Если говорить об уже существующей банковской системе, то транзакция внутри какого-нибудь Альфа-банка - это просто редактирование таблицы балансов, где уменьшается число напротив одного имени и увеличивается напротив другого. В случае с межбанковскими переводами подключаются некоторые сторонние организации, например SWIFT, но, по сути, все работает примерно так же.

Когда мы имеем дело с финансовой системой на основе блокчейна, то процесс денежного перевода выглядит совершенно иначе. В Bitcoin не существует никакой общей таблицы вида <адрес, баланс>, ровно как и не существует регулятора, который бы эту таблицу редактировал. В этой статье я покажу, что из себя представляет транзакция в Bitcoin, как она строится, и объясню, зачем же внутри Bitcoin добавлен свой язык программирования, про который все слышали, но никто не видел.

Table of content

Introduction

Как я уже сказал выше, в Bitcoin не существует никакой единой структуры, в которой каждому адресу был бы сопоставлен его текущий баланс. Вместо этого используется тот самый пресловутый блокчейн, то есть хранятся вообще все транзакции. Для простоты пока что можете считать, что это сообщения вида

```
<address 1> sent <amount> BTC to <address 2>
```

А значит, если пройтись по всему блокчейну, то можно посчитать, сколько монет "принадлежит" конкретному адресу.

Inputs & outputs

Реальная транзакция в сети Bitcoin, на самом деле, немного сложнее описанной выше. В действительности, это некоторая громоздкая структура, главными составляющими которой являются входы (inputs) и выходы (outputs).

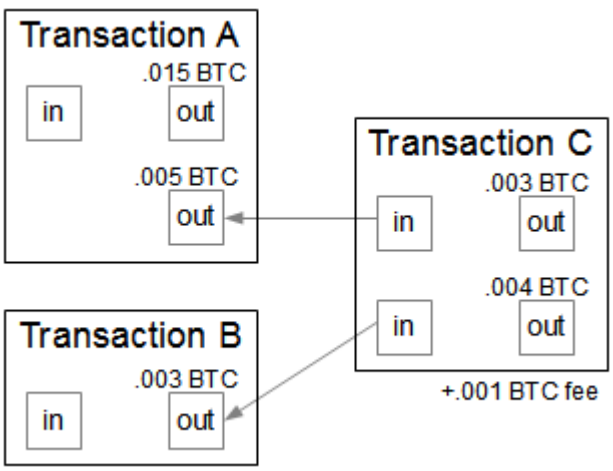
Inputs - это транзакции, на которые вы "ссылаетесь". Представим, что на ваш адрес **X** когда-то было отправлено три транзакции:

- **TXN_ID** - 123456, **VALUE** - 40 BTC
- **TXN_ID** - 6453795, **VALUE** - 10 BTC
- **TXN_ID** - 888888, **VALUE** - 100 BTC

Если вам нужно потратить, например, 45 BTC, то вы можете сослаться на транзакцию **888888**, или сразу на две транзакции: **123456** и **6453795**. При желании вы можете даже сослаться на все три транзакции, правда непонятно зачем.

Outputs - дословно "выходы". Пока что можете считать, что это адреса (хотя это не так), на которые в результате исполнения транзакции будут "отправлены" средства. Выходов также может быть несколько, и каждому из них указывается своя сумма.

На картинке ниже создается новая транзакция **C**, которая ссылается на две входящие - **A** и **B**. В результате на входе у транзакции получается **0.008 BTC**, которые потом разделяются на два выхода - на первый адрес отправляется **0.003 BTC**, а на второй **0.004 BTC**.



Возможность указать сразу несколько выходов - это очень важная фича, потому что транзакцию (а если точнее - ее выход) можно использовать как вход только один раз и только целиком. То есть если у вас есть входящая транзакция на 10 BTC, а вам нужно потратить 8 из них в каком-нибудь Старбаксе, вы просто создаете транзакцию с одним входом и двумя выходами: на 8 BTC в магазин и на 2 BTC обратно на свой адрес. Если же вы создадите транзакцию, в которой сумма выходов меньше суммы входов (как на картинке), то разница отправляется на адрес майнера, записавшего вашу транзакцию в блок.

Fee

Именно эта разница между суммой входов и суммой выходов и называется *transaction fee*, то есть комиссия за транзакцию. Она является вторым по важности источником дохода для майнеров и именно от нее зависит время включения транзакции в блокчейн. Это связано с тем, что у каждого майнера существует некоторый пул непроверенных транзакций, которые претендуют на попадание в блок, и, как правило, майнер просто сортирует их по убыванию комиссии, тем самым максимизируя свою прибыль. Поэтому чем больше комиссия, тем выше вы окажетесь в очереди и тем быстрее пройдет ваш платеж.

На картинке ниже вы можете видеть фоторобот майнера, которому пришла транзакция с комиссией в 135.000$.

UTXO

Как только новая транзакция занесена в блокчейн, ее выходы могут быть использованы в качестве входов. Для таких, пока еще непотраченных выходов, существует специальное название - **UTXO** (*unspent transaction output*). Как я уже говорил, каждый выход может быть использован в качестве входа только один раз, поэтому на практике интерес представляют именно непотраченные выходы, а уже использованные хранятся скорее как дань безопасности системы.

BTW под UTXO часто подразумевают весь массив непотраченных выходов, хотя воспитанные молодые люди должны писать *UTXO pool* ну или в крайнем случае *UTXO set*.

Возвращаясь к началу статьи, теперь вам должно быть понятно, что для подсчета баланса адреса не нужно перебирать весь блокчейн, а достаточно обойтись только перебором *UTXO pool*, что, очевидно, быстрее.

Structure

Общий вид транзакции описан в официальной спецификации протокола, здесь же я приведу живой пример, взятый из блога Ken Shirriff.

version		`01 00 00 00`
input count		`01`
input	previous output hash (reversed)	`48 4d 40 d4 5b 9e a0 d6 52 fc a8 25 8a b7 ca a4 25 41 eb 52 97 58 57 f9 6f b5 0c d7 32 c8 b4 81`
	previous output index	`00 00 00 00`
	script length	
	scriptSig	`script containing signature`
	sequence	`ff ff ff ff`
output count		`01`
output	value	`62 64 01 00 00 00 00 00`
	script length	
	scriptPubKey	`script containing destination address`
block lock time		`00 00 00 00`

По какой-то загадочной причине, *value* и *previous output hash* должны быть представлены в *little endian* форме, то есть в нашем случае хэш транзакции, на которую мы ссылаемся, вообще-то равен *81 b4 c8 32...*, хотя в транзакции он записывается в виде *...32 c8 b4 81*. Точно так же сумма транзакции равна *0.00091234* BTC или *0x016462* в hex, но в протоколе она записывается как *62 64 01 00 00 00 00 00*.

BTW хэш транзакции считается крайне просто - берете всю транзакцию в виде последовательности байт (в примере выше получается строка вида *010000000148....00*), два раза считаете от нее хэш SHA-256 и представляете результат в little endian форме.

previous output index - ссылаемся не на саму транзакцию, хэш которой указан в *previous output hash*, а на один из ее выходов. В этом параметре мы и указываем, какой конкретно выход нас интересует, нумерация начинается с нуля. Кстати, в тексте я часто буду говорить именно о "ссылке на транзакцию", но это только ради выразительности языка.

block lock time - этот параметр довольно редко используется на практике. Если он не равен 0 и меньше 500 млн, то это номер блока, начиная с которого данной транзакцией можно воспользоваться в качестве входа. Так как в среднем блоки появляются раз в 10 минут, то несложно прикинуть время, когда транзакция "откроется".

Если *lock time* больше 500 млн, то он означает UNIX timestamp, начиная с которого транзакция станет доступна. В нашем случае там стоит 0, то есть транзакция доступна сразу.

sequence - эта фича больше не используется, почитать про нее можно здесь.

Параметры со словом script в названии существенно сложнее, о них будет рассказано ниже.

Script

Скорее всего вы уже слышали, что в сети Bitcoin существует механизм, основанный на криптостойких алгоритмах + паре приватный / публичный ключ, позволяющий создать систему, в которой только владелец приватного ключа может воспользоваться монетами, ассоциированными с адресом, полученным из этого ключа. Сейчас я покажу, как это реализуется "под капотом".

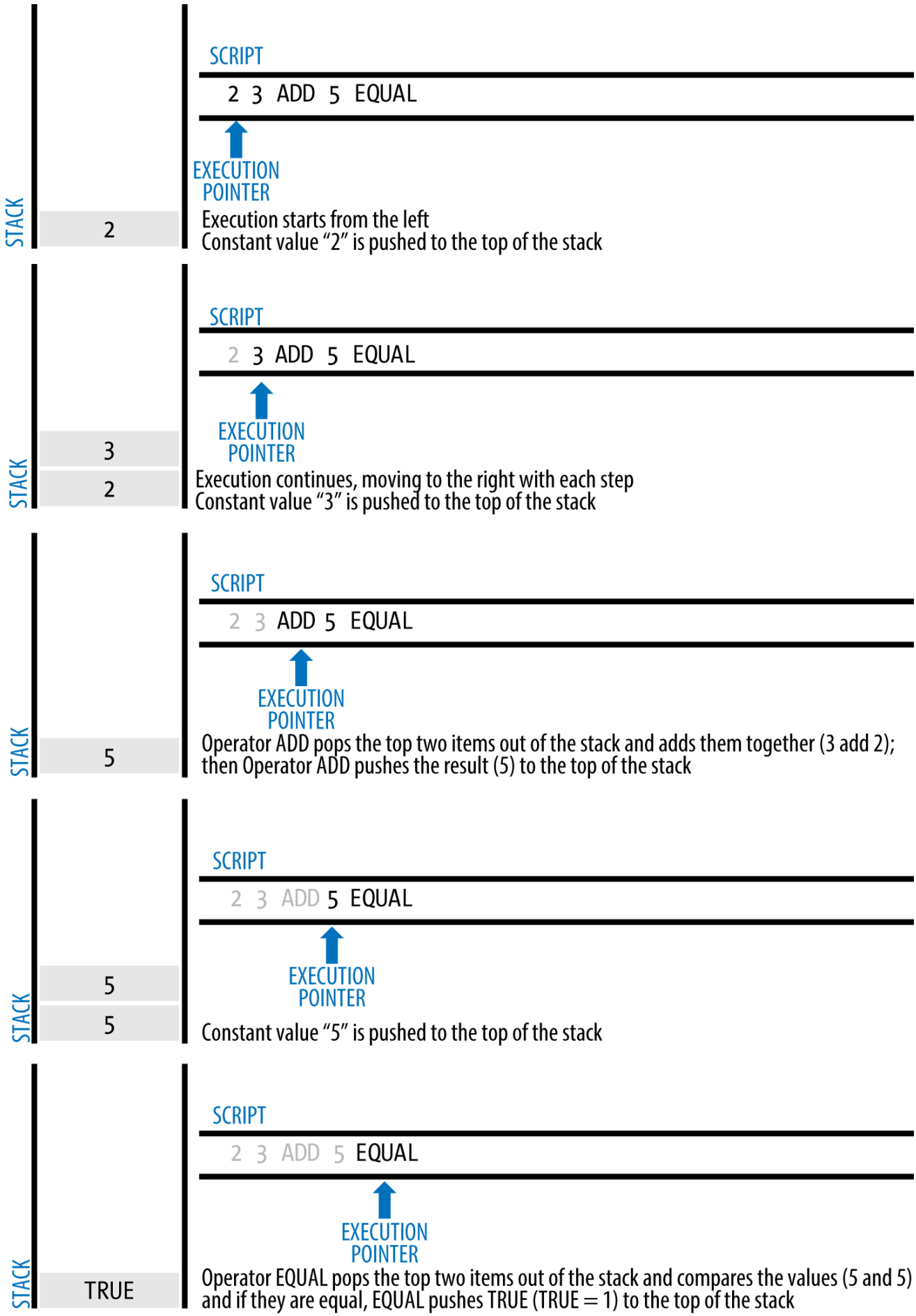
Начнем с того, что внутри Bitcoin существует свой собственный язык программирования, названный Script. Вот что о нем пишет Bitcoin wiki:

> Bitcoin uses a scripting system for transactions. Forth-like, Script is simple, stack-based, and processed from left to right. It is purposefully not Turing-complete, with no loops.

Суть в том, что язык прост как пробка, stack-based и Тьюринг-неполный. Вот пример типичной программы:

```
1 OP_DUP OP_DUP 5 OP_HASH160
```

Каждая инструкция называется *opcode* - всего их порядка 80, так что язык действительно довольно примитивен. На картинке ниже изображен процесс исполнения программы `2 3 OP_ADD 5 OP_EQUAL`:


SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
2
Execution starts from the left
Constant value "2" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
3
2
Execution continues, moving to the right with each step
Constant value "3" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
Operator ADD pops the top two items out of the stack and adds them together (3 add 2);
then Operator ADD pushes the result (5) to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
5
Constant value "5" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
TRUE
Operator EQUAL pops the top two items out of the stack and compares the values (5 and 5)
and if they are equal, EQUAL pushes TRUE (TRUE = 1) to the top of the stack

Lock & unlock transaction

Вернемся к языку чуть позже, а сначала давайте разберемся, зачем он здесь вообще нужен.
Для этого вспоминаем структуру транзакции и два параметра: *scriptSig* и *scriptPubKey*. В отличие от других параметров, назначение этих двух вообще не очевидно, и имхо это самое сложное, что есть в Bitcoin.

Я видел много попыток объяснить (как правило неудачных), что же из себя представляют скрипты в Bitcoin и как нужно их воспринимать на интуитивном уровне. Тем не менее я рискну и попробую привести еще одну аналогию. Для этого давайте рассмотрим *завещание*, вроде такого:

> 1.000.000$ переходят к Алисе только после того, как ей исполнится 18 лет

В этом случае, сам текст завещания - это некоторое условие, при котором можно воспользоваться деньгами (читай **можно воспользоваться транзакцией на 1.000.000$ как входом**), а ксерокопия паспорта в 19 лет - это доказательство того, что условие выполнено и самое время получить деньги.

Именно для того, чтобы **задать условие, при котором можно будет потратить выход, и для возможности подтвердить то, что условие выполнено** и нужен SCRIPT, приватные / публичные ключи и прочие сложности.

В случае Bitcoin, завещание - это *locking script*, который указывается в транзакции внутри поля pk_script. Его еще часто называют *scriptPubKey* из-за того, что чаще всего это программа, содержащая публичный ключ или адрес, хотя, вообще говоря, он может не иметь ничего общего с криптографией.

Своего рода "доказательство" того, что условие из *locking script* выполнено, называется *unlocking script*, пишется в поле signature script и часто называется *scriptSig*, догадайтесь почему.

Сам механизм проверки скрипта на валидность очень прост - для этого нужно соединить *unlocking script* + *locking script* и запустить получившуюся программу как одно целое. Если после исполнения, сверху стека останется `TRUE`, то транзакция валидна, и невалидна в любом другом случае.

Multiplication-based script

Скорее всего, вы ничего не поняли, поэтому давайте напишем какой-нибудь максимально простой скрипт, чтобы окончательно во всем разобраться. Идея состоит в том, чтобы заблокировать деньги с помощью какого-нибудь числа, например `370`. *Locking script* будет выглядеть как `OP_MUL 370 OP_EQUAL` и для того, чтобы *разблокировать* транзакцию, нужно будет указать два числа, дающие 370 в произведении.

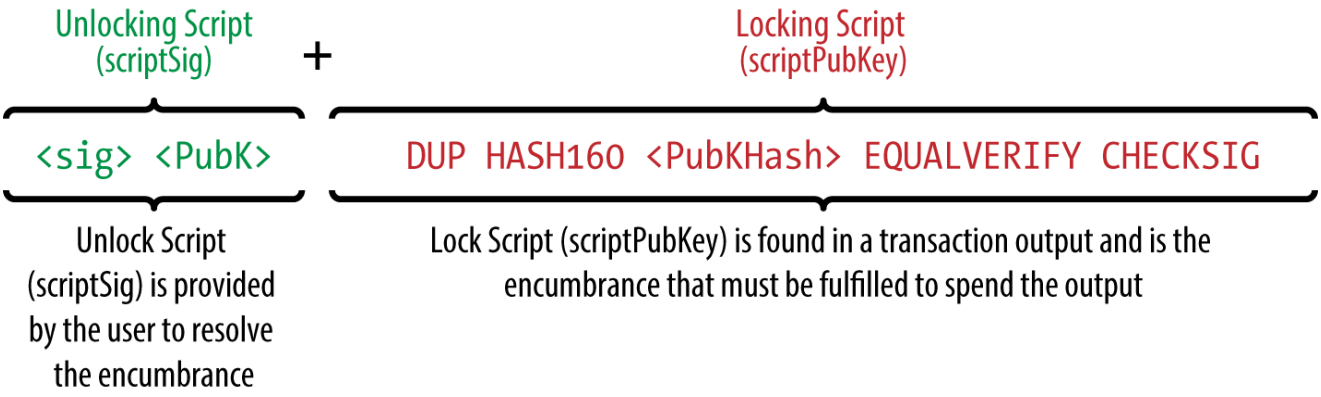
Для экспериментов со Script воспользуемся онлайн площадкой для запуска и дебага Bitcoin скриптов. В *unlocking script* запишем например `10 37`. Проверяем:

```
10 37 OP_MUL 370 OP_EQUAL
```

```
1  stack.push("10");
2  stack.push("37");
3  stack.OP_MUL();
4  stack.push("370");
5  stack.OP_EQUAL();
6  return stack.OP_VERIFY();
```

Pay to Public Key Hash (P2PKH)

P2PKH используется, наверное, в 99 транзакциях из 100, так что стоит понимать, как он работает. Вот его общий вид:

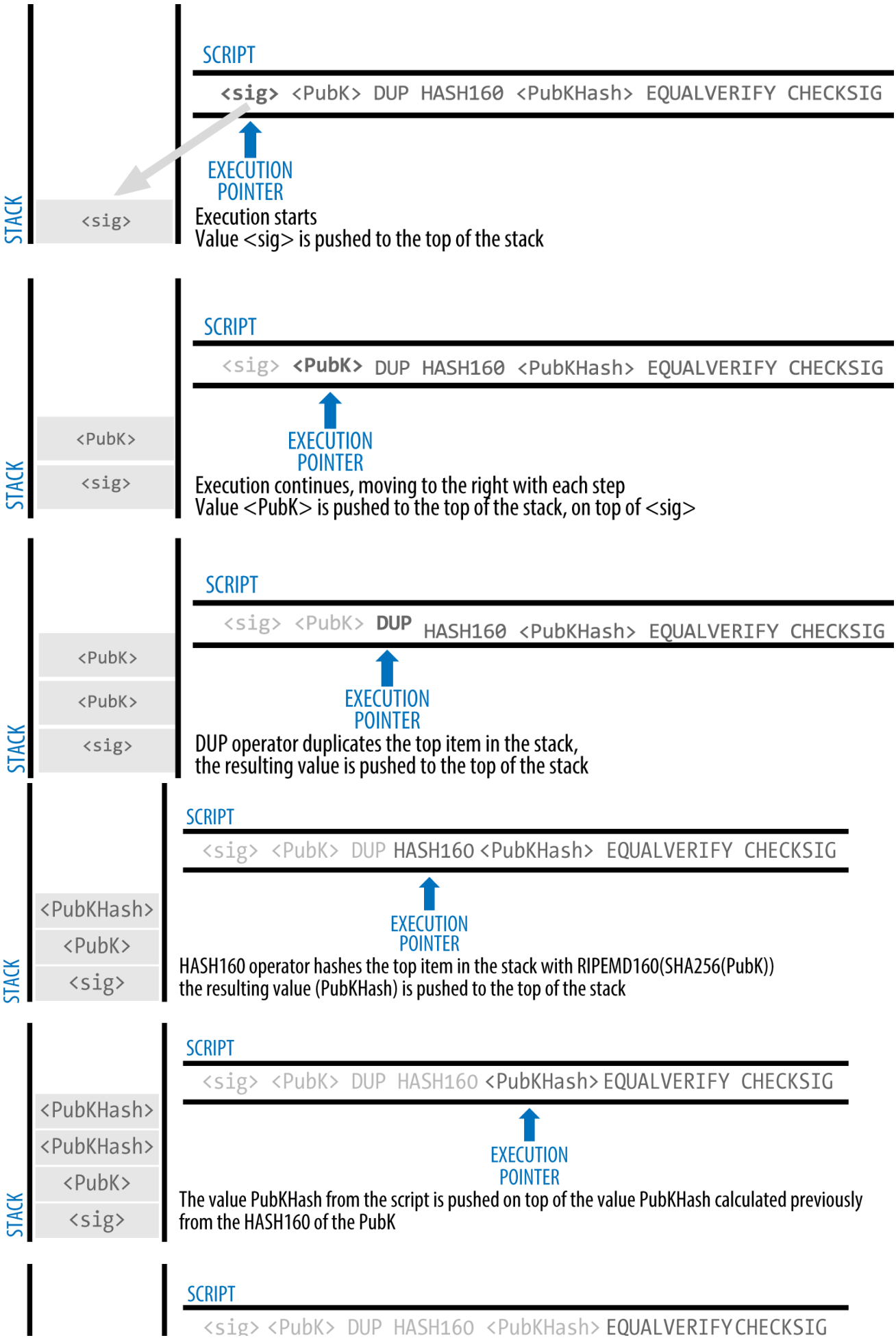


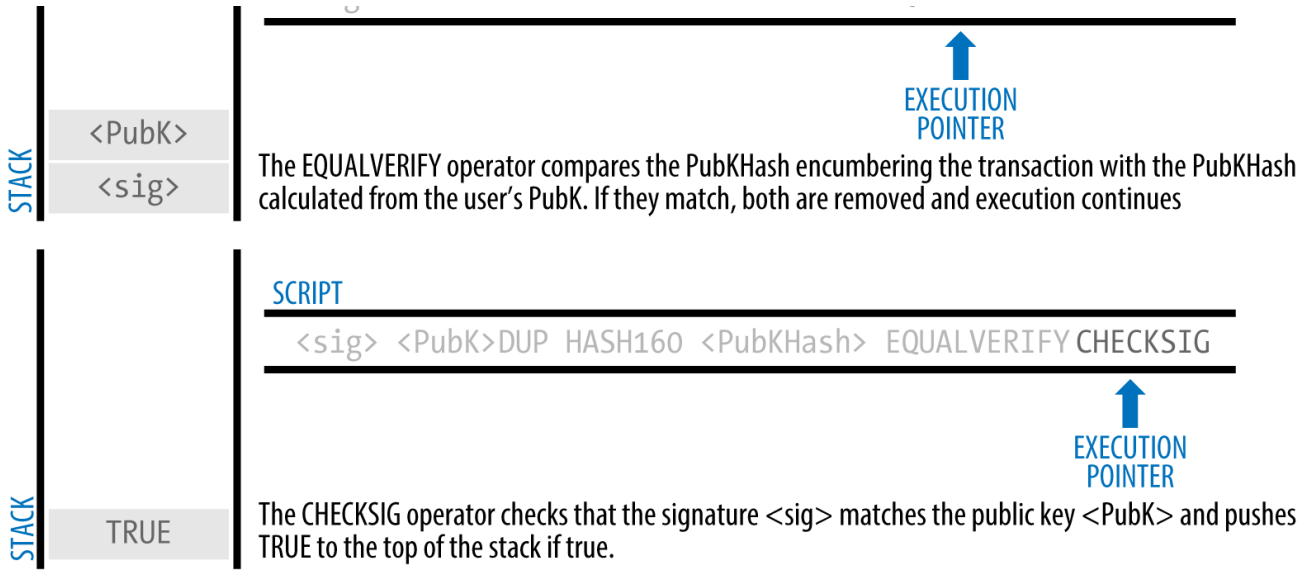
Этот скрипт известен с самого появления Bitcoin и, возможно, придуман самим Сатоши. Именно он выполняет ту задачу, о которой я писал выше: *сделать так, чтобы только владелец приватного ключа смог воспользоваться монетами, ассоциированными с адресом, полученным из этого ключа*.

На пальцах это выглядит следующим образом: пусть вашему другу **B** принадлежит приватный ключ **P**. Он получает из него публичный ключ **K**, адрес **A** и сообщает адрес вам. Далее вы отправляете на адрес **A** 1 BTC и в поле *locking script* пишите примерно следующее:

> Только тот, кто владеет приватным ключом для адреса **A**, сможет потратить эту транзакцию. В качестве доказательства запишите в unlocking script, во-первых, публичный ключ **K**, а во-вторых подпись свой транзакции приватным ключом **P**.

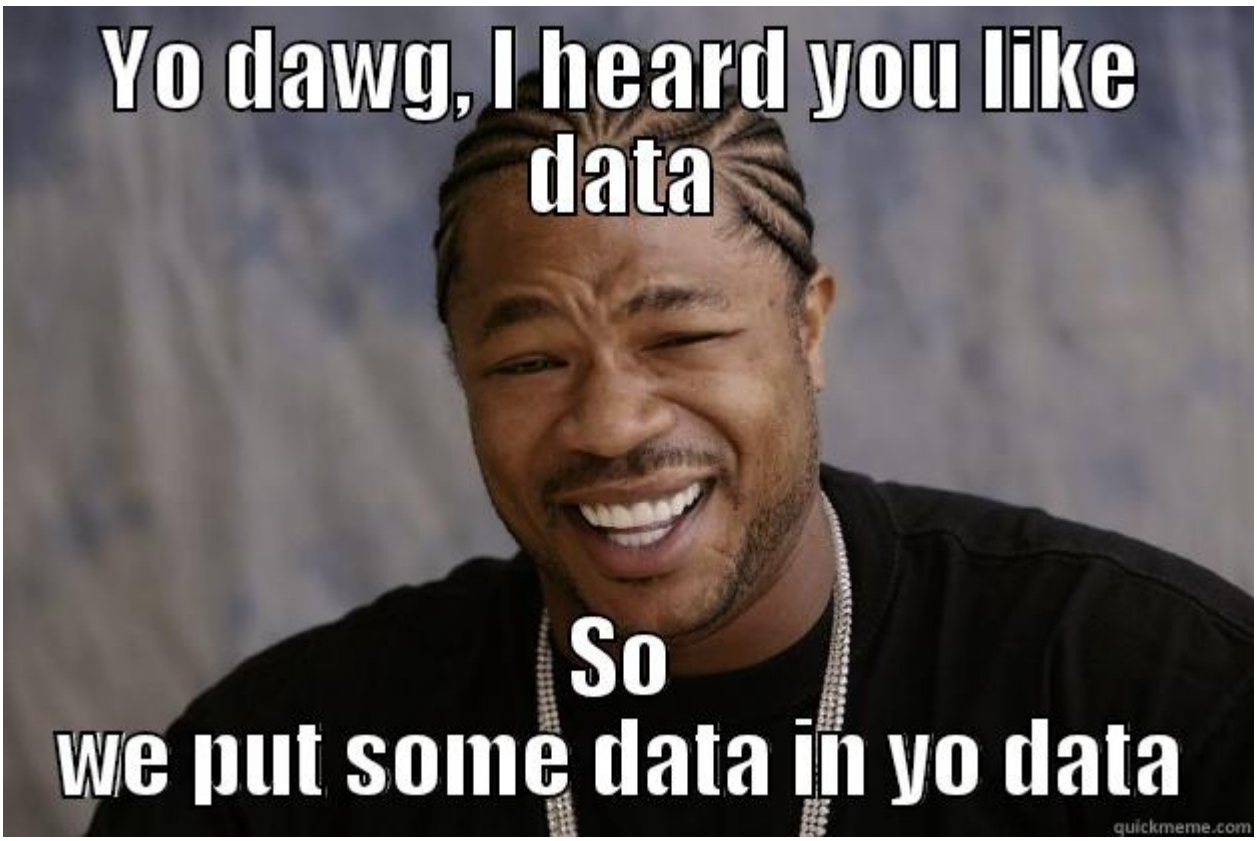
Когда **B** решит использовать вашу транзакцию в качестве входа, то он создаст свою, например, на 0.5 BTC, а в поле *unlocking script* вставит подпись своей транзакции приватным ключом **P** - `<sig>` и сам публичный ключ **K** - `<PubK>`.


SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<sig>
Execution starts
Value <sig> is pushed to the top of the stack
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubK>
<sig>
Execution continues, moving to the right with each step
Value <PubK> is pushed to the top of the stack, on top of <sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubK>
<PubK>
<sig>
DUP operator duplicates the top item in the stack,
the resulting value is pushed to the top of the stack
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubKHash>
<PubK>
<sig>
HASH160 operator hashes the top item in the stack with RIPEMD160(SHA256(PubK))
the resulting value (PubKHash) is pushed to the top of the stack
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubKHash>
<PubKHash>
<PubK>
<sig>
The value PubKHash from the script is pushed on top of the value PubKHash calculated previously
from the HASH160 of the PubK
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG



1. Подпись транзакции добавляется в стек
2. Публичный ключ добавляется в стек
3. `OP_DUP` берет верхний элемент стека и дублирует его, теперь в стеке сверху два публичных ключа
4. `OP_HASH160` заменяет верхний элемент стека на его хэш `RIPEMD160(SHA256(x))`
5. В стек добавляется такой же хэш публичного ключа, но уже посчитанный отправителем транзакции. Если вы внимательно читали Bitcoin in a nutshell - Cryptography, то для вас должно быть очевидно, что `RIPEMD160(SHA256(public_key))` и адрес - это в принципе одно и то же.
6. `OP_EQUALVERIFY` удаляет два верхних элемента стека, и если они не равны, то исполнение программы прерывается с ошибкой
7. `OP_CHECKSIG` проверяет подпись на соответствие транзакции. Если все верно, то удаляет подпись, удаляет публичный ключ и добавляет `TRUE`

P2P storage



Одно из самых интересных свойств Bitcoin, да и технологии блокчейн вообще, - это неизменяемость и гипотетическая "вечность" всего, что туда попадает. Неудивительно, что со временем нашлись люди, захотевшие использовать это в своих целях. И первое, что пришло им в голову - попытаться сохранить в блокчейн какие-нибудь сторонние данные и получить P2P дропбокс.

Я думаю вы уже поняли, как это делается. Берем строку `Make America great again` и просто записываем ее в *locking script*. Это все еще будет вполне корректный скрипт, другое дело, что к нему не получится придумать такой *unlocking script*, чтобы разблокировать средства. Но если вы отправите на выход с таким сриптом, условно говоря, 0.0000001 BTC, то в принципе и не жалко. Единственное ограничение - это размер вашей транзакции. Считайте, что она не может быть больше 100 КБ, хотя в реальности там все немного сложнее, можете почитать здесь.

Понятное дело, что такое положение дел по душе не всем. У Bitcoin и так большие проблемы с масштабируемостью, а тут еще и блокчейн, без того немаленький, начинает засоряться всякими левыми данными. Более того, помним, что такие транзакции нельзя потратить, а значит они навсегда останутся в *UTXO pool*, что ничуть не лучше.

Для того, чтобы достичь компромиса, был добавлен `OP_RETURN`, который позволяет "легально" хранить в блокчейне до 40 КБ данных.

> **OP_RETURN** is a script opcode used to mark a transaction output as invalid. Since the data after OP_RETURN are irrelevant to Bitcoin payments, arbitrary data can be added into the output after an OP_RETURN - Bitcoin wiki

Вот так выглядит простейший *locking script* с его участием: `OP_RETURN <40kb data>`. Что примечательно, выход с таким скриптом приобретает статус *provably unspendable*, то есть *доказуемо непотрачиваемый*. Из-за этого он даже не попадает в *UTXO pool*, тем самым экономя драгоценное место. Остальные причины использовать `OP_RETURN <data>` вместо `<data>` вы можете найти здесь.

Спойлер - их нет, если вы конечно не убежденный альтруист.

Links

- The Bitcoin Script language (pt. 1)
- The Bitcoin Script language (pt. 2)
- Standart scripts
- Explanation of what an OP_RETURN transaction looks like
- The Bitcoin Script Playground
- NPM bitcoin-script package