Optimize Edici Seçenekleri

Sözleşmenizi deploy etmeden önce, `solc --optimize --bin sourceFile.sol` kullanarak derleme yaparken optimize ediciyi etkinleştirmelisiniz. Standart olarak optimize edici, sözleşmenin ömrü boyunca 200 kez çağrıldığını varsayarak sözleşmeyi optimize edecektir (daha spesifik olarak, her bir işlem kodunun yaklaşık 200 kez çalıştırıldığını varsayar). İlk sözleşme dağıtımının daha ucuz olmasını ve daha sonraki fonksiyon yürütmelerinin(executions) daha pahalı olmasını istiyorsanız, `--optimize-runs=1` olarak ayarlayın. Çok sayıda işlem bekliyorsanız ve daha yüksek dağıtım maliyeti ve çıktı boyutunu önemsemiyorsanız, `--optimize-runs` değerini yüksek bir sayıya ayarlayın. Bu parametrenin aşağıdaki değerler üzerinde etkileri vardır (bu durum gelecekte değişebilir):

- fonksiyon gönderim prosedüründeki binary aramasının boyutu
- büyük sayılar veya dizeler gibi sabitlerin saklanma şekli

Base Path ve Import Remapping

Komut satırı derleyicisi içe aktarılan dosyaları dosya sisteminden otomatik olarak okuyacaktır, ancak aşağıdaki şekilde `prefix=path` kullanarak *path redirects* sağlamanız da mümkündür:

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

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`.

İçe aktarmaları aramak için dosya sistemine erişirken, :ref:` ./ veya ../ <direct-imports>` ile başlamayan dizinler, `--base-path` ve `--include-path` seçenekleri kullanılarak belirtilen dizinlere (veya temel yol belirtilmemişse geçerli çalışma dizinine) bağlı olarak değerlendirilir. Ayrıca, dizinin bu seçenekler aracılığıyla eklenen kısmı sözleşme metadatasında görünmeyecektir.

Güvenlik nedeniyle derleyicinin *hangi dizinlere erişebileceği konusunda kısıtlamaları vardır*. Komut satırında belirtilen kaynak dosyaların dizinlerine ve yeniden eşlemelerin hedef yollarına dosya okuyucu tarafından erişilmesine otomatik olarak izin verilir, ancak diğer her şey varsayılan olarak reddedilir. İlave yollara (ve bunların alt dizinlerine) `--allow-paths /sample/path,/another/sample/path` anahtarıyla izin verilebilir. `--base-path` ile belirtilen yol içindeki her şeye her zaman izin verilir.

Yukarıda anlatılanlar, derleyicinin içe aktarma yollarını nasıl ele aldığının basitleştirilmiş halidir. Örneklerle birlikte ayrıntılı bir açıklama ve uç noktaların tartışılması için lütfen *path resolution* bölümüne bakın.

Kütüphane Bağlantıları (Library Linking)

Sözleşmeleriniz *libraries* kullanıyorsa, bytecode'un `__$53aea86b7d70b31448b230b20ae141a537$__` şeklinde alt dizeler içerdiğini fark edeceksiniz. Bunlar gerçek kütüphane adresleri için yer tutucudardır. Yer tutucu, tam nitelikli kütüphane adının keccak256 hash'inin hex encoding'inin 34 karakterlik bir önekidir. Bayt kodu dosyası, yer tutucuların hangi kütüphaneleri temsil ettiğini belirlemeye yardımcı olmak için sonunda `// <placeholder> -> <fq library name>` şeklinde satırlar da içerecektir. Tam nitelikli kütüphane adının, kaynak dosyasının yolu ve `:` ile ayrılmış kütüphane adı olduğunu unutmayın. Bir bağlayıcı olarak `solc` kullanabilirsiniz, yani bu noktalarda sizin için kütüphane adreslerini ekleyecektir:

Her kütüphane için bir adres sağlamak üzere komutunuza `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890 file.sol:Heap=0xabCD5678901234567890123456789012345678890"` ekleyin (ayırıcı olarak virgül veya boşluk kullanın) veya dizeyi bir dosyada saklayın (satır başına bir kütüphane) ve `--libraries fileName` kullanarak `solc` çalıştırın.

Not: Solidity 0.8.1'den itibaren `=` kütüphane ve adres arasında ayırıcı olarak kabul etmektedir ve `:` ayırıcı olarak kullanımdan kaldırılmıştır. Gelecekte kaldırılacaktır. Şu anda `-libraries "file.sol:Math:0x1234567890123456789012345678901234567890 file.sol:Heap:0xabCD5678901234567890123456789012345678890"` da çalışacaktır.

Eğer `solc --standard-json` seçeneği ile çağrılırsa, standart girişte bir JSON girdisi (aşağıda açıklandığı gibi) bekleyecek ve standart çıkışta bir JSON çıktısı döndürecektir. Bu, daha karmaşık ve özellikle otomatikleştirilmiş kullanımlar için önerilen arayüzdür. İşlem her zaman "başarılı" durumda sonlanacak ve hataları JSON çıktısı aracılığıyla bildirecektir. `--base-path` seçeneği de standart-json modunda işlenir.

Eğer `solc --link` seçeneği ile çağrılırsa, tüm girdi dosyaları yukarıda verilen `__$53aea86b7d70b31448b230b20ae141a537$__`-formatında bağlanmamış binaryler (hex-encoded) olarak yorumlanır ve yerinde bağlanır (eğer girdi stdin`den okunuyorsa, stdout`a yazılır). Bu durumda `--libraries` dışındaki tüm seçenekler göz ardı edilir (`-o` dahil).

Uyarı: Sözleşme meta verilerini güncellemediğinden, oluşturulan bayt kodu üzerinde kütüphaneleri manuel olarak bağlamak önerilmez. Metadata derleme sırasında belirtilen kütüphanelerin bir listesini içerdiğinden ve bayt kodu bir metadata hash'i içerdiğinden, bağlama işleminin ne zaman yapıldığına bağlı olarak farklı binary dosyaları elde edersiniz.

Derleyiciye standart-JSON arayüzünü kullanıyorsanız `solc` seçeneğinin `--libraries` seçeneğini veya `libraries` anahtarını kullanarak bir sözleşme derlendiğinde derleyiciden kütüphaneleri bağlamasını istemelisiniz.

Not: Kütüphane yer tutucusu eskiden kütüphanenin hash'i yerine kütüphanenin kendisinin tam nitelikli adı olurdu. Bu biçim hala `solc --link` tarafından desteklenmektedir ancak derleyici artık bu biçimin çıktısını vermeyecektir. Bu değişiklik, tam nitelikli kütüphane adının yalnızca ilk 36 karakteri kullanılabildiğinden, kütüphaneler arasında bir

çakışma olasılığını azaltmak için yapılmıştır.

3.12.2 EVM Sürümünün Hedefe Ayarlanması

Sözleşme kodunuzu derlerken, belirli özelliklerden veya davranışlardan kaçınmak için derlenecek Ethereum sanal makine sürümünü belirtebilirsiniz.

Uyarı: Hatalı EVM sürümü için derleme yapmak yanlış, garip ve başarısız davranışlara neden olabilir. Lütfen, özellikle özel bir zincir çalıştırıyorsanız, uyumlu EVM sürümlerini kullandığınızdan emin olun.

Komut satırında, EVM sürümünü aşağıdaki gibi seçebilirsiniz:

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

ref:*standart JSON arayüzü <compiler-api> `de,* `"settings"`` alanında `"evmVersion"` anahtarını kullanın:

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

Hedef Seçenekleri

Aşağıda hedef EVM sürümlerinin bir listesi ve her sürümde derleyiciyle ilgili yapılan değişiklikler yer almaktadır. Her sürüm arasında geriye dönük uyumluluk garanti edilmez.

- **`homestead`**
  - (en eski sürüm)
- **`tangerineWhistle`**
  - Gaz tahmini ve optimize edici ile ilgili diğer hesaplara erişim için gaz maliyeti arttı.
  - Harici aramalar için varsayılan olarak gönderilen tüm gaz. Daha önce belirli bir miktarın tutulması gerekiyordu.
- **`spuriousDragon`**
  - Gaz tahmini ve optimize edici ile ilgili `exp` işlem kodu için gaz maliyeti arttı.
- **`byzantium`**
  - Assembly'de `returndatacopy`, `returndatasize` ve `staticcall` işlem kodları mevcuttur.
  - `staticcall` işlem kodu, kütüphane dışı görünüm veya pure fonksiyonları çağırırken kullanılır, bu da fonksiyonların EVM seviyesinde durumu değiştirmesini engeller, yani geçersiz tip dönüşümleri kullandığınızda bile geçerlidir.
  - Fonksiyon çağrılarından dönen dinamik verilere erişmek mümkündür.
  - `revert` işlem kodu tanıtıldı, bu da `revert()` işleminin gaz israfına yol açmayacağı anlamına geliyor.

- **constantinople**
  - Assembly'de `create2`, `extcodehash`, `shl`, `shr` ve `sar` işlem kodları mevcuttur.
  - Shifting operatörleri shifting opcodes kullanır ve bu nedenle daha az gaza ihtiyaç duyar.
- **petersburg**
  - Derleyici istanbul'da olduğu gibi aynı şekilde davranır.
- **istanbul**
  - Assembly'de `chainid` ve `selfbalance` opcode'ları mevcuttur.
- **berlin**
  - `SLOAD`, `*CALL`, `BALANCE`, `EXT*` ve `SELFDESTRUCT` için gaz maliyetleri arttı. Bu maliyetler derleyici bu tür operasyonlarda soğuk gaz maliyetlerini varsayar. Bu, gaz tahmini için geçerlidir ve optimize edicidir.
- **london (default)**
  - Bloğun taban ücretine (EIP-3198 ve EIP-1559) global `block.basefee` veya inline assembly`de `basefee()` aracılığıyla erişilebilir.

3.12.3 Derleyici JSON Girdisi ve Çıktısı Tanımı

Özellikle daha karmaşık ve otomatik kurulumlar için Solidity derleyicisi ile arayüz oluşturmanın önerilen yolu JSON-girdi-çıktı arayüzüdür. Aynı arayüz derleyicinin tüm dağıtımları tarafından sağlanır.

Alanlar genellikle değişikliğe tabidir, bazıları isteğe bağlıdır (belirtildiği gibi), ancak yalnızca geriye dönük uyumlu değişiklikler yapmaya çalışıyoruz.

Derleyici API'si JSON formatında bir girdi bekler ve derleme sonucunu JSON formatında bir çıktı olarak verir. Standart hata çıktısı kullanılmaz ve hatalar olsa bile işlem her zaman "başarılı" durumda sonlandırılır. Hatalar her zaman JSON çıktısının bir parçası olarak rapor edilir.

Aşağıdaki alt bölümlerde format bir örnek üzerinden açıklanmaktadır. Yorumlara elbette izin verilmez ve burada yalnızca açıklama amacıyla kullanılır.

Girdi Açıklaması

```
{
  // Gerekli: Kaynak kod dili. Şu anda "Solidity" ve "Yul" desteklenmektedir.
  "language": "Solidity",
  // Gerekli
  "sources":
  {
    // Buradaki anahtarlar kaynak dosyaların "global" isimleridir,
    // içe aktarmalar yeniden eşlemeler yoluyla diğer dosyaları kullanabilir (aşağıya␣
↪bakın).
    "myFile.sol":
    {
      // Opsiyonel: kaynak dosyanın keccak256 hash'i
      // URL'ler aracılığıyla içe aktarılmışsa alınan içeriği doğrulamak için kullanılır.
      "keccak256": "0x123...",
      // Gerekli ("content" kullanılmadığı sürece, aşağıya bakın): Kaynak dosyaya giden␣
↪URL(ler).
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
      // URL(ler) bu sırayla içe aktarılmalı ve sonuç keccak256 hash'iyle
      // (varsa) kontrol edilmelidir. Hash eşleşmezse veya URL(ler)den hiçbiri başarıyla
      // sonuçlanmazsa, bir hata oluşmalıdır.
      // Komut satırı arayüzü kullanılarak yalnızca dosya sistemi yolları desteklenir.
      // JavaScript arayüzü ile URL, kullanıcı tarafından sağlanan okuma geri çağrısına␣
↪aktarılır,
      // böylece geri çağrı tarafından desteklenen herhangi bir URL kullanılabilir.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "/tmp/path/to/file.sol"
        // Dosyalar kullanılıyorsa, dizinleri komut satırına şu yolla eklenmelidir
        // `--allow-paths <path>`.
      ]
    },
    "destructible":
    {
      // Opsiyonel: kaynak dosyanın keccak256 hash'i
      "keccak256": "0x234...",
      // Gerekli ("urls" kullanılmadığı sürece): kaynak dosyanın gerçek içeriği
      "content": "contract destructible is owned { function shutdown() { if (msg.sender␣
↪== owner) selfdestruct(owner); } }"
    }
  },
  // Opsiyonel
  "settings":
  {
    // Opsiyonel: Belirtilen aşamadan sonra derlemeyi durdurun. Şu anda burada sadece
↪"parsing" geçerlidir
    "stopAfter": "parsing",
    // Opsiyonel: Yeniden eşlemelerin sıralanmış listesi
    "remappings": [ ":g=/dir" ],
    // Opsiyonel: Optimize edici ayarları
    "optimizer": {
      // Varsayılan olarak devre dışıdır.
      // NOT: enabled=false hala bazı optimizasyonları açık bırakır. Aşağıdaki yorumlara␣
↪bakın.
      // UYARI: 0.8.6 sürümünden önce 'enabled' anahtarını atlamak, false olarak␣
↪ayarlamakla eşdeğer
      // değildi ve aslında tüm optimizasyonları devre dışı bırakıyordu.
      "enabled": true,
      // Kodu kaç kez çalıştırmayı planladığınıza göre optimize edin.
      // Düşük değerler ilk dağıtım maliyeti için daha fazla optimizasyon sağlarken,␣
↪yüksek
      // değerler yüksek frekanslı kullanım için daha fazla optimizasyon sağlayacaktır.
      "runs": 200,
      // Optimize edici bileşenleri ayrıntılı olarak açın veya kapatın.
      // Yukarıdaki "enabled" anahtarı, burada değiştirilebilecek iki
      // varsayılan değer sağlar. Eğer "details" verilmişse, "enabled" atlanabilir.
      "details": {
        // Ayrıntı verilmediğinde peephole optimizer her zaman açıktır,
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        // kapatmak için ayrıntıları kullanın.
        "peephole": true,
        // Ayrıntı verilmediğinde inliner her zaman açıktır,,
        // kapatmak için ayrıntıları kullanın.
        "inliner": true,
        // Kullanılmayan jumpdest kaldırıcı, ayrıntı verilmediğinde her zaman açıktır,
        // kapatmak için ayrıntıları kullanın.
        "jumpdestRemover": true,
        // Bazen değişmeli işlemlerde değişmezleri yeniden sıralar.
        "orderLiterals": false,
        // Yinelenen kod bloklarını kaldırır
        "deduplicate": false,
        // Ortak alt ifade eliminasyonu, bu en karmaşık adımdır ancak
        // aynı zamanda en büyük kazancı sağlayabilir.
        "cse": false,
        // Koddaki değişmez sayıların ve dizelerin gösterimini optimize edin.
        "constantOptimizer": false,
        // Yeni Yul optimize edici. Çoğunlukla ABI coder v2 ve inline assembly kodu
        // üzerinde çalışır.
        // Global optimizer ayarı ile birlikte etkinleştirilir ve
        // buradan devre dışı bırakılabilir.
        // Solidity 0.6.0'dan önce bu anahtar aracılığıyla etkinleştirilmesi gerekiyordu.
        "yul": false,
        // Yul optimize edici için ayarlama seçenekleri.
        "yulDetails": {
          // Değişkenler için yığın yuvalarının tahsisini iyileştirin, yığın yuvalarını␣
↪erken boşaltabilir.
          // Yul optimize edici etkinleştirilirse varsayılan olarak etkinleştirilir.
          "stackAllocation": true,
          // Uygulanacak optimizasyon adımlarını seçin.
          // İsteğe bağlıdır, atlanırsa optimize edici varsayılan sırayı kullanır.
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    // Derlenecek EVM sürümü.
    // Tip denetimini ve kod üretimini etkiler. Yerleşim yeri olabilir,
    // tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg, istanbul␣
↪or berlin
    "evmVersion": "byzantium",
    // Opsiyonel: Derleme işlem hattını Yul ara temsilinden geçecek şekilde değiştirin.
    // Bu varsayılan olarak yanlıştır.
    "viaIR": true,
    // Opsiyonel: Hata ayıklama ayarları
    "debug": {
      // Revert (ve require) sebep string' lerine nasıl işlem yapılır. Ayarlar
      // "default", "strip", "debug" ve "verboseDebug" şeklindedir.
      // "default" derleyici tarafından oluşturulan revert stringlerini enjekte etmez ve␣
↪kullanıcı tarafından sağlananları tutar.
      // "strip" tüm revert stringlerini (mümkünse, yani değişmezler kullanılıyorsa) yan␣
↪etkilerini koruyarak kaldırır
      // "debug" derleyici tarafından oluşturulan dahili geri dönüşler için stringler␣
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪enjekte eder, şimdilik ABI kodlayıcıları V1 ve V2 için uygulanmaktadır.
      // "verboseDebug" kullanıcı tarafından sağlanan revert stringlerine daha fazla␣
↪bilgi ekler (henüz uygulanmadı)
      "revertStrings": "default",
      // Opsiyonel: Üretilen EVM assembly ve Yul kodundaki yorumlara ne kadar ekstra
      // hata ayıklama bilgisi ekleneceği. Mevcut bileşenler şunlardır:
      // - `location`: Orijinal Solidity dosyasındaki ilgili öğenin konumunu belirten
      //    `@src <index>:<start>:<end>` biçimindeki ek açıklamalar, burada:
      //     - `<index>`, `@use-src` ek açıklamasıyla eşleşen dosya dizinidir,
      //     - `<start>` o konumdaki ilk baytın indeksidir,
      //     - `<end>` bu konumdan sonraki ilk baytın indeksidir.
      // - `snippet`: `@src` ile belirtilen konumdan tek satırlık bir kod parçacığı.
      //     Parçacık alıntılanır ve ilgili `@src` ek açıklamasını takip eder.
      // - `*`: Her şeyi talep etmek için kullanılabilecek joker karakter değeri.
      "debugInfo": ["location", "snippet"]
    },
    // Metadata ayarları (isteğe bağlı)
    "metadata": {
      // URL'leri değil, yalnızca gerçek içeriği kullan (varsayılan olarak false)
      "useLiteralContent": true,
      // Bayt koduna eklenen metadata hash'i için verilen hash yöntemini kullanın.
      // Metadata hash'i "none" seçeneği ile bayt kodundan kaldırılabilir.
      // Diğer seçenekler "ipfs" ve "bzzr1 "dir.
      // Seçenek atlanırsa, varsayılan olarak "ipfs" kullanılır.
      "bytecodeHash": "ipfs"
    },
    // Kütüphanelerin adresleri. Tüm kütüphaneler burada verilmezse,
    // çıktı verileri farklı olan bağlantısız nesnelerle sonuçlanabilir.
    "libraries": {
      // En üst düzey anahtar, kütüphanenin kullanıldığı kaynak dosyanın adıdır.
      // Yeniden eşlemeler kullanılıyorsa, bu kaynak dosya yeniden eşlemeler
      // uygulandıktan sonraki genel yolla eşleşmelidir.
      // Bu anahtar boş bir string ise, bu global bir seviyeyi ifade eder.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    },
    // Dosya ve sözleşme adlarına göre istenen çıktıları
    // seçmek için aşağıdakiler kullanılabilir.
    // Bu alan atlanırsa, derleyici yükler ve tür denetimi yapar,
    // ancak hatalar dışında herhangi bir çıktı üretmez.
    // Birinci seviye anahtar dosya adı, ikinci seviye anahtar ise sözleşme adıdır.
    // Boş bir sözleşme adı, bir sözleşmeye bağlı olmayan ancak AST gibi
    // tüm kaynak dosyaya bağlı olan çıktılar için kullanılır.
    // Sözleşme adı olarak bir yıldız, dosyadaki tüm sözleşmeleri ifade eder.
    // Benzer şekilde, dosya adı olarak bir yıldız tüm dosyalarla eşleşir.
    // Derleyicinin üretebileceği tüm çıktıları seçmek için
    // "outputSelection: { "*": { "*": [ "*" ], "": [ "*" ] } }"
    // ancak bunun derleme sürecini gereksiz yere yavaşlatabileceğini unutmayın.
    //
    // Mevcut çıktı türleri aşağıdaki gibidir:
    //
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    // Dosya seviyesi (sözleşme adı olarak boş dize gerekir):
    //   ast - Tüm kaynak dosyaların AST'si
    //
    // Sözleşme seviyesi (sözleşme adına veya "*" işaretine ihtiyaç duyar):
    //   abi - ABI
    //   devdoc - Geliştirici dokümantasyonu (natspec)
    //   userdoc - Kullanıcı dokümantasyonu (natspec)
    //   metadata - Metadata
    //   ir - Optimizasyondan önce kodun Yul ara temsili
    //   irOptimized - Optimizasyon sonrası ara temsil
    //   storageLayout - Sözleşmenin durum değişkenlerinin yuvaları, ofsetleri ve␣
↪türleri.
    //   evm.assembly - Yeni assembly formatı
    //   evm.legacyAssembly - JSON'daki eski tarz assembly formatı
    //   evm.bytecode.functionDebugData - Fonksiyon düzeyinde hata ayıklama bilgileri
    //   evm.bytecode.object - Bytecode objesi
    //   evm.bytecode.opcodes - Opcodes listesi
    //   evm.bytecode.sourceMap - Kaynak eşlemesi (hata ayıklama için yararlı)
    //   evm.bytecode.linkReferences - Bağlantı referansları (bağlantısı olmayan nesne␣
↪ise)
    //   evm.bytecode.generatedSources - Derleyici tarafından oluşturulan kaynaklar
    //   evm.deployedBytecode* - Deployed bytecode (evm.bytecode'un sahip olduğu tüm␣
↪seçeneklere sahiptir)
    //   evm.deployedBytecode.immutableReferences - AST kimliklerinden değişmezlere␣
↪referans veren bayt kodu aralıklarına eşleme
    //   evm.methodIdentifiers - Fonksiyon hash'lerinin listesi
    //   evm.gasEstimates - Fonksiyon gazı tahminleri
    //   ewasm.wast - WebAssembly S-expressions biçiminde Ewasm
    //   ewasm.wasm - WebAssembly binary formatında Ewasm
    //
    // Bir `evm`, `evm.bytecode`, `ewasm`, vb. kullanmanın bu çıktının her
    // hedef parçasını seçeceğini unutmayın. Ayrıca, `*` her şeyi istemek için joker␣
↪karakter olarak kullanılabilir.
    //
    "outputSelection": {
      "*": {
        "*": [
          "metadata", "evm.bytecode" // Her bir sözleşmenin metadata ve bytecode␣
↪çıktılarını etkinleştirin.
          , "evm.bytecode.sourceMap" // Her bir sözleşmenin kaynak eşleme çıktısını␣
↪etkinleştirin.
        ],
        "": [
          "ast" // Her bir dosyanın AST çıktısını etkinleştirin.
        ]
      },
      // Def dosyasında tanımlanan MyContract'ın abi ve opcodes çıktısını etkinleştirin.
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      }
    },
    // ModelChecker nesnesi deneyseldir ve değişikliklere tabidir.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    "modelChecker":
    {
      // Hangi sözleşmelerin konuşlandırılmış sözleşme olarak analiz edilmesi␣
↪gerektiğini seçin.
      "contracts":
      {
        "source1.sol": ["contract1"],
        "source2.sol": ["contract2", "contract3"]
      },
      // Bölme ve modulo işlemlerinin nasıl şifreleneceğini seçin.
      // `false` kullanıldığında, bunlar slack değişkenlerle çarpılarak
      // değiştirilir. Bu varsayılandır.
      // CHC motorunu kullanıyorsanız ve Horn çözücü olarak Spacer kullanmıyorsanız
      // (örneğin Eldarica kullanıyorsanız) burada `true` kullanılması önerilir.
      // Bu seçeneğin daha ayrıntılı bir açıklaması için Biçimsel Doğrulama bölümüne␣
↪bakın.
      "divModNoSlacks": false,
      // Hangi model denetleyici motorunun kullanılacağını seçin: all (varsayılan), bmc,␣
↪chc, none.
      "engine": "chc",
      // Kullanıcıya hangi tür değişmezlerin rapor edileceğini seçin: contract,␣
↪reentrancy.
      "invariants": ["contract", "reentrancy"],
      // Kanıtlanmamış tüm hedeflerin çıktısının alınıp alınmayacağını seçin. Varsayılan␣
↪değer `false`dir.
      "showUnproved": true,
      // Varsa, hangi çözücülerin kullanılması gerektiğini seçin.
      // Çözücülerin açıklaması için Biçimsel Doğrulama bölümüne bakın.
      "solvers": ["cvc4", "smtlib2", "z3"],
      // Hangi hedeflerin kontrol edilmesi gerektiğini seçin: constantCondition,
      // underflow, overflow, divByZero, balance, assert, popEmptyArray, outOfBounds.
      // Seçenek belirtilmezse, Solidity >=0.8.7 için underflow/overflow
      // hariç tüm hedefler varsayılan olarak kontrol edilir.
      // Hedeflerin açıklaması için Biçimsel Doğrulama bölümüne bakın.
      "targets": ["underflow", "overflow", "assert"],
      // Her SMT sorgusu için milisaniye cinsinden zaman aşımı.
      // Bu seçenek verilmezse, SMTChecker varsayılan olarak
      // deterministik bir kaynak sınırı kullanacaktır.
      // Verilen zaman aşımının 0 olması, herhangi bir sorgu için kaynak/zaman␣
↪kısıtlaması olmadığı anlamına gelir.
      "timeout": 20000
    }
  }
}
```

Çıktı Açıklaması

```
{
  // Opsiyonel: herhangi bir hata/uyarı/bilgi ile karşılaşılmadıysa mevcut değildir
  "errors": [
    {
      // Opsiyonel: Kaynak dosya içindeki konum.
      "sourceLocation": {
        "file": "sourceFile.sol",
        "start": 0,
        "end": 100
      },
      // Opsiyonel: Diğer yerler (örn. çelişkili beyanların olduğu yerler)
      "secondarySourceLocations": [
        {
          "file": "sourceFile.sol",
          "start": 64,
          "end": 92,
          "message": "Other declaration is here:"
        }
      ],
      // Zorunlu: Hata türü, örneğin "TypeError", "InternalCompilerError", "Exception",␣
↪vb.
      // Türlerin tam listesi için aşağıya bakınız.
      "type": "TypeError",
      // Zorunlu: Hatanın kaynaklandığı bileşen, örneğin "general", "ewasm", vb.
      "component": "general",
      // Zorunlu (" error", "warning" veya "info", ancak bunun gelecekte␣
↪genişletilebileceğini lütfen unutmayın)
      "severity": "error",
      // İsteğe bağlı: hatanın nedeni için benzersiz kod
      "errorCode": "3141",
      // Zorunlu
      "message": "Invalid keyword",
      // Opsiyonel: kaynak konumu ile biçimlendirilmiş mesaj
      "formattedMessage": "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // Bu, dosya düzeyinde çıktıları içerir.
  // OutputSelection ayarları ile sınırlandırılabilir/filtrelenebilir.
  "sources": {
    "sourceFile.sol": {
      // Kaynak tanımlayıcısı (kaynak eşlemelerinde kullanılır)
      "id": 1,
      // AST objesi
      "ast": {}
    }
  },
  // Bu, sözleşme düzeyindeki çıktıları içerir.
  // OutputSelection ayarları ile sınırlandırılabilir/filtrelenebilir.
  "contracts": {
    "sourceFile.sol": {
      // Kullanılan dilde sözleşme adı yoksa, bu alan boş bir dizeye eşit olmalıdır.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    "ContractName": {
      // Ethereum Sözleşmesi ABI'si. Boşsa, boş bir dizi olarak gösterilir.
      // bkz. https://docs.soliditylang.org/en/develop/abi-spec.html
      "abi": [],
      // Metadata Çıktısı belgelerine bakın (serileştirilmiş JSON stringi)
      "metadata": "{/* ... */}",
      // Kullanıcı dokümantasyonu (natspec)
      "userdoc": {},
      // Geliştirici dokümantasyonu (natspec)
      "devdoc": {},
      // Ara temsil (string)
      "ir": "",
      // Depolama Düzeni belgelerine bakın.
      "storageLayout": {"storage": [/* ... */], "types": {/* ... */} },
      // EVM'ye ilişkin çıktılar
      "evm": {
        // Assembly (string)
        "assembly": "",
        // Eski tarz assembly (object)
        "legacyAssembly": {},
        // Bytecode ve ilgili ayrıntılar.
        "bytecode": {
          // Fonksiyonlar düzeyinde veri hata ayıklama.
          "functionDebugData": {
            // Şimdi derleyicinin dahili ve kullanıcı tanımlı fonksiyonlarını içeren␣
↪bir fonksiyon kümesini takip edin.
            // Kümenin eksiksiz olması gerekmez.
            "@mint_13": { // Fonksiyonun dahili adı
              "entryPoint": 128, // Fonksiyonun başladığı byte offset bytecode (isteğe␣
↪bağlı)
              "id": 13, // Fonksiyon tanımının AST ID'si veya derleyiciye dahili␣
↪fonksiyonlar için null (isteğe bağlı)
              "parameterSlots": 2, // Fonksiyon parametreleri için EVM yığın yuvası␣
↪sayısı (isteğe bağlı)
              "returnSlots": 1 // Dönüş değerleri için EVM yığın yuvası sayısı (isteğe␣
↪bağlı)
            }
          },
          // Hex string olarak bytecode.
          "object": "00fe",
          // Opcodes listesi (string)
          "opcodes": "",
          // Bir string olarak kaynak eşlemesi. Kaynak eşleme tanımına bakın.
          "sourceMap": "",
          // Derleyici tarafından oluşturulan kaynakların dizisi. Şu anda yalnızca
          // tek bir Yul dosyası içerir.
          "generatedSources": [{
            // Yul AST
            "ast": {/* ... */},
            // Metin halindeki kaynak dosya (yorum içerebilir)
            "contents":"{ function abi_decode(start, end) -> data { data :=␣
↪calldataload(start) } }",
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            // Kaynak dosya ID'si, kaynak referansları için kullanılır, Solidity kaynak␣
↪dosyalarıyla aynı "ad alanı"
            "id": 2,
            "language": "Yul",
            "name": "#utility.yul"
          }],
          // Verilirse, bu bağlantısız bir nesnedir.
          "linkReferences": {
            "libraryFile.sol": {
              // Baytların bayt kodu içindeki ofsetleri.
              // Bağlantı, burada bulunan 20 baytın yerini alır.
              "Library1": [
                { "start": 0, "length": 20 },
                { "start": 200, "length": 20 }
              ]
            }
          }
        },
        "deployedBytecode": {
          /* ..., */ // Yukarıdaki ile aynı düzen.
          "immutableReferences": {
            // AST ID 3 ile değişmeze iki referans vardır, her ikisi de 32 bayt␣
↪uzunluğundadır. Bir tanesi
            // bytecode offset 42'de, diğeri bytecode offset 80'de.
            "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
          }
        },
        // Fonksiyon hash'lerinin listesi
        "methodIdentifiers": {
          "delegate(address)": "5c19a95c"
        },
        // Fonksiyon gaz tahminleri
        "gasEstimates": {
          "creation": {
            "codeDepositCost": "420000",
            "executionCost": "infinite",
            "totalCost": "infinite"
          },
          "external": {
            "delegate(address)": "25000"
          },
          "internal": {
            "heavyLifting()": "infinite"
          }
        }
      },
      // Ewasm ile ilgili çıktılar
      "ewasm": {
        // S-expressions biçimi
        "wast": "",
        // Binary formatı (hex string)
        "wasm": ""
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        }
      }
    }
  }
}
```

Hata Türleri

1. `JSONError`: JSON girdisi gerekli biçime uymuyor, örneğin girdi bir JSON nesnesi değil, dil desteklenmiyor vb.
2. `IOError`: Çözümlenemeyen URL veya sağlanan kaynaklardaki hash uyuşmazlığı gibi IO ve içe aktarma işleme hataları.
3. `ParserError`: Kaynak kodu dil kurallarına uygun değil.
4. `DocstringParsingError`: Yorum bloğundaki NatSpec etiketleri ayrıştırılamıyor.
5. `SyntaxError`: Sözdizimsel hata, örneğin `continue` bir `for` döngüsünün dışında kullanılmıştır.
6. `DeclarationError`: Geçersiz, çözümlenemeyen veya çakışan tanımlayıcı adları. ör. `Identifier not found`
7. `TypeError`: Geçersiz tür dönüşümleri, geçersiz atamalar vb. gibi tür sistemi içindeki hatalar.
8. `UnimplementedFeatureError`: Özellik derleyici tarafından desteklenmiyor, ancak gelecek sürümlerde desteklenmesi bekleniyor.
9. `InternalCompilerError`: Derleyicide tetiklenen dahili hata - bu bir sorun olarak raporlanmalıdır.
10. `Exception`: Derleme sırasında bilinmeyen hata - bu bir sorun olarak raporlanmalıdır.
11. `CompilerError`: Derleyici yığınının geçersiz kullanımı - bu bir sorun olarak raporlanmalıdır.
12. `FatalError`: Ölümcül hata doğru şekilde işlenmedi - bu bir sorun olarak raporlanmalıdır.
13. `Warning`: Derlemeyi durdurmayan, ancak mümkünse ele alınması gereken bir uyarı.
14. `Info`: Derleyicinin kullanıcının yararlı bulabileceğini düşündüğü, ancak tehlikeli olmayan ve mutlaka ele alınması gerekmeyen bilgiler.

3.12.4 Derleyici Araçları

solidity-upgrade

`solidity-upgrade` sözleşmelerinizi dil değişikliklerine yarı otomatik olarak yükseltmenize yardımcı olabilir. Her son sürüm için gerekli tüm değişiklikleri uygulamasa ve uygulayamasa da, aksi takdirde çok sayıda tekrarlayan manuel ayarlama gerektirecek olanları hala desteklemektedir.

Not: ''solidity-upgrade" işin büyük bir kısmını gerçekleştirir, ancak sözleşmelerinizin büyük olasılıkla daha fazla manuel ayarlamaya ihtiyacı olacaktır. Dosyalarınız için bir sürüm kontrol sistemi kullanmanızı öneririz. Bu, yapılan değişikliklerin gözden geçirilmesine ve sonunda geri alınmasına yardımcı olur.

Uyarı: `solidity-upgrade` tam veya hatasız olarak kabul edilmez, bu nedenle lütfen dikkatli kullanın.

Nasıl Çalışır?

Solidity kaynak dosya(lar)ını `solidity-upgrade [files]`'a aktarabilirsiniz. Bunlar, geçerli kaynak dosyanın dizini dışındaki dosyalara referans veren `import` ifadesini kullanıyorsa, `--allow-paths [directory]` seçeneğini geçerek dosyaların okunmasına ve içe aktarılmasına izin verilen dizinleri belirtmeniz gerekir. Eksik dosyaları `--ignore-missing` seçeneğini geçerek yok sayabilirsiniz.

`solidity-upgrade`, `libsolidity` tabanlıdır ve kaynak dosyalarınızı ayrıştırabilir, derleyebilir ve analiz edebilir ve içlerinde uygulanabilir kaynak yükseltmeleri bulabilir.

Kaynak yükseltmeleri, kaynak kodunuzda yapılan küçük metinsel değişiklikler olarak kabul edilir. Bunlar, verilen kaynak dosyaların bellek içi gösterimine uygulanır. İlgili kaynak dosyası varsayılan olarak güncellenir, ancak herhangi bir dosyaya yazmadan tüm yükseltme işlemini simüle etmek için `--dry-run` geçebilirsiniz.

Yükseltme işleminin iki aşaması vardır. İlk aşamada kaynak dosyalar ayrıştırılır ve kaynak kodu bu seviyede yükseltmek mümkün olmadığından, hatalar toplanır ve `--verbose` geçilerek günlüğe kaydedilebilir. Bu noktada kaynak yükseltmesi mevcut değildir.

İkinci aşamada, tüm kaynaklar derlenir ve tüm etkinleştirilmiş yükseltme analizi modülleri derleme ile birlikte çalıştırılır. Varsayılan olarak, mevcut tüm modüller etkinleştirilir. Daha fazla ayrıntı için lütfen *available modules* belgesini okuyun.

Bu, kaynak yükseltmeleri ile düzeltilebilecek derleme hatalarına neden olabilir. Hiçbir hata oluşmazsa, hiçbir kaynak yükseltmesi bildirilmez ve işiniz biter. Hatalar oluşursa ve bazı yükseltme modülleri bir kaynak yükseltmesi bildirirse, ilk bildirilen uygulanır ve verilen tüm kaynak dosyaları için derleme yeniden tetiklenir. Kaynak yükseltmeleri rapor edildiği sürece önceki adım tekrarlanır. Eğer hala hatalar oluşuyorsa, `--verbose` komutunu geçerek bunları günlüğe kaydedebilirsiniz. Herhangi bir hata oluşmazsa, sözleşmeleriniz günceldir ve derleyicinin en son sürümüyle derlenebilir.

Kullanılabilir Yükseltme Modülleri

Modül	Versiyon	Açıklama
`constructor`	0.5.0	Constructor"lar artık `constructor` anahtar sözcüğü kullanılarak tanımlanmalıdır.
`visibility`	0.5.0	Public fonksiyon görünürlüğü artık zorunlu, varsayılan değer `public`.
`abstract`	0.6.0	Bir sözleşme tüm fonksiyonlarını uygulamıyorsa `abstract` anahtar sözcüğü kullanılmalıdır.
`virtual`	0.6.0	Bir arayüz dışında uygulaması olmayan fonksiyonlar `virtual` olarak işaretlenmelidir.
`override`	0.6.0	Bir fonksiyon veya modifier geçersiz kılınırken, yeni `override` anahtar sözcüğü kullanılmalıdır
`dotsyntax`	0.7.0	Aşağıdaki sözdizimi kullanımdan kaldırılmıştır: `f.gas(...)()`, `f.value(...)()` ve `(new C).value(...)()`. Bu çağrıların yerine `f{gas: ..., value: ...}()` ve `(new C){value: ...}()`.
`now`	0.7.0	`now` anahtar sözcüğü kullanımdan kalktı. Bunun yerine *block.timestamp* ` kullanın.
`constructor-visibility`	0.5.0	Constructor'ların görünürlüğünü kaldırır.

Daha fazla ayrıntı için lütfen *0.5.0 release notes*, *0.6.0 release notes*, *0.7.0 release notes* ve *0.8.0 release notes* bölümlerini okuyun.

Özet bilgi(Synopsis)

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

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

Hata Raporları / Özellik Talepleri

Bir hata bulduysanız veya bir özellik isteğiniz varsa, lütfen Github'da bir sorun gönderin.

Örnek

`Source.sol` içinde aşağıdaki sözleşmeye sahip olduğunuzu varsayın:

```
pragma solidity >=0.6.0 <0.6.4;
// This will not compile after 0.7.0
// SPDX-License-Identifier: GPL-3.0
contract C {
    // BENİDÜZELT: constructor görünürlüğünü kaldırın ve sözleşmeyi abstract hale
↪getirin
    constructor() internal {}
}

contract D {
    uint time;

    function f() public payable {
        // BENİDÜZELT: now'u block.timestamp olarak değiştirin
        time = now;
    }
}

contract E {
    D d;

    // BENİDÜZELT: constructor görünürlüğünü kaldır
    constructor() public {}
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    function g() public {
        // BENİDÜZELT: .value(5) => {value: 5} olarak değiştirin
        d.f.value(5)();
    }
}
```

Gerekli Değişiklikler

Yukarıdaki sözleşme 0.7.0'dan itibaren derlenmeyecektir. Sözleşmeyi mevcut Solidity sürümüyle güncel hale getirmek için aşağıdaki yükseltme modüllerinin çalıştırılması gerekir: `constructor-visibility`, `now` ve `dotsyntax`. Daha fazla ayrıntı için lütfen *available modules* belgelendirmesini okuyun.

Yükseltmenin Çalıştırılması

Yükseltme modüllerinin `--modules` argümanı kullanılarak açıkça belirtilmesi önerilir.

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

Yukarıdaki komut aşağıda gösterildiği gibi tüm değişiklikleri uygular. Lütfen bunları dikkatlice inceleyin (pragmaların manuel olarak güncellenmesi gerekecektir).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
abstract contract C {
    // BENİDÜZELT: constructor görünürlüğünü kaldırın ve sözleşmeyi abstract hale␣
↪getirin
    constructor() {}
}

contract D {
    uint time;

    function f() public payable {
        // BENİDÜZELT: now'u block.timestamp olarak değiştirin
        time = block.timestamp;
    }
}

contract E {
    D d;

    // BENİDÜZELT: constructor görünürlüğünü kaldır
    constructor() {}

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

3.13 Analysing the Compiler Output

It is often useful to look at the assembly code generated by the compiler. The generated binary, i.e., the output of `solc --bin contract.sol`, is generally difficult to read. It is recommended to use the flag `--asm` to analyse the assembly output. Even for large contracts, looking at a visual diff of the assembly before and after a change is often very enlightening.

Consider the following contract (named, say `contract.sol`):

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function one() public pure returns (uint) {
        return 1;
    }
}
```

The following would be the output of `solc --asm contract.sol`

```
======= contract.sol:C =======
EVM assembly:
    /* "contract.sol":0:86  contract C {... */
  mstore(0x40, 0x80)
  callvalue
  dup1
  iszero
  tag_1
  jumpi
  0x00
  dup1
  revert
tag_1:
  pop
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x00
  codecopy
  0x00
  return
stop

sub_0: assembly {
        /* "contract.sol":0:86  contract C {... */
      mstore(0x40, 0x80)
      callvalue
      dup1
      iszero
      tag_1
      jumpi
      0x00
      dup1
      revert
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    tag_1:
      pop
      jumpi(tag_2, lt(calldatasize, 0x04))
      shr(0xe0, calldataload(0x00))
      dup1
      0x901717d1
      eq
      tag_3
      jumpi
    tag_2:
      0x00
      dup1
      revert
        /* "contract.sol":17:84  function one() public pure returns (uint) {... */
    tag_3:
      tag_4
      tag_5
      jump  // in
    tag_4:
      mload(0x40)
      tag_6
      swap2
      swap1
      tag_7
      jump  // in
    tag_6:
      mload(0x40)
      dup1
      swap2
      sub
      swap1
      return
    tag_5:
        /* "contract.sol":53:57  uint */
      0x00
        /* "contract.sol":76:77  1 */
      0x01
        /* "contract.sol":69:77  return 1 */
      swap1
      pop
        /* "contract.sol":17:84  function one() public pure returns (uint) {... */
      swap1
      jump  // out
        /* "#utility.yul":7:125   */
    tag_10:
        /* "#utility.yul":94:118   */
      tag_12
        /* "#utility.yul":112:117   */
      dup2
        /* "#utility.yul":94:118   */
      tag_13
      jump  // in
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    tag_12:
        /* "#utility.yul":89:92   */
      dup3
        /* "#utility.yul":82:119   */
      mstore
        /* "#utility.yul":72:125   */
      pop
      pop
      jump  // out
        /* "#utility.yul":131:353   */
    tag_7:
      0x00
        /* "#utility.yul":262:264   */
      0x20
        /* "#utility.yul":251:260   */
      dup3
        /* "#utility.yul":247:265   */
      add
        /* "#utility.yul":239:265   */
      swap1
      pop
        /* "#utility.yul":275:346   */
      tag_15
        /* "#utility.yul":343:344   */
      0x00
        /* "#utility.yul":332:341   */
      dup4
        /* "#utility.yul":328:345   */
      add
        /* "#utility.yul":319:325   */
      dup5
        /* "#utility.yul":275:346   */
      tag_10
      jump  // in
    tag_15:
        /* "#utility.yul":229:353   */
      swap3
      swap2
      pop
      pop
      jump  // out
        /* "#utility.yul":359:436   */
    tag_13:
      0x00
        /* "#utility.yul":425:430   */
      dup2
        /* "#utility.yul":414:430   */
      swap1
      pop
        /* "#utility.yul":404:436   */
      swap2
      swap1
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
      pop
      jump  // out

    auxdata:␣
↪0xa2646970667358221220a5874f19737ddd4c5d77ace1619e5160c67b3d4bedac75fce908fed32d98899864736f6c6378273
}
```

Alternatively, the above output can also be obtained from Remix, under the option "Compilation Details" after compiling a contract.

Notice that the `asm` output starts with the creation / constructor code. The deploy code is provided as part of the sub object (in the above example, it is part of the sub-object `sub_0`). The `auxdata` field corresponds to the contract *metadata*. The comments in the assembly output point to the source location. Note that `#utility.yul` is an internally generated file of utility functions that can be obtained using the flags `--combined-json generated-sources, generated-sources-runtime`.

Similarly, the optimized assembly can be obtained with the command: `solc --optimize --asm contract.sol`. Often times, it is interesting to see if two different sources in Solidity result in the same optimized code. For example, to see if the expressions `(a * b) / c`, `a * b / c` generates the same bytecode. This can be easily done by taking a `diff` of the corresponding assembly output, after potentially stripping comments that reference the source locations.

Not: The `--asm` output is not designed to be machine readable. Therefore, there may be breaking changes on the output between minor versions of solc.

3.14 Solidity IR-based Codegen Changes

Solidity can generate EVM bytecode in two different ways: Either directly from Solidity to EVM opcodes ("old codegen") or through an intermediate representation ("IR") in Yul ("new codegen" or "IR-based codegen").

The IR-based code generator was introduced with an aim to not only allow code generation to be more transparent and auditable but also to enable more powerful optimization passes that span across functions.

You can enable it on the command line using `--via-ir` or with the option `{"viaIR": true}` in standard-json and we encourage everyone to try it out!

For several reasons, there are tiny semantic differences between the old and the IR-based code generator, mostly in areas where we would not expect people to rely on this behaviour anyway. This section highlights the main differences between the old and the IR-based codegen.

3.14.1 Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- The order of state variable initialization has changed in case of inheritance.

  The order used to be:

  - All state variables are zero-initialized at the beginning.
  - Evaluate base constructor arguments from most derived to most base contract.
  - Initialize all state variables in the whole inheritance hierarchy from most base to most derived.
  - Run the constructor, if present, for all contracts in the linearized hierarchy from most base to most derived.

New order:

- All state variables are zero-initialized at the beginning.
- Evaluate base constructor arguments from most derived to most base contract.
- For every contract in order from most base to most derived in the linearized hierarchy:
  1. Initialize state variables.
  2. Run the constructor (if present).

This causes differences in contracts where the initial value of a state variable relies on the result of the constructor in another contract:

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

contract A {
    uint x;
    constructor() {
        x = 42;
    }
    function f() public view returns(uint256) {
        return x;
    }
}
contract B is A {
    uint public y = f();
}
```

Previously, `y` would be set to 0. This is due to the fact that we would first initialize state variables: First, `x` is set to 0, and when initializing `y`, `f()` would return 0 causing `y` to be 0 as well. With the new rules, `y` will be set to 42. We first initialize `x` to 0, then call A's constructor which sets `x` to 42. Finally, when initializing `y`, `f()` returns 42 causing `y` to be 42.

- When storage structs are deleted, every storage slot that contains a member of the struct is set to zero entirely. Formerly, padding space was left untouched. Consequently, if the padding space within a struct is used to store data (e.g. in the context of a contract upgrade), you have to be aware that `delete` will now also clear the added member (while it wouldn't have been cleared in the past).

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

contract C {
    struct S {
        uint64 y;
        uint64 z;
    }
    S s;
    function f() public {
        // ...
        delete s;
        // s occupies only first 16 bytes of the 32 bytes slot
        // delete will write zero to the full slot
    }
}
```

We have the same behavior for implicit delete, for example when array of structs is shortened.

- Function modifiers are implemented in a slightly different way regarding function parameters and return variables. This especially has an effect if the placeholder `_;` is evaluated multiple times in a modifier. In the old code generator, each function parameter and return variable has a fixed slot on the stack. If the function is run multiple times because `_;` is used multiple times or used in a loop, then a change to the function parameter's or return variable's value is visible in the next execution of the function. The new code generator implements modifiers using actual functions and passes function parameters on. This means that multiple evaluations of a function's body will get the same values for the parameters, and the effect on return variables is that they are reset to their default (zero) value for each execution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0;
contract C {
    function f(uint a) public pure mod() returns (uint r) {
        r = a++;
    }
    modifier mod() { _; _; }
}
```

If you execute `f(0)` in the old code generator, it will return 2, while it will return 1 when using the new code generator.

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

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

The function `C.foo()` returns the following values:

  - Old code generator: `1` as the return variable is initialized to `0` only once before the first `_;` evaluation and then overwritten by the `return 1;`. It is not initialized again for the second `_;` evaluation and `foo()` does not explicitly assign it either (due to `active == false`), thus it keeps its first value.
  - New code generator: `0` as all parameters, including return parameters, will be re-initialized before each `_;` evaluation.

- For the old code generator, the evaluation order of expressions is unspecified. For the new code generator, we try to evaluate in source order (left to right), but do not guarantee it. This can lead to semantic differences.

  For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function preincr_u8(uint8 a) public pure returns (uint8) {
        return ++a + a;
    }
}
```

The function `preincr_u8(1)` returns the following values:

- Old code generator: 3 (`1 + 2`) but the return value is unspecified in general
- New code generator: 4 (`2 + 2`) but the return value is not guaranteed

On the other hand, function argument expressions are evaluated in the same order by both code generators with the exception of the global functions `addmod` and `mulmod`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function add(uint8 a, uint8 b) public pure returns (uint8) {
        return a + b;
    }
    function g(uint8 a, uint8 b) public pure returns (uint8) {
        return add(++a + ++b, a + b);
    }
}
```

The function `g(1, 2)` returns the following values:

- Old code generator: `10` (`add(2 + 3, 2 + 3)`) but the return value is unspecified in general
- New code generator: `10` but the return value is not guaranteed

The arguments to the global functions `addmod` and `mulmod` are evaluated right-to-left by the old code generator and left-to-right by the new code generator. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f() public pure returns (uint256 aMod, uint256 mMod) {
        uint256 x = 3;
        // Old code gen: add/mulmod(5, 4, 3)
        // New code gen: add/mulmod(4, 5, 5)
        aMod = addmod(++x, ++x, x);
        mMod = mulmod(++x, ++x, x);
    }
}
```

The function `f()` returns the following values:

- Old code generator: `aMod = 0` and `mMod = 2`
- New code generator: `aMod = 4` and `mMod = 0`

- The new code generator imposes a hard limit of `type(uint64).max` (`0xffffffffffffffff`) for the free memory pointer. Allocations that would increase its value beyond this limit revert. The old code generator does not have this limit.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.8.0;
contract C {
    function f() public {
        uint[] memory arr;
        // allocation size: 576460752303423481
        // assumes freeMemPtr points to 0x80 initially
        uint solYulMaxAllocationBeforeMemPtrOverflow = (type(uint64).max - 0x80 -␣
↪31) / 32;
        // freeMemPtr overflows UINT64_MAX
        arr = new uint[](solYulMaxAllocationBeforeMemPtrOverflow);
    }
}
```

The function *f()* behaves as follows:

- Old code generator: runs out of gas while zeroing the array contents after the large memory allocation
- New code generator: reverts due to free memory pointer overflow (does not run out of gas)

3.14.2 Internals

Internal function pointers

The old code generator uses code offsets or tags for values of internal function pointers. This is especially complicated since these offsets are different at construction time and after deployment and the values can cross this border via storage. Because of that, both offsets are encoded at construction time into the same value (into different bytes).

In the new code generator, function pointers use internal IDs that are allocated in sequence. Since calls via jumps are not possible, calls through function pointers always have to use an internal dispatch function that uses the `switch` statement to select the right function.

The ID `0` is reserved for uninitialized function pointers which then cause a panic in the dispatch function when called.

In the old code generator, internal function pointers are initialized with a special function that always causes a panic. This causes a storage write at construction time for internal function pointers in storage.

Cleanup

The old code generator only performs cleanup before an operation whose result could be affected by the values of the dirty bits. The new code generator performs cleanup after any operation that can result in dirty bits. The hope is that the optimizer will be powerful enough to eliminate redundant cleanup operations.

For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f(uint8 a) public pure returns (uint r1, uint r2)
    {
        a = ~a;
        assembly {
            r1 := a
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        }
        r2 = a;
    }
}
```

The function `f(1)` returns the following values:

- Old code generator: (`fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe`, `0000000000000000000000000000000000000000000000000000000000000fe`)
- New code generator: (`0000000000000000000000000000000000000000000000000000000000000fe`, `0000000000000000000000000000000000000000000000000000000000000fe`)

Note that, unlike the new code generator, the old code generator does not perform a cleanup after the bit-not assignment (`a = ~a`). This results in different values being assigned (within the inline assembly block) to return value `r1` between the old and new code generators. However, both code generators perform a cleanup before the new value of `a` is assigned to `r2`.

3.15 Depolama Alanındaki Durum Değişkenlerinin Düzeni

Sözleşmelerin durum değişkenleri, birden fazla değerin bazen aynı depolama yuvasını(slot) kullanacağı şekilde kompakt bir şekilde depolanır. Dinamik olarak boyutlandırılmış diziler(arrays) ve mappingler (aşağıya bakınız) hariç olmak üzere, diğer tüm veriler `0` yuvasında saklanan ilk durum değişkeninden başlamak üzere bitişik bir şekilde öğe öğe saklanır. Her değişken için, değişkenin türüne göre bayt cinsinden bir boyut belirlenir. 32 bayttan daha az bir değere ihtiyaç duyan birden fazla bitişik öğe aşağıdaki kurallara uygun olarak eğer mümkünse tek bir depolama yuvasında paketlenir:

- Bir depolama yuvasındaki ilk öğe alt sıraya hizalanmış olarak saklanır.
- Değer türleri, depolanmak için yalnızca gerek duydukları kadar bayt kullanır.
- Bir değer türü bir depolama yuvasının kalan kısmına sığmazsa, bir sonraki depolama yuvasında saklanır.
- Struct'lar ve dizi(array) verileri için her zaman yeni bir yuva başlatılır. Ve öğeler bu kurallara göre sıkıca paketlenir.
- Struct veya dizi (array) verilerini izleyen öğeler için her zaman yeni bir depolama yuvası başlatır.

Kalıtım kullanan sözleşmeler için durum değişkenlerinin sıralaması, en temeldeki sözleşmeden başlayarak sözleşmelerin C3-doğrusallaştırılmış sırasına göre belirlenir. Eğer yukarıdaki kurallara da uygunsa, farklı sözleşmelerdeki durum değişkenleri aynı depolama yuvasını paylaşabilir.

Structure'ların ve dizilerin(arrays) elemanları, ayrı ayrı değerler şeklinde verilmiş gibi birbirlerinden sonra saklanırlar.

Uyarı: 32 bayttan daha küçük değerdeki elemanları kullanırken, sözleşmenizin gas kullanımı daha yüksek olabilir. Bunun nedeni, ESM'nin bir seferde 32 bayt üzerinde çalışmasıdır. Bu nedenle, eleman bundan daha küçükse, ESM'nin elemanın boyutunu 32 bayttan istenen boyuta düşürmek için daha fazla işlem kullanması gerekir.

Depolama değerleriyle uğraşıyorsanız, küçültülmüş boyutlu türleri kullanmak faydalı olabilir, çünkü derleyici birden fazla öğeyi tek bir depolama yuvasına yerleştirecek ve böylece birden fazla okuma veya yazma işlemini tek bir işlemde birleştirecektir. Ancak bir yuvadaki tüm değerleri aynı anda okumuyor veya yazmıyorsanız, bunun ters bir etkisi olabilir: Çok değerli bir depolama yuvasına bir değer yazıldığı zaman, depolama yuvasının önce okunması ve ardından aynı yuvadaki diğer verilerin yok edilmemesi için yeni değerler ile birleştirilmesi gerekir.

Fonksiyon argümanları veya bellek değerleriyle uğraşırken, derleyici bu değerleri paketlemediği için bu durumun herhangi bir faydası yoktur.

Son olarak, ESM'nin bunu optimize etmesine izin vermek için, depolama değişkenlerinizi ve `struct` üyelerinizi sıkıca paketlenebilecekleri şekilde sıralamaya çalıştığınızdan emin olun. Örneğin, saklama değişkenlerinizi `uint128, uint256, uint128` yerine `uint128, uint128, uint256` şeklinde bildirdiğinizde, ilk örnek yalnızca iki saklama alanı kaplarken ikincisi üç saklama alanı kaplayacaktır.

Not: Depolama alanındaki durum değişkenlerinin düzeni, depolama pointer'larının kütüphanelere aktarılabilmesi nedeniyle Solidity'nin harici arayüzünün bir parçası olarak kabul edilir. Bu, bu bölümde özetlenen kurallarda yapılacak herhangi bir değişikliğin dilde işleyişi bozan bir değişiklik olarak kabul edileceği ve kritik yapısı nedeniyle uygulanmadan önce çok dikkatli bir şekilde düşünülmesi gerekeceği anlamına gelir. Böyle bir işleyişi bozan değişiklik durumunda, derleyicinin eski düzeni(layout) destekleyecek bir bytecode üreteceği bir uyumluluk modu yayınlamak isteriz.

3.15.1 Mapping'ler ve Dinamik Diziler(Arrays)

Tahmin edilemeyen boyutları nedeniyle, mapping'ler ve dinamik boyutlu dizi türleri kendilerinden önceki ve sonraki durum değişkenlerinin "arasında" saklanamaz. Bunun yerine, *yukarıdaki* depolama kurallarına göre yalnızca 32 bayt kapladıkları kabul edilir ve içerdikleri öğeler bir Keccak-256 hash'i kullanılarak hesaplanan farklı bir depolama yuvasından başlayarak depolanır.

Mapping veya dizinin depolama konumunun *depolama düzeni kuralları* uygulandıktan sonra `p` yuvası olduğunu varsayalım. Dinamik diziler için, bu yuva dizideki eleman sayısını saklar (bayt dizileri ve stringler bir istisnadır, bkz. *aşağıda*). Mapping'ler için yuva boş kalır, ancak yine de yan yana duran iki mapping olsa bile içeriklerinin farklı depolama konumlarında sonlanmasını sağlamak için gereklidir.

Dizi(array) verileri `keccak256(p)` adresinden başlayarak yerleştirilir ve statik olarak boyutlandırılmış dizi verileriyle aynı biçimde düzenlenir: Elemanlar birbiri ardına sıralanır ve elemanlar 16 bayttan uzun değilse potansiyel olarak depolama yuvalarını paylaşırlar. Dinamik dizilerin dinamik dizileri bu kuralı özyinelemeli(recursive) şekilde uygular. `x` türünün `uint24[][]` olduğu `x[i][j]` öğesinin konumu aşağıdaki gibi hesaplanır (yine `x` öğesinin kendisinin `p` yuvasında saklandığını varsayarak): Yuva `keccak256(keccak256(p) + i) + floor(j / floor(256 / 24))` ve eleman `v` yuva verisinden `(v >> ((j % floor(256 / 24)) * 24)) & type(uint24).max`.

Bir `k` mapping anahtarına karşılık gelen değer `keccak256(h(k) . p)` adresinde bulunur; burada `.` birleştirme, `h` ise türüne bağlı olarak anahtara uygulanan bir fonksiyondur:

- değer türleri için, `h` değeri bellekte depolarken olduğu gibi 32 bayt olarak doldurur.
- stringler ve byte dizileri için, `h(k)` sadece doldurulmamış veridir.

Mapping değeri değer olmayan bir türse, hesaplanan yuva verinin başlangıcını işaret eder. Örneğin, değer struct türündeyse, üyeye ulaşmak için struct üyesine karşılık gelen bir ofset eklemeniz gerekir.

Örnek olarak, aşağıdaki sözleşmeye bakalım:

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


contract C {
    struct S { uint16 a; uint16 b; uint256 c; }
    uint x;
    mapping(uint => mapping(uint => S)) data;
}
```

Let us compute the storage location of `data[4][9].c`. The position of the mapping itself is 1 (the variable `x` with 32 bytes precedes it). This means `data[4]` is stored at `keccak256(uint256(4) . uint256(1))`. The type of `data[4]` is again a mapping and the data for `data[4][9]` starts at slot `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1)))`. The slot offset of the member `c` inside the struct `S` is 1 because `a` and `b` are packed in a single slot. This means the slot for `data[4][9].c` is `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`. The type of the value is `uint256`, so it uses a single slot.

`bytes` ve `string`

`bytes` ve `string` aynı şekilde şifrelenir. Genel olarak, şifreleme `bytes1[]` şifrelemesine benzer; dizinin kendisi için bir yuva ve bu yuvanın konumunun `keccak256` hash`i kullanılarak hesaplanan bir veri alanı vardır. Ancak, küçük değerler için (32 bayttan daha küçük) dizi elemanları uzunluklarıyla birlikte aynı yuvada saklanır.

Özellikle: veri en fazla `31` bayt uzunluğundaysa, elemanlar yüksek sıralı baytlarda (sola hizalı bir şekilde) saklanır ve en düşük sıralı baytta `uzunluk * 2` değeri saklanır. `32` veya daha fazla bayt uzunluğundaki verileri saklayan bayt dizileri için, `p` ana yuvası `length * 2 + 1` değerini saklar ve veriler her zamanki gibi `keccak256(p)` içinde saklanır. Bu, en düşük bit'in ayarlanıp ayarlanmadığını kontrol ederek kısa bir diziyi uzun bir diziden ayırt edebileceğiniz anlamına gelir: kısa (ayarlanmamış) ve uzun (ayarlanmış).

Not: Geçersiz olarak şifrelenmiş yuvaların işlenmesi şu anda desteklenmemektedir ancak gelecekte bu özellik eklenebilir. IR aracılığıyla derleme yapıyorsanız, geçersiz olarak kodlanmış bir yuvayı okumak `Panic(0x22)` hatasıyla sonuçlanır.

3.15.2 JSON Çıktısı

Bir sözleşmenin depolama düzeni *standart JSON arayüzü* aracılığıyla talep edilebilir. Çıktı, `storage` ve `types` olmak üzere iki anahtar içeren bir JSON nesnesidir. `storage` nesnesi, her bir elemanın aşağıdaki forma sahip olduğu bir dizidir:

```
{
    "astId": 2,
    "contract": "fileA:A",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

Yukarıdaki örnek, `fileA` kaynak biriminden `contract A { uint x; }` depolama düzenidir ve

- `astId` durum değişkeninin bildiriminin AST node'unun id'sidir
- `contract`, ön ek olarak yolunu da içeren sözleşmenin adıdır
- `label` durum değişkeninin adıdır
- `offset` şifrelemeye göre depolama yuvası içindeki bayt cinsinden ofsettir
- `slot` durum değişkeninin bulunduğu veya başladığı depolama yuvasıdır. Bu sayı çok büyük olabilir ve bu nedenle JSON değeri bir dize olarak gösterilir.
- `type` değişkenin tip bilgisi için anahtar olarak kullanılan bir tanımlayıcıdır (aşağıda açıklanmıştır)

Verilen `typep`, bu durumda `t_uint256`, `types` içinde şu forma sahip bir elemanı temsil eder:

```
{
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32",
}
```

nerede

- `encoding` verinin depolama alanında nasıl kodlandığı, olası değerler şunlardır:
  - `inplace`: veri depolama alanında bitişik olarak yerleştirilir (bkz *above*).
  - `mapping`: Keccak-256 hash tabanlı yöntem (bkz *above*).
  - `dynamic_array`: Keccak-256 hash tabanlı yöntem (bkz *above*).
  - `bytes`: veri boyutuna bağlı olarak tek slot veya Keccak-256 hash tabanlı (bkz *above*).
- `label` kanonik tip adıdır.
- **`numberOfBytes` kullanılan bayt sayısıdır (ondalık bir dize olarak).**
  Eğer `numberOfBytes` > `32` ise bunun birden fazla slot kullanıldığı anlamına geldiğini unutmayın.

Bazı türler yukarıdaki dört bilginin yanı sıra ekstra bilgilere de sahiptir. Mappingler `key` ve `value` türlerini içerir (yine bu tür mappingindeki bir girdiye referansta bulunur), diziler `base` türüne sahiptir ve structlar `members` türlerini üst düzey `storage` ile aynı formatta listeler (bkz *above*).

Not: Bir sözleşmenin depolama düzeninin JSON çıktısı hala deneysel olarak kabul edilir ve Solidity'nin işleyişi bozmayan sürümlerinde değiştirilebilir.

Aşağıdaki örnekte, değer ve referans türleri, paketlenmiş olarak şifrelenmiş türler ve iç içe geçmiş türler içeren bir sözleşme ve depolama düzeni gösterilmektedir.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    uint x;
    uint y;
    S s;
    address addr;
    mapping (uint => mapping (address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}
```

```
{
  "storage": [
```

(sonraki sayfaya devam)

(önceki sayfadan devam)


```
    {
      "astId": 15,
      "contract": "fileA:A",
      "label": "x",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
    },
    {
      "astId": 17,
      "contract": "fileA:A",
      "label": "y",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    },
    {
      "astId": 20,
      "contract": "fileA:A",
      "label": "s",
      "offset": 0,
      "slot": "2",
      "type": "t_struct(S)13_storage"
    },
    {
      "astId": 22,
      "contract": "fileA:A",
      "label": "addr",
      "offset": 0,
      "slot": "6",
      "type": "t_address"
    },
    {
      "astId": 28,
      "contract": "fileA:A",
      "label": "map",
      "offset": 0,
      "slot": "7",
      "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
    },
    {
      "astId": 31,
      "contract": "fileA:A",
      "label": "array",
      "offset": 0,
      "slot": "8",
      "type": "t_array(t_uint256)dyn_storage"
    },
    {
      "astId": 33,
      "contract": "fileA:A",
      "label": "s1",
```


(sonraki sayfaya devam)

(önceki sayfadan devam)


```
      "offset": 0,
      "slot": "9",
      "type": "t_string_storage"
    },
    {
      "astId": 35,
      "contract": "fileA:A",
      "label": "b1",
      "offset": 0,
      "slot": "10",
      "type": "t_bytes_storage"
    }
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    },
    "t_array(t_uint256)2_storage": {
      "base": "t_uint256",
      "encoding": "inplace",
      "label": "uint256[2]",
      "numberOfBytes": "64"
    },
    "t_array(t_uint256)dyn_storage": {
      "base": "t_uint256",
      "encoding": "dynamic_array",
      "label": "uint256[]",
      "numberOfBytes": "32"
    },
    "t_bool": {
      "encoding": "inplace",
      "label": "bool",
      "numberOfBytes": "1"
    },
    "t_bytes_storage": {
      "encoding": "bytes",
      "label": "bytes",
      "numberOfBytes": "32"
    },
    "t_mapping(t_address,t_bool)": {
      "encoding": "mapping",
      "key": "t_address",
      "label": "mapping(address => bool)",
      "numberOfBytes": "32",
      "value": "t_bool"
    },
    "t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
      "encoding": "mapping",
      "key": "t_uint256",
      "label": "mapping(uint256 => mapping(address => bool))",
```


(sonraki sayfaya devam)

(önceki sayfadan devam)


```
      "numberOfBytes": "32",
      "value": "t_mapping(t_address,t_bool)"
    },
    "t_string_storage": {
      "encoding": "bytes",
      "label": "string",
      "numberOfBytes": "32"
    },
    "t_struct(S)13_storage": {
      "encoding": "inplace",
      "label": "struct A.S",
      "members": [
        {
          "astId": 3,
          "contract": "fileA:A",
          "label": "a",
          "offset": 0,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 5,
          "contract": "fileA:A",
          "label": "b",
          "offset": 16,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 9,
          "contract": "fileA:A",
          "label": "staticArray",
          "offset": 0,
          "slot": "1",
          "type": "t_array(t_uint256)2_storage"
        },
        {
          "astId": 12,
          "contract": "fileA:A",
          "label": "dynArray",
          "offset": 0,
          "slot": "3",
          "type": "t_array(t_uint256)dyn_storage"
        }
      ],
      "numberOfBytes": "128"
    },
    "t_uint128": {
      "encoding": "inplace",
      "label": "uint128",
      "numberOfBytes": "16"
    },
```


(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    "t_uint256": {
      "encoding": "inplace",
      "label": "uint256",
      "numberOfBytes": "32"
    }
  }
}
```

3.16 Bellekteki Düzen

Solidity, belirli bayt aralıklarını (uç noktalar dahil) aşağıdaki şekilde kullanılmak üzere dört adet 32 baytlık yuva ayırır:

- `0x00` - `0x3f` (64 bytes): Hash metotları için scratch(kazıma) alanı
- `0x40` - `0x5f` (32 bytes): Şuan ayrılmış olan bellek boyutu (boş bellek pointer'ı olarak da bilinir)
- `0x60` - `0x7f` (32 bytes): zero slot (sıfır yuva)

Durumlar arasında (yani assembly içinde) scratch alanı kullanılabilir. Sıfır yuvası, dinamik bellek dizilerinin başlangıç değeri olarak kullanılır ve asla başlangıçta `0x80`'i gösteren boş bellek pointer noktasına yazılmamalıdır.

Solidity her zaman yeni nesneleri boş bellek işaretçisine yerleştirir ve hafıza asla serbest bırakılmaz (Bu özellik gelecekte değişebilir).

Solidity'de bulunan bellek dizilerindeki öğeler her zaman 32 baytın katlarını kaplar (Bu `bytes1[]` için bile doğrudur, ama `bytes` ve `string` için geçerli değildir). Çok boyutlu bellek dizileri, bellek dizilerinin işaretçileridir. Bir dinamik dizinin uzunluğu dizinin ilk yuvasında depolanır ve ardından dizinin elemanları gelir.

Uyarı: Solidity'de 64 bayttan daha büyük bir geçici bellek alanına ihtiyaç duyan ve bu nedenle scratch alanına sığmayan bazı işlemler vardır. Bu işlemler boş bellek noktalarına yerleştirilecektir, ama kısa ömürleri nedeniyle işaretçi güncellenemez. Bellek sıfırlanmış olabilir ya da olmayabilir. Bu nedenle, boş hafızanın sıfırlanmış hafızayı göstermesi beklenmemelidir.

Net bir sıfırlanmış bellek alanına ulaşmak için `msize` kullanmak iyi bir fikir gibi görünse de, boş bellek işaretçisi güncellenmeden geçici olmayan bir işaretçi kullanmak beklenmeyen sonuçlara neden olabilir.

3.16.1 Depolama Düzeni Farklılıkları

Yukarıda açıklandığı üzere bellekteki düzen ile depolama düzeni (*storage*) farklıdır. Aşağıda bunlara yönelik bazı örnekler bulunmaktadır.

Dizilerdeki Farklılıklara Bir Örnek

Aşağıdaki dizi, depolamada 32 bayt (1 yuva) yer kaplar, ancak bellekte 128 bayt (her biri 32 bayt olan 4 öğe) yer kaplar.

```
uint8[4] a;
```

Yapı(Struct) Düzeni Farklılıklarına Bir Örnek

Aşağıdaki struct, depolamada 96 bayt (32 baytlık 3 yuva) kaplar, ama bellekte 128 bayt (her biri 32 bayt olan 4 öğe) yer kaplar.

```
struct S {
    uint a;
    uint b;
    uint8 c;
    uint8 d;
}
```

3.17 Çağrı Verilerinin Düzeni

Bir fonksiyon çağrısı için alınan girdi verisinin *ABI belirtimi* tarafından tanımlanan formatta olduğu varsayılır. Diğerlerinin yanı sıra, ABI tanımlaması argümanların 32 baytın katları olacak şekilde eklenmesini zorunlu kılar. Dahili(internal) fonksiyon çağrıları bundan farklı bir kural kullanır.

Bir sözleşmenin constructor fonksiyonu için argümanlar, ABI şifrelemesinde de olduğu gibi sözleşmenin kodunun sonuna doğrudan eklenir. Constructor fonksiyonu argümanlara `codesize` işlem kodunu kullanarak değil, sabit kodlanmış bir ofset üzerinden erişir. Bunun nedeni ise koda veri eklerken bu ofsetin değişmesidir.

3.18 Değişkenlerin Temizlenmesi

Bir değer 256 bitten daha kısa olduğunda, bazı durumlarda kalan bitlerin temizlenmesi gerekir. Solidity derleyicisi, kalan bitlerdeki potansiyel çöplerden olumsuz etkilenebilecek herhangi bir işlemden önce bu tür kalan bitleri temizlemek üzere tasarlanmıştır. Örnek vermek gerekirse, belleğe bir değer yazmadan öncede kalan bitlerin temizlenmesi gerekir çünkü bellek içeriği hash değerlerinin hesaplanması için kullanılabilir veya bir mesaj çağrısının verisi olarak gönderilebilir. Benzer şekilde, bir değeri depolamadan öncede aynı durum geçerlidir çünkü aksi takdirde bozuk değer gözlemlenebilir.

Satır içi(inline) assembly yoluyla erişimin böyle bir işlem olarak kabul edilmediğini unutmayın: Eğer 256 bitten kısa Solidity değişkenlerine erişmek için satır içi (inline) assembly kullanırsanız, derleyici değerin düzgün bir şekilde temizlendiğini garanti etmez.

Dahası, hemen ardından gelen işlem tarafından etkilenmiyorsa bitleri temizlemeyiz. Örneğin, sıfır olmayan herhangi bir değer `JUMPI` komutu tarafından `true` olarak kabul edildiğinden, boolean değerlerini `JUMPI` için koşul olarak kullanılmadan önce temizlemiyoruz.

Yukarıdaki tasarım prensibine ek olarak, Solidity derleyicisi girdi verilerini yığına(stack) yüklendiğinde temizler.

Farklı türlerin geçersiz değerleri temizlemek için farklı kuralları vardır:

Tür	Geçerli Değerler	Geçersiz Değerlerin Anlamları
n üyeli bir enum	0'dan n - 1'e kadar	istisna
bool	0 ya da 1	1
işaretli tam sayılar	işareti uzatılmış kelime	sessizce doğru formata getirir; gelecekte istisnala r atılacaktır
işaretsiz tam sayılar	daha yüksek bit değerleri sıfırlandı	sessizce doğru formata getirir; gelecekte istisnala r atılacaktır

3.19 Kaynak Eşlemesi

AST çıktısının bir parçası olarak derleyici, AST'deki ilgili node tarafından temsil edilen kaynak kod aralığını sağlar. Bu durum AST'ye dayalı olarak rapor veren statik analiz araçlarından, yerel değişkenleri ve kullanımlarını vurgulayan hata ayıklama araçlarına kadar çeşitli amaçlar için kullanılabilir.

Ayrıca derleyici, bytecode'dan komutu oluşturan kaynak koddaki aralığa kadar bir mapping de oluşturabilir. Bu durum bytecode seviyesinde çalışan statik analiz araçları ve bir hata ayıklayıcı için kaynak koddaki mevcut konumu görüntülemek veya breakpoint işleme açısından önemlidir. Bu mapping aynı zamanda sıçrama komutu türü ve modifier derinliği gibi diğer bilgileri de içerir (aşağıya bakınız).

Her iki tür kaynak mapping'i de kaynak dosyalara başvurmak için tamsayı tanımlayıcıları kullanır. Bir kaynak dosyasının tanımlayıcısı `output['sources'][sourceName]['id']` içinde saklanır, buradaki `output` aynı zamanda JSON olarak ayrıştırılmış standart-json derleyici arayüzünün çıktısıdır. Bazı yardımcı program rutinleri için derleyici, orijinal girdinin bir parçası olmayan ancak kaynak mappinglerinden referans alınan "internal" kaynak dosyaları üretir. Bu kaynak dosyalar tanımlayıcılarıyla birlikte `output['contracts'][sourceName][contractName]['evm']['bytecode']['generatedSources']` aracılığıyla elde edilebilir.

Not: Belirli bir kaynak dosyasıyla ilişkilendirilmemiş talimatlar söz konusu olduğunda, kaynak mapping `-1` değerinde bir tamsayı tanımlayıcı atar. Bu, derleyici tarafından oluşturulan satır içi assembly komutlarından kaynaklanan bytecode bölümleri için söz konusu olabilir.

AST içindeki kaynak mapping'leri aşağıdaki gösterimi kullanır:

`s:l:f`

Burada `s` kaynak dosyadaki aralığın başlangıcındaki bayt ofseti, `l` kaynak aralığının bayt cinsinden uzunluğu ve `f` yukarıda belirtilen kaynak indeksidir.

Bayt kodu için kaynak mapping'deki kodlama daha karmaşıktır: Bu, `;` ile ayrılmış `s:l:f:j:m` listesidir. Bu öğelerin her biri bir komuta karşılık gelir, yani bayt ofsetini kullanamazsınız, ancak komut ofsetini kullanmanız gerekir (push komutları tek bir bayttan daha uzundur). `s`, `l` ve `f` alanları yukarıdaki gibidir. `j` alanı `i`, `o` ya da `-` olabilir, bu da bir atlama(jump) talimatının bir fonksiyona mı girdiğini, bir fonksiyondan mı döndüğünü ya da örneğin bir döngünün parçası olarak normal bir sıçrama komutu türü mü olduğunu gösterir. Son kısım olan `m`, " modifier derinliğini" ifade eden bir tamsayıdır. Bu derinlik, placeholder(yer tutucu) ifade (_) bir modifier'a her girildiğinde artırılır ve tekrar bırakıldığında azaltılır. Bu, hata ayıklayıcıların aynı modifier'ın iki kez kullanılması veya tek bir modifier'da birden fazla placeholder(yer tutucu) ifadenin kullanılması gibi zor durumları takip etmesini sağlar.

Özellikle bytecode için bu kaynak mapping'lerini sıkıştırmak amacıyla aşağıdaki kurallar uygulanır:

- Eğer bir alan boşsa, bir önceki elemanın değeri kullanılır.
- Eğer bir `:` eksikse, takip eden tüm alanlar boş kabul edilir.

Bu, aşağıdaki kaynak mapping'lerinin aynı bilgiyi temsil ettiği anlamına gelir:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2:1:2;;`

Dikkat edilmesi gereken önemli bir nokta, *verbatim* yerleşik öğesi kullanıldığında, kaynak mapping'lerinin geçersiz olacağıdır: Yerleşik komut, potansiyel olarak birden fazla komut yerine tek bir komut olarak kabul edilir.

3.20 Optimize Edici

Solidity derleyicisi iki farklı optimize edici modül kullanır: İşlem kodu düzeyinde çalışan "eski" iyileştirici ve Yul IR kodunda çalışan "yeni" iyileştirici.

İşlem kodu tabanlı optimize edici, işlem kodlarına bir dizi basitleştirme kuralı uygular. Ayrıca eşit kod kümelerini birleştirir ve kullanılmayan kodu kaldırır.

Yul tabanlı optimize edici, fonksiyon çağrıları arasında çalışabildiği için çok daha güçlüdür. Örneğin, Yul'da arbitrary jumps yapmak mümkün değildir, bu nedenle her bir fonksiyonun yan etkilerini hesaplamak mümkündür. İlkinin depolamayı değiştirmediği ve ikincisinin depolamayı değiştirdiği iki fonksiyon çağrısını düşünün. Argümanları ve dönüş değerleri birbirine bağlı değilse, fonksiyon çağrılarını yeniden sıralayabiliriz. Benzer şekilde, bir fonksiyon yan etkiden arındırılmışsa ve sonucu sıfırla çarpılırsa, fonksiyon çağrısını tamamen kaldırabilirsiniz.

Şu anda, "–optimize" parametresi, oluşturulan bayt kodu için işlem kodu tabanlı iyileştiriciyi ve dahili olarak Yul kodu için oluşturulan Yul iyileştiriciyi, örneğin ABI kodlayıcı v2'yi etkinleştirir. Bir Solidity kaynağına özel olarak optimize edilmiş bir Yul IR üretmek için `solc --ir-optimized --optimize` kullanılabilir. Benzer şekilde, bağımsız bir Yul modu için `solc --strict-assembly --optimize` kullanılabilir.

Aşağıda hem optimize edici modüller hem de optimizasyon adımları hakkında daha fazla ayrıntı bulabilirsiniz.

3.20.1 Solidity Kodunu Optimize Etmenin Faydaları

Genel olarak optimize ediciler, karmaşık ifadeleri sadeleştirmeye çalışır, bu da hem kod boyutunu hem de çalıştırma(execution) maliyetini azaltır, yani sözleşmenin devreye alınmasını ve sözleşmeye yapılan harici çağrılar için gereken gas miktarını azaltabilir. Ayrıca, fonksiyonları uzmanlaştırır veya sıralar. Özellikle satır içi fonksiyonları oluşturma, çok daha büyük kodlara neden olabilecek bir işlemdir, ancak daha fazla sadeleştirme fırsatlarına yol açtığı için sıklıkla yapılır.

3.20.2 Optimize Edilmiş ve Optimize Edilmemiş Kod Arasındaki Farklar

Genel olarak ikisi arasındaki en görünür fark, sabit ifadelerin derleme zamanındaki farklılıklardır. ASM çıktısı söz konusu olduğunda, eşdeğer veya yinelenen kod bloklarındaki gas miktarında azalma da fark edilebilir (`--asm` ve `--asm --optimize` işaretlerinin çıktısını karşılaştırın). Bununla birlikte, Yul/intermediate-representation söz konusu olduğunda, önemli farklılıklar olabilir, örneğin, fonksiyonlar satır içine alınabilir, birleştirilebilir veya fazlalıkları ortadan kaldırmak için yeniden yazılabilir, vb. (çıktıyı `--ir` ve `--optimize --ir-optimized` işaretleri ile birlikte karşılaştırabilirsiniz).

3.20.3 Optimize Edici Parametre Çalıştırmaları

Çalıştırma sayısı ("–optimize-runs"), dağıtılan kodun her bir işlem kodunun sözleşmenin ömrü boyunca yaklaşık olarak ne sıklıkta yürütüleceğini belirtir. Bu, kod boyutu (dağıtım maliyeti) ve kod yürütme maliyeti (dağıtımdan sonraki maliyet) arasında bir değiş tokuş parametresi olduğu anlamına gelir. "1" "runs" parametresi kısa ama pahalı olan bir kod üretecektir. Buna karşılık, daha büyük bir "runs" parametresi daha uzun ancak daha fazla gaz verimli kod üretecektir. Parametrenin maksimum değeri `2**32-1` dir.

Not: Yaygın bir yanlış anlama ise bu parametrenin optimize edicinin yineleme sayısını belirtmesidir. Ancak bu doğru değildir: Optimize edici her zaman kodu iyileştirebildiği kadar çalışır.

3.20.4 Opcode Tabanlı Optimize Edici Modülü

Opcode tabanlı optimize edici modül, assembly kodu üzerinde çalışır. Komut dizisini "JUMPs" ve "JUMPDESTs"de temel bloklara böler. Bu blokların içinde, optimize edici talimatları analiz eder ve yığında, bellekte veya depolamada yapılan her değişikliği, bir talimattan ve diğer ifadelere işaret eden bir argüman listesinden oluşan bir ifade olarak kaydeder.

Ek olarak, işlem kodu tabanlı optimize edici, diğer görevlerin yanı sıra (her girişte) her zaman eşit olan ifadeleri bulan ve bunları bir ifade sınıfında birleştiren "CommonSubexpressionEliminator" adlı bir bileşen kullanır. İlk önce her yeni ifadeyi önceden bilinen ifadeler listesinde bulmaya çalışır. Böyle bir eşleşme bulunamazsa, ifadeyi `constant + constant = sum_of_constants` veya `X * 1 = X` gibi kurallara göre sadeleştirir. Bu recursive(öz yinelemeli) bir süreç olduğundan, ikinci faktör her zaman bir olarak değerlendirdiğini bildiğimiz daha karmaşık bir ifadeyse, ikinci kuralı da uygulayabiliriz.

Belirli optimize edici adımları, depolama ve bellek konumlarını sembolik olarak izler. Örneğin bu bilgi, derleme sırasında değerlendirilebilecek Keccak-256 hashlerini hesaplamak için kullanılır. Bu sıralamayı düşünebilirsiniz:

```
PUSH 32
PUSH 0
CALLDATALOAD
PUSH 100
DUP2
MSTORE
KECCAK256
```

veya eşdeğeri Yul

```
let x := calldataload(0)
mstore(x, 100)
let value := keccak256(x, 32)
```

Bu durumda, optimize edici `calldataload(0)` bellek konumundaki değeri izler ve ardından Keccak-256 hash değerinin derleme zamanında değerlendirilebileceğini anlar. Bu, yalnızca `mstore` ve `keccak256` arasındaki belleği değiştiren başka bir komut yoksa çalışır. Yani belleğe (veya depolamaya) bilgi yazan bir talimat varsa, o zaman mevcut bilginin bellek (veya depolama) bilgisini silmemiz gerekir. Ancak, talimatın belirli bir yere yazmadığını kolayca görebildiğimizde, bu silme işleminin bir istisnası vardır.

Örneğin,

```
let x := calldataload(0)
mstore(x, 100)
// Mevcut bilgi hafıza konumu x -> 100
let y := add(x, 32)
// y'nin [x, x + 32)'ye bilgi yazmaması nedeniyle x -> 100 olduğu bilgisi silinmez
mstore(y, 200)
// Bu Keccak-256 artık değerlendirilebilir
let value := keccak256(x, 32)
```

Bu nedenle, depolama ve bellek konumlarında, örneğin `l` konumunda yapılan değişiklikler, `l``ye eşit olabilecek depolama veya bellek konumları hakkındaki bilgileri silmelidir. Daha spesifik olarak, depolama için, optimize edicinin ``l``ye eşit olabilecek tüm sembolik konum bilgilerini silmesi gerekir ve bellek için optimize edicinin en az 32 bayt uzakta olmayabilecek tüm sembolik konum bilgilerini silmesi gerekir. . Eğer ``m` arbitarry lokasyonu gösteriyorsa, o zaman bu silme kararı `sub(l, m)` değeri hesaplanarak yapılır. Depolama için, bu değer sıfırdan farklı bir hazır bilgi olarak değerlendirilirse, o zaman `m` ile ilgili bilgi tutulacaktır. Bellek için, değer `32` ile `2**256 - 32` arasında bir değer olarak değerlendirilirse, `m` ile ilgili bilgi korunur. Diğer tüm durumlarda, `m` hakkındaki bilgiler silinecektir.

Bu işlemden sonra, sonunda yığında(stack) hangi ifadelerin olması gerektiğini biliyoruz ve bellek ve depolamada yapılan değişikliklerin bir listesine sahibiz. Bu bilgi, temel bloklarla birlikte saklanır ve bunları birbirine bağlamak için kullanılır. Ayrıca yığın, depolama ve bellek yapılandırması hakkındaki bilgiler sonraki bloğa/bloklara iletilir.

Tüm `JUMP` ve `JUMPI` komutlarının hedeflerini biliyorsak, programın tam bir kontrol akış grafiğini oluşturabiliriz. Bilmediğimiz tek bir hedef varsa (bu prensipte olduğu gibi olabilir, jump targets girdilerden hesaplanabilir), bilinmeyen `JUMP` değerinin hedefi olabileceğinden bir bloğun girdi durumu hakkındaki tüm bilgileri silmemiz gerekir. İşlem kodu tabanlı optimize edici modül, koşulu bir sabite göre değerlendirilen bir `JUMPI` bulursa, bunu koşulsuz bir jump`a dönüştürür.

Son adım olarak, her bloktaki kod yeniden oluşturulur. Optimize edici, bloğun sonunda bulunan yığındaki ifadelerden bir bağımlılık grafiği oluşturur ve bu grafiğin parçası olmayan her işlemi bırakır. Değişiklikleri orijinal kodda yapıldıkları sırayla belleğe(memory) ve depolamaya(storage) uygulayan kod üretir (gerekli olmadığı tespit edilen değişiklikleri bırakarak). Son olarak yığında olması gereken tüm değerleri doğru yerde üretir.

Bu adımlar her temel bloğa uygulanır ve yeni oluşturulan kod daha küçükse yedek olarak kullanılır. Temel bir blok bir `JUMPI`'de bölünürse ve analiz sırasında koşul bir sabit olarak değerlendirilirse, `JUMPI` sabitin değerine göre değiştirilir. Aşağıda bulunan kodda olduğu gibi

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2) // bu koşul asla doğru değildir
  return 2;
else
  return 1;
```

bunu sadeleştirir:

```
data[7] = 9;
return 1;
```

Basit Inlining

Solidity 0.8.2 sürümünden bu yana, "jump" ile biten "simple" talimatları içeren bloklara yapılan belirli atlamaları bu talimatların bir kopyası ile değiştiren başka bir optimizer adımı bulunmaktadır. Bu, basit, küçük Solidity veya Yul fonksiyonlarının inlining'ine karşılık gelir. Özellikle, `PUSHTAG(tag) JUMP` dizisi, `JUMP` bir fonksiyona atlama olarak işaretlendiğinde ve `tag` arkasında bir fonksiyondan "dışarı" atlama olarak işaretlenen başka bir `JUMP` ile biten temel bir blok ("CommonSubexpressionEliminator" için yukarıda açıklandığı gibi) olduğunda değiştirilebilir.

Özellikle, dahili bir Solidity fonksiyonuna yapılan bir çağrı için oluşturulan aşağıdaki prototip assembly örneğini göz önünde bulundurun:

```
  tag_return
  tag_f
  jump      // içeri
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // dışarı
```

Fonksiyonun gövdesi sürekli bir temel blok olduğu sürece, "Inliner" `tag_f jump` yerine `tag_f` adresindeki blokla değiştirebilir ve sonuç olarak:

```
  tag_return
  ...body of function f...
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

Şimdi ideal olarak, yukarıda açıklanan diğer optimize edici adımlar, return etiketi push'unun kalan jump'a doğru hareket ettirilmesiyle sonuçlanacaktır:

```
  ...body of function f...
  tag_return
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // out
```

Bu durumda "PeepholeOptimizer" return jump'ı kaldıracaktır. İdeal olarak, tüm bunlar `tag_f`'ye yapılan tüm referanslar için yapılabilir, kullanılmadan bırakılabilir, s.t. kaldırılabilir, sonuç verir:

```
...body of function f...
...opcodes after call to f...
```

Böylece `f` fonksiyonuna yapılan çağrı satır içine alınır ve `f` fonksiyonunun orijinal tanımı kaldırılabilir.

Bir buluşsal yöntem, bir sözleşmenin ömrü boyunca inlining yapmanın inlining yapmamaktan daha ucuz olduğunu gösterdiğinde, bu durumdaki inlining denenir. Bu sezgisel yöntem, fonksiyon gövdesinin boyutuna, etiketine yapılan diğer referansların sayısına (fonksiyona yapılan çağrıların sayısına yaklaşık olarak) ve sözleşmenin beklenen yürütme sayısına (global optimizer parametresi "runs") bağlıdır.

3.20.5 Yul Tabanlı Optimize Edici Modülü

Yul tabanlı optimize edici, tümü AST'yi anlamsal olarak eşdeğer bir şekilde dönüştüren birkaç aşamadan ve bileşenden oluşur. Amaç, ya daha kısa ya da en azından marjinal olarak daha uzun olan ancak daha fazla optimizasyon adımına izin verecek bir kodla sonuçlandırmaktır.

Uyarı: Optimize edici yoğun bir geliştirme aşamasında olduğundan, buradaki bilgiler güncel olmayabilir. Belirli bir fonksiyonelliğe güveniyorsanız, lütfen doğrudan ekiple iletişime geçin.

Optimize edici şu anda tamamen greedy(metinsel olarak mümkün olduğunca fazla eşleşen) bir strateji izliyor ve herhangi bir geri izleme yapmıyor.

Yul tabanlı optimizer modülünün tüm bileşenleri aşağıda açıklanmıştır. Aşağıdaki dönüşüm adımları ana bileşenlerdir:

- SSA Transform
- Common Subexpression Eliminator

- Expression Simplifier
- Redundant Assign Eliminator
- Full Inliner

Optimize Edici Adımları

Bu, Yul tabanlı optimize edicinin alfabetik olarak sıralanmış tüm adımlarının bir listesidir. Her bir adım ve bunların sıralaması hakkında daha fazla bilgiyi aşağıda bulabilirsiniz.

- *BlockFlattener*.
- *CircularReferencesPruner*.
- *CommonSubexpressionEliminator*.
- *ConditionalSimplifier*.
- *ConditionalUnsimplifier*.
- *ControlFlowSimplifier*.
- *DeadCodeEliminator*.
- *EqualStoreEliminator*.
- *EquivalentFunctionCombiner*.
- *ExpressionJoiner*.
- *İfade Basitleştirici (Expression Simplifier)*.
- *ExpressionSplitter*.
- *ForLoopConditionIntoBody*.
- *ForLoopConditionOutOfBody*.
- *ForLoopInitRewriter*.
- *ExpressionInliner*.
- *FullInliner*.
- *FunctionGrouper*.
- *FunctionHoister*.
- *FunctionSpecializer*.
- *LiteralRematerialiser*.
- *LoadResolver*.
- *LoopInvariantCodeMotion*.
- *RedundantAssignEliminator*.
- *ReasoningBasedSimplifier*.
- *Rematerialiser*.
- *SSAReverser*.
- *SSATransform*.
- *StructuralSimplifier*.

- *UnusedFunctionParameterPruner*.
- *UnusedPruner*.
- *VarDeclInitializer*.

Optimizasyonları Seçme

Varsayılan olarak optimizer, oluşturulan assembly'ye önceden tanımlanmış optimizasyon adımları dizisini uygular. Bu diziyi geçersiz kılabilir ve `--yul-optimizations` seçeneğini kullanarak kendi dizinizi sağlayabilirsiniz:

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul'
```

`[...]` içinde yer alan dizi, Yul kodu değişmeden kalana kadar veya maksimum tur sayısına (şu anda 12) ulaşılana kadar bir döngü içinde birden çok kez uygulanacaktır.

Mevcut kısaltmalar *Yul optimize edici dokümanları* içinde listelenmiştir.

Ön İşleme (Preprocessing)

Ön işleme bileşenleri, programı üzerinde çalışılması daha kolay olan belirli normal bir forma sokmak için gerekli dönüşümleri gerçekleştirir. Bu normal formu optimizasyon sürecinin geri kalan bölümü boyunca muhafaza eder.

Disambiguator

Anlam ayrıştırıcı bir AST alır ve tüm tanımlayıcıların girdi AST'sinde benzersiz adlara sahip olduğu yeni bir kopya döndürür. Bu, diğer tüm optimize edici aşamalar için bir ön koşuldur. Avantajlarından biri, tanımlayıcının aranmanın kapsamları dikkate almasına gerek kalmamasıdır, bu da diğer adımlar için gereken analizi basitleştirir.

Sonraki tüm aşamalar, tüm isimlerin benzersiz kalması özelliğine sahiptir. Bu, herhangi bir yeni tanımlayıcı eklenmesi gerektiğinde yeni bir benzersiz isim üretileceği anlamına gelir.

FunctionHoister

Fonksiyon hoister, tüm fonksiyon tanımlarını en üstte bulunan bloğun sonuna taşır. Belirsizliği giderme aşamasından sonra gerçekleştirildiği sürece bu anlamsal olarak eşdeğer bir dönüşümdür. Bunun nedeni, bir tanımın daha yüksek seviyeli bir bloğa taşınmasının görünürlüğünü azaltamaması ve farklı bir fonksiyonda tanımlanan değişkenlere başvurmanın imkansız olmasıdır.

Bu aşamanın faydası, fonksiyon tanımlarının daha kolay aranabilmesi ve fonksiyonların, AST'yi tamamen geçmek zorunda kalmadan izole bir şekilde optimize edilebilmesidir.

FunctionGrouper

Fonksiyon grouper, Disambiguator ve FunctionHoister sonra uygulanmalıdır. Etkisi, işlev tanımları olmayan en üstteki tüm öğelerin, kök bloğun ilk ifadesi olan tek bir bloğa taşınmasıdır.

Bu adımdan sonra, bir program aşağıdaki normal forma sahiptir:

```
{ I F... }
```

Burada `I` herhangi bir fonksiyon tanımı içermeyen (rekürsif olarak bile) (potansiyel olarak boş) bir bloktur ve `F` hiçbir fonksiyonun bir fonksiyon tanımı içermediği bir fonksiyon tanımları listesidir.

Bu aşamanın faydası, fonksiyon listesinin nerede başladığını her zaman bilmemize olanak sağlamasıdır.

ForLoopConditionIntoBody

Bu dönüşüm, bir for döngüsünün döngü yineleme koşulunu döngü gövdesine taşır. Bu dönüşüme ihtiyacımız var çünkü *ExpressionSplitter* yineleme koşulu ifadelerine (aşağıdaki örnekte `C`) uygulanmayacaktır.

```
for { Init... } C { Post... } {
    Body...
}
```

dönüştürülür

```
for { Init... } 1 { Post... } {
    if iszero(C) { break }
    Body...
}
```

Bu dönüşüm aynı zamanda `LoopInvariantCodeMotion` ile eşleştirildiğinde de faydalı olabilir, çünkü döngüde değişmez koşullardaki invariant'lar daha sonra döngünün dışına alınabilir.

ForLoopInitRewriter

Bu dönüşüm, bir for-döngüsünün başlatma kısmını döngüden önceki kısmına taşır:

```
for { Init... } C { Post... } {
    Body...
}
```

dönüştürülür

```
Init...
for {} C { Post... } {
    Body...
}
```

Bu, döngü başlatma(genesis) bloğunun karmaşık kapsam belirleme kurallarını göz ardı edebileceğimiz için optimizasyon sürecinin geri kalanını kolaylaştırır.

VarDeclInitializer

Bu adım, değişken tanımlamalarını yeniden yazarak hepsinin başlatılmasını sağlar. `let x, y` gibi tanımlamalar birden fazla tanımlama (multiple declaration) ifadesine bölünür.

Şimdilik yalnızca sıfır literali ile başlatmayı destekliyor.

Pseudo-SSA Dönüşümü

Bu bileşenlerin amacı programı daha uzun bir forma sokmaktır, böylece diğer bileşenler onunla daha kolay çalışabilir. Final gösterimi statik-tek-atama (SSA) formuna benzer olacaktır, tek farkı kontrol akışının farklı kollarından(branch) gelen değerleri birleştiren açık "phi" fonksiyonlarını kullanmamasıdır çünkü böyle bir özellik Yul dilinde mevcut değildir. Bunun yerine, kontrol akışı birleştiğinde, kollardan(branch) birinde bir değişken yeniden atanırsa, mevcut değerini tutmak için yeni bir SSA değişkeni bildirilir, böylece aşağıdaki ifadelerin hala yalnızca SSA değişkenlerine başvurması gerekir.

Örnek bir dönüşüm aşağıda verilmiştir:

```
{
    let a := calldataload(0)
    let b := calldataload(0x20)
    if gt(a, 0) {
        b := mul(b, 0x20)
    }
    a := add(a, 1)
    sstore(a, add(b, 0x20))
}
```

Aşağıdaki tüm dönüşüm adımları uygulandığında, program aşağıdaki gibi görünecektir:

```
{
    let _1 := 0
    let a_9 := calldataload(_1)
    let a := a_9
    let _2 := 0x20
    let b_10 := calldataload(_2)
    let b := b_10
    let _3 := 0
    let _4 := gt(a_9, _3)
    if _4
    {
        let _5 := 0x20
        let b_11 := mul(b_10, _5)
        b := b_11
    }
    let b_12 := b
    let _6 := 1
    let a_13 := add(a_9, _6)
    let _7 := 0x20
    let _8 := add(b_12, _7)
    sstore(a_13, _8)
}
```

Bu kod parçasında yeniden atanan tek değişkenin `b` olduğuna dikkat edin. Bu yeniden atama işleminden kaçınılamaz çünkü `b` kontrol akışına bağlı olarak farklı değerlere sahiptir. Diğer tüm değişkenler tanımlandıktan sonra değerlerini asla değiştirmezler. Bu özelliğin avantajı, bu değerler yeni bağlamda hala geçerli olduğu sürece, değişkenlerin serbestçe hareket ettirilebilmesi ve bunlara yapılan referansların ilk değerleriyle (ve tersiyle) değiştirilebilmesidir.

Elbette, buradaki kod optimize edilmekten oldukça uzaktır. Aksine, çok daha uzundur. Buradaki beklentimiz, bu kodla çalışmanın daha kolay olacağı ve ayrıca, bu değişiklikleri geri alan ve sonunda kodu tekrar daha kompakt hale getiren optimize edici adımların var olmasıdır.

ExpressionSplitter

Expression splitter(İfade Ayırıcı), `add(mload(0x123), mul(mload(0x456), 0x20))` gibi ifadeleri, ilgili ifadenin alt ifadelerine atanan benzersiz değişkenleri bildiren bir diziye dönüştürür, böylece her fonksiyon çağrısında argüman olarak yalnızca değişkenler bulunur.

Yukarıdakiler şu şekle dönüştürülebilir:

```
{
    let _1 := 0x20
    let _2 := 0x456
    let _3 := mload(_2)
    let _4 := mul(_3, _1)
    let _5 := 0x123
    let _6 := mload(_5)
    let z := add(_6, _4)
}
```

Bu dönüşümün işlem kodlarının veya fonksiyon çağrılarının sırasını değiştirmediğini unutmayın.

Bu özellik döngü yineleme koşuluna(loop iteration-condition) uygulanmaz, çünkü döngü kontrol akışı her durumda iç ifadelerin(inner expressions) bu şekilde "outlining" yapılmasına izin vermez. Yineleme koşulunu döngü gövdesine taşımak için *ForLoopConditionIntoBody* uygulayarak bu sınırlamayı ortadan kaldırabiliriz.

Final programı öyle bir formda olmalıdır ki fonksiyon çağrıları (döngü koşulları hariç) ifadelerin içinde içiçe görünmemeli ve tüm fonksiyon çağrısı argümanları değişken olmalıdır.

Bu formun faydaları, işlem kodları dizisini yeniden sıralamanın çok daha kolay olması ve ayrıca fonksiyon çağrısı inlining'i yapmanın daha kolay hale getirmesidir. Ayrıca, ifadelerin tek tek parçalarını değiştirmek veya "expression tree"'yi yeniden düzenlemek daha kolaydır. Dezavantajı ise bu tür kodların insanlar tarafından okunmasının çok daha zor olmasıdır.

SSATransform

Bu aşama, mevcut değişkenlere tekrarlanan atamaları mümkün olduğunca yeni değişkenlerin tanımlamalarıyla değiştirmeye çalışır. Yeniden atamalar hala mevcuttur, ancak yeniden atanan değişkenlere yapılan tüm referanslar yeni bildirilen değişkenlerle değiştirilir.

Örnek:

```
{
    let a := 1
    mstore(a, 2)
    a := 3
}
```

dönüştürülür

```
{
    let a_1 := 1
    let a := a_1
    mstore(a_1, 2)
    let a_3 := 3
    a := a_3
}
```

Tam Semantik:

Kodda herhangi bir yere atanan bir a değişkeni için (değerle tanımlanan ve asla yeniden atanmayan değişkenler değiştirilmemektedir) aşağıdaki dönüşümleri gerçekleştirin:

- `let a := v` yerine `let a_i := v let a := a_i` yazın
- `a := v` yerine `let a_i := v a := a_i` yazın; buradaki `i`, `a_i` henüz kullanılmamış türde bir sayıdır.

Ayrıca, a için kullanılan `i` geçerli değerini her zaman saklamalı ve a değişkenine yapılan her referansı `a_i` ile değiştirmelisiniz. Bir a değişkeni için geçerli olan bir değer eşlemesi, atandığı her bloğun sonunda ve for döngü gövdesi veya post bloğu içinde atanmışsa for döngüsü init(başlangıç) bloğunun sonunda temizlenir. Bir değişkenin değeri yukarıdaki kurala göre temizlenirse ve değişken blok dışında bildirilirse, kontrol akışının birleştiği yerde yeni bir SSA değişkeni oluşturulur, buna döngü sonrası/gövde bloğunun başlangıcı ve If/Switch/ForLoop/Block ifadesinden hemen sonra gelen konum dahildir.

Bu aşamadan sonra, gereksiz ara atamaları kaldırmak için Redundant Assign Eliminator kullanılması önerilir.

Bu aşama, Expression Splitter (İfade Ayırıcı) ve Common Subexpression Eliminator (Ortak Alt İfade Giderici) hemen öncesinde çalıştırılırsa en iyi sonuçları verir, çünkü o zaman aşırı miktarda değişken üretmez. Öte yandan, Common Subexpression Eliminator (Ortak Alt İfade Giderici) SSA dönüşümünden sonra çalıştırılırsa daha verimli olabilir.

RedundantAssignEliminator

SSA dönüşümü her zaman `a := a_i` şeklinde bir atama üretir, ancak bunlar aşağıdaki örnekte olduğu gibi birçok durumda gereksiz olabilir:

```
{
    let a := 1
    a := mload(a)
    a := sload(a)
    sstore(a, 1)
}
```

SSA dönüşümü bu parçacığı aşağıdaki parçacığa dönüştürür:

```
{
    let a_1 := 1
    let a := a_1
    let a_2 := mload(a_1)
    a := a_2
    let a_3 := sload(a_2)
    a := a_3
    sstore(a_3, 1)
}
```

Redundant Assign Eliminator, a değerinin kullanılmaması nedeniyle a değerine yapılan üç atamayı da kaldırır ve böylece bu parçacığı strict SSA formuna dönüştürür:

```
{
    let a_1 := 1
    let a_2 := mload(a_1)
    let a_3 := sload(a_2)
    sstore(a_3, 1)
}
```

Elbette, bir atamanın gereksiz olup olmadığını belirlemenin karmaşık kısımları, kontrol akışının birleştirilmesiyle bağlantılıdır.

Bileşen ayrıntılı olarak aşağıdaki gibi çalışır:

AST iki kez taranır: bilgi toplama adımında ve asıl kaldırma adımında. Bilgi toplama sırasında, atama ifadelerinden "unused", "undecided" ve "used" olmak üzere üç duruma yönelik bir eşleştirme tutarız, bu da atanan değerin daha sonra değişkene yapılan bir referans tarafından kullanılıp kullanılmayacağını gösterir.

Bir atama işlemi gerçekleştirildiğinde, "undecided" durumdaki eşleştirmeye eklenir (aşağıdaki for döngüleriyle ilgili açıklamaya bakın) ardından aynı değişkene yapılan ve hala "kararsız" durumda olan diğer tüm atamalar "undecided" olarak değiştirilir. Bir değişkene referans verildiği zaman, o değişkene yapılan ve hala "unused" durumda olan tüm atamaların durumu "undecided" olarak değiştirilir.

Kontrol akışının bölündüğü noktalarda, eşleştirmenin bir kopyası her bir kola(branch) aktarılır. Kontrol akışının birleştiği noktalarda, iki koldan gelen iki eşleme aşağıdaki şekilde birleştirilir: Ve ayrıca Yalnızca bir eşlemede bulunan veya aynı duruma sahip olan ifadeler değiştirilmeden kullanılır. Çakışan İfade değerleri de aşağıdaki şekilde çözümlenir:

- "unused", "undecided" -> "undecided"
- "unused", "used" -> "used"
- "undecided, "used" -> "used"

For-döngüleri açısından koşul, gövde ve son bölüm, koşulda birleşen kontrol akışı dikkate alınarak iki kez kontrol edilir. Başka bir ifadeyle, temel olarak üç kontrol akış yolu oluşturulur: Döngünün sıfır çalıştırılması, tek çalıştırılması ve ardından iki kez çalıştırılması ve sonunda birleştirilmesi.

Üçüncü bir çalıştırma ya da daha fazlasını simüle etmek gereksizdir, bu da şekilde olduğu biçimde anlaşılabilir:

Yinelemenin başlangıcındaki bir atama durumu, deterministik olarak yinelemenin sonunda o atamanın bir durumuyla sonuçlanacaktır. Bu durum eşleme fonksiyonu `f` olarak adlandırılsın. Yukarıda açıklandığı gibi `unused`, `undecided` ve `used` üç farklı durum kombinasyonu, `unused = 0`, `undecided = 1` ve `used = 2` olan `max` operasyondur.

Doğru yol döngüden

```
max(s, f(s), f(f(s)), f(f(f(s))), ...)
```

sonra hesaplamak olacaktır. `f` sadece üç farklı değer aralığına sahip olduğundan, iterasyon en fazla üç iterasyondan sonra bir döngüye ulaşmalıdır ve bu nedenle `f(f(f(s)))` `s`, `f(s)` veya `f(f(s))` değerlerinden birine eşit olmalıdır ve böylece

```
max(s, f(s), f(f(s))) = max(s, f(s), f(f(s)), f(f(f(s))), ...).
```

Özetle, döngüyü en fazla iki kez çalıştırmak yeterlidir çünkü sadece üç farklı durum vardır.

"Varsayılan" duruma sahip switch ifadeleri için switch'i atlayan bir kontrol akışı parçası yoktur.

Bir değişken kapsam dışına çıktığında, değişken bir fonksiyonun geri dönüş parametresi olmadığı sürece, hala "undecided" durumundaki tüm ifadeler "unused" olarak değiştirilir - bu durumda durum "used" olarak değişir.

İkinci çaprazlamada, "unused" durumunda olan tüm atamalar kaldırılır.

Bu adım genellikle SSA dönüşümünden hemen sonra çalıştırılarak pseudo-SSA'nın oluşturulması tamamlanır.

Araçlar

Taşınabilirlik(Movability)

Taşınabilirlik(Movability) bir ifadenin özelliğidir. Kabaca, ifadenin yan etkisiz olduğu ve değerlendirmesinin yalnızca değişkenlerin değerlerine ve ortamın çağrı sabit durumuna bağlı olduğu anlamına gelir. Çoğu ifade taşınabilirdir. Aşağıdaki parçalar bir ifadeyi taşınamaz yapar:

- fonksiyon çağrıları (eğer fonksiyondaki tüm ifadeler taşınabilirse gelecekte gevşetilebilir)
- yan etkileri olan (olabilen) işlem kodları (`call` veya `selfdestruct` gibi)
- bellek, depolama veya harici durum bilgilerini okuyan veya yazan işlem kodları
- geçerli PC'ye, bellek boyutuna veya geri dönen veri boyutuna bağlı olan işlem kodları

DataflowAnalyzer

Dataflow Analyzer kendi başına bir optimizer adımı değildir ancak diğer bileşenler tarafından bir araç olarak kullanılır. AST'de gezinirken, bu değer hareketli bir ifade olduğu sürece her değişkenin mevcut değerini izler. O anda her bir diğer değişkene atanmış olan ifadenin parçası olan değişkenleri kaydeder. Bir `a` değişkenine yapılan her atamada, `a` değişkeninin saklanan mevcut değeri güncellenir ve `a` değişkeni `b` için saklanan ifadenin bir parçası olduğunda `b` değişkeninin saklanan tüm değerleri silinir.

Kontrol akışı birleşimlerinde, değişkenler hakkındaki bilgiler, kontrol akışı yollarından herhangi birinde atanmışlarsa veya atanacaklarsa temizlenir. Örneğin, bir for döngüsüne girildiğinde, gövde veya son blok sırasında atanacak tüm değişkenler temizlenir.

İfade-Ölçekli Basitleştirmeler (Expression-Scale Simplifications)

Bu sadeleştirme geçişleri ifadeleri değiştirir ve onları eşdeğer ve muhtemelen daha basit ifadelerle değiştirir.

CommonSubexpressionEliminator

Bu adım Dataflow Analyzer'ı kullanır ve bir değişkenin mevcut değeriyle sözdizimsel olarak eşleşen alt ifadeleri o değişkene bir referans yoluyla değiştirir. Bu bir eşdeğerlik dönüşümüdür çünkü bu tür alt ifadelerin taşınabilir olması gerekir.

Kendileri tanımlayıcı olan tüm alt ifadeler, değer bir tanımlayıcıysa mevcut değerleriyle değiştirilir.

Yukarıdaki iki kuralın kombinasyonu, yerel değer numaralandırmasının hesaplanmasına izin verir; bu da iki değişken aynı değere sahipse, bunlardan birinin her zaman kullanılmayacağı anlamına gelir. Unused Pruner veya Redundant Assign Eliminator daha sonra bu tür değişkenleri tamamen ortadan kaldırabilecektir.

Bu adım özellikle ifade ayırıcı çalıştırıldığında etkilidir. Kod pseudo-SSA formundaysa, değişkenlerin değerleri daha uzun bir süre için mevcuttur ve bu nedenle ifadelerin değiştirilebilir olma şansı daha yüksektir.

İfade basitleştirici daha iyi değiştirmeler gerçekleştirebilecektir eğer ortak alt ifade giderici kendisinden hemen önce çalıştırılmışsa.

İfade Basitleştirici (Expression Simplifier)

İfade Basitleştirici, Dataflow Analyzer'ı kullanarak kodu basitleştirmek için `X + 0 -> X` gibi ifadeler üzerinde bir denklik dönüşümleri listesi kullanmaktadır.

Her alt ifadede `X + 0` gibi kalıpları eşleştirmeye çalışır. Eşleştirme prosedürü sırasında, kod pseudo-SSA formunda olsa bile daha derin iç içe geçmiş kalıpları eşleştirebilmek için değişkenleri o anda atanmış ifadelerine göre çözümler.

`X - X -> 0` gibi bazı kalıplar yalnızca `X` ifadesi taşınabilir olduğu sürece uygulanabilir, çünkü aksi takdirde potansiyel yan etkilerini ortadan kaldırır. Değişken referansları, mevcut değerleri olmasa bile her zaman taşınabilir olduğundan, İfade Basitleştirici bölünmüş veya pseudo-SSA formunda yine daha etkilidir.

LiteralRematerialiser

Belgelenmek üzere…

LoadResolver

Eğer biliniyorsa, `sload(x)` ve `mload(x)` tipindeki ifadeleri o anda bellekte depolanan değerle değiştiren optimizasyon aşamasıdır.

Kod SSA formundaysa en iyi şekilde çalışır.

Prerequisite: Disambiguator, ForLoopInitRewriter.

ReasoningBasedSimplifier

Bu optimizer, `if` koşullarının sabit olup olmadığını kontrol etmek için SMT çözücülerini kullanır.

- Eğer `constraints AND condition` UNSAT ise, koşul hiçbir zaman doğru değildir ve tüm gövde kaldırılabilir.
- Eğer `constraints AND NOT condition` UNSAT ise, koşul her zaman doğrudur ve `1` ile değiştirilebilir.

Yukarıdaki basitleştirmeler yalnızca koşulun hareketli olması durumunda uygulanabilir.

Yalnızca EVM diyalektinde etkilidir, ancak diğer diyalektlerde kullanımı güvenlidir.

Prerequisite: Disambiguator, SSATransform.

İfade Ölçeğindeki Basitleştirmeler (Statement-Scale Simplifications)

CircularReferencesPruner

Bu aşama, birbirini çağıran ancak dışarıdan veya en dış bağlamdan referans verilmeyen fonksiyonları kaldırır.

ConditionalSimplifier

Koşullu Basitleştirici(ConditionalSimplifier), değer kontrol akışından itibaren belirlenebiliyorsa koşul değişikliklerine atamalar ekler.

SSA formunu yok eder.

Şu anda, bu araç çok sınırlıdır, çünkü henüz boolean değişken türleri için desteğimiz yoktur. Koşullar yalnızca ifadelerin sıfırdan farklı olup olmadığını kontrol ettiğinden, belirli bir değer atayamayız.

Mevcut özellikler:

- switch cases: insert "<condition> := <caseLabel>"
- kontrol akışını sonlandıran if ifadesinden sonra "<condition> := 0" ekleyin

Future features:

- allow replacements by "1"
- take termination of user-defined functions into account

En iyi SSA formu ile ve ölü kod kaldırma işlemi daha önce çalıştırılmışsa çalışır.

Ön koşul: Anlam Ayrıştırıcı.

ConditionalUnsimplifier

Koşullu Basitleştirici'nin(ConditionalSimplifier) tersi.

ControlFlowSimplifier

Çeşitli kontrol akışı yapılarını basitleştirir:

- if'i boş gövde ile pop(koşul) ile değiştirin
- boş varsayılan anahtar durumunu kaldırın
- varsayılan durum yoksa boş anahtar durumunu kaldırın
- switch'i no cases ile pop(expression) ile değiştirin
- tek durumlu anahtarı if'e dönüştürün
- switch'i pop(expression) ve body ile yalnızca varsayılan durumla değiştirin
- switch'i eşleşen case gövdesine sahip const expr ile değiştirin
- `for` yerine kontrol akışını sonlandıran ve diğer break/continue olmadan `if` yazın
- bir fonksiyonun sonundaki `leave` ifadesini kaldırın.

Bu işlemlerin hiçbiri veri akışına bağlı değildir. StructuralSimplifier, veri akışına bağlı olan benzer görevleri yerine getirir.

ControlFlowSimplifier, çaprazlama sırasında `break` ve `continue` deyimlerinin varlığını veya yokluğunu kaydeder.

Ön koşul: Disambiguator, FunctionHoister, ForLoopInitRewriter. Önemli: EVM işlem kodlarını tanıtır ve bu nedenle şimdilik yalnızca EVM kodu üzerinde kullanılabilir.

DeadCodeEliminator

Bu optimizasyon aşaması ulaşılamayan kodu kaldırır.

Ulaşılamayan kod, bir blok içinde öncesinde leave, return, invalid, break, continue, selfdestruct veya revert bulunan kodlardır.

Fonksiyon tanımları, daha önceki kodlar tarafından çağrılabilecekleri için korunur ve bu nedenle ulaşılabilir olarak kabul edilir.

Bir for döngüsünün init(başlangıç) bloğunda bildirilen değişkenlerin kapsamı döngü gövdesine genişletildiğinden, ForLoopInitRewriter'ın bu adımdan önce çalışmasını gerektirir.

Önkoşul: ForLoopInitRewriter, Function Hoister, Function Grouper

EqualStoreEliminator

Bu adım, `mstore(k, v)` ve `sstore(k, v)` çağrılarını, daha önce `mstore(k, v)` / `sstore(k, v)` çağrısı yapılmışsa, arada başka bir depo yoksa ve `k` ve `v` değerleri değişmemişse kaldırır.

Bu basit adım, SSA dönüşümü ve Common Subexpression Eliminator'den sonra çalıştırılırsa etkili olur, çünkü SSA değişkenlerin değişmeyeceğinden emin olur ve Common Subexpression Eliminator, değerin aynı olduğu biliniyorsa tam olarak aynı değişkeni yeniden kullanır.

Önkoşullar: Disambiguator, ForLoopInitRewriter

UnusedPruner

Bu adım, hiçbir zaman başvurulmayan tüm fonksiyonların tanımlarını kaldırır.

Ayrıca, hiçbir zaman başvurulmayan değişkenlerin tanımlarını da kaldırır. Tanımlama taşınabilir olmayan bir değer atarsa, ifade korunur ancak değeri atılır.

Tüm taşınabilir ifade ifadeleri (atanmamış ifadeler) kaldırılır.

StructuralSimplifier

Bu, yapısal düzeyde çeşitli basitleştirmeler gerçekleştiren genel bir adımdır:

- if ifadesini boş gövde ile `pop(koşul)` ile değiştirin
- if ifadesini gövdesine göre doğru koşulla değiştirin
- if deyimini yanlış koşulla kaldırın
- tek durumlu anahtarı if'e dönüştürün
- switch'i sadece varsayılan durumla `pop(expression)` ve gövde ile değiştirin
- case gövdesini eşleştirerek switch'i gerçek ifade ile değiştirin
- yanlış koşullu for döngüsünü başlatma kısmı ile değiştirin

Bu bileşen Dataflow Analyzer'ı kullanır.

BlockFlattener

Bu aşama, iç bloktaki ifadeyi dış bloktaki uygun yere yerleştirerek iç içe geçmiş blokları ortadan kaldırır. FunctionGrouper'a bağlıdır ve FunctionGrouper tarafından üretilen formu korumak için en dıştaki bloğu düzleştirmez.

```
{
    {
        let x := 2
        {
            let y := 3
            mstore(x, y)
        }
    }
}
```

dönüştürülür

```
{
    {
        let x := 2
        let y := 3
        mstore(x, y)
    }
}
```

Kodda belirsizlikler giderildiği sürece bu bir soruna yol açmaz çünkü değişkenlerin kapsamları yalnızca büyüyebilir.

LoopInvariantCodeMotion

Bu optimizasyon, taşınabilir SSA değişken tanımlamalarını döngünün dışına taşır.

Yalnızca bir döngünün gövdesindeki veya son bloğundaki en üst düzeydeki ifadeler dikkate alınır, yani koşullu branşların(branch) içindeki değişken tanımlamaları döngünün dışına taşınmaz.

Gereksinimler:

- Disambiguator, ForLoopInitRewriter ve FunctionHoister önceden çalıştırılmalıdır.
- İfade ayırıcı ve SSA dönüşümü daha iyi sonuç elde etmek için önceden çalıştırılmalıdır.

Fonksiyon Düzeyinde Optimizasyonlar

FunctionSpecializer

Bu adım, fonksiyonu gerçek argümanlarıyla özelleştirir.

Bir fonksiyon, örneğin `fonksiyon f(a, b) { sstore (a, b) }`, literal argümanlarla çağrılırsa, örneğin `f(x, 5)`, burada `x` bir tanımlayıcıdır, sadece bir argüman alan yeni bir `f_1` fonksiyonu oluşturularak özelleştirilebilir, yani,

```
function f_1(a_1) {
    let b_1 := 5
    sstore(a_1, b_1)
}
```

Diğer optimizasyon adımları fonksiyonda daha fazla basitleştirme yapabilecektir. Optimizasyon adımı esas olarak inline edilmeyecek fonksiyonlar için kullanışlıdır.

Önkoşullar: Disambiguator, FunctionHoister

LiteralRematerialiser, doğruluk için gerekli olmasa da bir ön koşul olarak önerilir.

UnusedFunctionParameterPruner

Bu adım, bir fonksiyondaki kullanılmayan parametreleri kaldırır.

Eğer bir parametre kullanılmıyorsa, `fonksiyon f(a,b,c) -> x, y { x := div(a,b) }` içindeki `c` ve `y` gibi, parametreyi kaldırırız ve aşağıdaki gibi yeni bir "bağlama" fonksiyonu oluştururuz:

```
function f(a,b) -> x { x := div(a,b) }
function f2(a,b,c) -> x, y { x := f(a,b) }
```

ve `f` öğesine yapılan tüm referansları `f2` ile değiştirmelisiniz. Tüm `f2` referanslarının `f` ile değiştirildiğinden emin olmak için inliner daha sonra çalıştırılmalıdır.

Önkoşullar: Disambiguator, FunctionHoister, LiteralRematerialiser.

LiteralRematerialiser adımı doğruluk için gerekli değildir. Aşağıdaki gibi durumlarla başa çıkmaya yardımcı olur: `fonksiyon f(x) -> y { revert(y, y} }` burada `y` değişmezi `0` değeri ile değiştirilecek ve fonksiyonu yeniden yazmamıza izin verecektir.

EquivalentFunctionCombiner

İki fonksiyon sözdizimsel(syntactically) olarak eşdeğerse, değişkenlerin yeniden adlandırılmasına izin verirken herhangi bir yeniden sıralamaya izin vermiyorsa, fonksiyonlardan birine yapılan herhangi bir referans diğeriyle değiştirilir.

Fonksiyonun asıl kaldırılma işlemi Unused Pruner tarafından gerçekleştirilir.

Fonksiyon Inlining (Function Inlining)

ExpressionInliner

Optimize edicinin bu bileşeni, fonksiyonel ifadeler içinde inline edilebilen fonksiyonları, yani tek bir değer döndüren fonksiyonları inline ederek kısıtlı fonksiyon inliningi gerçekleştirir:

- tek bir değer döndüren.
- `r := <fonksiyonel ifade>` gibi bir gövdeye sahip olan.
- ne kendilerine ne de sağ taraftaki `r` ye referans verirler.

Ayrıca, tüm parametreler için aşağıdakilerin tümünün doğru olması gerekir:

- Bağımsız değişken taşınabilir.
- Parametreye ya fonksiyon gövdesinde iki kereden az referans verilir ya da argüman oldukça ucuzdur ("cost" en fazla 1, 0xff'ye kadar bir sabit gibi).

Örnek: Inline edilecek fonksiyon `function f(...) -> r { r := E }` biçimindedir; burada `E`, `r` ye referans vermeyen bir ifadedir ve fonksiyon çağrısındaki tüm argümanlar taşınabilir ifadelerdir.

Bu inlining işleminin sonucu her zaman tek bir ifadedir.

Bu bileşen yalnızca benzersiz adlara sahip kaynaklarda kullanılabilir.

FullInliner

Full Inliner, belirli fonksiyonların belirli çağrılarını fonksiyonun gövdesi ile değiştirir. Bu çoğu durumda çok yararlı değildir, çünkü kod boyutunu artırır ayrıca bir faydası da yoktur. Genellikle kod çok pahalıdır ve daha verimli bir kod yerine daha kısa bir kodu tercih ederiz. Yine de aynı durumlarda, bir fonksiyonun inlining işleminin sonraki optimizer adımları üzerinde olumlu etkileri olabilir. Örneğin, fonksiyon argümanlarından birinin sabit olması durumunda durum böyledir.

Inlining sırasında, fonksiyon çağrısının inline edilip edilmeyeceğini söylemek için bir heuristic kullanılır. Mevcut heuristic, çağrılan fonksiyon küçük olmadığı sürece "büyük" fonksiyonları inline etmez. Sadece bir kez kullanılan fonksiyonların yanı sıra orta büyüklükteki fonksiyonlar da inline edilirken, sabit argümanlara sahip fonksiyon çağrıları biraz daha büyük fonksiyonlara izin verir.

Gelecekte, bir fonksiyonu hemen inline etmek yerine sadece uzmanlaştıran bir geri izleme bileşeni ekleyebiliriz, bu da belirli bir parametrenin her zaman bir sabitle değiştirildiği fonksiyonun bir kopyasının oluşturulacağı anlamına gelir. Bundan sonra, optimize ediciyi bu özelleştirilmiş fonksiyon üzerinde çalıştırabiliriz. Eğer büyük kazançlar elde edilirse, özelleştirilmiş fonksiyon korunur, aksi takdirde orijinal fonksiyon kullanılır.

Temizlik (Cleanup)

Temizleme, optimizer çalışmasının sonunda gerçekleştirilir. Bölünmüş ifadeleri tekrar derin iç içe geçmiş ifadelerle birleştirmeye çalışır ve ayrıca değişkenleri mümkün olduğunca ortadan kaldırarak yığın(stack) makineleri için "derlenebilirliği" iyileştirir.

ExpressionJoiner

Bu işlem, ifade ayırıcının(expression splitter) tersidir. Tam olarak bir referansı olan bir dizi değişken tanımlamasını karmaşık bir ifadeye dönüştürür. Bu aşama, fonksiyon çağrılarının ve işlem kodu yürütmelerinin sırasını tamamen korur. İşlem kodlarının değişebilirliğine ilişkin herhangi bir bilgi kullanmaz; bir değişkenin değerini kullanım yerine taşımak herhangi bir işlev çağrısının veya işlem kodu yürütmesinin sırasını değiştirecekse, dönüşüm gerçekleştirilmez.

Bileşenin, bir değişken atamasının atanmış değerini veya birden fazla kez başvurulan bir değişkeni taşımayacağını unutmayın.

`let x := add(0, 2) let y := mul(x, mload(2))` kod parçacığı dönüştürülmez, çünkü `add` ve `mload` işlem kodlarına yapılan çağrıların sırasının değiştirilmesine neden olur - ancak `add` taşınabilir olduğu için bu bir fark yaratmaz.

İşlem kodlarını bu şekilde yeniden sıralarken, değişken referansları ve literaller göz ardı edilir. Bu nedenle, `let x := add(0, 2) let y := mul(x, 3)` kod parçacığı, `add` işlem kodu `3` literalinin değerlendirilmesinden sonra çalıştırılacak olsa bile, `let y := mul(add(0, 2), 3)` olarak dönüştürülür.

SSAReverser

Bu, Common Subexpression Eliminator ve Unused Pruner ile birleştirildiğinde SSA dönüşümünün etkilerini tersine çevirmeye yardımcı olan küçük bir adımdır.

Ürettiğimiz SSA formu EVM ve WebAssembly'de kod üretimi için zararlıdır çünkü çok sayıda yerel değişken üretir. Yeni değişken bildirimleri yerine mevcut değişkenleri atamalarla yeniden kullanmak daha iyi sonuç verecektir.

SSA dönüşümleri şu şekilde

```
let a := calldataload(0)
mstore(a, 1)
```

yeniden yazılır

```
let a_1 := calldataload(0)
let a := a_1
mstore(a_1, 1)
let a_2 := calldataload(0x20)
a := a_2
```

Sorun, a değişkenine her başvurulduğunda a yerine a_1 değişkeninin kullanılmasıdır. SSA dönüşümü bu formdaki ifadeleri sadece tanımlama ve atamayı değiştirerek değiştirir. Yukarıdaki kod parçacığı şu şekle dönüşür

```
let a := calldataload(0)
let a_1 := a
mstore(a_1, 1)
a := calldataload(0x20)
let a_2 := a
```

Bu çok basit bir denklik dönüşümüdür, ancak şimdi Common Subexpression Eliminator'ü çalıştırdığımızda, a_1 değişkeninin tüm kullanımlarını a ile değiştirecektir (a yeniden atanana kadar). Unused Pruner daha sonra a_1 değişkenini tamamen ortadan kaldıracak ve böylece SSA dönüşümünü tamamen tersine çevirecektir.

StackCompressor

Ethereum Sanal Makinesi için kod oluşturmayı zorlaştıran bir sorun, ifade yığınına ulaşmak için 16 slotluk katı bir sınır olmasıdır. Bu da aşağı yukarı 16 yerel değişken sınırı anlamına gelmektedir. Yığın sıkıştırıcı Yul kodunu alır ve EVM bayt koduna derler. Yığın farkı çok büyük olduğunda, bunun hangi fonksiyonda gerçekleştiğini kaydeder.

Böyle bir soruna neden olan her bir fonksiyon için, değerlerinin maliyetine göre sıralanan belirli değişkenleri agresif bir şekilde ortadan kaldırmak için özel bir taleple Rematerialiser çağrılır.

Başarısızlık durumunda, bu prosedür birden çok kez tekrarlanır.

Rematerialiser

Rematerialisation aşaması, değişken referanslarını değişkene en son atanan ifade ile değiştirmeye çalışır. Bu elbette yalnızca bu ifadenin değerlendirilmesi nispeten daha ucuzsa faydalıdır. Ayrıca, yalnızca ifadenin değeri atama noktası ile kullanım noktası arasında değişmediyse anlamsal olarak denktir. Bu aşamanın ana faydası, bir değişkenin tamamen ortadan kaldırılmasına yol açarsa yığın yuvalarından tasarruf edebilmesidir (aşağıya bakın), ancak ifade çok ucuzsa EVM'de bir DUP işlem kodundan da tasarruf edebilir.

Rematerialiser, her zaman hareketli olan değişkenlerin mevcut değerlerini izlemek için Dataflow Analyzer'ı kullanır. Değer çok ucuzsa veya değişkenin ortadan kaldırılması açıkça istenmişse, değişken referansı geçerli değeriyle değiştirilir.

ForLoopConditionOutOfBody

ForLoopConditionIntoBody dönüşümünü tersine çevirir.

Herhangi bir taşınabilir c için,

```
for { ... } 1 { ... } {
if iszero(c) { break }
...
}
```

dönüşür

```
for { ... } c { ... } {
...
}
```

ve döner

```
for { ... } 1 { ... } {
if c { break }
...
}
```

dönüşür

```
for { ... } iszero(c) { ... } {
...
}
```

LiteralRematerialiser bu adımdan önce çalıştırılmalıdır.

WebAssembly'a özgü

Ana Fonksiyon(MainFunction)

En üstteki bloğu, girdisi veya çıktısı olmayan belirli bir ada ("main") sahip bir fonksiyon olarak değiştirir.

Fonksiyon Gruplayıcısına bağlıdır.

3.21 Sözleşme Meta Verisi

Solidity derleyicisi derlenen sözleşme hakkında bilgiler içeren "şözleşme meta verisi" adlı bir JSON dosyasını otomatik olarak oluşturur. Bu dosyayı derleyici sürümünü, kaynak dosyaları, ABI ve NatSpec dokümentasyonunu sorgulamak için kullanabilirsiniz. Bu sayede sözleşmenin kaynak kodunu doğrulayabilir ve sözleşmeyle daha güvenli bir şekilde etkileşime geçebilirsiniz.

Derleyici varsayılan şeklinde meta veri dosyasının IPFS hash'ini bayt kodun sonuna ekler (detaylar için aşağıya göz atınız). Böylelikle meta veri merkezi bir veri sağlayıcısına bağlı kalmadan doğrulanmış bir şekilde indirebilirsiniz. Bu konuda diğer seçenekler Swarm hash'ini kullanmak veya meta veri hash'ini bayt kodun sonuna eklememektir. Bu seçenekler *Standard JSON Arayüzü* üzerinden ayarlanabilir.

Meta veri dosyasına erişilebilmesi için dosyayı IPFS, Swarm veya başka bir serviste yayınlamanız gerekmektedir. Dosyayı `SözleşmeAdı_meta.json` adında bir dosya oluşturan `solc --metadata` komutunu kullanarak yaratabilirsiniz. Dosya kaynak kodu dosyalarının IPFS ve Swarm hash'lerini içerdiği için bütün kaynak kodu dosyalarını ve meta veri dosyasını yüklemeniz gerekmektedir.

Meta veri dosyası aşağıdaki formattadır. Fakat aşağıdaki örnek okuması kolay şekilde gösterilmektedir. Normalde düzgün şekilde formatlanmış meta veri tırnak işaretlerini doğru şekilde kullanmalı, metindeki boşlukları en aza indirmeli ve JSON nesnesinin anahtarlarını tutarlı bir formatlamaya ulaşmak için sıralamalıdır. Normalde JSON dosyalarında yorum satırlarına müsade edilmezken burada yalnızca gösterim amaçlı olarak eklenmiştir.

```
{
  // Mecburi: Meta veri formatının sürümü
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
  "version": "1",
  // Mecburi: Kaynak kodu dili. Spesifikasyonun bir "alt-sürümü"nü seçer.
  "language": "Solidity",
  // Mecburi: Derleyici hakkında detaylar. İçeriği kullanılan dile
  // göre değişebilir.
  "compiler": {
    // Solidity için mecburi: Derleyici sürümü.
    "version": "0.4.6+commit.2dabbdf0.Emscripten.clang",
    // Opsiyonel: Bu çıktıyı elde etmek için kullanılan
    // derleyici binary'sinin  hash'i
    "keccak256": "0x123..."
  },
  // Mecburi: Derleyici kaynak dosyaları/kaynak birimleri.
  // Her bir anahtar dosya adıdır.
  "sources":
  {
    "myFile.sol": {
      // Mecburi: kaynak dosyasının keccak256 hash'i.
      "keccak256": "0x123...",
      // Mecburi: Kaynak dosyasının sıralanmış URL'leri. Herhangi bir
      // protokol kullanılabilir fakat bir Swarm URL'i önerilir.
      // ("content" kullanıldığında mecburi değildir, aşağıya bakınız)
      "urls": [ "bzzr://56ab..." ],
      // Opsiyonel: Kaynak kodunda belirtilen şekilde SPDX lisans kodu
      "license": "MIT"
    },
    "destructible": {
      // Mecburi: Kaynak dosyasının keccak256 hash'i.
      "keccak256": "0x234...",
      // Mecburi: Kaynak dosyasının kelimesi kelimesine içeriği
      // ("url" kullanıldığında mecburi değildir)
      "content": "contract destructible is owned { function destroy() { if (msg.sender
↪== owner) selfdestruct(owner); } }"
    }
  },
  // Mecburi: Derleyici ayarları
  "settings":
  {
    // Solidity için mecburi: yeniden eşlemelerin sıralı listesi
    "remappings": [ ":g=/dir" ],
    // Opsiyonel: Optimize edici ayarları. "enabled" vs "runs" anahtarları
    // artık kullanılmamaktadır ve geriye dönük uyumluluk için verilmiştir.
    "optimizer": {
      "enabled": true,
      "runs": 500,
      "details": {
        // peephole'ün varsayılanı "true"dur
        "peephole": true,
        // inliner'ın varsayılanı "true"dur
        "inliner": true,
        // jumpdestRemover'ın varsayılanı "true"dur
        "jumpdestRemover": true,
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        "orderLiterals": false,
        "deduplicate": false,
        "cse": false,
        "constantOptimizer": false,
        "yul": true,
        // Opsiyonel: Yalnızca "yul" "true" ise mevcut
        "yulDetails": {
          "stackAllocation": false,
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    "metadata": {
      // Girdi json'da kullanılan ayarın aynısı. Varsayılan: "false"
      "useLiteralContent": true,
      // Girdi json'da kullanılan ayarın aynısı. Varsayılan: "ipfs"
      "bytecodeHash": "ipfs"
    },
    // Solidity için mecburi: Bu meta veri hangisi için yaratıldıysa o
    // dosya ile sözleşme veya kütüphanenin adı.
    "compilationTarget": {
      "myFile.sol": "MyContract"
    },
    // Solidity için mecburi: Kullanılan kütüphanelerin adresleri
    "libraries": {
      "MyLib": "0x123123..."
    }
  },
  // Mecburi: Sözleşme için oluşturulan bilgiler
  "output":
  {
    // Mecburi: Sözleşmenin ABI tanımı
    "abi": [/* ... */],
    // Mecburi: Sözleşmenin NatSpec kullanıcı dokümantasyonu
    "userdoc": [/* ... */],
    // Mecburi: Sözleşmenin NatSpec geliştirici dokümantasyonu
    "devdoc": [/* ... */]
  }
}
```

Uyarı: Elde edilen sözleşmenin bayt kodu meta veri hash'ini varsayılan şekilde içerdiği için meta veride yapılacak herhangi bir değişiklik bayt kodda bir değişikliğe sebep olabilir. Bir dosya adı veya yolunda yapılacak bir değişiklik veya meta veri bütün kaynakların hash'ini içerdiği için kaynaklarda eklenecek veya çıkarılacak bir boşluk farklı bir meta veri ile dolayısı ile farklı bir bayt kod ile sonuçlanabilir.

Not: Yukarıdaki ABI tanımının belirlenmiş bir sıralaması yoktur ve derleyici sürümlerine göre değişebilir. Fakat Solidity 0.5.12 sürümüyle birlikte ABI dizisi belirili bir sıralamayı takip eder.

3.21.1 Meta Veri Hash'inin Bayt Kod İçinde Kodlanması

Meta veriyi indirmenin farklı yollarını ileride destekleyebileceğimiz için `{"ipfs": <IPFS hash>, "solc": <compiler version>}` eşlemesi CBOR ile kodlanmıştır. Eşleme birden fazla anahtar içerebileceği için (aşağıdaki gibi) ve kodlamanın en başını bulması kolay olmayabileceği için kodlamanın uzunluğu 2 bayt big-endian şeklinde (sona) eklenmiştir. Solidity derleyicisinin mevcut sürümü çoğunlukla aşağıdaki kodu yüklenen bayt kodun sonuna ekler.

```
0xa2
0x64 'i' 'p' 'f' 's' 0x58 0x22 <34 bayt IPFS hash'i>
0x64 's' 'o' 'l' 'c' 0x43 <3 bayt sürüm kodlaması>
0x00 0x33
```

Meta veriyi indirmek için yüklenen bayt kodun sonu bu örüntüye uyuyor mu diye bakılabilir ve elde edilen IPFS hash'i ile dosya indirilebilir.

solc'in tamamlanmış sürümleri yukarıdaki 3 baytlık kodlama ile kodlanırken (her bir "büyük", "küçük", ve "yama" sürümü için birer bayt), tamamlanmamış ön sürümler sürümün derlenme tarihi ve commit hash'ini içeren komple bir string ile kodlanır.

Not: CBOR eşlemesi farklı anahtarlar kullanabileceği için bu kodlamanın `0xa264` ile başlamasına güvenmemek ve kodlamayı doğru şekilde çözmek gerekir. Örneğin, kod oluşturmayı etkileyecek herhangi bir deneysel özellik kullanıldıysa eşleme `"experimental": true`'yu içerir.

Not: Derleyici şu anda varsayılan olarak meta verinin IPFS hash'ini kullanıyor olsa da ileride bzzr1 hash veya daha farklı bir hash kullanabilir. Bu yüzden bu serinin `0xa2 0x64 'i' 'p' 'f' 's'` ile başlamasına güvenmemeniz gerekir. Bu CBOR yapısına ayrıca ileride farklı veriler ekleyebiliriz. Bu sebeple en doğrusu uygun bir CBOR ayrıştırıcı (parser) kullanmanızdır.

3.21.2 Otomatik Arayüz Oluşturmanın Kullanılması ve NatSpec

Meta veri şu şekilde kullanılır: Bir sözleşmeyle etkileşime geçmek isteyen bir bileşen (örn. Mist veya başka bir cüzdan) sözleşmenin kodunu indirir. Daha sonra bu koddan IPFS/Swarm hash'ini elde eder ve meta veri dosyası indirilir. Bu dosya yukarıdaki yapıya uygun şekilde JSON formatında çözülür.

İlgili bileşen, ABI'ı otomatik olarak basit bir kullanıcı arayüzü oluşturmak için kullanabilir.

Ek olarak cüzdan, kullanıcı bir sözleşmeyle etkileşime geçerken kullanıcıdan işlem için imza onayı istemenin yanında kullanıcıya bir onay mesajı göstermek için NatSpec kullanıcı dokümantasyonunu kullanabilir.

Daha fazla bilgi için *Ethereum Natural Language Specification (NatSpec) format* ını okuyunuz.

3.21.3 Kaynak Kodu Doğrulama için Kullanım

Derlemeyi doğrulamak için kaynaklar meta veri dosyasında verilen bağlantılar ile IPFS/Swarm'dan indirilebilir. Derleyicinin ("resmi" sürüm mü değil mi diye bakılan) doğru sürümü ilgili girdiye belirtilen ayarlar ile çağrılır. (Derlemeden) elde edilen bayt kodu (sözleşmeyi) yaratma işleminin verisi ile veya `CREATE` işlem kodu verisi ile karşılaştırılır. Bu, hash'i zaten bayt kodun bir parçası olduğu için meta veriyi otomatik olarak doğrular. Fazladan veri, kullanıcıya sunulan arayüze uygun şekilde çözülmesi gereken constructor girdi verisidir.

sourcify (npm paketi) deposunda bu özelliği nasıl kullanabileceğinize dair kodu görebilirsiniz.

3.22 Sözleşme ABI Spesifikasyonu

3.22.1 Temel Tasarım

Sözleşme Uygulama Binary Arayüzü (ABI), Ethereum ekosistemindeki sözleşmelerle hem blok zinciri dışından hem de sözleşmeler arası etkileşimde bulunmanın standart yoludur. Veriler, bu spesifikasyonda açıklandığı gibi türlerine göre kodlanır. Şifreleme kendi kendini tanımlamaz ve bu nedenle şifreyi çözmek için bir şema gerekir.

Bir sözleşmenin arayüz fonksiyonlarının güçlü bir şekilde yazıldığını, derleme zamanında bilindiğini ve statik olduğunu varsayıyoruz. Tüm sözleşmelerin, çağırdıkları sözleşmelerin arayüz tanımlamalarına derleme zamanında sahip olacağını varsayıyoruz.

Bu spesifikasyon, arayüzü dinamik olan veya başka bir şekilde yalnızca çalışma zamanında bilinen sözleşmeleri ele almaz.

3.22.2 Function Selector (Function Selector)

Bir fonksiyon çağrısı için çağrı verisinin(call data) ilk dört baytı çağrılacak fonksiyonu belirtmektedir. Bu, fonksiyonun imzasının Keccak-256 hash'inin ilk (sol, büyük endian'da yüksek dereceden) dört baytıdır. İmza, veri konumu belirteci olmadan temel prototipin kanonik ifadesi, yani parametre türlerinin parantezli listesiyle birlikte fonksiyon adı olarak tanımlanır. Parametre tipleri tek bir virgülle ayrılır - boşluk kullanılmaz.

Not: Bir fonksiyonun geri dönüş tipi bu imzanın bir parçası değildir. ref:*Solidity'nin fonksiyon aşırı yüklemesinde(overloading) <overload-function>* dönüş tipleri dikkate alınmaz. Bunun nedeni, fonksiyon çağrısı çözümlemesini içerikten bağımsız tutmaktır. Ancak *JSON ABI* tanımı hem girdileri hem de çıktıları içerir.

3.22.3 Argüman Şifreleme

Beşinci bayttan başlayarak şifrelenmiş tüm argümanları takip eder. Bu şifreleme başka yerlerde de kullanılır, örneğin geri dönüş değerleri ve event argümanları, fonksiyonu belirten dört bayt olmadan aynı şekilde şifrelenir.

3.22.4 Tipler (Types)

Aşağıdaki ana tipler mevcuttur:

- `uint<M>`: `M` bitlik işaretsiz tamsayı (unsigned) türü, `0 < M <= 256`, `M % 8 == 0`. e.g. `uint32`, `uint8`, `uint256`.
- `int<M>`: `M` bitlik ikiye tamamlayıcı işaretli tamsayı (signed integer) türü, `0 < M <= 256`, `M % 8 == 0`.
- `address`: varsayılan değerlendirme ve dil yazımı dışında `uint160` ile eşdeğerdir. Fonksiyon seçicisini hesaplarken `address` kullanılır.
- `uint`, `int`: sırasıyla `uint256`, `int256` için eş anlamlı terimlerdir. Fonksiyon seçicisini hesaplamak için `uint256` ve `int256` kullanılmalıdır.
- `bool`: 0 ve 1 değerleriyle sınırlandırılmış `uint8` ile eşdeğerdir. Fonksiyon seçicisini hesaplamak için `bool` kullanılır.
- `fixed<M>x<N>`: `M` bitlerinin işaretli(signed) fixed-point ondalık sayısı, `8 <= M <= 256`, `M % 8 == 0` ve `0 < N <= 80`, `v` değerini `v / (10 ** N)` olarak gösterir.
- `ufixed<M>x<N>`: `fixed<M>x<N>` öğesinin işaretsiz(unsigned) varyantı.
- `fixed`, `ufixed`: sırasıyla `fixed128x18`, `ufixed128x18` için eş anlamlı terimlerdir. Fonksiyon seçiciyi hesaplamak için `fixed128x18` ve `ufixed128x18` kullanılmalıdır.
- `bytes<M>`: `M` baytlarının binary tipi, `0 < M <= 32`.
- `function`: bir adres (20 bayt) ve ardından bir fonksiyon seçici (4 bayt). `bytes24` ile aynı biçimde şifrelenir.

Aşağıdaki (sabit boyutlu) dizi türü bulunmaktadır:

- `<tip>[M]`: verilen tipte bulunan `M` elemanlı, `M >= 0`, sabit uzunlukta bir dizidir.

> **Not:** Bu ABI spesifikasyonu sıfır elemanlı sabit uzunluklu dizileri ifade edebilse de, bunlar derleyici tarafından desteklenmez.

Aşağıdaki sabit boyutlu olmayan tipler de mevcuttur:

- `bytes`: dinamik boyutlu bayt sırası.
- `string`: UTF-8 şifrelenmiş olduğu varsayılan dinamik boyutlu unicode bir dizedir.
- `<type>[]`: belirtilen tipteki elemanlardan oluşan değişkenlik gösterebilen uzunlukta bir dizidir.

Tipler, virgülle ayrılmış parantezler içine alınarak bir tuple olarak birleştirilebilir:

- `(T1,T2,...,Tn)`: `T1`, …, `Tn` tiplerinden oluşan bir tuple, `n >= 0`

Tuple'ların tuple'larını, tuple'ların dizilerini ve benzerlerini oluşturmak mümkündür. Sıfır tuple oluşturmak da mümkündür (genellikle `n == 0`).

Solidity'yi ABI Tipleriyle Eşleştirme

Solidity, tuple'lar haricinde yukarıda aynı adlandırmalarla sunulan tüm tipleri destekler. Öte yandan, bazı Solidity tipleri ABI tarafından desteklenmez. Aşağıdaki tabloda sol sütunda bulunan ve ABI'nin bir parçası olmayan Solidity tipleri ve sağ sütunda ise bunları temsil eden ABI tipleri verilmiştir.

Solidity	ABI
address payable	`address`
contract	`address`
enum	`uint8`
kullanıcı tanımlı değişken tipleri	temel değer tipi
struct	`tuple`

Uyarı: `0.8.0` sürümünden önce enumlar 256`dan fazla elemana sahip olabiliyordu ve herhangi bir elemanın değerini tutmaya yetebilecek büyüklükteki en küçük tamsayı tipiyle ifade ediliyordu.

3.22.5 Şifreleme için Tasarım Kriterleri

Şifreleme aşağıdaki özelliklere sahip olacak şekilde tasarlanmıştır; bu özellikler özellikle bazı bağımsız değişkenlerin iç içe diziler olması halinde kullanışlıdır:

1. Bir değere erişmek için gereken okuma sayısı en büyük değerin argüman dizi yapısı içinde sahip olduğu derinlik kadardır, yani `a_i[k][l][r]` öğesini almak için dört okuma gerekir. ABI'nin önceki bir sürümünde, okuma sayısı en kötü senaryoda toplam dinamik parametre sayısı ile doğrusal olarak ölçeklenmekteydi.
2. Bir değişken veya dizi elemanının verileri diğer verilerle iç içe geçmez ve yeniden konumlandırılabilir, yani yalnızca ilişkili "adresler" kullanabilirler.

3.22.6 Şifrelemenin Formal Spesifikasyonu

Statik ve dinamik türleri birbirinden ayırırız. Statik tipler yerinde şifrelenirken, dinamik tipler mevcut bloktan sonra ayrı olarak atanmış bir konumda şifrelenir.

Tanım: Aşağıdaki tipler "dinamik" olarak adlandırılır:

- `bytes`
- `string`
- Herhangi bir `T` için `T[]`
- Herhangi bir dinamik `T` ve herhangi bir `k >= 0` için `T[k]`
- `(T1,...,Tk)` eğer `Ti` bazı `1 <= i <= k` için dinamik yapıda ise

Diğer tüm türler "statik" olarak adlandırılır.

Tanım: `len(a)`, `a` binary dizesinde bulunan bayt sayısıdır. Ayrıca `len(a)` türünün `uint256` olduğu varsayılır.

Gerçek şifreleme olan `enc``i, ABI tiplerindeki değerlerin binary stringlere eşlenmesi olarak tanımlıyoruz, öyle ki ``len(enc(X))` ancak ve ancak `X` tipi dinamik olduğu durumlarda `X` değerine bağlı olacaktır.

Tanım: For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being

- `(T1,...,Tk)` için `k >= 0` ve herhangi bir `T1`, ..., `Tk` tipi

  `enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`

  Burada `X = (X(1), ..., X(k))` ve `head` ve `tail` `Ti` için aşağıdaki gibi tanımlanır:

  eğer `Ti` statik ise:

  `head(X(i)) = enc(X(i))` ve `tail(X(i)) = ""` (boş dize)

  Aksi takdirde, yani `Ti` dinamik ise:

  `head(X(i)) = enc(len( head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1)) )) tail(X(i)) = enc(X(i))`

  Dinamik durumlarda, `head(X(i))` ifadesi iyi tanımlanmıştır çünkü başlık parçalarının uzunlukları değerlere değil sadece tiplere bağlıdır. `head(X(i))` değeri, `enc(X)` öğesinin başlangıç noktasına göre `tail(X(i))` öğesinin başlangıç noktasındaki ofset değeridir.

- Herhangi bir `T` ve `k` için `T[k]`:

  `enc(X) = enc((X[0], ..., X[k-1]))`

  Yani, aynı tipte `k` elemanlı bir tuple gibi şifrelenir.

- Herhangi bir `T` ve `k` için `T[k]`:

  `enc(X) = enc((X[0], ..., X[k-1]))`

  Yani, aynı tipte `k` elemanlı bir tuple gibi şifrelenir.

- k`` uzunluğunda `bytes` (`uint256` tipinde olduğu varsayılır):

  `enc(X) = enc(k) pad_right(X)`, yani bayt sayısı bir `uint256` olarak şifrelenir, ardından bayt sırası olarak `X``in gerçek değeri ve sonrasında ``len(enc(X))` 32'nin katı olacak uzunlukta minimum sıfır bayt sayısı gelir.

- `string`:

  `enc(X) = enc(enc_utf8(X))`, yani `X` UTF-8 biçiminde şifrelenir ve bu değer `bytes` türünde olarak değerlendirilir ve ardından şifreli hale getirilir. Bu sonraki şifreleme işleminde kullanılan uzunluğun karakter sayısı değil, UTF-8 kodlu stringin bayt sayısı olduğuna dikkat edin.

- `uint<M>`: `enc(X)`, `X`'in big-endian biçimindeki şifrelemesi olup, uzunluğu 32 bayt olacak şekilde yüksek dereceden (sol) tarafı sıfır bayt ile doldurulmuştur.

- `address`: `uint160` örneğinde olduğu gibi

- `int<M>`: `enc(X)`, negatif `X` için `0xff` baytları ile ve negatif olmayan `X` değerleri için sıfır baytları ile doldurulmuş ve uzunluğu 32 bayt olacak şekilde `X`'in big-endian ikiye tamamlayıcı şifrelemesidir.

- `bool`: `uint8` örneğinde olduğu gibi, `true` için `1` ve `false` için `0` değerini kullanır.

- `fixed<M>x<N>`: `enc(X)`, `enc(X * 10**N)``dir; burada ``X * 10**N` bir `int256` olarak yorumlanmaktadır..

- `fixed`: `fixed128x18` örneğinde olduğu gibi

- `ufixed<M>x<N>`: `enc(X)`, `enc(X * 10**N)``dir; burada ``X * 10**N` bir `uint256` olarak yorumlanmaktadır.

- `ufixed`: `ufixed128x18` örneğinde olduğu gibi

- `bytes<M>`: `enc(X)`, `X` içindeki baytların sondaki sıfır baytlarla beraber 32 bayt uzunluğa kadar doldurulmuş bir sırasıdır.

Herhangi bir `X` için `len(enc(X))` değerinin 32'nin bir katı olduğuna dikkat edin.

3.22.7 Fonksiyon Seçicisi ve Argüman Şifrelemesi

Sonuç olarak, `f` fonksiyonuna `a_1, ..., a_n` parametreleri ile yapılan bir çağrı şu şekilde şifrelenir

`function_selector(f) enc((a_1, ..., a_n))`

ve `f``nin ``v_1, ..., v_k` dönüş değerleri şu şekilde şifrelenmektedir.

```
enc((v_1, ..., v_k))
```

yani değerler bir tuple halinde birleştirilir ve kodlanır.

3.22.8 Örnekler

Sözleşmeye göre:

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

contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

Böylece `Foo` örneğimiz için `69` ve `true` parametreleriyle `baz` ı çağırmak istersek, toplam 68 bayt iletiriz, bu da şu şekilde ayrılabilir:

- `0xcdcd77c0`: Method ID. Bu, `baz(uint32,bool)` imzasının ASCII formunun Keccak hash'inin ilk 4 baytı olacak şekilde türetilecektir.
- `0x0000000000000000000000000000000000000000000000000000000000000045`: ilk parametre, 32 bayta doldurulmuş bir uint32 değeri `69`
- `0x0000000000000000000000000000000000000000000000000000000000000001`: ikinci parametre - boolean `true`, 32 bayta kadar doldurulur

Toplam olarak:

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000
```

Tek bir `bool` döndürür. Örneğin, `false` döndürürse, çıktısı tek bir bool olan `0x0000000000000000000000000000000000000000000000000000000000000000` tek bayt dizisi olacaktır.

Eğer `bar` argümanını `["abc", "def"]` ile çağırmak isteseydik, toplam 68 bayt aktarmamız gerekirdi:

- `0xfce353f6`: Method ID. Bu, `bar(bytes3[2])` imzasından türetilmiştir.
- `0x6162630000000000000000000000000000000000000000000000000000000000`: ilk parametrenin ilk kısmı, bir `bytes3` değeri olan `"abc"` (sola hizalı).
- `0x6465660000000000000000000000000000000000000000000000000000000000`: ilk parametrenin ikinci kısmı, bir `bytes3` değeri olan `"def"` (sola hizalı).

Toplam olarak:

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000000000006465660000000000000000000000000000
```

Eğer `sam``ı ``"dave"`, `true` ve `[1,2,3]` argümanlarıyla çağırmak isteseydik, toplam 292 bayt aktarmamız gerekirdi:

- `0xa5643bf2`: Method ID. Bu, `sam(bytes,bool,uint256[])` imzasından türetilmiştir. Burada `uint` yerine onun kanonik bir gösterimi olan `uint256`'nın kullanıldığını unutmayın.
- `0x0000000000000000000000000000000000000000000000000000000000000060`: argüman bloğunun başlangıcından itibaren bayt cinsinden ölçülen ilk parametrelerinin (dinamik tipteki) veri bölümünün konumu. Bu örnekte, `0x60` tır.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: ikinci parametre: boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0`: üçüncü parametrenin (dinamik tipteki) veri parçasının bayt cinsinden belirlenen konumu. Bu durumda, `0xa0` dır.
- `0x0000000000000000000000000000000000000000000000000000000000000004`: ilk argümanın veri parçası, bayt dizisinin elemanlar cinsinden uzunluğu ile başlar, bu örnekte 4'tür.
- `0x6461766500000000000000000000000000000000000000000000000000000000`: ilk argümanın içeriği: `"dave"` ifadesinin UTF-8 (bu durumda ASCII'ye eşittir) şifrelenmesi, sağdan 32 bayt olacak kadar uzunlukta doldurulur.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: üçüncü bağımsız değişkenin veri bölümü, dizinin eleman cinsinden uzunluğu ile başlar, bu durumda 3'tür.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: üçüncü parametrenin ilk giriş değeri.
- `0x0000000000000000000000000000000000000000000000000000000000000002`: üçüncü parametrenin ikinci giriş değeri.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: üçüncü parametrenin üçüncü giriş değeri.

Toplam olarak:

```
0xa5643bf20000000000000000000000000000000000000000000000000000000000000060000000000000000000000000000000000000000000
```

3.22.9 Dinamik Tiplerin Kullanımı

`f(uint256,uint32[],bytes10,bytes)` imzalı olan bir fonksiyona `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` değerleriyle yapılan herhangi bir çağrı aşağıdaki şekilde şifrelenecektir:

`sha3("f(uint256,uint32[],bytes10,bytes)")` ifadesinin ilk dört baytını, yani `0x8be65246` ifadesini alıyoruz. Daha sonra bu dört argümanın baş kısımlarını şifreliyoruz. Statik `uint256` ve `bytes10` tipleri açısından bunlar doğrudan iletmek istediğimiz değerlerdir, dinamik `uint32[]` ve `bytes` tipleri açısından ise değerlerin şifrelemesinin başlangıcından itibaren ölçülen (yani fonksiyon imzasının özetini içeren ilk dört baytı saymadan önce) veri bölgelerinin başlangıcındaki bayt cinsindeki ofseti kullanırız. Bunlar:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` 32 bayta kadar doldurulmuş)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (ikinci parametrenin veri kısmının baş kısmındaki ofset değeri, 4*32 bayt, tam olarak baş kısmının boyutu kadardır)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` sağda 32 bayta olacak kadar doldurulmuş)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (dördüncü parametrenin veri bölümünde bulunan başlangıç ofseti = birinci dinamik parametrenin veri bölümünde bulunan başlangıç ofseti + birinci dinamik parametrenin veri bölümünün boyutu = 4*32 + 3*32 (aşağı bakınız))

Bundan sonra, ilk dinamik argümanın veri kısmı olan `[0x456, 0x789]` gelir:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (dizinin eleman sayısı, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (ilk eleman)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (ikinci eleman)

Son olarak, ikinci dinamik argümanın veri kısmını şifreliyoruz, `"Hello, world!`:

- `0x000000000000000000000000000000000000000000000000000000000000000d` (eleman sayısı (bu örnekte bayt): 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` ("Hello, world!" sağda 32 bayt olacak kadar doldurulmuş)

Hepsi birlikte, şifreleme (fonksiyon seçiciden sonra satır sonu ve anlaşılabilirlik için her biri 32 bayt) şeklindedir:

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Aynı prensibi `g(uint256[][],string[])` imzalı olan bir fonksiyonun verilerini `([[1, 2], [3]], ["bir", "iki", "üç"])` değerleriyle şifrelemek için uygulayalım, ancak şifreleme işleminin en atomik kısımlarından başlayalım:

İlk olarak, birinci kök dizisinin `[[1, 2], [3]]` birinci gömülü dinamik dizisinin `[1, 2]` uzunluğunu ve verilerini şifreleyeceğiz:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (ilk dizideki eleman sayısı, 2; elemanların kendileri 1 ve 2)
- `0x0000000000000000000000000000000000000000000000000000000000000001` (ilk eleman)
- `0x0000000000000000000000000000000000000000000000000000000000000002` (ikinci eleman)

Ardından, ilk kök dizisinin `[[1, 2], [3]]` ikinci gömülü dinamik dizisinin `[3]` uzunluğunu ve verilerini şifreleyeceğiz:

- `0x0000000000000000000000000000000000000000000000000000000000000001` (ikinci dizideki eleman sayısı, 1; eleman 3 tür)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (ilk eleman)

Daha sonra `[1, 2]` ve `[3]` dinamik dizileri için `a` ve `b` ofsetlerini bulmamız gerekir. Ofsetleri hesaplamak için, şifrelenmiş her satırı numaralandırarak ilk kök dizinin `[[1, 2], [3]]` şifrelenmiş verilerine bakabiliriz:

```
0 - a                                                               - [ 1, 2 ] ofseti
1 - b                                                               - [3] ofseti
2 - 0000000000000000000000000000000000000000000000000000000000000002 - [1, 2] için sayım
3 - 0000000000000000000000000000000000000000000000000000000000000001 - 1 şifrelemesi
4 - 0000000000000000000000000000000000000000000000000000000000000002 - 2 şifrelemesi
5 - 0000000000000000000000000000000000000000000000000000000000000001 - [3] için sayım
6 - 0000000000000000000000000000000000000000000000000000000000000003 - 3 şifrelemesi
```

`a` ofseti, 2. satır (64 bayt) olan `[1, 2]` dizisinin içeriğinin başlangıcına doğru işaret eder; dolayısıyla `a` = `0x0000000000000000000000000000000000000000000000000000000000000040`.

`b` ofseti `[3]` dizisinin içeriğinin başlangıcına işaret eder, bu da 5. satır demektir (160 bayt); dolayısıyla `b` = `0x00000000000000000000000000000000000000000000000000000000000000a0`.

Daha sonra ikinci kök dizisinin gömülü stringlerini şifreleyeceğiz:

- `0x0000000000000000000000000000000000000000000000000000000000000003` (`"one"` kelimesindeki karakter sayısı)
- `0x6f6e650000000000000000000000000000000000000000000000000000000000` (`"one"` kelimesinin utf8 gösterimi)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (`"two"` kelimesindeki karakter sayısı)
- `0x74776f0000000000000000000000000000000000000000000000000000000000` (`"two"` kelimesinin utf8 gösterimi)
- `0x0000000000000000000000000000000000000000000000000000000000000005` (`"three"` kelimesindeki karakter sayısı)
- `0x7468726565000000000000000000000000000000000000000000000000000000` (`"three"` kelimesinin utf8 gösterimi)

İlk kök dizisine paralel olarak, diziler dinamik elemanlar olduğundan, `c`, `d` ve `e` ofsetlerini de bulmamız gerekir:

```
0 - c                                                                - "one" için ofset
1 - d                                                                - "two" için ofset
2 - e                                                                - "three" için ofset
3 - 0000000000000000000000000000000000000000000000000000000000000003 - "one" için sayım
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - "one" şifrelemesi
5 - 0000000000000000000000000000000000000000000000000000000000000003 - "two" için sayım
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - "two" şifrelemesi
7 - 0000000000000000000000000000000000000000000000000000000000000005 - "three" için sayım
8 - 7468726565000000000000000000000000000000000000000000000000000000 - "three"␣
↪şifrelemesi
```

`c` ofseti, 3. Satırda (96 bayt) bulunan `"one"` stringinin içeriğinin başlangıcına işaret eder; dolayısıyla `c` = `0x0000000000000000000000000000000000000000000000000000000000000060`.

`d` ofseti, 5. Satırda (160 bayt) bulunan `"two"` stringinin içeriğinin başlangıcına işaret eder; dolayısıyla `d` = `0x00000000000000000000000000000000000000000000000000000000000000a0`.

`e` ofseti, 7. Satırda (224 bayt) bulunan `"three"` stringinin içeriğinin başlangıcına işaret eder; dolayısıyla `e` = `0x0000000000000000000000000000000000000000000000000000000000000e0`.

Kök dizilerinin ve gömülü öğelerinin şifrelemelerinin birbirine bağlı olmadığını ve `g(string[],uint256[][])` imzalı olan bir fonksiyon için aynı şifrelemelere sahip olduğunu unutmayın.

Daha sonra ilk kök dizisinin uzunluğunu şifreleriz:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (ilk kök dizide bulunan eleman sayısı, 2; elemanların kendileri `[1, 2]` ve `[3]`)

Daha sonra ikinci kök dizisinin uzunluğunu şifreleriz:

- `0x0000000000000000000000000000000000000000000000000000000000000003` (ikinci kök dizisinde bulunan string sayısı, 3; stringlerin kendileri `"one"`, `"two"` ve `"three"`)

Son olarak, ilgili kök dinamik dizileri `[[1, 2], [3]]` ve `[" one", "two", "three"]` için `f` ve `g` ofsetlerini bulur ve parçaları doğru sırada birleştiririz:

```
0x2289b18c                                                                - fonksiyon imzası
 0 - f                                                                    - [[1, 2], [3]]␣
↪için ofset
 1 - g                                                                    - ["one", "two",
↪"three"] için ofset
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - [[1, 2], [3]]␣
↪için sayım
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - [1, 2] için ofset
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - [3] için ofset
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - [1, 2] için sayım
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - 1 şifrelemesi
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - 2 şifrelemesi
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - [3] için sayım
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - 3 şifrelemesi
10 - 0000000000000000000000000000000000000000000000000000000000000003 - ["one", "two",
↪"three"] için sayım
11 - 0000000000000000000000000000000000000000000000000000000000000060 - "one" için ofset
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - "two" için ofset
13 - 0000000000000000000000000000000000000000000000000000000000000e00 - "three" için␣
↪ofset
14 - 0000000000000000000000000000000000000000000000000000000000000003 - "one" için sayım
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - "one" şifrelemesi
16 - 0000000000000000000000000000000000000000000000000000000000000003 - "two" için sayım
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - "two" şifrelemesi
18 - 0000000000000000000000000000000000000000000000000000000000000005 - "three" için␣
↪sayım
19 - 7468726565000000000000000000000000000000000000000000000000000000 - "three"␣
↪şifrelemesi
```

`f` ofseti, 2. satırda(64 bayt) bulunan `[[1, 2], [3]]` dizisinin içeriğinin başlangıcına işaret eder; dolayısıyla `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

`g` ofseti, 10. satırda (320 bayt) bulunan `[" one", "two", "three"]` dizisinin içeriğinin başlangıcına işaret eder; dolayısıyla `g = 0x0000000000000000000000000000000000000000000000000000000000000140`.

3.22.10 Event'ler

Event'ler Ethereum loglama/olay izleme protokolünün bir özetidir. Günlük girdileri sözleşmenin adresini, dört maddeye kadar bir dizi ve bazı değişken uzunluktaki binary verileri sağlar. Event'ler, bunu (bir arayüz spesifikasyonu ile birlikte) uygun şekilde yazılmış bir yapı olarak yorumlamak için mevcut fonksiyon ABI'sinden yararlanır.

Bir event adı ve bir dizi event parametresi verildiğinde, bunları iki alt seriye ayırırız: indekslenenler ve indekslenmeyenler. İndekslenenler (anonim olmayan olaylar için) 3'e veya (anonim olanlar için) 4'e kadar numaralandırılabilir, kayıt girdisinin konu başlıklarını oluşturmak için event imzası Keccak hash'i ile birlikte kullanılır. İndekslenmemiş olanlar ise event'in bayt dizisini oluşturur.

Gerçekte, bu ABI'yi kullanan bir log girdisi şu şekilde açıklanır:

- `address` : sözleşmenin adresi (Ethereum tarafından dahili olarak sağlanır);
- `topics[0]` : `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` verilen bir argümanın kanonik tipini döndüren bir fonksiyondur, örneğin `uint indexed foo` için `uint256` değerini döndürür). Bu değer yalnızca event `anonymous` olarak tanımlanmamışsa `topics[0]` içinde bulunur;

- `topics[n]`: Event `anonymous` olarak tanımlanmamışsa `abi_encode(EVENT_INDEXED_ARGS[n - 1])` veya tanımlanmışsa `abi_encode(EVENT_INDEXED_ARGS[n])` (`EVENT_INDEXED_ARGS` indekslenen `EVENT_ARGS` serisidir);
- `data`: ABI şifrelemesi `EVENT_NON_INDEXED_ARGS` (`EVENT_NON_INDEXED_ARGS` indekslenmemiş `EVENT_ARGS` serisidir, `abi_encode` yukarıda açıklandığı gibi bir fonksiyondan bir dizi typed değer döndürmek için kullanılan ABI şifreleme fonksiyonudur).

En fazla 32 bayt uzunluğundaki tüm türler için, `EVENT_INDEXED_ARGS` dizisi, normal ABI şifrelemesinde olduğu gibi, değeri doğrudan, 32 bayta kadar doldurulmuş veya işaret uzatılmış (işaretli tamsayılar için) olarak içermektedir. Ancak, tüm diziler, `string`, `bytes` ve structlar dahil olmak üzere tüm "karmaşık" tipler veya dinamik uzunluktaki tipler için, `EVENT_INDEXED_ARGS` doğrudan şifrelenmiş değer yerine yerleşik olarak şifrelenmiş özel bir değerin (bkz *İndekslenmiş Event Parametrelerinin Şifrelenmesi*) *Keccak hash*'ini tutacaktır. Bu, uygulamaların dinamik uzunluktaki tiplerin değerlerini verimli bir şekilde sorgulamalarına olanak tanır (şifrelenmiş değerin hash'ini topic olarak ayarlayarak), ancak uygulamaların sorgulamadıkları indekslenmiş değerlerin şifresini çözememelerine imkan tanır. Dinamik uzunluklu tipler için, uygulama geliştiricileri önceden belirlenmiş değerler için hızlı arama (argüman indekslenmişse) ve rastgele değerlerin okunabilirliği (argümanların indekslenmemesini gerektirir) arasında bir trade-off ile karşı karşıyadır. Geliştiriciler, aynı değeri tutması amaçlanan biri indekslenmiş, diğeri indekslenmemiş iki bağımsız değişkene sahip event'ler tanımlayarak bu dengesizliğin üstesinden gelebilir ve hem verimli arama hem de değişken okunabilirlik elde edebileceklerdir.

3.22.11 Error'ler

Bir sözleşme içinde bir hata olması durumunda, sözleşme yürütme işlemini iptal etmek ve tüm durum değişikliklerini geri almak için özel bir opcode kullanabilir. Bu etkilere ek olarak, açıklayıcı veriler de çağırana döndürülebilir. Bu açıklayıcı veri, bir hatanın ve argümanlarının bir fonksiyon çağrısı için veri ile aynı şekilde şifrelenmesidir.

Örnek olarak, `transfer` fonksiyonu her zaman "yetersiz bakiye"(insufficient balance) özel hatası ile geri döndürülen aşağıdaki sözleşmeyi ele alalım:

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

contract TestToken {
    error InsufficientBalance(uint256 available, uint256 required);
    function transfer(address /*to*/, uint amount) public pure {
        revert InsufficientBalance(0, amount);
    }
}
```

Geri döndürülen veri, `InsufficientBalance(uint256,uint256)` fonksiyonuna `InsufficientBalance(0, amount)` fonksiyon çağrısı ile aynı şekilde şifrelenecektir, yani `0xcf479181`, `uint256(0)`, `uint256(amount)`.

`0x00000000` ve `0xffffff` hata(error) selektörleri gelecekte kullanılmak üzere saklanmıştır. "0x00000000" ve "0xffffffff" hata seçicileri ileride kullanılmak üzere ayrılmıştır.

Uyarı: Hata verilerine asla güvenmeyin. Hata verileri standart olarak harici çağrılar zinciri boyunca yayılır; bu da bir sözleşmenin doğrudan çağırdığı sözleşmelerin hiçbirinde tanımlanmamış bir hata alabileceği anlamına gelir. Ayrıca, herhangi bir sözleşme, hata hiçbir yerde tanımlanmamış olsa bile, bir hata imzasıyla eşleşen verileri döndürerek herhangi bir hatayı taklit edebilir.

3.22.12 JSON

Bir sözleşmenin arayüzü için oluşturulan JSON formatı, fonksiyon, olay ve hata açıklamalarından oluşan bir dizi ile verilir. Fonksiyon açıklaması, alanları içeren bir JSON nesnesidir:

- `type`: `"function"`, `"constructor"`, `"receive"` (*"receive Ether" fonksiyonu*) veya `"fallback"` (*"default" fonksiyonu*);
- `name`: fonksiyonun adı;
- `inputs`: her biri aşağıdakileri içeren bir nesne dizisi:
  - `name`: parametrenin adı.
  - `type`: parametrenin kanonik tipi (daha fazla bilgi aşağıdadır).
  - `components`: tuple türleri için kullanılır (daha fazla bilgi aşağıdadır).
- `outputs`: an array of objects similar to `inputs`.
- `stateMutability`: a string with one of the following values: `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` (function does not accept Ether - the default) and `payable` (function accepts Ether).

Constructor ve fallback fonksiyonu asla `name` veya `outputs` içermez. Fallback fonksiyonunda da `inputs` yoktur.

Not: Non-payable fonksiyonuna sıfır olmayan Ether gönderilmesi transferi geri çevirecektir(revert).

Not: Durum değişkenliği `nonpayable` Solidity'de bir durum değişkenliği modifier'ı belirtilmeden yansıtılmaktadır(reflected).

Bir event açıklaması, oldukça benzer özelliklere sahip bir JSON nesnesidir:

- `type`: her zaman `"event"`
- `name`: event adı.
- `inputs`: her biri aşağıdakileri içeren bir nesne dizisi:
  - `name`: the name of the parameter.
  - `type`: parametrenin kanonik tipi (daha fazla bilgi aşağıdadır).
  - `components`: tuple türleri için kullanılır (daha fazla bilgi aşağıdadır).
  - `indexed`: Eğer alan logun konularının bir parçasıysa `true`, logun veri segmentlerinden biriyse `false`.
- `anonymous`: Olay `anonymous` olarak tanımlanmışsa `true`.

Hatalar aşağıdaki gibi görünür:

- `type`: her zaman `"error"`
- `name`: error adı.
- `inputs`: her biri aşağıdakileri içeren bir nesne dizisi:
  - `name`: parametrenin adı.
  - `type`: parametrenin kanonik tipi (daha fazla bilgi aşağıdadır).
  - `components`: tuple türleri için kullanılır (daha fazla bilgi aşağıdadır).

Not: JSON dizisinde aynı ada ve hatta aynı imzaya sahip birden fazla hata(error) olabilir, örneğin hatalar akıllı sözleşmedeki farklı dosyalardan kaynaklanıyorsa veya başka bir akıllı sözleşmeden referans alınıyorsa. ABI için hatanın nerede tanımlandığı değil, yalnızca adı önemlidir.

Örneğin,

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


contract Test {
    constructor() { b = hex"12345678901234567890123456789012"; }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    error InsufficientBalance(uint256 available, uint256 required);
    function foo(uint a) public { emit Event(a, b); }
    bytes32 b;
}
```

JSON ile sonuçlanacaktır:

```
[{
"type":"error",
"inputs": [{"name":"available","type":"uint256"},{"name":"required","type":"uint256"}],
"name":"InsufficientBalance"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Tuple tiplerinin kullanılması

İsimler bilinçli olarak ABI şifrelemesinin bir parçası olmamasına rağmen, son kullanıcıya gösterilmesini sağlamak için JSON'a dahil edilmeleri çok önemlidir. Yapı aşağıdaki şekilde iç içe geçmiştir:

`name`, `type` ve potansiyel olarak `components` üyelerine sahip bir nesne, tiplendirilmiş bir değişkeni tanımlar. Kanonik tip, bir tuple tipine ulaşılana kadar belirlenir ve o noktaya kadar olan dize açıklaması `tuple` kelimesiyle `type` önekinde saklanır, yani `tuple` ve ardından `[]` ve `[k]` tamsayıları `k` ile bir dizi olacaktır. Tuple`ın bileşenleri daha sonra dizi tipinde olan ve üst düzey nesne ile aynı yapıya sahip olan `components` üyesinde saklanır, ancak `indexed` öğesine bu durumda izin verilmez.

Örnek olarak, kod

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.5 <0.9.0;
pragma abicoder v2;

contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory, T memory, uint) public pure {}
    function g() public pure returns (S memory, T memory, uint) {}
}
```

JSON ile sonuçlanacaktır:

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
              }
            ]
          }
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

3.22.13 Katı Şifreleme Modu

Sıkı şifreleme modu, yukarıdaki resmi spesifikasyonda tanımlandığı gibi tam olarak aynı şifrelemeye neden olan moddur. Bu, ofsetlerin veri alanlarında çakışma yaratmadan mümkün olduğunca küçük olması gerektiği ve dolayısıyla hiçbir boşluğa izin verilmediği anlamına gelir.

Genellikle, ABI şifre çözücüler sadece ofset işaretçilerini takip ederek basit bir şekilde yazılır, ancak bazı şifre çözücüler katı modu zorlayabilir. Solidity ABI şifre çözücü şu anda katı modu kullanmayı zorunlu kılmaz, ancak şifreleyici her zaman katı modda veri oluşturur.

3.22.14 Standart Olmayan Paket Modu

Solidity, `abi.encodePacked()` aracılığıyla standart olmayan bir paketlenmiş modu destekler:

- 32 bayttan kısa tipler, doldurma(padding) veya işaret(sign) uzantısı olmadan doğrudan birleştirilir
- dinamik tipler in-place olarak ve uzunluk olmadan şifrelenir
- dizi elemanları doldurulur, ancak yine de in-place olarak şifrelenir

Ayrıca, struct'ların yanı sıra iç içe diziler de desteklenmez.

Örnek olarak, `int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world!")` şeklinde bir şifreleme elde edilir:

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                                int16(-1)
      ^^                              bytes1(0x42)
        ^^^^                          uint16(0x03)
            ^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") without a length field
```

Daha spesifik olarak:

- Şifreleme sırasında her şey in-place olarak şifrelenir. Bu, ABI şifrelemesi gibi baş ve kuyruk arasında bir ayrım olmadığı ve bir dizinin uzunluğunun şifrelenmediği anlamına gelir.
- `abi.encodePacked` komutunun doğrudan argümanları, dizi (veya `string` veya `bytes`) olmadıkları sürece doldurma olmadan şifrelenir.
- Bir dizinin şifrelemesi, elemanlarının şifrelemelerinin **doldurma** ile birleştirilmesidir.
- `string`, `bytes` veya `uint[]` gibi dinamik olarak boyutlandırılan tipler uzunluk alanları olmadan şifrelenir.
- Bir dizinin veya struct'ın parçası olmadığı sürece `string` veya `bytes` şifrelemesinin sonuna doldurma uygulanmaz (bu durumda 32 baytın katlarına kadar doldurulur).

Genel olarak, eksik uzunluk değeri nedeniyle dinamik olarak boyutlandırılmış iki öğe olduğu anda şifreleme belirsizleşir.

Doldurma gerekiyorsa, açık tip dönüşümleri kullanılabilir: `abi.encodePacked(uint16(0x12)) == hex "0012"`.

Fonksiyonları çağırırken paketlenmiş şifreleme kullanılmadığından, bir fonksiyon seçicinin önüne ekleme yapmak için özel bir destek yoktur. Şifreleme belirsiz olduğundan, şifre çözme fonksiyonu yoktur.

Uyarı: Eğer `keccak256(abi.encodePacked(a, b))` kullanırsanız ve hem `a` hem de `b` dinamik tiplerse, `a``nın bazı kısımlarını ``b``ye taşıyarak veya tam tersini yaparak hash değerinde çakışmalar oluşturmak kolaydır. Daha spesifik olarak, ``abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. İmzalar, kimlik doğrulama veya veri bütünlüğü için `abi.encodePacked` kullanıyorsanız, her zaman aynı tipleri kullandığınızdan emin olun ve bunlardan en fazla birinin dinamik olduğunu kontrol edin. Mecburi bir neden olmadıkça, `abi.encode` tercih edilmelidir.

3.22.15 İndekslenmiş Event Parametrelerinin Şifrelenmesi

Değer türü olmayan indekslenmiş event parametreleri, yani diziler ve struct'lar doğrudan saklanmaz, bunun yerine bir şifrelemenin keccak256-hash'i saklanır. Bu şifreleme aşağıdaki gibi tanımlanmaktadır:

- bir `bytes` ve `string` değerinin şifrelenmesi, herhangi bir doldurma veya uzunluk öneki olmaksızın sadece string içeriğinden ibarettir.
- bir structın kodlaması, her zaman 32 baytın katları olacak şekilde (`bytes` ve `string` bile) üyelerinin şifrelemelerinin bir araya getirilmesiyle elde edilir.
- bir dizinin şifrelenmesi (hem dinamik hem de statik olarak boyutlandırılmış), her zaman 32 baytın katları olacak şekilde (`bytes` ve `string` bile) ve herhangi bir uzunluk öneki olmadan elemanlarının şifrelenmesinin birleşimidir

Yukarıda her zamanki gibi, negatif bir sayı işaret uzantısıyla doldurulur ve sıfırla doldurulmaz. `bytesNN` tipleri sağdan, `uintNN` / `intNN` tipleri ise soldan doldurulur.

Uyarı: Bir struct'ın şifrelenmesi, dinamik olarak boyutlandırılmış birden fazla dizi içeriyorsa belirsizdir. Bu nedenle, event verilerini her zaman yeniden kontrol edin ve yalnızca indekslenmiş parametrelere dayanan arama sonuçlarına güvenmeyin.

3.23 Solidity v0.5.0 İşleyişi Bozan Değişiklikler

Bu bölüm, Solidity 0.5.0 sürümünde getirilen değişikliklerin, eski sürümlerdeki ana işleyişi bozan kısımlarını değişikliklerin arkasındaki gerekçeleri ve etkilenen kodun nasıl güncelleneceğini vurgular. Tam liste için sürüm değişiklik günlüğü adresini kontrol edin.

Not: Solidity v0.5.0 ile derlenen sözleşmeler, eski sürümlerle derlenen sözleşmelerle ve hatta kütüphanelerle yeniden derlenmeden veya yeniden dağıtılmadan arayüz oluşturmaya devam edebilir. Arayüzleri veri konumlarını, görünürlük ve değişebilirlik belirleyicilerini içerecek şekilde değiştirmek yeterlidir. Aşağıdaki *Interoperability With Older Contracts* bölümüne bakınız.

3.23.1 Yalnızca Anlamsal (Semantik) Değişiklikleri

Bu bölümde yalnızca semantik olan, dolayısıyla mevcut kodda yeni ve farklı davranışları gizleme potansiyeli olan değişiklikler listelenmektedir.

- İşaretli sağa kaydırma artık uygun aritmetik kaydırma kullanır, yani sıfıra doğru yuvarlamak yerine negatif sonsuza doğru yuvarlar. İşaretli ve işaretsiz kaydırma Constantinople'da özel işlem kodlarına sahip olacak ve şu an için Solidity tarafından taklit edilmektedir.
- Bir `do...while` döngüsündeki `continue` deyimi artık bu tür durumlarda yaygın davranış olan koşula atlıyor. Eskiden döngü gövdesine atlıyordu. Böylece, koşul yanlışsa, döngü sonlandırılır.
- `.call()`, `.delegatecall()` ve `.staticcall()` fonksiyonları, tek bir `bytes` parametresi verildiğinde artık dolgu yapmıyor.
- Pure ve view fonksiyonları artık EVM sürümü Byzantium veya üstü ise `CALL` yerine `STATICCALL` opcode`u kullanılarak çağrılmaktadır. Bu, EVM düzeyinde durum değişikliklerine izin vermez.
- ABI kodlayıcı artık harici fonksiyon çağrılarında ve `abi.encode` içinde kullanıldığında çağrı verilerinden (`msg.data` ve harici fonksiyon parametreleri) bayt dizilerini ve dizeleri düzgün bir şekilde doldurur. Dolgusuz kodlama için `abi.encodePacked` kullanın.
- ABI dekoderi, fonksiyonların başında ve `abi.decode()` içinde, aktarılan calldata çok kısaysa veya sınırların dışına işaret ediyorsa geri döner. Yüksek dereceli kirli bitlerin hala basitçe göz ardı edildiğini unutmayın.
- Tangerine Whistle'dan başlayarak harici fonksiyon çağrıları ile mevcut tüm gazı iletin.

3.23.2 Semantik ve Sentaktik Değişiklikler

Bu bölümde sözdizimi ve anlambilimi etkileyen değişiklikler vurgulanmaktadır.

- `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` ve `ripemd160()` fonksiyonları artık sadece tek bir `bytes` argümanı kabul etmektedir. Ayrıca, argüman doldurulmamıştır. Bu, argümanların nasıl birleştirildiğini daha açık ve net hale getirmek için değiştirildi. Her `.call()` (ve ailesi) `.call("")` olarak ve her `.call(signature, a, b, c)` `.call(abi.encodeWithSignature(signature, a, b, c))` olarak değiştirildi (sonuncusu yalnızca değer türleri için çalışır). Her `keccak256(a, b, c)` ifadesini `keccak256(abi.encodePacked(a, b, c))` olarak değiştirin. İşleyişi bozan bir değişiklik olmasa da, geliştiricilerin `x.call(bytes4(keccak256("f(uint256)"), a, b)` öğesini `x.call(abi.encodeWithSignature("f(uint256)", a, b))` olarak değiştirmeleri önerilir.
- Geri dönüş verilerine erişim sağlamak için `.call()`, `.delegatecall()` ve `.staticcall()` fonksiyonları artık `(bool, bytes memory)` döndürmektedir. `bool success = otherContract.call("f")` ifadesini `(bool success, bytes memory data) = otherContract.call("f")` olarak değiştirin.
- Solidity artık fonksiyon yerel değişkenleri için C99 tarzı kapsam kurallarını uygulamaktadır, yani değişkenler yalnızca bildirildikten sonra ve yalnızca aynı veya iç içe kapsamlarda kullanılabilir. Bir `for` döngüsünün başlatma bloğunda bildirilen değişkenler, döngü içindeki herhangi bir noktada geçerlidir.

3.23.3 Açıklık Gereksinimleri

Bu bölüm, kodun artık daha açık olması gereken değişiklikleri listeler. Konuların çoğu için derleyici öneriler sağlayacaktır.

- Açık fonksiyon görünürlüğü artık zorunludur. Her fonksiyona ve constructor'a `public` ve görünürlüğünü zaten belirtmeyen her fallback veya arayüz fonksiyonuna `external` ekleyin.
- struct, array veya mapping türlerindeki tüm değişkenler için açık veri konumu artık zorunludur. Bu aynı zamanda fonksiyon parametrelerine ve dönüş değişkenlerine de uygulanır. Örneğin, `uint[] x = z` ifadesini `uint[] storage x = z` olarak ve `function f(uint[][] x)` ifadesini `function f(uint[][] memory x)` olarak değiştirin; burada `memory` veri konumudur ve uygun şekilde `storage` veya `calldata` ile değiştirilebilir. `external` fonksiyonlarının `calldata` veri konumuna sahip parametreler gerektirdiğini unutmayın.
- Sözleşme türleri, isim alanlarını ayırmak için artık `address` üyelerini içermemektedir. Bu nedenle, artık bir `address` üyesini kullanmadan önce sözleşme türünün değerlerini açıkça adreslere dönüştürmek gerekmektedir. Örnek: `c` bir sözleşme ise, `c.transfer(...)` değerini `address(c).transfer(...)` olarak ve `c.balance` değerini `address(c).balance` olarak değiştirin.
- İlişkisiz sözleşme türleri arasında açık dönüşümlere artık izin verilmemektedir. Bir sözleşme türünden yalnızca temel veya ata türlerinden birine dönüştürebilirsiniz. Bir sözleşmenin, miras almamasına rağmen dönüştürmek istediğiniz sözleşme türüyle uyumlu olduğundan eminseniz, önce `address` türüne dönüştürerek bunu aşabilirsiniz. Örnek: `A` ve `B` sözleşme türleriyse, `B` `A` türünden miras almıyorsa ve `b` `B` türünde bir sözleşmeyse, `A(adres(b))` kullanarak `b` türünü `A` türüne dönüştürebilirsiniz. Aşağıda açıklandığı gibi, eşleşen payable fallback fonksiyonlarına dikkat etmeniz gerektiğini unutmayın.
- `address` türü `address` ve `address payable` olarak ikiye ayrılmıştır, burada sadece `address payable` `transfer` fonksiyonunu sağlamaktadır. Bir `address payable` doğrudan bir `address` e dönüştürülebilir, ancak bunun tersine izin verilmez. `address`'i `address payable`'a dönüştürmek `uint160` vasıtasıyla dönüşüm yoluyla mümkündür. Eğer `c` bir sözleşme ise, `address(c)` sadece `c` bir payable fallback fonksiyonuna sahipse `address payable` ile sonuçlanır. Eğer *withdraw pattern* kullanıyorsanız, büyük olasılıkla kodunuzu değiştirmeniz gerekmez çünkü `transfer` saklanan adresler yerine sadece `msg.sender` üzerinde kullanılır ve `msg.sender` bir `address payable` dır.
- Farklı boyuttaki `bytesX` ve `uintY` arasındaki dönüşümler, sağdaki `bytesX` dolgusu ve soldaki `uintY` dolgusu nedeniyle artık izin verilmiyor ve bu da beklenmedik dönüşüm sonuçlarına neden olabilir. Boyut artık dönüştürmeden önce tür içinde ayarlanmalıdır. Örneğin, `bytes4` (4 bayt) değişkenini önce `bytes8` değişkenine ve

ardından `uint64` değişkenine dönüştürerek bir `bytes4` (4 bayt) değişkenini bir `uint64` (8 bayt) değişkenine dönüştürebilirsiniz. `uint32` üzerinden dönüştürme yaparken ters dolgu elde edersiniz. v0.5.0`dan önce `bytesX` ve `uintY` arasındaki herhangi bir dönüşüm `uint8X` üzerinden giderdi. Örneğin `uint8(bytes3(0x291807))`, `uint8(uint24(bytes3(0x291807)))`'e dönüştürülürdü (sonuç `0x07` dir).

- Payable olmayan fonksiyonlarda `msg.value` kullanımına (veya bir modifier aracılığıyla tanıtılmasına) güvenlik özelliği olarak izin verilmez. Fonksiyonu `payable` haline getirin veya `msg.value` kullanan program mantığı için yeni bir dahili fonksiyon oluşturun.
- Anlaşılabilirlik nedeniyle, standart girdi kaynak olarak kullanıldığında komut satırı arayüzü artık - gerektirmektedir. Translated with www.DeepL.com/Translator (free version)

3.23.4 Kullanımdan Kaldırılan Öğeler

Bu bölümde, önceki özellikleri veya sözdizimini kullanımdan kaldıran değişiklikler listelenmektedir. Bu değişikliklerin çoğunun `v0.5.0` deneysel modunda zaten etkin olduğunu unutmayın.

Komut Satırı ve JSON Arayüzleri

- Komut satırı seçeneği `--formal` (daha fazla biçimsel doğrulama için Why3 çıktısı oluşturmak için kullanılır) kullanımdan kaldırılmıştır ve artık silinmektedir. Yeni bir biçimsel doğrulama modülü olan SMTChecker, `pragma experimental SMTChecker;` ile etkinleştirilmiştir.
- Komut satırı seçeneği `--julia`, ara dil `Julia``nın ``Yul` olarak yeniden adlandırılması nedeniyle `--yul` olarak yeniden adlandırıldı.
- `--clone-bin` ve `--combined-json clone-bin` komut satırı seçenekleri kaldırıldı.
- Boş önek içeren yeniden eşlemelere izin verilmiyor.
- JSON AST alanları `constant` ve `payable` kaldırıldı. Bu bilgiler artık `stateMutability` alanında bulunmaktadır.
- `FunctionDefinition` node'unun JSON AST alanı `isConstructor`, `"constructor"`, `"fallback"` veya `"function"` değerine sahip olabilen `kind` adlı bir alanla değiştirildi.
- Bağlantısız ikili hex dosyalarında, kütüphane adres yer tutucuları artık `$...$` ile çevrelenmiş tam nitelikli kütüphane adının keccak256 hash'inin ilk 36 hex karakteridir. Önceden, sadece tam nitelikli kütüphane adı kullanılıyordu. Bu, özellikle uzun yollar kullanıldığında çakışma olasılığını azaltır. Binary dosyalar artık bu yer tutuculardan tam nitelikli adlara bir eşleme listesi de içeriyor.

Constructor'lar

- Constructor'lar artık `constructor` anahtar sözcüğü kullanılarak tanımlanmalıdır.
- Temel constructor'ların parantezler olmadan çağrılmasına artık izin verilmemektedir.
- Aynı kalıtım hiyerarşisinde temel constructor argümanlarının birden fazla kez belirtilmesine artık izin verilmemektedir.
- Argümanları olan ancak argüman sayısı yanlış olan bir constructor çağrılmasına artık izin verilmemektedir. Argüman vermeden yalnızca bir kalıtım ilişkisi belirtmek istiyorsanız, parantezleri hiç sağlamayın.

Fonksiyonlar

- Fonksiyon `callcode` artık izin verilmiyor (`delegatecall` lehine). Inline assembly ile kullanmak hala mümkündür.
- `suicide` artık izin verilmiyor (`selfdestruct` lehine).
- `sha3` artık izin verilmiyor (`keccak256` lehine).
- `throw` artık izin verilmiyor (`revert`, `require` ve `assert` lehine).

Dönüşümler

- Ondalık değişmezlerden `bytesXX` türlerine açık ve örtük dönüşümlere artık izin verilmiyor.
- Onaltılık değişmezlerden farklı boyuttaki `bytesXX` türlerine açık ve örtük dönüşümlere artık izin verilmiyor.

Literaller ve Sonekler

- Artık yıllarla ilgili karmaşıklıklar ve karışıklıklar nedeniyle `years` birim gösterimine artık izin verilmemektedir.
- Bir sayı tarafından takip edilmeyen sondaki noktalara artık izin verilmemektedir.
- Onaltılık sayıların birim değerleriyle birleştirilmesine (örneğin `0x1e wei`) artık izin verilmemektedir.
- Onaltılık sayılar için `0X` önekine izin verilmez, sadece `0x` mümkündür.

Değişkenler

- Anlaşılabilirlik için boş structların tanımlanmasına artık izin verilmiyor.
- `var` anahtar sözcüğüne artık netlik için izin verilmiyor.
- Farklı sayıda bileşene sahip tuple'lar arasındaki atamalara artık izin verilmiyor.
- Derleme zamanı sabitleri olmayan sabitler için değerlere izin verilmez.
- Uyumsuz sayıda değere sahip çok değişkenli bildirimlere artık izin verilmemektedir.
- Başlatılmamış depolama değişkenlerine artık izin verilmemektedir.
- Boş tuple bileşenlerine artık izin verilmiyor.
- Değişkenler ve struct'lardaki döngüsel bağımlılıkların algılanması özyinelemede 256 ile sınırlandırılmıştır.
- Uzunluğu sıfır olan sabit boyutlu dizilere artık izin verilmemektedir.

Sözdizimi

- Fonksiyon durumu değişebilirlik değiştiricisi olarak `constant` kullanımına artık izin verilmemektedir.
- Boolean ifadeler aritmetik işlemler kullanamaz.
- Unary + operatörüne artık izin verilmiyor.
- Harfler artık önceden açık bir türe dönüştürülmeden `abi.encodePacked` ile kullanılamaz.
- Bir veya daha fazla dönüş değeri olan fonksiyonlar için boş dönüş ifadelerine artık izin verilmemektedir.
- " loose assembly" sözdizimine artık tamamen izin verilmiyor, yani atlama etiketleri, atlamalar ve işlevsel olmayan talimatlar artık kullanılamaz. Bunun yerine yeni `while`, `switch` ve `if` yapılarını kullanın.

- Uygulaması olmayan fonksiyonlar artık modifier kullanamaz.
- Adlandırılmış dönüş değerlerine sahip fonksiyon tiplerine artık izin verilmemektedir.
- Blok olmayan if/while/for gövdeleri içindeki tek deyimli değişken bildirimlerine artık izin verilmiyor.
- Yeni anahtar kelimeler: `calldata` ve `constructor`.
- Yeni ayrılmış anahtar sözcükler: `alias`, `apply`, `auto`, `copyof`, `define`, `immutable`, `implements`, `macro`, `mutable`, `override`, `partial`, `promise`, `reference`, `sealed`, `sizeof`, `supports`, `typedef` ve `unchecked`.

3.23.5 Eski Sözleşmelerle Birlikte Çalışabilirlik

Solidity'nin v0.5.0'dan önceki sürümleri için yazılmış sözleşmeler için arayüzler tanımlayarak (veya tam tersi şekilde) arayüz oluşturmak hala mümkündür. Aşağıdaki 0.5.0 öncesi sözleşmenin zaten dağıtılmış olduğunu düşünün:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// Bu, derleyicinin 0.4.25 sürümüne kadar bir uyarı bildirecektir
// Bu 0.5.0'dan sonra derlenmeyecektir
contract OldContract {
    function someOldFunction(uint8 a) {
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

Bu artık Solidity v0.5.0 ile derlenmeyecektir. Ancak, bunun için uyumlu bir arayüz tanımlayabilirsiniz:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

Orijinal sözleşmede `constant` olarak tanımlanmasına rağmen `anotherOldFunction` fonksiyonunu `view` olarak tanımlamadığımıza dikkat edin. Bunun nedeni Solidity v0.5.0`dan itibaren `view` fonksiyonlarını çağırmak için `staticcall` kullanılmasıdır. v0.5.0 öncesinde `constant` anahtar sözcüğü zorunlu değildi, bu nedenle `constant` olarak bildirilen bir fonksiyonu `staticcall` ile çağırmak yine de geri dönebilir, çünkü `constant` fonksiyonu hala depolamayı değiştirmeye çalışabilir. Sonuç olarak, eski sözleşmeler için bir arayüz tanımlarken, `constant` yerine sadece fonksiyonun `staticcall` ile çalışacağından kesinlikle emin olduğunuz durumlarda `view` kullanmalısınız.

Yukarıda tanımlanan arayüz göz önüne alındığında, artık halihazırda dağıtılmış olan 0.5.0 öncesi sözleşmeyi kolayca kullanabilirsiniz:

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

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
}

contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

Benzer şekilde, 0.5.0 öncesi kütüphaneler, kütüphanenin fonksiyonları uygulanmadan tanımlanarak ve linking sırasında 0.5.0 öncesi kütüphanenin adresi verilerek kullanılabilir (linking için komut satırı derleyicisinin nasıl kullanılacağını öğrenmek için *Komut Satırı Derleyicisinin Kullanımı* bölümüne bakınız):

```
// This will not compile after 0.6.0
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

3.23.6 Örnek

Aşağıdaki örnekte bir sözleşme ve bu bölümde listelenen bazı değişikliklerle Solidity v0.5.0 için güncellenmiş sürümü gösterilmektedir.

Eski versiyon:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// Bu 0.5.0'dan sonra derlenmeyecektir

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // Fonksiyon değişebilirliği sağlanmadı, hata değil.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    function someInteger() internal returns (uint) { return 2; }

    // Fonksiyon görünürlüğü sağlanmadı, hata değil.
    // Fonksiyon değişebilirliği sağlanmadı, hata değil.
    function f(uint x) returns (bytes) {
        // Var bu versiyonda sorunsuz çalışıyor.
        var z = someInteger();
        x += z;
        // Throw bu versiyonda sorunsuz çalışıyor.
        if (x > 100)
            throw;
        bytes memory b = new bytes(x);
        y = -3 >> 1;
        // y == -1 (yanlış, -2 olmalı)
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' sonsuz döngüye neden olur.
        } while (x < 11);
        // Çağrı yalnızca bir Bool döndürür.
        bool success = address(other).call("f");
        if (!success)
            revert();
        else {
            // Yerel değişkenler kullanımlarından sonra bildirilebilir.
            int y;
        }
        return b;
    }

    // 'arr' için açık bir veri konumuna gerek yok
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
        otherContract.transfer(1 ether);

        // uint32 (4 bayt) bytes8'den (8 bayt) daha küçük olduğundan,
        // x'in ilk 4 baytı kaybolacaktır. Bu durum, bytesX sağa doğru
        // doldurulduğundan beklenmedik davranışlara yol açabilir.
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

Yeni versiyon:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;
// Bu 0.6.0'dan sonra derlenmeyecektir

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    }
    function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Fonksiyon değişebilirliği belirtilmelidir.
    function someInteger() internal pure returns (uint) { return 2; }

    // Fonksiyon görünürlüğü belirtilmelidir.
    // Fonksiyon değişebilirliği belirtilmelidir.
    function f(uint x) public returns (bytes memory) {
        // Tür şimdi açıkça verilmelidir.
        uint z = someInteger();
        x += z;
        // Throw'a artık izin verilmiyor.
        require(x <= 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' ile aşağıdaki koşula atlanır.
        } while (x < 11);

        // Çağrı (bool, bayt) döndürür.
        // Veri konumu belirtilmelidir.
        (bool success, bytes memory data) = address(other).call("f");
        if (!success)
            revert();
        return data;
    }

    using AddressMakePayable for address;
    // 'arr' için veri konumu belirtilmelidir
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract, address␣
↪unknownContract) public payable {
        // 'otherContract.transfer' sağlanmamıştır.
        // 'OtherContract' kodu bilindiğinden ve fallback fonksiyonuna sahip olduğundan,
        // address(otherContract) 'address payable' tipine sahiptir.
        address(otherContract).transfer(1 ether);

        // 'unknownContract.transfer' sağlanmadı.
        // 'address(unknownContract).transfer',
        // 'address(unknownContract)' 'address payable' olmadığı için sağlanmamıştır.
        // Fonksiyon para göndermek istediğiniz bir 'address' alırsa,
        // bunu 'uint160' aracılığıyla 'address payable'a dönüştürebilirsiniz.
        // Not: Bu tavsiye edilmez ve mümkün olduğunda açık
        // 'address payable' türü kullanılmalıdır.
        // Anlaşılabilirliği artırmak için, dönüşüm işleminde bir
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        // kütüphane kullanılmasını öneriyoruz (bu örnekte sözleşmeden sonra␣
↪verilmiştir).
        address payable addr = unknownContract.makePayable();
        require(addr.send(1 ether));

        // uint32 (4 bayt), bytes8'den (8 bayt) daha küçük
        // olduğu için dönüştürmeye izin verilmez.
        // Önce ortak bir boyuta dönüştürmemiz gerekiyor:
        bytes4 x4 = bytes4(x); // Dolgu sağ tarafta gerçekleşir
        uint32 y = uint32(x4); // Dönüşüm tutarlıdır
        // 'msg.value' bir 'non-payable' fonksiyonunda kullanılamaz.
        // Fonksiyonu ödenebilir hale getirmemiz gerekiyor
        myNumber += y + msg.value;
    }
}

// Geçici bir çözüm olarak ``address`` i açıkça
// ``address payable`` a dönüştürmek için bir kütüphane tanımlayabiliriz.
library AddressMakePayable {
    function makePayable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

3.24 Solidity v0.6.0 İşleyişi Bozan Değişiklikler

Bu bölüm, Solidity 0.6.0 sürümünde getirilen ana işleyişi bozan değişiklikleri, değişikliklerin arkasındaki gerekçeleri ve etkilenen kodun nasıl güncelleneceğini vurgular. Tam liste için sürüm değişiklik günlüğü adresini kontrol edin.

3.24.1 Derleyicinin Uyaramayabileceği Değişiklikler

Bu bölümde, kodunuzun davranışının derleyici size haber vermeden değişebileceği değişiklikler listelenmektedir.

- Bir üs alma işleminin sonuç türü tabanın türüdür. Simetrik işlemlerde olduğu gibi, hem tabanın türünü hem de üssün türünü tutabilen en küçük tür olarak kullanılır. Ayrıca, üs alma işleminin tabanı için işaretli türlere izin verilir.

3.24.2 Açıklık Gereksinimleri

Bu bölüm, kodun artık daha açık olması gereken, ancak anlamın değişmediği değişiklikleri listeler. Konuların çoğu için derleyici öneriler sağlayacaktır.

- Fonksiyonlar artık yalnızca `virtual` anahtar sözcüğü ile işaretlendiklerinde veya bir arayüzde tanımlandıklarında geçersiz kılınabilir. Bir arayüz dışında uygulaması olmayan fonksiyonlar `virtual` olarak işaretlenmelidir. Bir fonksiyon veya modifier geçersiz kılınırken, yeni `override` anahtar sözcüğü kullanılmalıdır. Birden fazla paralel tabanda tanımlanmış bir fonksiyon veya modifier geçersiz kılınırken, tüm tabanlar anahtar kelimeden sonra parantez içinde aşağıdaki gibi listelenmelidir: `override(Base1, Base2)`.
- Dizilerin `length` öğesine üye erişimi artık depolama dizileri için bile her zaman salt okunurdur. Depolama dizilerini uzunluklarına yeni bir değer atayarak yeniden boyutlandırmak artık mümkün değildir. Bunun yerine

`push()`, `push(value)` veya `pop()` kullanın ya da tam bir dizi atayın, bu da elbette mevcut içeriğin üzerine yazacaktır. Bunun arkasındaki neden, devasa depolama dizilerinin depolama çakışmalarını önlemektir.

- Yeni anahtar kelime `abstract` sözleşmeleri soyut olarak işaretlemek için kullanılabilir. Bir sözleşme tüm fonksiyonlarını uygulamıyorsa kullanılmalıdır. Soyut sözleşmeler `new` operatörü kullanılarak oluşturulamaz ve derleme sırasında bunlar için bytecode üretmek mümkün değildir.
- Kütüphaneler sadece internal olanları değil, tüm fonksiyonlarını uygulamak zorundadır.
- Inline assembly'de bildirilen değişkenlerin adları artık `_slot` veya `_offset` ile bitemez.
- Inline assembly'deki değişken bildirimleri artık inline assembly bloğunun dışındaki herhangi bir bildirimi gölgeleyemez. İsim bir nokta içeriyorsa, noktaya kadar olan öneki, inline assembly bloğu dışındaki herhangi bir bildirimle çakışmayabilir.
- Durum değişkeni gölgelemesine artık izin verilmemektedir. Türetilmiş bir sözleşme, yalnızca tabanlarının hiçbirinde aynı ada sahip görünür bir durum değişkeni yoksa `x` durum değişkenini bildirebilir.

3.24.3 Semantik ve Sentaktik Değişiklikler

Bu bölüm, kodunuzu değiştirmeniz gereken ve daha sonra başka bir şey yapan değişiklikleri listeler.

- External fonksiyon tiplerinden `address` e dönüşümlere artık izin verilmiyor. Bunun yerine harici fonksiyon tipleri, mevcut `selector` üyesine benzer şekilde `address` adlı bir üyeye sahiptir.
- Dinamik depolama dizileri için `push(value)` fonksiyonu artık yeni uzunluğu döndürmüyor (hiçbir şey döndürmüyor).
- Genellikle " fallback fonksiyonu" olarak adlandırılan isimsiz fonksiyon, `fallback` anahtar kelimesi kullanılarak tanımlanan yeni bir fallback fonksiyonuna ve `receive` anahtar kelimesi kullanılarak tanımlanan bir receive ether fonksiyonuna bölünmüştür.
  - Mevcutsa, çağrı verisi boş olduğunda (ether alınsın ya da alınmasın) ether alma fonksiyonu çağrılır. Bu fonksiyon örtük olarak `payable` dır.
  - Yeni fallback fonksiyonu, başka hiçbir fonksiyon uyuşmadığında çağrılır (eğer receive ether fonksiyonu mevcut değilse, bu boş çağrı verisine sahip çağrıları da içerir). Bu fonksiyonu `payable` yapabilir ya da yapmayabilirsiniz. Eğer `payable` değilse, değer gönderen başka bir fonksiyonla eşleşmeyen işlemler geri dönecektir. Yeni fallback fonksiyonunu yalnızca bir yükseltme veya proxy modelini takip ediyorsanız uygulamanız gerekir.

3.24.4 Yeni Özellikler

Bu bölümde Solidity 0.6.0 öncesinde mümkün olmayan veya başarılması daha zor olan şeyler listelenmektedir.

- ref:*try/catch deyimi* <*try-catch*> başarısız external çağrılara tepki vermenizi sağlar.
- `struct` ve `enum` türleri dosya düzeyinde bildirilebilir.
- Dizi dilimleri calldata dizileri için kullanılabilir, örneğin `abi.decode(msg.data[4:], (uint, uint))` fonksiyon çağrısı yükünün kodunu çözmenin düşük seviyeli bir yoludur.
- Natspec, geliştirici belgelerinde `@param` ile aynı adlandırma kontrolünü uygulayarak birden fazla dönüş parametresini destekler.
- Yul ve Inline Assembly, mevcut fonksiyondan çıkan `leave` adlı yeni bir deyime sahiptir.
- `address`'den `address payable`'a dönüşümler artık `payable(x)` ile mümkündür, burada `x` `address` tipinde olmalıdır.

3.24.5 Arayüz Değişiklikleri

Bu bölümde, dilin kendisiyle ilgili olmayan ancak derleyicinin arayüzleri üzerinde etkisi olan değişiklikler listelenmektedir. Bunlar derleyiciyi komut satırında nasıl kullandığınızı, programlanabilir arayüzünü nasıl kullandığınızı veya derleyici tarafından üretilen çıktıyı nasıl analiz ettiğinizi değiştirebilir.

Yeni Hata Raporlayıcısı

Komut satırında daha erişilebilir hata mesajları üretmeyi amaçlayan yeni bir hata raporlayıcı tanıtıldı. Öntanımlı olarak etkindir, ancak `--old-reporter` geçildiğinde kullanımdan kaldırılmış eski hata raporlayıcısına geri dönülür.

Metadata Hash Seçenekleri

Derleyici artık metadata dosyasının IPFS hash'ini varsayılan olarak bytecode'un sonuna ekliyor (ayrıntılar için *contract metadata* belgesine bakın). 0.6.0'dan önce derleyici varsayılan olarak Swarm hash'ini ekliyordu ve bu davranışı desteklemeye devam etmek için yeni komut satırı seçeneği `--metadata-hash` tanıtıldı. Bu, `--metadata-hash` komut satırı seçeneğine değer olarak `ipfs` veya `swarm` değerlerinden birini geçirerek üretilecek ve eklenecek hash'i seçmenize olanak tanır. `none` değerinin geçilmesi hash'i tamamen kaldırır.

Bu değişiklikler *Standard JSON Interface* aracılığıyla da kullanılabilir ve derleyici tarafından oluşturulan metadata JSON'u etkiler.

Metadata'ları okumak için önerilen yol, CBOR şifrelemesinin uzunluğunu belirlemek için son iki baytı okumak ve *metadata section* bölümünde açıklandığı gibi bu veri bloğu üzerinde uygun bir şifre çözme işlemi gerçekleştirmektir.

Yul Optimize Edici

Eski bytecode optimizer ile birlikte, *Yul* optimizer artık derleyiciyi `--optimize` ile çağırdığınızda varsayılan olarak etkinleştirilir. Derleyiciyi `--no-optimize-yul` ile çağırarak devre dışı bırakılabilir. Bu çoğunlukla ABI coder v2 kullanan kodları etkiler.

C API Değişiklikleri

`libsolc` C API`sini kullanan istemci kodu artık derleyici tarafından kullanılan belleğin kontrolünü elinde tutmaktadır. Bu değişikliği tutarlı hale getirmek için `solidity_free` fonksiyonu `solidity_reset` olarak yeniden adlandırıldı, `solidity_alloc` ve `solidity_free` fonksiyonları eklendi ve `solidity_compile` artık `solidity_free()` ile açıkça serbest bırakılması gereken bir string döndürüyor.

3.24.6 Kodunuzu nasıl güncelleyebilirsiniz?

Bu bölüm, her işleyişi bozan değişiklik için önceki kodun nasıl güncelleneceğine ilişkin ayrıntılı talimatlar vermektedir.

- `f` external fonksiyon tipinde olduğu için `address(f)` ifadesini `f.address` olarak değiştirin.
- `fonksiyon () external [payable] { ... }` yerine `receive() external payable { ... }`, `fallback() external [payable] { ... }` veya her ikisiyle. Mümkün olduğunda sadece `receive` fonksiyonunu kullanmayı tercih edin.
- `uint length = array.push(value)` ifadesini `array.push(value);` olarak değiştirin. Yeni uzunluğa `array.length` aracılığıyla erişilebilir.
- Bir depolama dizisinin uzunluğunu artırmak için `array.length++` öğesini `array.push()` olarak değiştirin ve azaltmak için `pop()` öğesini kullanın.

- Bir fonksiyonun `@dev` dokümantasyonundaki her adlandırılmış geri dönüş parametresi için, parametrenin adını ilk kelime olarak içeren bir `@return` girişi tanımlayın. Örneğin, `f()` fonksiyonu `function f() public returns (uint value)` şeklinde tanımlanmışsa ve `@dev` şeklinde bir açıklama varsa, geri dönüş parametrelerini aşağıdaki gibi belgeleyin: `@return value Dönüş değeri.`. Bildirimler tuple dönüş türünde göründükleri sırada olduğu sürece, adlandırılmış ve adlandırılmamış dönüş parametreleri belgelerini karıştırabilirsiniz.
- Inline assembly'deki değişken bildirimleri için inline assembly bloğu dışındaki bildirimlerle çakışmayan benzersiz tanımlayıcılar seçin.
- Geçersiz kılmayı düşündüğünüz her arayüz dışı işleve `virtual` ekleyin. Arayüzler dışında uygulaması olmayan tüm fonksiyonlara `virtual` ekleyin. Tekli kalıtım için, her geçersiz kılma fonksiyonuna `override` ekleyin. Çoklu kalıtım için, `override(A, B, ..)` ekleyin, burada parantez içinde geçersiz kılınan fonksiyonu tanımlayan tüm sözleşmeleri listelersiniz. Birden fazla taban aynı fonksiyonu tanımladığında, devralan sözleşme çakışan tüm fonksiyonları geçersiz kılmalıdır.

3.25 Solidity v0.7.0 İşleyişi Bozan Değişiklikler

Bu bölüm, Solidity 0.7.0 sürümünde getirilen ana işleyişi bozan değişiklikleri, değişikliklerin arkasındaki gerekçeleri ve etkilenen kodun nasıl güncelleneceğini vurgular. Tam liste için sürüm değişiklik günlüğü adresini kontrol edin.

3.25.1 Semantiğin Sessiz Değişiklikleri

- Literallerin literal olmayanlarla üslendirilmesi ve kaydırılması (örneğin `1 << x` veya `2 ** x`) işlemi gerçekleştirmek için her zaman `uint256` (negatif olmayan literaller için) veya `int256` (negatif literaller için) türünü kullanacaktır. Önceden, işlem kaydırma miktarı / üstel türde gerçekleştiriliyordu ve bu da yanıltıcı olabiliyordu.

3.25.2 Söz dizimindeki Değişiklikler

- External fonksiyon ve sözleşme oluşturma çağrılarında, Ether ve gas artık yeni bir sözdizimi kullanılarak belirtiliyor: `x.f{gaz: 10000, değer: 2 eter}(arg1, arg2)`. Eski sözdizimi – `x.f.gas(10000).value(2 ether)(arg1, arg2)` – bir hataya neden olacaktır.
- Global değişken `now` kullanımdan kaldırılmıştır, bunun yerine `block.timestamp` kullanılmalıdır. Tek tanımlayıcı `now` global bir değişken için çok geneldir ve işlem sırasında değiştiği izlenimini verebilir, oysa `block.timestamp` sadece bloğun bir özelliği olduğu gerçeğini doğru bir şekilde yansıtır.
- Değişkenler üzerindeki NatSpec yorumlarına yalnızca genel durum değişkenleri için izin verilir, yerel veya dahili değişkenler için izin verilmez.
- `gwei` belirteci artık bir anahtar kelimedir (örneğin `2 gwei` bir sayı olarak belirtmek için kullanılır) ve bir tanımlayıcı olarak kullanılamaz.
- String değişmezleri artık yalnızca yazdırılabilir ASCII karakterleri içerebilir ve bu aynı zamanda heksadesimal (`\xff`) ve unicode escapes (`\u20ac`) gibi çeşitli kaçış dizilerini de içerir.
- Unicode string literals artık geçerli UTF-8 dizilimlerini barındırmak için desteklenmektedir. Bunlar `unicode` öneki ile tanımlanır: `unicode "Hello "`.
- Durum Değiştirilebilirliği: Fonksiyonların durum değiştirilebilirliği artık kalıtım sırasında kısıtlanabilir: Varsayılan durum değiştirilebilirliğine sahip fonksiyonlar `pure` ve `view` fonksiyonları tarafından geçersiz kılınabilirken, `view` fonksiyonları `pure` fonksiyonları tarafından geçersiz kılınabilir. Aynı zamanda, genel durum değişkenleri sabitlerse `view` ve hatta `pure` olarak kabul edilir.

Inline Assembly

- Inline assembly'de kullanıcı tanımlı fonksiyon ve değişken isimlerinde `.` ifadesine izin vermeyin. Solidity'yi Yul-only modunda kullanırsanız bu durum hala geçerlidir.
- `x` depolama işaretçisi değişkeninin yuvasına ve ofsetine `x_slot` ve `x_offset` yerine `x.slot` ve `x.offset` üzerinden erişilir.

3.25.3 Kullanılmayan veya Güvenli Olmayan Özelliklerin Kaldırılması

Depolama dışındaki eşleştirmeler(Mappings outside Storage)

- Bir struct veya dizi bir mapping içeriyorsa, yalnızca depolama alanında kullanılabilir. Önceden, mapping üyeleri bellekte sessizce atlanıyordu, bu da kafa karıştırıcı ve hataya açıktı.
- Depolama alanındaki struct veya dizilere yapılan atamalar, mapping içeriyorsa çalışmaz. Önceden, mappingler kopyalama işlemi sırasında sessizce atlanıyordu, bu da yanıltıcı ve hataya açıktı.

Fonksiyonlar ve Event'ler

- Görünürlük (`public` / `internal`) artık constructor`lar için gerekli değildir: Bir sözleşmenin oluşturulmasını önlemek için, sözleşme `abstract` olarak işaretlenebilir. Bu, constructor'lar için görünürlük kavramını geçersiz kılar.
- Tip Denetleyicisi: Kütüphane fonksiyonları için `virtual` işaretine izin vermeyin: Kütüphanelerden miras alınamayacağı için, kütüphane fonksiyonları sanal olmamalıdır.
- Aynı kalıtım hiyerarşisinde aynı isme ve parametre türlerine sahip birden fazla event'e izin verilmez.
- `using A for B` yalnızca içinde bahsedildiği sözleşmeyi etkiler. Önceden, etki kalıtsaldı. Şimdi, özelliği kullanan tüm türetilmiş sözleşmelerde `using` ifadesini tekrarlamanız gerekir.

İfadeler

- İşaretli türlere göre kaydırmalara izin verilmez. Daha önce, negatif miktarlarla kaydırmalara izin veriliyordu, ancak çalışma zamanında geri döndürülüyordu.
- `finney` ve `szabo` değerleri kaldırılmıştır. Bunlar nadiren kullanılır ve gerçek miktarı kolayca görünür hale getirmez. Bunun yerine, `1e20` veya çok yaygın olan `gwei` gibi açık değerler kullanılabilir.

Bildiriler

- `var` anahtar sözcüğü artık kullanılamıyor. Önceden, bu anahtar sözcük ayrıştırılır ancak bir tür hatasına ve hangi türün kullanılacağına ilişkin bir öneriye neden olurdu. Şimdi, bir ayrıştırıcı hatasıyla sonuçlanıyor.

3.25.4 Arayüz Değişiklikleri

- JSON AST: Hex string değişmezlerini `kind: "hexString"` ile işaretleyin.
- JSON AST: Değeri `null` olan üyeler JSON çıktısından kaldırılır.
- NatSpec: Constructor ve fonksiyonlar tutarlı userdoc çıktısına sahiptir.

3.25.5 Kodunuzu nasıl güncelleyebilirsiniz?

Bu bölümde, her işleyişi bozan değişiklik için önceki kodun nasıl güncelleneceğine ilişkin ayrıntılı talimatlar verilmektedir.

- `x.f.value(...)()` ifadesini `x.f{value: ...}()` olarak değiştirin. Benzer şekilde `(new C).value(...)() new C{value: ...}()` ve `x.f.gas(...).value(...)() x.f{gas: ..., value: ...}()` olarak değiştirin.
- `now` ifadesini `block.timestamp` olarak değiştirin.
- Kaydırma operatörlerindeki sağ operand tiplerini işaretsiz tipler olarak değiştirin. Örneğin `x >> (256 - y)` ifadesini `x >> uint(256 - y)` olarak değiştirin.
- Gerekirse tüm türetilmiş sözleşmelerde `using A for B` ifadelerini tekrarlayın.
- Her constructor`dan `public` anahtar sözcüğünü kaldırın.
- Her constructor`dan `internal` anahtar sözcüğünü kaldırın ve sözleşmeye `abstract` ekleyin (henüz mevcut değilse).
- Inline assembly`deki `_slot` ve `_offset` soneklerini sırasıyla `.slot` ve `.offset` olarak değiştirin.

3.26 Solidity v0.8.0 İşleyişi Bozan Değişiklikler

Bu bölüm, Solidity 0.8.0 sürümünde sunulan ana işleyişi bozan değişiklikleri vurgular. Tam liste için sürüm değişiklik günlüğü adresini kontrol edin.

3.26.1 Semantikte Sessiz Değişiklikler

Bu bölüm, derleyici size bildirmeden mevcut kodun davranışını değiştirdiği değişiklikleri listeler.

- Aritmetik işlemler alttan taşma ve üstten taşma durumunda geri döner. Önceki paketleme davranışını kullanmak için `unchecked { ... }` öğesini kullanabilirsiniz.

  Üstten taşma kontrolleri çok yaygındır, bu nedenle gaz maliyetlerinde hafif bir artışa neden olsa bile kodun okunabilirliğini artırmak için bunları varsayılan hale getirdik.

- ABI coder v2 varsayılan olarak etkinleştirilmiştir.

  Eski davranışı kullanmayı `pragma abicoder v1;` kullanarak seçebilirsiniz. Pragma `pragma experimental ABIEncoderV2;` hala geçerlidir, ancak kullanımdan kaldırılmıştır ve hiçbir etkisi yoktur. Eğer doğrudan belirtmek istiyorsanız, lütfen bunun yerine `pragma abicoder v2;` kullanın.

  ABI coder v2'nin v1'den daha fazla türü desteklediğini ve girdiler üzerinde daha fazla sanity kontrolü gerçekleştirdiğini unutmayın. ABI coder v2 bazı fonksiyon çağrılarını daha pahalı hale getirir ve ayrıca parametre tiplerine uymayan veriler içerdiklerinde ABI coder v1 ile geri dönmeyen sözleşme çağrılarının geri dönmesine neden olabilir.

- Üs alma sağdan ilişkilidir, yani `a**b**c` ifadesi `a**(b**c)` olarak ayrıştırılır. 0.8.0'dan önce `(a**b)**c` olarak ayrıştırılıyordu.

  Bu, üs alma operatörünü ayrıştırmanın yaygın yoludur.

- Başarısız iddialar ve sıfıra bölme veya aritmetik üstten taşma gibi diğer dahili kontroller geçersiz işlem kodunu değil, bunun yerine geri döndürme işlem kodunu kullanır. Daha spesifik olarak, `Panic(uint256)` fonksiyon çağrısına eşit hata verilerini, koşullara özgü bir hata koduyla kullanacaklardır.

  Bu, hatalarda gaz tasarrufu sağlarken, statik analiz araçlarının bu durumları, başarısız bir `require` gibi geçersiz bir girdi üzerindeki bir geri dönüşten ayırt etmesine izin verir.

- Depolama alanında uzunluğu yanlış kodlanmış bir bayt dizisine erişilmesi paniğe neden olunur. Depolama bayt dizilerinin ham gösterimini değiştirmek için inline assembly kullanılmadığı sürece bir sözleşme bu duruma giremez.

- Dizi uzunluğu ifadelerinde sabitler kullanılması halinde Solidity'nin önceki sürümleri, değerlendirme ağacının (evaluation tree) tüm dallarında (branch) rastgele (arbitrary) kesinlik kullanırdı. Artık, sabit değişkenler ara ifadeler olarak kullanıldığında, değerleri, çalışma zamanı ifadelerinde kullanıldıklarında olduğu gibi uygun şekilde yuvarlanacaktır.

- `byte` türü kaldırıldı. Bu `bytes1` türünün bir takma adıydı.

3.26.2 Yeni Kısıtlamalar

Bu bölümde, mevcut sözleşmelerin artık derlenmemesine neden olabilecek değişiklikler listelenmektedir.

- Literallerin açık dönüşümleri ile ilgili yeni kısıtlamalar vardır. Aşağıdaki durumlarda önceki davranış muhtemelen belirsizdi:

  1. Negatif literallerden ve `type(uint160).max` değerinden büyük literallerden `address` değerine açık dönüşümlere izin verilmez.
  2. Literaller ve `T` tamsayı tipi arasındaki açık dönüşümlere yalnızca literal `type(T).min` ve `type(T).max` arasında yer alıyorsa izin verilir. Özellikle, `uint(-1)` kullanımlarını `type(uint).max` ile değiştirin.
  3. Literaller ve enumlar arasındaki açık dönüşümlere yalnızca literal enumdaki bir değeri temsil edebiliyorsa izin verilir.
  4. Literaller ve `address` türü arasındaki açık dönüşümler (örneğin `address(literal)`) `address payable` yerine `address` türüne sahiptir. Açık bir dönüşüm, yani `payable(literal)` kullanılarak payable(ödenebilir) bir adres türü elde edilebilir.

- *Address literals*, `address payable` yerine `address` türüne sahiptir. Açık bir dönüşüm kullanılarak `address payable` türüne dönüştürülebilirler, örneğin `payable(0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF)`.

- Açık tip dönüşümlerinde yeni kısıtlamalar vardır. Dönüşüme yalnızca işaret, genişlik veya tür kategorisinde en fazla bir değişiklik olduğunda izin verilir (`int`, `adres`, `bytesNN`, vb.). Birden fazla değişiklik yapmak için birden fazla dönüşüm kullanın.

  Burada, `T` ve `S` tipleri ve `x` de `S` tipindeki herhangi bir rastgele(arbitrary) değişken olmak üzere, `T(x)` açık dönüşümünü belirtmek için `T(S)` notasyonunu kullanalım. Hem genişliği (8 bitten 16 bite) hem de işareti (işaretli tamsayıdan işaretsiz tamsayıya) değiştirdiği için bu tür bir izin verilmeyen dönüştürmeye örnek olarak `uint16(int8)` verilebilir. Dönüştürmeyi yapmak için bir ara türden geçmek gerekir. Önceki örnekte, bu `uint16(uint8(int8))` veya `uint16(int16(int8))` olacaktır. Dönüştürmenin iki yolunun farklı sonuçlar üreteceğini unutmayın, örneğin `-1` için. Aşağıda, bu kural tarafından izin verilmeyen bazı dönüşüm örnekleri verilmiştir.

- address(uint) ve uint(address): hem tür kategorisini hem de genişliği dönüştürüyor. Bunu sırasıyla address(uint160(uint)) ve uint(uint160(address)) ile değiştirin. - payable(uint160), payable(bytes20) ve payable(integer-literal): hem tür kategorisini hem de durum değiştirilebilirliğini dönüştürüyor. Bunu sırasıyla payable(address(uint160)), payable(address(bytes20)) ve payable(address(integer-literal)) ile değiştirin. payable(0)``ın geçerli olduğunu ve kuralın bir istisnası olduğunu unutmayın. - ``int80(bytes10) ve bytes10(int80): hem tür kategorisini hem de işareti dönüştürüyor. Bunu sırasıyla int80(uint80(bytes10)) ve bytes10(uint80(int80) ile değiştirin. - Contract(uint): hem tür kategorisini hem de genişliği dönüştürüyor. Bunu Contract(address(uint160(uint))) ile değiştirin.

  Belirsizliği önlemek için bu dönüşümlere izin verilmemiştir. Örneğin, uint16 x = uint16(int8(-1)) ifadesinde, x değeri, işaret ve genişlik dönüşümünden hangisinin önce uygulandığına bağlı olacaktır.

- Fonksiyon çağrı seçenekleri sadece bir kez verilebilir, yani c.f{gas: 10000}{value: 1}() geçersizdir ve c.f{gas: 10000, value: 1}() olarak değiştirilmelidir.

- Global fonksiyonlar log0, log1, log2, log3 ve log4 kaldırılmıştır.

  Bunlar büyük ölçüde kullanılmayan düşük seviyeli fonksiyonlardır. Davranışlarına inline assembly'den erişilebilir.

- enum tanımları 256`dan fazla üye içeremez.

  Bu, ABI'deki temel türün her zaman uint8 olduğunu varsaymayı güvenli hale getirecektir.

- Public fonksiyonlar ve event`ler haricinde this, super ve _ isimli açıklamalara izin verilmez. İstisna, bu tür fonksiyon isimlerine izin veren Solidity dışındaki dillerde uygulanan sözleşmelerin arayüzlerini açıklamayı mümkün kılmaktır.

- Koddaki \b, \f ve \v kaçış dizileri için destek kaldırılmıştır. Bunlar, onaltılık kaçış dizileri aracılığıyla eklenmeye devam edebilir; örneğin, sırasıyla \x08, \x0c ve \x0b.

- Global değişkenler tx.origin ve msg.sender, address payable yerine address tipine sahiptir. Bunlar, açık bir dönüşüm kullanılarak address payable türüne, yani payable(tx.origin) veya payable(msg.sender) a dönüştürülebilir.

  Bu değişiklik, derleyicinin bu adreslerin ödenebilir olup olmadığını belirleyememesi nedeniyle yapılmıştır, bu nedenle artık bu gereksinimi görünür kılmak için açık bir dönüşüm gerektirmektedir.

- address türüne açık dönüşüm, her zaman, ödenebilir olmayan bir address türü döndürür. Özellikle, aşağıdaki açık dönüşümler address payable yerine address türüne sahiptir:

  - u``nun ``uint160 türü bir değişken olduğu address(u). u, iki açık dönüşüm kullanılarak payable(address(u)) şeklinde address payable türüne dönüştürebilir.
  - b``nin ``bytes20 türü bir değişken olduğu address(b). b, iki açık dönüşüm kullanılarak payable(address(b)) şeklinde `` address payable`` türüne dönüştürebilir.
  - c``nin bir sözleşme olduğu ``address(c). Önceden, bu dönüşümün dönüş türü, sözleşmenin Ether alıp alamayacağına (bir receive fonksiyonuna veya bir payable fallback fonksiyonuna sahip olarak) bağlıydı. payable(c) dönüşümü address payable türüne sahiptir ve yalnızca c sözleşmesi Ether alabildiğinde bu dönüşüme izin verilir. Genel olarak c, şu açık dönüşüm kullanılarak her zaman address payable türüne dönüştürülebilir: payable(address(c)). address(this) türünün address(c) ile aynı kategoriye girdiğini ve aynı kuralların onun için de geçerli olduğunu unutmayın.

- Inline assembly`de yerleşik chainid artık pure yerine view olarak kabul edilmektedir.

- Tekli negasyon artık işaretsiz tamsayılar üzerinde kullanılamaz, sadece işaretli tamsayılar üzerinde kullanılabilir.

3.26.3 Arayüz Değişiklikleri

- `--combined-json` çıktısı değişti: JSON alanları `abi`, `devdoc`, `userdoc` ve `storage-layout` artık alt nesnelerdir. 0.8.0'dan önce string olarak serileştiriliyorlardı.
- "Eski AST" kaldırıldı (komut satırı arayüzünde `--ast-json` ve standart JSON için `legacyAST`). Yerine "kompakt AST" (`--ast-compact--json` sırasıyla `AST`) kullanın.
- Eski hata raporlayıcı (`--old-reporter`) kaldırıldı.

3.26.4 Kodunuzu nasıl güncelleyebilirsiniz?

- Aritmetik paketlemeye güveniyorsanız, her işlemi `unchecked { ... }`.
- İsteğe bağlı: SafeMath veya benzer bir kütüphane kullanıyorsanız, `x.add(y)` ifadesini `x + y`, `x.mul(y)` ifadesini `x * y` vb. olarak değiştirin.
- Eski ABI kodlayıcı ile devam etmek istiyorsanız `pragma abicoder v1;` ekleyin.
- İsteğe bağlı olarak `pragma experimental ABIEncoderV2` veya `pragma abicoder v2` gereksiz olduğu için kaldırın.
- `byte` ifadesini `bytes1` olarak değiştirin.
- Gerekirse ara açık tip dönüşümler ekleyin.
- `c.f{gas: 10000}{value: 1}()` ifadesini `c.f{gas: 10000, value: 1}()` olarak birleştirin.
- `msg.sender.transfer(x)` öğesini `payable(msg.sender).transfer(x)` olarak değiştirin veya `address payable` türü saklanmış bir değişken kullanın.
- `x**y**z` ifadesini `(x**y)**z` olarak değiştirin.
- `log0`, ..., `log4` yerine inline assembly kullanın.
- İşaretsiz tamsayıları, türün maksimum değerinden çıkarıp 1 ekleyerek negatifleştirin (örneğin `type(uint256).max - x + 1`, `x` in sıfır olmadığından emin olarak)

3.27 NatSpec Formatı

Solidity sözleşmeleri, fonksiyonlar, dönüş değişkenleri ve daha fazlası için zengin dokümantasyon sağlamak üzere özel bir yorum biçimi kullanabilir. Bu özel form Ethereum Doğal Dil Belirtim Formatı(Ethereum Natural Language Specification Format) (NatSpec) olarak adlandırılır.

Not: NatSpec, Doxygen'den esinlenmiştir. Doxygen tarzı yorumlar ve etiketler kullansa da, Doxygen ile olan sıkı uyumluluğunu sürdürme niyeti yoktur. Lütfen aşağıda listelenen desteklenmiş etiketleri dikkatlice inceleyin.

Bu dokümantasyon, geliştirici odaklı mesajlar ve son kullanıcıya yönelik mesajlar olarak bölümlere ayrılmıştır. Bu mesajlar son kullanıcıya (insan) sözleşme ile etkileşime gireceği (örneğin bir işlem imzalayacağı) zaman gösterilebilir.

Solidity sözleşmelerinin tüm genel arayüzler (ABI'deki her şey) için NatSpec kullanılarak tamamen açıklanması önerilir.

NatSpec, akıllı sözleşme yazarının kullanacağı ve Solidity derleyicisi tarafından anlaşılan yorumlar için biçimlendirme içerir. Ayrıca bu yorumları makine tarafından okunabilir bir biçime dönüştüren Solidity derleyicisinin çıktısı da aşağıda detaylı olarak açıklanmıştır.

NatSpec, üçüncü taraf araçlar tarafından kullanılan ek açıklamaları da içerebilir. Bunlar büyük olasılıkla `@custom:<name>` etiketi aracılığıyla gerçekleştirilir ve iyi bir kullanım örneği analiz ve doğrulama araçlarıdır.

3.27.1 Dokümantasyon Örneği

Dokümantasyon, Doxygen notasyon formatı kullanılarak her `contract` , `interface` , `library` , `function` ve `event` üzerine eklenir. Bir `public` durum değişkeni, NatSpec'in kullanım amaçları doğrultusunda bir `fonksiyon` a eşdeğerdir.

- Solidity için tek veya çok satırlı yorumlar için `//` veya `/**` ve `*/` ile sonlandırmayı tercih edebilirsiniz.
- Vyper için, iç içeriğe yalın yorumlarla girintili `"""` kullanın. Vyper belgelerine <https://vyper.readthedocs.io/en/latest/natspec.html>`__ bakınız.

Aşağıdaki örnekte, mevcut tüm etiketler kullanılarak bir sözleşme ve bir fonksiyon gösterilmektedir.

Not: Solidity derleyicisi, etiketleri yalnızca external veya public olmaları durumunda yorumlamaktadır. Internal ve private fonksiyonlarınız için benzer yorumlar kullanabilirsiniz, ancak bunlar çözümlenmeyecektir.

Bu özellik belki gelecekte değişebilir.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title Ağaçlar için bir simülatör
/// @author Larry A. Gardner
/// @notice Bu sözleşmeyi yalnızca en sade simulasyonlar için kullanabilirsiniz
/// @dev Tüm fonksiyon çağrıları şu anda yan etkiler olmadan uygulanmaktadır
/// @custom:experimental Bu deneysel bir sözleşmedir.
contract Tree {
    /// @notice Canlı ağaçlar için ağaç yaşını yıl olarak hesaplayın, üst sayıya␣
↪yuvarlayın
    /// @dev Alexandr N. Tetearing algoritması doğruluğu artırabilir
    /// @param rings Dendrokronolojik örnekten elde edilen halka sayısı
    /// @return Yıl cinsinden yaş, kısmi yıllar için yuvarlanır
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice Ağacın sahip olduğu yaprak miktarını döndürür.
    /// @dev Yalnızca sabit bir sayı döndürür.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

contract Plant {
    function leaves() external virtual pure returns(uint256) {
        return 3;
    }
}
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
contract KumquatTree is Tree, Plant {
    function age(uint256 rings) external override pure returns (uint256) {
        return rings + 2;
    }

    /// Bu spesifik ağaç türünün sahip olduğu yaprak miktarını döndürür
    /// @inheritdoc Tree
    function leaves() external override(Tree, Plant) pure returns(uint256) {
        return 3;
    }
}
```

3.27.2 Tags

Tüm etiketler opsiyoneldir. Aşağıdaki tabloda her bir NatSpec etiketinin amacı ve nerede kullanılabileceği açıklanmaktadır. Özel bir durum olarak, hiçbir etiket kullanılmazsa Solidity derleyicisi bir `///` veya `/**` yorumunu `@notice` ile etiketlenmiş gibi yorumlayacaktır.

Etiket		Bağlam
`@title`	Sözleşmeyi/arayüzü tanımlaması gereken bir başlık	contract, library, interface
`@author`	Yazarın adı	contract, library, interface
`@notice`	Son kullanıcıya bunun ne işe yaradığını açıklayın	contract, library, interface, function, public state variable, event
`@dev`	Bir geliştiriciye ekstra ayrıntıları açıklayın	contract, library, interface, function, state variable, event
`@param`	Tıpkı Doxygen'de olduğu gibi bir parametreyi belgeler (parametre adının ardından gelmelidir)	function, event
`@return`	Bir sözleşmenin fonksiyonunun dönüş değişkenlerini belgeler	function, public state variable
`@inheritdoc`	Temel fonksiyondaki tüm eksik etiketleri kopyalar (ardından sözleşme adı gelmelidir)	function, public state variable
`@custom:...`	Özel etiket, semantiği uygulama tanımlıdır	everywhere

Fonksiyonunuz `(int quotient, int remainder)` gibi birden fazla değer döndürüyorsa, `@param` ifadeleriyle aynı formatta birden fazla `@return` ifadesi kullanın.

Özel etiketler `@custom:` ile başlar ve ardından bir veya daha fazla küçük harf veya kısa çizgi gelmelidir. Ancak kısa çizgi ile başlayamaz. Her yerde kullanılabilirler ve geliştirici belgelerinin bir parçasıdırlar.

Dinamik İfade Biçimleri

Solidity derleyicisi, NatSpec belgelerini Solidity kaynak kodunuzdan bu kılavuzda açıklandığı gibi JSON çıktısına aktaracaktır. Bu JSON çıktısının kullanıcısı, örneğin son kullanıcı istemci yazılımı, bunu son kullanıcıya doğrudan sunabilir veya bazı ön işlemler uygulayabilir.

Örneğin, bazı istemci yazılımları render edecektir:

```
/// @notice This function will multiply `a` by 7
```

son kullanıcıya:

```
This function will multiply 10 by 7
```

eğer bir fonksiyon çağrılıyorsa ve `a` girdisine 10 değeri atanmışsa.

Bu dinamik ifadelerin belirtilmesi Solidity dokümantasyonunun kapsamı dışındadır ve bu konuda daha fazla bilgiyi the radspec project adresinden edinebilirsiniz.

Kalıtım Notları

NatSpec içermeyen fonksiyonlar otomatik olarak temel fonksiyonlarının dokümantasyonunu devralacaktır. Bununla ilgili istisnalar şunlardır:

- Parametre adları farklı olduğunda.
- Birden fazla temel fonksiyon olduğunda.
- Kalıtım için hangi sözleşmenin kullanılması gerektiğini belirten açık bir `@inheritdoc` etiketi olduğunda.

3.27.3 Dokümantasyon Çıktısı

Derleyici tarafından çözümlendiğinde, yukarıdaki örnekteki gibi belgeler iki farklı JSON dosyası üretecektir. Biri son kullanıcı tarafından bir fonksiyon çalıştırıldığında bildirim olarak tüketilmek üzere, diğeri ise geliştirici tarafından kullanılmak üzere tasarlanmıştır.

Yukarıdaki sözleşme `ex1.sol` olarak kaydedilirse, belgeleri kullanarak oluşturabilirsiniz:

```
solc --userdoc --devdoc ex1.sol
```

Çıktı aşağıda verilmiştir.

Not: Solidity 0.6.11 sürümünden itibaren NatSpec çıktısı ayrıca bir `version` ve bir `kind` alanı içerir. Şu anda `version` 1 olarak ayarlanmıştır ve `kind` `user` veya `dev` alanlarından biri olmalıdır. Gelecekte, eski sürümleri kullanımdan kaldırarak yeni sürümlerin tanıtılması mümkündür.

Kullanıcı Dokümantasyonu

Yukarıdaki dokümantasyon çıktı olarak aşağıdaki kullanıcı dokümantasyonu JSON dosyasını üretecektir:

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculate tree age in years, rounded up, for live trees"
    }
  },
  "notice" : "You can use this contract for only the most basic simulation"
}
```

Metodları bulmak için anahtarın sadece fonksiyonun adı değil, *Contract ABI* da tanımlandığı gibi fonksiyonun kanonik imzası olduğunu unutmayın.

Geliştirici Dokümantasyonu

Kullanıcı dokümantasyon dosyasının yanı sıra, bir geliştirici dokümantasyon JSON dosyası da üretilmeli ve aşağıdaki gibi görünmelidir:

```
{
  "version" : 1,
  "kind" : "dev",
  "author" : "Larry A. Gardner",
  "details" : "All function calls are currently implemented without side effects",
  "custom:experimental" : "This is an experimental contract.",
  "methods" :
  {
    "age(uint256)" :
    {
      "details" : "The Alexandr N. Tetearing algorithm could increase precision",
      "params" :
      {
        "rings" : "The number of rings from dendrochronological sample"
      },
      "return" : "age in years, rounded up for partial years"
    }
  },
  "title" : "A simulator for trees"
}
```

3.28 Güvenlikle İlgili Değerlendirmeler

Genellikle öngörüldüğü gibi çalışan bir yazılım oluşturmak oldukça kolay olsa da, kimsenin bu yazılımı **öngörülmeyen** bir şekilde kullanamayacağını kontrol etmek oldukça zordur.

Solidity'de durum daha da önemlidir çünkü akıllı sözleşmeleri tokenları ya da muhtemelen daha değerli şeyleri yönetmek için kullanabilirsiniz. Dahası, bir akıllı sözleşme her yürütüldüğünde herkese görünür bir şekilde gerçekleşir ve buna ek olarak kaynak kodu da genellikle erişilebilirdir.

Elbette her zaman ne kadar tehlikede olduğunu göz önünde bulundurmanız gerekir: Bir akıllı sözleşmeyi halka (ve dolayısıyla kötü niyetli kişilere) açık ve hatta belki de açık kaynaklı bir web hizmeti ile karşılaştırabilirsiniz. Bu web hizmetinde yalnızca alışveriş listenizi saklıyorsanız, çok fazla dikkat etmeniz gerekmeyebilir, ancak banka hesabınızı bu web hizmetini kullanarak yönetiyorsanız, daha dikkatli olmalısınız.

Bu bölüm bazı tuzakları ve genel güvenlik önerilerini listeleyecektir, ancak elbette asla eksiksiz olamaz. Ayrıca, akıllı sözleşme kodunuz hatasız olsa bile derleyicide ya da platformun kendisinde bir hata bulunabileceğini unutmayın. Derleyicinin herkes tarafından bilinen güvenlikle ilgili bazı hatalarının bir listesi, makine tarafından da okunabilen :ref: *bilinen hataların listesi<known_bugs>* bölümünde bulunabilir. Solidity derleyicisinin kod oluşturucusunu kapsayan bir hata ödül programı olduğunu unutmayın.

Her zaman olduğu gibi, açık kaynak belgelerinde, lütfen bu bölümü genişletmemize yardımcı olun (özellikle, bazı örneklerin hiç kimseye zararı dokunmaz)!

NOT: Aşağıdaki listeye ek olarak, Guy Lando'nun bilgi listesinde ve Consensys GitHub reposunda daha fazla güvenlik önerisi ve en iyi uygulamaları bulabilirsiniz.

3.28.1 Tuzaklar

Özel(Private) Bilgiler ve Rastgelelik

Bir akıllı sözleşmede kullandığınız her şey, yerel değişkenler ve `private` olarak işaretlenmiş durum değişkenleri de dahil olmak üzere herkes tarafından görülebilir.

Madencilerin hile yapabilmesini istemiyorsanız, akıllı sözleşmelerde rastgele sayılar kullanmak oldukça zordur.

Yeniden Giriş (Re-Entrancy)

Bir sözleşmeden (A) başka bir sözleşmeye (B) herhangi bir etkileşim ve herhangi bir Ether transferi, kontrolü o sözleşmeye (B) devreder. Bu, B'nin bu etkileşim tamamlanmadan önce A'yı geri çağırmasını mümkün kılar. Bir örnek vermek gerekirse, aşağıdaki kod bir hata içermektedir (bu sadece bir kod parçacığıdır ve tam bir sözleşme değildir):

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

// BU SÖZLEŞME BUG İÇERİR: KULLANMAYIN
contract Fund {
    /// @dev Sözleşmenin ether paylarının eşleştirilmesi.
    mapping(address => uint) shares;
    /// Payınızı geri çekin.
    function withdraw() public {
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

Burada sorun, `send``in bir parçası olarak sınırlı gas miktarı nedeniyle çok ciddi değildir, ancak yine de bir zafiyet ortaya çıkarmaktadır: Ether transferi her zaman kod yürütmeyi içerebilir, bu nedenle alıcı ``withdraw``a geri çağıran bir sözleşme olabilir. Bu, birden fazla geri ödeme almasına ve temelde sözleşmedeki tüm Ether'i geri almasına izin verecektir. Özellikle, aşağıdaki sözleşme, varsayılan olarak kalan tüm gazı ileten ``call` kullandığı için bir saldırganın birden fazla kez geri ödeme yapmasına izin verecektir:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

// BU SÖZLEŞME BUG İÇERİR: KULLANMAYIN
contract Fund {
    /// @dev Sözleşmenin ether paylarının eşleştirilmesi.
    mapping(address => uint) shares;
    /// Payınızı geri çekin.
    function withdraw() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

Yeniden Giriş'den(Re-entrancy) kaçınmak için, aşağıda daha ayrıntılı olarak açıklandığı gibi Checks-Effects-Interactions kalıbını kullanabilirsiniz:

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

contract Fund {
    /// @dev Sözleşmenin ether paylarının eşleştirilmesi.
    mapping(address => uint) shares;
    /// Payınızı geri çekin.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        payable(msg.sender).transfer(share);
    }
}
```

Yeniden girişin yalnızca Ether aktarımının değil, başka bir sözleşmedeki herhangi bir fonksiyon çağrısının da bir etkisi olduğunu unutmayın. Ayrıca, çoklu sözleşme içeren durumları da hesaba katmanız gerekmektedir. Çağrılan bir sözleşme, bağımlı olduğunuz başka bir sözleşmenin yapısını değiştirebilir.

Gas Limiti ve Döngüler

Sabit sayıda iterasyona sahip olmayan döngüler, örneğin depolama değerine bağlı döngüler, dikkatli bir şekilde kullanılmalıdır: Blok gas limiti nedeniyle, işlemler yalnızca belirli bir miktarda gas tüketebilir. Ya açıkça ya da sadece normal çalışma nedeniyle, bir döngüdeki yineleme sayısı blok gas limitinin ötesine geçebilir ve bu da tüm sözleşmenin belirli bir noktada durmasına neden olabilir. Bu durum, yalnızca blok zincirinden veri okumak için çalıştırılan `view` fonksiyonları için geçerli olmayabilir. Yine de, bu tür fonksiyonlar zincir üzerindeki işlemlerin bir parçası olarak diğer sözleşmeler tarafından çağrılabilir ve bunları durdurabilir. Lütfen sözleşmelerinizin dokümantasyonunda bu tür durumlar hakkında açıkça bilgi verin.

Ether Gönderme ve Alma

- Ne sözleşmeler ne de "harici hesaplar" şu anda birinin onlara Ether göndermesini engelleyememektedir. Sözleşmeler normal bir transfere yanıt verebilir ve reddedebilir, ancak bir mesaj çağrısı oluşturmadan Ether'i taşımanın yolları vardır. Bir yol basitçe sözleşme adresine "mine to" yapmak, ikinci yol ise `selfdestruct(x)` kullanmaktır.
- Bir sözleşme Ether alırsa (bir fonksiyon çağrılmadan), ya *receive Ether* ya da *fallback* fonksiyonu çalıştırılır. Eğer bir receive ya da fallback fonksiyonu yoksa, Ether reddedilir (bir istisna gönderilerek). Bu fonksiyonlardan birinin yürütülmesi sırasında, sözleşme yalnızca o anda kendisine aktarılan "gas stipend "in (2300 gas) kullanılabilir olmasına güvenebilir. Ancak bu miktarı depolamayı değiştirmek için yeterli değildir (bunu kesin olarak kabul etmeyin, gelecekteki hard fork'larla miktar değişebilir). Sözleşmenizin bu şekilde Ether alabileceğinden emin olmak için, receive ve fallback fonksiyonlarının gas gereksinimlerini kontrol etmeyi unutmayın (örneğin Remix'teki "ayrıntılar" bölümünde).
- Daha fazla gas'ı `addr.call{value: x}("")` kullanarak alıcı sözleşmeye iletmenin bir yolu vardır. Bu aslında `addr.transfer(x)` ile aynıdır, sadece kalan tüm gas miktarını iletir ve alıcının daha pahalı eylemler gerçekleştirmesine olanak sağlar (ve hatayı otomatik olarak iletmek yerine bir hata kodu döndürür). Bu, gönderici sözleşmeyi geri çağırmayı veya aklınıza gelmemiş olabilecek diğer durum değişikliklerini içerebilir. Dolayısıyla güvenilir kullanıcılar için olduğu kadar kötü niyetli kullanıcılar için de büyük esneklik sağlar.
- Wei miktarını temsil etmek için mümkün olan en kesin birimleri kullanın, çünkü kesinlik eksikliği nedeniyle yuvarlanan her şeyi kaybedersiniz.
- Eğer `address.transfer` kullanarak Ether göndermek istiyorsanız, dikkat etmeniz gereken bazı detaylar var:

1. Alıcı bir sözleşme ise, alıcı veya fallback fonksiyonunun yürütülmesine neden olur ve bu da gönderen sözleşmeyi geri çağırabilir.
2. Ether gönderimi, çağrı derinliğinin 1024'ün üzerine çıkması nedeniyle başarısız olabilir. Çağrı derinliği tamamen çağıranın kontrolünde olduğundan, aktarımı başarısız olmaya zorlayabilirler; bu olasılığı göz önünde bulundurun veya `send` kullanın ve dönüş değerini her zaman kontrol ettiğinizden emin olun. Daha da iyisi, sözleşmenizi alıcının Ether çekebileceği bir model kullanarak yazın.
3. Ether göndermek, alıcı sözleşmenin yürütülmesi için tahsis edilen gas miktarından daha fazlası gerektiği için de başarısız olabilir (açıkça *require*, *assert*, *revert* kullanarak veya işlem çok pahalı olduğu için) - "gas biter" (OOG). Dönüş değeri kontrolü ile `transfer` veya `send` kullanırsanız, bu, alıcının gönderim sözleşmesindeki ilerlemeyi engellemesi için bir yöntem sağlayabilir. Burada da en iyi uygulama "send" pattern yerine bir *"withdraw" pattern* kullanmaktır.

Çağrı Yığını Derinliği

External fonksiyon çağrıları, 1024 olan maksimum çağrı yığını boyutu sınırını aştıkları için her an başarısız olabilirler. Bu gibi durumlarda Solidity bir istisna gönderir. Kötü niyetli kişiler, sözleşmenizle etkileşime girmeden önce çağrı yığınını yüksek bir değere zorlayabilir. Tangerine Whistle <https://eips.ethereum.org/EIPS/eip-608>`_ hardfork olduğundan, 63/64 kuralı çağrı yığını derinliği saldırısını kullanışsız hale getirir. Ayrıca, her ikisinin de 1024 yığın yuvası boyut sınırına sahip olmasına rağmen, çağrı yığını ve ifade yığınının birbiriyle alakasız olduğunu unutmayın.

Eğer çağrı yığını tükenirse `.send()` fonksiyonunun **bir istisna göndermediğini**, bu durumda `false` döndürdüğünü unutmayın. Düşük seviyeli fonksiyonlar `.call()`, `.delegatecall()` ve `.staticcall()` da aynı şekilde davranırlar.

Yetkilendirilmiş Proxyler (Authorized Proxies)

Sözleşmeniz bir proxy olarak hareket edebiliyorsa, yani kullanıcı tarafından sağlanan verilerle rastgele sözleşmeleri çağırabiliyorsa, kullanıcı esasen proxy sözleşmesinin kimliğini üstlenebilir. Başka koruyucu önlemleriniz olsa bile, sözleşme sisteminizi proxy'nin herhangi bir izne sahip olmayacağı şekilde (kendisi için bile) oluşturmak en iyisidir. Gerekirse bunu ikinci bir proxy kullanarak gerçekleştirebilirsiniz:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ProxyWithMoreFunctionality {
    PermissionlessProxy proxy;

    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return proxy.callOther(addr, payload);
    }
    // Diğer fonksiyonlar ve diğer fonksiyonellikler
}

// Bu tam sözleşmedir, başka hiçbir fonksiyonu yoktur ve çalışması
// için hiçbir ayrıcalık gerektirmez.
contract PermissionlessProxy {
    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return addr.call(payload);
    }
}
```

tx.origin

Doğrulama için asla tx.origin kullanmayın. Diyelim ki şöyle bir cüzdan sözleşmeniz var:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// BU SÖZLEŞME BUG İÇERİR : KULLANMAYIN
contract TxUserWallet {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint amount) public {
        // BUG burada, tx.origin yerine msg.sender kullanın
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Şimdi birisi sizi bu saldırı cüzdanının adresine Ether göndermeniz için kandırıyor:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

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

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Cüzdanınız doğrulama için `msg.sender` adresini kontrol etseydi, sahibinin adresi yerine saldırı cüzdanının adresini alırdı. Ancak `tx.origin` adresini kontrol ederek, işlemi başlatan orijinal adresi, yani hala sahibinin adresini alır. Saldırgan cüzdan anında tüm paranızı çeker.

Two's Complement / Underflows / Overflows

Birçok programlama dilinde olduğu gibi, Solidity'nin integer türleri aslında tam sayı değildir. Değerler küçük olduğunda tamsayılara benzerler, ancak keyfi olarak büyük sayıları temsil edemezler.

Aşağıdaki kod bir taşmaya neden olur çünkü toplama işleminin sonucu `uint8` tipinde saklanamayacak kadar büyüktür:

```
uint8 x = 255;
uint8 y = 1;
return x + y;
```

Solidity'nin bu taşmaları ele aldığı iki modu bulunmaktadır: Kontrollü ve Kontrolsüz veya "wrapping" modu.

Varsayılan kontrollü mod, taşmaları tespit eder ve başarısız bir doğrulamaya neden olur. Bu kontrolü `unchecked { ... }` kullanarak bu kontrolü devre dışı bırakabilir ve taşmanın sessizce göz ardı edilmesine neden olabilirsiniz. Yukarıdaki kod `unchecked { ... }` içine sarılmış olsaydı `0` döndürürdü. .

Kontrollü modda bile, taşma hatalarından korunduğunuzu sanmayın. Bu modda, taşmalar her zaman geri döndürülecektir. Eğer taşmadan kaçınmak mümkün değilse, bu durum akıllı sözleşmenin belirli bir durumda takılı kalmasına neden olabilir.

Genel olarak, işaretli sayılar için bazı daha özel uç durumlara sahip olan ikiye tamamlayan sayı gösteriminin sınırları hakkında bilgi edinmelisiniz.

Girdilerin boyutunu makul bir aralıkla sınırlamak için `require` kullanmayı deneyin ve olası taşmaları bulmak için *SMT checker* kullanın.

Mappingleri Temizleme

Yalnızca depolama amaçlı bir anahtar-değer veri yapısı olan Solidity tipi `mapping` (bkz. *Eşleme Türleri*), sıfır olmayan bir değer atanmış anahtarların kaydını tutmaz. Bu nedenle, yazılan anahtarlar hakkında ekstra bilgi olmadan bir mapping'i temizlemek mümkün değildir. Bir dinamik depolama dizisinin temel türü olarak bir `mapping` kullanılıyorsa, dizinin silinmesi veya boşaltılmasının `mapping` elemanları üzerinde hiçbir etkisi olmayacaktır. Aynı durum, örneğin, bir dinamik depolama dizisinin temel türü olan bir `struct``ın eleman türünün bir ``mapping` olması durumunda da geçerlidir. Bir `mapping` içeren struct veya dizilerin atamalarında da `mapping` göz ardı edilir.

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

contract Map {
    mapping (uint => uint)[] array;

    function allocate(uint newMaps) public {
        for (uint i = 0; i < newMaps; i++)
            array.push();
    }

    function writeMap(uint map, uint key, uint value) public {
        array[map][key] = value;
    }

    function readMap(uint map, uint key) public view returns (uint) {
        return array[map][key];
    }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    function eraseMaps() public {
        delete array;
    }
}
```

Yukarıdaki örneği ve aşağıdaki çağrı dizisini göz önünde bulundurun: `allocate(10)`, `writeMap(4, 128, 256)`. Bu noktada, `readMap(4, 128)` çağrısı 256 değerini döndürür. Eğer `eraseMaps` çağrısı yaparsak, `array` durum değişkeninin uzunluğu sıfırlanır, ancak `mapping` elemanları sıfırlanamadığından, bilgileri sözleşmenin deposunda canlı kalır. Diziyi sildikten sonra, `allocate(5)` çağrısı `array[4]` öğesine tekrar erişmemizi sağlar ve `readMap(4, 128)` çağrısı, başka bir `writeMap` çağrısı olmadan bile 256 döndürür.

Eğer `mapping` bilgilerinizin silinmesi gerekiyorsa, `iterable mapping <https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol>`_ benzeri bir kütüphane kullanmayı düşünün, bu sayede anahtarlar arasında gezinebilir ve uygun ``mapping` içindeki değerleri silebilirsiniz.

Küçük Detaylar

- Tam 32 baytı kaplamayan türler "kirli yüksek dereceli bitler" içerebilir. Bu durum özellikle `msg.data` türüne eriştiğinizde önemlidir - bu bir değiştirilebilirlik riski oluşturur: Bir `f(uint8 x)` fonksiyonunu `0xff000001` ve `0x00000001` ham bayt argümanı ile çağıran işlemler oluşturabilirsiniz. Her ikisi de sözleşmeye gönderilir ve `x` söz konusu olduğunda her ikisi de 1 sayısı gibi görünecektir, ancak `msg.data` farklı olacaktır, bu nedenle herhangi bir şey için `keccak256(msg.data)` kullanırsanız, farklı sonuçlar elde edersiniz.

3.28.2 Öneriler

Uyarıları Ciddiye Alın

Derleyici sizi bir konuda uyarıyorsa, bunu değiştirmelisiniz. Bu uyarının güvenlikle ilgili olduğunu düşünmeseniz bile, altında başka bir sorun yatıyor olabilir. Verdiğimiz herhangi bir derleyici uyarısı, kodda yapılacak küçük değişikliklerle giderilebilir.

Yeni eklenen tüm uyarılardan haberdar olmak için her zaman derleyicinin en son sürümünü kullanın.

Derleyici tarafından verilen `info` türündeki mesajlar tehlikeli değildir ve sadece derleyicinin kullanıcı için yararlı olabileceğini düşündüğü ekstra önerileri ve isteğe bağlı bilgileri temsil eder.

Ether Miktarını Kısıtlayın

Akıllı bir sözleşmede saklanabilecek Ether (veya diğer tokenler) miktarını kısıtlayın. Kaynak kodunuzda, derleyicide veya platformda bir hata varsa, bu fonlar kaybolabilir. Kaybınızı sınırlamak istiyorsanız, Ether miktarını sınırlayın.

Küçük ve Modüler Tutun

Sözleşmelerinizi küçük ve kolayca anlaşılabilir tutun. Diğer sözleşmelerdeki veya kütüphanelerdeki ilgisiz fonksiyonları ayırın. Kaynak kod kalitesiyle ilgili genel tavsiyeler elbette geçerlidir: Yerel değişkenlerin miktarını, fonksiyonların uzunluğunu ve benzerlerini sınırlayın. Başkalarının niyetinizin ne olduğunu ve kodun yapıldığından farklı olup olmadığını görebilmesi için fonksiyonlarınızı belgeleyin.

Kontroller-Etkiler-Etkileşimler Modelini Kullanın

Çoğu fonksiyon önce bazı kontroller yapacaktır (fonksiyonu kim çağırdı, argümanlar aralıkta mı, yeterince Ether gönderdiler mi, kişinin tokenleri var mı, vb.) Bu kontroller önce yapılmalıdır.

İkinci adım olarak, tüm kontroller geçerse, mevcut sözleşmenin durum değişkenlerine etkiler yapılmalıdır. Diğer sözleşmelerle etkileşim herhangi bir fonksiyonda en son adım olmalıdır.

İlk sözleşmeler bazı etkileri geciktirir ve harici fonksiyon çağrılarının hatasız bir durumda dönmesini beklerdi. Bu, yukarıda açıklanan yeniden giriş sorunu nedeniyle genellikle ciddi bir hatadır.

Ayrıca, bilinen sözleşmelere yapılan çağrıların da bilinmeyen sözleşmelere çağrı yapılmasına neden olabileceğini unutmayın, bu nedenle bu kalıbı her zaman uygulamak her zaman daha iyidir.

Arızaya Karşı Güvenli Mod Ekleyin

Sisteminizi tamamen merkeziyetsiz hale getirmek herhangi bir aracıyı ortadan kaldıracak olsa da, özellikle yeni kodlar için bir tür arıza güvenliği mekanizması eklemek iyi bir fikir olabilir:

Akıllı sözleşmenize "Herhangi bir Ether sızdı mı?", "Tokenların toplamı sözleşmenin bakiyesine eşit mi?" gibi kendi kendine kontroller gerçekleştiren bir fonksiyon ekleyebilirsiniz. Bunun için çok fazla gaz kullanamayacağınızı unutmayın, bu nedenle zincir dışı hesaplamalar yoluyla yardım gerekebilir.

Kendi kendine kontrol başarısız olursa, sözleşme otomatik olarak bir tür "arıza emniyetli" moda geçer; örneğin, özelliklerin çoğunu devre dışı bırakır, kontrolü sabit ve güvenilir bir üçüncü tarafa devreder veya sözleşmeyi basit bir "paramı geri ver" sözleşmesine dönüştürür.

Peer İncelemesi İsteyin

Bir kod parçası ne kadar çok kişi tarafından incelenirse, o kadar çok sorun bulunur. İnsanlardan kodunuzu incelemelerini istemek, kodunuzun kolay anlaşılır olup olmadığını anlamak için bir çapraz kontrol olarak da yardımcı olur - iyi akıllı sözleşmeler için çok önemli bir kriterdir.

3.29 SMTChecker and Formal Verification

Using formal verification it is possible to perform an automated mathematical proof that your source code fulfills a certain formal specification. The specification is still formal (just as the source code), but usually much simpler.

Note that formal verification itself can only help you understand the difference between what you did (the specification) and how you did it (the actual implementation). You still need to check whether the specification is what you wanted and that you did not miss any unintended effects of it.

Solidity implements a formal verification approach based on SMT (Satisfiability Modulo Theories) and Horn solving. The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements. That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true. If an assertion failure is found, a counterexample may be given to the user showing

how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe.

The other verification targets that the SMTChecker checks at compile time are:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.
- Out of bounds index access.
- Insufficient funds for a transfer.

All the targets above are automatically checked by default if all engines are enabled, except underflow and overflow for Solidity >=0.8.7.

The potential warnings that the SMTChecker reports are:

- `<failing property> happens here.`. This means that the SMTChecker proved that a certain property fails. A counterexample may be given, however in complex situations it may also not show a counterexample. This result may also be a false positive in certain cases, when the SMT encoding adds abstractions for Solidity code that is either hard or impossible to express.
- `<failing property> might happen here`. This means that the solver could not prove either case within the given timeout. Since the result is unknown, the SMTChecker reports the potential failure for soundness. This may be solved by increasing the query timeout, but the problem might also simply be too hard for the engine to solve.

To enable the SMTChecker, you must select *which engine should run*, where the default is no engine. Selecting the engine enables the SMTChecker on all files.

Not: Prior to Solidity 0.8.4, the default way to enable the SMTChecker was via `pragma experimental SMTChecker;` and only the contracts containing the pragma would be analyzed. That pragma has been deprecated, and although it still enables the SMTChecker for backwards compatibility, it will be removed in Solidity 0.9.0. Note also that now using the pragma even in a single file enables the SMTChecker for all files.

Not: The lack of warnings for a verification target represents an undisputed mathematical proof of correctness, assuming no bugs in the SMTChecker and the underlying solver. Keep in mind that these problems are *very hard* and sometimes *impossible* to solve automatically in the general case. Therefore, several properties might not be solved or might lead to false positives for large contracts. Every proven property should be seen as an important achievement. For advanced users, see *SMTChecker Tuning* to learn a few options that might help proving more complex properties.

3.29.1 Tutorial

Overflow

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

contract Overflow {
    uint immutable x;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        return add(x, y);
    }
}
```

The contract above shows an overflow check example. The SMTChecker does not check underflow and overflow by default for Solidity >=0.8.7, so we need to use the command line option `--model-checker-targets "underflow, overflow"` or the JSON option `settings.modelChecker.targets = ["underflow", "overflow"]`. See *this section for targets configuration*. Here, it reports the following:

```
Warning: CHC: Overflow (resulting value larger than 2**256 - 1) happens here.
Counterexample:
x = 1, y = 115792089237316195423570985008687907853269984665640564039457584007913129639935
 = 0

Transaction trace:
Overflow.constructor(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935)
State: x = 1, y =␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935
Overflow.stateAdd()
    Overflow.add(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935) --␣
↪internal call
 --> o.sol:9:20:
  |
9 |             return x_ + y_;
  |                    ^^^^^^^
```

If we add `require` statements that filter out overflow cases, the SMTChecker proves that no overflow is reachable (by not reporting warnings):

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

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        require(x < type(uint128).max);
        require(y < type(uint128).max);
        return add(x, y);
    }
}
```

Assert

An assertion represents an invariant in your code: a property that must be true *for all transactions, including all input and storage values*, otherwise there is a bug.

The code below defines a function `f` that guarantees no overflow. Function `inv` defines the specification that `f` is monotonically increasing: for every possible pair `(a, b)`, if `b > a` then `f(b) > f(a)`. Since `f` is indeed monotonically increasing, the SMTChecker proves that our property is correct. You are encouraged to play with the property and the function definition to see what results come out!

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

contract Monotonic {
    function f(uint x) internal pure returns (uint) {
        require(x < type(uint128).max);
        return x * 42;
    }

    function inv(uint a, uint b) public pure {
        require(b > a);
        assert(f(b) > f(a));
    }
}
```

We can also add assertions inside loops to verify more complicated properties. The following code searches for the maximum element of an unrestricted array of numbers, and asserts the property that the found element must be greater or equal every element in the array.

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

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            assert(m >= a[i]);

        return m;
    }
}
```

Note that in this example the SMTChecker will automatically try to prove three properties:

1. ++i in the first loop does not overflow.
2. ++i in the second loop does not overflow.
3. The assertion is always true.

Not: The properties involve loops, which makes it *much much* harder than the previous examples, so beware of loops!

All the properties are correctly proven safe. Feel free to change the properties and/or add restrictions on the array to see different results. For example, changing the code to

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

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        require(a.length >= 5);
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m > a[i]);

        return m;
    }
}
```

gives us:

```
Warning: CHC: Assertion violation happens here.
Counterexample:

a = [0, 0, 0, 0, 0]
 = 0

Transaction trace:
Test.constructor()
Test.max([0, 0, 0, 0, 0])
  --> max.sol:14:4:
   |
14 |            assert(m > a[i]);
```

State Properties

So far the examples only demonstrated the use of the SMTChecker over pure code, proving properties about specific operations or algorithms. A common type of properties in smart contracts are properties that involve the state of the contract. Multiple transactions might be needed to make an assertion fail for such a property.

As an example, consider a 2D grid where both axis have coordinates in the range (-2^128, 2^128 - 1). Let us place a robot at position (0, 0). The robot can only move diagonally, one step at a time, and cannot move outside the grid. The robot's state machine can be represented by the smart contract below.

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

contract Robot {
    int x = 0;
    int y = 0;

    modifier wall {
        require(x > type(int128).min && x < type(int128).max);
        require(y > type(int128).min && y < type(int128).max);
        _;
    }

    function moveLeftUp() wall public {
        --x;
        ++y;
    }

    function moveLeftDown() wall public {
        --x;
        --y;
    }

    function moveRightUp() wall public {
        ++x;
        ++y;
    }

    function moveRightDown() wall public {
        ++x;
        --y;
    }

    function inv() public view {
        assert((x + y) % 2 == 0);
    }
}
```

Function `inv` represents an invariant of the state machine that `x + y` must be even. The SMTChecker manages to prove that regardless how many commands we give the robot, even if infinitely many, the invariant can *never* fail. The interested reader may want to prove that fact manually as well. Hint: this invariant is inductive.

We can also trick the SMTChecker into giving us a path to a certain position we think might be reachable. We can add the property that (2, 4) is *not* reachable, by adding the following function.

```
function reach_2_4() public view {
    assert(!(x == 2 && y == 4));
}
```

This property is false, and while proving that the property is false, the SMTChecker tells us exactly *how* to reach (2, 4):

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 2, y = 4

Transaction trace:
Robot.constructor()
State: x = 0, y = 0
Robot.moveLeftUp()
State: x = (- 1), y = 1
Robot.moveRightUp()
State: x = 0, y = 2
Robot.moveRightUp()
State: x = 1, y = 3
Robot.moveRightUp()
State: x = 2, y = 4
Robot.reach_2_4()
  --> r.sol:35:4:
   |
35 |            assert(!(x == 2 && y == 4));
   |            ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Note that the path above is not necessarily deterministic, as there are other paths that could reach (2, 4). The choice of which path is shown might change depending on the used solver, its version, or just randomly.

External Calls and Reentrancy

Every external call is treated as a call to unknown code by the SMTChecker. The reasoning behind that is that even if the code of the called contract is available at compile time, there is no guarantee that the deployed contract will indeed be the same as the contract where the interface came from at compile time.

In some cases, it is possible to automatically infer properties over state variables that are still true even if the externally called code can do anything, including reenter the caller contract.

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

interface Unknown {
    function run() external;
}

contract Mutex {
    uint x;
    bool lock;

    Unknown immutable unknown;

    constructor(Unknown u) {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        require(address(u) != address(0));
        unknown = u;
    }

    modifier mutex {
        require(!lock);
        lock = true;
        _;
        lock = false;
    }

    function set(uint x_) mutex public {
        x = x_;
    }

    function run() mutex public {
        uint xPre = x;
        unknown.run();
        assert(xPre == x);
    }
}
```

The example above shows a contract that uses a mutex flag to forbid reentrancy. The solver is able to infer that when `unknown.run()` is called, the contract is already "locked", so it would not be possible to change the value of `x`, regardless of what the unknown called code does.

If we "forget" to use the `mutex` modifier on function `set`, the SMTChecker is able to synthesize the behaviour of the externally called code so that the assertion fails:

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 1, lock = true, unknown = 1

Transaction trace:
Mutex.constructor(1)
State: x = 0, lock = false, unknown = 1
Mutex.run()
    unknown.run() -- untrusted external call, synthesized as:
        Mutex.set(1) -- reentrant call
  --> m.sol:32:3:
   |
32 |                assert(xPre == x);
   |                ^^^^^^^^^^^^^^^^
```

3.29.2 SMTChecker Options and Tuning

Timeout

The SMTChecker uses a hardcoded resource limit (`rlimit`) chosen per solver, which is not precisely related to time. We chose the `rlimit` option as the default because it gives more determinism guarantees than time inside the solver.

This options translates roughly to "a few seconds timeout" per query. Of course many properties are very complex and need a lot of time to be solved, where determinism does not matter. If the SMTChecker does not manage to solve the contract properties with the default `rlimit`, a timeout can be given in milliseconds via the CLI option `--model-checker-timeout <time>` or the JSON option `settings.modelChecker.timeout=<time>`, where 0 means no timeout.

Verification Targets

The types of verification targets created by the SMTChecker can also be customized via the CLI option `--model-checker-target <targets>` or the JSON option `settings.modelChecker.targets=<targets>`. In the CLI case, `<targets>` is a no-space-comma-separated list of one or more verification targets, and an array of one or more targets as strings in the JSON input. The keywords that represent the targets are:

- Assertions: `assert`.
- Arithmetic underflow: `underflow`.
- Arithmetic overflow: `overflow`.
- Division by zero: `divByZero`.
- Trivial conditions and unreachable code: `constantCondition`.
- Popping an empty array: `popEmptyArray`.
- Out of bounds array/fixed bytes index access: `outOfBounds`.
- Insufficient funds for a transfer: `balance`.
- All of the above: `default` (CLI only).

A common subset of targets might be, for example: `--model-checker-targets assert,overflow`.

All targets are checked by default, except underflow and overflow for Solidity >=0.8.7.

There is no precise heuristic on how and when to split verification targets, but it can be useful especially when dealing with large contracts.

Unproved Targets

If there are any unproved targets, the SMTChecker issues one warning stating how many unproved targets there are. If the user wishes to see all the specific unproved targets, the CLI option `--model-checker-show-unproved` and the JSON option `settings.modelChecker.showUnproved = true` can be used.

Verified Contracts

By default all the deployable contracts in the given sources are analyzed separately as the one that will be deployed. This means that if a contract has many direct and indirect inheritance parents, all of them will be analyzed on their own, even though only the most derived will be accessed directly on the blockchain. This causes an unnecessary burden on the SMTChecker and the solver. To aid cases like this, users can specify which contracts should be analyzed as the deployed one. The parent contracts are of course still analyzed, but only in the context of the most derived contract, reducing the complexity of the encoding and generated queries. Note that abstract contracts are by default not analyzed as the most derived by the SMTChecker.

The chosen contracts can be given via a comma-separated list (whitespace is not allowed) of <source>:<contract> pairs in the CLI: `--model-checker-contracts "<source1.sol:contract1>,<source2.sol:contract2>,<source2.sol:contract3>"`, and via the object `settings.modelChecker.contracts` in the *JSON input*, which has the following form:

```
"contracts": {
    "source1.sol": ["contract1"],
    "source2.sol": ["contract2", "contract3"]
}
```

Reported Inferred Inductive Invariants

For properties that were proved safe with the CHC engine, the SMTChecker can retrieve inductive invariants that were inferred by the Horn solver as part of the proof. Currently two types of invariants can be reported to the user:

- Contract Invariants: these are properties over the contract's state variables that are true before and after every possible transaction that the contract may ever run. For example, `x >= y`, where `x` and `y` are a contract's state variables.
- Reentrancy Properties: they represent the behavior of the contract in the presence of external calls to unknown code. These properties can express a relation between the value of the state variables before and after the external call, where the external call is free to do anything, including making reentrant calls to the analyzed contract. Primed variables represent the state variables' values after said external call. Example: `lock -> x = x'`.

The user can choose the type of invariants to be reported using the CLI option `--model-checker-invariants "contract,reentrancy"` or as an array in the field `settings.modelChecker.invariants` in the *JSON input*. By default the SMTChecker does not report invariants.

Division and Modulo With Slack Variables

Spacer, the default Horn solver used by the SMTChecker, often dislikes division and modulo operations inside Horn rules. Because of that, by default the Solidity division and modulo operations are encoded using the constraint `a = b * d + m` where `d = a / b` and `m = a % b`. However, other solvers, such as Eldarica, prefer the syntactically precise operations. The command line flag `--model-checker-div-mod-no-slacks` and the JSON option `settings.modelChecker.divModNoSlacks` can be used to toggle the encoding depending on the used solver preferences.

Natspec Function Abstraction

Certain functions including common math methods such as `pow` and `sqrt` may be too complex to be analyzed in a fully automated way. These functions can be annotated with Natspec tags that indicate to the SMTChecker that these functions should be abstracted. This means that the body of the function is not used, and when called, the function will:

- Return a nondeterministic value, and either keep the state variables unchanged if the abstracted function is view/pure, or also set the state variables to nondeterministic values otherwise. This can be used via the annotation `/// @custom:smtchecker abstract-function-nondet`.
- Act as an uninterpreted function. This means that the semantics of the function (given by the body) are ignored, and the only property this function has is that given the same input it guarantees the same output. This is currently under development and will be available via the annotation `/// @custom:smtchecker abstract-function-uf`.

Model Checking Engines

The SMTChecker module implements two different reasoning engines, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have different characteristics. The engines are independent and every property warning states from which engine it came. Note that all the examples above with counterexamples were reported by CHC, the more powerful engine.

By default both engines are used, where CHC runs first, and every property that was not proven is passed over to BMC. You can choose a specific engine via the CLI option `--model-checker-engine {all,bmc,chc,none}` or the JSON option `settings.modelChecker.engine={all,bmc,chc,none}`.

Bounded Model Checker (BMC)

The BMC engine analyzes functions in isolation, that is, it does not take the overall behavior of the contract over multiple transactions into account when analyzing each function. Loops are also ignored in this engine at the moment. Internal function calls are inlined as long as they are not recursive, directly or indirectly. External function calls are inlined if possible. Knowledge that is potentially affected by reentrancy is erased.

The characteristics above make BMC prone to reporting false positives, but it is also lightweight and should be able to quickly find small local bugs.

Constrained Horn Clauses (CHC)

A contract's Control Flow Graph (CFG) is modelled as a system of Horn clauses, where the life cycle of the contract is represented by a loop that can visit every public/external function non-deterministically. This way, the behavior of the entire contract over an unbounded number of transactions is taken into account when analyzing any function. Loops are fully supported by this engine. Internal function calls are supported, and external function calls assume the called code is unknown and can do anything.

The CHC engine is much more powerful than BMC in terms of what it can prove, and might require more computing resources.

SMT and Horn solvers

The two engines detailed above use automated theorem provers as their logical backends. BMC uses an SMT solver, whereas CHC uses a Horn solver. Often the same tool can act as both, as seen in z3, which is primarily an SMT solver and makes Spacer available as a Horn solver, and Eldarica which does both.

The user can choose which solvers should be used, if available, via the CLI option `--model-checker-solvers {all,cvc4,smtlib2,z3}` or the JSON option `settings.modelChecker.solvers=[smtlib2,z3]`, where:

- `cvc4` is only available if the `solc` binary is compiled with it. Only BMC uses `cvc4`.
- `smtlib2` outputs SMT/Horn queries in the smtlib2 format. These can be used together with the compiler's call-back mechanism so that any solver binary from the system can be employed to synchronously return the results of the queries to the compiler. This is currently the only way to use Eldarica, for example, since it does not have a C++ API. This can be used by both BMC and CHC depending on which solvers are called.
- `z3` is available
  - if `solc` is compiled with it;
  - if a dynamic `z3` library of version 4.8.x is installed in a Linux system (from Solidity 0.7.6);
  - statically in `soljson.js` (from Solidity 0.6.9), that is, the Javascript binary of the compiler.

Not: z3 version 4.8.16 broke ABI compatibility with previous versions and cannot be used with solc <=0.8.13. If you are using z3 >=4.8.16 please use solc >=0.8.14.

Since both BMC and CHC use `z3`, and `z3` is available in a greater variety of environments, including in the browser, most users will almost never need to be concerned about this option. More advanced users might apply this option to try alternative solvers on more complex problems.

Please note that certain combinations of chosen engine and solver will lead to the SMTChecker doing nothing, for example choosing CHC and `cvc4`.

3.29.3 Abstraction and False Positives

The SMTChecker implements abstractions in an incomplete and sound way: If a bug is reported, it might be a false positive introduced by abstractions (due to erasing knowledge or using a non-precise type). If it determines that a verification target is safe, it is indeed safe, that is, there are no false negatives (unless there is a bug in the SMTChecker).

If a target cannot be proven you can try to help the solver by using the tuning options in the previous section. If you are sure of a false positive, adding `require` statements in the code with more information may also give some more power to the solver.

SMT Encoding and Types

The SMTChecker encoding tries to be as precise as possible, mapping Solidity types and expressions to their closest SMT-LIB representation, as shown in the table below.

Solidity type	SMT sort	Theories
Boolean	Bool	Bool
intN, uintN, address, bytesN, enum, contract	Integer	LIA, NIA
array, mapping, bytes, string	Tuple (Array elements, Integer length)	Datatypes, Arrays, LIA
struct	Tuple	Datatypes
other types	Integer	LIA

Types that are not yet supported are abstracted by a single 256-bit unsigned integer, where their unsupported operations are ignored.

For more details on how the SMT encoding works internally, see the paper SMT-based Verification of Solidity Smart Contracts.

Function Calls

In the BMC engine, function calls to the same contract (or base contracts) are inlined when possible, that is, when their implementation is available. Calls to functions in other contracts are not inlined even if their code is available, since we cannot guarantee that the actual deployed code is the same.

The CHC engine creates nonlinear Horn clauses that use summaries of the called functions to support internal function calls. External function calls are treated as calls to unknown code, including potential reentrant calls.

Complex pure functions are abstracted by an uninterpreted function (UF) over the arguments.

Functions	BMC/CHC behavior
`assert`	Verification target.
`require`	Assumption.
internal call	BMC: Inline function call. CHC: Function summaries.
external call to known code	BMC: Inline function call or erase knowledge about state variables and local storage references. CHC: Assume called code is unknown. Try to infer invariants that hold after the call returns.
Storage array push/pop	Supported precisely. Checks whether it is popping an empty array.
ABI functions	Abstracted with UF.
`addmod`, `mulmod`	Supported precisely.
`gasleft`, `blockhash`, `keccak256`, `ecrecover` `ripemd160`	Abstracted with UF.
pure functions without implementation (external or complex)	Abstracted with UF
external functions without implementation	BMC: Erase state knowledge and assume result is nondeterminisc. CHC: Nondeterministic summary. Try to infer invariants that hold after the call returns.
transfer	BMC: Checks whether the contract's balance is sufficient. CHC: does not yet perform the check.
others	Currently unsupported

Using abstraction means loss of precise knowledge, but in many cases it does not mean loss of proving power.

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

contract Recover
{
    function f(
        bytes32 hash,
        uint8 v1, uint8 v2,
        bytes32 r1, bytes32 r2,
        bytes32 s1, bytes32 s2
    ) public pure returns (address) {
        address a1 = ecrecover(hash, v1, r1, s1);
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        require(v1 == v2);
        require(r1 == r2);
        require(s1 == s2);
        address a2 = ecrecover(hash, v2, r2, s2);
        assert(a1 == a2);
        return a1;
    }
}
```

In the example above, the SMTChecker is not expressive enough to actually compute `ecrecover`, but by modelling the function calls as uninterpreted functions we know that the return value is the same when called on equivalent parameters. This is enough to prove that the assertion above is always true.

Abstracting a function call with an UF can be done for functions known to be deterministic, and can be easily done for pure functions. It is however difficult to do this with general external functions, since they might depend on state variables.

Reference Types and Aliasing

Solidity implements aliasing for reference types with the same *data location*. That means one variable may be modified through a reference to the same data area. The SMTChecker does not keep track of which references refer to the same data. This implies that whenever a local reference or state variable of reference type is assigned, all knowledge regarding variables of the same type and data location is erased. If the type is nested, the knowledge removal also includes all the prefix base types.

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

contract Aliasing
{
    uint[] array1;
    uint[][] array2;
    function f(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint[] storage d
    ) internal {
        array1[0] = 42;
        a[0] = 2;
        c[0][0] = 2;
        b[0] = 1;
        // Erasing knowledge about memory references should not
        // erase knowledge about state variables.
        assert(array1[0] == 42);
        // However, an assignment to a storage reference will erase
        // storage knowledge accordingly.
        d[0] = 2;
        // Fails as false positive because of the assignment above.
        assert(array1[0] == 42);
        // Fails because `a == b` is possible.
        assert(a[0] == 2);
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        // Fails because `c[i] == b` is possible.
        assert(c[0][0] == 2);
        assert(d[0] == 2);
        assert(b[0] == 1);
    }
    function g(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint x
    ) public {
        f(a, b, c, array2[x]);
    }
}
```

After the assignment to `b[0]`, we need to clear knowledge about `a` since it has the same type (`uint[]`) and data location (memory). We also need to clear knowledge about `c`, since its base type is also a `uint[]` located in memory. This implies that some `c[i]` could refer to the same data as `b` or `a`.

Notice that we do not clear knowledge about `array` and `d` because they are located in storage, even though they also have type `uint[]`. However, if `d` was assigned, we would need to clear knowledge about `array` and vice-versa.

Contract Balance

A contract may be deployed with funds sent to it, if `msg.value` > 0 in the deployment transaction. However, the contract's address may already have funds before deployment, which are kept by the contract. Therefore, the SMTChecker assumes that `address(this).balance >= msg.value` in the constructor in order to be consistent with the EVM rules. The contract's balance may also increase without triggering any calls to the contract, if

- `selfdestruct` is executed by another contract with the analyzed contract as the target of the remaining funds,
- the contract is the coinbase (i.e., `block.coinbase`) of some block.

To model this properly, the SMTChecker assumes that at every new transaction the contract's balance may grow by at least `msg.value`.

3.29.4 Real World Assumptions

Some scenarios can be expressed in Solidity and the EVM, but are expected to never occur in practice. One of such cases is the length of a dynamic storage array overflowing during a push: If the `push` operation is applied to an array of length 2^256 - 1, its length silently overflows. However, this is unlikely to happen in practice, since the operations required to grow the array to that point would take billions of years to execute. Another similar assumption taken by the SMTChecker is that an address' balance can never overflow.

A similar idea was presented in EIP-1985.

3.30 Kaynaklar

3.30.1 Genel Kaynaklar

- Ethereum.org Geliştirici Portalı
- Ethereum StackExchange
- Solidity Portal
- Solidity Değişiklik Günlüğü
- Solidity Kaynak Kodu GitHub'da
- Solidity Language Users Chat
- Solidity Compiler Developers Chat
- Awesome Solidity
- Solidity by Example
- Solidity Dokümantasyon Topluluğu Çevirileri

3.30.2 Entegre (Ethereum) Geliştirme Ortamları

- Brownie Ethereum Sanal Makinesini hedefleyen akıllı sözleşmeler için Python tabanlı geliştirme ve test çerçevesi.
- Dapp Komut satırından akıllı sözleşmeler oluşturmak, test etmek ve dağıtmak için bir araç.
- Embark Merkezi olmayan uygulamalar oluşturmak ve dağıtmak için geliştirici platformu.
- Foundry Rust ile yazılmış Ethereum uygulama geliştirme için hızlı, taşınabilir ve modüler araç seti.
- Hardhat Yerel Ethereum ağı, hata ayıklama özellikleri ve eklenti ekosistemi ile Ethereum geliştirme ortamı.
- Remix Sunucu tarafı bileşenleri olmayan entegre derleyici ve Solidity çalışma zamanı ortamına sahip tarayıcı tabanlı IDE.
- Truffle Ethereum geliştirme çerçevesi.

3.30.3 Editör Entegrasyonları

- Atom
  - **Etheratom**
    Sözdizimi vurgulama, derleme ve çalışma zamanı ortamı (Backend node ve VM uyumlu) içeren Atom editörü için eklenti.
  - **Atom Solidity Linter**
    Solidity linting sağlayan Atom editörü için eklenti.
  - **Atom Solium Linter**
    Solium'u (şimdi Ethlint) temel olarak kullanan Atom için yapılandırılabilir Solidity linter.
- Emacs
  - **Emacs Solidity**
    Emacs editörü için sözdizimi vurgulama ve derleme hatası raporlama sağlayan eklenti.
- IntelliJ

  - **IntelliJ IDEA eklentisi**
    IntelliJ IDEA (ve diğer tüm JetBrains IDE'leri) için Solidity eklentisi

- Sublime
  - **SublimeText için paket - Solidity dil sözdizimi**
    SublimeText editörü için Solidity sözdizimi vurgulama.

- Vim
  - **Vim Solidity**
    Vim düzenleyicisi için sözdizimi vurgulama sağlayan eklenti.
  - **Vim Syntastic**
    Derleme denetimi sağlayan Vim düzenleyicisi için eklenti.

- Visual Studio Code
  - **Visual Studio Kod uzantısı**
    Microsoft Visual Studio Code için sözdizimi vurgulama ve Solidity derleyicisi içeren Solidity eklentisi.
  - **Solidity Görsel Denetçi uzantısı**
    Visual Studio Code'a güvenlik merkezli sözdizimi ve anlamsal vurgulama ekler.

3.30.4 Solidity Araçları

- **ABI - Solidity arayüz dönüştürücüsü**
  Akıllı bir sözleşmenin ABI'sinden sözleşme arayüzleri oluşturmak için bir betik.
- **abi-to-sol**
  Belirli bir ABI JSON'dan Solidity arayüz kaynağı oluşturmak için araç.
- **Doxity**
  Solidity için Dokümantasyon Oluşturucu.
- **Ethlint**
  Solidity'deki stil ve güvenlik sorunlarını tanımlamak ve düzeltmek için Linter.
- **evmdis**
  Ham EVM işlemlerinden daha yüksek bir soyutlama düzeyi sağlamak için bytecode üzerinde statik analiz gerçekleştiren EVM Disassembler.
- **EVM Lab**
  EVM ile etkileşim için zengin araç paketi. Bir VM, Etherchain API ve gaz maliyeti göstergeli bir izleme görüntüleyici içerir.
- **hevm**
  EVM hata ayıklayıcı ve sembolik yürütme motoru.
- **leafleth**
  Solidity akıllı sözleşmeleri için bir dokümantasyon oluşturucu.
- **PIET**
  Basit bir grafik arayüz aracılığıyla Solidity akıllı sözleşmelerini geliştirmek, denetlemek ve kullanmak için bir araç.
- **Scaffold-ETH**
  Hızlı ürün yinelemelerine odaklanan forklanabilir Ethereum geliştirme yığını.
- **sol2uml**
  Solidity sözleşmeleri için Birleşik Modelleme Dili (UML) sınıf diyagramı oluşturucu.

- **solc-select**
  Solidity derleyici sürümleri arasında hızlıca geçiş yapmak için bir betik.
- **Solidity prettier eklentisi**
  Solidity için prettier.
- **Solidity REPL**
  Solidity'yi bir komut satırı Solidity konsolu ile anında deneyin.
- **solgraph**
  Solidity kontrol akışını görselleştirin ve potansiyel güvenlik açıklarını vurgulayın.
- **Solhint**
  Akıllı sözleşme doğrulaması için güvenlik, stil kılavuzu ve en iyi uygulama kuralları sağlayan Solidity linter.
- **Sourcify**
  Merkezi olmayan otomatik sözleşme doğrulama hizmeti ve sözleşme meta verilerinin halka açık deposu.
- **Sūrya**
  Akıllı sözleşme sistemleri için bir dizi görsel çıktı ve sözleşmelerin yapısı hakkında bilgi sunan yardımcı araç. Ayrıca fonksiyon çağrı grafiğini sorgulamayı da destekler.
- **Universal Mutator**
  Yapılandırılabilir kurallar ve Solidity ve Vyper desteği ile mutasyon üretimi için bir araç.

3.30.5 Üçüncü Parti Solidity Ayrıştırıcıları ve Gramerleri

- **Solidity Parser for JavaScript**
  Sağlam bir ANTLR4 gramerinin üzerine inşa edilmiş JS için bir Solidity ayrıştırıcısı.

3.31 Import Path Resolution

In order to be able to support reproducible builds on all platforms, the Solidity compiler has to abstract away the details of the filesystem where source files are stored. Paths used in imports must work the same way everywhere while the command-line interface must be able to work with platform-specific paths to provide good user experience. This section aims to explain in detail how Solidity reconciles these requirements.

3.31.1 Virtual Filesystem

The compiler maintains an internal database (*virtual filesystem* or *VFS* for short) where each source unit is assigned a unique *source unit name* which is an opaque and unstructured identifier. When you use the *import statement*, you specify an *import path* that references a source unit name.

Import Callback

The VFS is initially populated only with files the compiler has received as input. Additional files can be loaded during compilation using an *import callback*, which is different depending on the type of compiler you use (see below). If the compiler does not find any source unit name matching the import path in the VFS, it invokes the callback, which is responsible for obtaining the source code to be placed under that name. An import callback is free to interpret source unit names in an arbitrary way, not just as paths. If there is no callback available when one is needed or if it fails to locate the source code, compilation fails.

The command-line compiler provides the *Host Filesystem Loader* - a rudimentary callback that interprets a source unit name as a path in the local filesystem. The JavaScript interface does not provide any by default, but one can be provided by the user. This mechanism can be used to obtain source code from locations other then the local filesystem (which may not even be accessible, e.g. when the compiler is running in a browser). For example the Remix IDE provides a versatile callback that lets you import files from HTTP, IPFS and Swarm URLs or refer directly to packages in NPM registry.

Not: Host Filesystem Loader's file lookup is platform-dependent. For example backslashes in a source unit name can be interpreted as directory separators or not and the lookup can be case-sensitive or not, depending on the underlying platform.

For portability it is recommended to avoid using import paths that will work correctly only with a specific import callback or only on one platform. For example you should always use forward slashes since they work as path separators also on platforms that support backslashes.

Initial Content of the Virtual Filesystem

The initial content of the VFS depends on how you invoke the compiler:

1. **solc / command-line interface**

   When you compile a file using the command-line interface of the compiler, you provide one or more paths to files containing Solidity code:

   ```
   solc contract.sol /usr/local/dapp-bin/token.sol
   ```

   The source unit name of a file loaded this way is constructed by converting its path to a canonical form and, if possible, making it relative to either the base path or one of the include paths. See *CLI Path Normalization and Stripping* for a detailed description of this process.

2. **Standard JSON**

   When using the *Standard JSON* API (via either the JavaScript interface or the `--standard-json` command-line option) you provide input in JSON format, containing, among other things, the content of all your source files:

   ```
   {
       "language": "Solidity",
       "sources": {
           "contract.sol": {
               "content": "import \"./util.sol\";\ncontract C {}"
           },
           "util.sol": {
               "content": "library Util {}"
           },
           "/usr/local/dapp-bin/token.sol": {
   ```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            "content": "contract Token {}"
        }
    },
    "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
}
```

The `sources` dictionary becomes the initial content of the virtual filesystem and its keys are used as source unit names.

3. **Standard JSON (via import callback)**

   With Standard JSON it is also possible to tell the compiler to use the import callback to obtain the source code:

```
{
    "language": "Solidity",
    "sources": {
        "/usr/local/dapp-bin/token.sol": {
            "urls": [
                "/projects/mytoken.sol",
                "https://example.com/projects/mytoken.sol"
            ]
        }
    },
    "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
}
```

   If an import callback is available, the compiler will give it the strings specified in `urls` one by one, until one is loaded successfully or the end of the list is reached.

   The source unit names are determined the same way as when using `content` - they are keys of the `sources` dictionary and the content of `urls` does not affect them in any way.

4. **Standard input**

   On the command line it is also possible to provide the source by sending it to compiler's standard input:

```
echo 'import "./util.sol"; contract C {}' | solc -
```

   `-` used as one of the arguments instructs the compiler to place the content of the standard input in the virtual filesystem under a special source unit name: `<stdin>`.

Once the VFS is initialized, additional files can still be added to it only through the import callback.

3.31.2 Imports

The import statement specifies an *import path*. Based on how the import path is specified, we can divide imports into two categories:

- *Direct imports*, where you specify the full source unit name directly.
- *Relative imports*, where you specify a path starting with `./` or `../` to be combined with the source unit name of the importing file.

Listeleme 1: contracts/contract.sol

```
import "./math/math.sol";
import "contracts/tokens/token.sol";
```

In the above `./math/math.sol` and `contracts/tokens/token.sol` are import paths while the source unit names they translate to are `contracts/math/math.sol` and `contracts/tokens/token.sol` respectively.

Direct Imports

An import that does not start with `./` or `../` is a *direct import*.

```
import "/project/lib/util.sol";         // source unit name: /project/lib/util.sol
import "lib/util.sol";                  // source unit name: lib/util.sol
import "@openzeppelin/address.sol";     // source unit name: @openzeppelin/address.sol
import "https://example.com/token.sol"; // source unit name: https://example.com/token.
↪sol
```

After applying any *import remappings* the import path simply becomes the source unit name.

Not: A source unit name is just an identifier and even if its value happens to look like a path, it is not subject to the normalization rules you would typically expect in a shell. Any `/./` or `/../` segments or sequences of multiple slashes remain a part of it. When the source is provided via Standard JSON interface it is entirely possible to associate different content with source unit names that would refer to the same file on disk.

When the source is not available in the virtual filesystem, the compiler passes the source unit name to the import callback. The Host Filesystem Loader will attempt to use it as a path and look up the file on disk. At this point the platform-specific normalization rules kick in and names that were considered different in the VFS may actually result in the same file being loaded. For example `/project/lib/math.sol` and `/project/lib/../lib///math.sol` are considered completely different in the VFS even though they refer to the same file on disk.

Not: Even if an import callback ends up loading source code for two different source unit names from the same file on disk, the compiler will still see them as separate source units. It is the source unit name that matters, not the physical location of the code.

Relative Imports

An import starting with `./` or `../` is a *relative import*. Such imports specify a path relative to the source unit name of the importing source unit:

Listeleme 2: /project/lib/math.sol

```
import "./util.sol" as util;    // source unit name: /project/lib/util.sol
import "../token.sol" as token; // source unit name: /project/token.sol
```

Listeleme 3: lib/math.sol

```
import "./util.sol" as util;    // source unit name: lib/util.sol
import "../token.sol" as token; // source unit name: token.sol
```

Not: Relative imports **always** start with `./` or `../` so `import "util.sol"`, unlike `import "./util.sol"`, is a direct import. While both paths would be considered relative in the host filesystem, `util.sol` is actually absolute in the VFS.

Let us define a *path segment* as any non-empty part of the path that does not contain a separator and is bounded by two path separators. A separator is a forward slash or the beginning/end of the string. For example in `./abc/..//` there are three path segments: `.`, `abc` and `..`.

The compiler computes a source unit name from the import path in the following way:

1. First a prefix is computed
    - Prefix is initialized with the source unit name of the importing source unit.
    - The last path segment with preceding slashes is removed from the prefix.
    - Then, the leading part of the normalized import path, consisting only of `/` and `.` characters is considered. For every `..` segment found in this part the last path segment with preceding slashes is removed from the prefix.
2. Then the prefix is prepended to the normalized import path. If the prefix is non-empty, a single slash is inserted between it and the import path.

The removal of the last path segment with preceding slashes is understood to work as follows:

1. Everything past the last slash is removed (i.e. `a/b//c.sol` becomes `a/b//`).
2. All trailing slashes are removed (i.e. `a/b//` becomes `a/b`).

The normalization rules are the same as for UNIX paths, namely:

- All the internal `.` segments are removed.
- Every internal `..` segment backtracks one level up in the hierarchy.
- Multiple slashes are squashed into a single one.

Note that normalization is performed only on the import path. The source unit name of the importing module that is used for the prefix remains unnormalized. This ensures that the `protocol://` part does not turn into `protocol:/` if the importing file is identified with a URL.

If your import paths are already normalized, you can expect the above algorithm to produce very intuitive results. Here are some examples of what you can expect if they are not:

Listeleme 4: lib/src/../contract.sol

```
import "./util/./util.sol";         // source unit name: lib/src/../util/util.sol
import "./util//util.sol";          // source unit name: lib/src/../util/util.sol
import "../util/../array/util.sol"; // source unit name: lib/src/array/util.sol
import "../../../../util.sol";      // source unit name: util.sol
import "../../../../../util.sol";   // source unit name: util.sol
```

Not: The use of relative imports containing leading `..` segments is not recommended. The same effect can be achieved in a more reliable way by using direct imports with *base path and include paths*.

3.31.3 Base Path and Include Paths

The base path and include paths represent directories that the Host Filesystem Loader will load files from. When a source unit name is passed to the loader, it prepends the base path to it and performs a filesystem lookup. If the lookup does not succeed, the same is done with all directories on the include path list.

It is recommended to set the base path to the root directory of your project and use include paths to specify additional locations that may contain libraries your project depends on. This lets you import from these libraries in a uniform way, no matter where they are located in the filesystem relative to your project. For example, if you use npm to install packages and your contract imports `@openzeppelin/contracts/utils/Strings.sol`, you can use these options to tell the compiler that the library can be found in one of the npm package directories:

```
solc contract.sol \
    --base-path . \
    --include-path node_modules/ \
    --include-path /usr/local/lib/node_modules/
```

Your contract will compile (with the same exact metadata) no matter whether you install the library in the local or global package directory or even directly under your project root.

By default the base path is empty, which leaves the source unit name unchanged. When the source unit name is a relative path, this results in the file being looked up in the directory the compiler has been invoked from. It is also the only value that results in absolute paths in source unit names being actually interpreted as absolute paths on disk. If the base path itself is relative, it is interpreted as relative to the current working directory of the compiler.

Not: Include paths cannot have empty values and must be used together with a non-empty base path.

Not: Include paths and base path can overlap as long as it does not make import resolution ambiguous. For example, you can specify a directory inside base path as an include directory or have an include directory that is a subdirectory of another include directory. The compiler will only issue an error if the source unit name passed to the Host Filesystem Loader represents an existing path when combined with multiple include paths or an include path and base path.

CLI Path Normalization and Stripping

On the command line the compiler behaves just as you would expect from any other program: it accepts paths in a format native to the platform and relative paths are relative to the current working directory. The source unit names assigned to files whose paths are specified on the command line, however, should not change just because the project is being compiled on a different platform or because the compiler happens to have been invoked from a different directory. To achieve this, paths to source files coming from the command line must be converted to a canonical form, and, if possible, made relative to the base path or one of the include paths.

The normalization rules are as follows:

- If a path is relative, it is made absolute by prepending the current working directory to it.
- Internal `.` and `..` segments are collapsed.
- Platform-specific path separators are replaced with forward slashes.

- Sequences of multiple consecutive path separators are squashed into a single separator (unless they are the leading slashes of an UNC path).
- If the path includes a root name (e.g. a drive letter on Windows) and the root is the same as the root of the current working directory, the root is replaced with `/`.
- Symbolic links in the path are **not** resolved.
  - The only exception is the path to the current working directory prepended to relative paths in the process of making them absolute. On some platforms the working directory is reported always with symbolic links resolved so for consistency the compiler resolves them everywhere.
- The original case of the path is preserved even if the filesystem is case-insensitive but case-preserving and the actual case on disk is different.

Not: There are situations where paths cannot be made platform-independent. For example on Windows the compiler can avoid using drive letters by referring to the root directory of the current drive as `/` but drive letters are still necessary for paths leading to other drives. You can avoid such situations by ensuring that all the files are available within a single directory tree on the same drive.

After normalization the compiler attempts to make the source file path relative. It tries the base path first and then the include paths in the order they were given. If the base path is empty or not specified, it is treated as if it was equal to the path to the current working directory (with all symbolic links resolved). The result is accepted only if the normalized directory path is the exact prefix of the normalized file path. Otherwise the file path remains absolute. This makes the conversion unambiguous and ensures that the relative path does not start with `../`. The resulting file path becomes the source unit name.

Not: The relative path produced by stripping must remain unique within the base path and include paths. For example the compiler will issue an error for the following command if both `/project/contract.sol` and `/lib/contract.sol` exist:

```
solc /project/contract.sol --base-path /project --include-path /lib
```

Not: Prior to version 0.8.8, CLI path stripping was not performed and the only normalization applied was the conversion of path separators. When working with older versions of the compiler it is recommended to invoke the compiler from the base path and to only use relative paths on the command line.

3.31.4 Allowed Paths

As a security measure, the Host Filesystem Loader will refuse to load files from outside of a few locations that are considered safe by default:

- Outside of Standard JSON mode:
  - The directories containing input files listed on the command line.
  - The directories used as *remapping* targets. If the target is not a directory (i.e does not end with `/`, `/.` or `/..`) the directory containing the target is used instead.
  - Base path and include paths.
- In Standard JSON mode:
  - Base path and include paths.

Additional directories can be whitelisted using the `--allow-paths` option. The option accepts a comma-separated list of paths:

```
cd /home/user/project/
solc token/contract.sol \
    lib/util.sol=libs/util.sol \
    --base-path=token/ \
    --include-path=/lib/ \
    --allow-paths=../utils/,/tmp/libraries
```

When the compiler is invoked with the command shown above, the Host Filesystem Loader will allow importing files from the following directories:

- `/home/user/project/token/` (because `token/` contains the input file and also because it is the base path),
- `/lib/` (because `/lib/` is one of the include paths),
- `/home/user/project/libs/` (because `libs/` is a directory containing a remapping target),
- `/home/user/utils/` (because of `../utils/` passed to `--allow-paths`),
- `/tmp/libraries/` (because of `/tmp/libraries` passed to `--allow-paths`),

Not: The working directory of the compiler is one of the paths allowed by default only if it happens to be the base path (or the base path is not specified or has an empty value).

Not: The compiler does not check if allowed paths actually exist and whether they are directories. Non-existent or empty paths are simply ignored. If an allowed path matches a file rather than a directory, the file is considered whitelisted, too.

Not: Allowed paths are case-sensitive even if the filesystem is not. The case must exactly match the one used in your imports. For example `--allow-paths tokens` will not match `import "Tokens/IERC20.sol"`.

Uyarı: Files and directories only reachable through symbolic links from allowed directories are not automatically whitelisted. For example if `token/contract.sol` in the example above was actually a symlink pointing at `/etc/passwd` the compiler would refuse to load it unless `/etc/` was one of the allowed paths too.

3.31.5 Import Remapping

Import remapping allows you to redirect imports to a different location in the virtual filesystem. The mechanism works by changing the translation between import paths and source unit names. For example you can set up a remapping so that any import from the virtual directory `github.com/ethereum/dapp-bin/library/` would be seen as an import from `dapp-bin/library/` instead.

You can limit the scope of a remapping by specifying a *context*. This allows creating remappings that apply only to imports located in a specific library or a specific file. Without a context a remapping is applied to every matching import in all the files in the virtual filesystem.

Import remappings have the form of `context:prefix=target`:

- `context` must match the beginning of the source unit name of the file containing the import.

- `prefix` must match the beginning of the source unit name resulting from the import.
- `target` is the value the prefix is replaced with.

For example, if you clone https://github.com/ethereum/dapp-bin/ locally to `/project/dapp-bin` and run the compiler with:

```
solc github.com/ethereum/dapp-bin/=dapp-bin/ --base-path /project source.sol
```

you can use the following in your source file:

```
import "github.com/ethereum/dapp-bin/library/math.sol"; // source unit name: dapp-bin/
↪library/math.sol
```

The compiler will look for the file in the VFS under `dapp-bin/library/math.sol`. If the file is not available there, the source unit name will be passed to the Host Filesystem Loader, which will then look in `/project/dapp-bin/library/iterable_mapping.sol`.

Uyarı: Information about remappings is stored in contract metadata. Since the binary produced by the compiler has a hash of the metadata embedded in it, any modification to the remappings will result in different bytecode.

For this reason you should be careful not to include any local information in remapping targets. For example if your library is located in `/home/user/packages/mymath/math.sol`, a remapping like `@math/=/home/user/packages/mymath/` would result in your home directory being included in the metadata. To be able to reproduce the same bytecode with such a remapping on a different machine, you would need to recreate parts of your local directory structure in the VFS and (if you rely on Host Filesystem Loader) also in the host filesystem.

To avoid having your local directory structure embedded in the metadata, it is recommended to designate the directories containing libraries as *include paths* instead. For example, in the example above `--include-path /home/user/packages/` would let you use imports starting with `mymath/`. Unlike remapping, the option on its own will not make `mymath` appear as `@math` but this can be achieved by creating a symbolic link or renaming the package subdirectory.

As a more complex example, suppose you rely on a module that uses an old version of dapp-bin that you checked out to `/project/dapp-bin_old`, then you can run:

```
solc module1:github.com/ethereum/dapp-bin/=dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=dapp-bin_old/ \
     --base-path /project \
     source.sol
```

This means that all imports in `module2` point to the old version but imports in `module1` point to the new version.

Here are the detailed rules governing the behaviour of remappings:

1. **Remappings only affect the translation between import paths and source unit names.**

   Source unit names added to the VFS in any other way cannot be remapped. For example the paths you specify on the command-line and the ones in `sources.urls` in Standard JSON are not affected.

   ```
   solc /project/=/contracts/ /project/contract.sol # source unit name: /project/
   ↪contract.sol
   ```

   In the example above the compiler will load the source code from `/project/contract.sol` and place it under that exact source unit name in the VFS, not under `/contract/contract.sol`.

2. **Context and prefix must match source unit names, not import paths.**

- This means that you cannot remap `./` or `../` directly since they are replaced during the translation to source unit name but you can remap the part of the name they are replaced with:

```
solc ./=a/ /project/=b/ /project/contract.sol # source unit name: /project/
↪contract.sol
```

Listeleme 5: /project/contract.sol

```
import "./util.sol" as util; // source unit name: b/util.sol
```

- You cannot remap base path or any other part of the path that is only added internally by an import callback:

```
solc /project/=/contracts/ /project/contract.sol --base-path /project # source␣
↪unit name: contract.sol
```

Listeleme 6: /project/contract.sol

```
import "util.sol" as util; // source unit name: util.sol
```

3. **Target is inserted directly into the source unit name and does not necessarily have to be a valid path.**
   - It can be anything as long as the import callback can handle it. In case of the Host Filesystem Loader this includes also relative paths. When using the JavaScript interface you can even use URLs and abstract identifiers if your callback can handle them.
   - Remapping happens after relative imports have already been resolved into source unit names. This means that targets starting with `./` and `../` have no special meaning and are relative to the base path rather than to the location of the source file.
   - Remapping targets are not normalized so `@root/=./a/b//` will remap `@root/contract.sol` to `./a/b//contract.sol` and not `a/b/contract.sol`.
   - If the target does not end with a slash, the compiler will not add one automatically:

```
solc /project/=/contracts /project/contract.sol # source unit name: /project/
↪contract.sol
```

Listeleme 7: /project/contract.sol

```
import "/project/util.sol" as util; // source unit name: /contractsutil.sol
```

4. **Context and prefix are patterns and matches must be exact.**
   - `a//b=c` will not match `a/b`.
   - source unit names are not normalized so `a/b=c` will not match `a//b` either.
   - Parts of file and directory names can match as well. `/newProject/con:/new=old` will match `/newProject/contract.sol` and remap it to `oldProject/contract.sol`.
5. **At most one remapping is applied to a single import.**
   - If multiple remappings match the same source unit name, the one with the longest matching prefix is chosen.
   - If prefixes are identical, the one specified last wins.
   - Remappings do not work on other remappings. For example `a=b b=c c=d` will not result in `a` being remapped to `d`.
6. **Prefix cannot be empty but context and target are optional.**

- If `target` is the empty string, `prefix` is simply removed from import paths.
- Empty `context` means that the remapping applies to all imports in all source units.

3.31.6 Using URLs in imports

Most URL prefixes such as `https://` or `data://` have no special meaning in import paths. The only exception is `file://` which is stripped from source unit names by the Host Filesystem Loader.

When compiling locally you can use import remapping to replace the protocol and domain part with a local path:

```
solc :https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin contract.sol
```

Note the leading `:`, which is necessary when the remapping context is empty. Otherwise the `https:` part would be interpreted by the compiler as the context.

3.32 Yul

Yul (önceden JULIA veya IULIA olarak da adlandırılmıştır), farklı backend'ler için bayt koduna derlenebilen bir ara dildir.

EVM 1.0, EVM 1.5 ve Ewasm desteği planlanmış olup, her üç platformda kullanılabilir bir ortak paydası olacak şekilde tasarlanmıştır. Halihazırda Solidity içinde bağımsız modda ve "inline (satır-içi) Assembly" için kullanılabilir ve Solidity derleyicisinin Yul'u bir ara dil olarak kullanan deneysel bir uygulaması bulunmaktadır. Yul, üst düzey optimizasyon aşamaları için tüm hedef platformlara eşit olarak fayda sağlayabilecek iyi bir araçtır.

3.32.1 Motivasyon ve Üst-düzey Tanımı

Yul'un tasarımı birkaç hedef doğrultusunda çalışmaktadır:

1. Yul'da yazılan programlar, yazılan kod Solidity veya bir başka üst düzey dilden bir derleyici tarafından oluşturulmuş olsa bile okunabilir olmalıdır.
2. Kontrol akışı, manuel incelemeye, resmi doğrulamaya ve optimizasyona yardımcı olmak amacıyla anlaşılması kolay olmalıdır.
3. Yul'den bytecode'a çeviri mümkün olduğunca basit olmalıdır.
4. Yul, programın tüm optimizasyonu için uygun olmalıdır..

Yul, birinci ve ikinci amaca ulaşmak için `for` döngüleri, `if` ve `switch` ifadeleri ve fonksiyon çağrıları gibi üst düzey yapılar sağlar. Bunlar, assembly programları için kontrol akışını yeterince ifade edebilir olmalıdır. Bu nedenle, `SWAP`, `DUP`, `JUMPDEST`, `JUMP` ve `JUMPI` için belirgin ifadeler sağlanmamaktadır, çünkü ilk ikisi veri akışını ve son ikisi de kontrol akışını belirsizleştirmektedir. Ayrıca, `mul(add(x,y), 7)` biçimindeki fonksiyonel ifadeler, `7 y x add mul` gibi saf işlem kodu ifadelerine tercih edilir, çünkü birinci biçimde, hangi işlemcinin hangi işlem kodu için kullanıldığını görmek çok daha kolaydır.

Stack (yığın) makineleri için tasarlanmış olsa da, Yul stack karmaşıklığını ortaya çıkarmaz. Programcı veya denetçinin stack hakkında endişelenmesine gerek yoktur.

Üçüncü hedef, daha üst düzey yapılar çok düzenli bir şekilde bytecode'a derlenerek elde edilir. Derleyici tarafından gerçekleştirilen lokal olmayan tek işlem, kullanıcının atadığı tanımlayıcıların (fonksiyonlar, değişkenler, ...) ad araması ve yığından (stack) yerel değişkenlerin temizlenmesidir.

Değerler ve referanslar gibi kavramlar arasındaki karışıklığı önlemek için Yul statik olarak yazılır. Aynı zamanda, okunabilirliğe yardımcı olmak için her zaman ihmal edilebilecek bir varsayılan tür (genellikle hedef makinenin tamsayı sözcüğü) vardır.

Dili basit ve esnek tutmak için Yul, saf haliyle herhangi bir gömülü işlem, fonksiyon veya türe sahip değildir. Bunlar, Yul'un farklı hedef platformların ve özellik kümelerinin gereksinimlerine göre özelleştirilmesine izin veren bir Yul diyalekti belirlenirken semantikleriyle birlikte eklenir.

Şu anda, Yul'un belirlenmiş yalnızca bir tane diyalekti var. Bu diyalekt, gömülü fonksiyonlar olarak EVM işlem kodlarını kullanır (aşağıya bakınız) ve yalnızca EVM'nin yerel 256 bit türü olan `u256` türünü tanımlar. Bu nedenle, aşağıdaki örneklerde türlerden bahsetmeyeceğiz.

3.32.2 Basit Bir Örnek

Aşağıdaki örnek program EVM diyalektiyle yazılmıştır ve üs alma işlemini hesaplar. `solc --strict-assembly` kullanılarak derlenebilir. Yerleşik fonksiyonlar olan `mul` ve `div`, sırasıyla çarpma ve bölme işlemlerini yapar.

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

Aynı fonksiyonu özyineleme (recursion) yerine bir for döngüsü kullanarak da uygulamak mümkündür. Burada `lt(a, b)`, `a`'nın `b`'den küçük olup olmadığını hesaplar.

```
{
    function power(base, exponent) -> result
    {
        result := 1
        for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
        {
            result := mul(result, base)
        }
    }
}
```

Bölümün sonunda, ERC-20 standardı ile ilgili eksiksiz bir uygulama bulunabilir.

3.32.3 Tek Başına Kullanım

Yul'u Solidity derleyicisini kullanarak EVM diyalektinde tek başına kullanabilirsiniz. Bu, Yul nesne notasyonunu kullanır, böylece sözleşmeleri deploy etmek için koda veri olarak atıfta bulunulabilir. Bu Yul modu, komut satırı derleyicisi (`--strict-assembly` kullanın) ve *standard-json arayüzü* için kullanılabilir:

```
{
    "language": "Yul",
    "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
    "settings": {
        "outputSelection": { "*": { "*": ["*"], "": [ "*" ] } },
        "optimizer": { "enabled": true, "details": { "yul": true } }
    }
}
```

Uyarı: Yul aktif geliştirme aşamasındadır ve bayt kodu oluşturma, yalnızca hedef olarak EVM 1.0 ile Yul'un EVM diyalekti için tam olarak uygulanabilir.

3.32.4 Yul'un Resmi Olmayan Tanımı

Aşağıda Yul dilinin bütün yönleri hakkında konuşacağız. Örneklerde varsayılan EVM diyalektini kullanacağız.

Sözdizimi (Syntax)

Yul, yorumları, değişmez değerleri ve tanımlayıcıları Solidity ile aynı şekilde ayrıştırır, böylece örneğin yorumları belirtmek için `//` ve `/* */` kullanabilirsiniz. Bir istisna vardır: Yul'daki tanımlayıcılar noktalar içerebilir: `..`

Yul, kod, veri ve alt nesnelerden oluşan "nesneler" belirleyebilir. Bununla ilgili ayrıntılar için lütfen aşağıdaki *Yul Nesneleri* bölümüne bakın. Bu bölümde, bu tür bir nesnenin sadece kod kısmı ile ilgileniyoruz. Bu kod bölümü her zaman süslü parantezlerle ayrılmış bir bloktan oluşur. Çoğu araç, bir nesnenin olması beklenen yerde yalnızca bir kod bloğu tanımlamayı destekler.

Bir kod bloğu içinde aşağıdaki öğeler kullanılabilir (daha fazla ayrıntı için sonraki bölümlere bakınız):

- değişmez değerler (literal), yani `0x123`, `42` veya `"abc"` (32 karaktere kadar string'ler)
- gömülü fonksiyonlara yapılan çağrılar, örneğin `add(1, mload(0))`
- değişken tanımlamaları, örneğin `let x := 7`, `let x := add(y, 3)` veya `let x` (başlangıç değeri olarak 0 atanır)
- tanımlayıcılar (değişkenler), örneğin `add(3, x)`
- atamalar, örneğin `x := add(y, 3)`
- yerel değişkenlerin kapsam dahilinde olduğu bloklar, ör., örneğin `{ let x := 3 { let y := add(x, 1) } }`
- if ifadeleri, örneğin `if lt(a, b) { sstore(0, 1) }`
- switch ifadeleri, örneğin `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- for döngünleri, örneğin `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- fonksiyon tanımlamaları, örneğin `function f(a, b) -> c { c := add(a, b) }`

Birden fazla sözdizimsel öğe, yalnızca boşlukla ayrılmış olarak birbirini takip edebilir, yani ; ile sonlandırma yoktur veya yeni satıra geçilmelidir.

Değişmezler (Literal)

Değişmezler olarak şunları kullanabilirsiniz:

- Ondalık (decimal) veya onaltılık (hexadecimal) notasyonda tamsayı sabitleri..
- ASCII dizeleri (ör. "abc"), \xNN onaltılı çıkışlarını ve N'nin onaltılık basamaklar olduğu \uNNNN Unicode çıkışlarını içerebilir.
- Onaltılık string'ler (örneğin hex"616263").

Yul'un EVM diyalektinde, değişmezler aşağıdaki gibi 256 bitlik sözcükleri temsil eder.:

- Ondalık veya onaltılık sabitler 2**256 değerinden küçük olmalıdır. Soldan okumalı (big-endian) kodlamada bu değere sahip 256 bitlik kelimeyi işaretsiz bir tamsayı olarak temsil ederler.
- Bir ASCII string ifadesi ilk önce bir bayt dizisi olarak görüntülenir ve bunu çıkartılmamış bir ASCII karakterini değeri ASCII kodu olan tek bir bayt olarak, \xNN çıkışını bu değere sahip tek bayt olarak ve \uNNNN çıkışını o kod noktasındaki UTF-8 bayt dizisi olarak gerçekleştirir. Bayt dizisi 32 baytı geçmemelidir. Bayt dizisi, 32 bayta ulaşmak için sağdaki sıfırlarla doldurulur; başka bir deyişle, dize sola hizalı olarak saklanır. Sıfırlarla doldurulmuş bayt dizisi, en önemli 8 biti ilk bayttakilerden oluşan 256 bitlik bir kelimeyi temsil eder, yani baytlar soldan okumalı (big-endian) biçiminde yorumlanır.
- Bir onaltılık string, önce her bir bitişik onaltılık basamak çifti bir bayt olarak görüntülenecek şekilde bir bayt dizisi olarak görüntülenir. Bayt dizisi 32 baytı (yani 64 onaltılık basamak) geçmemelidir ve yukarıdaki gibi işlem görür.

EVM için derlenirken bu, uygun bir PUSHi komutuna dönüştürülecektir. Aşağıdaki örnekte, 3 ve 2 eklenerek 5 elde edilir ve ardından bitsel and ile "abc" string'i hesaplanır. Sonuç değeri, x adlı yerel bir değişkene atanır.

Yukarıdaki 32 baytlık sınır, değişmez (literal) bağımsız değişkenler gerektiren gömülü fonksiyonlara geçirilen string değişmezleri (string literal) için geçerli değildir (örneğin, setimmutable veya loadimmutable). Bu dizeler asla oluşturulan bayt kodunda bitmez.

```
let x := and("abc", add(3, 2))
```

Unless it is the default type, the type of a literal has to be specified after a colon: Varsayılan tür olmadığı sürece, bir değişmez (literal) türünün iki nokta üst üste (:) işaretinden sonra belirtilmesi gerekir:

```
// Bu derlenmeyecek (u32 ve u256 türü henüz uygulanmadı)
let x := and("abc":u32, add(3:u256, 2:u256))
```

Fonksiyon Çağrıları

Hem gömülü hem de kullanıcı tanımlı fonksiyonlar (aşağıya bakın) önceki örnekte gösterildiği gibi çağrılabilir. Fonksiyon tek bir değer döndürürse, tekrar doğrudan bir ifadenin içinde kullanılabilir. Birden fazla değer döndürürse, yerel değişkenlere atanmaları gerekir.

```
function f(x, y) -> a, b { /* ... */ }
mstore(0x80, add(mload(0x80), 3))
// Burada, kullanıcı tanımlı `f` fonksiyonu iki değer döndürür.
let x, y := f(1, mload(0))
```

EVM'nin gömülü fonksiyonları için, fonksiyonel ifadeler doğrudan bir işlem kodu akışına çevrilebilir: İşlem kodlarını elde etmek için ifadeyi sağdan sola okumanız yeterlidir. Örnekteki ilk satır söz konusu olduğunda, bu `PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE`'dur.

Kullanıcı tanımlı fonksiyonlara yapılan çağrılar için, bağımsız değişkenler de yığına sağdan sola doğru yerleştirilir ve bu, bağımsız değişken listelerinin değerlendirilme sırasıdır. Yine de, return edilen değerler yığında (stack) soldan sağa olması beklenir, yani bu örnekte, `y` yığının üstünde ve `x` onun altındadır.

Değişken Atamaları

Değişkenleri atamak için `let` anahtar sözcüğünü kullanabilirsiniz. Bir değişken sadece tanımlandığı `{...}`-blokunun içinde görünür. EVM'ye derlenirken, değişken için ayrılmış yeni bir yığın (stack) yuvası oluşturulur ve bloğun sonuna ulaşıldığında otomatik olarak tekrar kaldırılır. Değişken için bir başlangıç değeri atayabilirsiniz. Bir değer atamazsanız, değişken sıfıra eşitlenerek başlatılır.

Değişkenler yığında depolandığından, belleği veya hafızayı doğrudan etkilemezler, ancak gömülü fonksiyonlar olan `mstore`, `mload`, `sstore` ve `sload`'da belleğe veya hafıza konumlarına işaretçiler (pointer) olarak kullanılabilirler. Gelecekteki diyalektler, bu tür işaretçiler için belirlenmiş türler sağlayabilir.

Bir değişkene referans verildiğinde, mevcut değeri kopyalanır. EVM için bu, bir `DUP` talimatı anlamına gelir.

```
{
    let zero := 0
    let v := calldataload(zero)
    {
        let y := add(sload(v), 1)
        v := y
    } // y burada "serbest bırakılmıştır"
    sstore(v, zero)
} // v ve sıfır burada "serbest bırakılmıştır"
```

Atanan değişkenin varsayılan (default) türden farklı bir türde olması gerekiyorsa, iki nokta üst üste işareti ile bunu belirtirsiniz. Ayrıca, birden çok değer döndüren bir fonksiyon çağrısından atama yaptığınızda, tek bir ifadede birden çok değişken atayabilirsiniz.

```
// Bu derlenmeyecek (u32 ve u256 türü henüz uygulanmadı)
{
    let zero:u32 := 0:u32
    let v:u256, t:u32 := f()
    let x, y := g()
}
```

Optimize edici ayarlarına bağlı olarak derleyici, değişken hala kod bloğu kapsamında olsa bile, son kez kullanıldıktan sonra yığın yuvalarını serbest bırakabilir.

Atamalar

Değişkenler, tanımlarından sonra `:=` operatörü kullanılarak atanabilir. Aynı anda birden fazla değişken atamak mümkündür. Bunun için değerlerin sayı ve türlerinin eşleşmesi gerekir. Birden çok return parametresi olan bir fonksiyondan döndürülen değerleri atamak istiyorsanız, birden çok değişken tanımlamanız gerekir. Aynı değişken, bir atamanın sol tarafında birden çok kez bulunamaz, örn. `x, x := f()` geçersizdir.

```
let v := 0
// v değişkenini tekrar atama
v := 2
let t := add(v, 2)
function f() -> a, b { }
// birden çok değer atama
v, t := f()
```

If

if ifadesi, koşullu olarak kod çalıştırmak için kullanılabilir. "else" bloğu tanımlanamaz. Birden fazla alternatife ihtiyacınız varsa, bunun yerine "switch" kullanmayı düşünebilirsiniz (aşağıya göz atın).

```
if lt(calldatasize(), 4) { revert(0, 0) }
```

Kod bloğu için süslü parantez gereklidir.

Switch

if ifadesinin genişletilmiş bir versiyonu olarak bir switch ifadesi kullanabilirsiniz. Switch, bir ifadenin değerini alır ve onu birkaç değişmez sabitle karşılaştırır. Eşleşen sabite karşılık gelen kısım değerlendirmeye alınır. Diğer programlama dillerinin aksine, güvenlik nedeniyle, kontrol akışı bir durumdan diğerine devam etmez. Değişmez sabitlerin hiçbiri eşleşmezse alınan ve `default` olarak adlandırılan bir varsayılan ifade veya bir alternatif durum olabilir.

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

Switch ifadesindeki case'ler süslü parantezle çevrelenmez, ancak case'lerin kod blokları için süslü parantezle çevreleme zorunluluğu vardır.

Döngüler (Loop)

Yul, bir başlatma bölümü, bir koşul, bir iterasyon sonrası bölümü ve bir kod gövdesi içeren döngüleri destekler. Koşul bölümü bir ifade olmalıdır, diğer üçü ise bloklar şeklindedir. Başlatma bölümünde herhangi bir değişken en üst düzeyde atanırsa, bu değişkenlerin kapsamı döngünün diğer tüm bölümlerine kadar genişler.

`break` ve `continue` ifadeleri kod gövdesinde sırasıyla döngüden çıkmak veya iterasyon sonrası bölümüne atlamak için kullanılabilir.

Aşağıdaki örnek, bellekteki bir alanın toplamını hesaplar.

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

For döngüleri, while döngülerinin yerine de kullanılabilir: Başlatma ve iterasyon sonrası bölümlerini boş bırakmanız yeterlidir.

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

Fonksiyon Atamaları

Yul, fonksiyonların tanımlanmasına izin verir. Bunlar, hiçbir zaman bir sözleşmenin harici arayüzünün parçası olmadıkları ve Solidity fonksiyonlarından ayrı bir ad alanının parçası oldukları için Solidity'deki fonksiyonlarla karıştırılmamalıdır.

EVM için, Yul fonksiyonları bağımsız değişkenlerini (ve bir return PC'sini) yığından alır ve ayrıca sonuçları yığına koyar. Kullanıcı tanımlı fonksiyonlar ve gömülü fonksiyonlar tam olarak aynı şekilde çağrılır.

Fonksiyonlar herhangi bir yerde tanımlanabilir ve tanımlandıkları blokta görülebilir olurlar. Bir fonksiyonun içinde, o fonksiyonun dışında tanımlanan yerel değişkenlere erişemezsiniz.

Fonksiyonlar, Solidity'ye benzer şekilde parametreleri atar ve değişkenleri döndürür. Bir değer döndürmek için, onu return değişken(ler)ine atarsınız.

Birden çok değer döndüren bir fonksiyonu çağırırsanız, bunları
`a, b := f(x)` veya `let a, b := f(x)` kullanarak birden çok değişkene atamanız gerekir.

`leave` ifadesi, geçerli fonksiyondan çıkmak için kullanılabilir. Diğer dillerdeki `return` ifadesi gibi çalışır, sadece döndürmek için bir değer almaz, sadece fonksiyonlardan çıkar ve fonksiyon, dönüş (return) değişkenlerine o anda atanmış olan değerleri döndürür.

EVM diyalektinin, yalnızca geçerli yul fonksiyonundan değil, tam çalıştırma bağlamından (dahili mesaj çağrısı) çıkan `return` adlı gömülü bir fonksiyonu olduğunu unutmayın.

Aşağıdaki örnek, power adlı fonksiyonun kare-ve-çarpma yöntemiyle bir uygulamasıdır.

```
{
    function power(base, exponent) -> result {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

3.32.5 Yul Tanımlaması

Bu bölüm Yul kodunu resmi olarak açıklar. Yul kodu genellikle kendi bölümlerinde açıklandığı üzere Yul nesnelerinin içine yerleştirilir.

```
Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':=' Expression )?
Assignment =
    IdentifierList ':=' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
Leave = 'leave'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9.]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )*
Literal =
    (NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?
NumberLiteral = HexNumber | DecimalNumber
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

Dilbilgisi ile İlgili Kısıtlamalar

Doğrudan dilbilgisi tarafından dayatılanların dışında, aşağıdaki kısıtlamalar geçerlidir:

Switc ifadelerinin en az bir case'i (durumu) olmalıdır (default case dahil). Tüm case değerlerinin aynı türe ve farklı değerlere sahip olması gerekir. İfade türünün tüm olası değerleri kapsam dahilindeyse, default bir case ifadesine izin verilmez (yani, hem doğru hem de yanlış bir duruma sahip bir `bool` ifadesine sahip bir switch, default bir case'e izin vermez).

Her ifade sıfır veya daha fazla değer olarak ele alınır. Tanımlayıcılar (identifier) ve Değişmez Değerler (literal) tam bir değer olarak ele alınır ve fonksiyon çağrıları, çağrılan fonksiyonun return değişkenlerinin sayısına eşit sayıda değer olarak ele alınır.

Değişken bildirimlerinde ve atamalarında, eğer varsa sağ taraftaki ifadenin, sol taraftaki değişkenlerin sayısına eşit sayıda değer alması gerekir. Bu, birden fazla değeri ele alan bir ifadeye izin verilen tek durumdur. Aynı değişken adı, bir atamanın veya değişken bildiriminin sol tarafında birden fazla olamaz.

Aynı zamanda komut olan ifadeler (yani blok seviyesinde) 0 değeri olarak değerlendirilmelidir.

Diğer tüm durumlarda, ifadeler tam olarak tek bir değere göre ele alınmalıdır.

`continue` veya `break` ifadesi yalnızca aşağıdaki gibi bir for-loop gövdesi içinde kullanılabilir. İfadeyi içeren en içteki loop döngüsünü düşünün. Döngü ve ifade aynı fonksiyonda olmalı veya her ikisi de en üst seviyede olmalıdır. İfade, loop döngüsünün gövde bloğunda olmalıdır; döngünün başlatma bloğunda veya güncelleme bloğunda olamaz. Bu kısıtlamanın yalnızca `continue` veya `break` deyimini içeren en içteki döngü için geçerli olduğunu vurgulamakta fayda var: bu en içteki döngü (loop) ve dolayısıyla `continue` veya `break` ifadesi, bir dış döngünün herhangi bir yerinde, muhtemelen bir dış döngünün başlatma bloğunda veya güncelleme bloğunda görünebilir. Örneğin, aşağıdakiler yasaldır, çünkü `break`, dış döngünün güncelleme bloğunda da meydana gelmesine rağmen, iç döngünün gövde bloğunda meydana gelir:

```
for {} true { for {} true {} { break } }
{
}
```

For döngüsünün koşul kısmı tam olarak bir değere göre değerlendirilmelidir.

`leave` ifadesi yalnızca bir fonksiyon içinde kullanılabilir.

Fonksiyonlar, döngü başlatma blokları söz konusu olduğunda herhangi bir yerde tanımlanamaz.

Değişmezler kendi türlerinden daha büyük olamaz. Tanımlanan en büyük tür 256 bit genişliğindedir.

Atamalar ve fonksiyon çağrıları sırasında ilgili değerlerin türlerinin eşleşmesi gerekir. Örtülü (implicit) tür dönüşümü yoktur. Genel olarak tür dönüştürme, yalnızca diyalekt bir türün değerini alan ve farklı bir türün değerini döndüren uygun bir gömülü fonksiyon sağladığında gerçekleştirilebilir.

Kapsam Belirleme Kuralları

Yul'daki kapsamlar (scope) Bloklara bağlıdır (fonksiyonlar ve aşağıda açıklandığı gibi for döngüsü hariç) ve tüm bildirimler (`FunctionDefinition`, `VariableDeclaration`) bu kapsamlara yeni tanımlayıcılar (identifier) getirir.

Tanımlayıcılar, tanımlandıkları blokta görünürler (tüm alt düğümler ve alt bloklar dahil): fonksiyonlar tüm blokta (hatta tanımlandıkları yerden önce bile) görünürken, değişkenler yalnızca `VariableDeclaration`'dan sonraki ifadeden başlayarak görünür.

Özellikle, değişkenlere kendi değişken atamalarının sağ tarafında referans verilemez. Fonksiyonlara, atamalarından önce referans verilebilir (eğer görünürlerse).

Genel kapsam belirleme kuralının bir istisnası olarak, for döngüsünün "init" bölümünün (ilk blok) kapsamı, for döngüsünün diğer tüm bölümlerini içine alır. Bu, init bölümünde bildirilen (ancak init parçasının içindeki bir bloğun içerisinde değil) değişkenlerin ve fonksiyonların for döngüsünün diğer tüm bölümlerinde görünür olduğu anlamına gelir.

For döngüsünün diğer bölümlerinde bildirilen tanımlayıcılar, normal sözdizimsel kapsam belirleme kurallarına uyar.

Bu demektir ki `for { I... } C { P... } { B... }` şeklindeki bir for döngüsü
`{ I... for {} C { P... } { B... } }` ifadesine eşittir.

Fonksiyonların parametreleri ve return parametreleri fonksiyon gövdesinde görünür ve isimleri farklı olmalıdır.

Fonksiyonların içinde, o fonksiyonun dışında bildirilen bir değişkene referans vermek mümkün değildir.

Gölgelemeye (shadowing) izin verilmez, yani aynı ada sahip başka bir tanımlayıcının da görünür olduğu bir noktada, geçerli işlevin dışında bildirildiği için ona başvurmak mümkün olmasa bile bir tanımlayıcı (identifier) atayamazsınız.

Resmi Şartname

AST'nin çeşitli düğümlerinde(node) aşırı yüklenmiş bir E değerlendirme fonksiyonu sağlayarak resmi olarak Yul'u tanımlarız. Gömülü fonksiyonların yan etkileri olabileceğinden, E iki durum nesnesini (state object) ve AST düğümünü alır ve iki yeni durum nesnesi ve değişken sayıda başka değer döndürür. Bu iki durum nesnesinden birisi global durum nesnesi (EVM bağlamında blok zincirinin belleği, depolanması ve durumudur) ve diğeri de yerel durum nesnesidir (yerel değişkenlerin durumu, yani EVM'deki yığının bir bölümü).

AST düğümü bir ifadeyse, E iki durum nesnesini ve `break`, `continue` ve `leave` komutları için kullanılan bir "mod"u döndürür. AST düğümü bir ifadeyse, E, iki durum nesnesini ve ifadenin değerlendirdiği sayıda değeri döndürür.

Bu üst düzey açıklama için global durumun (state) kesin hatları belirtilmemiştir. L yerel durumu , `i` tanımlayıcılarının `L[i] = v` olarak gösterilen `v` değerlerine eşlenmesidir.

Bir `v` tanımlayıcısı (identifier) için, tanımlayıcının adı `$v` olsun.

AST düğümleri (node) için bir destructuring notasyonu kullanacağız.

```
E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    L2, L1'in L tanımlayıcılarına bir kısıtlaması olsun
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        eğer mode regular ise
            E(G1, L1, St2, ..., Stn)
        değilse
            G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
    E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
    L1 in L nin kopyası olduğu durumda L1[$var_i] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    L2 nin L1 in kopyası olduğu durumda L2[$var_i] = vi for i = 1, ..., n
    G1, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L1, mode = E(G, L, i1, ..., in)
        // mode regular olmalı veya sözdizimsel kısıtlamalar nedeniyle terk edilmelidir
        eğer mode leave ise o zaman
            G1, L1 değişkenleri L, leave değişkenlerine kısıtlıdır
        değilse
            let G2, L2, mode = E(G1, L1, for {} condition post body)
            G2, L2 değişkenleri L, mode değişkenlerine kısıtlıdır
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            otherwise if mode is leave:
                G2, L2, leave
            else:
                G3, L3, mode = E(G2, L2, post)
                if mode is leave:
                    G3, L3, leave
                otherwise
                    E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
    G, L, continue
E(G, L, leave: Leave) =
    G, L, leave
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // Değişmezleri (literal) değerlendirin, bağlam önemli değil
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    vi = v olacak şekilde en küçük i varsa:
        E(G0, L0, sti)
    else:
        E(G0, L0, st')

E(G, L, <name>: Identifier) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
    ...
    G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
    Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
    Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
    be the function of name $fname visible at the point of the call.
    Let L' be a new local state such that
    L'[$parami] = vi and L'[$reti] = 0 for all i.
    Let G'', L'', mode = E(Gn, L', block)
    G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: StringLiteral) = G, L, str(l),
    burada str, EVM diyalekti için yukarıdaki 'Değişmezler' bölümünde
    tanımlanan string değerlendirme fonksiyonudur.
E(G, L, n: HexNumber) = G, L, hex(n)
    burada hex, bir onaltılık (hexadecimal) basamak dizisini soldan okumalı (big endian)
    değerine dönüştüren onaltılık değerlendirme fonksiyonudur.
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal evaluation function,
    which turns a sequence of decimal digits into their big endian value
    burada dec, ondalık (decimal) basamak dizisini soldan okumalı (büyük endian) değerine
    dönüştüren ondalık değerlendirme fonksiyonudur.
```

EVM Dialect

Yul'un varsayılan lehçesi şu anda EVM'nin mevcut sürümü için olan EVM lehçesidir.

EVM'nin bir sürümü ile birlikte. Bu lehçede kullanılabilen tek tür, Ethereum Sanal Makinesinin 256 bit yerel türü olan `u256`'dır. Bu tür, lehçenin varsayılan türü olduğu için görmezden gelinebilir.

Aşağıdaki tablo tüm gömülü fonksiyonları (EVM sürümüne bağlı olarak) listeler ve fonksiyonun / işlem kodunun semantiğinin kısa bir açıklamasını sunar. Bu belge, Ethereum sanal makinesinin tam bir açıklaması olmak istemediği için kesin semantikleriyle ilgileniyorsanız, lütfen farklı bir belgeye bakınız.

- ile işaretlenen işlem kodları bir sonuç döndürmez ve diğerleri tam olarak bir değer döndürür. `F`, `H`, `B`, `C`, `I` ve `L` ile işaretlenen işlem kodları sırasıyla Frontier, Homestead, Byzantium, Constantinople, Istanbul veya London'dan beri mevcuttur.

Aşağıda, `mem[a...b)`, `a` konumundan başlayan ancak `b` konumuna kadar olmayan bellek baytlarını belirtir ve `storage[p]`, `p` yuvasındaki depolama içeriğini belirtir.

Yul, yerel değişkenleri ve kontrol akışını yönettiğinden, bu özelliklere müdahale eden işlem kodları mevcut değildir. Bu, `dup` ve `swap` talimatlarının yanı sıra `jump` talimatlarını, etiketleri ve `push` talimatlarını içerir.

Komut			Açıklama
stop()	-	F	çalışmayı durdurur, return(0, 0) ile eşdeğerdir
add(x, y)		F	x + y
sub(x, y)		F	x - y
mul(x, y)		F	x * y
div(x, y)		F	x / y veya 0 eğer y == 0 ise
sdiv(x, y)		F	x / y, ikinin tümleyenindeki işaretli sayılar için, y == 0 ise 0
mod(x, y)		F	x % y, y == 0 ise 0
smod(x, y)		F	x % y, ikinin tümleyenindeki işaretli sayılar için, y == 0 ise 0
exp(x, y)		F	x in y ninci kuvveti
not(x)		F	x'in bit düzeyinde "değil"i (x'in her biti reddedilir)
lt(x, y)		F	x < y ise 1, değilse 0
gt(x, y)		F	x > y ise 1, 0 değilse 0
slt(x, y)		F	x < y ise 1, değilse 0, ikinin tümleyenindeki işaretli sayılar için
sgt(x, y)		F	x > y ise 1, değilse 0, ikinin tümleyenindeki işaretli sayılar için
eq(x, y)		F	x == y ise 1, değilse 0
iszero(x)		F	x == 0 ise 1, değilse 0
and(x, y)		F	x ve y için bit düzeyinde "and"
or(x, y)		F	x ve y için bit düzeyinde "or"
xor(x, y)		F	x ve y için bit düzeyinde "xor"
byte(n, x)		F	x'in n. baytı, burada en önemli bayt 0. bayttır
shl(x, y)		C	y ile x bit sola mantıksal kaydırma
shr(x, y)		C	y ile x bit sağa mantıksal kaydırma
sar(x, y)		C	işaretli aritmetik kaydırma sağa y ile x bit
addmod(x, y, m)		F	(x + y) % m keyfi kesinlikli aritmetik ile, m == 0 ise 0
mulmod(x, y, m)		F	(x * y) % m keyfi kesinlikli aritmetik ile, m == 0 ise 0
signextend(i, x)		F	işaret, en önemsizden başlayarak (i*8+7). bitten başlayarak genişler
keccak256(p, n)		F	keccak(mem[p…(p+n)))
pc()		F	koddaki geçerli konum
pop(x)	-	F	x değerini at
mload(p)		F	mem[p…(p+32))
mstore(p, v)	-	F	mem[p…(p+32)) := v
mstore8(p, v)	-	F	mem[p] := v & 0xff (yalnızca tek bir baytı değiştirir)

Komut			Açıklama
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
msize()		F	bellek boyutu, yani erişilen en büyük bellek indeksi
gas()		F	gaz hala uygulama için kullanılabilir
address()		F	mevcut sözleşmenin / uygulama bağlamının adresi
balance(a)		F	a adresindeki wei bakiyesi
selfbalance()		I	balance(address()) ile eşdeğer, ancak daha ucuz
caller()		F	sender'ı çağırır (`delegatecall`'u' hariç tutarak)
callvalue()		F	mevcut çağrı ile birlikte gönderilen wei
calldataload(p)		F	p konumundan başlayan çağrı verileri (32 bayt)
calldatasize()		F	bayt cinsinden çağrı verilerinin boyutu
calldatacopy(t, f, s)	-	F	f konumundaki çağrı verilerinden t konumundaki mem'e s bayt kopyalayın
codesize()		F	mevcut sözleşme / uygulama bağlamının kodunun boyutu
codecopy(t, f, s)	-	F	s baytını f konumundaki koddan t konumundaki mem'e kopyalayın
extcodesize(a)		F	a adresindeki kodun boyutu
extcodecopy(a, t, f, s)	-	F	codecopy(t, f, s) gibi ama a adresindeki kodu alır
returndatasize()		B	son returndata'nın boyutu
returndatacopy(t, f, s)	-	B	f konumundaki returndata'dan t konumundaki mem'e s bayt kopyalayın
extcodehash(a)		C	a adresinin hash kodu
create(v, p, n)		F	mem[p…(p+n)) koduyla yeni sözleşme oluştur ve v wei gönder ve yeni adresi retu
create2(v, p, n, s)		C	keccak256(0xff . this .s .keccak256(mem[p…(p+n))) adresinde mem[p…(p+n)) k
call(g, a, v, in, insize, out, outsize)		F	a adresindeki mem[in…(in+insize) girişli çağrı sözleşmesi, g gaz, v wei ve mem[
callcode(g, a, v, in, insize, out, outsize)		F	`call` ile aynıdır, ancak yalnızca a kodunu kullanın ve aksi takdirde mevcut sözleş
delegatecall(g, a, in, insize, out, outsize)		H	`callcode` ile eşdeğerdir ama aynı zamanda `caller` ve `callvalue` değerini de tu
staticcall(g, a, in, insize, out, outsize)		B	`call(g, a, 0, in, insize, out, outsize)` ile eşdeğerdir ama durum değ
return(p, s)	-	F	yürütmeyi sonlandırır, veriyi dönderir mem[p…(p+s))
revert(p, s)	-	B	yürütmeyi sonlandırır, durum değişikliklerini geri alır mem[p…(p+s) verisini dön
selfdestruct(a)	-	F	yürütmeyi sonlandırır, mevcut sözleşmeyi yok eder ve parayı a'ya gönderir
invalid()	-	F	geçersiz talimatla yürütmeyi sonlandır
log0(p, s)	-	F	topic ve mem[p…(p+s)) datası olmadan log aç
log1(p, s, t1)	-	F	t1 ve mem[p…(p+s)) datası ile log aç
log2(p, s, t1, t2)	-	F	t1, t2 topic'leri ve mem[p…(p+s)) datası ile log aç
log3(p, s, t1, t2, t3)	-	F	t1, t2, t3 topic'leri ve mem[p…(p+s)) datası ile log aç
log4(p, s, t1, t2, t3, t4)	-	F	topics t1, t2, t3, t4 topic'leri ve mem[p…(p+s)) datası ile log aç
chainid()		I	Yürütme zincirinin kimliği (EIP-1344)
basefee()		L	mevcut bloğun taban ücreti (EIP-3198 ve EIP-1559)
origin()		F	işlem gönderen
gasprice()		F	işlemin gaz fiyatı
blockhash(b)		F	b nolu bloğun hash değeri - mevcut hariç yalnızca son 256 blok için
coinbase()		F	mevcut madencilik faydalanıcısı
timestamp()		F	çağlardan bu yana geçerli bloğun saniye cinsindenzaman damgası
number()		F	mevcut blok numarası
difficulty()		F	mevcut bloğun zorluğu
gaslimit()		F	mevcut bloğun gaz limitini engelle

Not: `call*` komutları, return veya hata verilerinin yerleştirildiği bellekte bir alanı tanımlamak için `out` ve `outsize` parametrelerini kullanır. Bu alan, çağrılan sözleşmenin kaç bayt döndüğüne bağlı olarak yazılır. Daha fazla veri döndürürse, yalnızca ilk `outsize` baytlar yazılır. Geri kalan verilere `returndatacopy` işlem kodunu kullanarak erişebilirsiniz. Daha az veri döndürürse, kalan baytlara hiç dokunulmaz. Bu bellek alanının hangi bölümünün geri dönüş

verilerini içerdiğini kontrol etmek için `returndatasize` işlem kodunu kullanmanız gerekir. Kalan baytlar, çağrıdan önceki değerlerini koruyacaktır.

Bazı dahili diyalektlerde ek fonksiyonlar vardır:

datasize, dataoffset, datacopy

`datasize(x)`, `dataoffset(x)` ve `datacopy(t, f, l)` fonksiyonları bir Yul nesnesinin diğer bölümlerine erişmek için kullanılır.

`datasize` ve `dataoffset` argüman olarak yalnızca string değişmezlerini (diğer nesnelerin adlarını)
alabilir ve sırasıyla veri alanındaki boyutu ve ofseti döndürebilir. EVM için `datacopy` fonksiyonu `codecopy` fonksiyonu ile eşdeğerdir.

setimmutable, loadimmutable

`setimmutable(offset, "name", value)` ve `loadimmutable("name")` fonksiyonları, Solidity'deki değişmez mekanizma için kullanılır ve saf Yul ile hoş bir şekilde eşleşmez. `setimmutable(offset, "name", value)` çağrısı, verilen adlandırılmış değişmezi içeren sözleşmenin çalışma zamanı (runtime) kodunun ofsette `offset` belleğe kopyalandığını ve yer tutucuyu (placeholder) içeren bellekteki tüm konumlara (`offset`'e göre) `value` yazacağını varsayar. Bu, çalışma zamanı kodunda `loadimmutable("name")` çağrıları için oluşturulmuştur.

linkersymbol

`linkersymbol("library_id")` fonksiyonu, bağlayıcı (linker) tarafından değiştirilecek bir adres değişmezi (literal) için bir yer tutucudur (placeholder). İlk ve tek bağımsız değişkeni bir string değişmezi olmalıdır ve eklenecek adresi benzersiz şekilde temsil eder. Tanımlayıcılar (identifier) isteğe bağlı olabilir, ancak derleyici Solidity kaynaklarından Yul kodu ürettiğinde, o kitaplığı tanımlayan kaynak birimin adıyla nitelenmiş bir kitaplık adı kullanır. Kodu belirli bir kitaplık adresiyle ilişkilendirmek için, komut satırındaki `--libraries` seçeneğine aynı tanımlayıcı verilmelidir.

Örneğin aşağıdaki kod

```
let a := linkersymbol("file.sol:Math")
```

bağlayıcı (linker) `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890` seçeneği ile çağrıldığında şu koda eşittir:

```
let a := 0x1234567890123456789012345678901234567890
```

Solidity bağlayıcı (linker) hakkında ayrıntılar için *Komut Satırı Derleyicisini Kullanma* bölümüne bakın.

memoryguard

Bu fonksiyon, nesnelerle birlikte EVM lehçesinde mevcuttur. `let ptr := memoryguard(size)` (`size`'ın' değişmez bir sayı olması gerektiği yerde) çağıranı, yalnızca `[0, size)` aralığında veya `ptr`'dan başlayan sınırsız aralıkta bellek kullandıklarında garanti verir.

Bir `memoryguard` çağrısının varlığı, tüm bellek erişiminin bu kısıtlamaya bağlı olduğunu gösterdiğinden, optimize edicinin ek optimizasyon adımları gerçekleştirmesine izin verir, örneğin, aksi takdirde belleğe erişilemeyecek olan yığın (stack) değişkenlerini taşımaya çalışan yığın limiti kaçağı gibi.

Yul optimizer, amaçları için yalnızca bellek aralığını `[size, ptr)` kullanmayı vaat eder. Optimize edicinin herhangi bir bellek ayırması gerekmiyorsa, `ptr == size` boyutunu tutar.

`memoryguard` birden çok kez çağrılabilir, ancak bir Yul alt nesnesinde bağımsız değişkenle aynı değişmeze sahip olması gerekir. Bir alt nesnede en az bir `memoryguard` çağrısı bulunursa, bunun üzerinde ek optimize edici adımlar çalıştırılır.

verbatim

`verbatim...` gömülü fonksiyonlar kümesi, Yul derleyicisi tarafından bilinmeyen işlem kodları için bayt kodu oluşturmanıza olanak tanır. Ayrıca, optimize edici tarafından değiştirilmeyecek olan bayt kodu dizileri oluşturmanıza da olanak tanır.

Fonksiyonlar şu şekildedir: `verbatim_<n>i_<m>o("<data>", ...)`, burada

- `n` giriş (input) yığını yuvalarının/değişkenlerinin sayısını belirten 0 ile 99 arasında bir ondalık sayıdır
- çıktı (output) yığını yuvalarının / değişkenlerinin sayısını belirten 0 ile 99 arasında bir ondalık sayıdır
- `data` bayt dizisini içeren bir string değişmezidir

Örneğin, optimize edicinin sabit değer olan ikiye dokunmadan girişi iki ile çarpan bir fonksiyon tanımlamak istiyorsanız, şöyle kullanabilirsiniz:

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

Bu kod, `x`'i doğrudan almak amacıyla (yine de optimize edici, `calldataload` işlem kodunun sonucunu doğrudan yeniden kullanabilir) bir `dup1` işlem kodunun ardından `600202` ile sonuçlanır. Kodun, kopyalanan `x` değerini tükettiği ve sonucu yığının en üstünde ürettiği varsayılır. Derleyici daha sonra `double` için bir yığın yuvası tahsis etmek ve sonucu orada saklamak için kod üretir.

Tüm işlem kodlarında olduğu gibi, değişmez değerler en soldaki değişmez değer en üstte olacak şekilde yığın üzerinde düzenlenirken, return değerleri ise en sağdaki değişken, yığının en üstünde olacak şekilde düzenlendiği varsayılır.

`verbatim` isteğe bağlı işlem kodları ve hatta Solidity derleyicisi tarafından bilinmeyen işlem kodları oluşturmak için kullanılabildiğinden, optimize edici ile birlikte `verbatim` kullanılırken dikkatli olunmalıdır. Optimize edici kapatıldığında bile, kod oluşturucu yığın düzenini belirlemelidir, bu da örneğin yığın yüksekliğini değiştirmek için `verbatim` kullanmak istediğinizde tanımsız davranışa yol açabilir.

Aşağıda, derleyici tarafından kontrol edilmeyen verbatim bayt kodundaki kısıtlamaların kapsamlı olmayan bir listesi bulunmaktadır. Bu kısıtlamaların ihlali, tanımlanmamış davranışlara neden olabilir.

- Kontrol akışı verbatim bloklarının içine veya dışına atlamamalıdır, ancak aynı verbatim bloğu içinde atlayabilir.
- Giriş ve çıkış parametreleri dışındaki yığın içeriklerine erişilmemelidir.
- Yığın yükseklik farkı tam olarak `m - n` olmalıdır (çıkış yuvaları eksi giriş yuvaları).
- Verbatim bayt kodu, kapsayan bayt kodu hakkında herhangi bir varsayımda bulunamaz. Gerekli tüm parametreler yığın değişkenleri olarak iletilmelidir.

Optimize edici "verbatim" bayt kodunu analiz etmez ve her zaman durumun tüm yönlerini değiştirdiğini ve bu nedenle `verbatim` fonksiyon çağrılarında yalnızca çok az optimizasyon yapabileceğini varsayar.

Optimize edici, verbatim bayt kodunu opak bir kod bloğu olarak ele alır. Bölmez, ancak aynı verbatim bayt kodu bloklarıyla taşıyabilir, çoğaltabilir veya birleştirebilir. Bir "verbatim" bayt kodu bloğuna kontrol akışı tarafından ulaşılamıyorsa, kaldırılabilir.

Uyarı: EVM iyileştirmelerinin mevcut akıllı sözleşmeleri bozup bozmayacağı konusundaki tartışmalar sırasında, `verbatim` içindeki özellikler, Solidity derleyicisinin kullandığı özelliklerle aynı değerlendirmeyi alamaz.

Not: Karışıklığı önlemek için, "verbatim" string'i başlayan tüm tanımlayıcılar reserv edilmiştir ve kullanıcıların atadığı tanımlayıcılar için kullanılamaz.

3.32.6 Specification of Yul Object

Yul nesneleri, adlandırılmış kod ve veri bölümlerini gruplandırmak için kullanılır. `datasize`, `dataoffset` ve `datacopy` fonksiyonları bu bölümlere kod içinden erişmek için kullanılabilir. Onaltılı (hex) string'ler, onaltılı kodlamada verileri belirtmek için kullanılabilir, yerel (native) kodlamada normal string'ler kullanılır. Kod için `datacopy`, birleştirilmiş ikili (binary) gösterimine erişecektir.

```
Object = 'object' StringLiteral '{' Code ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral ( HexLiteral | StringLiteral )
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
```

Yukarıda `Block`, önceki bölümde Yul kodu dilbilgisinde açıklanan `Block` anlamına gelir.

Not: `_deployed` ile biten bir ada sahip bir nesne, Yul optimizer tarafından deploy edilmiş kod olarak değerlendirilir. Bunun tek sonucu, optimize edicide bulgusal olarak farklı bir gaz maliyeti yöntemidir.

Not: Adında `.` bulunan veri nesneleri veya alt nesneler tanımlanabilir, ancak bunlara `datasize`, `dataoffset` veya `datacopy` üzerinden erişim mümkün değildir, çünkü `.`, başka bir nesnenin içindeki nesnelere erişmek için ayırıcı olarak kullanılır.

Not: `".metadata"` adı verilen veri nesnesinin özel bir anlamı vardır: Koddan erişilemez ve nesnedeki konumundan bağımsız olarak her zaman bayt kodunun en sonuna eklenir.

Gelecekte özel öneme sahip diğer veri nesneleri eklenebilir, ancak adları her zaman bir `.` ile başlayacaktır.

Örnek bir Yul Nesnesi aşağıda gösterilmiştir:

```
// Bir sözleşme, dağıtılacak kodu veya oluşturabileceği diğer sözleşmeleri
// temsil eden alt nesnelere sahip tek bir nesneden oluşur.
// Tek "kod" düğümü, nesnenin yürütülebilir kodudur.
// Her (diğer) adlandırılmış nesne veya veri bölümü serileştirilir ve
//  özel gömülü fonksiyonlar olan datacopy / dataoffset / datasize için␣
↪erişilebilir hale getirilir.
// Geçerli nesnenin içindeki geçerli nesne, alt nesneler ve
// veri öğeleri kapsam içindedir.
object "Contract1" {
    // Bu, sözleşmenin yapıcı (constructor) kodudur.
    code {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // ilk olarak  "Contract2" oluştur
        let size := datasize("Contract2")
        let offset := allocate(size)
        // Bu, EVM için kod kopyasına dönüşecek
        datacopy(offset, dataoffset("Contract2"), size)
        // yapıcı parametresi tek bir sayıdır 0x1234
        mstore(add(offset, size), 0x1234)
        pop(create(offset, add(size, 32), 0))

        // şimdi çalışma zamanı nesnesini döndür
        // (şu anda yürütülmekte olan kod, yapıcı kodudur)
        size := datasize("Contract1_deployed")
        offset := allocate(size)
        // Bu, Ewasm için bir memory->memory kopyasına ve
        // EVM için bir kod kopyasına dönüşecektir.
        datacopy(offset, dataoffset("Contract1_deployed"), size)
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "Contract1_deployed" {
        code {
            function allocate(size) -> ptr {
                ptr := mload(0x40)
                if iszero(ptr) { ptr := 0x60 }
                mstore(0x40, add(ptr, size))
            }

            // runtime code

            mstore(0, "Hello, World!")
            return(0, 0x20)
        }
    }

    // Gömülü nesne. Kullanım durumu, dışarının bir fabrika sözleşmesi olması ve
    // Sözleşme2'nin fabrika tarafından oluşturulacak kod olmasıdır.
    object "Contract2" {
        code {
            // kod buraya ...
        }

        object "Contract2_deployed" {
            code {
                // kod buraya ...
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            }
        }

        data "Table1" hex"4123"
    }
}
```

3.32.7 Yul Optimizer

Yul optimize edicisi Yul kodunda çalışır ve giriş, çıkış ve ara durumlar için aynı dili kullanır. Bu, optimize edicinin kolay hata ayıklamasını ve doğrulanmasını sağlar.

Farklı optimizasyon aşamaları ve optimize edicinin nasıl kullanılacağı hakkında daha fazla ayrıntı için lütfen *optimize edici dökümantasyonu* bölümüne bakın.

Solidity'yi bağımsız Yul modunda kullanmak istiyorsanız, optimize ediciyi `--optimize` kullanarak etkinleştirirsiniz ve isteğe bağlı olarak `--optimize-runs` ile *beklenen sözleşme yürütme sayısı* belirtirsiniz:

```
solc --strict-assembly --optimize --optimize-runs 200
```

Solidity modunda Yul optimizer, normal optimizer ile birlikte etkinleştirilir.

Optimizasyon Adım Sırası

Varsayılan olarak Yul optimizer(optimize edici), önceden tanımlanmış optimizasyon adımları dizisini oluşturulan assembly'ye uygular. Bu sırayı geçersiz kılabilir ve `--yul-optimizations` seçeneğini kullanarak kendinizinkini uygulatabilirsiniz:

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul'
```

Adımların sırası önemlidir ve çıktının kalitesini etkiler. Ayrıca, bir adımı uygulamak, daha önce uygulanmış olan diğerleri için yeni optimizasyon fırsatlarını ortaya çıkarabilir, bu nedenle adımları tekrarlamak genellikle faydalıdır. Dizinin bir kısmını köşeli parantezler (`[]`) içine alarak, optimize ediciye, sonuçta ortaya çıkan assemby'nin boyutunu artık iyileştirmeyene kadar o kısmı tekrar tekrar uygulamasını söylersiniz. Köşeli ayraçları tek bir sırada birden çok kez kullanabilirsiniz ancak iç içe geçemezler.

Aşağıdaki optimizasyon adımları uygulanabilir:

Kısaltma	Tam adı
f	BlockFlattener
l	CircularReferencesPruner
c	CommonSubexpressionEliminator
C	ConditionalSimplifier
U	ConditionalUnsimplifier
n	ControlFlowSimplifier
D	DeadCodeEliminator
v	EquivalentFunctionCombiner
e	ExpressionInliner
j	ExpressionJoiner
s	ExpressionSimplifier
x	ExpressionSplitter

sonraki sayfaya devam

Tablo 2 – önceki sayfadan devam

Kısaltma	Tam adı
I	ForLoopConditionIntoBody
O	ForLoopConditionOutOfBody
o	ForLoopInitRewriter
i	FullInliner
g	FunctionGrouper
h	FunctionHoister
F	FunctionSpecializer
T	LiteralRematerialiser
L	LoadResolver
M	LoopInvariantCodeMotion
r	RedundantAssignEliminator
R	ReasoningBasedSimplifier - son derece deneysel
m	Rematerialiser
V	SSAReverser
a	SSATransform
t	StructuralSimplifier
u	UnusedPruner
p	UnusedFunctionParameterPruner
d	VarDeclInitializer

Bazı adımlar `BlockFlattener`, `FunctionGrouper`, `ForLoopInitRewriter` tarafından sağlanan özelliklere bağlıdır. Bu nedenle Yul optimize edicisi, kullanıcı tarafından sağlanan herhangi bir adımı uygulamadan önce bunları uygular.

ReasoningBasedSimplifier, şu anda varsayılan adımlar kümesinde etkinleştirilmeyen bir optimize edici adımıdır. Aritmetik ifadeleri ve boole koşullarını basitleştirmek için bir SMT çözücüsü kullanır. Henüz kapsamlı bir test veya doğrulama almamıştır ve tekrarlanamayan sonuçlar üretebilir, bu yüzden lütfen dikkatli kullanın!

3.32.8 Tamamlanmış ERC20 Örneği

```
object "Token" {
    code {
        // Oluşturucuyu sıfır yuvasında saklayın.
        sstore(0, caller())

        // Contratı deploy edin
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            // Ether göndermeye karşı koruma
            require(iszero(callvalue()))

            // Transfer edici
            switch selector()
            case 0x70a08231 /* "balanceOf(address)" */ {
                returnUint(balanceOf(decodeAsAddress(0)))
            }
            case 0x18160ddd /* "totalSupply()" */ {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            returnUint(totalSupply())
        }
        case 0xa9059cbb /* "transfer(address,uint256)" */ {
            transfer(decodeAsAddress(0), decodeAsUint(1))
            returnTrue()
        }
        case 0x23b872dd /* "transferFrom(address,address,uint256)" */ {
            transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))
            returnTrue()
        }
        case 0x095ea7b3 /* "approve(address,uint256)" */ {
            approve(decodeAsAddress(0), decodeAsUint(1))
            returnTrue()
        }
        case 0xdd62ed3e /* "allowance(address,address)" */ {
            returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))
        }
        case 0x40c10f19 /* "mint(address,uint256)" */ {
            mint(decodeAsAddress(0), decodeAsUint(1))
            returnTrue()
        }
        default {
            revert(0, 0)
        }

        function mint(account, amount) {
            require(calledByOwner())

            mintTokens(amount)
            addToBalance(account, amount)
            emitTransfer(0, account, amount)
        }
        function transfer(to, amount) {
            executeTransfer(caller(), to, amount)
        }
        function approve(spender, amount) {
            revertIfZeroAddress(spender)
            setAllowance(caller(), spender, amount)
            emitApproval(caller(), spender, amount)
        }
        function transferFrom(from, to, amount) {
            decreaseAllowanceBy(from, caller(), amount)
            executeTransfer(from, to, amount)
        }

        function executeTransfer(from, to, amount) {
            revertIfZeroAddress(to)
            deductFromBalance(from, amount)
            addToBalance(to, amount)
            emitTransfer(from, to, amount)
        }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            /* ---------- calldata decoding fonksiyonları ----------- */
            function selector() -> s {
                s := div(calldataload(0),␣
↪0x100000000000000000000000000000000000000000000000000000000)
            }

            function decodeAsAddress(offset) -> v {
                v := decodeAsUint(offset)
                if iszero(iszero(and(v,␣
↪not(0xffffffffffffffffffffffffffffffffffffffff)))) {
                    revert(0, 0)
                }
            }
            function decodeAsUint(offset) -> v {
                let pos := add(4, mul(offset, 0x20))
                if lt(calldatasize(), add(pos, 0x20)) {
                    revert(0, 0)
                }
                v := calldataload(pos)
            }
            /* ---------- calldata encoding fonksiyonları ---------- */
            function returnUint(v) {
                mstore(0, v)
                return(0, 0x20)
            }
            function returnTrue() {
                returnUint(1)
            }

            /* -------- olaylar (events) ---------- */
            function emitTransfer(from, to, amount) {
                let signatureHash :=␣
↪0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
                emitEvent(signatureHash, from, to, amount)
            }
            function emitApproval(from, spender, amount) {
                let signatureHash :=␣
↪0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
                emitEvent(signatureHash, from, spender, amount)
            }
            function emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {
                mstore(0, nonIndexed)
                log3(0, 0x20, signatureHash, indexed1, indexed2)
            }

            /* -------- depolama düzeni ---------- */
            function ownerPos() -> p { p := 0 }
            function totalSupplyPos() -> p { p := 1 }
            function accountToStorageOffset(account) -> offset {
                offset := add(0x1000, account)
            }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
            function allowanceStorageOffset(account, spender) -> offset {
                offset := accountToStorageOffset(account)
                mstore(0, offset)
                mstore(0x20, spender)
                offset := keccak256(0, 0x40)
            }

            /* -------- depolama erişimi ---------- */
            function owner() -> o {
                o := sload(ownerPos())
            }
            function totalSupply() -> supply {
                supply := sload(totalSupplyPos())
            }
            function mintTokens(amount) {
                sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))
            }
            function balanceOf(account) -> bal {
                bal := sload(accountToStorageOffset(account))
            }
            function addToBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                sstore(offset, safeAdd(sload(offset), amount))
            }
            function deductFromBalance(account, amount) {
                let offset := accountToStorageOffset(account)
                let bal := sload(offset)
                require(lte(amount, bal))
                sstore(offset, sub(bal, amount))
            }
            function allowance(account, spender) -> amount {
                amount := sload(allowanceStorageOffset(account, spender))
            }
            function setAllowance(account, spender, amount) {
                sstore(allowanceStorageOffset(account, spender), amount)
            }
            function decreaseAllowanceBy(account, spender, amount) {
                let offset := allowanceStorageOffset(account, spender)
                let currentAllowance := sload(offset)
                require(lte(amount, currentAllowance))
                sstore(offset, sub(currentAllowance, amount))
            }

            /* ---------- faydalı fonksiyonlar ---------- */
            function lte(a, b) -> r {
                r := iszero(gt(a, b))
            }
            function gte(a, b) -> r {
                r := iszero(lt(a, b))
            }
            function safeAdd(a, b) -> r {
                r := add(a, b)
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
                if or(lt(r, a), lt(r, b)) { revert(0, 0) }
            }
            function calledByOwner() -> cbo {
                cbo := eq(owner(), caller())
            }
            function revertIfZeroAddress(addr) {
                require(addr)
            }
            function require(condition) {
                if iszero(condition) { revert(0, 0) }
            }
        }
    }
}
```

3.33 Stil Klavuzu

3.33.1 Giriş

Bu kılavuz, Solidity kodu yazmak için kodlama kuralları sağlamayı amaçlamaktadır. Bu kılavuz, yararlı kurallar bulundukça ve eski kurallar kullanılmaz hale geldikçe zaman içinde değişen bir belge olarak düşünülmelidir.

Birçok proje kendi stil kılavuzlarını uygulayabilir. Uyuşmazlık durumunda, projeye özgü stil kılavuzları önceliklidir.

Bu stil kılavuzunun yapısı ve içindeki önerilerin çoğu python'un pep8 stil kılavuzundan alınmıştır.

Bu kılavuzun amacı *Solidity kodu yazmanın doğru yolu ya da en iyi yolunu göstermek değildir*. Bu kılavuzun amacı *tutarlılıktır*. Python'un pep8 adlı kitabından bir alıntı bu kavramı iyi özetlemektedir.

Not: Stil rehberi tutarlılıkla ilgilidir. Bu stil rehberi ile tutarlılık önemlidir. Bir proje içindeki tutarlılık daha önemlidir. Bir modül veya fonksiyon içindeki tutarlılık ise en mühim olanıdır.

Ama en önemlisi: **Ne zaman tutarsız olmanız gerektiğini bilmenizdir** — bazen bu stil kılavuzu geçerli olmayabilir. Şüpheye düştüğünüzde, en iyi kararınızı verip yolunuza devam edin. Diğer örneklere bakın ve neyin en iyi göründüğüne karar verin. Ayrıca soru sormaktan çekinmeyin!

3.33.2 Kod Düzeni

Girintiler

Her girinti seviyesi için 4 boşluk kullanın.

Sekmeler(Tab) veya Boşluklar

Boşluklar en çok tercih edilen girinti oluşturma yöntemidir.

Tablar ve boşlukları karıştırmaktan kaçınmalısınız.

Boş Satırlar

Solidity kaynağındaki üst düzey bildirimleri iki boş satırla çevreleyin.

Yapın:

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

contract A {
    // ...
}


contract B {
    // ...
}


contract C {
    // ...
}
```

Yapmayın:

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

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Bir sözleşme içinde fonksiyon tanımlarının etrafını tek bir boş satırla çevreleyin.

Birbiriyle ilişkili tek satırlık gruplar arasında boş satırlar atlanabilir (abstract sözleşme için stub fonksiyonları gibi)

Yapın:

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

abstract contract A {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    function spam() public virtual pure;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}
```

Yapmayın:

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

abstract contract A {
    function spam() virtual pure public;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}
```

Maksimum Satır Uzunluğu

Tavsiye edilen maksimum satır uzunluğu 120 karakterdir.

Sarılmış(Wrapped) satırlar aşağıdaki yönergelere uygun olmalıdır.

1. İlk argüman açılış parantezine eklenmemelidir.
2. Bir ve yalnızca bir girinti kullanılmalıdır.
3. Her argüman kendi satırında yer almalıdır.
4. Sonlandırıcı öğe, `);`, tek başına son satıra yerleştirilmelidir.

Fonksiyon Çağrıları

Yapın:

```
thisFunctionCallIsReallyLong(
    longArgument1,
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    longArgument2,
    longArgument3
);
```

Yapmayın:

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Atama İfadeleri

Yapın:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

Yapmayın:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(argument1,
                                                                 argument2,
                                                                 argument3,
                                                                 argument4);
```

Event Tanımları ve Event Emitterları

Yapın:

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

Yapmayın:

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Kaynak Dosya Encoding

UTF-8 yada ASCII encoding tercih edilir.

Imports (İçe Aktarmalar)

İçe aktarma ifadeleri her zaman dosyanın en üstüne yerleştirilmelidir.

Yapın:

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

import "./Owned.sol";

contract A {
    // ...
}


contract B is Owned {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

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

Yapmayın:

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

contract A {
    // ...
}


import "./Owned.sol";


contract B is Owned {
    // ...
}
```

Fonksiyonların Sıralaması

Sıralandırma, okuyucuların hangi fonksiyonları çağırabileceklerini belirlemelerine ve constructor ve fallback tanımlamalarını daha kolay bulmalarına yardımcı olur.

Fonksiyonlar görünürlük durumlarına göre gruplandırılmalı ve sıralanmalıdır:

- constructor
- receive fonksiyon (eğer mevcutsa)
- fallback fonksiyon (eğer mevcutsa)
- external
- public
- internal
- private

Bir gruplandırma yaparken, `view` ve `pure` fonksiyonlarını en sona yerleştirin.

Yapın:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {
    constructor() {
        // ...
    }

    receive() external payable {
        // ...
    }
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    fallback() external {
        // ...
    }

    // External functions
    // ...

    // External functions that are view
    // ...

    // External functions that are pure
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

Yapmayın:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {

    // External functions
    // ...

    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // Private functions
    // ...

    // Public functions
    // ...

    constructor() {
        // ...
    }

    // Internal functions
    // ...
}
```

İfadelerde Boşluk Bırakma

Aşağıdaki durumlarda gereksiz boşluk bırakmaktan kaçının:

Tek satırlık fonksiyon tanımlamaları hariç olmak üzere, parantez, köşeli parantez veya ayraçların hemen içinde.

Yapın:

```
spam(ham[1], Coin({name: "ham"}));
```

Yapmayın:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

İstisna:

```
function singleLine() public { spam(); }
```

Virgülden, noktalı virgülden hemen önce:

Yapın:

```
function spam(uint i, Coin coin) public;
```

Yapmayın:

```
function spam(uint i , Coin coin) public ;
```

Bir atama veya başka bir operatörün etrafında, diğeriyle hizalamak için birden fazla boşluk:

Yapın:

```
x = 1;
y = 2;
longVariable = 3;
```

Yapmayın:

```
x             = 1;
y             = 2;
longVariable = 3;
```

receive ve fallback fonksiyonlarına boşluk eklemeyin:

Yapın:

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

Yapmayın:

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

Kontrol Yapıları (Control Structures)

Bir sözleşmenin, kütüphanenin, fonksiyonların ve struct'ların gövdelerini belirten parantezler:

- Bildirim (Declaration) ile aynı satırda açılmalıdır
- Bildirimin başlangıcıyla aynı girinti seviyesinde kendi satırlarında kapanmalıdır.
- Açılış parantezinden önce tek bir boşluk bırakılmalıdır.

Yapın:

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

contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

Yapmayın:

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

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

Aynı öneriler `if`, `else`, `while` ve `for` kontrol yapıları için de geçerlidir.

Ayrıca, `if`, `while` ve `for` kontrol yapıları ile koşulu temsil eden parantez bloğu arasında tek bir boşluk ve koşullu parantez bloğu ile açılış parantezi arasında tek bir boşluk olmalıdır.

Yapın:

```
if (...) {
    ...
}

for (...) {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    ...
}
```

Yapmayın:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

Gövdesi tek bir ifade içeren kontrol yapıları için, parantezleri atlamak *eğer* ifade tek bir satırda yer alıyorsa uygundur.

Yapın:

```
if (x < 10)
    x += 1;
```

Yapmayın:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

Bir `else` veya `else if` ibaresi içeren `if` blokları için, `else` ibaresi `if` ibaresinin kapanış paranteziyle aynı satıra yerleştirilmelidir. Bu, diğer blok benzeri yapıların kurallarına kıyasla bir istisnadır.

Yapın:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

Yapmayın:

```
if (x < 3) {
    x += 1;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
}
else {
    x -= 1;
}
```

Fonksiyon Tanımlamaları

Kısa fonksiyon bildirimleri için, fonksiyon gövdesinin açılış ayracının fonksiyon bildirimiyle aynı satırda tutulması önerilir.

Kapanış parantezi fonksiyon bildirimi ile aynı girinti seviyesinde olmalıdır.

Açılış ayracından önce tek bir boşluk bırakılmalıdır.

Yapın:

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}
```

Yapmayın:

```
function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
    }

function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

Bir fonksiyon için modifier sırası şöyle olmalıdır:

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

Yapın:

```
function balance(uint from) public view override returns (uint)  {
    return balanceOf[from];
}

function shutdown() public onlyOwner {
    selfdestruct(owner);
}
```

Yapmayın:

```
function balance(uint from) public override view returns (uint)  {
    return balanceOf[from];
}

function shutdown() onlyOwner public {
    selfdestruct(owner);
}
```

Uzun fonksiyon bildirimleri için, her argümanın fonksiyon gövdesiyle aynı girinti seviyesinde kendi satırına bırakılması önerilir. Kapanış parantezi ve açılış parantezi de fonksiyon bildirimi ile aynı girinti seviyesinde kendi satırlarına yerleştirilmelidir.

Yapın:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
{
    doSomething();
}
```

Yapmayın:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

Uzun bir fonksiyon bildiriminde modifier'lar varsa, her modifier kendi satırına bırakılmalıdır.

Yapın:

```
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();
}
```

Yapmayın:

```
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
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    onlyOwner
    priced
    returns (address) {
    doSomething();
}
```

Çok satırlı çıktı parametreleri ve return ifadeleri, *Maximum Line Length* bölümünde bulunan uzun satırları çevrelemek için önerilen aynı stili izlemelidir.

Yapın:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

Yapmayın:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
             uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

Tabanları argüman gerektiren inherited sözleşmelerdeki constructor fonksiyonları için, fonksiyon bildirimi uzunsa veya okunması zorsa, temel constructor'ların modifier'larla aynı şekilde yeni satırlara bırakılması önerilir.

Yapın:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
        x = param5;
    }
}
```

Yapmayın:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
contract D {
    constructor(uint) {
    }
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4) {
        x = param5;
    }
}


contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
            x = param5;
        }
}
```

Kısa fonksiyonları tek bir ifadeyle bildirirken, bunu tek bir satırda yapmaya izin verilir.

İzin verilebilir:

```
function shortFunction() public { doSomething(); }
```

Fonksiyon bildirimleri için bu kılavuzun amacı okunabilirliği artırmaktır. Bu kılavuz, fonksiyon bildirimleri için olası tüm olasılıkları kapsamaya çalışmadığından, yazarlar en iyi kararlarını vermelidir.

Mappingler

Değişken bildirimlerinde, `mapping` anahtar sözcüğünü türünden bir boşlukla ayırmayın. İç içe geçmiş `mapping` anahtar sözcüğünü türünden boşluk ile ayırmayın.

Yapın:

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

Yapmayın:

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

Değişken Bildirimleri

Dizi değişkenlerinin bildirimlerinde tür ile parantezler arasında boşluk olmamalıdır.

Yapın:

```
uint[] x;
```

Yapmayın:

```
uint [] x;
```

Diğer Öneriler

- Stringler tek tırnak yerine çift tırnak ile alıntılanmalıdır.

Yapın:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

Yapmayın:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- Operatörleri her iki tarafta tek bir boşlukla çevrelendirmelisiniz.

Yapın:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

Yapmayın:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Diğerlerinden daha yüksek önceliğe sahip operatörler, önceliği belirtmek için çevreleyen beyaz boşluğu kaldırabilir. Bunun amacı, karmaşık ifadeler için daha iyi okunabilirlik sağlamaktır. Bir operatörün her iki tarafında da her zaman aynı miktarda boşluk kullanmalısınız:

Yapın:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

Yapmayın:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

3.33.3 Yerleşim Sırası

Sözleşme unsurlarını aşağıdaki sıraya göre düzenleyin:

1. Pragma ifadeleri
2. Import ifadeleri
3. Interface'ler
4. Library'ler
5. Sözleşmeler

Her bir sözleşme, kütüphane veya arayüzün içinde aşağıdaki sıralamayı kullanın:

1. Type bildirimleri
2. Durum değişkenleri
3. Event'ler
4. Modifier'lar
5. Fonksiyonlar

Not: Türleri, event'lerde veya durum değişkenlerinde kullanımlarına yakın bir yerde bildirmek daha anlaşılır olabilir.

3.33.4 Adlandırma Kuralları

Adlandırma kuralları benimsendiğinde ve geniş çapta kullanıldığında güçlüdür. Farklı konvansiyonların kullanımı, aksi takdirde hemen elde edilemeyecek önemli *meta* bilgileri aktarabilir.

Burada verilen adlandırma önerileri okunabilirliği artırmayı amaçlamaktadır ve bu nedenle kural değil, daha ziyade nesnelerin adları aracılığıyla en fazla bilgiyi iletmeye yardımcı olacak kılavuzlardır.

Son olarak, bir kod tabanı içindeki tutarlılık her zaman bu belgede özetlenen kuralların yerine geçmelidir.

Adlandırma Stili

Karışıklığı önlemek için, farklı adlandırma stillerine atıfta bulunmak üzere aşağıdaki adlar kullanılacaktır.

- `b` (tek küçük harf)
- `B` (tek büyük harf)
- `lowercase`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (veya CapWords)
- `mixedCase` (ilk küçük harf karakteri ile CapitalizedWords`den farklıdır!)

Not: CapWords'te baş harfleri kullanırken, baş harflerin tüm harflerini büyük yazın. Bu nedenle HTTPServerError, HttpServerError adlandırmasından daha iyidir. MixedCase'de baş harfleri kullanırken, baş harflerin tüm harflerini büyük yazın, ancak ismin başındaysa ilk harfi küçük tutun. Bu nedenle xmlHTTPRequest, XMLHTTPRequest adlandırmasından daha iyidir.

Uzak Durulması Gereken İsimler

- `l` - Küçük harf le
- `O` - Büyük harf o
- `I` - Büyük harf I

Bunlardan hiçbirini tek harfli değişken adları için kullanmayın. Bunlar genellikle bir ve sıfır rakamlarından ayırt edilemez.

Sözleşme ve Kütüphane Adları

- Sözleşmeler ve kütüphaneler CapWords stili kullanılarak adlandırılmalıdır. Örnekler: `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
- Sözleşme ve kütüphane adları da dosya adlarıyla eşleşmelidir.
- Bir sözleşme dosyası birden fazla sözleşme ve/veya kütüphane içeriyorsa, dosya adı *çekirdek sözleşme* ile eşleşmelidir. Ancak kaçınılması mümkünse bu önerilmez.

Aşağıdaki örnekte gösterildiği gibi, sözleşme adı `Congress` ve kütüphane adı `Owned` ise, ilişkili dosya adları `Congress.sol` ve `Owned.sol` olmalıdır.

Yapın:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Owned.sol
contract Owned {
    address public owner;

    constructor() {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

ve `Congress.sol` içinde:

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

import "./Owned.sol";


contract Congress is Owned, TokenRecipient {
    //...
}
```

Yapmayın:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// owned.sol
contract owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

ve `Congress.sol` içinde:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
import "./owned.sol";


contract Congress is owned, tokenRecipient {
    //...
}
```

Struct Adları

Struct'lar CapWords stili kullanılarak adlandırılmalıdır. Örnekler: `MyCoin`, `Position`, `PositionXY`.

Event Adları

Event'ler CapWords stili kullanılarak adlandırılmalıdır. Örnekler: `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

Fonksiyon Adları

Fonksiyonlar mixedCase kullanmalıdır. Örnekler: `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Fonksiyon Argüman Adları

Fonksiyon argümanları mixedCase kullanmalıdır. Örnekler: `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

Özel bir struct üzerinde çalışan kütüphane fonksiyonları yazarken, struct ilk argüman olmalı ve her zaman `self` olarak adlandırılmalıdır.

Yerel ve Durum Değişkeni Adları

MixedCase kullanın. Örnekler: `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constant'lar (Sabitler)

Constantlar, sözcükleri ayıran alt çizgiler ile tüm büyük harflerle adlandırılmalıdır. Örnekler: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Modifier Adları

MixedCase kullanın. Örnekler: `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enumlar

Enumlar, basit tip bildirimleri tarzında, CapWords stili kullanılarak adlandırılmalıdır. Örnekler: `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Adlandırma Çakışmalarını Önleme

- `singleTrailingUnderscore_`

Bu kural, istenen adın halihazırda varolan bir durum değişkeni, fonkisyon, ayrılmış veya yerleşik bir adla çakışması durumunda önerilir.

3.33.5 NatSpec

Solidity sözleşmeleri NatSpec yorumları da içerebilir. Bunlar üçlü eğik çizgi (`///`) veya çift yıldız bloğu (`/** ... */`) ile yazılır ve doğrudan fonksiyon bildirimlerinin veya ifadelerin üzerinde kullanılmalıdır.

Örneğin, *a simple smart contract* sözleşmesi yorumlar eklendiğinde aşağıdaki gibi görünür:

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

/// @author The Solidity Team
/// @title A simple storage example
contract SimpleStorage {
    uint storedData;

    /// Store `x`.
    /// @param x the new value to store
    /// @dev stores the number in the state variable `storedData`
    function set(uint x) public {
        storedData = x;
    }

    /// Return the stored value.
    /// @dev retrieves the value of the state variable `storedData`
    /// @return the stored value
    function get() public view returns (uint) {
        return storedData;
    }
}
```

Solidity sözleşmelerinin tüm genel arayüzler (ABI'deki her şey) için *NatSpec* kullanılarak tam olarak açıklanması önerilir.

Ayrıntılı açıklama için lütfen *NatSpec* ile ilgili bölüme bakın.

3.34 Sık Kullanılan Modeller

3.34.1 Sözleşmelerden Para Çekme

Bir etkiden sonra önerilen fon gönderme yöntemi, para çekme modelini kullanmaktır. Bir etki sonucunda, anlaşılması en kolay Ether gönderme yöntemi doğrudan `transfer` çağrısı olsa da, potansiyel güvenlik riski oluşturduğundan bu önerilmez. Bu konuda daha fazla bilgiye *Güvenlikle İlgili Değerlendirmeler* sayfasından ulaşabilirsiniz.

King of the Ether <https://www.kingoftheether.com/>'de olduğu gibi, amacın "en zengin" olmak için sözleşmeye en fazla parayı göndermek olduğu bir sözleşmede para çekme modelinin nasıl kullanıldığına dair uygulamalı bir örnek aşağıda verilmiştir.

Aşağıdaki sözleşmede, artık en zengin olan değilseniz o anda en zengin olan kişinin fonlarını alırsınız.

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

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    /// Gönderilen Ether miktarı şu anki en yüksek
    /// miktardan yüksek değildi.
    error NotEnoughEther();

    constructor() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Tekrar girme(re-entrancy), saldırılarını önlemek için gönderim
        // öncesinde geri ödemeyi sıfırlamayı unutmayın
        pendingWithdrawals[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

Akla daha yatkın olan gönderme modeli aşağıdaki gibidir ama güvenlik açığı içerir:

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

contract SendContract {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    address payable public richest;
    uint public mostSent;

    /// Gönderilen Ether miktarı şu anki en yüksek
    /// miktardan yüksek değildi.
    error NotEnoughEther();

    constructor() payable {
        richest = payable(msg.sender);
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        // Bu satır sorunlara neden olabilir (aşağıda açıklanmıştır).
        richest.transfer(msg.value);
        richest = payable(msg.sender);
        mostSent = msg.value;
    }
}
```

Bu örnekte, bir saldırgan, `richest`'ın başarısız olan bir receive veya callback fonksiyonuna sahip bir sözleşmenin adresi olmasına sebep olarak (örneğin, `revert()` kullanarak veya yalnızca, onlara aktarılan 2300 gas ücretinden daha fazlasını tüketerek) sözleşmeyi kullanılamayacak bir duruma düşürebilir. Bu şekilde, fonları "zehirlenmiş" sözleşmeye iletmek için `transfer` her çağrıldığında başarısız olur, dolayısıyla `becomeRichest` fonksiyonu da başarısız olur ve sözleşme sonsuza kadar kilitli / takılı kalır.

Bunun aksine, ilk örnekten "çekme" modelini kullanırsanız saldırgan sözleşmenin kalanındaki işleyişin değil, yalnızca kendi çekim işleminin başarısız olmasına sebep olabilir.

3.34.2 Erişimi Kısıtlamak

Erişimi kısıtlamak sözleşmeler için yaygın bir modeldir. Herhangi bir insanı veya bilgisayarı, işlemlerinizin içeriğini veya sözleşmenizin durumunu okumak konusunda kesinlikle kısıtlayamayacağınızı unutmayın. Şifreleme kullanarak bunu bir miktar zorlaştırabilirsiniz ancak sözleşmenizin veri okumasına izin verilmişse diğer herkes de okuyacaktır.

Sözleşme durum degişkenlerinin okuma erişimini **diğer sözleşmeler** ile kısıtlayabilirsiniz. Bu aslında, durum değişkenlerinizi `public` olarak bildirmediğiniz sürece varsayılandır.

Ayrıca, sözleşmenizin durumunda değişiklik yapabilecek kişileri kısıtlayabilir veya sözleşmenizin fonksiyonlarını çağırabilirsiniz; bu bölümün konusu da budur.

Fonksiyon modifier'larının kullanımı bu kısıtlamaları oldukça okunur hale getirir.

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

contract AccessRestriction {
    // Bunlar, `msg.sender`'ın bu sözleşmeyi
    // oluşturan hesap olduğu yapım aşamasında
    // atanacaktır.
    address public owner = msg.sender;
    uint public creationTime = block.timestamp;
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    // Altta, bu sözleşmenin oluşturabileceği
    // hataların bir listesi, özel yorumlarda
    // yazılı bir açıklamayla birlikte
    // verilmiştir.

    /// Gönderici bu işlem için yetkili
    /// değildir.
    error Unauthorized();

    /// Fonksiyon çok erken çağrıldı.
    error TooEarly();

    /// Fonksiyon çağrısıyla yeterince Ether gönderilmedi.
    error NotEnoughEther();

    // Modifier'lar bir fonksiyonun gövdesini
    // değiştirmek için kullanılabilir.
    // Bu modifier kullanılırsa başa,
    // yalnızca fonksiyon belirli bir
    // adresten çağrıldığında geçen bir
    // kontrol ekleyecektir.
    modifier onlyBy(address account)
    {
        if (msg.sender != account)
            revert Unauthorized();
        // "_;" işaretini unutmayın! Modifier
        // kullanıldığında bu, gerçek fonksiyon
        // gövdesi ile değiştirilecektir.
        _;
    }

    /// `newOwner`'ı bu sözleşmenin yeni
    /// sahibi yapın.
    function changeOwner(address newOwner)
        public
        onlyBy(owner)
    {
        owner = newOwner;
    }

    modifier onlyAfter(uint time) {
        if (block.timestamp < time)
            revert TooEarly();
        _;
    }

    /// Sahiplik bilgilerini silin.
    /// Yalnızca sözleşme oluşturulduktan
    /// 6 hafta sonra çağrılabilir.
    function disown()
        public
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // Bu modifier, bir fonksiyon çağrısının belirli
    // bir ücretle ilişkilendirilmesini gerektirir.
    // Çağıran kişi çok fazla göndermişse yalnızca
    // fonksiyon gövdesinden sonrası iade edilir.
    // Bu, `_;` sonrasındaki kısmı atlamanın mümkün
    // olduğu Solidity sürümü 0.4.0 öncesinde tehlikeliydi.
    modifier costs(uint amount) {
        if (msg.value < amount)
            revert NotEnoughEther();

        _;
        if (msg.value > amount)
            payable(msg.sender).transfer(msg.value - amount);
    }

    function forceOwnerChange(address newOwner)
        public
        payable
        costs(200 ether)
    {
        owner = newOwner;
        // yalnızca örnek bir koşul
        if (uint160(owner) & 0 == 1)
            // Sürüm 0.4.0 öncesinde bu, Solidity
            // iade yapmıyordu.
            return;
        // fazla ödenen ücretleri iade et
    }
}
```

Fonksiyon çağrılarına erişimin kısıtlanabileceği daha özel bir yol, bir sonraki örnekte incelenecektir.

3.34.3 Durum Makinesi

Sözleşmeler, sıklıkla, bir durum makinesi işlevi görür; bu, içinde farklı davrandıkları veya farklı fonksiyonların çağrılabildiği belirli **aşamalara** sahip oldukları anlamına gelir.Bir fonksiyon çağrısı genellikle bir aşamayı sonlandırır ve sözleşmeyi bir sonraki aşamaya geçirir (özellikle sözleşme, **etkileşimi** modellediğinde). Bazı aşamalara belirli bir **anda** otomatik olarak ulaşılması da yaygındır.

Bunun bir örneği, "kör teklifleri kabul etme" aşamasından başlayan, "teklifleri açıklama" aşamasına geçen ve "ihale sonucunu belirleme" ile sonlanan kör ihale sözleşmesidir.

Bu durumda, durumları modellemek ve sözleşmenin yanlış kullanımına karşı korunmak için fonksiyon modifier'ları kullanılabilir.

Örnek

Aşağıdaki örnekte, `atStage` modifier'ı fonksiyonun yalnızca belirli bir aşamada çağrılmasını sağlar.

Otomatik zaman ayarlı geçişler, tüm fonksiyonlar tarafından kullanılması gereken `timedTransitions` modifier'ı ele alınır.

Not: Modifier Sırası Önemlidir. atStage, timedTransitions ile birleştirilirse yeni aşamanın dikkate alınması için atStage'i timedTransitions'tan sonra belirttiğinizden emin olun.

Son olarak, fonksiyon sonlandığında otomatik olarak bir sonraki aşamaya gitmek için `transitionNext` modifier'ı kullanılabilir.

Not: Modifier Atlanabilir. Bu, yalnızca 0.4.0 öncesi Solidity sürümlerinde geçerlidir: Modifier'lar, fonksiyon çağrısı kullanarak değil, yalnızca kodu değiştirerek uygulandığından fonksiyonun kendisi return kullanırsa transitionNext modifier'ındaki kod atlanabilir. Bunu yapmak isterseniz nextStage'i o fonksiyonlardan manuel olarak çağırdığınızdan emin olun. 0.4.0 sürümünden itibaren modifier kodu, fonksiyon açıkça retun etse dahi çalışacaktır.

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

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }
    /// Bu noktada fonksiyon çağrılamaz.
    error FunctionInvalidAtThisStage();

    // Mevcut aşama budur.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = block.timestamp;

    modifier atStage(Stages stage_) {
        if (stage != stage_)
            revert FunctionInvalidAtThisStage();
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Zaman ayarlı geçişler gerçekleştirin. Önce bu
    // modifier'ı belirttiğinizden emin olun aksi halde
    // korumalar yeni aşamayı dikkate almaz.
    modifier timedTransitions() {
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        if (stage == Stages.AcceptingBlindedBids &&
                    block.timestamp >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                block.timestamp >= creationTime + 12 days)
            nextStage();
        // Diğer aşamalar işleme göre geçiş yapar
        _;
    }

    // Burada modifier'ların sırası önemlidir!
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // Onu burada uygulamayacağız
    }

    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // Bu modifier, fonksiyonun tamamlanmasının
    // ardından sonraki aşamaya geçer.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

3.35 Bilinen Bugların Listesi

Aşağıda, Solidity derleyicisindeki güvenlikle ilgili bilinen bazı hataların JSON biçimli bir listesini bulabilirsiniz. Dosyanın kendisi Github repository'de barındırılmaktadır. Liste 0.3.0 sürümüne kadar uzanmaktadır, yalnızca bundan önceki sürümlerde mevcut olduğu bilinen hatalar listelenmemiştir.

Hangi hataların derleyicinin belirli bir sürümünü etkilediğini kontrol etmek için kullanılabilecek bugs_by_version.json adlı başka bir dosya daha vardır.

Sözleşme kaynağı doğrulama araçları ve ayrıca sözleşmelerle etkileşime giren diğer araçlar aşağıdaki kriterlere göre bu listeye başvurmalıdır:

- Bir sözleşmenin yayınlanmış bir sürüm yerine gecelik bir derleyici sürümüyle derlenmiş olması biraz şüphelidir. Bu liste yayınlanmamış veya gecelik sürümlerin kaydını tutmaz.
- Bir sözleşmenin, sözleşmenin oluşturulduğu sırada en yeni sürüm olmayan bir sürümle derlenmiş olması da hafif derecede şüphelidir. Diğer sözleşmelerden oluşturulan sözleşmeler için, oluşturma zincirini bir işleme kadar takip etmeniz ve oluşturma tarihi olarak bu işlemin tarihini kullanmanız gerekir.
- Bir sözleşmenin bilinen bir hata içeren bir derleyici ile derlenmiş olması ve sözleşmenin, düzeltme içeren daha yeni bir derleyici sürümünün zaten yayınlanmış olduğu bir zamanda oluşturulmuş olması son derece şüphelidir.

Aşağıdaki bilinen hataların JSON dosyası, her hata için bir tane olmak üzere aşağıdaki anahtarlara sahip bir nesne dizisidir:

uid
: Hataya `SOL-<year>-<number>` şeklinde verilen benzersiz tanımlayıcı. Aynı uid ile birden fazla giriş olması mümkündür. Bu, birden fazla sürüm aralığının aynı hatadan etkilendiği anlamına gelir.

name
: Hataya verilen benzersiz isim

summary
: Hatanın kısa açıklaması

description
: Hatanın ayrıntılı açıklaması

link
: Daha ayrıntılı bilgi içeren bir web sitesinin URL'si, isteğe bağlı

introduced
: Hatayı içeren ilk yayınlanan derleyici sürümü, isteğe bağlıdır

fixed
: Artık hata içermeyen ilk yayınlanan derleyici sürümü

publish
: Hatanın kamuoyu tarafından bilindiği tarih, isteğe bağlıdır

severity
: Hatanın ciddiyeti: çok düşük, düşük, orta, yüksek. Sözleşme testlerinde keşfedilebilirliği, ortaya çıkma olasılığını ve istismarların potansiyel zararını dikkate alır.

conditions
: Hatayı tetiklemek için karşılanması gereken koşullar. Aşağıdaki anahtarlar kullanılabilir: `optimizer`, hatayı etkinleştirmek için optimize edicinin açık olması gerektiği anlamına gelen Boolean değeri. `evmVersion`, hangi EVM sürümü derleyici ayarlarının hatayı tetiklediğini gösteren bir dize. Dize karşılaştırma operatörleri içerebilir. Örneğin, `">=constantinople"`, hatanın EVM sürümü `constantinople` veya üstü olarak ayarlandığında mevcut olduğu anlamına gelir. Herhangi bir koşul belirtilmezse, hatanın mevcut olduğu varsayılır.

check
: Bu alan, akıllı sözleşmenin hatayı içerip içermediğini bildiren farklı kontroller içerir. İlk kontrol türü, hatanın mevcut olması durumunda kaynak kodla ("source-regex") eşleştirilecek Javascript düzenli ifadeleridir. Eşleşme yoksa, hata büyük olasılıkla mevcut değildir. Eğer bir eşleşme varsa, hata mevcut olabilir. Daha iyi tutarlılık için, kontroller yorumlar çıkarıldıktan sonra kaynak koda uygulanmalıdır. İkinci kontrol türü, Solidity programının kompakt AST'sinde ("ast-compact-json-path") kontrol edilecek kalıplardır. Belirtilen arama sorgusu bir JsonPath ifadesidir. Solidity AST'nin en az bir yolu sorguyla eşleşiyorsa, hata muhtemelen mevcuttur.

```
[
    {
        "uid": "SOL-2022-6",
        "name": "AbiReencodingHeadOverflowWithStaticArrayCleanup",
        "summary": "ABI-encoding a tuple with a statically-sized calldata array in the␣
↪last component would corrupt 32 leading bytes of its first dynamically encoded␣
↪component.",
        "description": "When ABI-encoding a statically-sized calldata array, the␣
↪compiler always pads the data area to a multiple of 32-bytes and ensures that the␣
↪padding bytes are zeroed. In some cases, this cleanup used to be performed by always␣
↪writing exactly 32 bytes, regardless of how many needed to be zeroed. This was done␣
↪with the assumption that the data that would eventually occupy the area past the end␣
↪of the array had not yet been written, because the encoder processes tuple components␣
↪in the order they were given. While this assumption is mostly true, there is an␣
↪important corner case: dynamically encoded tuple components are stored separately from␣
↪the statically-sized ones in an area called the *tail* of the encoding and the tail␣
↪immediately follows the *head*, which is where the statically-sized components are␣
↪placed. The aforementioned cleanup, if performed for the last component of the head␣
↪would cross into the tail and overwrite up to 32 bytes of the first component stored␣
↪there with zeros. The only array type for which the cleanup could actually result in␣
↪an overwrite were arrays with ``uint256`` or ``bytes32`` as the base element type and␣
↪in this case the size of the corrupted area was always exactly 32 bytes. The problem␣
↪affected tuples at any nesting level. This included also structs, which are encoded as␣
↪tuples in the ABI. Note also that lists of parameters and return values of functions,␣
↪events and errors are encoded as tuples.",
        "introduced": "0.5.8",
        "fixed": "0.8.16",
        "severity": "medium",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2022-5",
        "name": "DirtyBytesArrayToStorage",
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        "summary": "Copying ``bytes`` arrays from memory or calldata to storage may␣
↪result in dirty storage values.",
        "description": "Copying ``bytes`` arrays from memory or calldata to storage is␣
↪done in chunks of 32 bytes even if the length is not a multiple of 32. Thereby, extra␣
↪bytes past the end of the array may be copied from calldata or memory to storage.␣
↪These dirty bytes may then become observable after a ``.push()`` without arguments to␣
↪the bytes array in storage, i.e. such a push will not result in a zero value at the␣
↪end of the array as expected. This bug only affects the legacy code generation␣
↪pipeline, the new code generation pipeline via IR is not affected.",
        "link": "https://blog.soliditylang.org/2022/06/15/dirty-bytes-array-to-storage-
↪bug/",
        "introduced": "0.0.1",
        "fixed": "0.8.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2022-4",
        "name": "InlineAssemblyMemorySideEffects",
        "summary": "The Yul optimizer may incorrectly remove memory writes from inline␣
↪assembly blocks, that do not access solidity variables.",
        "description": "The Yul optimizer considers all memory writes in the outermost␣
↪Yul block that are never read from as unused and removes them. This is valid when that␣
↪Yul block is the entire Yul program, which is always the case for the Yul code␣
↪generated by the new via-IR pipeline. Inline assembly blocks are never optimized in␣
↪isolation when using that pipeline. Instead they are optimized as a part of the whole␣
↪Yul input. However, the legacy code generation pipeline (which is still the default)␣
↪runs the Yul optimizer individually on an inline assembly block if the block does not␣
↪refer to any local variables defined in the surrounding Solidity code. Consequently,␣
↪memory writes in such inline assembly blocks are removed as well, if the written␣
↪memory is never read from in the same assembly block, even if the written memory is␣
↪accessed later, for example by a subsequent inline assembly block.",
        "link": "https://blog.soliditylang.org/2022/06/15/inline-assembly-memory-side-
↪effects-bug/",
        "introduced": "0.8.13",
        "fixed": "0.8.15",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2022-3",
        "name": "DataLocationChangeInInternalOverride",
        "summary": "It was possible to change the data location of the parameters or␣
↪return variables from ``calldata`` to ``memory`` and vice-versa while overriding␣
↪internal and public functions. This caused invalid code to be generated when calling␣
↪such a function internally through virtual function calls.",
        "description": "When calling external functions, it is irrelevant if the data␣
↪location of the parameters is ``calldata`` or ``memory``, the encoding of the data␣
↪does not change. Because of that, changing the data location when overriding external␣
↪functions is allowed. The compiler incorrectly also allowed a change in the data␣
↪location for overriding public and internal functions. Since public functions can be␣
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪called internally as well as externally, this causes invalid code to be generated when␣
↪such an incorrectly overridden function is called internally through the base contract.
↪ The caller provides a memory pointer, but the called function interprets it as a␣
↪calldata pointer or vice-versa.",
        "link": "https://blog.soliditylang.org/2022/05/17/data-location-inheritance-bug/
↪",
        "introduced": "0.6.9",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-2",
        "name": "NestedCallataArrayAbiReencodingSizeValidation",
        "summary": "ABI-reencoding of nested dynamic calldata arrays did not always␣
↪perform proper size checks against the size of calldata and could read beyond␣
↪``calldatasize()``.",
        "description": "Calldata validation for nested dynamic types is deferred until␣
↪the first access to the nested values. Such an access may for example be a copy to␣
↪memory or an index or member access to the outer type. While in most such accesses␣
↪calldata validation correctly checks that the data area of the nested array is␣
↪completely contained in the passed calldata (i.e. in the range [0, calldatasize()]),␣
↪this check may not be performed, when ABI encoding such nested types again directly␣
↪from calldata. For instance, this can happen, if a value in calldata with a nested␣
↪dynamic array is passed to an external call, used in ``abi.encode`` or emitted as␣
↪event. In such cases, if the data area of the nested array extends beyond␣
↪``calldatasize()``, ABI encoding it did not revert, but continued reading values from␣
↪beyond ``calldatasize()`` (i.e. zero values).",
        "link": "https://blog.soliditylang.org/2022/05/17/calldata-reencode-size-check-
↪bug/",
        "introduced": "0.5.8",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-1",
        "name": "AbiEncodeCallLiteralAsFixedBytesBug",
        "summary": "Literals used for a fixed length bytes parameter in ``abi.
↪encodeCall`` were encoded incorrectly.",
        "description": "For the encoding, the compiler only considered the types of the␣
↪expressions in the second argument of ``abi.encodeCall`` itself, but not the parameter␣
↪types of the function given as first argument. In almost all cases the abi encoding of␣
↪the type of the expression matches the abi encoding of the parameter type of the given␣
↪function. This is because the type checker ensures the expression is implicitly␣
↪convertible to the respective parameter type. However this is not true for number␣
↪literals used for fixed bytes types shorter than 32 bytes, nor for string literals␣
↪used for any fixed bytes type. Number literals were encoded as numbers instead of␣
↪being shifted to become left-aligned. String literals were encoded as dynamically␣
↪sized memory strings instead of being converted to a left-aligned bytes value.",
        "link": "https://blog.soliditylang.org/2022/03/16/encodecall-bug/",
        "introduced": "0.8.11",
        "fixed": "0.8.13",
        "severity": "very low"
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    },
    {
        "uid": "SOL-2021-4",
        "name": "UserDefinedValueTypesBug",
        "summary": "User defined value types with underlying type shorter than 32 bytes␣
↪used incorrect storage layout and wasted storage",
        "description": "The compiler did not correctly compute the storage layout of␣
↪user defined value types based on types that are shorter than 32 bytes. It would␣
↪always use a full storage slot for these types, even if the underlying type was␣
↪shorter. This was wasteful and might have problems with tooling or contract upgrades.",
        "link": "https://blog.soliditylang.org/2021/09/29/user-defined-value-types-bug/",
        "introduced": "0.8.8",
        "fixed": "0.8.9",
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-3",
        "name": "SignedImmutables",
        "summary": "Immutable variables of signed integer type shorter than 256 bits can␣
↪lead to values with invalid higher order bits if inline assembly is used.",
        "description": "When immutable variables of signed integer type shorter than 256␣
↪bits are read, their higher order bits were unconditionally set to zero. The correct␣
↪operation would be to sign-extend the value, i.e. set the higher order bits to one if␣
↪the sign bit is one. This sign-extension is performed by Solidity just prior to when␣
↪it matters, i.e. when a value is stored in memory, when it is compared or when a␣
↪division is performed. Because of that, to our knowledge, the only way to access the␣
↪value in its unclean state is by reading it through inline assembly.",
        "link": "https://blog.soliditylang.org/2021/09/29/signed-immutables-bug/",
        "introduced": "0.6.5",
        "fixed": "0.8.9",
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-2",
        "name": "ABIDecodeTwoDimensionalArrayMemory",
        "summary": "If used on memory byte arrays, result of the function ``abi.decode``␣
↪can depend on the contents of memory outside of the actual byte array that is decoded.
↪",
        "description": "The ABI specification uses pointers to data areas for everything␣
↪that is dynamically-sized. When decoding data from memory (instead of calldata), the␣
↪ABI decoder did not properly validate some of these pointers. More specifically, it␣
↪was possible to use large values for the pointers inside arrays such that computing␣
↪the offset resulted in an undetected overflow. This could lead to these pointers␣
↪targeting areas in memory outside of the actual area to be decoded. This way, it was␣
↪possible for ``abi.decode`` to return different values for the same encoded byte array.
↪",
        "link": "https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/",
        "introduced": "0.4.16",
        "fixed": "0.8.4",
        "conditions": {
            "ABIEncoderV2": true
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        },
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-1",
        "name": "KeccakCaching",
        "summary": "The bytecode optimizer incorrectly re-used previously evaluated␣
↪Keccak-256 hashes. You are unlikely to be affected if you do not compute Keccak-256␣
↪hashes in inline assembly.",
        "description": "Solidity's bytecode optimizer has a step that can compute Keccak-
↪256 hashes, if the contents of the memory are known during compilation time. This step␣
↪also has a mechanism to determine that two Keccak-256 hashes are equal even if the␣
↪values in memory are not known during compile time. This mechanism had a bug where␣
↪Keccak-256 of the same memory content, but different sizes were considered equal. More␣
↪specifically, ``keccak256(mpos1, length1)`` and ``keccak256(mpos2, length2)`` in some␣
↪cases were considered equal if ``length1`` and ``length2``, when rounded up to nearest␣
↪multiple of 32 were the same, and when the memory contents at ``mpos1`` and ``mpos2``␣
↪can be deduced to be equal. You maybe affected if you compute multiple Keccak-256␣
↪hashes of the same content, but with different lengths inside inline assembly. You are␣
↪unaffected if your code uses ``keccak256`` with a length that is not a compile-time␣
↪constant or if it is always a multiple of 32.",
        "link": "https://blog.soliditylang.org/2021/03/23/keccak-optimizer-bug/",
        "fixed": "0.8.3",
        "conditions": {
            "optimizer": true
        },
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-11",
        "name": "EmptyByteArrayCopy",
        "summary": "Copying an empty byte array (or string) from memory or calldata to␣
↪storage can result in data corruption if the target array's length is increased␣
↪subsequently without storing new data.",
        "description": "The routine that copies byte arrays from memory or calldata to␣
↪storage stores unrelated data from after the source array in the storage slot if the␣
↪source array is empty. If the storage array's length is subsequently increased either␣
↪by using ``.push()`` or by assigning to its ``.length`` attribute (only before 0.6.0),␣
↪the newly created byte array elements will not be zero-initialized, but contain the␣
↪unrelated data. You are not affected if you do not assign to ``.length`` and do not␣
↪use ``.push()`` on byte arrays, or only use ``.push(<arg>)`` or manually initialize␣
↪the new elements.",
        "link": "https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/",
        "fixed": "0.7.4",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-10",
        "name": "DynamicArrayCleanup",
        "summary": "When assigning a dynamically-sized array with types of size at most␣
↪16 bytes in storage causing the assigned array to shrink, some parts of deleted slots␣
↪were not zeroed out.",
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        "description": "Consider a dynamically-sized array in storage whose base-type is␣
↪small enough such that multiple values can be packed into a single slot, such as␣
↪`uint128[]`. Let us define its length to be `l`. When this array gets assigned from␣
↪another array with a smaller length, say `m`, the slots between elements `m` and `l`␣
↪have to be cleaned by zeroing them out. However, this cleaning was not performed␣
↪properly. Specifically, after the slot corresponding to `m`, only the first packed␣
↪value was cleaned up. If this array gets resized to a length larger than `m`, the␣
↪indices corresponding to the unclean parts of the slot contained the original value,␣
↪instead of 0. The resizing here is performed by assigning to the array `length`, by a␣
↪`push()` or via inline assembly. You are not affected if you are only using `.push(
↪<arg>)` or if you assign a value (even zero) to the new elements after increasing the␣
↪length of the array.",
        "link": "https://blog.soliditylang.org/2020/10/07/solidity-dynamic-array-cleanup-
↪bug/",
        "fixed": "0.7.3",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-9",
        "name": "FreeFunctionRedefinition",
        "summary": "The compiler does not flag an error when two or more free functions␣
↪with the same name and parameter types are defined in a source unit or when an␣
↪imported free function alias shadows another free function with a different name but␣
↪identical parameter types.",
        "description": "In contrast to functions defined inside contracts, free␣
↪functions with identical names and parameter types did not create an error. Both␣
↪definition of free functions with identical name and parameter types and an imported␣
↪free function with an alias that shadows another function with a different name but␣
↪identical parameter types were permitted due to which a call to either the multiply␣
↪defined free function or the imported free function alias within a contract led to the␣
↪execution of that free function which was defined first within the source unit.␣
↪Subsequently defined identical free function definitions were silently ignored and␣
↪their code generation was skipped.",
        "introduced": "0.7.1",
        "fixed": "0.7.2",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-8",
        "name": "UsingForCalldata",
        "summary": "Function calls to internal library functions with calldata␣
↪parameters called via ``using for`` can result in invalid data being read.",
        "description": "Function calls to internal library functions using the ``using␣
↪for`` mechanism copied all calldata parameters to memory first and passed them on like␣
↪that, regardless of whether it was an internal or an external call. Due to that, the␣
↪called function would receive a memory pointer that is interpreted as a calldata␣
↪pointer. Since dynamically sized arrays are passed using two stack slots for calldata,␣
↪but only one for memory, this can lead to stack corruption. An affected library call␣
↪will consider the JUMPDEST to which it is supposed to return as part of its arguments␣
↪and will instead jump out to whatever was on the stack before the call.",
        "introduced": "0.6.9",
        "fixed": "0.6.10",
```

(sonraki sayfaya devam)

(önceki sayfadan devam)


```
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-7",
        "name": "MissingEscapingInFormatting",
        "summary": "String literals containing double backslash characters passed␣
↪directly to external or encoding function calls can lead to a different string being␣
↪used when ABIEncoderV2 is enabled.",
        "description": "When ABIEncoderV2 is enabled, string literals passed directly to␣
↪encoding functions or external function calls are stored as strings in the intemediate␣
↪code. Characters outside the printable range are handled correctly, but backslashes␣
↪are not escaped in this procedure. This leads to double backslashes being reduced to␣
↪single backslashes and consequently re-interpreted as escapes potentially resulting in␣
↪a different string being encoded.",
        "introduced": "0.5.14",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-6",
        "name": "ArraySliceDynamicallyEncodedBaseType",
        "summary": "Accessing array slices of arrays with dynamically encoded base types␣
↪(e.g. multi-dimensional arrays) can result in invalid data being read.",
        "description": "For arrays with dynamically sized base types, index range␣
↪accesses that use a start expression that is non-zero will result in invalid array␣
↪slices. Any index access to such array slices will result in data being read from␣
↪incorrect calldata offsets. Array slices are only supported for dynamic calldata types␣
↪and all problematic type require ABIEncoderV2 to be enabled.",
        "introduced": "0.6.0",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-5",
        "name": "ImplicitConstructorCallvalueCheck",
        "summary": "The creation code of a contract that does not define a constructor␣
↪but has a base that does define a constructor did not revert for calls with non-zero␣
↪value.",
        "description": "Starting from Solidity 0.4.5 the creation code of contracts␣
↪without explicit payable constructor is supposed to contain a callvalue check that␣
↪results in contract creation reverting, if non-zero value is passed. However, this␣
↪check was missing in case no explicit constructor was defined in a contract at all,␣
↪but the contract has a base that does define a constructor. In these cases it is␣
↪possible to send value in a contract creation transaction or using inline assembly␣
↪without revert, even though the creation code is supposed to be non-payable.",
        "introduced": "0.4.5",
```


(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        "fixed": "0.6.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-4",
        "name": "TupleAssignmentMultiStackSlotComponents",
        "summary": "Tuple assignments with components that occupy several stack slots, i.
↪e. nested tuples, pointers to external functions or references to dynamically sized␣
↪calldata arrays, can result in invalid values.",
        "description": "Tuple assignments did not correctly account for tuple components␣
↪that occupy multiple stack slots in case the number of stack slots differs between␣
↪left-hand-side and right-hand-side. This can either happen in the presence of nested␣
↪tuples or if the right-hand-side contains external function pointers or references to␣
↪dynamic calldata arrays, while the left-hand-side contains an omission.",
        "introduced": "0.1.6",
        "fixed": "0.6.6",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-3",
        "name": "MemoryArrayCreationOverflow",
        "summary": "The creation of very large memory arrays can result in overlapping␣
↪memory regions and thus memory corruption.",
        "description": "No runtime overflow checks were performed for the length of␣
↪memory arrays during creation. In cases for which the memory size of an array in bytes,
↪ i.e. the array length times 32, is larger than 2^256-1, the memory allocation will␣
↪overflow, potentially resulting in overlapping memory areas. The length of the array␣
↪is still stored correctly, so copying or iterating over such an array will result in␣
↪out-of-gas.",
        "link": "https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-bug/",
        "introduced": "0.2.0",
        "fixed": "0.6.5",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and continue␣
↪and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared inside␣
↪a for loop if they can be removed in a control-flow branch that ends with ``break`` or␣
↪``continue`` even though they cannot be removed in other control-flow branches.␣
↪Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.6.0",
        "fixed": "0.6.1",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        }
    },
    {
        "uid": "SOL-2020-2",
        "name": "privateCanBeOverridden",
        "summary": "Private methods can be overridden by inheriting contracts.",
        "description": "While private methods of base contracts are not visible and␣
↪cannot be called directly from the derived contract, it is still possible to declare a␣
↪function of the same name and type and thus change the behaviour of the base contract
↪'s function.",
        "introduced": "0.3.0",
        "fixed": "0.5.17",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue0.5",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and continue␣
↪and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared inside␣
↪a for loop if they can be removed in a control-flow branch that ends with ``break`` or␣
↪``continue`` even though they cannot be removed in other control-flow branches.␣
↪Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.5.8",
        "fixed": "0.5.16",
        "severity": "low",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2019-10",
        "name": "ABIEncoderV2LoopYulOptimizer",
        "summary": "If both the experimental ABIEncoderV2 and the experimental Yul␣
↪optimizer are activated, one component of the Yul optimizer may reuse data in memory␣
↪that has been changed in the meantime.",
        "description": "The Yul optimizer incorrectly replaces ``mload`` and ``sload``␣
↪calls with values that have been previously written to the load location (and␣
↪potentially changed in the meantime) if all of the following conditions are met: (1)␣
↪there is a matching ``mstore`` or ``sstore`` call before; (2) the contents of memory␣
↪or storage is only changed in a function that is called (directly or indirectly) in␣
↪between the first store and the load call; (3) called function contains a for loop␣
↪where the same memory location is changed in the condition or the post or body block.␣
↪When used in Solidity mode, this can only happen if the experimental ABIEncoderV2 is␣
↪activated and the experimental Yul optimizer has been activated manually in addition␣
↪to the regular optimizer in the compiler settings.",
        "introduced": "0.5.14",
        "fixed": "0.5.15",
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true,
            "optimizer": true,
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2019-9",
        "name":
↪"ABIEncoderV2CalldataStructsWithStaticallySizedAndDynamicallyEncodedMembers",
        "summary": "Reading from calldata structs that contain dynamically encoded, but␣
↪statically-sized members can result in incorrect values.",
        "description": "When a calldata struct contains a dynamically encoded, but␣
↪statically-sized member, the offsets for all subsequent struct members are calculated␣
↪incorrectly. All reads from such members will result in invalid values. Only calldata␣
↪structs are affected, i.e. this occurs in external functions with such structs as␣
↪argument. Using affected structs in storage or memory or as arguments to public␣
↪functions on the other hand works correctly.",
        "introduced": "0.5.6",
        "fixed": "0.5.11",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-8",
        "name": "SignedArrayStorageCopy",
        "summary": "Assigning an array of signed integers to a storage array of␣
↪different type can lead to data corruption in that array.",
        "description": "In two's complement, negative integers have their higher order␣
↪bits set. In order to fit into a shared storage slot, these have to be set to zero.␣
↪When a conversion is done at the same time, the bits to set to zero were incorrectly␣
↪determined from the source and not the target type. This means that such copy␣
↪operations can lead to incorrect values being stored.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.7",
        "fixed": "0.5.10",
        "severity": "low/medium"
    },
    {
        "uid": "SOL-2019-7",
        "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
        "summary": "Storage arrays containing structs or other statically-sized arrays␣
↪are not read properly when directly encoded in external function calls or in abi.
↪encode*.",
        "description": "When storage arrays whose elements occupy more than a single␣
↪storage slot are directly encoded in external function calls or using abi.encode*,␣
↪their elements are read in an overlapping manner, i.e. the element pointer is not␣
↪properly advanced between reads. This is not a problem when the storage data is first␣
↪copied to a memory variable or if the storage array only contains value types or␣
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪dynamically-sized arrays.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
        "introduced": "0.4.16",
        "fixed": "0.5.10",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-6",
        "name": "DynamicConstructorArgumentsClippedABIV2",
        "summary": "A contract's constructor that takes structs or arrays that contain␣
↪dynamically-sized arrays reverts or decodes to invalid data.",
        "description": "During construction of a contract, constructor parameters are␣
↪copied from the code section to memory for decoding. The amount of bytes to copy was␣
↪calculated incorrectly in case all parameters are statically-sized but contain␣
↪dynamically-sized arrays as struct members or inner arrays. Such types are only␣
↪available if ABIEncoderV2 is activated.",
        "introduced": "0.4.16",
        "fixed": "0.5.9",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special jump␣
↪target only considered the situation after deployment.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor_0.4.x",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special jump␣
↪target only considered the situation after deployment.",
        "introduced": "0.4.5",
        "fixed": "0.4.26",
        "severity": "very low"
    },
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries",
        "summary": "Contract types used in events in libraries cause an incorrect event␣
↪signature hash",
        "description": "Instead of using the type `address` in the hashed signature, the␣
↪actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries_0.4.x",
        "summary": "Contract types used in events in libraries cause an incorrect event␣
↪signature hash",
        "description": "Instead of using the type `address` in the hashed signature, the␣
↪actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.3.0",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but can␣
↪also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.0",
        "fixed": "0.5.7",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage_0.4.x",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but can␣
↪also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪abiencoderv2-bug/",
        "introduced": "0.4.19",
        "fixed": "0.4.26",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-2",
        "name": "IncorrectByteInstructionOptimization",
        "summary": "The optimizer incorrectly handles byte opcodes whose second argument␣
↪is 31 or a constant expression that evaluates to 31. This can result in unexpected␣
↪values.",
        "description": "The optimizer incorrectly handles byte opcodes that use the␣
↪constant 31 as second argument. This can happen when performing index access on␣
↪bytesNN types with a compile-time constant value (not index) of 31 or when using the␣
↪byte opcode in inline assembly.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.7",
        "severity": "very low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2019-1",
        "name": "DoubleShiftSizeOverflow",
        "summary": "Double bitwise shifts by large constants whose sum overflows 256␣
↪bits can result in unexpected values.",
        "description": "Nested logical shift operations whose total shift size is 2**256␣
↪or more are incorrectly optimized. This only applies to shifts by numbers of bits that␣
↪are compile-time constant expressions.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.6",
        "severity": "low",
        "conditions": {
            "optimizer": true,
            "evmVersion": ">=constantinople"
        }
    },
    {
        "uid": "SOL-2018-4",
        "name": "ExpExponentCleanup",
        "summary": "Using the ** operator with an exponent of type shorter than 256 bits␣
↪can result in unexpected values.",
        "description": "Higher order bits in the exponent are not properly cleaned␣
↪before the EXP opcode is applied if the type of the exponent expression is smaller␣
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪than 256 bits and not smaller than the type of the base. In that case, the result␣
↪might be larger than expected if the exponent is assumed to lie within the value range␣
↪of the type. Literal numbers as exponents are unaffected as are exponents or bases of␣
↪type uint256.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "fixed": "0.4.25",
        "severity": "medium/high",
        "check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
    },
    {
        "uid": "SOL-2018-3",
        "name": "EventStructWrongData",
        "summary": "Using structs in events logged wrong data.",
        "description": "If a struct is used in an event, the address of the struct is␣
↪logged instead of the actual data.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.4.17",
        "fixed": "0.4.25",
        "severity": "very low",
        "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')]..[?
↪(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
↪'struct'))]"}
    },
    {
        "uid": "SOL-2018-2",
        "name": "NestedArrayFunctionCallDecoder",
        "summary": "Calling functions that return multi-dimensional fixed-size arrays␣
↪can result in memory corruption.",
        "description": "If Solidity code calls a function that returns a multi-
↪dimensional fixed-size array, array elements are incorrectly interpreted as memory␣
↪pointers and thus can cause memory corruption if the return values are accessed.␣
↪Calling functions with multi-dimensional fixed-size arrays is unaffected as is␣
↪returning fixed-size arrays from function calls. The regular expression only checks if␣
↪such functions are present, not if they are called, which is required for the contract␣
↪to be affected.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.1.4",
        "fixed": "0.4.22",
        "severity": "medium",
        "check": {"regex-source": "returns[^;{]*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\]\\
↪s*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\][^{;]*[;{]"}
    },
    {
        "uid": "SOL-2018-1",
        "name": "OneOfTwoConstructorsSkipped",
        "summary": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored.",
        "description": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored. There will be a compiler␣
↪warning about the old-style constructor, so contracts only using new-style␣
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪constructors are fine.",
        "introduced": "0.4.22",
        "fixed": "0.4.23",
        "severity": "very low"
    },
    {
        "uid": "SOL-2017-5",
        "name": "ZeroFunctionSelector",
        "summary": "It is possible to craft the name of a function such that it is␣
↪executed instead of the fallback function in very specific circumstances.",
        "description": "If a function has a selector consisting only of zeros, is␣
↪payable and part of a contract that does not have a fallback function and at most five␣
↪external functions in total, this function is called instead of the fallback function␣
↪if Ether is sent to the contract without data.",
        "fixed": "0.4.18",
        "severity": "very low"
    },
    {
        "uid": "SOL-2017-4",
        "name": "DelegateCallReturnValue",
        "summary": "The low-level .delegatecall() does not return the execution outcome,␣
↪but converts the value returned by the functioned called to a boolean instead.",
        "description": "The return value of the low-level .delegatecall() function is␣
↪taken from a position in memory, where the call data or the return data resides. This␣
↪value is interpreted as a boolean and put onto the stack. This means if the called␣
↪function returns at least 32 zero bytes, .delegatecall() returns false even if the␣
↪call was successful.",
        "introduced": "0.3.0",
        "fixed": "0.4.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2017-3",
        "name": "ECRecoverMalformedInput",
        "summary": "The ecrecover() builtin can return garbage for malformed input.",
        "description": "The ecrecover precompile does not properly signal failure for␣
↪malformed input (especially in the 'v' argument) and thus the Solidity function can␣
↪return data that was previously present in the return area in memory.",
        "fixed": "0.4.14",
        "severity": "medium"
    },
    {
        "uid": "SOL-2017-2",
        "name": "SkipEmptyStringLiteral",
        "summary": "If \"\" is used in a function call, the following function arguments␣
↪will not be correctly passed to the function.",
        "description": "If the empty string literal \"\" is used as an argument in a␣
↪function call, it is skipped by the encoder. This has the effect that the encoding of␣
↪all arguments following this is shifted left by 32 bytes and thus the function call␣
↪data is corrupted.",
        "fixed": "0.4.12",
        "severity": "low"
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    },
    {
        "uid": "SOL-2017-1",
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode by␣
↪routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.soliditylang.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-11",
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying memory,
↪ ignored its return value. On the public chain, calls to the identity precompile can␣
↪be made in a way that they never fail, but this might be different on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "uid": "SOL-2016-10",
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At␣
↪jump destinations, multiple code paths join and thus it has to compute a common state␣
↪from the incoming edges. Computing this common state was simplified to just use the␣
↪empty state, but this implementation was not done properly. This bug can cause data␣
↪corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-9",
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly and␣
↪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␣
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
↪higher order bytes were not cleaned properly, which made it sometimes possible to␣
↪overwrite a variable in storage when writing to another one.",
        "link": "https://blog.soliditylang.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "uid": "SOL-2016-8",
        "name": "OptimizerStaleKnowledgeAboutSHA3",
        "summary": "The optimizer did not properly reset its knowledge about SHA3␣
↪operations resulting in some hashes (also used for storage variable positions) not␣
↪being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-
↪evaluating expressions whose value is already known. This knowledge was not properly␣
↪reset across control flow paths and thus the optimizer sometimes thought that the␣
↪result of a SHA3 operation is already present on the stack. This could result in data␣
↪corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-7",
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that␣
↪received Ether.",
        "description": "Library functions are protected against sending them Ether␣
↪through a call. Since the DELEGATECALL opcode forwards the information about how much␣
↪Ether was sent with a call, the library function incorrectly assumed that Ether was␣
↪sent to the library and threw an exception.",
        "severity": "low",
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "uid": "SOL-2016-6",
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if no␣
↪Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a␣
↪certain amount of gas from the EVM to handle the transfer. In the case of a zero-
↪transfer, this gas is not provided which causes the recipient to throw an exception.",
        "severity": "low",
        "fixed": "0.4.0"
    },
    {
        "uid": "SOL-2016-5",
        "name": "DynamicAllocationInfiniteLoop",
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        "summary": "Dynamic allocation of an empty memory array caused an infinite loop␣
↪and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this length␣
↪is zero, code was generated that did not terminate and thus consumed all gas.",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "uid": "SOL-2016-4",
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At␣
↪jump destinations, multiple code paths join and thus it has to compute a common state␣
↪from the incoming edges. Computing this common state was not done correctly. This bug␣
↪can cause data corruption, but it is probably quite hard to use for targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-3",
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned before␣
↪comparison.",
        "description": "Two variables of type bytesNN were considered different if their␣
↪higher order bits, which are not part of the actual value, were different. An attacker␣
↪might use this to reach seemingly unreachable code paths by providing incorrectly␣
↪formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "uid": "SOL-2016-2",
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32 bytes␣
↪did not correctly clean the higher order bits, causing corruption in other array␣
↪elements.",
        "description": "Multiple elements of an array of values that are shorter than 17␣
↪bytes are packed into the same storage slot. Writing to a single element of such an␣
↪array did not properly clean the higher order bytes and thus could lead to data␣
↪corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "uid": "SOL-2016-1",
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several␣
↪undocumented or undiscovered bugs.",
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        "description": "The list of bugs is only kept for compiler versions starting␣
↪from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
]
```

3.36 Katkıda Bulunmak

Yardıma her zaman açığız ve Solidity'ye nasıl katkıda bulunabileceğinize dair pek çok seçenek var.

Özellikle aşağıdaki alanlardaki destek için minnettar olduğumuzu belirtmek isteriz:

- Sorunların raporlanması.
- Solidity'nin GitHub sorunlarını düzeltmek ve yanıtlamak, özellikle de dışarıdan katkıda bulunanlar için giriş sorunları olarak tasarlanan "good first issue" olarak etiketlenenler.
- Dokümantasyonun iyileştirilmesi.
- Dokümantasyonun daha fazla dile çevrilmesi.
- StackExchange'de diğer kullanıcıların sorularını yanıtlama ve Solidity Gitter Chat.
- Solidity forumunda dil değişiklikleri veya yeni özellikler önererek ve geri bildirim sağlayarak dil tasarım sürecine dahil olmak.

Başlamak için, Solidity bileşenlerine ve derleme sürecine aşina olmak için *Kaynağından Kurulum* u deneyebilirsiniz. Ayrıca, Solidity'de akıllı sözleşmeler yazma konusunda uzmanlaşmak da faydalı olabilir.

Lütfen bu projenin bir Katılımcı Davranış Kuralları ile yayınlandığını unutmayın. Bu projeye katılarak - sorunlarda, pull request' lerde veya Gitter kanallarında - şartlarına uymayı kabul etmiş olursunuz.

3.36.1 Takım Toplantıları

Tartışmak istediğiniz sorunlar veya pull request'ler varsa ya da ekibin ve katkıda bulunanların neler üzerinde çalıştığını duymak istiyorsanız, herkese açık takım toplantılarımıza katılabilirsiniz:

- Pazartesi günleri saat 15:00 CET/CEST. Türkiye saati ile 16:00,
- Çarşamba günleri 14:00 CET/CEST.

Her iki çağrı da Jitsi üzerinde gerçekleşir.

3.36.2 Sorunlar Nasıl Rapor Edilir

Bir sorunu bildirmek için lütfen GitHub sorunları izleyicisini kullanın. Sorunları bildirirken lütfen aşağıdaki ayrıntıları belirtin:

- Solidity sürümü.
- Kaynak kodu (varsa).
- İşletim sistemi.
- Sorunu yeniden üretmek için adımlar.

- Mevcut ve beklenen davranış.

Soruna neden olan kaynak kodunu en aza indirmek her zaman sorunların çözümüne yardımcı olur ve hatta bazen bir yanlış anlaşılmayı açıklığa kavuşturur.

3.36.3 Pull Request'ler için İş Akışı(Workflow)

Katkıda bulunmak için lütfen `develop` dalını forklayın ve değişikliklerinizi orada yapın. Commit mesajlarınızda *ne* yaptığınızın yanı sıra *neden* değişiklik yaptığınız da belirtilmelidir (çok küçük bir değişiklik olmadığı sürece).

Fork yaptıktan sonra `develop``tan herhangi bir değişiklik çekmeniz(pull) gerekiyorsa (örneğin, olası merge conflict`leri çözmek için), lütfen ``git merge` kullanmaktan kaçının ve bunun yerine branch`inizi `git rebase` yapın. Bu, değişikliğinizi daha kolay gözden geçirmemize yardımcı olacaktır.

Ayrıca, yeni bir özellik yazıyorsanız, lütfen `test/` altına uygun test örneklerini eklediğinizden emin olun (aşağıya bakınız).

Bununla birlikte, daha büyük bir değişiklik yapıyorsanız, lütfen önce Solidity Development Gitter kanalına (yukarıda bahsedilenden farklı olarak, bu kanal dil kullanımı yerine derleyici ve dil geliştirmeye odaklanmıştır) danışın.

Yeni özellikler ve hata düzeltmeleri `Changelog.md` dosyasına eklenmelidir: lütfen uygun durumlarda önceki girişlerin stilini takip edin.

Son olarak, lütfen bu proje için kodlama stiline uyduğunuzdan emin olun. Ayrıca, CI testi yapmamıza rağmen, lütfen kodunuzu test edin ve bir pull request göndermeden önce yerel olarak derlendiğinden emin olun.

Yardımlarınız için teşekkür ederiz!

3.36.4 Derleyici Testlerini Çalıştırma

Ön Koşullar

Tüm derleyici testlerini çalıştırmak için isteğe bağlı olarak birkaç bağlayıcı faktör yüklemek isteyebilirsiniz. yüklemek isteyebilirsiniz (evmone, libz3 ve libhera).

macOS üzerinde bazı test komut dosyaları GNU coreutils'in kurulu olmasını beklemektedir. Bu en kolay Homebrew kullanılarak gerçekleştirilebilir: `brew install coreutils`.

Windows sistemlerinde ortak bağlantı oluşturma ayrıcalığına sahip olduğunuzdan emin olun, aksi takdirde bazı testler başarısız olabilir. Yöneticilerin bu ayrıcalığa sahip olması gerekir, ancak diğer kullanıcılara da verebilirsiniz veya Geliştirici Modunu etkinleştirebilirsiniz.

Testleri Çalıştırma

Solidity, çoğu Boost C++ Test Framework uygulaması `soltest` içinde paketlenmiş farklı test türleri içerir. Çoğu değişiklik için `build/test/soltest` veya onun paketleyicisi olan `scripts/soltest.sh` dosyasını çalıştırmak yeterlidir.

./scripts/tests.sh`` betiği, Boost C++ Test Framework uygulaması `soltest` (veya paketleyicisi `scripts/soltest.sh`) ile birlikte komut satırı testleri ve derleme testleri de dahil olmak üzere çoğu Solidity testini otomatik olarak yürütür.

Test sistemi, anlamsal testleri çalıştırmak için otomatik olarak evmone konumunu keşfetmeye çalışır.

`evmone` kütüphanesi, geçerli çalışma dizinine, üst dizinine veya üst dizinin üst dizinine göre `deps` veya `deps/lib` dizininde bulunmalıdır. Alternatif olarak `evmone` paylaşımlı nesnesi için açık bir konum `ETH_EVMONE` ortam değişkeni aracılığıyla belirtilebilir.

`evmone` esas olarak semantik ve gaz testlerini çalıştırmak için gereklidir. Eğer yüklü değilse, `scripts/soltest.sh` dosyasına `--no-semantic-tests` parametresini girerek bu testleri atlayabilirsiniz.

Ewasm testlerinin çalıştırılması varsayılan olarak devre dışıdır ve `./scripts/soltest.sh --ewasm` aracılığıyla açıkça etkinleştirilebilir ve `hera <https://github.com/ewasm/hera>`_ kütüphanesinin ``soltest`` tarafından bulunmasını gerektirir. `hera` kütüphanesini bulma mekanizması `evmone` ile aynıdır, ancak açık bir konum belirtmek için kullanılan değişken `ETH_HERA` olarak adlandırılır.

`evmone` ve `hera` kütüphanelerinin her ikisi de Linux'ta `.so`, Windows sistemlerinde `.dll` ve macOS'ta `.dylib` dosya adı uzantısı ile bitmelidir.

SMT testlerini çalıştırmak için, `libz3` kütüphanesi yüklenmeli ve derleyici yapılandırma aşamasında `cmake` tarafından bulunabilmelidir.

Eğer `libz3` kütüphanesi sisteminizde yüklü değilse, `./scripts/tests.sh` dosyasını çalıştırmadan önce `SMT_FLAGS=--no-smt` komutunu vererek veya `./scripts/soltest.sh -no smt` dosyasını çalıştırarak SMT testlerini devre dışı bırakmalısınız. Bu testler `libsolidity/smtCheckerTests` ve `libsolidity/smtCheckerTestsJSON` testleridir.

Not: Soltest tarafından çalıştırılan tüm birim testlerinin bir listesini almak için `./build/test/soltest --list_content=HRF` komutunu çalıştırın.

Daha hızlı sonuç almak için testlerin bir alt kümesini veya belirli testleri çalıştırabilirsiniz.

To run a subset of tests, you can use filters: `./scripts/soltest.sh -t TestSuite/TestName`, where `TestName` can be a wildcard *.

Ya da örneğin, yul disambiguator ile ilgili tüm testleri çalıştırmak için: `./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`.

`./build/test/soltest --help` mevcut tüm seçenekler hakkında ayrıntılı bir yardım sağlar.

Özellikle bakınız:

- Testin tamamlandığını göstermek için show_progress (-p),
- Belirli test durumlarını çalıştırmak için run_test (-t) ve
- report-level (-r) daha ayrıntılı bir rapor verir.

Not: Windows ortamında çalışanlar yukarıdaki temel setleri libz3 olmadan çalıştırmak isterler. Git Bash kullanarak, şunları kullanabilirsiniz: `./build/test/Release/soltest.exe -- --no-smt`. Bunu düz Komut İstemi'nde çalıştırıyorsanız, `.\build\test\Release\soltest.exe -- --no-smt` kullanın.

GDB kullanarak hata ayıklamak istiyorsanız, "normalden" farklı bir şekilde derlediğinizden emin olun. Örneğin, `build` klasörünüzde aşağıdaki komutu çalıştırabilirsiniz: .. code-block:: bash

    cmake -DCMAKE_BUILD_TYPE=Debug .. make

Bu, `--debug` parametresini kullanarak bir testte hata ayıkladığınızda, bozabileceğiniz veya yazdırabileceğiniz fonksiyonlara ve değişkenlere erişebilmeniz için semboller oluşturur.

CI, Emscripten hedefinin derlenmesini gerektiren ek testler (`solc-js` ve üçüncü taraf Solidity çerçevelerinin test edilmesi dahil) çalıştırır.

Sözdizimi Testleri Yazma ve Çalıştırma

Sözdizimi testleri, derleyicinin geçersiz kod için doğru hata mesajlarını oluşturduğunu ve geçerli kodu düzgün bir şekilde kabul ettiğini kontrol eder. Bunlar `tests/libsolidity/syntaxTests` klasörü içindeki ayrı dosyalarda saklanır. Bu dosyalar, ilgili testin beklenen sonuç(lar)ını belirten ek açıklamalar içermelidir. Test paketi bunları derler ve verilen beklentilere göre kontrol eder.

Örneğin: `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Tanımlayıcı zaten bildirilmiş.
```

Bir sözdizimi testi, en azından test edilen sözleşmenin kendisini ve ardından `// ----` ayırıcısını içermelidir. Ayırıcıyı takip eden yorumlar, beklenen derleyici hatalarını veya uyarılarını tanımlamak için kullanılır. Sayı aralığı, kaynakta hatanın meydana geldiği konumu belirtir. Sözleşmenin herhangi bir hata veya uyarı olmadan derlenmesini istiyorsanız, ayırıcıyı ve onu takip eden yorumları dışarıda bırakabilirsiniz.

Yukarıdaki örnekte, `variable` durum değişkeni iki kez bildirilmiştir, buna izin verilmez. Bu, tanımlayıcının zaten bildirilmiş olduğunu belirten bir `DeclarationError` ile sonuçlanır.

Bu testler için `isoltest` aracı kullanılır ve bu aracı `./build/test/tools/` altında bulabilirsiniz. Tercih ettiğiniz metin editörünü kullanarak başarısız sözleşmelerin düzenlenmesine izin veren etkileşimli bir araçtır. Şimdi `variable` ifadesinin ikinci bildirimini kaldırarak bu testi çözmeye çalışalım:

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Tanımlayıcı zaten bildirilmiş.
```

Tekrar `./build/test/tools/isoltest` çalıştırıldığında test başarısız olur:

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
        contract test {
            uint256 variable;
        }

      Beklenen sonuç:
          DeclarationError: (36-52): Tanımlayıcı zaten bildirilmiş.
      Elde edilen sonuç:
          Başarılı
```

`isoltest` elde edilen sonucun yanına beklenen sonucu yazdırır ve ayrıca mevcut sözleşme dosyasını düzenlemek, güncellemek veya atlamak ya da uygulamadan çıkmak için bir yol sağlar.

Başarısız testler için çeşitli seçenekler sunar:

- `edit`: `isoltest` sözleşmeyi bir editörde açmaya çalışır, böylece onu ayarlayabilirsiniz. Ya komut satırında (`isoltest --editor /path/to/editor` şeklinde), ya `EDITOR` ortam değişkeninde ya da sadece `/usr/bin/editor` (bu sırayla) verilen editörü kullanır.
- `update`: Test edilen sözleşme için beklentileri günceller. Bu, karşılanmamış beklentileri kaldırarak ve eksik beklentileri ekleyerek ek açıklamaları günceller. Test daha sonra tekrar çalıştırılır.

- `skip`: Bu belirli testin yürütülmesini atlar.
- `quit`: isoltest`` testinden çıkar.

Bu seçeneklerin tümü, tüm test sürecini durduran `quit` dışında mevcut sözleşme için geçerlidir.

Yukarıdaki testin otomatik olarak güncellenmesi onu şu şekilde değiştirir

```
contract test {
    uint256 variable;
}
// ----
```

ve testi yeniden çalıştırır. Şimdi tekrar geçer:

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

Not: Sözleşme dosyası için neyi test ettiğini açıklayan bir isim seçin, örneğin `double_variable_declaration.sol`. Kalıtım veya çapraz sözleşme çağrılarını test etmediğiniz sürece, tek bir dosyaya birden fazla sözleşme koymayın. Her dosya yeni özelliğinizin bir yönünü test etmelidir.

3.36.5 Fuzzer'ı AFL ile Çalıştırma

Fuzzing, istisnai yürütme durumlarını (segmentasyon hataları, istisnalar, vb.) bulmak için programları az çok rastgele girdiler üzerinde çalıştıran bir tekniktir. Modern fuzzer'lar akıllıdır ve girdi içinde yönlendirilmiş bir arama yaparlar. Kaynak kodunu girdi olarak alan ve dahili bir derleyici hatası, segmentasyon hatası veya benzeriyle karşılaştığında başarısız olan, ancak örneğin kod bir hata içeriyorsa başarısız olmayan `solfuzzer` adlı özel bir binary'ye sahibiz. Bu şekilde, fuzzing araçları derleyicideki dahili sorunları bulabilir.

Biz fuzzing için çoğunlukla AFL kullanıyoruz. AFL paketlerini depolarınızdan indirip kurmanız (afl, afl-clang) ya da elle derlemeniz gerekir. Ardından, derleyiciniz olarak AFL ile Solidity'yi (veya sadece `solfuzzer` binary'sini) derleyin:

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

Bu aşamada aşağıdakine benzer bir mesaj görebilmeniz gerekir:

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

Program mesajları görünmediyse, AFL'nin clang binary'lerine işaret eden cmake bayraklarını değiştirmeyi deneyin:

```
# if previously failed
make clean
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Aksi takdirde, yürütme sırasında fuzzer binary'nin enstrümante edilmediğini belirten bir hata ile duracaktır:

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6920
```

Ardından, bazı örnek kaynak dosyalara ihtiyacınız var. Bu, fuzzer'ın hataları bulmasını çok daha kolay hale getirir. Sözdizimi testlerinden bazı dosyaları kopyalayabilir ya da dokümantasyondan veya diğer testlerden test dosyalarını çıkarabilirsiniz:

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↪SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs
```

AFL dokümantasyonunda corpus'un (ilk girdi dosyaları) çok büyük olmaması gerektiği belirtilmektedir. Dosyaların kendileri 1 kB'den büyük olmamalıdır ve fonksiyonellik başına en fazla bir girdi dosyası olmalıdır, bu nedenle az sayıda dosya ile başlamak daha iyidir. Binary'nin benzer davranışına neden olan girdi dosyalarını kırpabilen `afl-cmin` adlı bir araç da bulunmaktadır.

Şimdi fuzzer'ı çalıştırın (`-m` bellek boyutunu 60 MB'a genişletir):

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

Fuzzer, `/tmp/fuzzer_reports` içinde hatalara yol açan kaynak dosyaları oluşturur. Genellikle aynı hatayı üreten birçok benzer kaynak dosya bulur. Benzersiz hataları filtrelemek için `scripts/uniqueErrors.sh` aracını kullanabilirsiniz.

3.36.6 Whiskers

Whiskers, Mustache benzeri bir dize şablonlama sistemidir. Derleyici tarafından çeşitli yerlerde kodun okunabilirliğine ve dolayısıyla korunabilirliğine ve doğrulanabilirliğine yardımcı olmak için kullanılır.

Sözdizimi Mustache'den önemli bir farkla birlikte gelir. Ayrıştırmaya yardımcı olmak ve *Yul* ile çakışmaları önlemek için {{ ve }}` şablon işaretleyicileri ``< ve > ile değiştirilir (< ve > sembolleri inline assembly'de geçersizdir, { ve } ise blokları sınırlandırmak için kullanılır). Bir başka sınırlama da listelerin yalnızca bir derinlikte çözümlenebilmesi ve özyinelemeye tabi tutulmamasıdır. Bu gelecekte değişebilir.

Kaba bir tanımlama aşağıdaki gibidir:

Herhangi bir <name> oluşumu, herhangi bir kaçış olmadan ve yinelenen değiştirmeler olmadan sağlanan name değişkeninin dize değeri ile değiştirilir. Bir alan <#name>...</name> ile sınırlandırılabilir. Şablon sistemine sağlanan değişken kümeleri kadar içeriğinin bir araya getirilmesiyle değiştirilir ve her seferinde herhangi bir <inner> öğesi ilgili değeriyle değiştirilir. Üst düzey değişkenler de bu tür alanların içinde kullanılabilir.

Ayrıca <?name>...<!name>...</name> biçiminde koşullular da vardır, burada şablon değiştirmeleri name boolean parametresinin değerine bağlı olarak birinci ya da ikinci segmentte özyinelemeli olarak devam eder. Eğer <?+name>...<!+name>...</+name> kullanılırsa, o zaman name string parametresinin boş olup olmadığı kontrol edilir.

3.36.7 Dokümantasyon Stil Rehberi

Aşağıdaki bölümde özellikle Solidity'ye yapılan dokümantasyon katkılarına odaklanan stil önerileri bulacaksınız.

İngilizce Dili

Proje veya marka isimleri kullanmadığınız sürece İngilizce kullanın ve İngiliz İngilizcesi imla kurallarını tercih edin. Yerel argo ve referansların kullanımını azaltmaya çalışın ve dilinizi tüm okuyucular için mümkün olduğunca anlaşılır hale getirin. Aşağıda size yardımcı olacak bazı referanslar verilmiştir:

- Basitleştirilmiş teknik İngilizce
- Uluslararası İngilizce
- İngiliz İngilizcesi yazılışı

Not: Resmi Solidity dokümantasyonu İngilizce olarak yazılmış olsa da, diğer dillerde topluluk katkılı :ref: *translations* mevcuttur. Topluluk çevirilerine nasıl katkıda bulunabileceğiniz hakkında bilgi için lütfen çeviri kılavuzuna bakın.

Başlıklar için Başlık Düzeni

Başlıklar için title case kullanın. Bu, başlıklardaki tüm ana sözcüklerin büyük harfle yazılması, ancak başlığa başlamadıkları sürece artikellerin, bağlaçların ve edatların büyük harfle yazılmaması anlamına gelir.

Örneğin, aşağıdakilerin hepsi doğrudur:

- Başlıklar için Başlık Düzeni.
- Başlıklar İçin Başlık Düzenini Kullanın.
- Yerel ve Eyalet Değişken Adları.
- Düzen Sırası.

Genişletme Kısaltmaları

Örneğin, sözcükler için genişletilmiş kısaltmalar kullanın:

- "Don't" yerine "Do not".
- "Can't" yerine "Can not".

Aktif ve Pasif Ses

Aktif ses, okuyucunun bir görevi kimin veya neyin gerçekleştirdiğini anlamasına yardımcı olduğu için genellikle öğretici tarzı dokümantasyon için önerilir. Ancak, Solidity dokümantasyonu öğretici ve referans içeriklerin bir karışımı olduğundan, pasif ses bazen daha uygundur.

Özetlemek gerekirse:

- Teknik referanslar için pasif ses kullanın, örneğin dil tanımı ve Ethereum VM'nin dahili özellikleri.
- Solidity'nin bir yönünün nasıl uygulanacağına ilişkin önerileri açıklarken aktif ses kullanın.

Örneğin, aşağıdaki metin Solidity'nin bir yönünü belirttiği için pasif seslidir:

> Fonksiyonlar `pure` olarak bildirilebilir, bu takdirde durumdan okuma yapmayacaklarına veya durumu değiştirmeyeceklerine söz verirler.

Örneğin, aşağıda Solidity'nin bir uygulaması tartışılırken aktif ses kullanılmıştır:

> Derleyiciyi çağırırken, bir yolun ilk öğesinin nasıl bulunacağını ve ayrıca yol öneki yeniden eşlemelerini belirtebilirsiniz.

Genel Terimler

- "Fonksiyon parametreleri" ve "dönüş değişkenleri", girdi ve çıktı parametreleri değil.

Kod Örnekleri

Bir CI süreci, bir PR oluşturduğunuzda `./test/cmdlineTests.sh` betiğini kullanarak `pragma solidity`, `contract`, `library` veya `interface` ile başlayan tüm kod bloğu biçimlendirilmiş kod örneklerini test eder. Yeni kod örnekleri ekliyorsanız, PR oluşturmadan önce bunların çalıştığından ve testleri geçtiğinden emin olun.

Tüm kod örneklerinin, sözleşme kodunun geçerli olduğu en geniş alanı kapsayan bir `pragma` sürümü ile başladığından emin olun. Örneğin `pragma solidity >=0.4.0 <0.9.0;`.

Dokümantasyon Testlerini Çalıştırma

Dokümantasyon için gerekli bağımlılıkları yükleyen ve kırık bağlantılar veya sözdizimi sorunları gibi sorunları kontrol eden `./docs/docs.sh` dosyasını çalıştırarak katkılarınızın dokümantasyon testlerimizi geçtiğinizden emin olun.

3.36.8 Solidity Dili Tasarımı

Dil tasarım sürecine aktif olarak dahil olmak ve Solidity'nin geleceği ile ilgili fikirlerinizi paylaşmak için lütfen Solidity forum'a katılın.

Solidity forumu, yeni dil özelliklerinin ve bunların uygulanmasının ilk aşamalarında veya mevcut özelliklerin modifikasyonlarının önerildiği ve tartışıldığı bir yer olarak hizmet vermektedir.

Öneriler daha somut hale gelir gelmez, bunların uygulanması da Solidity GitHub repository'de sorunlar şeklinde tartışılacaktır.

Forum ve sorun tartışmalarına ek olarak, seçilen konuların, sorunların veya özellik uygulamalarının ayrıntılı olarak tartışıldığı dil tasarımı tartışma çağrılarına düzenli olarak ev sahipliği yapıyoruz. Bu çağrılar için davetiye forum üzerinden paylaşılmaktadır.

Ayrıca geri bildirim anketlerini ve dil tasarımıyla ilgili diğer içerikleri de forumda paylaşıyoruz.

Ekibin yeni özelliklerin uygulanması konusunda ne durumda olduğunu öğrenmek istiyorsanız, Solidity Github projesi adresinden uygulama durumunu takip edebilirsiniz. Tasarım birikimindeki konular daha fazla spesifikasyona ihtiyaç duyar ve ya bir dil tasarımı çağrısında ya da normal bir ekip çağrısında tartışılacaktır. Varsayılan branch'ten (*develop*) breaking branch'e geçerek bir sonraki breaking release için gelecek değişiklikleri görebilirsiniz.

Geçici durumlar ve sorularınız için, Solidity derleyicisi ve dil geliştirme ile ilgili konuşmalar için özel bir sohbet odası olan Solidity dev Gitter kanalı üzerinden bize ulaşabilirsiniz.

Dil tasarım sürecini daha işbirlikçi ve şeffaf hale getirmek için neler yapabileceğimiz konusundaki düşüncelerinizi duymaktan mutluluk duyarız.

3.37 Solidity Marka Kılavuzu

Bu marka rehberi, Solidity'nin marka politikası ve logo kullanım yönergeleri hakkında bilgi içerir.

3.37.1 Solidity Markası

Solidity programlama dili, çekirdek bir ekip tarafından yönetilen açık kaynaklı bir topluluk projesidir. Çekirdek ekip Ethereum Foundation tarafından desteklenmektedir.

Bu belge, Solidity marka adı ve logosunun en iyi şekilde nasıl kullanılacağı hakkında bilgi vermeyi amaçlamaktadır.

Marka adını veya logoyu kullanmadan önce bu belgeyi dikkatlice okumanızı öneririz. İşbirliğiniz çok takdir edilmektedir!

3.37.2 Solidity Marka Adı

"Solidity" yalnızca Solidity programlama diline atıfta bulunmak için kullanılmalıdır.

Lütfen "Solidity" i aşağıdaki şekillerde kullanmayın:

- Başka herhangi bir programlama diline atıfta bulunmak için.
- Yanıltıcı olacak veya ilgisiz modüllerin, araçların, belgelerin veya diğer kaynakların Solidity programlama diliyle ilişkilendirildiğini ima edecek şekilde.
- Solidity programlama dilinin açık kaynaklı ve ücretsiz olup olmadığı konusunda topluluğun kafasını karıştıracak şekilde.

3.37.3 Solidity Logo Lisansı

Solidity logosu Creative Commons Attribution 4.0 International License altında dağıtılmakta ve lisanslanmaktadır.

Bu, en serbest Creative Commons lisansıdır ve herhangi bir amaç için yeniden kullanıma ve değişikliklere izin verir.

Bunları yapmakta özgürsünüz:

- **Paylaş** - Materyali herhangi bir ortamda veya formatta kopyalayın ve yeniden dağıtın.
- **Uyarlamak** - Ticari olarak bile olsa herhangi bir amaçla materyali yeniden karıştırmak, dönüştürmek ve üzerine inşa etmek.

Aşağıdaki şartlar altında:

- **Atıf** - Uygun şekilde atıfta bulunmalı, lisansa bir bağlantı vermeli ve değişiklik yapılıp yapılmadığını belirtmelisiniz. Bunu makul herhangi bir şekilde yapabilirsiniz, ancak Solidity çekirdek ekibinin sizi veya kullanımınızı onayladığını gösteren herhangi bir şekilde yapamazsınız.

Solidity logosunu kullanırken lütfen Solidity logo yönergelerine uyun.

3.37.4 Solidity Logo Kılavuzu

(İndirmek için logoya sağ tıklayın.)

Lütfen şunları yapmayınız:

- Logonun oranını değiştirmek (uzatmayın veya kesmeyin).
- Kesinlikle gerekli olmadığı sürece logonun renklerini değiştirmek.

3.37.5 Credits

Bu belge kısmen Python Yazılım Vakfı Ticari Marka Kullanım Politikası ve Rust Media Guide'dan türetilmiştir.

3.38 Language Influences

Solidity is a curly-bracket language that has been influenced and inspired by several well-known programming languages.

Solidity is most profoundly influenced by C++, but also borrowed concepts from languages like Python, JavaScript, and others.

The influence from C++ can be seen in the syntax for variable declarations, for loops, the concept of overloading functions, implicit and explicit type conversions and many other details.

In the early days of the language, Solidity used to be partly influenced by JavaScript. This was due to function-level scoping of variables and the use of the keyword `var`. The JavaScript influence was reduced starting from version 0.4.0. Now, the main remaining similarity to JavaScript is that functions are defined using the keyword `function`. Solidity also supports import syntax and semantics that are similar to those available in JavaScript. Besides those points, Solidity looks like most other curly-bracket languages and has no major JavaScript influence anymore.

Another influence to Solidity was Python. Solidity's modifiers were added trying to model Python's decorators with a much more restricted functionality. Furthermore, multiple inheritance, C3 linearization, and the `super` keyword are taken from Python as well as the general assignment and copy semantics of value and reference types.

Dizin

Semboller

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y