scilla 中文文档

发布 *0.7.1*

Babbage

2021 年 09 月 09 日

Contents

Scilla （即 *Smart Contract Intermediate-Level LAnguage*）是为 Zilliqa 区块链开发的智能合约中级语言。Scilla 是按照原则性语言进行设计的，并且充分考虑了智能合约的安全性。

Scilla 通过在智能合约上强加一种结构，从而可以直接在语言级别消除某些已知漏洞，使应用程序不易被攻击。此外，Scilla 的原则性结构设计将使应用程序从根源上解决安全问题，而且这种设计更易于接受形式验证。

该语言与语义形式化以及将其嵌入 Coq 证明助手 （一种用于对程序属性进行机械化证明的先进工具）进行着并行开发。Coq 是基于高级的依赖类型理论，并以大量的数学库为其特征。在此之前，Coq 已经成功应用于实现认证 (即完全机械验证) 编译器、并发和分布式应用程序，其中就包括区块链等。

Zilliqa 是一个具有可扩展性的，Scilla 合约在其上运行的基础区块链平台。它采用分片的思想进行并行事务验证。Zilliqa 有一个名为 *Zilling* （简称 ZIL）的内置通证，此通证需要在 Zilliqa 上运行智能合约。

CHAPTER 1

发展现状

Scilla 正在积极的研究和开发中，因此本文档中描述的部分规范可能会发生变化。Scilla 目前附带了一个解释器二进制文件，已集成到两个 Scilla 特定的基于 web 的 IDE 中。*Scilla 尝鲜* 展示了这两个 IDE 的特性。

CHAPTER 2

相关资源

有关 Scilla 和 Zilliqa 的学习资料有不少，下面列了一些经常会用到的：

Scilla

- Scilla 设计论文
- Scilla 讲义
- Scilla 语言基础语法
- Scilla 设计背后的故事之为什么我们需要一门新语言

Zilliqa

- Zilliqa 设计背后的故事之网络分片（一）
- Zilliqa 设计背后的故事之共识协议（二）
- Zilliqa 设计背后的故事之共识效率（三）
- 技术白皮书
- Zilliqa 技术答疑

CHAPTER 3

目录

3.1 Scilla 设计原则

智能合约提供了一种表达区块链计算属性的机制，即去中心化的拜占庭容错分布式账本。随着智能合约的出现，构建去中心化应用程序（简称 Dapps）已经成为可能。这些 Dapps 的程序和业务逻辑以智能合约的形式编码，可以在去中心化的区块链网络上运行。

在去中心的网络上运行应用程序消除了对受信任的中心机构或其他应用程序典型的服务器的需求。智能合约的这些特性如今已变得非常流行，它们正在通过众筹、游戏、去中心化交易所、支付处理器等应用程序推动着现实世界的经济。

然而，过去几年的经验表明，智能合约语言的语义话操作实现有着相当微妙的情况，这与合约开发人员头脑中对该语言的直观理解相背离。这种理解上的分歧导致了对智能合约的大量攻击，例如对 DAO 合约和 Parity 钱包的攻击等等。由于区块链的不可篡改特性，智能合约无法直接更新，因此这个问题就变得更加严重。由此可见，确保部署后的智能合约的安全性就显得至关重要。

在其他领域，验证和模型检查等形式方法已被证明在提高软件系统的安全性方面是卓有成效的，因此探索它们在提高智能合约的可读性和安全性方面的适用性就是很自然的事情了。此外，通过形式方法，可以对合约的行为作出严格的保证。

然而，对现有语言（如 Solidity）应用形式验证工具并不是一项简单的任务，因为图灵完备语言具有典型的极端表达能力。实际上，在使语言更易于理解和易于接受形式验证以及使其更具表达性之间存在一种取舍。例如，比特币的脚本语言选择了更简单的处理方式，即不处理状态对象。而在表达性方面则更倾向于图灵完备语言，如 Solidity。

Scilla 是一种全新的（中级）智能合约语言，设计的目的是同时实现表达性和可溯性，同时通过采用一些基本设计原则来实现合约行为的形式推理，具体如下所示:

计算和通信分离

Scilla 中的合约被构造为一种自动通信机制：合约中的每个计算（例如，改变其平衡或计算一个函数的值）都被实现为独立的原子跃迁，也就是说，这种计算不涉及任何第三方。无论何时需要参与（例如，将控制权转移给另一方），这种转换都将通过发送和接收消息的形式以明确的通信方式结束。基于自动化的结构使得从区块链范围的交互（即发送/接收资金和消息）中分离出特定于合约的影响（即转换）成为可能，从而提供了一个关于合约构成和不可变量的清晰推理机制。

有效计算和纯计算分离

任何发生在 transition 内的合约计算都必须终止，并对合约的状态和执行产生可预测的影响。为了实现这一点，Scilla 从函数式编程中汲取灵感，在区分纯表达式（例如，具有原始数据类型和映射的表达式）、非纯局部状态操作（例如，读/写合约字段）和区块链映射（例如，读取当前块高度）方面都有显著提高。通过精心设计纯语言和非纯语言之间的交互语义，Scilla 保留了许多关于合约转换的基本属性，如进程和类型保存，同时也使它们能够通过独立工具进行交互和/或自动验证。

Invocation 和 Continuation 分离

作为自动通信构造合约，它提供了一个只允许尾部调用计算模型，也就是说，每个对外部函数（也就是另一个合约）的调用都必须作为最后一条指令来执行，而且要严格遵守。

3.2 Scilla 尝鲜

Scilla 目前还正在积极开发中。但是你可以使用线上 IDE 来编写 Scilla。

3.2.1 Savant IDE

Neo Savant IDE 是一个基于 Web 的开发环境，它允许你与模拟测试网环境、开发者测试网和主网进行交互。另外它针对在 Chrome 浏览器中的使用进行了优化。Neo Savant IDE 允许你从外部钱包（如 Ledger 或密钥库文件）导入帐户。

当钱包成功导入时，IDE 会自动请求水龙头向你支付测试网 $ZIL。在模拟的测试网环境中，你将获得 10,000 枚 ZIL。在开发者测试网上，你将获得 300 枚 ZIL。主网没有水龙头。

在使用 隔离服务器 和 Zilliqa-JS 等工具进行自动化脚本测试之前，Neo Savant IDE 可以充当临时环境。要试用 Neo Savant IDE，用户需要访问 Neo Savant IDE 。

3.2.2 示例合约

Savant IDE 附带以下用 Scilla 编写的示例智能合约：

- **HelloWorld** ： 这是一个简单的合约，允许指定的帐户 `owner` 设置欢迎消息。设置欢迎消息是通过 `setHello (msg: String)` 完成的。该合约还提供了一个接口 `getHello()` 以允许任何帐户在调用时返回欢迎消息。
- **BookStore**： CRUD 应用程序的演示。只有合约的 `owner` 才能添加 `members` 。所有 `members` 都将具有读/写权限，可以使用 *book title* 、*author* 和 *bookID* 对库存中的书籍进行 OR 操作。
- **CrowdFunding**： Crowdfunding 实现了一个 Kickstarter-style 的众筹活动，用户可以使用 `Donate()` 向合约捐赠资金。如果众筹成功，即在给定时间段内筹集到足够的资金，则可以通过 `GetFunds()` 将筹集的资金发送给预定义的帐户 `owner` 。相反，如果众筹失败，则贡献者可以通过名为 `ClaimBack()` 的 ransition 收回他们的捐款。
- **Auction**： 一个简单的公开拍卖合约，投标人可以使用 `Bid()` 进行投标，最高中标金额进入预定帐户。没有获胜的投标人可以使用名为 `Withdraw()` 的 transition 收回他们的投标。拍卖的组织者可以通过调用名为 `AuctionEnd()` 的 transition 来获取最高出价。
- **FungibleToken**： ZRC-2 同质化通证标准合约，用于创建可替代的数字资产，例如稳定币、实用通证和忠诚度积分。
- **NonFungible Token**： ZRC-1 非同质化代币（即 NFT）标准合约，用于创建独特的数字资产，例如数字收藏品、音乐唱片、艺术和域名。
- **ZilGame**： 一个两人游戏，目标是找到与给定 SHA256 摘要（`puzzle` ）最接近的原像。更具体地说，就是对于一些 *Distance* 函数，给定摘要 *d* 和两个值 *x* 和 *y*，如果 Distance(SHA-256(x), d) < Distance(SHA-256(y), d)，与 *d* 相比，则称 *x* 比 *y* 更接近原像。游戏分两个阶段进行：在第一阶段，玩家使用名为 `Play(guess: ByStr32)` 的 transition 提交他们的哈希值，即 SHA-256(x) 和 SHA-256(y)。一旦第一个玩家提交了她的哈希值，第二个玩家则必须在限定时间内来提交她的哈希值。如果第二位玩家没有在规定时间内提交她的哈希，那么第一位玩家就可能成为获胜者；在第二阶段，玩家必须使用名为 `ClaimReward(solution: Int128)` 的 transition 来提交相应的值 `x` 或 `y` ，提交的哈希值最接近原像的玩家被宣布为获胜者并获得奖励。该合约还提供了一个名为 `Withdraw ()` 的 transition 以在没有玩家玩游戏的情况下收回资金并发送给指定的 `owner` 。
- **SchnorrTest**： 一个用于测试 Schnorr 公钥/私钥对的生成、使用私钥对 `msg` 进行签名以及验证签名的示例合约。
- **ECDSATest**： 一个用于测试 ECDSA 公钥/私钥对的生成、使用私钥对消息进行签名以及验证签名的示例合约。

3.3 Scilla 实战演练

3.3.1 HelloWorld

首先，我们来编写具有以下规范的经典 HelloWorld.scilla 合约：

- 此合约中有一个不可变的合约参数 owner ，由合约的创建者初始化。该参数是不可变的，即意味着一旦在合约部署期间进行初始化，其值就无法更改。其中 owner 是 ByStr20 类型（表示 20 字节地址的十六进制字节字符串）。
- 此合约中有一个 String 类型的可变字段 welcome_msg 且被初始化为 "" 。此处的可变性是指，即使在部署合约后也可以修改变量的值。
- owner 且 **只有她**才能够通过接口 setHello 修改 welcome_msg 。该接口将 msg （String 类型）作为输入，并允许 owner 将 welcome_msg 的值设置为 msg 。
- 此合约有一个接口 getHello ，允许任何带有 welcome_msg 的调用者调用。此外 getHello 不会接受任何输入。

定义合约、不可变参数和可变字段

我们将使用关键字 contract 来作为声明一个合约的开始。关键字后跟合约的名称，而在我们的示例中就是 HelloWorld 。综上，以下代码片段声明了一个 HelloWorld 合约。

```
contract HelloWorld
```

注解： 在目前的实现中，一个 Scilla 合约只能包含一个单一的合约声明，因此任何跟在 contract 关键字之后的代码都是合约声明的一部分。换句话说，没有明确的关键字来声明合约定义的结束。

声明合约后，接下来是其不可变参数的声明，其代码块由 () 包裹。每个不可变参数都通过以下方式声明：vname: vtype ，其中 vname 是参数名称，vtype 是参数类型。不可变参数由 , 分隔。根据规范，合约将只有一个 ByStr20 类型的不可变参数 owner ，具体参考以下代码片段。

```
(owner: ByStr20)
```

合约中的可变字段通过关键字 field 声明。每个可变字段以下列方式声明：field vname : vtype = init_val ，其中 vname 是字段名称，vtype 是它的类型，init_val 是字段必须被初始化的值。HelloWorld 合约有一个 String 类型的可变字段 welcome_msg ，且被初始化为 "" 。具体实现参考以下代码片段：

```
field welcome_msg : String = ""
```

在此阶段，我们的 HelloWorld.scilla 合约将具有以下形式，其中包括合约名称、不可变参数和可变字段：

```
contract HelloWorld
(owner: ByStr20)

field welcome_msg : String = ""
```

定义接口即 Transitions

像 `setHello` 这样的接口在 Scilla 中被称为 *transitions* 。Transitions 类似于其他语言中的函数或方法。然而，他们有一个重要的区别，大多数语言允许它们的函数或方法被并行运行的线程”中断”，但 Scilla 不会让 transitions 被中断，以此来确保不会出现所谓的重入问题。

注解: 术语 *transition* 来自 Scilla 中的底层计算模型，它遵循通信自动机。Scilla 中的合约是具有某种状态的自动机。自动机的状态可以使用 *transition* 来更改，该 *transition* 采用先前状态和输入并产生新的状态。访问 维基百科条目 以阅读有关 *transition* 系统的更多信息。

使用关键字 `transition` 声明一个 *transition* 。使用关键字 `end` 声明 *transition* 代码块的结束。`transition` 关键字后跟 *transition* 名称，在我们的示例中为 `setHello` 。然后是 `()` 内的输入参数。每个输入参数由 `,` 分隔，并以以下格式声明：`vname : vtype` 。根据规范，`setHello` 只接受一个 `String` 类型的名为 `msg` 的参数。具体参照以下代码片段：

```
transition setHello (msg : String)
```

transition 声明之后是 transition 的主体部分。下面给出了第一个 transition 的代码段，此代码段为调用 `setHello (msg : String)` 以设置 `welcome_msg`：

```
1  transition setHello (msg : String)
2    is_owner = builtin eq owner _sender;
3    match is_owner with
4    | False =>
5      e = {_eventname : "setHello"; code : not_owner_code};
6      event e
7    | True =>
8      welcome_msg := msg;
9      e = {_eventname : "setHello"; code : set_hello_code};
10     event e
11   end
12 end
```

首先，使用第 2 行中的指令 `builtin eq owner _sender` 来检查 transition 的调用者是否与 `owner` 相等。为了比较两个地址，我们使用定义为函数 `eq` 的 `builtin` 运算符。该运算符返回布尔值 `True` 或 `False` 。

注解: Scilla 内部定义了一些具有特殊语义的变量。这些特殊变量通常以 _ 为前缀。例如，Scilla 中的 `_sender`

是指调用当前合约的帐户地址。

根据比较的结果，transition 通过不同的分支来进行模式匹配，其语法如下面的代码片段所示。

```
match expr with
| pattern_1 => expr_1
| pattern_2 => expr_2
end
```

上面的代码检查 expr 的计算结果是否与 pattern_1 或 pattern_2 匹配。如果 expr 的计算结果与 pattern_1 匹配，那么接下来要执行的表达式将是 expr_1 。否则，如果 expr 的计算结果与 pattern_2 匹配，那么接下来要执行的表达式就是 expr_2 。

因此，以下代码块实现了类似于 if-then-else 的指令：

```
match expr with
| True  => expr_1
| False => expr_2
end
```

调用者不是所有者

如果调用者与 owner 不同，则 transition 会进入 False 分支，即合约使用指令 event 来触发一个事件。

事件是指存储在区块链上供所有人查看的信号。如果用户使用客户端应用程序调用合约的 transition，客户端应用程序就可以监听合约可能触发的事件，并提醒用户。

更具体地说，这种情况下的事件代码如下：

```
e = {_eventname : "setHello"; code : not_owner_code};
```

一个事件由许多 vname : value 对组成，并由 ; 分隔。在一对花括号 {} 内。一个事件必须包含必填字段 _eventname ，同时也可以包含其他字段，例如上面示例中的 code 字段。

注解： 在我们的示例中，我们选择使用 transition 的名称来命名事件的名称，但时你可以选择任何其他名称。不过呢，还是建议你以易读性为标准来命名，即使用执行事件的那部分代码的名称来命名事件。

调用者是所有者

如果调用者是 owner ，则合约允许调用者将可变字段 welcome_msg 的值设置为输入参数 msg 。这些是通过以下指令完成的：

```
welcome_msg := msg;
```

注解： 给可变字段赋值是使用运算符 := 完成的。

和前面的例子一样，合约随后会触发一个 code 值为 set_hello_code 的事件。

库定义

Scilla 合约会有一些辅助库，它们声明合约的纯功能组件，即没有状态操作的组件。使用关键字 library 然后跟上库名称，就完成了在合约的开始部分中库的声明。在我们当前的示例中，库声明如下所示：

```
library HelloWorld
```

该库可能包含使用 let ident = expr 定义的效用函数和程序常量。在我们的示例中，库定义的部分只包含错误代码的定义：

```
let not_owner_code  = Uint32 1
let set_hello_code  = Uint32 2
```

到这里为止，我们的合约片段会变成下面这个样子：

```
library HelloWorld

 let not_owner_code  = Uint32 1
 let set_hello_code  = Uint32 2


 contract HelloWorld
 (owner: ByStr20)

 field welcome_msg : String = ""

 transition setHello (msg : String)
   is_owner = builtin eq owner _sender;
   match is_owner with
   | False =>
     e = {_eventname : "setHello"; code : not_owner_code};
```

(下页继续)

(续上页)

```
    event e
  | True =>
    welcome_msg := msg;
    e = {_eventname : "setHello"; code : set_hello_code};
    event e
  end
end
```

增加另一个 Transition

我们现在可以增加第二个 transition `getHello()`，它允许客户端应用程序知道 `welcome_msg` 是什么。声明与 `setHello (msg : String)` 类似，只是 `getHello()` 不带参数。

```
transition getHello ()
    r <- welcome_msg;
    e = {_eventname: "getHello"; msg: r};
    event e
end
```

注解: 读取到本地可变字段（即将当前合约中定义的字段读取到本地）是使用运算符 `<-` 完成的。

在 transition `getHello()` 中，我们将首先从可变字段中读取，然后构造并触发事件。

Scilla 版本号

合约一旦在网络上部署，就无法更改。因此需要指定合约是用哪个版本的 Scilla 编写的，以确保即使对 Scilla 规范进行更改，合约的行为也不会改变。

合约的 Scilla 版本是使用关键字 `scilla_version` 声明的：

```
scilla_version 0
```

版本声明必须出现在任何库或合约代码之前。

代码完整示例

下面给出了实现所需规范的完整合约，其中我们使用 (* *) 结构添来添加注释：

```
(* HelloWorld contract *)

(***************************************************)
(*                 Scilla version                  *)
(***************************************************)

scilla_version 0

(***************************************************)
(*               Associated library                *)
(***************************************************)
library HelloWorld

let not_owner_code  = Uint32 1
let set_hello_code  = Uint32 2

(***************************************************)
(*             The contract definition             *)
(***************************************************)

contract HelloWorld
(owner: ByStr20)

field welcome_msg : String = ""

transition setHello (msg : String)
  is_owner = builtin eq owner _sender;
  match is_owner with
  | False =>
    e = {_eventname : "setHello"; code : not_owner_code};
    event e
  | True =>
    welcome_msg := msg;
    e = {_eventname : "setHello"; code : set_hello_code};
    event e
  end
end

transition getHello ()
  r <- welcome_msg;
  e = {_eventname: "getHello"; msg: r};
```

(下页继续)

(续上页)

```
  event e
end
```

3.3.2 示例二：众筹

在本节中，我们展示了一个运行众筹活动的稍微复杂的合约。在众筹活动中，项目所有者希望通过社区捐款筹集资金。

假设所有者（owner）希望一直运行该活动，直到在区块链（max_block）上达到某个预定的区块高度。所有者还希望提高最低数量的 QA（goal），否则项目将无法启动。因此，合约具有三个不可变参数 owner、max_block 和 goal。

不可变参数会在部署合约时提供。在这一点上，我们希望添加一个合理性检查，其中 goal 必须是一个严格意义上的正数。如果合约中的 goal 意外地以 0 值进行初始化，则该合约就不会被部署。

到目前为止，捐赠给该活动的总金额存储在字段 _balance 中。Scilla 中的任何合约都有一个隐式的 _balance 字段，类型为 Uint128，在合同部署时初始化为 0，它包含了区块链上合约账户中 QA 的数量。

如果所有者能在规定的时间内达到目标，则视为众筹成功。如果众筹失败，捐款将返还给在竞选期间捐款的项目支持者。并且需要支持者有明确要求退款的操作。

合约包含两个可变字段：

- backers：从贡献者地址（类型为 ByStr20 的值）到贡献金额的字段映射，用类型为 Uint128 的值表示。由于最初没有支持者，因此此映射被初始化为"Emp"（即空）映射。该映射使合约能够注册捐赠者，以此防止重复捐赠并在活动失败时可以退还资金。
- funded：一个布尔值的标志，初始化为 False，指所有者是否已经在活动结束后转移了资金。

该合约包含三个 transitions：Donate () 允许任何人为众筹活动捐款，GetFunds () **仅允许所有者 ** 提取捐赠金额并将其转账给 owner，以及 ClaimBack() 允许贡献者在活动不成功时取回他们的捐款。

合约参数的合理性检查

为了确保 goal 是一个严格的正数，我们使用了一个合约约束：

```
with
  let zero = Uint128 0 in
  builtin lt zero goal
=>
```

上面 with 和 => 之间的布尔表达式在合约部署期间进行判断，只有判断结果为 True 时才会部署合约。这确保了合约不会被错误地将 goal 的值以 0 的进行部署。

读取当前区块高度

截止日期以区块高度给出，因此要检查是否已过截止日期，我们必须将截止日期与当前区块高度进行比较。

当前块高度读取如下：

```
blk <- & BLOCKNUMBER;
```

区块高度在 Scilla 中有一个专用类型 `BNum`，以免将它们与常规无符号整数混淆。

注解： 从区块链读取数据是使用运算符 `<- &` 完成的。且区块链数据不能直接从合约中更新。

读取以及更新当前余额

在部署合约时，活动的目标由所有者在不可变参数 `goal` 中指定。要检查是否达到目标，我们必须将筹集的总额与目标进行比较。

QA 筹集的金额存储在区块链上的合约账户中，可以通过隐式声明的 `_balance` 字段访问，如下所示：

```
bal <- _balance;
```

Money 表示为 `Uint128` 类型的值。

注解： 像任何其他合约字段一样，`_balance` 字段是使用运算符 `<-` 读取的。但是，`_balance` 字段只能通过接受来自传入 messages 的资金（使用指令 `accept`）或通过明确地将资金转移到其他帐户（使用如下所述的 `send` 指令）来更新。

发送 Messages

在 Scilla 中，transitions 可以通过两种方式传输数据。一种方法是通过 events，如前一个示例中所述。另一种是通过使用指令 `send` 发送 messages。

`send` 用于向其他帐户发送 messages，以便调用另一个智能合约上的 transitions，或者将资金转移到用户帐户。另一方面，events 是智能合约可以用来将数据传输到客户端应用程序的分派信号。

要定义 message，我们可以使用与定义 events 时类似的语法：

```
msg = {_tag : ""; _recipient : owner; _amount : bal; code : got_funds_code};
```

message 必须包含必填的 message 字段 `_tag`、`_recipient` 和 `_amount`。`_recipient` 字段是消息要发送到的区块链地址（类型为 `ByStr20`），`_amount` 字段是要转移到该帐户的 QA 数量。

_tag 字段的值表示的是要部署在 _recipient 地址的合约调用的 transition（String 类型）的名称。如果 _recipient 是用户帐户的地址，则忽略 _tag 的值，因此为简单起见，我们这里定义为空，即 "" 。

注解： 为了能够同时退还合约和用户帐户（对于使用钱包合约进行捐赠的支持者非常有用），请使用 ZRC-5 中的标准 transition 名称，如 AddFunds 。

除了必填字段之外，message 还可以包含其他字段，例如上面的代码所示。但是，如果 message 的接收者是合约，则附加字段的名称和类型必须与在接收者合约上调用的 transition 参数相同。

发送 message 是使用 send 指令完成的，该指令将 messages 列表作为参数。由于我们在众筹合约中一次只会发送一条 message，因此我们定义了一个库函数 one_msg 来构造一个由 message 组成的列表：

```
let one_msg =
  fun (msg : Message) =>
  let nil_msg = Nil {Message} in
    Cons {Message} msg nil_msg
```

要发送 message，我们首先构造 message，并将其插入列表，然后发送：

```
msg = {_tag : ""; _recipient : owner; _amount : bal; code : got_funds_code};
msgs = one_msg msg;
send msgs
```

Procedures

Scilla 合约的 transitions 通常需要执行相同的小指令序列。为了防止代码重复，合约可以定义许多 procedures，这些 procedures 可以通过合约的 transitions 调用。procedures 还有助于将合约代码分成独立的、自包含的部分，这些部分更容易阅读和单独推理。

使用关键字 procedure 来声明 procedure。procedure 的结束使用关键字 end 声明。procedure 关键字后面是 transition 名称，然后是 () 中的输入参数，然后是 procedure 的语句。

在我们的示例中，名为 Donate 的 transition 将在三种情况下触发一个 event： 如果捐赠在截止日期之后发生，则触发一个错误 event；如果捐赠者以前捐赠过，则触发另一个错误 event；由于大部分 event 触发代码是相同的，我们决定定义一个名为 DonationEvent 的 procedure，负责发出正确的 event：

```
procedure DonationEvent (failure : Bool, error_code : Int32)
  match failure with
  | False =>
    e = {_eventname : "DonationSuccess"; donor : _sender;
         amount : _amount; code : accepted_code};
    event e
  | True =>
    e = {_eventname : "DonationFailure"; donor : _sender;
```

(下页继续)

(续上页)

```
        amount : _amount; code : error_code};
    event e
  end
end
```

该 procedure 接受两个参数：一个 Bool 类型，代表捐赠是否失败；一个代表当失败发生时，失败类型的错误代码。

procedure 会对参数 failure 进行 match 。如果捐赠没有失败，错误代码将被忽略，并触发一个名为 DonationSuccess 的 event。否则，如果捐赠失败，则触发一个名为 DonationFailure 的 event，该 event 带有作为 procedure 的第二个参数进行传递的错误代码。

下面的代码展示了如何使用参数 True 和 0 调用名为 DonationEvent 的 procedure：

```
c = True;
err_code = Int32 0;
DonationEvent c err_code;
```

注解： 特殊参数 _sender 、_origin 和 _amount 对 procedure 是可用的，即使 procedure 是由 transition 而不是由传入消息调用的。没有必要将这些特殊参数作为实参传递给 procedure。

注解： procedures 与库函数相似，它们可以从任何 transition 中调用 (只要 transition 是在 procedures 之后定义的)。但是两者也有不同之处，标准库函数不能访问合约状态，procedures 不能有返回值。

procedures 与 transitions 有点类似，因为它们都可以访问和更改合约状态，以及读取传入消息和发送传出消息。但是，procedures 不能在链上调用，只能从合约外部调用 transitions，因此 procedures 可以被视为私有 transitions。

代码完整示例

完整的众筹合约代码如下

```
(***************************************************)
(*                  Scilla version                 *)
(***************************************************)

scilla_version 0

(***************************************************)
(*               Associated library                *)
(***************************************************)
```

(下页继续)

(续上页)

```
import BoolUtils

library Crowdfunding

let one_msg =
  fun (msg : Message) =>
    let nil_msg = Nil {Message} in
    Cons {Message} msg nil_msg

let blk_leq =
  fun (blk1 : BNum) =>
  fun (blk2 : BNum) =>
    let bc1 = builtin blt blk1 blk2 in
    let bc2 = builtin eq blk1 blk2 in
    orb bc1 bc2

let get_funds_allowed =
  fun (cur_block : BNum) =>
  fun (max_block : BNum) =>
  fun (balance : Uint128) =>
  fun (goal : Uint128) =>
    let in_time = blk_leq cur_block max_block in
    let deadline_passed = negb in_time in
    let target_not_reached = builtin lt balance goal in
    let target_reached = negb target_not_reached in
    andb deadline_passed target_reached

let claimback_allowed =
  fun (balance : Uint128) =>
  fun (goal : Uint128) =>
  fun (already_funded : Bool) =>
    let target_not_reached = builtin lt balance goal in
    let not_already_funded = negb already_funded in
    andb target_not_reached not_already_funded

let accepted_code = Int32 1
let missed_deadline_code = Int32 2
let already_backed_code  = Int32 3
let not_owner_code  = Int32 4
let too_early_code  = Int32 5
let got_funds_code  = Int32 6
let cannot_get_funds  = Int32 7
let cannot_reclaim_code = Int32 8
```

(下页继续)

(续上页)

```
let reclaimed_code = Int32 9

(***************************************************)
(*             The contract definition             *)
(***************************************************)
contract Crowdfunding

(*  Parameters *)
(owner     : ByStr20,
max_block : BNum,
goal      : Uint128)

(* Contract constraint *)
with
  let zero = Uint128 0 in
  builtin lt zero goal
=>

(* Mutable fields *)
field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
field funded : Bool = False

procedure DonationEvent (failure : Bool, error_code : Int32)
  match failure with
  | False =>
    e = {_eventname : "DonationSuccess"; donor : _sender;
         amount : _amount; code : accepted_code};
    event e
  | True =>
    e = {_eventname : "DonationFailure"; donor : _sender;
         amount : _amount; code : error_code};
    event e
  end
end

procedure PerformDonate ()
  c <- exists backers[_sender];
  match c with
  | False =>
    accept;
    backers[_sender] := _amount;
    DonationEvent c accepted_code
  | True =>
```

(下页继续)

(续上页)

```
    DonationEvent c already_backed_code
  end
end

transition Donate ()
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    PerformDonate
  | False =>
    t = True;
    DonationEvent t missed_deadline_code
  end
end

procedure GetFundsFailure (error_code : Int32)
  e = {_eventname : "GetFundsFailure"; caller : _sender;
       amount : _amount; code : error_code};
  event e
end

procedure PerformGetFunds ()
  bal <- _balance;
  tt = True;
  funded := tt;
  msg = {_tag : ""; _recipient : owner; _amount : bal; code : got_funds_code};
  msgs = one_msg msg;
  send msgs
end

transition GetFunds ()
  is_owner = builtin eq owner _sender;
  match is_owner with
  | False =>
    GetFundsFailure not_owner_code
  | True =>
    blk <- & BLOCKNUMBER;
    bal <- _balance;
    allowed = get_funds_allowed blk max_block bal goal;
    match allowed with
    | False =>
      GetFundsFailure cannot_get_funds
```

(下页继续)

(续上页)

```
    | True =>
      PerformGetFunds
    end
  end
end

procedure ClaimBackFailure (error_code : Int32)
  e = {_eventname : "ClaimBackFailure"; caller : _sender;
       amount : _amount; code : error_code};
  event e
end

procedure PerformClaimBack (amount : Uint128)
  delete backers[_sender];
  msg = {_tag : ""; _recipient : _sender; _amount : amount; code : reclaimed_code};
  msgs = one_msg msg;
  e = { _eventname : "ClaimBackSuccess"; caller : _sender; amount : amount; code :␣
↪reclaimed_code};
  event e;
  send msgs
end

transition ClaimBack ()
  blk <- & BLOCKNUMBER;
  after_deadline = builtin blt max_block blk;
  match after_deadline with
  | False =>
    ClaimBackFailure too_early_code
  | True =>
    bal <- _balance;
    f <- funded;
    allowed = claimback_allowed bal goal f;
    match allowed with
    | False =>
      ClaimBackFailure cannot_reclaim_code
    | True =>
      res <- backers[_sender];
      match res with
      | None =>
        (* Sender has not donated *)
        ClaimBackFailure cannot_reclaim_code
      | Some v =>
        PerformClaimBack v
```

(下页继续)

(续上页)

```
      end
    end
  end
end
```

3.3.3 示例三：代币交易所

在第三个例子中，我们来看一下用 Scilla 编写的合约是如何通过相互传递消息和读取彼此的状态来进行交互的。作为我们的示例应用程序，我们选择了一个简化的代币转换合约，其中用户可以将一种类型的同质化代币交换为另一种类型。

同质化代币

回想一下，同质化代币是指与其他相同类型的代币难以区分的代币。例如，1 美元钞票与任何其他 1 美元钞票没有区别 (至少在用于支付商品、服务或其他代币时没有区别)。

Zilliqa 合约参考库 提供了常用合约类型的规范和参考实现，ZRC2 标准为同质化代币指定了一个标准，我们将在本例中使用它。我们不会详细讨论代币合约是如何工作的，只是指出实现代币转换所需的几个重要方面。

交易规范

我们希望我们的简单转换支持以下功能：

- 该交易所有许多上市的代币，这些代币可以彼此自由转换。每个列出的代币都由其代币代简称标识 (例如，”USD”表示美元)。
- 交易所应该始终有一个管理员。管理员负责批准代币合约，并将其在交易所上市。管理员可以将管理员角色传递给其他人。
- 任何用户都可以在交易所下单。为了下单，用户指定他想要出售哪个代币，他提供多少代币，他想购买哪个代币，以及他想要多少代币作为回报。该合约跟踪每一个活跃的（不匹配的）订单。
- 当用户试图下单出售一些代币时，交易所会检查用户是否确实有这些代币要出售。如果他确实有，那么交易所就会索取这些代币并持有它们，直到订单匹配为止。
- 任何用户都可以匹配交易所的活跃订单。为了匹配到订单，用户必须明确指定要匹配的订单。
- 当用户试图匹配订单时，交易所会检查该用户是否确实拥有订单发布者想要购买的代币。如果他确实有，那么交易所就会将订单下发时所要求的代币转移到订单匹配器，并将下单者想要从订单匹配器购买的代币转移给他。在代币被转移之后，交易所会删除已完成的订单。

为了使示例简洁，我们的交易所将不支持下架代币、取消订单、限时订单、订单排序以便订单匹配器获得尽可能好的交易、订单的部分匹配、确保交易所不被滥用、交易手续费等。我们鼓励读者通过实现未实现的功能来进一步熟悉 Scilla。

管理员角色

交易所必须始终有管理员，包括首次部署时。管理员可能会随着时间的推移而改变，因此我们定义了一个可变字段 admin 来跟踪当前管理员，并将其初始化为 initial_admin，它作为一个不可变参数给出：

```
contract SimpleExchange
(
  initial_admin : ByStr20 with end
)

field admin : ByStr20 with end = initial_admin
```

admin 字段的类型是 ByStr20 with end，这是一种地址类型。正如在前面的例子中，ByStr20 是长度为 20 的字符串类型，但我们现在要添加额外的要求，当该字符串被解释为网络上的地址时，该地址必须在使用中，并且该地址处的内容必须满足 with 和 end 关键字之间的任何内容。

在这种情况下，with 和 end 之间没有任何内容，因此我们没有额外的要求。但是，该地址必须由用户或其他合约使用 - 否则 Scilla 将不接受其具有合法地址类型。(当交易所与列出的代币合约交互时，我们将更详细地介绍地址类型。)

有多个 transition 需要检查 _sender 是否为当前的 admin，因此让我们定义一个 procedure 来检查这种情况：

```
procedure CheckSenderIsAdmin()
  current_admin <- admin;
  is_admin = builtin eq _sender current_admin;
  match is_admin with
  | True =>  (* Nothing to do *)
  | False =>
    (* Construct an exception object and throw it *)
    e = { _exception : "SenderIsNotAdmin" };
    throw e
  end
end
```

如果 _sender 是当前管理员，则什么也不会发生，调用此 procedure 的任何 transition 都可以继续。但是，如果 "_sender" 是其他人，则该 procedure 会引发异常，从而导致当前事务中止。

我们希望管理员能够将管理员角色传递给其他人，因此我们定义我们的第一个 transition SetAdmin，如下：

```
transition SetAdmin(new_admin : ByStr20 with end)
  (* Only the former admin may appoint a new admin *)
  CheckSenderIsAdmin;
  admin := new_admin
end
```

transition 会应用 procedure CheckSenderIsAdmin，如果没有抛出异常，则发送者确实是当前管理员，因此允许将管理员角色传递给其他人。新管理员必须同样是正在使用的地址。

Intermezzo：代表代币所有者转移代币

在我们继续向我们的交易所添加功能之前，我们必须首先了解代币合约如何在用户之间转移代币。

ZRC2 代币标准定义了一个字段 `balances` 来跟踪每个用户拥有多少代币：

```
field balances: Map ByStr20 Uint128
```

然而，这对我们的交易所并不是特别有用，因为代币合约不允许交易所转移属于交易所本身以外的其他人的代币。

相反的，ZRC2 标准定义了另一个字段 `allowances`，拥有令牌的用户可以使用它来允许另一个用户部分访问所有者的令牌：

```
field allowances: Map ByStr20 (Map ByStr20 Uint128)
```

例如，如果 Alice 给了 Bob 100 个代币的配额，那么代币合约中的 `allowances` 映射将包含值 `allowances[<address of Alice>][<address of Bob>] = 100`。这会允许 Bob 使用这 100 个 Alice 的代币，就像使用他自己的一样。（Alice 当然可以提取配额，只要 Bob 还没有花掉这些代币）。

在用户下订单之前，用户应该向交易所提供他想要出售的代币额度以支付订单。然后用户才可以下订单，同时交易所会检查配额是否足够。然后交易所会将代币转移到自己的账户中持有，直到订单匹配为止。

类似地，在用户匹配订单之前，用户应该向交易所提供下单者想要购买的代币的配额。然后用户才可以匹配订单，同时交易所会检查配额是否足够。然后，交易所将这些代币转移给下订单的用户，并将代币转移给匹配的用户，这些转移的代币就是在下订单时转移给自己的那些。

为了检查用户提供给交易所的当前配额，我们需要在代币地址类型中指定 `allowances` 字段。我们会像下面这样做：

```
ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20 Uint128) end
```

与 `admin` 字段一样，我们要求地址正在使用中。此外，还必须满足 `with` 和 `end` 之间的要求：

- 关键字 `contract` 指定：地址必须由合约使用，而不是由用户使用。
- 关键字 `field` 指定：相关合约必须包含具有指定名称和指定类型的可变字段。

上架新币

交易所跟踪其上市代币，即允许在交易所交易哪些代币。我们通过定义从代币代码（`String`）到代币地址的映射来实现这一点。

```
field listed_tokens :
  Map String (ByStr20 with contract
                           field allowances : Map ByStr20 (Map ByStr20 Uint128)
                       end)
```

(下页继续)

(续上页)

```
  = Emp String (ByStr20 with contract
                                field allowances : Map ByStr20 (Map ByStr20 Uint128)
                           end)
```

只有允许管理员上架新代币，因此我们在这里再次利用 procedure `CheckSenderIsAdmin` 。

此外，我们只想上架与之前上架的代币具有不同代币代码的代币。为此，我们定义了一个 procedure `CheckIsTokenUnlisted` 来检查代币代码是否被定义为 `listed_tokens` 映射中的键。:

```
library SimpleExchangeLib

let false = False

...

contract SimpleExchange (...)

...

procedure ThrowListingStatusException(
  token_code : String,
  expected_status : Bool,
  actual_status : Bool)
  e = { _exception : "UnexpectedListingStatus";
        token_code: token_code;
        expected : expected_status;
        actual : actual_status };
  throw e
end

procedure CheckIsTokenUnlisted(
  token_code : String
  )
  (* Is the token code listed? *)
  token_code_is_listed <- exists listed_tokens[token_code];
  match token_code_is_listed with
  | True =>
    (* Incorrect listing status *)
    ThrowListingStatusException token_code false token_code_is_listed
  | False => (* Nothing to do *)
  end
end
```

这次我们定义了一个辅助 procedure `ThrowListingStatusException` ，它无条件地抛出异常。这将在我们稍后编写用于下订单的 transition 时有用，因为我们需要检查订单中涉及的代币是否已上架。

我们还在合约的库中定义了常量 `false` 。这是因为 Scilla 要求所有值在用于计算之前都必须命名。在库代码中定义常量可以防止我们用常量定义混淆 transition 代码：

```
(* Incorrect listing status *)
false = False; (* We don't want to do it like this *)
ThrowListingStatusException token_code false token_code_is_listed
```

有了帮助 procedure ，我们现在可以定义 transition `ListToken` 了，如下所示：

```
transition ListToken(
  token_code : String,
  new_token : ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20␣
↪Uint128) end
  )
  (* Only the admin may list new tokens.  *)
  CheckSenderIsAdmin;
  (* Only new token codes are allowed.  *)
  CheckIsTokenUnlisted token_code;
  (* Everything is ok. The token can be listed *)
  listed_tokens[token_code] := new_token
end
```

下单

要下订单，用户必须指定他想要出售的代币代码和数量，以及他想要购买的代币代码和数量。如果没有代币代码上架，我们将调用 procedure `ThrowListingStatusException` 过程：

```
transition PlaceOrder(
  token_code_sell : String,
  sell_amount : Uint128,
  token_code_buy: String,
  buy_amount : Uint128
  )
  (* Check that the tokens are listed *)
  token_sell_opt <- listed_tokens[token_code_sell];
  token_buy_opt <- listed_tokens[token_code_buy];
  match token_sell_opt with
  | Some token_sell =>
    match token_buy_opt with
    | Some token_buy =>
      ...
    | None =>
      (* Unlisted token *)
      ThrowListingStatusException token_code_buy true false
```

(下页继续)

(续上页)

```
    end
  | None =>
    (* Unlisted token *)
    ThrowListingStatusException token_code_sell true false
  end
end
```

如果两个代币都上架了，我们必须首先检查用户是否为交易所提供了足够的配额。当另一个用户匹配订单时，我们将需要类似的检查，因此我们定义了一个帮助 procedure `CheckAllowance` 来执行检查：

```
procedure CheckAllowance(
  token : ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20 Uint128)␣
↪end,
  expected : Uint128
  )
  ...
end
```

为了执行检查，我们需要远程读取代币合约中的配额字段。我们对 `_sender` 给交易所的配额感兴趣，其地址由一个特殊的不可变字段 `_this_address` 给出，所以我们想远程读取代币合约中的配额 `allowances[_sender][_this_address]` 的值。

Scilla 中的远程读取是使用运算符 `<- &` 执行的，我们使用 `.` 指定我们要远程读取的合约的符号。因此，远程读取的整个语句如下：

```
actual_opt <-& token.allowances[_sender][_this_address];
```

就像我们执行本地读取映射一样，从远程映射读取的结果是一个可选值。如果对于 `v` 计算结果为 `Some v`，则用户向交易所提供了 `v` 代币的配额，如果结果为 `None`，则用户根本没有提供配额。因此，我们需要对结果进行模式匹配以获得实际的配额：

```
(* Find actual allowance. Use 0 if None is given *)
actual = match actual_opt with
         | Some x => x
         | None => zero
         end;
```

为方便起见，我们再次在合约库中定义常量 `zero = Uint128 0`。

我们现在可以将实际配额与我们期望的配额进行比较，如果实际配额不足，则抛出异常：

```
is_sufficient = uint128_le expected actual;
  match is_sufficient with
  | True => (* Nothing to do *)
  | False =>
```

(下页继续)

(续上页)

```
    ThrowInsufficientAllowanceException token expected actual
  end
```

函数 uint128_le 是一个实用函数，它对 Uint128 类型的值执行小于或等于比较。该函数是在标准库的 IntUtils 部分定义的，所以为了使用该函数，我们必须将 IntUtils 导入合约，这部分定义在 scilla_version 之后，合约库定义之前：

```
scilla_version 0

import IntUtils

library SimpleExchangeLib
...
```

我们还利用辅助 procedure ThrowInsufficientAllowanceException 在配额不足时抛出异常，因此 procedure CheckAllowance 最终如下所示：

```
procedure ThrowInsufficientAllowanceException(
  token : ByStr20,
  expected : Uint128,
  actual : Uint128)
  e = { _exception : "InsufficientAllowance";
        token: token;
        expected : expected;
        actual : actual };
  throw e
end

procedure CheckAllowance(
  token : ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20 Uint128)␣
↪end,
  expected : Uint128
  )
  actual_opt <-& token.allowances[_sender][_this_address];
  (* Find actual allowance. Use 0 if None is given *)
  actual = match actual_opt with
           | Some x => x
           | None => zero
           end;
  is_sufficient = uint128_le expected actual;
  match is_sufficient with
  | True => (* Nothing to do *)
  | False =>
    ThrowInsufficientAllowanceException token expected actual
```

(下页继续)

(续上页)

```
  end
end

transition PlaceOrder(
  token_code_sell : String,
  sell_amount : Uint128,
  token_code_buy: String,
  buy_amount : Uint128
  )
  (* Check that the tokens are listed *)
  token_sell_opt <- listed_tokens[token_code_sell];
  token_buy_opt <- listed_tokens[token_code_buy];
  match token_sell_opt with
  | Some token_sell =>
    match token_buy_opt with
    | Some token_buy =>
      (* Check that the placer has allowed sufficient funds to be accessed *)
      CheckAllowance token_sell sell_amount;
      ...
    | None =>
      (* Unlisted token *)
      ThrowListingStatusException token_code_buy true false
    end
  | None =>
    (* Unlisted token *)
    ThrowListingStatusException token_code_sell true false
  end
end
```

如果用户给了交易所足够的额度，交易所可以向代币合约发送消息，以便从交易所自己的余额中转移代币。我们需要在代币合约上调用的 transition 称为 `TransferFrom`，而不是 `Transfer`，后者从发送者自己的代币余额中转移资金，而不是从发送者允许的其他人余额中转移资金。

由于消息看起来很像订单匹配时我们需要的消息，因此我们使用合约库中的辅助函数生成消息（我们还需要一个新的常量 `true`）：

```
library SimpleExchangeLib

let true = True

...

let one_msg : Message -> List Message =
  fun (msg : Message) =>
```

(下页继续)

(续上页)

```
    let mty = Nil { Message } in
    Cons { Message } msg mty

let mk_transfer_msg : Bool -> ByStr20 -> ByStr20 -> ByStr20 -> Uint128 -> Message =
  fun (transfer_from : Bool) =>
  fun (token_address : ByStr20) =>
  fun (from : ByStr20) =>
  fun (to : ByStr20) =>
  fun (amount : Uint128) =>
    let tag = match transfer_from with
              | True => "TransferFrom"
              | False => "Transfer"
              end
    in
    { _recipient : token_address;
     _tag : tag;
     _amount : Uint128 0;  (* No Zil are transferred, only custom tokens *)
     from : from;
     to : to;
     amount : amount }

let mk_place_order_msg : ByStr20 -> ByStr20 -> ByStr20 -> Uint128 -> List Message =
  fun (token_address : ByStr20) =>
  fun (from : ByStr20) =>
  fun (to : ByStr20) =>
  fun (amount : Uint128) =>
    (* Construct a TransferFrom messsage to transfer from seller's allowance to␣
↪exhange *)
    let msg = mk_transfer_msg true token_address from to amount in
    (* Create a singleton list *)
    one_msg msg

contract SimpleExchange (...)

...

transition PlaceOrder(
  token_code_sell : String,
  sell_amount : Uint128,
  token_code_buy: String,
  buy_amount : Uint128
  )
  (* Check that the tokens are listed *)
```

(下页继续)

(续上页)

```
  token_sell_opt <- listed_tokens[token_code_sell];
  token_buy_opt <- listed_tokens[token_code_buy];
  match token_sell_opt with
  | Some token_sell =>
    match token_buy_opt with
    | Some token_buy =>
      (* Check that the placer has allowed sufficient funds to be accessed *)
      CheckAllowance token_sell sell_amount;
      (* Transfer the sell tokens to the exchange for holding. Construct a␣
↪TransferFrom message to the token contract. *)
      msg = mk_place_order_msg token_sell _sender _this_address sell_amount;
      (* Send message when the transition completes. *)
      send msg;
      ...
    | None =>
      (* Unlisted token *)
      ThrowListingStatusException token_code_buy true false
    end
  | None =>
    (* Unlisted token *)
    ThrowListingStatusException token_code_sell true false
  end
end
```

最后，我们需要存储新订单，以便用户将来可以匹配订单。为此，我们定义了一个新类型 `Order`，它包含最终匹配订单时所需的所有信息：

```
(* Order placer, sell token, sell amount, buy token, buy amount *)
type Order =
| Order of ByStr20
           (ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20␣
↪Uint128) end)
           Uint128
           (ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20␣
↪Uint128) end)
           Uint128
```

`Order` 类型的值由类型构造函数 `Order`、代币地址和要出售的代币数量以及代币地址和要购买的代币数量给出。

我们现在需要一个包含从订单号（类型为 `Uint128`）到 `Order` 的映射的字段，它表示当前活动的订单。此外，我们将需要一种生成唯一订单号的方法，因此我们将定义一个字段来保存要使用的下一个订单号：

```
field active_orders : Map Uint128 Order = Emp Uint128 Order

field next_order_no : Uint128 = zero
```

要添加新订单，我们需要生成一个新订单号，将生成的订单号和新订单存储在 active_orders 映射中，最后增加 next_order_no 字段（使用库常量 one = Uint128 1），以便为生成下一个订单。我们将把它放在一个辅助 procedure AddOrder 中，并在 transition PlaceOrder 中添加一个对该 procedure 的调用：

```
procedure AddOrder(
  order : Order
  )
  (* Get the next order number *)
  order_no <- next_order_no;
  (* Add the order *)
  active_orders[order_no] := order;
  (* Update the next_order_no field *)
  new_order_no = builtin add order_no one;
  next_order_no := new_order_no
end

transition PlaceOrder(
  token_code_sell : String,
  sell_amount : Uint128,
  token_code_buy: String,
  buy_amount : Uint128
  )
  (* Check that the tokens are listed *)
  token_sell_opt <- listed_tokens[token_code_sell];
  token_buy_opt <- listed_tokens[token_code_buy];
  match token_sell_opt with
  | Some token_sell =>
    match token_buy_opt with
    | Some token_buy =>
      (* Check that the placer has allowed sufficient funds to be accessed *)
      CheckAllowance token_sell sell_amount;
      (* Transfer the sell tokens to the exchange for holding. Construct a␣
↪TransferFrom message to the token contract. *)
      msg = mk_place_order_msg token_sell _sender _this_address sell_amount;
      (* Send message when the transition completes. *)
      send msg;
      (* Create order and add to list of active orders  *)
      order = Order _sender token_sell sell_amount token_buy buy_amount;
      AddOrder order
    | None =>
```

(下页继续)

(续上页)

```
      (* Unlisted token *)
      ThrowListingStatusException token_code_buy true false
    end
  | None =>
    (* Unlisted token *)
    ThrowListingStatusException token_code_sell true false
  end
end
```

PlaceOrder 现在已经完成，但还缺少一件事。ZRC2 代币标准规定，当执行 transition TransferFrom 时，代币将消息发送给接收者和 _sender （称为发起者），以此来通知他们传输成功。这些通知称为回调。由于我们的交易所在卖出代币上执行 transition TransferFrom ，并且由于交易所是这些代币的接收者，我们需要指定可以处理这两个回调的 transition ——如果我们不这样做，那么回调将不会被识别，从而导致整个 PlaceOrder 交易失败。

代币合约通知代币转移的接收者，因为此类通知增加了额外的保护措施，以防止将代币转移到无法处理代币所有权的合约的风险。例如，如果有人在本节的第一个示例中将代币转移到 HelloWorld 合约，那么代币将被永久锁定，因为 HelloWorld 合约无法对代币进行任何操作。

我们的交易所只能处理有活跃订单的代币，但原则上没有什么可以阻止用户不下订单就向交易所转账，所以我们需要确保交易所只涉及代币自己发起的转账。因此，我们定义了一个 procedure CheckInitiator ，如果交换涉及它本身没有启动的代币传输，它会抛出异常，并从所有回调 transition 中调用该 procedure：

```
procedure CheckInitiator(
  initiator : ByStr20)
  initiator_is_this = builtin eq initiator _this_address;
  match initiator_is_this with
  | True => (* Do nothing *)
  | False =>
    e = { _exception : "UnexpecedTransfer";
          token_address : _sender;
          initiator : initiator };
    throw e
  end
end

transition RecipientAcceptTransferFrom (
  initiator : ByStr20,
  sender : ByStr20,
  recipient : ByStr20,
  amount : Uint128)
  CheckInitiator initiator
end
```

(下页继续)

(续上页)

```
transition TransferFromSuccessCallBack (
  initiator : ByStr20,
  sender : ByStr20,
  recipient : ByStr20,
  amount : Uint128)
  CheckInitiator initiator
end
```

匹配订单

对于 transition `MatchOrder`，我们可以利用上一节中定义的许多辅助函数和 procedure 。

用户指定他希望匹配的订单。然后我们在 `active_orders` 映射中查找订单号，如果找不到订单则抛出异常：

```
transition MatchOrder(
  order_id : Uint128)
  order <- active_orders[order_id];
  match order with
  | Some (Order order_placer sell_token sell_amount buy_token buy_amount) =>
    ...
  | None =>
    e = { _exception : "UnknownOrder";
          order_id : order_id };
    throw e
  end
end
```

为了匹配订单，匹配者必须提供足够的买入代币额度。这是由我们之前定义的 procedure `CheckAllowance` 检查的，因此我们在这里简单地重用该 procedure：

```
transition MatchOrder(
  order_id : Uint128)
  order <- active_orders[order_id];
  match order with
  | Some (Order order_placer sell_token sell_amount buy_token buy_amount) =>
    (* Check that the placer has allowed sufficient funds to be accessed *)
    CheckAllowance buy_token buy_amount;
    ...
  | None =>
    e = { _exception : "UnknownOrder";
          order_id : order_id };
    throw e
```

(下页继续)

(续上页)

```
  end
end
```

我们现在需要生成两条转账消息：一条消息是买入代币上的 TransferFrom 消息，将匹配者的配额转移给下订单的用户，另一条消息是卖出代币上的 Transfer 消息，将被持有的代币转移到订单匹配器。同样的，我们定义辅助函数来生成消息：

```
library SimpleExchangeLib

...

let two_msgs : Message -> Message -> List Message =
  fun (msg1 : Message) =>
  fun (msg2 : Message) =>
    let first = one_msg msg1 in
    Cons { Message } msg2 first

let mk_make_order_msgs : ByStr20 -> Uint128 -> ByStr20 -> Uint128 ->
                         ByStr20 -> ByStr20 -> ByStr20 -> List Message =
  fun (token_sell_address : ByStr20) =>
  fun (sell_amount : Uint128) =>
  fun (token_buy_address : ByStr20) =>
  fun (buy_amount : Uint128) =>
  fun (this_address : ByStr20) =>
  fun (order_placer : ByStr20) =>
  fun (order_maker : ByStr20) =>
    (* Construct a Transfer messsage to transfer from exchange to maker *)
    let sell_msg = mk_transfer_msg false token_sell_address this_address order_maker␣
↪sell_amount in
    (* Construct a TransferFrom messsage to transfer from maker to placer *)
    let buy_msg = mk_transfer_msg true token_buy_address order_maker order_placer␣
↪buy_amount in
    (* Create a singleton list *)
    two_msgs sell_msg buy_msg

...

contract SimpleExchange (...)

...

transition MatchOrder(
  order_id : Uint128)
```

(下页继续)

(续上页)

```
    order <- active_orders[order_id];
    match order with
    | Some (Order order_placer sell_token sell_amount buy_token buy_amount) =>
      (* Check that the placer has allowed sufficient funds to be accessed *)
      CheckAllowance buy_token buy_amount;
      (* Create the two transfer messages and send them *)
      msgs = mk_make_order_msgs sell_token sell_amount buy_token buy_amount _this_
↪address order_placer _sender;
      send msgs;
      ...
    | None =>
      e = { _exception : "UnknownOrder";
            order_id : order_id };
      throw e
    end
end
```

由于订单现在已经匹配，所以不应再将其列为活动订单，因此我们删除 `active_orders` 中的条目：

```
transition MatchOrder(
  order_id : Uint128)
  order <- active_orders[order_id];
  match order with
  | Some (Order order_placer sell_token sell_amount buy_token buy_amount) =>
    (* Check that the placer has allowed sufficient funds to be accessed *)
    CheckAllowance buy_token buy_amount;
    (* Create the two transfer messages and send them *)
    msgs = mk_make_order_msgs sell_token sell_amount buy_token buy_amount _this_
↪address order_placer _sender;
    send msgs;
    (* Order has now been matched, so remove it *)
    delete active_orders[order_id]
  | None =>
    e = { _exception : "UnknownOrder";
          order_id : order_id };
    throw e
  end
end
```

transition `MatchOrder` 到此就结束了，但我们需要定义一个额外的回调 transition。下订单时，我们执行了 transition `TransferFrom`，但现在我们也执行 transition `Transfer`，这会产生不同的回调：

```
transition TransferSuccessCallBack (
  initiator : ByStr20,
```

(下页继续)

(续上页)

```
  sender : ByStr20,
  recipient : ByStr20,
  amount : Uint128)
  (* The exchange only accepts transfers that it itself has initiated.  *)
  CheckInitiator initiator
end
```

请注意，我们不需要指定处理从 transition `Transfer` 中接收代币的 transition，因为交易所永远不会将自己作为接收者执行 `Transfer` 。通过完全不定义回调 transition，我们还处理了用户以交易所作为接收方进行转账的情况，因为接收方回调在交易所上不会有匹配的 transition，这将导致整个转账交易失败。

代码完整示例

我们现在已经准备好指定整个合约的一切：

```
scilla_version 0

import IntUtils

library SimpleExchangeLib

(* Order placer, sell token, sell amount, buy token, buy amount *)
type Order =
| Order of ByStr20
           (ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20␣
↪Uint128) end)
           Uint128
           (ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20␣
↪Uint128) end)
           Uint128

(* Helper values and functions *)
let true = True
let false = False

let zero = Uint128 0
let one = Uint128 1

let one_msg : Message -> List Message =
  fun (msg : Message) =>
    let mty = Nil { Message } in
    Cons { Message } msg mty
```

(下页继续)

(续上页)

```
let two_msgs : Message -> Message -> List Message =
  fun (msg1 : Message) =>
  fun (msg2 : Message) =>
    let first = one_msg msg1 in
    Cons { Message } msg2 first

let mk_transfer_msg : Bool -> ByStr20 -> ByStr20 -> ByStr20 -> Uint128 -> Message =
  fun (transfer_from : Bool) =>
  fun (token_address : ByStr20) =>
  fun (from : ByStr20) =>
  fun (to : ByStr20) =>
  fun (amount : Uint128) =>
    let tag = match transfer_from with
              | True => "TransferFrom"
              | False => "Transfer"
              end
    in
    { _recipient : token_address;
     _tag : tag;
     _amount : Uint128 0;  (* No Zil are transferred, only custom tokens *)
     from : from;
     to : to;
     amount : amount }

let mk_place_order_msg : ByStr20 -> ByStr20 -> ByStr20 -> Uint128 -> List Message =
  fun (token_address : ByStr20) =>
  fun (from : ByStr20) =>
  fun (to : ByStr20) =>
  fun (amount : Uint128) =>
    (* Construct a TransferFrom messsage to transfer from seller's allowance to␣
↪exhange *)
    let msg = mk_transfer_msg true token_address from to amount in
    (* Create a singleton list *)
    one_msg msg

let mk_make_order_msgs : ByStr20 -> Uint128 -> ByStr20 -> Uint128 ->
                         ByStr20 -> ByStr20 -> ByStr20 -> List Message =
  fun (token_sell_address : ByStr20) =>
  fun (sell_amount : Uint128) =>
  fun (token_buy_address : ByStr20) =>
  fun (buy_amount : Uint128) =>
  fun (this_address : ByStr20) =>
  fun (order_placer : ByStr20) =>
```

(下页继续)

(续上页)

```
  fun (order_maker : ByStr20) =>
    (* Construct a Transfer messsage to transfer from exchange to maker *)
    let sell_msg = mk_transfer_msg false token_sell_address this_address order_maker␣
↪sell_amount in
    (* Construct a TransferFrom messsage to transfer from maker to placer *)
    let buy_msg = mk_transfer_msg true token_buy_address order_maker order_placer buy_
↪amount in
    (* Create a singleton list *)
    two_msgs sell_msg buy_msg


contract SimpleExchange
(
  (* Ensure that the initial admin is an address that is in use *)
  initial_admin : ByStr20 with end
)

(* Active admin. *)
field admin : ByStr20 with end = initial_admin

(* Tokens listed on the exchange. *)
(* We identify the token by its exchange code, and map it to the address     *)
(* of the contract implementing the token. The contract at that address must *)
(* contain an allowances field that we can remote read.                      *)
field listed_tokens :
  Map String (ByStr20 with contract
                             field allowances : Map ByStr20 (Map ByStr20 Uint128)
                       end)
  = Emp String (ByStr20 with contract
                               field allowances : Map ByStr20 (Map ByStr20 Uint128)
                         end)

(* Active orders, identified by the order number *)
field active_orders : Map Uint128 Order = Emp Uint128 Order

(* The order number to use when the next order is placed *)
field next_order_no : Uint128 = zero

procedure ThrowListingStatusException(
  token_code : String,
  expected_status : Bool,
  actual_status : Bool)
  e = { _exception : "UnexpectedListingStatus";
```

(下页继续)

(续上页)

```
        token_code: token_code;
        expected : expected_status;
        actual : actual_status };
  throw e
end

procedure ThrowInsufficientAllowanceException(
  token : ByStr20,
  expected : Uint128,
  actual : Uint128)
  e = { _exception : "InsufficientAllowance";
        token: token;
        expected : expected;
        actual : actual };
  throw e
end

(* Check that _sender is the active admin.           *)
(* If not, throw an error and abort the transaction *)
procedure CheckSenderIsAdmin()
  current_admin <- admin;
  is_admin = builtin eq _sender current_admin;
  match is_admin with
  | True =>  (* Nothing to do *)
  | False =>
    (* Construct an exception object and throw it *)
    e = { _exception : "SenderIsNotAdmin" };
    throw e
  end
end

(* Change the active admin *)
transition SetAdmin(
  new_admin : ByStr20 with end
  )
  (* Only the former admin may appoint a new admin *)
  CheckSenderIsAdmin;
  admin := new_admin
end

(* Check that a given token code is not already listed. If it is, throw an error. *)
procedure CheckIsTokenUnlisted(
  token_code : String
```

(下页继续)

(续上页)

```
  )
  (* Is the token code listed? *)
  token_code_is_listed <- exists listed_tokens[token_code];
  match token_code_is_listed with
  | True =>
    (* Incorrect listing status *)
    ThrowListingStatusException token_code false token_code_is_listed
  | False => (* Nothing to do *)
  end
end

(* List a new token on the exchange. Only the admin may list new tokens.  *)
(* If a token code is already in use, raise an error                       *)
transition ListToken(
  token_code : String,
  new_token : ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20␣
↪Uint128) end
  )
  (* Only the admin may list new tokens.  *)
  CheckSenderIsAdmin;
  (* Only new token codes are allowed.  *)
  CheckIsTokenUnlisted token_code;
  (* Everything is ok. The token can be listed *)
  listed_tokens[token_code] := new_token
end

(* Check that the sender has allowed access to sufficient funds *)
procedure CheckAllowance(
  token : ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20 Uint128)␣
↪end,
  expected : Uint128
  )
  actual_opt <-& token.allowances[_sender][_this_address];
  (* Find actual allowance. Use 0 if None is given *)
  actual = match actual_opt with
           | Some x => x
           | None => zero
           end;
  is_sufficient = uint128_le expected actual;
  match is_sufficient with
  | True => (* Nothing to do *)
  | False =>
    ThrowInsufficientAllowanceException token expected actual
```

(下页继续)

(续上页)

```
  end
end

procedure AddOrder(
  order : Order
  )
  (* Get the next order number *)
  order_no <- next_order_no;
  (* Add the order *)
  active_orders[order_no] := order;
  (* Update the next_order_no field *)
  new_order_no = builtin add order_no one;
  next_order_no := new_order_no
end

(* Place an order on the exchange *)
transition PlaceOrder(
  token_code_sell : String,
  sell_amount : Uint128,
  token_code_buy: String,
  buy_amount : Uint128
  )
  (* Check that the tokens are listed *)
  token_sell_opt <- listed_tokens[token_code_sell];
  token_buy_opt <- listed_tokens[token_code_buy];
  match token_sell_opt with
  | Some token_sell =>
    match token_buy_opt with
    | Some token_buy =>
      (* Check that the placer has allowed sufficient funds to be accessed *)
      CheckAllowance token_sell sell_amount;
      (* Transfer the sell tokens to the exchange for holding. Construct a␣
↪TransferFrom message to the token contract. *)
      msg = mk_place_order_msg token_sell _sender _this_address sell_amount;
      (* Send message when the transition completes. *)
      send msg;
      (* Create order and add to list of active orders  *)
      order = Order _sender token_sell sell_amount token_buy buy_amount;
      AddOrder order
    | None =>
      (* Unlisted token *)
      ThrowListingStatusException token_code_buy true false
    end
```

(下页继续)

(续上页)

```
  | None =>
    (* Unlisted token *)
    ThrowListingStatusException token_code_sell true false
  end
end

transition MatchOrder(
  order_id : Uint128)
  order <- active_orders[order_id];
  match order with
  | Some (Order order_placer sell_token sell_amount buy_token buy_amount) =>
    (* Check that the placer has allowed sufficient funds to be accessed *)
    CheckAllowance buy_token buy_amount;
    (* Create the two transfer messages and send them *)
    msgs = mk_make_order_msgs sell_token sell_amount buy_token buy_amount _this_
↪address order_placer _sender;
    send msgs;
    (* Order has now been matched, so remove it *)
    delete active_orders[order_id]
  | None =>
    e = { _exception : "UnknownOrder";
          order_id : order_id };
    throw e
  end
end

procedure CheckInitiator(
  initiator : ByStr20)
  initiator_is_this = builtin eq initiator _this_address;
  match initiator_is_this with
  | True => (* Do nothing *)
  | False =>
    e = { _exception : "UnexpecedTransfer";
          token_address : _sender;
          initiator : initiator };
    throw e
  end
end

transition RecipientAcceptTransferFrom (
  initiator : ByStr20,
  sender : ByStr20,
  recipient : ByStr20,
```

(下页继续)

(续上页)

```
  amount : Uint128)
  (* The exchange only accepts transfers that it itself has initiated.  *)
  CheckInitiator initiator
end

transition TransferFromSuccessCallBack (
  initiator : ByStr20,
  sender : ByStr20,
  recipient : ByStr20,
  amount : Uint128)
  (* The exchange only accepts transfers that it itself has initiated.  *)
  CheckInitiator initiator
end

transition TransferSuccessCallBack (
  initiator : ByStr20,
  sender : ByStr20,
  recipient : ByStr20,
  amount : Uint128)
  (* The exchange only accepts transfers that it itself has initiated.  *)
  CheckInitiator initiator
end
```

正如介绍中提到的，我们保持交流简单化，以保持对 Scilla 功能的关注。

为了进一步熟悉 Scilla，我们鼓励读者添加其他功能，例如取消代币上市、取消订单、有到期时间的订单、以便订单匹配器获得最佳交易的优先订单、订单的部分匹配、反对滥用的交易保护、交易所交易费用等。

3.4 Scilla 深度解析

3.4.1 Scilla 合约结构

Scilla 合约的一般结构在下面的代码片段中给出：

- 合约以 `scilla_version` 的声明开始，它指明合约使用的 Scilla 主版本号。
- 然后是一个包含纯数学函数的 `library` 声明，例如，一个计算两位布尔 `AND` 的函数，或一个计算给定自然数阶乘的函数。
- 然后是遵循使用关键字 `contract` 声明的实际合约定义。
- 在合约中，有四个不同的部分：
  1. 第一部分声明了合约的不可变参数。
  2. 第二部分描述了合约的约束，约束指明在部署合约时必须是有效的。

3. 第三部分声明了可变字段。

4. 第四部分包含所有 `transition` 和 `procedure` 定义。

```
(* Scilla contract structure *)

(***************************************************)
(*                 Scilla version                  *)
(***************************************************)

scilla_version 1

(***************************************************)
(*               Associated library                *)
(***************************************************)

library MyContractLib


(* Library code block follows *)



(***************************************************)
(*             Contract definition                 *)
(***************************************************)

contract MyContract

(* Immutable contract parameters declaration *)

(vname_1 : vtype_1,
 vname_2 : vtype_2)

(* Contract constraint *)
with
  (* Constraint expression *)
=>

(* Mutable fields declaration *)

field vname_1 : vtype_1 = init_val_1
field vname_2 : vtype_2 = init_val_2

(* Transitions and procedures *)
```

(下页继续)

(续上页)

```
(* Procedure signature *)
procedure firstProcedure (param_1 : type_1, param_2 : type_2)
  (* Procedure body *)

end

(* Transition signature *)
transition firstTransition (param_1 : type_1, param_2 : type_2)
  (* Transition body *)

end

(* Procedure signature *)
procedure secondProcedure (param_1 : type_1, param_2 : type_2)
  (* Procedure body *)

end

transition secondTransition (param_1: type_1)
  (* Transition body *)

end
```

不可变合约参数

不可变参数是合约的初始参数，其值在合约部署时定义，之后无法修改。

不可变参数使用以下语法声明：

```
(vname_1 : vtype_1,
 vname_2 : vtype_2,
  ...  )
```

每个声明都包含一个参数名称（一个标识符），后跟它的类型，用 `:` 分隔。多个参数声明由 `,` 分隔。参数的初始化值将在部署合约时指定。

注解： 除了显式声明的不可变参数外，Scilla 合约还具有以下隐式声明的不可变合约参数

1. `_this_address` of type `ByStr20`, which is initialised to the address of the contract when the contract is deployed.

2. `_creation_block` of type `BNum`, which is initialized to the block number at which the contract is /

was deployed.

这些参数可以在实现中自由读取，而不必使用 <- 来取消它的引用调用，并且不能使用 := 修改。

合约约束

合约约束是对合约不可变参数的要求。合约约束提供了一种在合约部署后立即建立合约不变性的方法，从而防止合约使用无意义的参数进行部署。

使用以下语法声明合约约束：

```
with
  ...
=>
```

约束必须是 Bool 类型的表达式。

在部署合约时检查约束。仅当约束判定为 True 时，合同部署才会成功。如果判定结果为 False ，则部署失败。

下面是一个简单的例子，它使用合约约束来确保在该期限之后不会部署具有有限功能期限的合约：

```
contract Mortal(end_of_life : BNum)
with
  builtin blt _creation_block end_of_life
=>
```

上面的代码片段使用了不可变合约参数 中描述的隐式合约参数 _creation_block 。

注解： 声明合同约束是可选的。如果未声明约束，则假定该约束被简单地默认为 True 。

可变字段

可变字段代表合约的可变状态（可变变量）。它们在不可变参数之后声明，每个声明都以关键字 field 为前缀。

```
field vname_1 : vtype_1 = expr_1
field vname_2 : vtype_2 = expr_2
...
```

这里的每个表达式都是相关字段的初始值设定项。这些定义在创建时完成了合约的初始状态。当合约执行 transition 时，这些字段的值会被修改。

注解: 除了显式声明的可变字段之外, Scilla 合约还有一个隐式声明的 `Uint128` 类型的可变字段 `_balance`, 在部署合约时将其初始化为 0。`_balance` 字段保存合约持有的资金量, 以 QA 衡量(1 ZIL = 1,000,000,000,000 QA)。该字段可以在实现中自由读取，但只能通过明确地将资金转移到其他帐户（使用 `send`）或通过接收来自传入消息的资金（使用 `accept`）来修改。

注解: 可变字段和不可变参数都必须是可存储类型:

- 消息、事件和特殊 `Unit` 类型不可存储。所有其他原始类型（如整数和字符串）都是可存储的。
- 函数类型不可存储。
- 涉及未实例化类型变量的复杂类型不可存储。
- 如果映射和 ADT 的子值类型是可存储的，则它们是可存储的。对于映射，这意味着键类型和值类型都必须是可存储的，对于 ADT，这意味着每个构造函数参数的类型都必须是可存储的。

单位

Zilliqa 协议支持三种基本的代币单元——ZIL、LI (10^-6 ZIL) 和 QA (10^-12 ZIL)。

Scilla 智能合约中使用的基本单位是 QA。因此，在使用货币变量时，重要的是需要明确表明最后有多少个 0 附加在 QA 中。

```
(* fee is 1 QA *)
let fee = Uint128 1

(* fee is 1 LI *)
let fee = Uint128 1000000

(* fee is 1 ZIL *)
let fee = Uint128 1000000000000
```

Transitions

transition 是一种定义合约状态如何改变的方式。合约的 transition 定义了合约的公共接口，因为可以通过向合约发送消息来调用 transition。

transition 是用关键字 `transition` 定义的，后跟要传递的参数。定义以 `end` 关键字结束。

```
transition foo (vname_1 : vtype_1, vname_2 : vtype_2, ...)
  ...
end
```

其中 vname : vtype 指定每个参数的名称和类型，多个参数之间用 , 分隔。

注解： 除了定义中显式声明的参数外，每个 transition 都有以下隐式参数：

- _amount : Uint128：发送方发送的传入金额，在 QA 中（请参阅上面有关单位的部分）。要将钱从发送方转移到合约，transition 必须使用 accept 指令明确接受资金。如果 transition 不执行 accept，则不会发生汇款。
- _sender : ByStr20 with end：触发此 transition 的帐户地址。如果 transition 是由合约账户而不是用户账户调用的，则 _sender 是调用此 transition 的合约的地址。在链式调用中，这是发送调用当前 transition 的消息的合约。
- _origin : ByStr20 with end：发起当前交易的账户地址（可能是链式调用）。这始终是用户地址，因为合约永远无法发起交易。

ByStr20 with end 的类型是地址类型。地址类型在地址 部分有详细说明。

注解： transition 的参数必须是可序列化的类型：

- 消息、事件和特殊 Unit 类型不可序列化。
- 字节字符串是可序列化的。地址只能作为 ByStr20 值进行序列化。所有其他原始类型（如整数和字符串）都是可序列化的。
- 函数类型和映射类型不可序列化。
- 涉及未实例化类型变量的复杂类型不可序列化。
- 如果 ADT 的子值的类型是可序列化的，则 ADT 是可序列化的。这意味着每个构造函数参数的类型都必须是可序列化的。

Procedures

procedure 是另一种定义合约状态的方法，现在合约的状态可能会改变，但与 transition 相反，procedure 不是合约公共接口的一部分，并且不能通过向合约发送消息来调用。调用 procedure 的唯一方法是从 transition 或从另一个 procedure 调用它。

procedure 是用关键字 procedure 定义的，后跟要传递的参数。定义以 end 关键字结束。

```
procedure foo (vname_1 : vtype_1, vname_2 : vtype_2, ...)
  ...
end
```

其中 vname : vtype 指定每个参数的名称和类型，多个参数之间用 , 分隔。

一旦定义了 procedure，就可以从合约文件其余部分的 transition 和 procedure 中调用它。但是不能从合约中之前定义的 transition 或 procedure 中调用它，procedure 也不能递归调用自身。

使用 procedure 名称后跟 procedure 的实际参数来调用 procedure：

```
v1 = ...;
v2 = ...;
foo v1 v2;
```

调用 procedure 时必须提供所有参数。procedure 不返回结果。

注解： 隐式 transition 参数 `_sender` 、`_origin` 和 `_amount` 被隐式传递给 transition 调用的所有 procedure。因此，在定义 procedure 时无需显式声明这些参数。

注解： procedure 参数不能是（或包含）映射。如果一个 procedure 需要访问一个映射，就必须让该 procedure 直接访问包含该映射的合约字段，或者使用库函数在映射上执行必要的计算。

表达式

表达式处理纯操作。Scilla 包含以下类型的表达式：

- `let x = f`：在合约中把 `f` 命名为 `x`。`x` 到 `f` 的绑定是全局的，并延伸到合约结束。以下代码片段定义了一个常量 `one`，其值在整个合约中都是 `Int32` 类型的 `1`。

  ```
  let one = Int32 1
  ```

- `let x = f in expr`：将 `f` 绑定到表达式 `expr` 中的名称 `x`。这里的绑定对 `expr` 来说仅仅是 **local**。以下示例将 `Int32` 类型的 `1` 的值绑定到 `one` 以及将 `Int32` 类型的 `2` 的值绑定到 `two` 并使用内置表达式 `builtin add one two` 计算，该表达式将 `1` 与 `2` 相加，因此计算结果为 `Int32` 类型的 `3`。

  ```
  let sum =
    let one = Int32 1 in
    let two = Int32 2 in
    builtin add one two
  ```

- `{ <entry>_1 ; <entry>_2 ... }`：消息或事件表达式，其中每个条目具有以下形式：`b : x`。这里 `b` 是一个标识符，`x` 是一个变量，其值绑定到消息中的标识符。

- `fun (x : T) => expr`：一个函数，它接受类型为 `T` 的输入 `x` 并返回表达式 `expr` 求值结果。

- `f x`：将函数 `f` 应用于参数 `x`。

- `tfun 'T => expr`：一个类型函数，它将 `'T` 作为参数类型并返回表达式 `expr` 求值结果。这些通常用于构建库函数。有关示例，请参阅*fst* 的实现。

注解: 当前不允许隐藏类型变量。例如。`tfun 'T => tfun 'T => expr` 不是有效的表达式。

- `@x T`：将类型函数 `x` 应用于类型 `T` 。这通过将 `x` 的第一个类型变量实例化为 `T` 来专门化类型函数 `x` 。类型应用程序通常在即将应用库函数时使用。有关示例，请参阅*fst* 的示例应用程序。
- `builtin f x`：在 `x` 上应用内置函数 `f` 。
- `match` 表达式：将绑定变量与模式匹配并判断该子句中的表达式。`match` 表达式类似于 OCaml 中的 `match` 表达式。要匹配的模式可以是带有子模式、变量或通配符 _ 的 ADT 构造函数（请参阅*ADTs* ）。如果子模式匹配相应的子值，则 ADT 构造函数模式匹配使用相同构造函数构造的值。变量匹配任何内容，并将变量绑定到它在该子句的表达式中匹配的值。通配符匹配任何内容，但随后会忽略该值。

```
match x with
| pattern_1 =>
  expression_1 ...
| pattern_2 =>
  expression_2 ...
| _ => (*Wildcard*)
  expression ...
end
```

注解: 模式匹配必须是详尽的，即 `x` 的每个合法（类型安全）值都必须与模式匹配。此外，每个模式都必须是可达的，即对于每个模式，必须有一个合法（类型安全）的 `x` 值与该模式匹配，并且不匹配它之前的任何模式。

语句

Scilla 中的语句是有效的运算，因此不是纯粹的数学运算。Scilla 包含以下类型的语句：

- `x <- f`：获取合约字段 `f` 的值，并将其存储到局部变量 `x` 中。
- `f := x`：用 `x` 的值更新可变合约字段 `f` 。`x` 可能是一个局部变量，或另一个合约字段。
- `x <- & BLOCKNUMBER`：获取区块链状态变量 `BLOCKNUMBER` 的值，并将其存储到局部变量 `x` 中。
- `x <- & c.f`：远程获取。获取地址 `c` 处的合约字段 `f` 的值，并将其存储到局部变量 `x` 中。请注意，`c` 的类型必须是包含字段 `f` 的地址类型。有关地址类型的详细信息，请参阅地址 部分。
- `v = e`：计算表达式 `e` ，并将值赋给局部变量 `v` 。
- `p x y z`：使用参数 `x`、`y` 和 `z` 调用 procedure `p`。提供的参数数量必须与 procedure 采用的参数数量相对应。
- `forall ls p`：为列表 `ls` 中的每个元素调用 procedure `p`。`p` 应该被定义为只接受一个类型等于列表 `ls` 元素的参数。

- match：语句级别的模式匹配：

```
match x with
| pattern_1 =>
  statement_11;
  statement_12;
  ...
| pattern_2 =>
  statement_21;
  statement_22;
  ...
| _ => (*Wildcard*)
  statement_n1;
  statement_n2;
  ...
end
```

- accept：接受调用 transition 的消息的 QA。该金额会自动添加到合约的 _balance 字段中。如果消息包含 QA，但调用的 transition 不接受这笔钱，则将钱转回给消息的发送者。不接受传入金额（当它非零时）不是错误。
- send 和 event：与区块链的通信。有关详细信息，请参阅下一节。
- 原位映射操作：对 Map 类型的合约字段的操作。有关详细信息，请参阅*Maps* 部分。

语句序列必须用分号分隔 ;：

```
transition T ()
  statement_1;
  statement_2;
  ...
  statement_n
end
```

请注意，最后的语句没有尾随 ;，因为 ; 用于分隔语句而不是终止它们。

通信

一个合约可以通过 send 指令与其他合约和用户账户进行通信：

- send msgs：发送 msgs 消息列表。

  以下代码片段定义了一条 msg，其中包含四个条目 _tag、_recipient、_amount 和 param。

```
(*Assume contractAddress is the address of the contract being called and the␣
↪contract contains the transition setHello*)
msg = { _tag : "setHello"; _recipient : contractAddress; _amount : Uint128 0;␣
↪param : Uint32 0 };
```

(下页继续)

(续上页)

传递给 send 的消息必须包含必填字段 _tag、_recipient 和 _amount。

_recipient 字段（ByStr20 类型）是消息要发送到的区块链地址，_amount 字段（Uint128 类型）是要转移到该帐户的 QA 数量。

_tag 字段（String 类型）仅在 _recipient 字段的值为合约地址时使用。在这种情况下，_tag 字段的值是要在接收方合约上调用的 transition 的名称。如果收件人是用户帐户，则忽略 _tag 字段。

注解： 为了能够将资金从合约转移到合约和用户账户，请使用 ZRC-5 中的标准 transition 名称，即 AddFunds。请务必检查您打算向其发送资金的合约是否符合 ZRC-5 公约。

除了必填字段之外，消息还可以包含其他字段（任何类型），例如上面的 param。但是，如果消息接收者是合约，则附加字段的名称和类型必须与在接收者合约上调用的 transition 参数相同。

这是一个发送多条消息的示例。

```
msg1 = { _tag : "setFoo"; _recipient : contractAddress1; _amount : Uint128 0;
↪ foo : Uint32 101 };
msg2 = { _tag : "setBar"; _recipient : contractAddress2; _amount : Uint128 0;
↪ bar : Uint32 100 };
msgs =
  let nil = Nil {Message} in
  let m1 = Cons {Message} msg1 nil in
  Cons msg2 m1
  ;
send msgs
```

注解： transition 可以在执行期间的任何时候执行“send“（包括在它调用的 procedure 的执行期间），但是直到 transition 完成后才会继续发送消息。更多细节可以在链式调用 部分找到。

合约还可以通过发出事件与外界进行通信。事件是存储在区块链上供所有人查看的信号。如果用户使用客户端应用程序调用合约上的 transition，则客户端应用程序可以侦听合约可能发出的事件，并提醒用户。

- event e：将消息 e 作为事件发出。以下代码发出一个名为 e_name 的事件。

```
e = { _eventname : "e_name"; <entry>_2 ; <entry>_3 };
event e
```

发出的事件必须包含强制字段 _eventname （String 类型），并且还可以包含其他条目。_eventname 条目的值必须是字符串文字。所有具有相同名称的事件必须具有相同的条目名称和类型。

注解： 与发送消息一样，transition 可以在执行期间的任何时候（包括在它调用的过程的执行期间）发出事件，但在 transition 完成之前，该事件在区块链上是不可见的。更多细节可以在链式调用 部分找到。

运行时错误

transition 在执行过程中可能会遇到运行时错误，例如 out-of-gas 错误、整数溢出或故意抛出的异常。运行时错误会导致 transition 突然终止，并中止整个事务。但是 gas 费用仍旧会被扣除，直到出现错误为止。

抛出异常的语法类似于事件和消息的语法。

```
e = { _exception : "InvalidInput"; <entry>_2; <entry>_3 };
throw e
```

与 `event` 或 `send` 不同，`throw` 的参数是可选的，可以省略。不带参数的抛出异常将导致错误，该错误只会展示发生 `throw` 的位置而没有更多信息。

注解： 如果在执行 transition 期间发生运行时错误，则整个交易将被中止，并且当前合约和其他合约中的任何状态更改都将回滚。（其他合约的状态可能因链式调用而发生变化）。

特别是：

- 即使在错误发生之前执行了 `accept`，所有转移的资金都会返回给各自的发送者。
- 消息队列被清除，因此即使在错误之前执行了 `send`，也不会再继续发送尚未处理的消息。
- 事件列表被清除，因此即使在错误之前执行了 `event`，也不会发出任何事件。

在发生运行时错误之前，仍会为交易收取 Gas 费用。

注解： Scilla 没有异常处理程序。抛出异常总是会中止整个事务。

Scilla 的 Gas 消耗量

部署合约并在其中执行 transition 会消耗 gas。这里 解释了详细的成本机制。

Nucleus 钱包 页面可用于估算某些交易的 gas 成本。

3.4.2 原始数据类型和操作

整数类型

Scilla 定义了 32、64、128 和 256 位的有符号和无符号整数类型。这些整数类型可以用关键字 `IntX` 和 `UintX` 指定，其中 `X` 可以是 32、64、128 或 256。例如，32 位无符号整数的类型是 `Uint32`。

以下代码片段声明了一个 `Uint32` 类型的变量：

```
let x = Uint32 43
```

Scilla 支持以下内置的整数运算。每个操作都采用两个整数 `IntX` / `UintX`（相同类型）作为参数。有两个除外，即 `pow` 的第二个参数始终是 `Uint32` 以及 `isqrt` 接受单个 `UintX` 参数。

- `builtin eq i1 i2`：判断 `i1` 是否等于 `i2`。返回 `Bool`。
- `builtin add i1 i2`：整数值 `i1` 和 `i2` 相加。返回相同类型的整数。
- `builtin sub i1 i2`：从 `i1` 中减去 `i2`。返回相同类型的整数。
- `builtin mul i1 i2`：`i1` 和 `i2` 的整数乘积。返回相同类型的整数。
- `builtin div i1 i2`：`i1` 除以 `i2` 的整数。返回相同类型的整数。
- `builtin rem i1 i2`：`i1` 除以 `i2` 的整数余数。返回相同类型的整数。
- `builtin lt i1 i2`：`i1` 是否小于 `i2`，返回 `Bool`。
- `builtin pow i1 i2`：`i1` 提升到 `i2` 的幂运算。返回与 `i1` 类型相同的整数。
- `builtin isqrt i`：计算 `i` 的整数平方根，即最大整数 `j` 使得 `j * j <= i`。返回与 `i` 类型相同的整数。
- `builtin to_nat i1`：将 `Uint32` 类型的值转换为 `Nat` 类型的等效值。
- `builtin to_(u)int32/64/128/256`：将 `UintX` / `IntX` 或 `String`（表示十进制数）值转换为 `Option UintX` 或 `Option IntX` 类型的结果。如果转换成功，则返回 `Some res`，否则返回 `None`。转换可能会在下面这些情况下失败
  - 没有足够的位来表示结果；
  - 将负整数（或表示负整数的字符串）转换为无符号类型的值时；
  - 输入字符串不能解析为整数。

  以下是具体转换内置函数列表：`to_int32`、`to_int64`、`to_int128`、`to_int256`、`to_uint32`、`to_uint64`、`to_uint128`、`to_uint256`。

加法、减法、乘法、幂、除法和余数运算可能会引发整数溢出、下溢和除零错误。这会中止当前 transition 的执行并还原迄今为止所做的所有状态更改。

注解: 与区块链货币相关的变量，例如消息的 _amount 条目或合约的 _balance 字段，属于 Uint128 类型。

字符串

Scilla 中的 String 文字使用双引号括起来的字符序列表示。可以通过使用关键字 String 指定来声明变量。

以下代码片段声明了一个 String 类型的变量:

```
let x = "Hello"
```

Scilla 支持以下对字符串的内置操作:

- builtin eq s1 s2 : s1 是否等于 s2。返回 Bool。s1 和 s2 必须都是字符串类型。
- builtin concat s1 s2 : 将字符串 s1 与字符串 s2 连接起来。返回一个 String。
- builtin substr s idx len : 从位置 idx 开始提取长度为 len 的 s 子串。idx 和 len 必须是 Uint32 类型。字符串中的字符索引从 0 开始。如果输入参数的组合导致无效的子字符串，则返回 String 或失败并显示运行时错误。
- builtin to_string x:将 x 转换为字符串文字。x 的有效类型是 IntX、UintX、ByStrX 和 ByStr。返回一个 String。字节字符串被转换为文本十六进制表示。
- builtin strlen s : 计算 s （String 类型）的长度。返回一个 Uint32。
- builtin strrev s : 返回字符串 s 的反转。
- builtin to_ascii h : 将字节字符串（ByStr 或 ByStrX）重新解释为可打印的 ASCII 字符串并返回等效的 String 值。如果字节字符串包含任何不可打印的字符，则会引发运行时错误。

字节字符串

Scilla 中的字节字符串使用 ByStr 和 ByStrX 类型表示，其中 X 是一个数字。ByStr 是指任意长度的字节串，从而对应于任意 X，ByStrX 是指固定长度 X 的字节串。例如，ByStr20 是长度为 20 的字节串类型，ByStr32 是长度为 32 的字节串类型，等等。

Scilla 中的字节字符串文字是使用以 0x 为前缀的十六进制字符编写的。请注意，指定 1 个字节需要 2 个十六进制字符，因此 ByStrX 文字需要 2 * X 个十六进制字符。以下代码片段声明了一个 ByStr32 类型的变量:

```
let x = 0x1234567890123456789012345678901234567890123456789012345678901234567890abff
```

Scilla 支持以下用于计算和转换字节字符串类型的内置操作:

- builtin to_bystr h : 将 ByStrX 类型的值 h （对于某些已知的 X）转换为 ByStr 类型的任意长度之一。

- builtin to_bystrX h:(请注意，这里的 X 是数字参数，而不是内置名称的一部分，请参见下面的示例)
  - 如果参数 h 的类型为 ByStr：将任意大小的字节字符串值 h （类型为 ByStr）转换为固定大小的 ByStrX 类型的字节字符串，长度为 X。在这种情况下，结果为 Option ByStrX 类型：如果参数的长度等于 X，内置函数返回 Some res ，否则为 None 。例如，如果 bs 的长度为 42，则 builtin to_bystr42 bs 返回 Some bs'。
  - 如果参数 h 是 Uint(32/64/128/256) 类型：将无符号整数转换为它们的大端字节表示，返回一个 ByStr(4/8/16/32) 值（注意在这个情况下它不是一个可选类型）。例如，builtin to_bystr4 x （仅在 x 的类型为 Uint32 时进行类型检查）或 builtin to_bystr16 x （仅在 x 的类型为 Uint128 时进行类型检查)。
- builtin to_uint(32/64/128/256) h ： 将固定大小的字节字符串值 h 转换为 Uint(32/64/128/256) 类型的等效值。对于一些小于或等于 (4/8/16/32) 的已知 X，h 必须是 ByStrX 类型。假设采用大端表示。
- builtin concat h1 h2：连接字节字符串 h1 和 h2。
  - 如果 h1 的类型为 ByStrX 而 h2 的类型为 ByStrY，则结果的类型为 ByStr(X+Y) 。
  - 如果参数是 ByStr 类型，则结果也是 ByStr 类型。
- builtin strlen h：字节串 (ByStr) h 的长度。返回 Uint32。
- eq a1 a2: a1 是否等于 a2 ，返回一个 Bool。

地址

Zilliqa 网络上的地址是 20 字节的字符串，因此原始地址由 ByStr20 类型的值表示。

此外，Scilla 支持结构化地址类型，即等价于 ByStr20 的类型，但当被解释为网络上的地址时，会提供有关该地址内容的附加信息。地址类型使用形式为 ByStr20 with <address contents> end，其中 <address contents> 指地址包含的内容。

地址类型的层次结构如下：

- ByStr20：长度为 20 的原始字节字符串。该类型不提供任何关于地址处的内容的保证。(通常，ByStr20 不被视为地址类型，因为它可以引用任何长度为 20 的字节串，无论它是否表示地址。)
- ByStr20 with end：一个 ByStr20，当解释为网络地址时，指的是一个正在使用的地址。如果地址包含合约，或者地址的余额或随机数大于 0，则该地址正在使用中。(地址余额是地址帐户持有的 Qa 数量。地址的随机数是从该地址发起的交易数量)。
- ByStr20 with contract end：一个 ByStr20，当解释为网络地址时，指的是合约的地址。
- ByStr20 with contract field f1 : t1, field f2 : t2, ... end: 一个 ByStr20，当解释为网络地址时，指的是包含类型为 t1 的可变字段 f1 、类型为 t2 的可变字段 f2 的合约地址，等等。有一个问题是合约可以定义比类型中指定的更多的字段，但类型中指定的字段必须在合约中定义。

注解: 所有使用中的地址，以及因此通过扩展的所有合约地址，都隐式定义了一个可变字段 `_balance : Uint128`。对于用户帐户，`_balance` 字段是指帐户余额。

注解: 不支持指定不可变参数或合约 transition 的地址类型。

地址子类型

地址类型的层次结构定义了一个子类型关系：

- 任何 `ByStr20 with ... end` 的地址类型都是 `ByStr20` 的子类型。这意味着可以使用任何地址类型代替 `ByStr20`，例如，使用 `builtin eq` 比较相等性，或作为消息的 `_recipient` 值。
- 任何 `ByStr20 with contract ... end` 的合约地址类型都是 `ByStr20 with end` 的子类型。
- 任何指定显式字段 `` `ByStr20 with contract field f1: t11, field f2: t12, ... end `` 的合约地址类型都是指定这些字段的子集 `ByStr20 with contract field f1: t21, field f2: t22 , ... end` 的合约地址类型的子类型，前提是 `t11` 是 `t21` 的子类型，`t12` 是 `t22` 的子类型，两种合同类型中指定的每个字段都依此类推。
- 对于带有 `List` 或 `Option` 等类型参数的 ADT，如果 `T` 与 `S` 相同，并且 `t1` 是 `s1` 的子类型，`t2` 是 `s2` 的子类型，那么 `T t1 t2 ...` 是 `S s1 s2 ...` 的子类型，以此类推。
- 如果 `kt1` 是 `kt2` 的子类型且 `vt1` 是 `vt2` 的子类型，则具有键类型 `kt1` 和值类型 `vt1` 的映射是具有键类型 `kt2` 和值类型 `vt2` 的另一个映射的子类型。

地址的动态类型检查

通常，Scilla 检查器不能完全静态地对地址类型进行类型检查。例如下面的情况就可能会发生，由于字节字符串是一个 transition 参数，因此不能静态地知道它的值，或者因为字节字符串指的是当前不包含合约但将来可能包含合约的地址。

出于这个原因，只有当实际字节字符串已知时，不可变参数（即部署合约时提供的合约参数）和地址类型的 transition 参数才可以进行动态类型检查。

例如，合约可能会指定一个如下所示的不可变字段 `init_owner`：

```
contract MyContract (init_owner : ByStr20 with end)
```

当合约被部署时，作为 `init_owner` 提供的字节字符串被查找为区块链上的地址，如果该地址的内容与地址类型匹配（在地址被用户或合约使用的情况下），那么继续部署，并且 `init_owner` 可以在整个合约中被视为一个 `ByStr20 with end`。

类似地，transition 可能指定如下的参数 `token_contract`：

```
transition Transfer (
    token_contract : ByStr20 with contract
                                    field balances : Map ByStr20 Uint128
                                end
    )
```

当 transition 被调用时，作为 token_contract 参数提供的字节字符串被查找为区块链上的地址，如果该地址的内容与地址类型匹配（在地址包含一个具有 Map ByStr20 Uint128 类型的字段 balances 的合约的情况下），那么 transition 参数成功初始化，并且 token_contract 可以在整个转换调用中被视为 ByStr20 with contract field balances : Map ByStr20 Uint128 end。

在任何一种情况下，如果地址的内容与指定的类型不匹配，则动态类型检查不成功，同时会导致部署（对于失败的不可变参数）或 transition 调用（对于 transition 参数）失败。失败的动态类型检查被视为运行时错误，同时会导致当前事务中止。（出于对不可变字段的动态类型检查的目的，合约的部署被视为交易）。

注解： 无法指定 ByStr20 文字并将其解释为地址。换句话说，以下代码片段将导致静态类型错误：

```
let x : ByStr20 with end = 0x1234567890123456789012345678901234567890
```

针对地址类型验证字节字符串的唯一方法是将其作为不可变字段的值或作为适当类型的 transition 参数传递。

远程获取

要执行远程获取 x <- & c.f，c 的类型必须是某种声明字段 f 的地址类型。例如，如果 c 的类型为 ByStr20 with contract field paused : Bool end，则可以使用语句 x <- & c.paused 获取在地址 c 处 paused 字段的值，而无法获取 c 中未声明字段的值（例如，admin），即使地址 c 的合约确实包含字段 admin。为了能够获取 admin 字段的值，c 的类型也必须包含 admin 字段，例如，ByStr20 with contract field paused : Bool, field admin : ByStr20 end

可以使用与本地声明的地图字段相同的方式使用就地操作执行映射字段的远程获取，即 x <- & cm[key]，x <- & cm[key1][key2]，x <- & exists m[key] 等。与远程获取映射字段一样，远程映射字段必须以 c 类型声明，例如，ByStr20 with contract field m : Map Uint128 (Map Uint32 Bool) end。

不允许写入远程字段。

加密内置插件

Scilla 中的散列是使用数据类型 ByStr32 声明的。一个 ByStr32 代表一个 32 字节（64 个十六进制字符）的十六进制字节串。ByStr32 文字以 0x 为前缀。

Scilla 支持以下对哈希和其他加密原语（包括字节序列）的内置操作。在下面的描述中，Any 可以是 IntX、UintX、String、ByStr20 或 ByStr32 类型。

- builtin eq h1 h2：h1 是否等于 h2？两个输入都是相同类型的 ByStrX（或者都是 ByStr 类型）。返回一个布尔值。
- builtin sha256hash x：将 Any 类型的 x 转换为其 SHA256 哈希。返回一个 ByStr32。
- builtin keccak256hash x：将 Any 类型的 x 转换为其 Keccak256 哈希。返回一个 ByStr32。
- builtin ripemd160hash x：将 Any 类型的 x 转换为其 RIPEMD-160 哈希。返回一个 ByStr20。
- builtin substr h idx len：从位置 idx 开始提取长度为 len 的 h 的子字节串。idx 和 len 必须是 Uint32 类型。字节字符串中的字符索引从 0 开始。返回 ByStr 或失败（导致运行时错误）。
- builtin strrev h：反向字节字符串（ByStr 或 ByStrX）。返回与参数类型相同的值。
- builtin schnorr_verify pubk data sig：使用类型为 ByStr33 的 Schnorr 公钥 pubk 对类型为 ByStr 的字节字符串 data 验证类型为 ByStr64 的签名 sig。
- builtin schnorr_get_address pubk：给定一个 ByStr33 类型的公钥，返回与该公钥对应的 ByStr20 Zilliqa 地址。
- builtin ecdsa_verify pubk data sig：使用类型为 ByStr33 的 ECDSA 公钥 pubk 验证类型为 ByStr64 的签名 sig 与类型为 ByStr 的字节字符串 data。
- builtin ecdsa_recover_pk data sig recid：恢复 data（ByStr 类型），同时具有签名 sig（ByStr64 类型）和 Uint32 类型的恢复整数 recid，其值限制为 0、1、2 或 3，未压缩的公钥，返回一个 ByStr65 的值。
- builtin bech32_to_bystr20 prefix addr。内置函数采用 String 类型的网络特定前缀（"zil" / "tzil"）和输入 bech32 字符串（String 类型），如果输入有效，则将其转换为原始字节地址 (ByStr20)。返回类型是 Option ByStr20。成功时，返回 Some addr，输入无效时返回 None。
- builtin bystr20_to_bech32 prefix addr。内置函数采用 String 类型的网络特定前缀（"zil" / "tzil"）和输入 ByStr20 地址，如果输入有效，则将其转换为 bech32 地址。返回类型是 Option String。成功时，返回 Some addr，无效输入返回“None‘。
- builtin alt_bn128_G1_add p1 p2。内置函数在 alt_bn128 曲线上取两个点 p1、p2，并返回基础组 G1 中的点的总和。输入点和结果点都是一个 Pair {Bystr32 ByStr32}。一个点的每个标量分量 ByStr32 都是一个大端编码的数字。另见 https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md
- builtin alt_bn128_G1_mul p s。内置函数取 alt_bn128 曲线上的一个点 p（如前所述）和一个标量 ByStr32 的值 s 并返回点 p 取 s 次的总和。结果是曲线上的一个点。

- builtin alt_bn128_pairing_product pairs。这个内置函数接受一对点的 pairs 的列表。每对由 G1 组中的一个点（Pair {Bystr32 ByStr32}）作为第一个分量和 G2 组中的一个点（Pair {Bystr64 ByStr64}）作为第二个分量。因此，参数的类型为 List {(Pair (Pair ByStr32 ByStr32) (Pair ByStr64 ByStr64)) }。该函数在每个点上应用一个配对函数来检查是否相等，并根据配对检查是成功还是失败返回 True 或 False。另见 https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md

映射

Map kt vt 类型的值提供了一个键值存储，其中 kt 是键的类型，vt 是值的类型（在其他一些编程语言中，像 Scilla 的 Map 这样的数据类型被称为关联数组、符号表或字典）。映射键 kt 的类型可以是以下 基元类型 中的任意一种：String、IntX、UintX、ByStrX、ByStr 或 BNum。值 vt 的类型可以是除函数类型之外的任何类型，这包括内置和用户定义的代数数据类型。

由于不支持复合类型作为映射键类型，因此建模的方式是通过使用诸如一对值与另一个值的关联的 嵌套映射 实现的。例如，如果一个帐户要与一个特定的受信任用户相关联，同时允许受信任用户代表该帐户花费一定的资金限制，可以使用以下嵌套映射：

```
field operators: Map ByStr20 (Map ByStr20 Uint128)
  = Emp ByStr20 (Map ByStr20 Unit)
```

第一个和第二个键是 ByStr20 类型的，分别代表账户和可信用户。我们用 Uint128 类型表示资金限制。

Scilla 支持多种映射操作，可以归纳为

- 就地操作修改字段映射而不制作任何副本，因此它们属于 Scilla 的命令式片段。这些操作是非常有效的，建议在所有情况下都使用；
- 函数映射操作旨在用于纯函数，例如，在设计 Scilla 库时，因为它们从不修改调用它们的原始映射。这些操作可能会导致显着的性能开销，因为其中一些操作会创建输入映射的新（修改）副本。在语法上，复制操作都以 builtin 关键字为前缀（见下文）。请注意，要在字段映射上调用函数内置函数，首先需要使用如下命令复制字段映射：map_copy <- field_map，这会导致 gas 消耗与 field_map 的大小成正比。

就地映射操作

- m[k] := v: 就地插入。将绑定到值 v 的键 k 插入到映射 m 中。如果 m 已经包含键 k，则绑定到 k 的旧值将被映射中的 v 替换。m 必须引用当前合约中的可变字段。使用语法 m[k1][k2][...] := v 支持插入嵌套映射。如果嵌套映射中不存在中间键，则将它们连同与其关联的映射值一起重新创建。
- x <- m[k]：就地本地获取。获取与映射 m 中的键 k 关联的值。m 必须引用当前合约中的可变字段。如果 k 在 m 中具有关联值 v，则获取的结果为 Some v （请参阅下面的 Option 类型），否则结果为 None。在获取之后，结果被绑定到局部变量 x。使用语法 x <- m[k1][k2][...] 支持从嵌套映射中获取。如果对应的映射中不存在一个或多个中间键，则获取的结果为 None。

- `x <- & c.m[k]`：就地远程获取。除了 `m` 必须引用地址 `c` 处合约中的可变字段，其他以与本地获取操作相同的方式工作。
- `x <- exists m[k]`：就地本地密钥存在检查。`m` 必须引用当前合约中的可变字段。如果 `k` 在映射 `m` 中具有关联值，则检查的结果（`Bool` 类型）为 `True`，否则结果为 `False`。检查后，结果绑定到局部变量 `x`。语法 `x <- exists m[k1][k2][...]` 支持通过嵌套映射进行存在检查。如果对应的映射中不存在一个或多个中间键，则结果为 `False`。
- `b <- & exists c.m[k]`：就地远程密钥存在检查。除了 `m` 必须引用地址 `c` 处合约中的可变字段，其他以与本地密钥存在性检查相同的方式工作。
- `delete m[k]`：就地删除。从映射 `m` 中删除键 `k` 及其关联值。标识符 `m` 必须引用当前合约中的可变字段。使用语法 `delete m[k1][k2][...]` 支持从嵌套映射中删除。如果对应的映射中不存在一个或多个中间键，则该操作无效。请注意，在嵌套删除 `delete m[k1][...][kn-1][kn]` 的情况下，仅删除 `kn` 的键值关联。`k1` 到 `kn-1` 的键值绑定将保留在映射中。

函数式映射操作

- `builtin put m k v`：将绑定到值 `v` 的键 `k` 插入到映射 `m` 中。返回一个新映射，它是 `m` 的副本，但 `k` 与 `v` 关联。如果 `m` 已经包含键 `k`，则绑定到 `k` 的旧值在结果映射中被 `v` 替换。`m` 的值不变。`put` 函数通常用于库函数中。请注意，`put` 在插入键值对之前制作了 `m` 的副本。
- `builtin get m k`：获取映射 `m` 中与键 `k` 关联的值。返回一个可选值（参见下面的 `Option` 类型）——如果 `k` 在 `m` 中有一个关联值 `v`，那么结果是 `Some v`，否则结果是 `None`。`get` 函数通常用于库函数中。
- `builtin contains m k`：键 `k` 是否与映射 `m` 中的值相关联？返回一个 `Bool` 值。`contains` 函数通常用于库函数中。
- `builtin remove m k`：从映射 `m` 中删除键 `k` 及其关联值。返回一个新映射，它是 `m` 的副本，但 `k` 与值无关。`m` 的值不变。如果 `m` 不包含键 `k`，则 `remove` 函数仅返回 `m` 的副本，而不会指示缺少 `k`。`remove` 函数通常用于库函数中。请注意，在删除键值对之前，`remove` 会复制 `m`。
- `builtin to_list m`：将映射 `m` 转换为 `List (Pair kt vt)`，其中 `kt` 和 `vt` 分别是键和值类型（参见下面的 `List` 类型）。
- `builtin size m`：返回映射 `m` 中的绑定数。结果类型为 `Uint32`。调用这个内置函数会消耗少量恒定的 gas 费用。但是不支持直接在 字段映射上调用它，这意味着在避免昂贵的复制的同时获取字段映射的大小需要更多的脚手架，你可以在字段映射大小 部分找到相关信息。

注解： 内置函数 `put` 和 `remove` 返回一个新映射，它可能是原始映射的修改副本。这可能会影响性能！

注解： 可以使用 `Emp` 关键字构造空映射，指定键和值类型作为其参数。这是将 `Map` 字段初始化为空的方法。例如 `field foomap : Map Uint128 String = Emp Uint128 String` 声明一个 `Map` 字段，其键是

Uint128 类型，值是 String 类型，它被初始化为空映射。

区块高度

区块高度在 Scilla 中有一个专用类型 BNum。这种类型的变量用关键字 BNum 指定，后跟一个整数值（例如 BNum 101）。

Scilla 支持以下对区块高度的内置操作：

- eq b1 b2: b1 是否等于 b2？返回一个 Bool 值。
- blt b1 b2: b1 是否小于 b2？返回一个 Bool 值。
- badd b1 i1: 将 UintX 类型的 i1 添加到 BNum 类型的 b1。返回一个 BNum。
- bsub b1 b2: b1 减去 b2，均为 BNum 类型。返回一个 Int256。

3.4.3 代数数据类型

代数数据类型 (ADT) 是函数式编程中常用的复合类型。每个 ADT 被定义为一组 **构造函数**。每个构造函数都接受一组特定类型的参数。

Scilla 配备了许多内置 ADT，如下所述。此外，Scilla 允许用户定义他们自己的 ADT。

布尔值

布尔值使用 Bool 类型指定。Bool ADT 有两个构造函数 True 和 False，它们都不带任何参数。因此，以下代码片段使用构造函数 True 构造了一个 Bool 类型的值：

```
x = True
```

可选值

可选值使用类型 Option t 指定，其中 t 是某种类型。Option ADT 有两个构造函数：

- Some 代表一个值的存在。Some 构造函数接受一个参数（t 类型的值）。
- None 表示没有值。None 构造函数不接受任何参数。

以下代码片段构造了两个可选值。第一个值是一个不存在的字符串类型的值，使用 None 构造。第二个值是 Int32 类型的值 10，因为该值存在，所以使用 Some 构造：

```
let none_value = None {String}

let some_value =
```

(下页继续)

(续上页)

```
  let ten = Int32 10 in
  Some {Int32} ten
```

可选值对于初始化值未知的字段非常有用：

```
field empty_bool : Option Bool = None {Bool}
```

可选值对于可能没有结果的函数也非常有用，例如映射的 get 函数：

```
getValue = builtin get m _sender;
match getValue with
| Some v =>
  (* _sender was associated with v in m *)
  v = v + v;
  ...
| None =>
  (* _sender was not associated with a value in m *)
  ...
end
```

列表

使用 List t 类型指定值列表，其中 t 是某种类型。List ADT 有两个构造函数：

- Nil 表示一个空列表。Nil 构造函数不接受任何参数。
- Cons 代表一个非空列表。Cons 构造函数接受两个参数：列表的第一个元素（t 类型），以及另一个表示列表其余部分的列表（List t 类型）。

列表中的所有元素都必须属于同一类型 t。换句话说，不能将两个不同类型的值添加到同一个列表中。

以下示例显示如何构建 Int32 值列表。首先，我们使用 Nil 构造函数创建一个空列表。然后我们使用 Cons 构造函数一一添加其他四个值。注意列表是如何通过添加最后一个元素，然后是倒数第二个元素，依此类推向后构造的，因此最终列表是 [11; 10; 2; 1]：

```
let one = Int32 1 in
let two = Int32 2 in
let ten = Int32 10 in
let eleven = Int32 11 in

let nil = Nil {Int32} in
let l1 = Cons {Int32} one nil in
let l2 = Cons {Int32} two l1 in
let l3 = Cons {Int32} ten l2 in
  Cons {Int32} eleven l3
```

Scilla 为列表提供了三种结构递归原语，可用于遍历任意列表的所有元素：

- `list_foldl: ('B -> 'A -> 'B) -> 'B -> (List 'A) -> 'B`：从前到后递归处理列表中的元素，同时跟踪 *累加器*（可以认为是一个累计）。`list_foldl` 接受三个参数，它们都依赖于两个类型变量 `'A` 和 `'B`：
  - 处理元素的函数。这个函数有两个参数。第一个参数是累加器（`'B` 类型）的当前值。第二个参数是要处理的下一个列表元素（类型为“’ A“）。该函数的结果是累加器的下一个值（`'B` 类型）。
  - 累加器的初始值（`'B` 类型）。
  - 要处理的元素列表（类型 `(List 'A)`）。

  应用 `list_foldl` 的结果是所有列表元素都被处理后的累加器（`'B` 类型）的值。
- `list_foldr: ('A -> 'B -> 'B) -> 'B -> (List 'A) -> 'B`：与 `list_foldl` 类似，只是列表元素是从后向前处理的。另请注意，处理函数以与 `list_foldl` 中的顺序相反的顺序获取列表元素和累加器。
- `list_foldk: ('B -> 'A -> ('B -> 'B) -> 'B) -> 'B -> (List 'A) -> 'B`：根据 *折叠函数* 递归处理列表中的元素，同时跟踪 *累加器*。`list_foldk` 是左右折叠的更通用版本，顺便说一下，这两个折叠都可以通过它来实现。`list_foldk` 接受三个参数，它们都依赖于两个类型变量 `'A` 和 `'B`：
  - 描述折叠步骤的函数。这个函数接受三个参数。第一个参数是累加器（`'B` 类型）的当前值。第二个参数是要处理的下一个列表元素（类型为 `'A`）。第三个参数表示延迟的递归调用（类型 `'B -> 'B`）。该函数的结果是累加器的下一个值（`'B` 类型）。如果开发者不调用延迟的递归调用，则计算终止。这是 `list_foldk` 和左右折叠之间的主要区别，左右折叠无条件地从头到尾处理它们的输入列表。
  - 累加器 `z` 的初始值（`'B` 类型）。
  - 要处理的元素列表（类型 `List 'A`）。

注解： 当 ADT 接受类型参数（例如 `List 'A`），并出现在更大的类型（例如 `list_foldl` 的类型）中时，必须使用括号将 ADT 及其参数分组 `( )`。即使当 ADT 作为另一个 ADT 的唯一参数出现时也是如此。例如，当构造类型为 `Int32` 的可选值的空列表时，必须使用语法 `Nil {(Option Int32)}` 来实例化列表类型。

为了进一步说明 Scilla 中的 `List` 类型，我们展示了一个使用 `list_foldl` 来计算列表中元素数量的小例子。有关 `list_foldk` 的示例，请参阅*list_find*。

```
let list_length : forall 'A. List 'A -> Uint32 =
  tfun 'A =>
  fun (l : List 'A) =>
  let foldl = @list_foldl 'A Uint32 in
  let init = Uint32 0 in
  let one = Uint32 1 in
```

(下页继续)

(续上页)

```
7    let iter =
8      fun (z : Uint32) =>
9      fun (h : 'A) =>
10       builtin add one z
11   in
12     foldl iter init l
```

list_length 定义了一个函数，它接受一个类型参数 'A 和一个 List 'A 类型的普通（值）参数 l。

'A 是一个类型变量，必须由打算使用 list_length 的代码实例化。类型变量在第 2 行中指定。

在第 4 行，我们实例化 list_foldl 的类型。由于我们正在遍历类型为 'A 的值的列表，我们将 'A 作为第一个类型参数传递给 list_foldl，并且由于我们正在计算列表的长度（一个非负整数），我们将 Uint32 作为累加器类型传递。

在第 5 行中，我们定义了累加器的初始值。由于空列表的长度为 0，因此累加器的初始值为 0（类型为 Uint32，以匹配累加器类型）。

在第 6-10 行，我们指定了处理函数 iter，它接受当前累加器值 z 和当前列表元素 h。在这种情况下，处理函数会忽略列表元素，并将累加器加 1。当列表中的所有元素都被处理后，累加器将增加与列表中元素一样多的次数，因此累加器的最终值将等于列表的长度。

在第 12 行，我们将第 4 行的 list_foldl 的类型实例化版本应用于处理函数、初始累加器和值列表。

作为 Scilla 标准库分发的一部分，ListUtils 库中提供了 List 类型（包括 list_length）的常用实用程序。

Pair

值 Pairs 使用类型 Pair t1 t2 指定，其中 t1 和 t2 是类型。Pair ADT 有一个构造函数：

- Pair 代表一对值。Pair 构造函数接受两个参数，即 pair 的两个值，分别是 t1 和 t2 类型。

注解： Pair 既是类型的名称，也是该类型构造函数的名称。ADT 和构造函数通常仅在构造函数是 ADT 的唯一构造函数时共享它们的名称。

Pair 值可能包含不同类型的值。换句话说，t1 和 t2 不需要是相同的类型。

下面是一个例子，我们声明了一个 Pair String Uint32 类型的字段 pp，然后我们通过构造一个由 String 类型的值和 Uint32 类型的值组成的 pair 来初始化它：

```
field pp: Pair String Uint32 =
              let s1 = "Hello" in
              let num = Uint32 2 in
              Pair {String Uint32} s1 num
```

请注意我们如何将字段类型指定为 `Pair A' B'` 以及如何将提供给构造函数的值类型指定为 `Pair { A' B' }` 的不同。

我们现在说明如何使用模式匹配从 `Pair` 中提取第一个元素。下面显示的函数 `fst` 是在 Scilla 标准库的 `PairUtils` 库中定义的。

```
let fst =
  tfun 'A =>
  tfun 'B =>
  fun (p : Pair ('A) ('B)) =>
    match p with
    | Pair a b => a
    end
```

要应用 `fst`，则必须首先实例化类型变量 `'A` 和 `'B`，具体操作如下：

```
p <- pp;
fst_specialised = @fst String Uint32;
p_fst = fst_specialised p
```

与标识符 `p_fst` 关联的值将是字符串 `"Hello"`。

注解： 通常不鼓励使用 `Pair`。相反，开发者应该定义一个专门用于特定用例中所需的特定类型 pair 的 ADT。请参阅下面有关*User-defined ADTs* 的部分。

Nat

Peano 数字使用 `Nat` 类型指定。`Nat` ADT 有两个构造函数：

- `Zero` 代表数字 0。`Zero` 构造函数不接受任何参数。
- `Succ` 代表另一个 Peano 数的后继者。`Succ` 构造函数接受一个参数（`Nat` 类型），它表示比当前数小 1 的 Peano 数。

下面的代码展示了如何构建对应于整数 3 的 Peano 数：

```
let three =
  let zero = Zero in
  let one  = Succ zero in
  let two  = Succ one in
  Succ two
```

Scilla 为 Peano 数提供了两种结构递归原语，可用于遍历从给定 `Nat` 到 `Zero` 的所有 Peano 数：

- `nat_fold: ('A -> Nat -> 'A) -> 'A -> Nat -> 'A`：递归处理从 `Nat` 到 `Zero` 的数字序列，同时跟踪累加器。`nat_fold` 接受三个参数，其中两个取决于类型变量 `'A`：

  - 处理数字的函数。这个函数有两个参数。第一个参数是累加器（'A 类型）的当前值。第二个参数是下一个要处理的 Peano 数（类型为 Nat）。顺便说一下，下一个要处理的数字是当前正在处理的数字的前身。该函数的结果是累加器的下一个值（'A 类型）。
  - 累加器的初始值（'A 类型）。
  - 要处理的第一个 Peano 数（Nat 类型）。

  应用 nat_fold 的结果是累加器（'A 类型）的值，当所有 Peano 数都被处理到 Zero 时。

- nat_foldk: ('A -> Nat -> ('A ->'A) ->'A) ->'A -> Nat ->'A: 根据折叠函数递归处理 Peano 数到零，同时跟踪累加器。nat_foldk 是左折叠的更通用版本，允许提前终止。它需要三个参数，两个取决于类型变量 'A。
  - 描述折叠步骤的函数。这个函数接受三个参数。第一个参数是累加器（'A 类型）的当前值。第二个参数是正在处理的 Peano 数的前身（Nat 类型）。第三个参数表示延迟的递归调用（类型 'A -> 'A）。该函数的结果是累加器的下一个值（'A 类型）。如果开发者不调用延迟的递归调用，则计算终止。左折叠必须处理整个列表，而 nat_foldk 在这方面可能有所不同。
  - 累加器 z 的初始值（'A 类型）。
  - 要处理的 Peano 数（Nat 类型）。

为了更好地理解 nat_foldk，我们解释了 nat_eq 的工作原理。nat_eq 检查两个 Peano 数是否相等。下面是带有行号和解释的具体程序。

```
1  let nat_eq : Nat -> Nat -> Bool =
2  fun (n : Nat) => fun (m : Nat) =>
3    let foldk = @nat_foldk Nat in
4    let iter =
5      fun (n : Nat) => fun (ignore : Nat) => fun (recurse : Nat -> Nat) =>
6        match n with
7        | Succ n_pred => recurse n_pred
8        | Zero => m    (* m is not zero in this context *)
9        end in
10   let remaining = foldk iter n m in
11   match remaining with
12   | Zero => True
13   |   _ => False
14   end
```

第 2 行指定我们采用两个 Peano 数 m 和 n。第 3 行实例化了 nat_foldk 的类型，因为我们将传递一个 Nat 值作为折叠累加器，所以我们给它 Nat。

第 4 到 8 行指定了折叠描述，这是 nat_foldk 通常采用的第一个参数，类型为 'A -> Nat -> ('A -> 'A) -> 'A，但我们在这种情况下指定了 'A 是 Nat。我们的函数采用累加器 n 并且 ignore : Nat 是正在处理的数字的前身，在这种特殊情况下我们不关心它。

本质上，我们从 n 开始累积最终结果并最多迭代 m 次（参见第 10 行），在每个递归步骤（第 4 - 9 行）递减 n

和 m。m 变量隐式递减，因为这就是 nat_foldk 在幕后工作的方式。我们使用模式匹配（第 6、7 行）显式地递减 n。为了继续迭代递减 m 和 n，我们在第 7 行使用 recurse。如果两个输入数字相等，我们最终将得到等于 0 的累加器 (n)。我们在第 10 行调用累加器的 remaining 来获取最终值。最后，我们将检查我们的累加器是否最终为 Zero，以判断输入数字是否相等。如上所述，在最后几行，当折叠结果为 Zero 时返回 True，否则返回 False。

在累加器 n 达到零（第 8 行）而 m 仍未完全处理的情况下，我们停止迭代（因此该行没有 recurse）并返回一个非零自然数以表示不等式。任何数字（例如 Succ Zero）都可以，但为了使代码简洁，我们返回原始输入数字 m，因为我们知道只有当 m 不为零时才会在 m 上调用 iter。

因为 remaining 得到 n 的最终值，所以在 m 达到零而累加器 n 仍然严格为正的对称情况下，我们表示不等式。

用户自定义 ADT

除了上述内置的 ADT 之外，Scilla 还支持用户自定义 ADT。

ADT 定义可能只出现在程序的库部分：要么出现在合约的库部分，要么出现在导入的库中。ADT 定义在定义它的整个库的范围内，除了 ADT 定义引用之前在同一库或导入库中定义的其他 ADT 定义。特别是，ADT 定义可能不会以归纳/递归方式引用自身。

每个 ADT 定义了一组构造函数。每个构造函数都指定了许多类型，这些类型对应于构造函数采用的参数的数量和类型。构造函数可以指定为不带参数。

一个合约的 ADT 必须有不同的名称，合约中所有 ADT 的所有构造函数的集合也必须有不同的名称。ADT 和构造函数名称都必须以大写字母开头（‘A’ - ‘Z’）。但是，构造函数和 ADT 可能具有相同的名称，就像 Pair 类型的情况一样，它的唯一构造函数也称为 Pair。

作为用户定义的 ADT 的示例，请考虑来自实现名为 Shogi 或 Japanese Chess (https://en.wikipedia.org/wiki/Shogi) 的类象棋游戏的合约中的以下类型声明。当轮到你时，玩家可以选择移动他的一个棋子，将先前捕获的棋子放回棋盘上，或者弃权并将胜利授予对手。

可以使用以下类型 Piece 定义游戏的棋子：

```
type Piece =
| King
| GoldGeneral
| SilverGeneral
| Knight
| Lance
| Pawn
| Rook
| Bishop
```

每个构造函数都代表游戏中的一种棋子。每个构造函数都不接受任何参数。

棋盘表示为一组方块，其中每个方块有两个坐标：

```
type Square =
| Square of Uint32 Uint32
```

类型 Square 是一个类型的示例，其中构造函数与类型具有相同的名称。这通常发生在一个类型只有一个构造函数时。构造函数 Square 接受两个参数，都是 Uint32 类型，它们是棋盘上正方形的坐标（行和列）。

与类型 Piece 的定义类似，我们可以使用构造函数为每个合法方向定义运动方向的类型，如下所示：

```
type Direction =
| East
| SouthEast
| South
| SouthWest
| West
| NorthWest
| North
| NorthEast
```

我们现在可以定义用户在以下情况下可以选择执行的可能操作类型：

```
type Action =
| Move of Square Direction Uint32 Bool
| Place of Piece Square
| Resign
```

如果玩家选择移动一个棋子，她应该使用构造函数 Move，并提供四个参数：

- Square 类型的参数，指示她要移动的棋子的当前位置。
- Direction 类型的参数，指示移动的方向。
- Uint32 类型的参数，指示棋子应移动的距离。
- Bool 类型的参数，指示被移动的棋子在被移动后是否应该被提升。

相反，如果玩家选择将先前捕获的棋子放回棋盘上，她应该使用构造函数 Place，并提供两个参数：

- Piece 类型的参数，指示将哪一块放置在板上。
- Square 类型的参数，指示棋子应放置的位置。

最后，如果玩家选择弃权并将胜利授予她的对手，她应该使用构造函数 Resign。由于 Resign 不接受任何参数，因此不应提供任何参数。

要检查玩家选择了哪个动作，我们使用 match 语句或 match 表达式：

```
transition PlayerAction (action : Action)
  ...
  match action with
```

(下页继续)

(续上页)

```
  | Resign =>
    ...
  | Place piece square =>
    ...
  | Move square direction distance promote =>
    ...
  end;
  ...
end
```

用户自定义 ADT 的类型标识

注解: 由于 Scilla 实现中的错误，本节中的信息仅适用于 Scilla 0.10.0 及更高版本。使用 0.10.0 之前的 Scilla 版本编写并发现此错误的合约必须重新编写和重新部署，因为它们将不再适用于 0.10.0 及更高版本。

每个类型声明定义了一个唯一的类型。特别需要注意的是，这意味着即使两个库都定义了相同的类型，这些类型也被认为是不同的。

例如，思考以下两个合约:

```
library C1Lib

type T =
| C1 of Uint32
| C2 of Bool

contract Contract1()

field contract2_address : ByStr20 = 0x1234567890123456789012345678901234567890

transition Sending ()
  c2 <- contract2_address;
  x = Uint32 0;
  out = C1 x;
  msg = { _tag : "Receiving" ; _recipient : c2 ; _amount : Uint128 0 ;
          param : out };
  no_msg = Nil {Message};
  msgs = Cons {Message} msg no_msg;
  send msgs
end
```

(下页继续)

(续上页)

```
(* ****************************** *)

(* Contract2 is deployed at address 0x1234567890123456789012345678901234567890 *)
library C2Lib

type T =
| C1 of Uint32
| C2 of Bool

contract Contract2()

transition Receiving (param : T)
  match param with
  | C1 v =>
  | C2 b =>
  end
end
```

尽管两个合约都定义了相同的类型 `T`，但这两种类型在 Scilla 中被认为是不同的。特别是，这意味着从 `Contract1` 发送到 `Contract2` 的消息不会触发 transition `Receiving`，因为作为 `param` 消息字段发送的值具有来自 `Contract1` 的类型 `T`，而真正需要的是来自 `Contract2` 的类型 `T`。

为了将用户自定义 ADT 的值作为参数传递给 transition，必须在用户定义的库中定义类型，发送和接收合约都必须导入：

```
library MutualLib

type T =
| C1 of Uint32
| C2 of Bool

(* ****************************** *)

import MutualLib

library C1Lib

contract Contract1()

field contract2_address : ByStr20 = 0x1234567890123456789012345678901234567890

transition Sending ()
  c2 <- contract2_address;
  x = Uint32 0;
```

(下页继续)

(续上页)

```
  out = C1 x;
  msg = { _tag : "Receiving" ; _recipient : c2 ; _amount : Uint128 0 ;
          param : out };
  no_msg = Nil {Message};
  msgs = Cons {Message} msg no_msg;
  send msgs
end

(* ****************************** *)

(* Contract2 is deployed at address 0x1234567890123456789012345678901234567890 *)

scilla_version 0

import MutualLib

library C2Lib

contract Contract2()

transition Receiving (param : T)
  match param with
  | C1 v =>
  | C2 b =>
  end
end
```

用户自定义库 部分提供了有关如何定义和使用库的更多信息。

3.4.4 更多 ADT 示例

为了进一步说明如何使用 ADT，我们提供了更多示例并详细描述了它们。下面描述的两个函数的版本都可以在*Scilla* 标准库 的 `ListUtils` 部分中找到。

计算列表的头部

函数 `list_head` 返回列表的第一个元素。

由于列表可能为空，`list_head` 可能并不总是能够计算结果，因此应该返回 `Option` 类型的值。如果列表非空，并且第一个元素是 `h`，那么 `list_head` 应该返回 `Some h`。否则，如果列表为空，`list_head` 应返回 `None`。

以下代码片段显示了 `list_head` 的实现以及如何应用它：

```
let list_head =
  tfun 'A =>
  fun (l : List 'A) =>
    match l with
    | Cons h t =>
      Some {'A} h
    | Nil =>
      None {'A}
    end

let int_head = @list_head Int32 in

let one = Int32 1 in
let two = Int32 2 in
let three = Int32 3 in
let nil = Nil {Int32} in

let l1 = Cons {Int32} three nil in
let l2 = Cons {Int32} two l1 in
let l3 = Cons {Int32} one l2 in
int_head l3
```

第 2 行指定 'A 是函数的类型参数，而第 3 行指定 l 是 List 'A 类型的（值）参数。换句话说，第 1-3 行指定了一个函数 list_head，它可以为任何类型 'A 实例化，并将 List 'A 类型的值作为参数。

第 4-9 行中的模式匹配匹配 l 的值。在第 5 行，我们匹配列表构造函数 Cons h t，其中 h 是列表的第一个元素，而 t 是列表的其余部分。如果列表不为空，则匹配成功，我们将第一个元素作为可选值返回 Some h。在第 7 行，我们匹配列表构造函数 Nil。如果列表为空，则匹配成功，我们返回可选值 None 表示列表中没有头元素。

第 11 行实例化了 Int32 类型的 list_head 函数，以便将 list_head 应用于 List Int32 类型的值。第 13-20 行构建一个 List Int32 类型的列表，第 21 行在构建的列表上调用实例化的 list_head 函数。

计算左折叠

函数 list_foldl 在给定 function f : 'B -> 'A -> 'B、累加器 z : 'B 和列表 xs : List 'A 的情况下返回左折叠的结果。这可以实现为递归原语或列表实用程序函数。

左折叠是累加器 z 和下一个重复使用 f 的列表元素 x : 'A 的递归应用，直到没有更多列表元素为止。例如，使用从累加器 0 开始的减法在 [1,2,3] 上的左折叠将是 ((0-1)-2)-3 = -6。左折叠在下面的伪代码中解释，注意结果总是累加器类型。

```
list_foldl _ z [] = z
list_foldl f z (x:xs) = list_foldl f (f z x) xs
```

使用 list_foldk 也可以通过部分应用左折叠描述来实现相同的效果；这避免了非法的直接递归。我们的折叠描述 left_f : 'B -> 'A -> ('B -> 'B) -> 'B 接受参数累加器、下一个列表元素和递归调用。递归调用将由 list_foldk 函数提供。下面将解释一个具体实现。

```
1 let list_foldl : forall 'A. forall 'B. ( 'B -> 'A -> 'B) -> 'B -> List 'A -> 'B =
2 tfun 'A => tfun 'B =>
3 fun (f : 'B -> 'A -> 'B) =>
4 let left_f = fun (z: 'B) => fun (x: 'A) =>
5   fun (recurse : 'B -> 'B) => let res = f z x in
6   recurse res in
7 let folder = @list_foldk 'A 'B in
8 folder left_f
```

我们根据第一段所述，在第 1 行，声明名称和类型签名。在第二行，我们说该函数采用两种类型作为参数 'A 和 'B。如第二段所述，第三行我们使用一些函数 f 来处理列表元素和累加器。

在第 4 行，我们使用 f 定义折叠描述。折叠描述不采用函数，而是应该根据某些函数来实现，正如根据类型签名那样，left_f : 'B -> 'A -> ('B -> 'B) -> 'B。left_f 接受第二段中描述的参数。我们计算新的累加器 f z x 并将其称为 res。然后我们用新的累加器递归调用。

在第 7 行，我们使用类型应用程序实例化一个 list_foldk 实例，该实例具有适合作业的正确类型。

在第 8 行，我们部分应用了带有左折叠描述的 folder。list_foldk 的重要之处在于，在调用描述时，它提供对自身的递归调用，每次更改为列表中的下一个元素和相应的尾部。这导致一个函数只需要用户在描述中提供更新的累加器。

计算右折叠

在给定一些函数 f : 'A -> 'B -> 'B、累加器 z : 'B 和列表 xs : List 'A 的情况下，函数 list_foldr 返回右折叠的结果。与 list_foldl 一样，这可以是递归原语或列表实用程序函数。

右折叠类似于左折叠，但在某种程度上是相反的。右折叠从末尾开始应用一个带有累加器 z 的函数 f，然后与倒数第二个元素、倒数第三个元素等组合，直到它到达开头。例如，列表 [1,2,3] 上的右折叠从累加器 0 开始减法直到等于 1-(2-(3-0)) = 2。下面以伪代码列出，注意结果始终是累加器类型。

```
1 list_foldr _ z [] = z
2 list_foldr f z (x:xs) = f x (list_foldr f z xs)
```

像以前一样，通过部分应用右折叠描述，可以使用 list_foldk 实现相同的效果。折叠描述采用参数累加器 z : 'B，下一个列表元素 x : 'A 和递归调用 recurse : 'B -> 'B。递归调用将由 list_foldk 函数提供。下面将解释一个具体实现。

```
1 let list_foldr : forall 'A. forall 'B. ('A -> 'B -> 'B) -> 'B -> List 'A -> 'B =
2 tfun 'A => tfun 'B =>
3 fun (f : 'A -> 'B -> 'B) =>
```

(下页继续)

(续上页)

```
4  let right_f = fun (z: 'B) => fun (x: 'A) =>
5    fun (recurse : 'B -> 'B) => let res = recurse z in f x res in
6  let folder = @list_foldk 'A 'B in
7  folder right_f
```

这与以前非常相似。在第 1 行，我们根据第一段所述，声明名称和类型签名。在第 2 行，我们采用两个类型参数 `'A` 和 `'B`。第三行表示我们使用一些函数 `f` 来处理列表元素 `x : 'A` 和累加器 `z`。参数顺序必然与左折叠的顺序不同。

之后，我们像以前一样编写折叠描述。`list_foldk` 从左到右处理列表。但是我们需要 `list_foldr` 来模拟从右到左的遍历。通过在第 5 行调用 `recurse z` 作为我们的第一个动作，我们使用组合函数 `f` 将实际计算推迟到最后，保留原始累加器。一旦递归调用到达一个空列表，它就会返回原始累加器。然后函数调用 `f x res`（第 5 行）将评估从结尾到开头的向外组合，参见第二段。

第 5 行的递归调用 `recurse z` 似乎每次都一样，但改变的是我们处理的列表元素。

在第 6 行，我们通过应用类型 `'A` 和 `'B` 来实例化 `list_foldk`，并以此创建特定于类型的函数。最后一行我们部分应用带有正确折叠描述的 `folder`。和之前一样，`list_foldk` 的特别之处在于它通过递归调用自身来调用这个函数，每次都会稍微截断列表；它提供了递归。

检查列表中的存在性

函数 `list_exists` 接受一个谓词函数和一个列表，并返回一个值，该值指示谓词是否至少对列表中的一个元素而成立。

谓词函数是一个返回布尔值的函数，由于我们希望将其应用于列表中的元素，因此函数的参数类型应该与列表的元素类型相同。

`list_exists` 应该返回 `True`（如果谓词至少对一个元素成立）或 `False`（如果谓词对列表中的任何元素都不成立），所以 `list_exists` 的返回类型应该是 `Bool`。

以下代码片段显示了 `list_exists` 的实现，以及如何应用它：

```
1  let list_exists =
2    tfun 'A =>
3    fun (f : 'A -> Bool) =>
4    fun (l : List 'A) =>
5      let folder = @list_foldl 'A Bool in
6      let init = False in
7      let iter =
8        fun (z : Bool) =>
9        fun (h : 'A) =>
10         let res = f h in
11         match res with
12         | True =>
```

(下页继续)

(续上页)

```
13            True
14          | False =>
15            z
16          end
17      in
18        folder iter init l
19
20  let int_exists = @list_exists Int128 in
21  let f =
22    fun (a : Int128) =>
23      let three = Int128 3 in
24      builtin lt a three
25
26  (* build list l3 similar to the previous example *)
27  ...
28
29  (* check if l3 has at least one element satisfying f *)
30  int_exists f l3
```

与前面的示例一样，'A 是函数的类型变量。该函数有两个参数:

- 谓词 f，即返回 Bool 的函数。在这种情况下，f 将应用于列表的元素，那么谓词的参数类型应为 'A。因此，f 的类型应该是 'A -> Bool。
- 类型为 List 'A 的元素 l 的列表，以便列表中元素的类型与 f 的参数类型相匹配。

为了遍历输入列表 l 的元素，我们使用 list_foldl。在第 5 行，我们为具有类型 'A 的元素的列表和类型为 Bool 的累加器实例化 list_foldl。在第 6 行中，我们将初始累加器值设置为 False 以指示尚未看到满足谓词的元素。

第 7-16 行中定义的处理函数 iter 测试当前列表元素上的谓词，并返回更新的累加器。如果找到满足谓词的元素，则累加器设置为 True 并在剩余的遍历中保留此值。

累加器的最终值要么是 True，表示 f 对列表中的至少一个元素返回 True，要么是 False，表示 f 对列表中的所有元素都返回 False。

在第 20 行，我们实例化 list_exists 以处理 Int128 类型的列表。在第 21-24 行中，我们定义了谓词，如果其参数小于 3，则返回 True，否则返回 False。

第 27 行省略了构建与前一个示例相同的列表 l3。在第 30 行，我们将实例化的 list_exists 应用于谓词和列表。

找到满足谓词的第一次出现

函数 list_find 在满足谓词 p : 'A -> Bool 的列表中搜索第一次出现。它接受谓词和列表，如果 x 是第一个元素使得 p x 则返回 Some {'A} x :: Option 'A，否则返回 None {'A} :: Option 'A 。

下面我们有一个 list_find 的实现，它说明了如何使用 list_foldk。

```
1  let list_find : forall 'A. ('A -> Bool) -> List 'A -> Option 'A =
2  tfun 'A =>
3  fun (p : 'A -> Bool) =>
4    let foldk = @list_foldk 'A (Option 'A) in
5    let init = None {'A} in
6    (* continue fold on None, exit fold when Some compare st. p(compare) *)
7    let predicate_step =
8      fun (ignore : Option 'A) => fun (x : 'A) =>
9      fun (recurse: Option 'A -> Option 'A) =>
10       let p_x = p x in
11       match p_x with
12       | True => Some {'A} x
13       | False => recurse init
14       end in
15   foldk predicate_step init
```

和以前一样，我们在第 2 行取一个类型变量 'A，并在下一行取谓词。我们首先使用这个类型变量来实例化 foldk，给它我们的处理类型和返回类型。处理类型为列表元素类型，结果类型为 Option 'A。下一行是我们的累加器，我们假设在搜索开始时没有满足条件。

在第 7 行，我们为 foldk 写了一个折叠描述。这体现了递归的顺序和递归的条件。predicate_step 的类型为 Option 'A -> 'A -> (Option 'A -> Option 'A) -> Option 'A。第一个参数是累加器，第二个 x 是下一个要处理的元素，第三个递归是递归调用。我们不在乎累加器 ignore 的是什么，因为如果重要，我们就已经终止了。

在第 10 到 12 行检查 p x，如果成立，则返回 Some {'A} x。在 p x 不成立的情况下，通过递归从头开始尝试下一个元素，依此类推。recurse init 在伪代码中等于 λk. foldk predicate_step init k xs，其中 xs 是我们要处理的元素列表的尾部。

在最后一行中，我们部分应用了 foldk 以便它只接受一个列表参数并给出我们的最终答案。foldk 的第一个参数为我们提供了我们想要的特定折叠，例如，如果你想要一个左折叠，你可以用其他东西替换 predicate_step。

3.4.5 标准库

Scilla 标准库在*Scilla 标准库* 中单独记录。

3.4.6 用户自定义库

除了 Scilla 提供的标准库，用户还可以在区块链上部署库代码。库文件只允许包含纯 Scilla 代码（这与合约库代码的限制相同）。库文件必须使用 `.scillib` 文件扩展名。

下面是一个用户自定义库的示例，它定义了一个函数 `add_if_equal`，如果它们相等，则添加到 `Uint128` 值，否则返回 `0`。

```
import IntUtils

library ExampleLib

let add_if_equal =
  fun (a : Uint128) => fun (b : Uint128) =>
  let eq = uint128_eq a b in
  match eq with
  | True => builtin add a b
  | False => Uint128 0
```

库文件的结构类似于 Scilla 合约的库部分的结构。库文件包含变量和纯库函数的定义，但不包含带有参数、字段、transition 等的实际合约定义。

特别重要的是库不能声明字段。因此，所有库都是无状态的，只能包含纯代码。

与合约导入库的方式类似，库也可以导入其他库（包括用户自定义库）。导入库中变量的范围仅限于直接导入者。因此，如果 `X` 导入库 `Y` 而后者又导入库 `Z`，则 `Z` 中的名称不在 `X` 的范围内，而仅在 `Y` 中。导入中的循环依赖项是不允许的，并在检查阶段标记为错误。

使用用户自定义库进行本地开发

要使用在外部（用户定义）库模块中声明的变量和函数，Scilla 可执行文件的命令行参数必须包含 `-libdir` 选项，以及作为参数的目录列表。如果 Scilla 文件导入库 `ALib`，则 Scilla 可执行文件将在提供的目录中搜索名为 `ALib.scillib` 的库文件。如果多个目录包含具有正确名称的文件，则这些目录的优先级与它们提供给 Scilla 可执行文件的顺序相同。或者，可以将环境变量 `SCILLA_STDLIB_PATH` 设置为库目录列表。

`scilla-checker` 以与合约模块相同的方式检查库模块。同样，`scilla-runner` 可以部署库。请注意，`scilla-runner` 将 `blockhain.json` 作为参数（它是*合约创建* 所使用的方式）作为与合约创建兼容的命令行参数。

区块链上的用户自定义库

虽然 Zilliqa 区块链旨在为执行合约提供标准 Scilla 库，但必须为其提供额外信息以支持用户自定义库。

库的 `init.json` 必须包含一个名为 `_library` 的 `Bool` 条目，设置为 `True`。此外，导入用户自定义库的合约或库必须在其 `init.json` 中包含一个名为 `_extlibs` 的条目，属于 Scilla 类型 `List (Pair String ByStr20)`。列表中的每个条目都将导入的库名称映射到其在区块链中的地址。

继续上一个示例，导入 `Examplelib` 的合约或库应在其 `init.json` 中包含以下条目：

```
[
  ...,
  {
      "vname" : "_library",
      "type" : "Bool",
      "value": { "constructor": "True", "argtypes": [], "arguments": [] }
  }
  {
    "vname" : "_extlibs",
    "type" : "List(Pair String ByStr20)",
    "value" : [
        {
            "constructor" : "Pair",
            "argtypes" : ["String", "ByStr20"],
            "arguments" : ["ExampleLib", "0x9865567890123456789012345678901234567890123456abcd"]
        },
        ...
    ]
  }
]
```

命名空间

导入语句可用于为导入的名称定义单独的命名空间。要将名称从库 `Foo` 推送到命名空间 `Bar`，请使用语句 `import Foo as Bar`。现在必须使用限定名称 `Bar.v` 来访问 `Foo` 中的变量 `v`。这在导入多个定义相同名称的库时很有用。

同一个变量名不能在同一个命名空间中定义多次，因此如果多个导入的库定义了相同的名称，那么最多一个库可以驻留在默认（非限定）命名空间中。所有其他冲突的库必须推送到单独的命名空间。

扩展我们之前的示例，如下所示是一个为了使用函数 `add_if_equal` 而在命名空间 `Bar` 中导入 `ExampleLib` 的合约。

```
scilla_version 0
```

(下页继续)

(续上页)

```
import ExampleLib as Bar

library MyContract

let adder = fun (a : Uint128) => fun (b : Uint128) =>
  Bar.add_if_equal a b

contract MyContract ()
...
```

3.4.7 Scilla 版本

主要和次要版本

Scilla 版本有一个主要版本、次要版本和补丁号，表示为 `X.Y.Z`，其中 `X` 是主要版本，`Y` 是次要版本，`Z` 是补丁号。

- 补丁通常是不影响现有合约行为的错误修复。补丁向后兼容。
- 次要版本通常包括不影响现有合约行为的性能改进和功能添加。次要版本向后兼容，直到最新的主要版本。
- 主要版本不向后兼容。我们希望矿工可以访问 Scilla 的每个主要版本的实现，以运行设置为该主要版本的合约。

在主要版本中，建议矿工使用最新的小版本。

命令 `scilla-runner -version` 将打印被调用的解释器的主要、次要和补丁版本。

合约语法

每个 Scilla 合约都必须以主要版本声明开始。语法如下所示：

```
(***************************************************)
(*                  Scilla version                 *)
(***************************************************)

scilla_version 0

(***************************************************)
(*               Associated library                *)
(***************************************************)

library MyContractLib
```

(下页继续)

(续上页)

```
...

(*************************************************)
(*             Contract definition               *)
(*************************************************)

contract MyContract

...
```

部署合约时，解释器的输出包含字段 `scilla_version : X.Y.Z`，区块链代码将使用该字段来跟踪合约的版本。同样，`scilla-checker` 也会在检查成功时报告合约的版本。

`init.json` 文件

除了合约源代码中指定的版本外，还要求合约的 `init.json` 在部署合约和调用合约 transition 时指定相同的版本。这简化了区块链代码决定调用哪个解释器的过程。

`init.json` 和源代码中指定的版本不匹配将导致解释器产生 gas-charged 错误。

`init.json` 示例：

```
[
   {
      "vname" : "_creation_block",
      "type" : "BNum",
      "value" : "1"
   },
   {
      "vname" : "_scilla_version",
      "type" : "Uint32",
      "value" : "1",
   }
 ]
```

链式调用

当用户通过发送以合约地址作为接收者的消息来调用合约的 transition 时，那么该 transition 可能会继续发送一条或多条消息，也可能会调用其他合约的其他 transition。由此产生的消息、资金转移、transition 调用和合约状态更改的集合称为 *交易*。

发送消息调用另一个 transition （通常在另一个合约上）的 transition 称为 *链式调用*。

在交易期间，维护未处理消息的 LIFO 队列（即堆栈）。最初，消息队列仅包含原始用户发送的单个消息，但当 transition 执行链式调用时，可能会向队列中添加其他消息。

当 transition 完成时，它的传出消息被添加到消息队列中。然后从队列中删除队列中的第一条消息以进行处理。如果没有消息要处理，则交易完成，所有状态更改和资金转移都提交到区块链。

当 transition 发送多条消息时，消息按以下顺序添加到队列中：

- 如果执行多条 `send` 语句，则最后一次 `send` 的消息最先添加。这意味着首先处理第一次 `send` 的消息。
- 如果给 `send` 语句提供了一个包含多条消息的列表，则在添加列表尾部的消息之前，将列表的头部添加到队列中。这意味着列表中的最后一条消息（首先添加到列表中的消息）首先得到处理。

交易执行期间的任何运行时故障都会导致整个交易中止，不再执行进一步的语句，不再处理进一步的消息，回滚所有状态更改，并将所有转移的资金返还给各自的发送者。然而，直到故障点为止，仍然会为交易收取 gas 费用。

单笔交易可以发送的消息总数目前设置为 10。该数字将来可能会进行修订。

不同 Scilla 版本的合约可能会相互调用 transition。合约之间消息传递的语义保证在主要版本之间向后兼容。

Accounting

对于原生 ZIL 资金的转移，Scilla 遵循 *接受语义*。要进行转账，接收方必须通过执行 `accept` 语句明确接受资金——仅发送方执行 `send` 语句是不够的。

当合约执行 `accept` 语句时，传入消息的 `_amount` 被添加到合约的 `_balance` 字段中。同时，`_amount` 从发送方的余额中扣除（如果发送方是合约，则是 `_balance` 字段，或者如果发送方是用户，则从用户账户余额中扣除）。

相反，当合约执行 `send` 语句时，外发消息的 `_amount` 值不会从 `_balance` 字段中扣除，因为外发资金尚未被接收者接受。

注解： 用户帐户（即不持有合约的地址）隐式接受所有传入资金，但在携带资金的消息被处理之前不会这样做。

使用接受语义进行转账意味着转换有可能发送比其合约当前的 `_balance` 多的资金。只有在一个或多个接收者不接受资金，或者如果多个传出消息中的一个（它会导致当前合约接收（和接受）额外资金来支付尚未处理的消息的传出）导致一系列链式调用时，才必须注意要这样做。

如果在交易过程中的任何时候，接收者接受的资金多于发送者余额中的可用资金，则会发生运行时错误，整个交易将被中止。换句话说，在交易期间的任何时候，账户余额都不会低于 0。

3.5 Scilla 标准库

Scilla 标准库包含五个库：`BoolUtils.scilla`、`IntUtils.scilla`、`ListUtils.scilla`、`NatUtils.scilla` 和 `PairUtils.scilla`。顾名思义，这些合约分别为 `Bool`、`IntX`、`List`、`Nat` 和 `Pair` 类型实现了实用操作。

要在合约中使用标准库中的函数，必须使用 `import` 声明导入相关库文件。以下代码片段显示了如何从 `ListUtils` 和 `IntUtils` 库中导入函数：

```
import ListUtils IntUtils
```

`import` 声明必须紧接在合约自己的库声明之前，例如：

```
import ListUtils IntUtils

library WalletLib
... (* The declarations of the contract's own library values and functions *)

contract Wallet ( ... )
... (* The transitions and procedures of the contract *)
```

下面，我们介绍每个库中定义的函数。

3.5.1 BoolUtils

- `andb : Bool -> Bool -> Bool`：计算两个 `Bool` 值的逻辑与。
- `orb : Bool -> Bool -> Bool`：计算两个 `Bool` 值的逻辑或。
- `negb : Bool -> Bool`：计算 Bool 值的逻辑否定。
- `bool_to_string : Bool -> String`：将 `Bool` 值转换为 `String` 值。`True` 变 `"True"`，`False` 变 `"False"`。

3.5.2 IntUtils

- intX_eq： IntX -> IntX -> Bool：专用于 IntX 类型的相等运算符。

```
let int_list_eq = @list_eq Int64 in

let one = Int64 1 in
let two = Int64 2 in
let ten = Int64 10 in
let eleven = Int64 11 in

let nil = Nil {Int64} in
let l1 = Cons {Int64} eleven nil in
let l2 = Cons {Int64} ten l1 in
let l3 = Cons {Int64} two l2 in
let l4 = Cons {Int64} one l3 in

let f = int64_eq in
(* See if [2,10,11] = [1,2,10,11] *)
int_list_eq f l3 l4
```

- uintX_eq ： UintX -> UintX -> Bool：专用于 UintX 类型的相等运算符。
- intX_lt ： IntX -> IntX -> Bool：专用于 IntX 类型的小于运算符。
- uintX_lt ： UintX -> UintX -> Bool：专用于 UintX 类型的小于运算符。
- intX_neq ： IntX -> IntX -> Bool：专用于 IntX 类型的不等运算符。
- uintX_neq : UintX -> UintX -> Bool: 专用于 UintX 类型的不等运算符。
- intX_le : IntX -> IntX -> Bool: 小于或等于 IntX 类型的运算符。
- uintX_le : UintX -> UintX -> Bool: 小于或等于 UintX 类型的运算符。
- intX_gt ： IntX -> IntX -> Bool：专用于 IntX 类型的大于运算符。
- uintX_gt ： UintX -> UintX -> Bool：专用于 UintX 类型的大于运算符。
- intX_ge : IntX -> IntX -> Bool: 大于或等于 IntX 类型的运算符。
- uintX_ge : UintX -> UintX -> Bool: 大于或等于 UintX 类型的运算符。

3.5.3 ListUtils

- list_map : ('A -> 'B) -> List 'A -> : List 'B。

  将 f : 'A -> 'B 应用于 l : List 'A 的每个元素，构造结果的列表（类型为 List 'B）。

```
(* Library *)
let f =
  fun (a : Int32) =>
    builtin sha256hash a

(* Contract transition *)
(* Assume input is the list [ 1 ; 2 ; 3 ] *)
(* Apply f to all values in input *)
hash_list_int32 = @list_map Int32 ByStr32;
hashed_list = hash_list_int32 f input;
(* hashed_list is now [ sha256hash 1 ; sha256hash 2 ; sha256hash 3 ] *)
```

- list_filter : ('A -> Bool) -> List 'A -> List 'A。

  根据谓词 f 'A -> Bool 过滤掉列表中的元素。如果一个元素满足 f，它将在结果列表中，否则将被删除。元素的顺序被保留。

```
(*Library*)
let f =
  fun (a : Int32) =>
    let ten = Int32 10 in
    builtin lt a ten

(* Contract transition *)
(* Assume input is the list [ 1 ; 42 ; 2 ; 11 ; 12 ] *)
less_ten_int32 = @list_filter Int32;
less_ten_list = less_ten_int32 f l
(* less_ten_list is now  [ 1 ; 2 ]*)
```

- list_head : (List 'A) -> (Option 'A)。

  返回列表 l List 'A 的头元素作为可选值。如果 l 的第一个元素 h 不为空，则结果为 Some h。如果 l 为空，则结果为 None。

- `list_tail : (List 'A) -> (Option List 'A)`。

  返回列表 `l : List 'A` 的尾部作为可选值。如果 `l` 是形式为 `Cons h t` 的非空列表，则结果为 `Some t`。如果 `l` 为空，则结果为 `None`。

- `list_foldl_while : ('B -> 'A -> Option 'B) -> 'B -> List 'A -> 'B`

  给定一个函数 `f : 'B -> 'A -> Option 'B`，累加器 `z : 'B` 以及列表 `ls : List 'A`，当我们给定的函数使用 `f z x : 'B` 返回 `Some x : Option 'B` 或者列表为空，但在 `None : Option 'B` 的情况下提前终止，并返回 `z`，来执行左折叠。

```
(* assume zero = 0, one = 1, negb is in scope and ls = [10,12,9,7]
 given a max and list with elements a_0, a_1, ..., a_m
 find largest n s.t. sum of i from 0 to (n-1) a_i <= max *)
let prefix_step = fun (len_limit : Pair Uint32 Uint32) => fun (x : Uint32) =>
  match len_limit with
  | Pair len limit => let limit_lt_x = builtin lt limit x in
    let x_leq_limit = negb limit_lt_x in
    match x_leq_limit with
    | True => let len_succ = builtin add len one in let l_sub_x = builtin sub limit x␣
↪in
      let res = Pair {Uint32 Uint32} len_succ l_sub_x in
      Some {(Pair Uint32 Uint32)} res
    | False => None {(Pair Uint32 Uint32)}
    end
  end in
let fold_while = @list_foldl_while Uint32 (Pair Uint32 Uint32) in
let max = Uint32 31 in
let init = Pair {Uint32 Uint32} zero max in
let prefix_length = fold_while prefix_step init ls in
match prefix_length with
| Pair length _ => length
end
```

- `list_append : (List 'A -> List 'A -> List 'A)`。

  将第一个列表附加到第二个列表的前面，保持两个列表中元素的顺序。请注意，`list_append` 在第一个参数列表的长度上具有线性时间复杂度。

- `list_reverse : (List 'A -> List 'A)`。

  Return the reverse of the input list. Note that `list_reverse` has linear time complexity in the length of the argument list.

- `list_flatten : (List List 'A) -> List 'A`。

  构造一个所有元素都在一个列表中的列表。输入列表（类型为 `List List 'A`）的每个元素（类型为 `List 'A`）都连接在一起，并保持输入列表的顺序。请注意，`list_flatten` 在所有列表中的元素总数中具有线性时间复杂度。

- `list_length : List 'A -> Uint32`

  计算列表中元素的数量。请注意，`list_length` 在列表中的元素数量方面具有线性时间复杂度。

- `list_eq : ('A -> 'A -> Bool) -> List 'A -> List 'A -> Bool`。

  使用谓词函数 `f : 'A -> 'A -> Bool` 逐个元素比较两个列表。如果 `f` 对于每对元素都返回 `True`，则 `list_eq` 返回 `True`。如果 `f` 至少对一对元素返回 `False`，或者如果列表具有不同的长度，则 `list_eq` 返回 `False`。

- `list_mem : ('A -> 'A -> Bool) -> 'A -> List 'A -> Bool`。

  检查元素 `a : 'A` 是否是列表 `l : List'A` 中的元素。`f : 'A -> 'A -> Bool` 应该提供等式比较。

```
(* Library *)
let f =
  fun (a : Int32) =>
  fun (b : Int32) =>
    builtin eq a b

(* Contract transition *)
```

(下页继续)

(续上页)

```
(* Assume input is the list [ 1 ; 2 ; 3 ; 4 ] *)
keynumber = Int32 5;
list_mem_int32 = @list_mem Int32;
check_result = list_mem_int32 f keynumber input;
(* check_result is now False *)
```

- `list_forall : ('A -> Bool) -> List 'A -> Bool`。

  检查列表 `l : List 'A` 的所有元素是否满足谓词 `f : 'A -> Bool`。如果所有元素都满足 `f`，则 `list_forall` 返回 `True`，如果至少有一个元素不满足 `f`，则返回 `False`。

- `list_exists : ('A -> Bool) -> List 'A -> Bool`。

  检查列表 `l : List 'A` 的至少一个元素是否满足谓词 `f : 'A -> Bool`。如果至少有一个元素满足 `f`，则 `list_exists` 返回 `True`，如果没有一个元素满足 `f`，则返回 `False`。

- `list_sort : ('A -> 'A -> Bool) -> List 'A -> List 'A`。

  使用插入排序对输入列表 `l : List 'A` 进行排序。如果第一个参数小于第二个参数，则提供的比较函数 `flt : 'A -> 'A -> Bool` 必须返回 `True`。`list_sort` 具有二次时间复杂度。

```
let int_sort = @list_sort Uint64 in

let flt =
  fun (a : Uint64) =>
  fun (b : Uint64) =>
    builtin lt a b

let zero = Uint64 0 in
let one = Uint64 1 in
let two = Uint64 2 in
let three = Uint64 3 in
let four = Uint64 4 in

(* l6 = [ 3 ; 2 ; 1 ; 2 ; 3 ; 4 ; 2 ] *)
let l6 =
  let nil = Nil {Uint64} in
```

(下页继续)

(续上页)

```
  let l0 = Cons {Uint64} two nil in
  let l1 = Cons {Uint64} four l0 in
  let l2 = Cons {Uint64} three l1 in
  let l3 = Cons {Uint64} two l2 in
  let l4 = Cons {Uint64} one l3 in
  let l5 = Cons {Uint64} two l4 in
  Cons {Uint64} three l5

(* res1 = [ 1 ; 2 ; 2 ; 2 ; 3 ; 3 ; 4 ] *)
let res1 = int_sort flt l6
```

- `list_find : ('A -> Bool) -> List 'A -> Option 'A`。

  返回列表 `l : List 'A` 中满足谓词 `f : 'A -> Bool` 的第一个元素。如果列表中至少有一个元素满足谓词，并且这些元素中的第一个元素是 `x`，则结果是 `Some x`。如果没有元素满足谓词，则结果为 `None`。

- `list_zip : List 'A -> List 'B -> List (Pair 'A 'B)`。

  将两个列表逐个元素组合在一起，产生一个对列表。如果列表具有不同的长度，则最长列表的尾随元素将被忽略。

- `list_zip_with : ('A -> 'B -> 'C) -> List 'A -> List 'B -> List 'C )`。

  使用组合函数 `f : 'A -> 'B -> 'C` 逐个元素地组合两个列表。`list_zip_with` 的结果是将 `f` 应用于两个列表的元素的结果列表。如果列表具有不同的长度，则最长列表的尾随元素将被忽略。

- `list_unzip : List (Pair 'A 'B) -> Pair (List 'A) (List 'B)`。

  将一个对列表拆分为一对列表，这些列表由原始列表对的元素组成。

- `list_nth : Uint32 -> List 'A -> Option 'A`。

  从列表中返回元素编号 `n`。如果列表至少有 `n` 个元素，并且元素编号 `n` 为 `x`，则 `list_nth` 返回 `Some`

x。如果列表少于 n 个元素，则 list_nth 返回 None。

3.5.4 NatUtils

- nat_prev : Nat -> Option Nat: 返回比当前小 1 的 Peano 数字。如果当前数字为 Zero，则结果为 None。如果当前数字是 Succ x，则结果是 Some x。
- nat_fold_while : ('T -> Nat -> Option 'T) -> 'T -> Nat -> 'T: 接受参数 f : 'T -> Nat -> Option 'T 、z : `T 以及 m : Nat。这是提前终止的 nat_fold。只要 f 用新的累加器 y 返回 Some y，就继续递归。一旦 f 返回 None，递归终止。
- is_some_zero : Nat -> Bool：Peano 数字的零检查。
- nat_eq : Nat -> Nat -> Bool: 专门用于 Nat 类型的相等检查。
- nat_to_int : Nat -> Uint32: 将 Peano 数转换为其等效的 Uint32 整数。
- uintX_to_nat : UintX -> Nat：将 UintX 整数转换为其等效的 Peano 数。该整数必须足够小，以适应 Uint32。如果不是，那么就会发生溢出错误。
- intX_to_nat : IntX -> Nat：将 IntX 整数转换为其等效的 Peano 数。该整数必须是非负数，并且必须足够小以适合 Uint32。如果不是，则会发生下溢或上溢错误。

3.5.5 PairUtils

- fst : Pair 'A 'B -> 'A: 提取 Pair 的第一个元素。

```
let fst_strings = @fst String String in
let nick_name = "toby" in
let dog = "dog" in
let tobias = Pair {String String} nick_name dog in
fst_strings tobias
```

- snd : Pair 'A 'B -> 'B: 提取 Pair 的第二个元素。

3.5.6 Conversions

该库提供了 b/w Scilla 类型的转换，特别是整数和字节字符串之间的转换。

为了能够为这些函数指定整数参数的编码，因此为字节序定义了一个类型。

```
type IntegerEncoding =
  | LittleEndian
  | BigEndian
```

下面的函数连同它们的主要结果，也返回 next_pos : Uint32，它指可以从输入 ByStr 值中提取任何进一步数据的位置。这在反序列化字节流时很有用。换句话说，next_pos 指此函数停止从输入字节字符串读取字节的位置。

- substr_safe : ByStr -> Uint32 -> Uint32 -> Option ByStr 虽然 Scilla 提供了一个内置函数来提取字节字符串的子字符串 (ByStr)，但它并不是异常安全的。当提供不正确的参数时，它会抛出异常。此库函数作为异常安全函数提供，用于从字符串 s : ByStr 中提取从位置 pos : Uint32 开始且长度为 len : Uint32 的子字符串。它在成功时返回 Some ByStr，在失败时返回 None。
- extract_uint32 : IntegerEncoding -> ByStr -> Uint32 -> Option (Pair Uint32 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 Uint32 的值。成功时，返回 Some extract_uint32_value next_pos，否则返回 None。
- extract_uint64 : IntegerEncoding -> ByStr -> Uint32 -> Option (Pair Uint64 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 Uint64 的值。成功返回 Some extract_uint64_value next_pos，否则返回 None。
- extract_uint128 : IntegerEncoding -> ByStr -> Uint32 -> Option (Pair Uint128 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 Uint128 的值。成功后，会返回 Some extract_uint128_value next_pos，否则返回 None。
- extract_uint256 : IntegerEncoding -> ByStr -> Uint32 -> Option (Pair Uint256 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 Uint256 的值。成功时，返回 Some extract_uint256_value next_pos，否则返回 None。
- extract_bystr1 : ByStr -> Uint32 -> Option (Pair ByStr1 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 ByStr1 的值。成功后，会返回 Some extract_bystr1_value next_pos，否则返回 None。
- extract_bystr2 : ByStr -> Uint32 -> Option (Pair ByStr2 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 ByStr2 的值。成功时，返回 Some extract_bystr2_value next_pos，否则返回 None。
- extract_bystr20 : ByStr -> Uint32 -> Option (Pair ByStr20 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 ByStr2 的值。成功后，会返回 Some extract_bystr20_value next_pos，否则返回 None。
- extract_bystr32 : ByStr -> Uint32 -> Option (Pair ByStr32 Uint32) 从位置 pos : Uint32 开始，在 bs : ByStr 中提取一个 ByStr2 的值。成功后，会返回 Some extract_bystr32_value next_pos，否则返回 None。
- append_uint32 : IntegerEncoding -> ByStr -> Uint32 -> ByStr 序列化一个 Uint32 的值（使用指定的编码）并将其附加到提供的 ByStr 并返回结果 ByStr。
- append_uint64 : IntegerEncoding -> ByStr -> Uint32 -> ByStr 序列化一个 Uint64 的值（使用指定的编码）并将其附加到提供的 ByStr 并返回结果 ByStr。
- append_uint128 : IntegerEncoding -> ByStr -> Uint32 -> ByStr 序列化一个 Uint128 的值（使用指定的编码）并将其附加到提供的 ByStr 并返回结果 ByStr。

- `append_uint256 : IntegerEncoding -> ByStr -> Uint32 -> ByStr` 序列化一个 `Uint256` 的值（使用指定的编码）并将其附加到提供的 `ByStr` 并返回结果 `ByStr`。

3.5.7 Polynetwork 支持库

该库提供了用于构建 Zilliqa Polynetwork 网桥的实用函数。这些功能从 Polynetwork 的以太坊支持 迁移而来，合约本身单独部署。

3.6 Scilla 注意事项及使用技巧

3.6.1 性能

映射大小

如果你的合约需要知道字段映射的大小，即字段变量是映射，那么将字段映射读入变量并应用内置 `size` 的明显实现（如以下代码片段所示）可能非常低效，因为它需要制作相关映射的副本。

```
field accounts : Map ByStr20 Uint128 = Emp ByStr20 Uint128

transition Foo()
  ...
  accounts_copy <- accounts;
  accounts_size = builtin size accounts_copy;
  ...
end
```

为了解决这个问题，可以跟踪相应字段变量中的大小信息，如下面的代码片段所示。请注意，现在不是复制映射，而是从 `accounts_size` 字段变量中读取。

```
let uint32_one = Uint32 1

field accounts : Map ByStr20 Uint128 = Emp ByStr20 Uint128
field accounts_size : Uint32 = 0

transition Foo()
  ...
  num_of_accounts <- accounts_size;
  ...
end
```

现在，为了确保映射及其大小保持同步，在使用内置的就地语句时需要更新映射的大小，例如使用 `m[k] := v` 或 `delete m[k]` 时，或者更好地定义并使用系统化的程序来做到这一点。

这是更新帐户映射中的键/值对并相应地更改其大小的 procedure 定义。

```
procedure insert_to_accounts (key : ByStr20, value : Uint128)
  already_exists <- exists accounts[key];
  match already_exists with
  | True =>
      (* do nothing as the size does not change *)
  | False =>
      size <- accounts_size;
      new_size = builtin add size uint32_one;
      accounts_size := new_size
  end;
  accounts[key] := value
end
```

这是从帐户映射中删除键/值对并相应地更改其大小的 procedure 定义。

```
procedure delete_from_accounts (key : ByStr20)
  already_exists <- exists accounts[key];
  match already_exists with
  | False =>
    (* do nothing as the map and its size do not change *)
  | True =>
    size <- accounts_size;
    new_size = builtin sub size uint32_one;
    accounts_size := new_size
  end;
  delete accounts[key]
end
```

3.6.2 资金俗语

部分接受资金

假设你正在编写一个让人们互相提示的合约。你很自然地希望避免一个人因为打字错误而给太多小费的情况。要求 Scilla 部分接受传入的资金会很好，但是却没有内置 `accept <cap>`。你可以完全不接受或完全接受资金。我们可以通过完全接受传入资金，然后在小费超过某个上限时立即退还小费来解决此限制。

事实证明，我们可以将这种行为封装为一个可重用的 procedure。

```
procedure accept_with_cap (cap : Uint128)
  sent_more_than_necessary = builtin lt cap _amount;
  match sent_more_than_necessary with
  | True =>
```

(下页继续)

(续上页)

```
      amount_to_refund = builtin sub _amount cap;
      accept;
      msg = { _tag : ""; _recipient: _sender; _amount: amount_to_refund };
      msgs = one_msg msg;
      send msgs
  | False =>
      accept
  end
end
```

现在，procedure `accept_with_cap` 可以如下使用。

```
<contract library and procedures here>

contract Tips (tip_cap : Uint128)

transition Tip (message_from_tipper : String)
  accept_with_cap tip_cap;
  e = { _eventname: "ThanksForTheTip" };
  event e
end
```

3.6.3 安全性

转让合约所有权

如果你的合约有一个所有者（通常意味着拥有管理员权限的人，例如添加/删除用户帐户或暂停/取消暂停合约）可以在运行时更改，那么乍一看，这可以在代码中形式化为字段 `owner` 和像 `ChangeOwner` 一样的 transition：

```
contract Foo (initial_owner : ByStr20)

field owner : ByStr20 = initial_owner

transition ChangeOwner(new_owner : ByStr20)
  (* continue executing the transition if _sender is the owner,
     throw exception and abort otherwise *)
  isOwner;
  owner := new_owner
end
```

但是，这可能会导致当前所有者无法控制合约的情况。例如，当调用 transition `ChangeOwner` 时，由于地址参数中的拼写错误，当前所有者可能会将合约所有权转移到不存在的地址。在这里，我们提供了一种设计模

式来规避这个问题。

确保新的所有者活跃的一种方法是分两个阶段进行所有权转让：

- 当前所有者提出将所有权转让给新的所有者，请注意，此时当前所有者仍然是合约所有者；
- 未来的新所有者接受待处理的所有权转让并成为当前所有者；
- 在未来的新所有者接受转让之前的任何时刻，当前所有者都可以中止所有权转让。

这是上述模式的可能实现（未展示 procedure `isOwner` 的代码）：

```
contract OwnershipTransfer (initial_owner : ByStr20)

field owner : ByStr20 = initial_owner
field pending_owner : Option ByStr20 = None {ByStr20}

transition RequestOwnershipTransfer (new_owner : ByStr20)
  isOwner;
  po = Some {ByStr20} new_owner;
  pending_owner := po
end

transition ConfirmOwnershipTransfer ()
  optional_po <- pending_owner;
  match optional_po with
  | Some pend_owner =>
      caller_is_new_owner = builtin eq _sender pend_owner;
      match caller_is_new_owner with
      | True =>
          (* transfer ownership *)
          owner := pend_owner;
          none = None {ByStr20};
          pending_owner := none
      | False => (* the caller is not the new owner, do nothing *)
      end
  | None => (* ownership transfer is not in-progress, do nothing *)
  end
end
```

具有上述 transition 的合约的所有权转让应该如下所示发生：

- 当前所有者使用新的所有者地址作为其唯一的显式参数调用 transition `RequestOwnershipTransfer`；
- 新所有者调用 `ConfirmOwnershipTransfer`。

在所有权转换完成之前，当前所有者可以通过使用自己的地址调用 `RequestOwnershipTransfer` 来中止它。可以添加（冗余）专用 transition，使其对合约所有者和用户更加明显。

3.7 Scilla 检查器

Scilla 检查器 (`scilla-checker`) 用作编译器前端，解析合约并执行静态检查，包括类型检查。

3.7.1 Scilla 检查器的阶段

Scilla 检查器在不同的阶段运行，每个阶段都可以执行检查（并可能拒绝未通过检查的合约）并为每段语法添加注释：

- *词法分析*读取合约代码并构建抽象语法树（AST）。树中的每个节点都以行号和列号的形式使用源文件中的位置进行注释。
- *ADT 检查*检查用户定义的 ADT 上的各种约束。
- *类型检查*检查合约中的值是否以与类型系统一致的方式使用。类型检查器还用它的类型注释每个表达式。
- *模式检查*检查合约中的每个模式匹配是否是详尽的（这样不会因为找不到匹配而导致执行失败），并且每个模式都可以到达（这样开发者不会无意中引入一个永远无法到达的模式分支）。
- *Event-info* 检查合约中的消息和事件是否包含所有必需的字段。对于事件，如果合约发出两个具有相同名称（`_eventname`）的事件，则它们的结构（字段和类型）也必须相同。
- *现金流分析*分析变量、字段和 ADT 的使用，并尝试确定哪些字段用于表示（原生）区块链货币。不执行任何检查，但表达式、变量、字段和 ADT 都使用标记进行注释，表明它们的用法。
- *付款承兑检查*检查付款承兑合约。如果合约没有接受付款的 transition，那么它会发出警告。此外，如果 transition 具有任何包含多个接受语句的代码路径的可能，则检查会引发警告。此检查不会引发错误，因为不可能通过静态分析在所有情况下都确定是否会到达多个接受语句（如果存在），原因是这可能取决于仅在运行时已知的条件。Scilla 检查器仅在命令行上指定 `-cf` 标志时才执行此检查，即它与现金流分析一起执行。例如，可互换代币合约通常不需要接受语句，因此此检查不是强制性的。
- *健全性检查*执行多种次要检查，例如，transition 或 procedure 的所有参数都具有不同的名称。

注释

Scilla 检查器中的每个阶段都可以为抽象语法树中的每个节点添加注释。注解的类型通过模块签名 `Rep` 的实例化来指定。`Rep` 指定类型 `rep`，即注释的类型：

```
module type Rep = sig
  type rep
  ...
end
```

除了注释类型之外，`Rep` 的实例化还可以声明辅助函数，允许后续阶段访问先前阶段的注释。其中一些函数在 `Rep` 签名中声明，因为它们涉及创建新的抽象语法节点，必须从解析器开始使用注释创建这些节点：

```
module type Rep = sig
  ...

  val mk_id_address : string -> rep ident
  val mk_id_uint128 : string -> rep ident
  val mk_id_bnum    : string -> rep ident
  val mk_id_string  : string -> rep ident

  val rep_of_sexp : Sexp.t -> rep
  val sexp_of_rep : rep -> Sexp.t

  val parse_rep : string -> rep
  val get_rep_str: rep -> string
end
```

如果注释是已执行的阶段之一，则 mk_id_<type> 使用适当的类型注释创建标识符。如果类型检查器还没有被执行，函数就简单地创建一个（无类型的）标识符和一个虚拟位置。

rep_of_sexp 和 sexp_of_rep 用于美观打印。如果 rep 是用 *[@@deriving sexp]*‘ 指令定义的，那么它们会自动生成。

parse_rep 和 get_rep_str 用于缓存类型检查的库，因此如果它们没有改变就不需要再次检查它们。这些可能会在 Scilla 检查器的未来版本中被删除。

例如，考虑注释模块 TypecheckerERep：

```
module TypecheckerERep (R : Rep) = struct
  type rep = PlainTypes.t inferred_type * R.rep
  [@@deriving sexp]

  let get_loc r = match r with | (_, rr) -> R.get_loc rr

  let mk_id s t =
    match s with
    | Ident (n, r) -> Ident (n, (PlainTypes.mk_qualified_type t, r))

  let mk_id_address s = mk_id (R.mk_id_address s) (bystrx_typ address_length)
  let mk_id_uint128 s = mk_id (R.mk_id_uint128 s) uint128_typ
  let mk_id_bnum    s = mk_id (R.mk_id_bnum s) bnum_typ
  let mk_id_string  s = mk_id (R.mk_id_string s) string_typ

  let mk_rep (r : R.rep) (t : PlainTypes.t inferred_type) = (t, r)

  let parse_rep s = (PlainTypes.mk_qualified_type uint128_typ, R.parse_rep s)
  let get_rep_str r = match r with | (_, rr) -> R.get_rep_str rr
```

(下页继续)

(续上页)

```
  let get_type (r : rep) = fst r
end
```

函子（参数化结构）将前一阶段的注释作为参数 R。在 Scilla 检查器中，前一阶段是解析器，但可以通过在顶层解释器指定阶段来把任何阶段添加到两个阶段之间。

rep 类型指定新注释是类型和前一个注释的一对。

函数 get_loc 只是作为上一阶段 get_loc 函数的代理。

函数 mk_id 是 mk_id_<type> 函数的辅助函数，它使用适当的类型注释创建标识符。

mk_rep 函数是类型检查器使用的辅助函数。

美观打印不输出 AST 节点的类型，因此函数 parse_rep 和 get_rep_str 忽略类型注释。

最后，函数 get_type 提供对后续阶段的类型信息的访问。这个函数在 Rep 签名中没有提到，因为一旦类型注释被添加到 AST，它就会被类型检查器使用。

抽象语法

ScillaSyntax 函子定义了 AST 节点类型。每个阶段都会将函子实例化两次，一次用于输入语法，一次用于输出语法。这两个语法实例的区别仅在于每个语法节点的注释类型。如果阶段不产生额外的注释，则两个实例化将是相同的。

参数 SR 和 ER 都是 Rep 类型，分别定义语句和表达式的注释。

```
module ScillaSyntax (SR : Rep) (ER : Rep) = struct

  type expr_annot = expr * ER.rep
  and expr = ...

  type stmt_annot = stmt * SR.rep
  and stmt = ...
end
```

初始注释

解析器生成初始注释，其中仅包含有关语法节点在源文件中的位置的信息。函数 get_loc 允许后续阶段访问该位置。

ParserRep 结构用于语句和表达式的注释。

```
module ParserRep = struct
  type rep = loc
  [@@deriving sexp]

  let get_loc l = l
  ...
end
```

典型阶段

产生附加注释的每个阶段都需要提供 Rep 模块类型的新实现。实现应该以之前的注释类型（作为实现 Rep 模块类型的结构体）作为参数，这样阶段的注释就可以添加到之前阶段的注释中。

类型检查器向 AST 中的每个表达式节点添加类型，但不会向语句节点注释添加任何内容。因此，类型检查器只为表达式定义一个注释类型。

此外，Rep 实现定义了一个函数 get_type，以便后续阶段可以访问注释中的类型。

```
module TypecheckerERep (R : Rep) = struct
  type rep = PlainTypes.t inferred_type * R.rep
  [@@deriving sexp]

  let get_loc r = match r with | (_, rr) -> R.get_loc rr

  ...
  let get_type (r : rep) = fst r
end
```

Scilla 类型检查器获取上一阶段的语句和表达式注释，然后实例化 TypeCheckerERep（创建新的注释类型）、ScillaSyntax（创建前一阶段的抽象语法类型，作为类型检查器的输入）和 ScillaSyntax（创建类型检查器输出的抽象语法类型）。

```
module ScillaTypechecker
  (SR : Rep)
  (ER : Rep) = struct

  (* No annotation added to statements *)
  module STR = SR
  (* TypecheckerERep is the new annotation for expressions *)
  module ETR = TypecheckerERep (ER)

  (* Instantiate ScillaSyntax with source and target annotations *)
  module UntypedSyntax = ScillaSyntax (SR) (ER)
  module TypedSyntax = ScillaSyntax (STR) (ETR)
```

(下页继续)

(续上页)

```
  (* Expose target syntax and annotations for subsequent phases *)
  include TypedSyntax
  include ETR

  (* Instantiate helper functors *)
  module TU = TypeUtilities (SR) (ER)
  module TBuiltins = ScillaBuiltIns (SR) (ER)
  module TypeEnv = TU.MakeTEnv(PlainTypes)(ER)
  module CU = ScillaContractUtil (SR) (ER)
  ...
end
```

至关重要的是，类型检查器模块公开了它生成的注释和语法类型，以便它们可用于下一阶段。

类型检查器最终会实例化辅助函子，例如 `TypeUtilities` 和 `ScillaBuiltIns`。

3.7.2 现金流分析

现金流分析阶段分析合约变量、字段和 ADT 构造函数的使用，并尝试确定哪些字段和 ADT 用于表示（原生）区块链货币。每个合约字段都用一个标签进行注释，指示该字段的用法。

结果标签是基于合约字段、变量和 ADT 构造函数的使用的近似值。标签不保证准确，但旨在作为帮助合约开发人员以预期方式使用其字段的工具。

运行分析

通过使用选项 `-cf` 运行 `scilla-checker` 来激活现金流分析。默认情况下不运行分析，因为它仅打算在合约开发期间使用。

合约永远不会因为现金流分析的结果而被拒绝。由合约开发者决定现金流标签是否与每个合约字段的预期用途一致。

详细分析

分析的工作方式是不断分析合约的 transition 和 procedure，直到没有收集到进一步的信息。

分析的起点是调用合约 transition 的传入消息、合约可能发送的传出消息和事件、合约的账户余额以及从区块链读取的任何字段，例如当前区块高度。

传入和传出消息都包含一个字段 `_amount`，其值是消息在帐户之间转移的金额。每当传入消息的 `_amount` 字段的值加载到局部变量中时，该局部变量就会被标记为代表资金。类似地，用于初始化外发消息的 `_amount` 字段的局部变量也被标记为表示资金。

相反，已知消息字段 `_sender`、`_origin`、`_recipient` 和 `_tag`、事件字段 `_eventname`、异常字段 `_exception` 和区块链字段 `BLOCKNUMBER` 不代表货币，因此任何用于初始化这些字段或从这些字段之一读取保存值的变量都被标记为不代表资金。

一旦标记了某些变量，它们的用法就意味着如何标记其他变量。例如，如果两个标记为货币的变量相加，则结果也被视为代表货币。相反，如果两个标记为非货币的变量相加，则结果被视为代表非货币。

当加载或存储合约字段而使用局部变量时，会发生合约字段的标记。在这些情况下，该字段被认为具有与局部变量相同的标签。

自定义 ADT 的标记是在它们用于构造值时以及在模式匹配中使用时完成的。

一旦 transition 或 procedure 被分析时，局部变量和它们的标签就会被保存并且分析继续到下一个 transition 或 procedure，同时保留合约字段和 ADT 的标签。分析一直持续到所有 transition 和 procedure 都已分析完毕，同时没有任何现有标签已经更改。

标签

分析使用以下标签集：

- *No information*：未收集到有关该变量的信息。这有时（但并非总是）表明该变量未被使用，同时还表明它存在潜在错误。
- *Money*：变量代表货币。
- *Not money*：变量代表货币以外的东西。
- *Map t'*（其中 *'t* 是一个标签）：该变量表示一个映射或一个函数，其共同域被标记为 *t*。因此，当在映射中执行查找时，或对存储在映射中的值应用函数时，结果用 *t* 标记。映射的键总是被假定为 *Not money*。使用变量作为函数参数不会产生标签。
- *T t1 ···tn'*（其中 *'T* 是 ADT，*t1 ···tn* 是标签）：变量表示 ADT 的值，例如 *List* 或 *Option*。标签 *t1 ···tn* 对应于 ADT 的每个类型参数的标签。（请参阅下面的简单*example*。）
- *Inconsistent*：该变量已被用于表示货币和非货币。不一致的用法表示存在错误。

仅部分支持库函数和本地函数，因为没有尝试将参数标签连接到结果标签。但是，完全支持内置函数。

一个简单的例子

思考以下代码片段：

```
match p with
| Nil =>
| Cons x xs =>
  msg = { _amount : x ; ...}
  ...
end
```

x 用于初始化消息的 _amount 字段，因此 x 被标记为 *Money*。由于 xs 是列表的尾部，其中 x 是第一个元素，因此 xs 必须是与 x 具有相同标签的元素列表。由于存在 List 'A 类型具有一个类型参数的事实，因此 xs 被标记为 *List Money*。

类似地，p 与模式 Nil 和 Cons x xs 匹配。Nil 是一个列表，但由于列表是空的，我们对列表的内容一无所知，因此 Nil 模式对应于标签 *List (No information)*。Cons x xs 也是一个列表，但这次我们确实知道了一些关于内容的信息，即第一个元素 x 用 *Money* 标记，列表的尾部用 *List Money* 标记。因此，Cons x xs 对应于 *List Money*。

统一两个标签 *List (No information)* 和 *List Money* 从而给出了 *List Money* 标签，所以 p 被标记为 *List Money*。

ADT 构造函数标记

除了标记字段和局部变量，现金流分析器还标记自定义 ADT 的构造函数。

要了解其工作原理，请思考以下自定义 ADT：

```
type Transaction =
| UserTransaction of ByStr20 Uint128
| ContractTransaction of ByStr20 String Uint128
```

用户交易是接收方为用户账户的交易，因此 UserTransaction 构造函数采用两个参数：接收方用户账户的地址和要转账的金额。

合约交易是接收方是另一个合约的交易，因此 ContractTransaction 接受三个参数：接收方合约的地址、在接收方合约上调用的 transition 名称以及要转移的金额。

就现金流而言，很明显，两个构造函数的最后一个参数都用于表示货币，而所有其他参数都用于表示非货币。因此，现金流分析器尝试使用前几节中描述的原则，用适当的标签来标记两个构造函数的参数。

一个更详细的例子

例如，思考一下用 Scilla 编写的众筹合约。这样的合约可以声明以下不可变参数和可变字段：

```
contract Crowdfunding

(*  Parameters *)
(owner     : ByStr20,
max_block : BNum,
goal      : Uint128)

(* Mutable fields *)
field backers : Map ByStr20 Uint128 = ...
field funded : Bool = ...
```

owner 参数表示部署合约的人的地址。goal 参数是所有者试图筹集的金额，max_block 参数表示达到目标的截止日期。

字段 backers 是捐赠者地址与捐赠金额的映射，funded 字段表示目标是否已经达到。

由于字段 goal 代表金额，因此分析应将目标标记为 *Money*。类似地，backers 字段是一个带有代表 *Money* 的共同域的映射，因此应使用 *Map Money* 标记捐赠者。

相反，owner、max_block 和 funded 都代表货币以外的东西，所以它们都应该被标记为 *Not money*。

现金流分析将根据参数和字段在合约的 transition 和 procedure 中的使用情况对其进行标记，如果结果标签与预期不符，则合约可能在某处包含错误。

3.8 解释器接口

Scilla 解释器提供了一个调用接口，使用户能够使用指定的输入调用 transition 并获得输出。使用提供的输入执行 transition 将产生一组输出，以及智能合约可变状态的变化。

3.8.1 调用接口

在合约所定义的 transaction 可以通过发行事务，或者从其他合约消息调用而被调用。相同的调用接口将被用来调用通过外部交易和合约间的消息调用合约。

解释器 (scilla-runner) 的输入由四个输入 JSON 文件组成，如下所述。每次调用解释器来执行 transition 都必须提供这四个 JSON 输入：

```
./scilla-runner -init init.json -istate input_state.json -iblockchain input_
↪blockchain.json -imessage input_message.json -o output.json -i input.scilla
```

解释器可执行文件可以运行以创建合约（表示为 CreateContract）或调用合约中的 transition（InvokeContract）。根据这两者中的任何一个，都将导致某些论点不存在。下表列出了在这两种情况下应该出现的论点。CreateContract 与 InvokeContract 的区别在于 input_message.json 和 input_state.json 的存在。如果这些参数不存在，那么解释器会将其评估为 CreateContract。否则，它会将其视为 InvokeContract。请注意，对于 CreateContract，解释器仅执行基本检查，例如将合约的不可变参数与 init.json 匹配以及合约定义有没有语法错误。

		当前值	
输入	描述	CreateContract	InvokeContract
init.json	不可变合约参数	Yes	Yes
input_state.json	可变合约状态	No	Yes
input_blockchain.json	区块链状态	Yes	Yes
input_message.json	transition 和参数	No	Yes
output.json	输出	Yes	Yes
input.scilla	输入合约	Yes	Yes

除了上面提供的命令行参数之外，解释器还需要一个强制的 `-gaslimit X` 参数（其中 `X` 是一个正整数值）。如果合约或库模块导入其他库（包括标准库），则必须提供 `-libdir` 选项，以目录列表（标准 PATH 格式）作为参数，指出要搜索的目录从而查找库。

3.8.2 初始化不可变状态

`init.json` 限定了合约的不可变参数的值。它不会在调用之间改变。JSON 是一个包含以下字段的对象数组：

字段	描述
`vname`	不可变合约参数的名称
`type`	不可变合约参数的类型
`value`	不可变合约参数的值

`init.json` 必须指定类型为 `Uint32` 的 `_scilla_version`，指定一个与合约源中指定的值相同的值。此外，区块链将提供两个隐式合约参数 `_this_address`，一个表示合约本身地址的 `ByStr20` 值，和一个 `BNum` 值 `_creation_block`，表示以前和现在创建合约的区块。在使用离线解释器时，你可能需要自己在 `init.json` 中提供这些值。

例 1

对于下面给出的 `HelloWorld.scilla` 合约片段，我们只有一个不可变参数 `owner`。

```
contract HelloWorld
 (* Immutable parameters *)
 (owner: ByStr20)
```

此合约的示例 `init.json` 将如下所示：

```
[
    {
        "vname" : "_scilla_version",
        "type" : "Uint32",
        "value" : "0"
    },
    {
        "vname" : "owner",
        "type" : "ByStr20",
        "value" : "0x1234567890123456789012345678901234567890"
    },
    {
        "vname" : "_this_address",
```

(下页继续)

(续上页)

```
        "type" : "ByStr20",
        "value" : "0xabfeccdc9012345678901234567890f777567890"
    },
    {
        "vname" : "_creation_block",
        "type" : "BNum",
        "value" : "1"
    }
]
```

例 2

对于下面给出的 `Crowdfunding.scilla` 合约片段，我们有三个不可变参数 `owner`、`max_block` 和 `goal`。

```
contract Crowdfunding
    (* Immutable parameters *)
    (owner     : ByStr20,
     max_block : BNum,
     goal      : UInt128)
```

此合约的示例 `init.json` 将如下所示：

```
[
  {
      "vname" : "_scilla_version",
      "type" : "Uint32",
      "value" : "0"
  },
  {
      "vname" : "owner",
      "type" : "ByStr20",
      "value" : "0x1234567890123456789012345678901234567890"
  },
  {
      "vname" : "max_block",
      "type" : "BNum" ,
      "value" : "199"
  },
  {
      "vname" : "_this_address",
      "type" : "ByStr20",
      "value" : "0xabfeccdc9012345678901234567890f777567890"
  },
```

(下页继续)

(续上页)

```
  {
      "vname" : "goal",
      "type" : "Uint128",
      "value" : "500000000000000"
  },
  {
      "vname" : "_creation_block",
      "type" : "BNum",
      "value" : "1"
  }
]
```

例 3：使用地址类型

每当一个合约有一个地址类型的不可变参数时，类型 `ByStr20` 必须用于初始化参数。

对于 `SimpleExchange`，我们有一个不可变参数，它具有地址类型：

```
contract SimpleExchange
(
  initial_admin : ByStr20 with end
)
```

`initial_admin` 参数的 JSON 条目必须使用类型 `ByStr20` 而不是类型 `ByStr20 with end`，因此此合约的示例 `init.json` 可能如下所示：

```
[
  {
      "vname" : "_scilla_version",
      "type" : "Uint32",
      "value" : "0"
  },
  {
      "vname" : "_this_address",
      "type" : "ByStr20",
      "value" : "0xabfeccdc9012345678901234567890f777567890"
  },
  {
      "vname" : "_creation_block",
      "type" : "BNum",
      "value" : "1"
  },
  {
```

(下页继续)

(续上页)

```
      "vname" : "initial_admin",
      "type" : "ByStr20",
      "value" : "0x1234567890123456789012345678901234567890"
  }
]
```

3.8.3 输入区块链状态

input_blockchain.json 将当前区块链状态提供给解释器。它与 init.json 类似，不同之处在于它是一个固定大小的对象数组，其中每个对象只有来自预定集合（对应于实际区块链状态变量）的 vname 字段。

允许 JSON 字段：目前，暴露于合约的唯一区块链值是当前 BLOCKNUMBER。

```
[
    {
        "vname" : "BLOCKNUMBER",
        "type"  : "BNum",
        "value" : "3265"
    }
]
```

3.8.4 输入消息

input_message.json 包含调用 transition 所需的信息。这个 json 是一个包含以下四个对象的数组：

字段	描述
_tag	要调用的 transition
_amount	要转移的 QA 数量
_sender	调用者的地址（在链式调用中，这是直接调用者）
_origin	发起交易的地址
params	参数对象数组

所有四个字段都是强制性的。如果 transition 不带任何参数，则 params 可以为空。

params 数组的编码方式类似于 init.json 的编码方式，每个参数指定必须传递给正在调用的 transition 的 (vname、type、value)。

例 1

对于以下 transition：

```
transition SayHello()
```

下面给出了一个示例 input_message.json：

```
{
    "_tag"    : "SayHello",
    "_amount" : "0",
    "_sender" : "0x1234567890123456789012345678901234567890",
    "_origin" : "0x1234567890123456789012345678901234567890",
    "params"  : []
}
```

例 2

对于以下 transition：

```
transition TransferFrom (from : ByStr20, to : ByStr20, tokens : Uint128)
```

下面给出了一个示例 input_message.json：

```
{
  "_tag"    : "TransferFrom",
  "_amount" : "0",
  "_sender" : "0x6434567890123456789012345678901234567 8cd",
  "_origin" : "0x6434567890123456789012345678901234567 8cd",
  "params"  : [
    {
      "vname" : "from",
      "type"  : "ByStr20",
      "value" : "0x1234567890123456789012345678901234567890"
    },
    {
      "vname" : "to",
      "type"  : "ByStr20",
      "value" : "0x7834567890123456789012345678901234567 8cd"
    },
    {
      "vname" : "tokens",
      "type"  : "Uint128",
      "value" : "500000000000000"
```

(下页继续)

(续上页)

```
    }
  ]
}
```

例 3：使用用户自定义类型

注解： 由于 Scilla 实现中的错误，本节中的信息仅从 Scilla 版本 0.10.0 开始有效。使用 0.10.0 之前的 Scilla 版本编写并碰到此错误的合约必须重新编写和重新部署，因为它们将不再适用于 0.10.0 及更高版本。

当通过解释器接口传递用户自定义类型的值时，json 结构与前面示例中描述的相同。但是，在解释器接口中，所有类型都必须是完全限定的，其定义如下：

- 对于在地址 `A` 部署的模块中定义的用户自定义类型 `T`，完全限定名称是 `A.T`。
- 对于在地址 `A` 部署的模块中定义的用户自定义构造函数 `C`，完全限定名称是 `A.C`。

注解： 为了离线开发，模块的地址被定义为模块的文件名，没有文件扩展名。也就是说，如果合约在文件 `F.scilla` 中定义了带有构造函数 `C` 的类型 `T`，则该类型的完全限定名称为 `F.T`，构造函数的完全限定名称为 `F.C`。

例如，考虑一个实现简单棋盘游戏的合约。合约可能定义一个类型 `Direction` 和一个 transition `MoveAction`，如下：

```
type Direction =
| East
| South
| West
| North

...

transition MoveAction (dir : Direction, spaces : Uint32)
  ...
```

假设合约已部署在地址 `0x1234567890123456789012345678906784567890`。为了调用带有参数 `East` 和 2 的 transition，则需使用类型名称 `0x1234567890123456789012345678906784567890.Direction` 和在消息 JSON 中的构造名称 `0x1234567890123456789012345678906784567890.East`：

```
{
    "_tag": "MoveAction",
```

(下页继续)

(续上页)

```
    "_amount": "0",
    "_sender" : "0x6434567890123456789012345678901234567890cd",
    "_origin" : "0x6434567890123456789012345678901234567890cd",
    "params": [
        {
            "vname" : "dir",
            "type"  : "0x1234567890123456789012345678906784567890.Direction",
            "value" :
            {
                "constructor" : "0x1234567890123456789012345678906784567890.East",
                "argtypes"    : [],
                "arguments"   : []
            }
        },
        {
            "vname" : "spaces",
            "type"  : "Uint32",
            "value" : "2"
        }
    ]
}
```

如果合约具有用户自定义类型的不可变字段，则还必须使用关联的 `init.json` 中的完全限定名称初始化这些字段。

例 4：使用地址类型

传递地址值时，必须使用类型 `ByStr20`。不能在消息中使用地址类型（`ByStr20 with ... end`）。

这意味着对于接下来的 transition

```
transition ListToken(
  token_code : String,
  new_token : ByStr20 with contract field allowances : Map ByStr20 (Map ByStr20␣
↪Uint128) end
  )
```

所述 `input_message.json` 必须使用类型 `ByStr20` 作为 `new_token` 参数，例如，如下所示：

```
{
  "_tag"    : "ListToken",
  "_amount" : "0",
  "_sender" : "0x6434567890123456789012345678901234567890cd",
  "_origin" : "0x6434567890123456789012345678901234567890cd",
```

(下页继续)

(续上页)

```
  "params"  : [
    {
      "vname" : "token_code",
      "type"  : "String",
      "value" : "XYZ"
    },
    {
      "vname" : "new_token",
      "type"  : "ByStr20",
      "value" : "0x7834567890123456789012345678901234567 8cd"
    }
  ]
}
```

3.8.5 解释器输出

解释器将返回一个带有以下字段的 JSON 对象 (`output.json`):

字段	描述
`scilla_major_version`	本合约的 Scilla 语言的主要版本。
`gas_remaining`	调用或部署合约后剩余的 gas。
`_accepted`	传入的 QA 是否已被接受（`"true"` 或 `"false"`）
`message`	要发送到另一个合约/非合约账户的消息（如果有）。
`states`	形成新合约状态的对象数组
`events`	由 transition 及其调用的 procedure 发出的事件数组。

- `message` 是一个与 `input_message.json` 格式类似的 JSON 对象，除了它有一个 `_recipient` 字段而不是 `_sender` 字段。消息中的字段如下:

字段	描述
`_tag`	要调用的 transition
`_amount`	要转移的 QA 数量
`_recipient`	接收者地址
`params`	要传递的参数对象数组

  `params` 数组的编码方式类似于 `init.json` 的编码方式，每个参数指定必须传递给正在调用的 transition 的 (`vname`、`type`、`value`)。

- `states` 是一个对象数组,表示合约的可变状态。状态数组的每个条目还指定了(`vname`、`type`、`value`)。
- `events` 是一个对象数组，表示 transition 发出的事件。事件数组中每个对象的字段如下:

字段	描述
_eventname	事件名称
params	额外的事件字段数组

params 数组的编码方式类似于 init.json 的编码方式，每个参数都需指定事件字段的 (vname、type、value)。

例 1

下面给出了 Crowdfunding.scilla 生成的输出示例。该示例还显示了合同状态中映射的格式。

```
{
  "scilla_major_version": "0",
  "gas_remaining": "7365",
  "_accepted": "false",
  "message": {
    "_tag": "",
    "_amount": "100000000000000",
    "_recipient": "0x1234567890123456789012345678901234567890ab",
    "params": []
  },
  "states": [
    { "vname": "_balance", "type": "Uint128", "value": "300000000000000" },
    {
      "vname": "backers",
      "type": "Map (ByStr20) (Uint128)",
      "value": [
        { "key": "0x1234567890123456789012345678901234567890cd", "val": "200000000000000
↪" },
        { "key": "0x1234567890123456789012345678901234567890abcd", "val": "100000000000000
↪" }
      ]
    },
    {
      "vname": "funded",
      "type": "Bool",
      "value": { "constructor": "False", "argtypes": [], "arguments": [] }
    }
  ],
  "events": [
    {
      "_eventname": "ClaimBackSuccess",
      "params": [
```

(下页继续)

(续上页)

```
        {
          "vname": "caller",
          "type": "ByStr20",
          "value": "0x1234567890123456789012345678901234567890"
        },
        { "vname": "amount", "type": "Uint128", "value": "100000000000000" },
        { "vname": "code", "type": "Int32", "value": "9" }
      ]
    }
  ]
}
```

例 2

对于 ADT 类型的值，`value` 字段包含三个子字段：

- `constructor`：用于构造值的构造函数的名称。
- `argtypes`：类型实例的数组。对于 `List` 和 `Option` 类型，这个数组将包含一种类型，分别指示列表中的元素或可选值的类型。对于 `Pair` 类型，数组将包含两种类型，表示该类型的 pair 中的两个值。对于所有其他抽象数据类型，数组都将是空数组。
- `arguments`：该构造函数的参数。

以下示例显示了如何在输出 json 中表示 `List` 和 `Option` 类型的值：

```
{
  "scilla_major_version": "0",
  "gas_remaining": "7733",
  "_accepted": "false",
  "message": null,
  "states": [
    { "vname": "_balance", "type": "Uint128", "value": "0" },
    {
      "vname": "gpair",
      "type": "Pair (List (Int64)) (Option (Bool))",
      "value": {
        "constructor": "Pair",
        "argtypes": [ "List (Int64)", "Option (Bool)" ],
        "arguments": [
          [],
          { "constructor": "None", "argtypes": [ "Bool" ], "arguments": [] }
        ]
      }
```

(下页继续)

(续上页)

```
    },
    { "vname": "llist", "type": "List (List (Int64))", "value": [] },
    { "vname": "plist", "type": "List (Option (Int32))", "value": [] },
    {
      "vname": "gnat",
      "type": "Nat",
      "value": { "constructor": "Zero", "argtypes": [], "arguments": [] }
    },
    {
      "vname": "gmap",
      "type": "Map (ByStr20) (Pair (Int32) (Int32))",
      "value": [
        {
          "key": "0x1234567890123456789012345678901234567890ab",
          "val": {
            "constructor": "Pair",
            "argtypes": [ "Int32", "Int32" ],
            "arguments": [ "1", "2" ]
          }
        }
      ]
    }
  ],
  "events": []
}
```

3.8.6 输入可变合约状态

input_state.json 包含可变状态变量的当前值。它与 output.json 中的 states 字段具有相同的形式。下面给出了 Crowdfunding.scilla 的 input_state.json 示例。

```
[
  {
    "vname": "backers",
    "type": "Map (ByStr20) (Uint128)",
    "value": [
      {
        "key": "0x1234567890123456789012345678901234567890cd",
        "val": "200000000000000"
      },
      {
        "key": "0x1234567890123456789012345678901234567890ab",
```

(下页继续)

(续上页)


```
        "val": "100000000000000"
      }
    ]
  },
  {
    "vname": "funded",
    "type": "Bool",
    "value": {
      "constructor": "False",
      "argtypes": [],
      "arguments": []
    }
  },
  {
    "vname": "_balance",
    "type": "Uint128",
    "value": "300000000000000"
  }
]
```


3.9 联系我们

有疑问？欢迎加入 Discord 与我们一起探讨。