Pact Language Reference Documentation

Release 2.3.8

Stuart Popejoy

May 19, 2018

Contents

Contents:

CHAPTER 1

Pact Smart Contract Language Reference

This document is a reference for the Pact smart-contract language, designed for correct, transactional execution on a high-performance blockchain. For more background, please see the white paper or the pact home page.

CHAPTER 2

Rest API

As of version 2.1.0 Pact ships with a built-in HTTP server and SQLite backend. This allows for prototyping blockchain applications with just the `pact` tool.

To start up the server issue `pact -s config.yaml`, with a suitable config. The `pact-lang-api` JS library is available via npm for web development.

2.1 `cmd` field and "Stringified" Transaction JSON

Transactions sent into the blockchain must be hashed in order to ensure the received command is correct; this is also the value that is signed with the required private keys. To ensure the JSON for the transaction matches byte-for-byte with the value used to make the hash, the JSON must be *encoded* into the payload as a string (i.e., "stringified").

The *send*, *private*, and *local* endpoints support the `cmd` field to hold the executable code and data of the transaction as an encoded string. The format of the JSON to be encoded is as follows.

```
{
  "nonce": "[nonce value, needs to be unique for every call]",
  "payload": {
    "exec": "[pact code to be executed]",
    "data": {
       /* arbitrary user data to accompany code */
    }
  }
}
```

When assembling the message, this JSON should be "stringified" and provided for the `cmd` field. If you inspect the output of the *request formatter in the pact tool*, you will see that the `"cmd"` field is a String of encoded, escaped JSON.

2.2 Endpoints

All endpoints are served from `api/v1`. Thus a `send` call would be sent to (http://localhost:8080/api/v1/send){[}http://localhost:8080/api/v1/send], if running on `localhost:8080`.

2.2.1 /send

Asynchronous submit of one or more *public* (unencrypted) commands to the blockchain. See `` `cmd` field format <#cmd-field-and-stringified-transaction-json>`__ regarding the stringified JSON data.

Request JSON:

```
{
  "cmds": [
  {
    "hash": "[blake2 hash in base16 of 'cmd' string value]",
    "sigs": [
      {
        "sig": "[crypto signature by secret key of 'hash' value]",
        "pubKey": "[base16-format of public key of signing keypair]",
        "scheme": "ED25519" /* optional field, defaults to ED25519, will support␣
↪other curves as needed */
      }
    ]
    "cmd": "[stringified transaction JSON]"
  }
  // ... more commands
  ]
}
```

Response JSON:

```
{
  "status": "success|failure",
  "response": {
    "requestKeys": [
      "[matches hash from each sent/processed command, use with /poll or /listen to␣
↪get tx results]"
    ]
  }
}
```

2.2.2 /private

Asynchronous submit of one or more *private* commands to the blockchain, using supplied address info to securely encrypt for only sending and receiving entities to read. See `` `cmd` field format <#cmd-field-and-stringified-transaction-json>`__ regarding the stringified JSON data.

Request JSON:

```
{
  "cmds": [
  {
    "hash": "[blake2 hash in base16 of 'cmd' string value]",
```

(continues on next page)

(continued from previous page)

```
    "sigs": [
      {
        "sig": "[crypto signature by secret key of 'hash' value]",
        "pubKey": "[base16-format of public key of signing keypair]",
        "scheme": "ED25519" /* optional field, defaults to ED25519, will support␣
↪other curves as needed */
      }
    ]
    "cmd": "[stringified transaction JSON]"
  }
  ]
}
```

Response JSON:

```
{
  "status": "success|failure",
  "response": {
    "requestKeys": [
      "[matches hash from each sent/processed command, use with /poll or /listen to␣
↪get tx results]"
    ]
  }
}
```

2.2.3 /poll

Poll for command results.

Request JSON:

```
{
  "requestKeys": [
    "[hash from desired commands to poll]"
  ]
}
```

Response JSON:

```
{
  "status": "success|failure",
  "response": {
    "[command hash]": {
      "result": {
        "status": "success|failure",
        "data": /* data from Pact execution represented as JSON */
      },
      "txId": /* integer transaction id, for use in querying history etc */
    }
  }
}
```

2.2.4 /listen

Blocking call to listen for a single command result, or retrieve an already-executed command.

Request JSON:

```
{
  "listen": "[command hash]"
}
```

Response JSON:

```
{
  "status": "success|failure",
  "response": {
    "result": {
      "status": "success|failure",
      "data": /* data from Pact execution represented as JSON */
    },
    "txId": /* integer transaction id, for use in querying history etc */
  }
}
```

2.2.5 /local

Blocking/sync call to send a command for non-transactional execution. In a blockchain environment this would be a node-local "dirty read". Any database writes or changes to the environment are rolled back. See `cmd field format <#cmd-field-and-stringified-transaction-json>`__ regarding the stringified JSON data.

Request JSON:

```
{
  "hash": "[blake2 hash in base16 of 'cmd' value]",
  "sigs": [
    {
      "sig": "[crypto signature by secret key of 'hash' value]",
      "pubKey": "[base16-format of public key of signing keypair]",
      "scheme": "ED25519" /* optional field, defaults to ED25519, will support other␣
↪curves as needed */
    }
  ]
  "cmd": "[stringified transaction JSON]"
}
```

Response JSON:

```
{
  "status": "success|failure",
  "response": {
    "status": "success|failure",
    "data": /* data from Pact execution represented as JSON */
  }
}
```

2.3 API request formatter

As of Pact 2.2.3, the `pact` tool now accepts the `-a` option to format API request JSON, using a YAML file describing the request. The output can then be used with a POST tool like Postman or even piping into `curl`.

For instance, a yaml file called "apireq.yaml" with the following contents:

```
code: "(+ 1 2)"
data:
  name: Stuart
  language: Pact
keyPairs:
  - public: ba54b224d1924dd98403f5c751abdd10de6cd81b0121800bf7bdbdcfaec7388d
    secret: 8693e641ae2bbe9ea802c736f42027b03f86afe63cae315e7169c9c496c17332
```

can be fed into `pact` to obtain a valid API request:

```
$ pact -a tests/apireq.yaml -l
{"hash":
↪"444669038ea7811b90934f3d65574ef35c82d5c79cedd26d0931fddf837cccd2c9cf19392bf62c485f33535983f5e04c3e
↪","sigs":[{"sig":
↪"9097304baed4c419002c6b9690972e1303ac86d14dc59919bf36c785d008f4ad7efa3352ac2b8a47d0b688fe2909dbf392
↪","scheme":"ED25519","pubKey":
↪"ba54b224d1924dd98403f5c751abdd10de6cd81b0121800bf7bdbdcfaec7388d"}],"cmd":"{\
↪"address\":null,\"payload\":{\"exec\":{\"data\":{\"name\":\"Stuart\",\"language\":\
↪"Pact\"},\"code\":\"(+ 1 2)\"}},\"nonce\":\"\\\"2017-09-27 19:42:06.696533 UTC\\\"\
↪"}"}
```

Here's an example of piping into curl, hitting a pact server running on port 8080:

```
$ pact -a tests/apireq.yaml -l | curl -d @- http://localhost:8080/api/v1/local
{"status":"success","response":{"status":"success","data":3}}
```

2.3.1 Request YAML file format

The Request yaml takes the following keys:

```
code: Transaction code
codeFile: Transaction code file
data: JSON transaction data
dataFile: JSON transaction data file
keyPairs: list of key pairs for signing (use pact -g to generate): [
  public: base 16 public key
  secret: base 16 secret key
  ]
nonce: optional request nonce, will use current time if not provided
from: entity name for addressing private messages
to: entity names for addressing private messages
```

CHAPTER 3

Concepts

3.1 Execution Modes

Pact is designed to be used in distinct *execution modes* to address the performance requirements of rapid linear execution on a blockchain. These are:

1. Contract definition.
2. Transaction execution.
3. Queries and local execution.

3.1.1 Contract Definition

In this mode, a large amount of code is sent into the blockchain to establish the smart contract, as comprised of code (modules), tables (data), and keysets (authorization). This can also include "transactional" (database-modifying) code, for instance to initialize data.

For a given smart contract, these should all be sent as a single message into the blockchain, so that any error will rollback the entire smart contract as a unit.

Keyset definition

Keysets are customarily defined first, as they are used to specify admin authorization schemes for modules and tables. Definition creates the keysets in the runtime environment and stores their definition in the global keyset database.

Module declaration

Modules contain the API and data definitions for smart contracts. They are comprised of:

- *functions*
- *schema* definitions

- *table* definitions
- *"pact"* special functions
- *const* values

When a module is declared, all references to native functions or definitions from other modules are resolved. Resolution failure results in transaction rollback.

Modules can be re-defined as controlled by their admin keyset. Module versioning is not supported, except by including a version sigil in the module name (e.g., "accounts-v1"). However, *module hashes* are a powerful feature for ensuring code safety. When a module is imported with *use*, the module hash can be specified, to tie code to a particular release.

As of Pact 2.2, `use` statements can be issued within a module declaration. This combined with module hashes provides a high level of assurance, as updated module code will fail to import if a dependent module has subsequently changed on the chain; this will also propagate changes to the loaded modules' hash, protecting downstream modules from inadvertent changes on update.

Module names must be globally unique.

Table Creation

Tables are *created* at the same time as modules. While tables are *defined* in modules, they are *created* "after" modules, so that the module may be redefined later without having to necessarily re-create the table.

The relationship of modules to tables is important, as described in *Table Guards*.

There is no restriction on how many tables may be created. Table names are namespaced with the module name.

Tables can be typed with a *schema*.

3.1.2 Transaction Execution

"Transactions" refer to business events enacted on the blockchain, like a payment, a sale, or a workflow step of a complex contractual agreement. A transaction is generally a single call to a module function. However there is no limit on how many statements can be executed. Indeed, the difference between "transactions" and "smart contract definition" is simply the *kind* of code executed, not any actual difference in the code evaluation.

3.1.3 Queries and Local Execution

Querying data is generally not a business event, and can involve data payloads that could impact performance, so querying is carried out as a *local execution* on the node receiving the message. Historical queries use a *transaction ID* as a point of reference, to avoid any race conditions and allow asynchronous query execution.

Transactional vs local execution is accomplished by targeting different API endpoints; pact code has no ability to distinguish between transactional and local execution.

3.2 Database Interaction

Pact presents a database metaphor reflecting the unique requirements of blockchain execution, which can be adapted to run on different back-ends.

3.2.1 Atomic execution

A single message sent into the blockchain to be evaluated by Pact is *atomic*: the transaction succeeds as a unit, or does not succeed at all, known as "transactions" in database literature. There is no explicit support for rollback handling, except in *multi-step transactions*.

3.2.2 Key-Row Model

Blockchain execution can be likened to OLTP (online transaction processing) database workloads, which favor denormalized data written to a single table. Pact's data-access API reflects this by presenting a *key-row* model, where a row of column values is accessed by a single key.

As a result, Pact does not support *joining* tables, which is more suited for an OLAP (online analytical processing) database, populated from exports from the Pact database. This does not mean Pact cannot *record* transactions using relational techniques – for example, a Customer table whose keys are used in a Sales table would involve the code looking up the Customer record before writing to the Sales table.

3.2.3 Queries and Performance

As of Pact 2.3, Pact offers a powerful query mechanism for selecting multiple rows from a table. While visually similar to SQL, the *select* and *where* operations offer a *streaming interface* to a table, where the user provides filter functions, and then operates on the rowset as a list datastructure using *sort* and other functions.

```
;; the following selects Programmers with salaries >= 90000 and sorts by age␣
↪descending

(reverse (sort ['age]
  (select 'employees ['first-name,'last-name,'age]
    (and? (where 'title (= "Programmer"))
          (where 'salary (< 90000))))))

;; the same quert could be performed on a list with 'filter':

(reverse (sort ['age]
  (filter (and? (where 'title (= "Programmer"))
                (where 'salary (< 90000)))
          employees)))
```

In a transactional setting, Pact database interactions are optimized for single-row reads and writes, meaning such queries can be slow and prohibitively expensive computationally. However, using the *local* execution capability, Pact can utilize the user filter functions on the streaming results, offering excellent performance.

The best practice is therefore to use select operations via local, non-transactional operations, and avoid using select on large tables in the transactional setting.

3.2.4 No Nulls

Pact has no concept of a NULL value in its database metaphor. The main function for computing on database results, *with-read*, will error if any column value is not found. Authors must ensure that values are present for any transactional read. This is a safety feature to ensure *totality* and avoid needless, unsafe control-flow surrounding null values.

3.2.5 Versioned History

The key-row model is augmented by every change to column values being versioned by transaction ID. For example, a table with three columns "name", "age", and "role" might update "name" in transaction #1, and "age" and "role" in transaction #2. Retreiving historical data will return just the change to "name" under transaction 1, and the change to "age" and "role" in transaction #2.

3.2.6 Back-ends

Pact guarantees identical, correct execution at the smart-contract layer within the blockchain. As a result, the backing store need not be identical on different consensus nodes. Pact's implementation allows for integration of industrial RDBMSs, to assist large migrations onto a blockchain-based system, by facilitating bulk replication of data to downstream systems.

3.3 Types and Schemas

With Pact 2.0, Pact gains explicit type specification, albeit optional. Pact 1.0 code without types still functions as before, and writing code without types is attractive for rapid prototyping.

Schemas provide the main impetus for types. A schema *is defined* with a list of columns that can have types (although this is also not required). Tables are then *defined* with a particular schema (again, optional).

Note that schemas also can be used on/specified for object types.

3.3.1 Runtime Type enforcement

Any types declared in code are enforced at runtime. For table schemas, this means any write to a table will be typechecked against the schema. Otherwise, if a type specification is encountered, the runtime enforces the type when the expression is evaluated.

3.3.2 Static Type Inference on Modules

With the *typecheck* repl command, the Pact interpreter will analyze a module and attempt to infer types on every variable, function application or const definition. Using this in project repl scripts is helpful to aid the developer in adding "just enough types" to make the typecheck succeed. Fully successful typecheck is usually a matter of providing schemas for all tables, and argument types for ancilliary functions that call ambigious or overloaded native functions.

3.3.3 Formal Verification

Pact's typechecker is designed to output a fully typechecked, inlined AST for use generating formal proofs in SMT-LIB2. If the typecheck does not fully succeed, the module is not considered "provable".

We see, then, that Pact code can move its way up a "safety" gradient, starting with no types, then with "enough" types, and lastly, with formal proofs.

Note that as of Pact 2.0 the formal verification function is still under development.

3.4 Keysets and Authorization

Pact is inspired by Bitcoin scripts to incorporate public-key authorization directly into smart contract execution and administration.

3.4.1 Keyset definition

Keysets are *defined* by *reading* definitions from the message payload. Keysets consist of a list of public keys and a *keyset predicate*.

Examples of valid keyset JSON productions:

```
/* examples of valid keysets */
{
  "fully-specified":
    { "keys": ["abc6bab9b88e08d","fe04ddd404feac2"], "pred": "keys-2" }

  "keysonly":
    { "keys": ["abc6bab9b88e08d","fe04ddd404feac2"] } /* defaults to "keys-all" pred␣
↪*/

  "keylist": ["abc6bab9b88e08d","fe04ddd404feac2"] /* makes a "keys-all" pred keyset␣
↪*/
}
```

3.4.2 Keyset Predicates

A keyset predicate references a function by name which will compare the public keys in the keyset to the key or keys used to sign the blockchain message. The function accepts two arguments, "count" and "matched", where "count" is the number of keys in the keyset and "matched" is how many keys on the message signature matched a keyset key.

Support for multiple signatures is the responsibility of the blockchain layer, and is a powerful feature for Bitcoin-style "multisig" contracts (ie requiring at least two signatures to release funds).

Pact comes with built-in keyset predicates: *keys-all*, *keys-any*, *keys-2*. Module authors are free to define additional predicates.

If a keyset predicate is not specified, it is defaulted to *keys-all*.

3.4.3 Key rotation

Keysets can be rotated, but only by messages authorized against the current keyset definition and predicate. Once authorized, the keyset can be easily *redefined*.

3.4.4 Module Table Guards

When *creating* a table, a module name must also be specified. By this mechanism, tables are "guarded" or "encapsulated" by the module, such that direct access to the table via *data-access functions* is authorized by the module's admin keyset. However, *within module functions*, table access is unconstrained. This gives contract authors great flexibility in designing data access, and is intended to enshrine the module as the main "user" data access API.

3.4.5 Row-level keysets

Keysets can be stored as a column value in a row, allowing for *row-level* authorization. The following code indicates how this might be achieved:

```
(defun create-account (id)
  (insert accounts id { "balance": 0.0, "keyset": (read-keyset "owner-keyset") }))

(defun read-balance (id)
  (with-read accounts id { "balance":= bal, "keyset":= ks }
    (enforce-keyset ks)
    (format "Your balance is {}" [bal])))
```

In the example, `create-account` reads a keyset definition from the message payload using *read-keyset* to store as "keyset" in the table. `read-balance` only allows that owner's keyset to read the balance, by first enforcing the keyset using *enforce-keyset*.

3.5 Computational Model

Here we cover various aspects of Pact's approach to computation.

3.5.1 Turing-Incomplete

Pact is turing-incomplete, in that there is no recursion (recursion is detected before execution and results in an error) and no ability to loop indefinitely. Pact does support operation on list structures via *map*, *fold* and *filter*, but since there is no ability to define infinite lists, these are necessarily bounded.

Turing-incompleteness allows Pact module loading to resolve all references in advance, meaning that instead of addressing functions in a lookup table, the function definition is directly injected (or "inlined") into the callsite. This is an example of the performance advantages of a Turing-incomplete language.

3.5.2 Single-assignment Variables

Pact allows variable declarations in *let expressions* and *bindings*. Variables are immutable: they cannot be re-assigned, or modified in-place.

A common variable declaration occurs in the *with-read* function, assigning variables to column values by name. The *bind* function offers this same functionality for objects.

Module-global constant values can be declared with *defconst*.

3.5.3 Data Types

Pact code can be explicitly typed, and is always strongly-typed under the hood as the native functions perform strict typechecking as indicated in their documented type signatures. language, but does use fixed type representations "under the hood" and does no coercion of types, so is strongly-typed nonetheless.

Pact's supported types are:

- *Strings*
- *Integers*
- *Decimals*

- *Booleans*
- *Key sets*
- *Lists*
- *Objects*
- *Function* and *pact* definitions
- *JSON values*
- *Tables*
- *Schemas*

3.5.4 Performance

Pact is designed to maximize the performance of *transaction execution*, penalizing queries and module definition in favor of fast recording of business events on the blockchain. Some tips for fast execution are:

Single-function transactions

Design transactions so they can be executed with a single function call.

Call with references instead of `use`

When calling module functions in transactions, use *reference syntax* instead of importing the module with *use*. When defining modules that reference other module functions, `use` is fine, as those references will be inlined at module definition time.

Hardcoded arguments vs. message values

A transaction can encode values directly into the transactional code:

```
(accounts.transfer "Acct1" "Acct2" 100.00)
```

or it can read values from the message JSON payload:

```
(defun transfer-msg ()
  (transfer (read-msg "from") (read-msg "to")
            (read-decimal "amount")))
...
(accounts.transfer-msg)
```

The latter will execute slightly faster, as there is less code to interpret at transaction time.

Types as necessary

With table schemas, Pact will be strongly typed for most use cases, but functions that do not use the database might still need types. Use the typecheck REPL function to add the necessary types. There is a small cost for type enforcement at runtime, and too many type signatures can harm readability. However types can help document an API, so this is a judgement call.

3.5.5 Control Flow

Pact supports conditionals via *if*, bounded looping, and of course function application.

"If" considered harmful

Consider avoiding `if` wherever possible: every branch makes code harder to understand and more prone to bugs. The best practice is to put "what am I doing" code in the front-end, and "validate this transaction which I intend to succeed" code in the smart contract.

Pact's original design left out `if` altogether (and looping), but it was decided that users should be able to judiciously use these features as necessary.

Use enforce

"If" should never be used to enforce business logic invariants: instead, *enforce* is the right choice, which will fail the transaction.

Indeed, failure is the only *non-local exit* allowed by Pact. This reflects Pact's emphasis on *totality*.

Note that *enforce-one* (added in Pact 2.3) allows for testing a list of enforcements such that if any pass, the whole expression passes. This is the sole example in Pact of "exception catching" in that a failed enforcement simply results in the next test being executed, short-circuiting on success.

Use built-in keysets

The built-in keyset functions *keys-all*, *keys-any*, *keys-2* are hardcoded in the interpreter to execute quickly. Custom keysets require runtime resolution which is slower.

3.5.6 Functional Concepts

Pact includes the functional-programming "greatest hits": *map*, *fold* and *filter*. These all employ *partial application*, where the list item is appended onto the application arguments in order to serially execute the function.

```
(map (+ 2) [1 2 3])
(fold (+) ["Concatenate" " " "me"]
```

Pact also has *compose*, which allows "chaining" applications in a functional style.

3.5.7 Pure execution

In certain contexts Pact can guarantee that computation is "pure", which simply means that the database state will not be accessed or modified. Currently, `enforce`, `enforce-one` and keyset predicate evaluation are all executed in a pure context. *defconst* memoization is also pure.

3.5.8 LISP

Pact's use of LISP syntax is intended to make the code reflect its runtime representation directly, allowing contract authors focus directly on program execution. Pact code is stored in human-readable form on the ledger, such that the code can be directly verified, but the use of LISP-style *s-expression syntax* allows this code to execute quickly.

3.5.9 Message Data

Pact expects code to arrive in a message with a JSON payload and signatures. Message data is read using *read-msg* and related functions, while signatures are not directly readable or writable – they are evaluated as part of *keyset predicate* enforcement.

JSON support

Values returned from Pact transactions are expected to be directly represented as JSON values.

When reading values from a message via *read-msg*, Pact coerces JSON types as follows:

- String -> String
- Number -> Integer (rounded)
- Boolean -> Boolean
- Object -> Object
- Array -> List
- Null -> JSON Value

Decimal values are represented as Strings and read using *read-decimal*.

3.6 Confidentiality

Pact is designed to be used in a *confidentiality-preserving* environment, where messages are only visible to a subset of participants. This has significant implications for smart contract execution.

3.6.1 Entities

An *entity* is a business participant that is able or not able to see a confidential message. An entity might be a company, a group within a company, or an individual.

3.6.2 Disjoint Databases

Pact smart contracts operate on messages organized by a blockchain, and serve to produce a database of record, containing results of transactional executions. In a confidential environment, different entities execute different transactions, meaning the resulting databases are now *disjoint*.

This does not affect Pact execution; however, database data can no longer enact a "two-sided transaction", meaning we need a new concept to handle enacting a single transaction over multiple disjoint datasets.

3.6.3 Confidential Pacts

An important feature for confidentiality in Pact is the ability to orchestrate disjoint transactions in sequence to be executed by targeted entities. This is described in the next section.

3.7 Asynchronous Transaction Automation with "Pacts"

"Pacts" are multi-stage sequential transactions that are defined as a single body of code called a *pact*. Definining a multi-step interaction as a pact ensures that transaction participants will enact an agreed sequence of operations, and offers a special "execution scope" that can be used to create and manage data resources only during the lifetime of a given multi-stage interaction.

Pacts are a form of *coroutine*, which is a function that has multiple exit and re-entry points. Pacts are composed of *steps* such that only a single step is executed in a given blockchain transaction. Steps can only be executed in strict sequential order.

A pact is defined with arguments, similarly to function definition. However, arguments values are only evaluated in the execution of the initial step, after which those values are available unchanged to subsequent steps. To share new values with subsequent steps, a step can *yield* values which the subsequent step can recover using the special *resume* binding form.

Pacts are designed to run in one of two different contexts, private and public. A private pact is indicated by each step identifying a single entity to execute the step, while public steps do not have entity indicators. A pact can only be uniformly public or private: if some steps has entity indicators and others do not, this results in an error at load time.

3.7.1 Public Pacts

Public pacts are comprised of steps that can only execute in strict sequence. Any enforcement of who can execute a step happens within the code of the step expression. All steps are "manually" initiated by some participant in the transaction with RESUME commands sent into the blockchain.

3.7.2 Private Pacts

Private pacts are comprised of steps that execute in sequence where each step only executes on entity nodes as selected by the provided 'entity' argument in the step; other entity nodes "skip" the step. Private pacts are executed automatically by the blockchain platform after the initial step is sent in, with the executing entity's node automatically sending the RESUME command for the next step.

3.7.3 Failures, Rollbacks and Cancels

Failure handling is dramatically different in public and private pacts.

In public pacts, a rollback expression is specified to indicate that the pact can be "cancelled" at this step with a partipant sending in a CANCEL message before the next step is executed. Failures in public steps are no different than a failure in a non-pact transaction: all changes are rolled back. Pacts can therefore only be canceled explicitly and should be modeled to offer all necessary cancel options.

In private pacts, the sequential execution of steps is automated by the blockchain platform itself. A failure results in a ROLLBACK message being sent from the executing entity node which will trigger any rollback expression specified in the previous step, to be executed by that step's entity. This failure will then "cascade" to the previous step as a new ROLLBACK transaction, completing when the first step is rolled back.

3.7.4 Yield and Resume

A step can yield values to the following step using *yield* and *resume*. In public, this is an unforgeable value as it is maintained within the blockchain pact scope. In private this is simply a value sent with a RESUME message from the executed entity.

3.7.5 Pact execution scope and `pact-id`

Every time a pact is initiated, it is given a unique ID which is retrievable using the *pact-id* function, which will return the ID of the currently executing pact, or fail if not running within a pact scope. This mechanism can thus be used to guard access to resources, analogous to the use of keysets and signatures. The classic use of this is to create escrow accounts that can only be used within the context of a given pact, eliminating the need for a trusted third party for many use-cases.

3.7.6 Testing pacts

Pacts can be tested in repl scripts using the *env-entity*, *env-step* and *pact-state* repl functions to simulate pact executions.

It is not possible yet (as of Pact 2.3.0) to simulate pact execution in the pact server API.

3.8 Dependency Management

Pact supports a number of features to manage a module's dependencies on other Pact modules.

3.8.1 Module Hashes

Once loaded, a Pact module is associated with a hash computed from the module's source code text. This module hash uniquely identifies the version of the module. Module hashes can be examined with *describe-module*:

```
pact> (at "hash" (describe-module 'accounts))

↪"9d6f4d3acb2fd528206330d09a8926da6abdd9ac5e8c4b24cc35955203f234688c25f9545ead56f783c5269fe4be6a62a
↪"
```

3.8.2 Pinning module versions with `use`

The *use* special form allows a module hash to be specified, in order to pin the dependency version. When used within a module declaration, it introduces the dependency hash value into the module's hash. This allows a "dependency-only" upgrade to push the upgrade to the module version.

3.8.3 Inlined Dependencies: "No Leftpad"

Pact inlines all user-code references when a module is loaded, meaning that upstream definitions are injected into downstream code. At this point, upstream definitions are permanent: the only way to upgrade dependencies is to re-load the module code.

This permanence is great for downstream/client code: the upstream provider cannot change what code gets executed in your module, once loaded. It creates a big problem for upstream/provider code, as providers cannot upgrade the downstream code to address an exploit, or to introduce new features.

3.8.4 Blessing hashes

A trade-off is needed to balance these opposing interests. Pact offers the ability for upstream code to break downstream dependent code at runtime. Table access is guarded to enforce that the module hash of the inlined dependency either matches the runtime version, or is in a set of "blessed" hashes, as specified by *bless* in the module declaration:

```
(module provider 'keyset
  (bless
↪"e4cfa39a3d37be31c59609e807970799caa68a19bfaa15135f165085e01d41a65ba1e1b146aeb6bd0092b49eac214c103
↪")
  (bless
↪"ca002330e69d3e6b84a46a56a6533fd79d51d97a3bb7cad6c2ff43b354185d6dc1e723fb3db4ae0737e120378424c714b
↪")
  ...
)
```

Dependencies with these hashes will continue to function after the module is loaded. Unrecognized hashes will cause the transaction to fail. However, "pure" code that does not access the database are unaffected. This prevents a "leftpad situation" where trivial utility functions cannot harm downstream code stability.

3.8.5 Phased upgrades with "v2" modules

Upstream providers can use the bless mechanism to phase-in an important upgrade, by renaming the upgraded module to indicate the new version, and replacing the old module with a new, empty module that only blesses the last version (and whatever earlier versions desired). New clients will fail to import the "v1" code, requiring them to use the new version, while existing users can continue to use the old version, presumably up to some advertised time limit. The "empty" module can offer migration functions to handle migrating user data to the new module, for the user to self-upgrade in the time window.

CHAPTER 4

Syntax

4.1 Literals

4.1.1 Strings

String literals are created with double-ticks:

```
pact> "a string"
"a string"
```

Strings also support multiline by putting a backslash before and after whitespace (not interactively).

```
(defun id (a)
  "Identity function. \
  \Argument is returned."
  a)
```

4.1.2 Symbols

Symbols are string literals representing some unique item in the runtime, like a function or a table name. Their representation internally is simply a string literal so their usage is idiomatic.

Symbols are created with a preceding tick, thus they do no support whitespace or multiline.

```
pact> 'a-symbol
"a-symbol"
```

4.1.3 Integers

Integer literals are unbounded positive naturals. For negative numbers use the unary - function.

```
pact> 12345
12345
```

4.1.4 Decimals

Decimal literals are positive decimals to exact expressed precision.

```
pact> 100.25
100.25
pact> 356452.23451872
356452.23451872
```

4.1.5 Booleans

Booleans are represented by `true` and `false` literals.

```
pact> (and true false)
false
```

4.1.6 Lists

List literals are created with brackets. Uniform literal lists are given a type in parsing.

```
pact> [1 2 3]
[1 2 3]
pact> (typeof [1 2 3])
"[integer]"
pact> (typeof [1 2 true])
"list"
```

4.1.7 Objects

Objects are dictionaries, created with curly-braces specifying key-value pairs using a colon `:`. For certain applications (database updates), keys must be strings.

```
pact> { "foo": (+ 1 2), "bar": "baz" }
(TObject [("foo",3),("bar","baz")])
```

4.1.8 Bindings

Bindings are dictionary-like forms, also created with curly braces, to bind database results to variables using the `:=` operator. They are used in *with-read*, *with-default-read*, *bind* and *resume* to assign variables to named columns in a row, or values in an object.

```
(defun check-balance (id)
  (with-read accounts id { "balance" := bal }
    (enforce (> bal 0) (format "Account in overdraft: {}" [bal]))))
```

4.2 Type specifiers

Types can be specified in syntax with the colon `:` operator followed by a type literal or user type specification.

4.2.1 Type literals

- `string`
- `integer`
- `decimal`
- `bool`
- `time`
- `keyset`
- `list`, or `[type]` to specify the list type
- `object`, which can be further typed with a schema
- `table`, which can be further typed with a schema
- `value` (JSON values)

4.2.2 Schema type literals

A schema defined with *defschema* is referenced by name enclosed in curly braces.

```
table:{accounts}
object:{person}
```

4.2.3 What can be typed

Function arguments and return types

```
(defun prefix:string (pfx:string str:string) (+ pfx str))
```

Let variables

```
(let ((a:integer 1) (b:integer 2)) (+ a b))
```

Tables and objects

Tables and objects can only take a schema type literal.

```
(deftable accounts:{account})

(defun get-order:{order} (id) (read orders id))
```

Consts

```
(defconst PENNY:decimal 0.1)
```

4.3 Special forms

4.3.1 bless

```
(bless HASH)
```

Within a module declaration, bless a previous version of that module as identified by HASH. See *Dependency management* for a discussion of the blessing mechanism.

```
(module provider 'keyset
  (bless
↪"e4cfa39a3d37be31c59609e807970799caa68a19bfaa15135f165085e01d41a65ba1e1b146aeb6bd0092b49eac214c103
↪")
  (bless
↪"ca002330e69d3e6b84a46a56a6533fd79d51d97a3bb7cad6c2ff43b354185d6dc1e723fb3db4ae0737e120378424c714bb
↪")
  ...
)
```

4.3.2 defun

```
(defun NAME ARGLIST [DOCSTRING] BODY...)
```

Define NAME as a function, accepting ARGLIST arguments, with optional DOCSTRING. Arguments are in scope for BODY, one or more expressions.

```
(defun add3 (a b c) (+ a (+ b c)))

(defun scale3 (a b c s) "multiply sum of A B C times s"
  (* s (add3 a b c)))
```

4.3.3 defconst

```
(defun NAME VALUE [DOCSTRING])
```

Define NAME as VALUE, with option DOCSTRING. Value is evaluated upon module load and "memoized".

```
(defconst COLOR_RED="#FF0000" "Red in hex")
(defconst COLOR_GRN="#00FF00" "Green in hex")
(defconst PI 3.14159265 "Pi to 8 decimals")
```

4.3.4 defpact

```
(defpact NAME ARGLIST [DOCSTRING] STEPS...)
```

Define NAME as a *pact*, a multistep computation intended for private transactions. Identical to *defun* except body must be comprised of *steps* to be executed in strict sequential order. Steps must uniformly be "public" (no entity indicator) or "private" (with entity indicator). With private steps, failures result in a reverse-sequence "rollback cascade".

```
(defpact payment (payer payer-entity payee
                  payee-entity amount)
  (step-with-rollback payer-entity
    (debit payer amount)
    (credit payer amount))
  (step payee-entity
    (credit payee amount)))
```

4.3.5 defschema

```
(defschema NAME [DOCSTRING] FIELDS...)
```

Define NAME as a *schema*, which specifies a list of FIELDS. Each field is in the form `FIELDNAME[:FIELDTYPE]`.

```
(defschema accounts
  "Schema for accounts table".
  balance:decimal
  amount:decimal
  ccy:string
  data)
```

4.3.6 deftable

```
(deftable NAME[:SCHEMA] [DOCSTRING])
```

Define NAME as a *table*, used in database functions. Note the table must still be created with *create-table*.

4.3.7 let

```
(let (BINDPAIR [BINDPAIR [...]]) BODY)
```

Bind variables in BINDPAIRs to be in scope over BODY. Variables within BINDPAIRs cannot refer to previously-declared variables in the same let binding; for this use *let**.

```
(let ((x 2)
      (y 5))
  (* x y))
> 10
```

4.3.8 let*

```
(let\* (BINDPAIR [BINDPAIR [...]]) BODY)
```

Bind variables in BINDPAIRs to be in scope over BODY. Variables can reference previously declared BINDPAIRS in the same let. `let\*` is expanded at compile-time to nested `let` calls for each BINDPAIR; thus `let` is preferred where possible.

```
(let* ((x 2)
       (y (* x 10)))
  (+ x y))
> 22
```

4.3.9 step

```
(step EXPR)
(step ENTITY EXPR)
```

Define a step within a *defpact* such that any prior steps will be executed in prior transactions, and later steps in later transactions. With ENTITY, indicates that this step is intended for confidential transactions such that only ENTITY will execute the step, while other participants will "skip" the step. in order of execution specified in containing *defpact*.

4.3.10 step-with-rollback

```
(step-with-rollback EXPR ROLLBACK-EXPR)
(step-with-rollback ENTITY EXPR ROLLBACK-EXPR)
```

Define a step within a *defpact* similarly to *step* but specifying ROLLBACK-EXPR. With ENTITY, ROLLBACK-EXPR will only be executed upon failure of a subsequent step, as part of a reverse-sequence "rollback cascade" going back from the step that failed to the first step. Without ENTITY, ROLLBACK-EXPR functions as a "cancel function" to be explicitly executed by a participant.

4.3.11 use

```
(use MODULE)
(use MODULE HASH)
```

Import an existing MODULE into namespace. Can only be issued at top-level, or within a module declaration. MODULE can be a string, symbol or bare atom. With HASH, validate that module hash matches HASH, failing if not. Use *describe-module* to query for the hash of a loaded module on the chain.

```
(use accounts)
(transfer "123" "456" 5 (time "2016-07-22T11:26:35Z"))
"Write succeeded"
```

4.3.12 module

```
(module NAME KEYSET [DOCSTRING] DEFS...)
```

Define and install module NAME, guarded by keyset KEYSET, with optional DOCSTRING. DEFS must be *defun* or *defpact* expressions only.

```
(module accounts 'accounts-admin
  "Module for interacting with accounts"

  (defun create-account (id bal)
   "Create account ID with initial balance BAL"
   (insert accounts id { "balance": bal }))

  (defun transfer (from to amount)
   "Transfer AMOUNT from FROM to TO"
   (with-read accounts from { "balance": fbal }
    (enforce (<= amount fbal) "Insufficient funds")
     (with-read accounts to { "balance": tbal }
      (update accounts from { "balance": (- fbal amount) })
      (update accounts to { "balance": (+ tbal amount) }))))
)
```

4.4 Expressions

Expressions may be *literals*, atoms, s-expressions, or references.

4.4.1 Atoms

Atoms are non-reserved barewords starting with a letter or allowed symbol, and containing letters, digits and allowed symbols. Allowed symbols are `%#+-_&$@<>=?*!|/`. Atoms must resolve to a variable bound by a *defun*, *defpact*, *binding* form, or to symbols imported into the namespace with *use*.

4.4.2 S-expressions

S-expressions are formed with parentheses, with the first atom determining if the expression is a *special form* or a function application, in which case the first atom must refer to a definition.

Partial application

An application with less than the required arguments is in some contexts a valid *partial application* of the function. However, this is only supported in Pact's *functional-style functions*; anywhere else this will result in a runtime error.

4.4.3 References

References are two atoms joined by a dot `.` to directly resolve to module definitions.

```
pact> accounts.transfer
"(defun accounts.transfer (src,dest,amount,date) \"transfer AMOUNT from
SRC to DEST\")"
pact> transfer
Eval failure:
transfer<EOF>: Cannot resolve transfer
pact> (use 'accounts)
```

(continues on next page)

(continued from previous page)

```
"Using \"accounts\""
pact> transfer
"(defun accounts.transfer (src,dest,amount,date) \"transfer AMOUNT from
SRC to DEST\")"
```

References are preferred to `use` for transactions, as references resolve faster. However in module definition, `use` is preferred for legibility.

CHAPTER 5

Time formats

Pact leverages the Haskell thyme library for fast computation of time values. The *parse-time* and *format-time* functions accept format codes that derive from GNU `strftime` with some extensions, as follows:

`%%` - literal `"%"`

`%z` - RFC 822/ISO 8601:1988 style numeric time zone (e.g., `"-0600"` or `"+0100"`)

`%N` - ISO 8601 style numeric time zone (e.g., `"-06:00"` or `"+01:00"`) /EXTENSION/

`%Z` - timezone name

`%c` - The preferred calendar time representation for the current locale. As 'dateTimeFmt' `locale` (e.g. `%a %b %e %H:%M:%S %Z %Y`)

`%R` - same as `%H:%M`

`%T` - same as `%H:%M:%S`

`%X` - The preferred time of day representation for the current locale. As 'timeFmt' `locale` (e.g. `%H:%M:%S`)

`%r` - The complete calendar time using the AM/PM format of the current locale. As 'time12Fmt' `locale` (e.g. `%I:%M:%S %p`)

`%P` - day-half of day from ('amPm' `locale`), converted to lowercase, `"am"`, `"pm"`

`%p` - day-half of day from ('amPm' `locale`), `"AM"`, `"PM"`

`%H` - hour of day (24-hour), 0-padded to two chars, `"00"-"23"`

`%k` - hour of day (24-hour), space-padded to two chars, `" 0"-"23"`

`%I` - hour of day-half (12-hour), 0-padded to two chars, `"01"-"12"`

`%l` - hour of day-half (12-hour), space-padded to two chars, `" 1"-"12"`

`%M` - minute of hour, 0-padded to two chars, `"00"-"59"`

`%S` - second of minute (without decimal part), 0-padded to two chars, `"00"-"60"`

`%v` - microsecond of second, 0-padded to six chars, `"000000"-"999999"`. /EXTENSION/

`%Q` - decimal point and fraction of second, up to 6 second decimals, without trailing zeros. For a whole number of seconds, `%Q` produces the empty string. /EXTENSION/

`%s` - number of whole seconds since the Unix epoch. For times before the Unix epoch, this is a negative number. Note that in `%s.%q` and `%s%Q` the decimals are positive, not negative. For example, 0.9 seconds before the Unix epoch is formatted as `"-1.1"` with `%s%Q`.

`%D` - same as `%m\/%d\/%y`

`%F` - same as `%Y-%m-%d`

`%x` - as 'dateFmt' `locale` (e.g. `%m\/%d\/%y`)

`%Y` - year, no padding.

`%y` - year of century, 0-padded to two chars, `"00"-"99"`

`%C` - century, no padding.

`%B` - month name, long form ('fst' from 'months' `locale`), `"January"-"December"`

`%b`, `%h` - month name, short form ('snd' from 'months' `locale`), `"Jan"-"Dec"`

`%m` - month of year, 0-padded to two chars, `"01"-"12"`

`%d` - day of month, 0-padded to two chars, `"01"-"31"`

`%e` - day of month, space-padded to two chars, `" 1"-"31"`

`%j` - day of year, 0-padded to three chars, `"001"-"366"`

`%G` - year for Week Date format, no padding.

`%g` - year of century for Week Date format, 0-padded to two chars, `"00"-"99"`

`%f` - century for Week Date format, no padding. /EXTENSION/

`%V` - week of year for Week Date format, 0-padded to two chars, `"01"-"53"`

`%u` - day of week for Week Date format, `"1"-"7"`

`%a` - day of week, short form ('snd' from 'wDays' `locale`), `"Sun"-"Sat"`

`%A` - day of week, long form ('fst' from 'wDays' `locale`), `"Sunday"-"Saturday"`

`%U` - week of year where weeks start on Sunday (as 'sundayStartWeek'), 0-padded to two chars, `"00"-"53"`

`%w` - day of week number, `"0"` (= Sunday) – `"6"` (= Saturday)

`%W` - week of year where weeks start on Monday (as 'Data.Thyme.Calendar.WeekdayOfMonth.mondayStartWeek'), 0-padded to two chars, `"00"-"53"`

Note: `%q` (picoseconds, zero-padded) does not work properly so not documented here.

5.1 Default format and JSON serialization

The default format is a UTC ISO8601 date+time format: "%Y-%m-%dT%H:%M:%SZ", as accepted by the *time* function. While the time object internally supports up to microsecond resolution, values returned from the Pact interpreter as JSON will be serialized with the default format. When higher resolution is desired, explicitly format times with `%v` and related.

5.2 Examples

5.2.1 ISO8601

```
pact> (format-time "%Y-%m-%dT%H:%M:%S%N" (time "2016-07-23T13:30:45Z"))
"2016-07-23T13:30:45+00:00"
```

5.2.2 RFC822

```
pact> (format-time "%a, %_d %b %Y %H:%M:%S %Z" (time "2016-07-23T13:30:45Z"))
"Sat, 23 Jul 2016 13:30:45 UTC"
```

5.2.3 YYYY-MM-DD hh:mm:ss.000000

```
> (format-time "%Y-%m-%d %H:%M:%S.%v" (add-time (time "2016-07-23T13:30:45Z") 0.
↪001002))
"2016-07-23 13:30:45.001002"
```

CHAPTER 6

Built-in Functions

6.1 General

6.1.1 at

idx `integer` *list* `[<l>]` → `<a>`

idx `string` *object* `object:<{o}>` → `<a>`

Index LIST at IDX, or get value with key IDX from OBJECT.

```
pact> (at 1 [1 2 3])
2
pact> (at "bar" { "foo": 1, "bar": 2 })
2
```

6.1.2 bind

src `object:<{row}>` *binding* `binding:<{row}>` → `<a>`

Special form evaluates SRC to an object which is bound to with BINDINGS over subsequent body statements.

```
pact> (bind { "a": 1, "b": 2 } { "a" := a-value } a-value)
1
```

6.1.3 compose

x `(x:<a> -> <b>)` *y* `(x:<b> -> <c>)` *value* `<a>` → `<c>`

Compose X and Y, such that X operates on VALUE, and Y on the results of X.

```
pact> (filter (compose (length) (< 2)) ["my" "dog" "has" "fleas"])
["dog" "has" "fleas"]
```

6.1.4 constantly

value <a> *ignore1* <b> → <a>

value <a> *ignore1* <b> *ignore2* <c> → <a>

value <a> *ignore1* <b> *ignore2* <c> *ignore3* <d> → <a>

Ignore (lazily) arguments IGNORE* and return VALUE.

```
pact> (filter (constantly true) [1 2 3])
[1 2 3]
```

6.1.5 contains

value <a> *list* `[<a>]` → `bool`

key <a> *object* `object:<{o}>` → `bool`

value `string` *string* `string` → `bool`

Test that LIST or STRING contains VALUE, or that OBJECT has KEY entry.

```
pact> (contains 2 [1 2 3])
true
pact> (contains 'name { 'name: "Ted", 'age: 72 })
true
pact> (contains "foo" "foobar")
true
```

6.1.6 drop

count `integer` *list* `<a[[<l>],string]>` → `<a[[<l>],string]>`

keys `[string]` *object* `object:<{o}>` → `object:<{o}>`

Drop COUNT values from LIST (or string), or entries having keys in KEYS from OBJECT. If COUNT is negative, drop from end.

```
pact> (drop 2 "vwxyz")
"xyz"
pact> (drop (- 2) [1 2 3 4 5])
[1 2 3]
pact> (drop ['name] { 'name: "Vlad", 'active: false})
{"active": false}
```

6.1.7 enforce

test `bool` *msg* `string` → `bool`

Fail transaction with MSG if pure function TEST fails, or returns true.

```
pact> (enforce (!= (+ 2 2) 4) "Chaos reigns")
<interactive>:0:0: Chaos reigns
```

6.1.8 enforce-one

msg `string` *tests* `[bool]` → `bool`

Run TESTS in order (in pure context, plus keyset enforces). If all fail, fail transaction. Short-circuits on first success.

```
pact> (enforce-one "Should succeed on second test" [(enforce false "Skip me")␣
↪(enforce (= (+ 2 2) 4) "Chaos reigns")])
true
```

6.1.9 enforce-pact-version

min-version `string` → `bool`

min-version `string` *max-version* `string` → `bool`

Enforce runtime pact version as greater than or equal MIN-VERSION, and less than or equal MAX-VERSION. Version values are matched numerically from the left, such that '2', '2.2', and '2.2.3' would all allow '2.2.3'.

```
pact> (enforce-pact-version "2.3")
true
```

6.1.10 filter

app `(x:<a> -> bool)` *list* `[<a>]` → `[<a>]`

Filter LIST by applying APP to each element to get a boolean determining inclusion.

```
pact> (filter (compose (length) (< 2)) ["my" "dog" "has" "fleas"])
["dog" "has" "fleas"]
```

6.1.11 fold

app `(x:<b> y:<a> -> <a>)` *init* `<a>` *list* `[<b>]` → `<a>`

Iteratively reduce LIST by applying APP to last result and element, starting with INIT.

```
pact> (fold (+) 0 [100 10 5])
115
```

6.1.12 format

template `string` *vars* `list` → `string`

Interpolate VARS into TEMPLATE using {}.

```
pact> (format "My {} has {}" ["dog" "fleas"])
"My dog has fleas"
```

6.1.13 hash

value <a> → string

Compute BLAKE2b 512-bit hash of VALUE. Strings are converted directly while other values are converted using their JSON representation.

```
pact> (hash "hello")

↪"e4cfa39a3d37be31c59609e807970799caa68a19bfaa15135f165085e01d41a65ba1e1b146aeb6bd0092b49eac214c103c
↪"
pact> (hash { 'foo: 1 })

↪"61d3c8775e151b4582ca7f9a885a9b2195d5aa6acc58ddca61a504e9986bb8c06eeb37af722ad848f9009053b6379677bf
↪"
```

6.1.14 identity

value <a> → <a>

Return provided value.

```
pact> (map (identity) [1 2 3])
[1 2 3]
```

6.1.15 if

cond bool *then* <a> *else* <a> → <a>

Test COND, if true evaluate THEN, otherwise evaluate ELSE.

```
pact> (if (= (+ 2 2) 4) "Sanity prevails" "Chaos reigns")
"Sanity prevails"
```

6.1.16 length

x <a[[<l>],string,object:<{o}>]> → integer

Compute length of X, which can be a list, a string, or an object.

```
pact> (length [1 2 3])
3
pact> (length "abcdefgh")
8
pact> (length { "a": 1, "b": 2 })
2
```

6.1.17 list

elems * → list

Create list from ELEMS. Deprecated in Pact 2.1.1 with literal list support.

```
pact> (list 1 2 3)
[1 2 3]
```

6.1.18 list-modules

→ `[string]`

List modules available for loading.

6.1.19 make-list

length `integer` *value* `<a>` → `[<a>]`

Create list by repeating VALUE LENGTH times.

```
pact> (make-list 5 true)
[true true true true true]
```

6.1.20 map

app `(x:<b> -> <a>)` *list* `[<b>]` → `[<a>]`

Apply elements in LIST as last arg to APP, returning list of results.

```
pact> (map (+ 1) [1 2 3])
[2 3 4]
```

6.1.21 pact-id

→ `integer`

Return ID if called during current pact execution, failing if not.

6.1.22 pact-version

→ `string`

Obtain current pact build version.

```
pact> (pact-version)
"2.3.8"
```

6.1.23 read-decimal

key `string` → `decimal`

Parse KEY string or number value from top level of message data body as decimal.

```
(defun exec ()
   (transfer (read-msg "from") (read-msg "to") (read-decimal "amount")))
```

6.1.24 read-integer

key `string` → `integer`

Parse KEY string or number value from top level of message data body as integer.

```
(read-integer "age")
```

6.1.25 read-msg

→ `<a>`

key `string` → `<a>`

Read KEY from top level of message data body, or data body itself if not provided. Coerces value to pact type: String -> string, Number -> integer, Boolean -> bool, List -> value, Object -> value. NB value types are not introspectable in pact.

```
(defun exec ()
   (transfer (read-msg "from") (read-msg "to") (read-decimal "amount")))
```

6.1.26 remove

key `string` *object* `object:<{o}>` → `object:<{o}>`

Remove entry for KEY from OBJECT.

```
pact> (remove "bar" { "foo": 1, "bar": 2 })
{"foo": 1}
```

6.1.27 resume

binding `binding:<{y}>` *body* `*` → `<a>`

Special form binds to a yielded object value from the prior step execution in a pact.

6.1.28 reverse

l `[<a>]` → `[<a>]`

Reverse a list.

```
pact> (reverse [1 2 3])
[3 2 1]
```

6.1.29 sort

values `[<a>]` → `[<a>]`

fields `[string]` *values* `[object:<{o}>]` → `[object:<{o}>]`

Sort monotyped list of primitive VALUES, or objects using supplied FIELDS list.

```
pact> (sort [3 1 2])
[1 2 3]
pact> (sort ['age] [{'name: "Lin",'age: 30} {'name: "Val",'age: 25}])
[{"name": "Val", "age": 25} {"name": "Lin", "age": 30}]
```

6.1.30 take

count `integer` *list* `<a[[<l>],string]>` → `<a[[<l>],string]>`

keys `[string]` *object* `object:<{o}>` → `object:<{o}>`

Take COUNT values from LIST (or string), or entries having keys in KEYS from OBJECT. If COUNT is negative, take from end.

```
pact> (take 2 "abcd")
"ab"
pact> (take (- 3) [1 2 3 4 5])
[3 4 5]
pact> (take ['name] { 'name: "Vlad", 'active: false})
{"name": "Vlad"}
```

6.1.31 tx-hash

→ `string`

Obtain hash of current transaction as a string.

```
pact> (tx-hash)

↪"786a02f742015903c6c6fd852552d272912f4740e15847618a86e217f71f5419d25e1031afee585313896444934eb04b90
↪"
```

6.1.32 typeof

x `<a>` → `string`

Returns type of X as string.

```
pact> (typeof "hello")
"string"
```

6.1.33 where

field `string` *app* `(x:<a> -> bool)` *value* `object:<{row}>` → `bool`

Utility for use in 'filter' and 'select' applying APP to FIELD in VALUE.

```
pact> (filter (where 'age (> 20)) [{'name: "Mary",'age: 30} {'name: "Juan",'age: 15}])
[{"name": "Juan", "age": 15}]
```

6.1.34 yield

OBJECT `object:<{y}>` → `object:<{y}>`

Yield OBJECT for use with 'resume' in following pact step. The object is similar to database row objects, in that only the top level can be binded to in 'resume'; nested objects are converted to opaque JSON values.

```
(yield { "amount": 100.0 })
```

6.2 Database

6.2.1 create-table

table `table:<{row}>` → `string`

Create table TABLE.

```
(create-table accounts)
```

6.2.2 describe-keyset

keyset `string` → `value`

Get metadata for KEYSET

6.2.3 describe-module

module `string` → `value`

Get metadata for MODULE. Returns an object with 'name', 'hash', 'blessed', and 'code' fields.

6.2.4 describe-table

table `string` → `value`

Get metadata for TABLE

6.2.5 insert

table `table:<{row}>` *key* `string` *object* `object:<{row}>` → `string`

Write entry in TABLE for KEY of OBJECT column data, failing if data already exists for KEY.

```
(insert 'accounts { "balance": 0.0, "note": "Created account." })
```

6.2.6 keylog

table `table:<{row}>` *key* `string` *txid* `integer` → `[object]`

Return updates to TABLE for a KEY in transactions at or after TXID, in a list of objects indexed by txid.

```
(keylog 'accounts "Alice" 123485945)
```

6.2.7 keys

table `table:<{row}>` → `[string]`

Return all keys in TABLE.

```
(keys 'accounts)
```

6.2.8 read

table `table:<{row}>` *key* `string` → `object:<{row}>`

table `table:<{row}>` *key* `string` *columns* `[string]` → `object:<{row}>`

Read row from TABLE for KEY returning database record object, or just COLUMNS if specified.

```
(read 'accounts id ['balance 'ccy])
```

6.2.9 select

table `table:<{row}>` *where* `(row:object:<{row}> -> bool)` → `[object:<{row}>]`

table `table:<{row}>` *columns* `[string]` *where* `(row:object:<{row}> -> bool)` → `[object:<{row}>]`

Select full rows or COLUMNS from table by applying WHERE to each row to get a boolean determining inclusion.

```
(select people ['firstName,'lastName] (where 'name (= "Fatima")))
(select people (where 'age (> 30)))
```

6.2.10 txids

table `table:<{row}>` *txid* `integer` → `[integer]`

Return all txid values greater than or equal to TXID in TABLE.

```
(txids 'accounts 123849535)
```

6.2.11 txlog

table `table:<{row}>` *txid* `integer` → `[value]`

Return all updates to TABLE performed in transaction TXID.

```
(txlog 'accounts 123485945)
```

6.2.12 update

table `table:<{row}>` *key* `string` *object* `object:<{row}>` → `string`

Write entry in TABLE for KEY of OBJECT column data, failing if data does not exist for KEY.

```
(update 'accounts { "balance": (+ bal amount), "change": amount, "note": "credit" })
```

6.2.13 with-default-read

table `table:<{row}>` *key* `string` *defaults* `object:<{row}>` *bindings* `binding:<{row}>` → <a>

Special form to read row from TABLE for KEY and bind columns per BINDINGS over subsequent body statements. If row not found, read columns from DEFAULTS, an object with matching key names.

```
(with-default-read 'accounts id { "balance": 0, "ccy": "USD" } { "balance":= bal, "ccy
↪":= ccy }
   (format "Balance for {} is {} {}" [id bal ccy]))
```

6.2.14 with-read

table `table:<{row}>` *key* `string` *bindings* `binding:<{row}>` → <a>

Special form to read row from TABLE for KEY and bind columns per BINDINGS over subsequent body statements.

```
(with-read 'accounts id { "balance":= bal, "ccy":= ccy }
   (format "Balance for {} is {} {}" [id bal ccy]))
```

6.2.15 write

table `table:<{row}>` *key* `string` *object* `object:<{row}>` → `string`

Write entry in TABLE for KEY of OBJECT column data.

```
(write 'accounts { "balance": 100.0 })
```

6.3 Time

6.3.1 add-time

time `time` *seconds* `decimal` → `time`

time `time` *seconds* `integer` → `time`

Add SECONDS to TIME; SECONDS can be integer or decimal.

```
pact> (add-time (time "2016-07-22T12:00:00Z") 15)
"2016-07-22T12:00:15Z"
```

6.3.2 days

n `decimal` → `decimal`

n `integer` → `decimal`

N days, for use with 'add-time'

```
pact> (add-time (time "2016-07-22T12:00:00Z") (days 1))
"2016-07-23T12:00:00Z"
```

6.3.3 diff-time

time1 `time` *time2* `time` → `decimal`

Compute difference between TIME1 and TIME2 in seconds.

```
pact> (diff-time (parse-time "%T" "16:00:00") (parse-time "%T" "09:30:00"))
23400
```

6.3.4 format-time

format `string` *time* `time` → `string`

Format TIME using FORMAT. See *"Time Formats" docs* for supported formats.

```
pact> (format-time "%F" (time "2016-07-22T12:00:00Z"))
"2016-07-22"
```

6.3.5 hours

n `decimal` → `decimal`

n `integer` → `decimal`

N hours, for use with 'add-time'

```
pact> (add-time (time "2016-07-22T12:00:00Z") (hours 1))
"2016-07-22T13:00:00Z"
```

6.3.6 minutes

n `decimal` → `decimal`

n `integer` → `decimal`

N minutes, for use with 'add-time'.

```
pact> (add-time (time "2016-07-22T12:00:00Z") (minutes 1))
"2016-07-22T12:01:00Z"
```

6.3.7 parse-time

format `string` *utcval* `string` → `time`

Construct time from UTCVAL using FORMAT. See *"Time Formats" docs* for supported formats.

```
pact> (parse-time "%F" "2016-09-12")
"2016-09-12T00:00:00Z"
```

6.3.8 time

utcval `string` → `time`

Construct time from UTCVAL using ISO8601 format (%Y-%m-%dT%H:%M:%SZ).

```
pact> (time "2016-07-22T11:26:35Z")
"2016-07-22T11:26:35Z"
```

6.4 Operators

6.4.1 !=

x `<a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset]>` *y* `<a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset]>` → `bool`

True if X does not equal Y.

```
pact> (!= "hello" "goodbye")
true
```

6.4.2 *

x `<a[integer,decimal]>` *y* `<a[integer,decimal]>` → `<a[integer,decimal]>`

x `<a[integer,decimal]>` *y* `<b[integer,decimal]>` → `decimal`

Multiply X by Y.

```
pact> (* 0.5 10.0)
5
pact> (* 3 5)
15
```

6.4.3 +

x `<a[integer,decimal]>` *y* `<a[integer,decimal]>` → `<a[integer,decimal]>`

x `<a[integer,decimal]>` *y* `<b[integer,decimal]>` → `decimal`

x `<a[string,[<l>],object:<{o}>]>` *y* `<a[string,[<l>],object:<{o}>]>` → `<a[string,[<l>],object:<{o}>]>`

Add numbers, concatenate strings/lists, or merge objects.

```
pact> (+ 1 2)
3
pact> (+ 5.0 0.5)
5.5
pact> (+ "every" "body")
"everybody"
pact> (+ [1 2] [3 4])
[1 2 3 4]
pact> (+ { "foo": 100 } { "foo": 1, "bar": 2 })
{"bar": 2, "foo": 100}
```

6.4.4 -

x <a[integer,decimal]> *y* <a[integer,decimal]> → <a[integer,decimal]>

x <a[integer,decimal]> *y* <b[integer,decimal]> → decimal

x <a[integer,decimal]> → <a[integer,decimal]>

Negate X, or subtract Y from X.

```
pact> (- 1.0)
-1.0
pact> (- 3 2)
1
```

6.4.5 /

x <a[integer,decimal]> *y* <a[integer,decimal]> → <a[integer,decimal]>

x <a[integer,decimal]> *y* <b[integer,decimal]> → decimal

Divide X by Y.

```
pact> (/ 10.0 2.0)
5
pact> (/ 8 3)
2
```

6.4.6 <

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X < Y.

```
pact> (< 1 3)
true
pact> (< 5.24 2.52)
false
pact> (< "abc" "def")
true
```

6.4.7 <=

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X <= Y.

```
pact> (<= 1 3)
true
pact> (<= 5.24 2.52)
false
pact> (<= "abc" "def")
true
```

6.4.8 =

x <a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset]> *y* <a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset]> → bool

True if X equals Y.

```
pact> (= [1 2 3] [1 2 3])
true
pact> (= 'foo "foo")
true
pact> (= { 1: 2 } { 1: 2})
true
```

6.4.9 >

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X > Y.

```
pact> (> 1 3)
false
pact> (> 5.24 2.52)
true
pact> (> "abc" "def")
false
```

6.4.10 >=

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X >= Y.

```
pact> (>= 1 3)
false
pact> (>= 5.24 2.52)
true
pact> (>= "abc" "def")
false
```

6.4.11 ^

x <a[integer,decimal]> *y* <a[integer,decimal]> → <a[integer,decimal]>

x <a[integer,decimal]> *y* <b[integer,decimal]> → decimal

Raise X to Y power.

```
pact> (^ 2 3)
8
```

6.4.12 abs

x decimal → decimal

x integer → integer

Absolute value of X.

```
pact> (abs (- 10 23))
13
```

6.4.13 and

x bool *y* bool → bool

Boolean logic with short-circuit.

```
pact> (and true false)
false
```

6.4.14 and? {#and?}

a (x:<r> -> bool) *b* (x:<r> -> bool) *value* <r> → bool

Apply logical 'and' to the results of applying VALUE to A and B, with short-circuit.

```
pact> (and? (> 20) (> 10) 15)
false
```

6.4.15 ceiling

x decimal *prec* integer → decimal

x decimal → integer

Rounds up value of decimal X as integer, or to PREC precision as decimal.

```
pact> (ceiling 3.5)
4
pact> (ceiling 100.15234 2)
100.16
```

6.4.16 exp

x <a[integer,decimal]> → <a[integer,decimal]>

Exp of X

```
pact> (round (exp 3) 6)
20.085537
```

6.4.17 floor

x decimal *prec* integer → decimal

x decimal → integer

Rounds down value of decimal X as integer, or to PREC precision as decimal.

```
pact> (floor 3.5)
3
pact> (floor 100.15234 2)
100.15
```

6.4.18 ln

x <a[integer,decimal]> → <a[integer,decimal]>

Natural log of X.

```
pact> (round (ln 60) 6)
4.094345
```

6.4.19 log

x <a[integer,decimal]> *y* <a[integer,decimal]> → <a[integer,decimal]>

x <a[integer,decimal]> *y* <b[integer,decimal]> → decimal

Log of Y base X.

```
pact> (log 2 256)
8
```

6.4.20 mod

x integer *y* integer → integer

X modulo Y.

```
pact> (mod 13 8)
5
```

6.4.21 not

x `bool` → `bool`

Boolean logic.

```
pact> (not (> 1 2))
true
```

6.4.22 not? {#not?}

app `(x:<r> -> bool)` *value* `<r>` → `bool`

Apply logical 'not' to the results of applying VALUE to APP.

```
pact> (not? (> 20) 15)
false
```

6.4.23 or

x `bool` *y* `bool` → `bool`

Boolean logic with short-circuit.

```
pact> (or true false)
true
```

6.4.24 or? {#or?}

a `(x:<r> -> bool)` *b* `(x:<r> -> bool)` *value* `<r>` → `bool`

Apply logical 'or' to the results of applying VALUE to A and B, with short-circuit.

```
pact> (or? (> 20) (> 10) 15)
true
```

6.4.25 round

x `decimal` *prec* `integer` → `decimal`

x `decimal` → `integer`

Performs Banker's rounding value of decimal X as integer, or to PREC precision as decimal.

```
pact> (round 3.5)
4
pact> (round 100.15234 2)
100.15
```

6.4.26 sqrt

x `<a[integer,decimal]>` → `<a[integer,decimal]>`

Square root of X.

```
pact> (sqrt 25)
5
```

6.5 Keysets

6.5.1 define-keyset

name `string` *keyset* `string` → `string`

Define keyset as NAME with KEYSET. If keyset NAME already exists, keyset will be enforced before updating to new value.

```
(define-keyset 'admin-keyset (read-keyset "keyset"))
```

6.5.2 enforce-keyset

keyset-or-name `<k[string,keyset]>` → `bool`

Special form to enforce KEYSET-OR-NAME against message keys before running BODY. KEYSET-OR-NAME can be a symbol of a keyset name or a keyset object.

```
(with-keyset 'admin-keyset ...)
(with-keyset (read-keyset "keyset") ...)
```

6.5.3 keys-2

count `integer` *matched* `integer` → `bool`

Keyset predicate function to match at least 2 keys in keyset.

```
pact> (keys-2 3 1)
false
```

6.5.4 keys-all

count `integer` *matched* `integer` → `bool`

Keyset predicate function to match all keys in keyset.

```
pact> (keys-all 3 3)
true
```

6.5.5 keys-any

count `integer` *matched* `integer` → `bool`

Keyset predicate function to match any (at least 1) key in keyset.

```
pact> (keys-any 10 1)
true
```

6.5.6 read-keyset

key `string` → `keyset`

Read KEY from message data body as keyset ({ "keys": KEYLIST, "pred": PREDFUN }). PREDFUN should resolve to a keys predicate.

```
(read-keyset "admin-keyset")
```

6.6 REPL-only functions

The following functions are loaded magically in the interactive REPL, or in script files with a `.repl` extension. They are not available for blockchain-based execution.

6.6.1 begin-tx

→ `string`

name `string` → `string`

Begin transaction with optional NAME.

```
(begin-tx "load module")
```

6.6.2 bench

exprs * → `string`

Benchmark execution of EXPRS.

```
(bench (+ 1 2))
```

6.6.3 commit-tx

→ `string`

Commit transaction.

```
(commit-tx)
```

6.6.4 env-data

json `<a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset,value]>` → `string`

Set transaction JSON data, either as encoded string, or as pact types coerced to JSON.

```
pact> (env-data { "keyset": { "keys": ["my-key" "admin-key"], "pred": "keys-any" } })
"Setting transaction data"
```

6.6.5 env-entity

→ `string`

entity `string` → `string`

Set environment confidential ENTITY id, or unset with no argument. Clears any previous pact execution state.

```
(env-entity "my-org")
(env-entity)
```

6.6.6 env-hash

hash `string` → `string`

Set current transaction hash. HASH must be a valid BLAKE2b 512-bit hash.

```
pact> (env-hash (hash "hello"))
"Set tx hash to␣
↪e4cfa39a3d37be31c59609e807970799caa68a19bfaa15135f165085e01d41a65ba1e1b146aeb6bd0092b49eac214c103cc
↪"
```

6.6.7 env-keys

keys `[string]` → `string`

Set transaction signature KEYS.

```
pact> (env-keys ["my-key" "admin-key"])
"Setting transaction keys"
```

6.6.8 env-step

→ `string`

step-idx `integer` → `string`

step-idx `integer` *rollback* `bool` → `string`

step-idx `integer` *rollback* `bool` *resume* `object:<{y}>` → `string`

Set pact step state. With no arguments, unset step. With STEP-IDX, set step index to execute. ROLLBACK instructs to execute rollback expression, if any. RESUME sets a value to be read via 'resume'.Clears any previous pact execution state.

```
(env-step 1)
(env-step 0 true)
```

6.6.9 expect

doc `string` *expected* <a> *actual* <a> → `string`

Evaluate ACTUAL and verify that it equals EXPECTED.

```
pact> (expect "Sanity prevails." 4 (+ 2 2))
"Expect: success: Sanity prevails."
```

6.6.10 expect-failure

doc `string` *exp* <a> → `string`

Evaluate EXP and succeed only if it throws an error.

```
pact> (expect-failure "Enforce fails on false" (enforce false "Expected error"))
"Expect failure: success: Enforce fails on false"
```

6.6.11 json

exp <a> → `value`

Encode pact expression EXP as a JSON value. This is only needed for tests, as Pact values are automatically represented as JSON in API output.

```
pact> (json [{ "name": "joe", "age": 10 } {"name": "mary", "age": 25 }])
[{"age":10,"name":"joe"},{"age":25,"name":"mary"}]
```

6.6.12 load

file `string` → `string`

file `string` *reset* `bool` → `string`

Load and evaluate FILE, resetting repl state beforehand if optional NO-RESET is true.

```
(load "accounts.repl")
```

6.6.13 pact-state

→ `object`

Inspect state from previous pact execution. Returns object with fields 'yield': yield result or 'false' if none; 'step': executed step; 'executed': indicates if step was skipped because entity did not match.

```
(pact-state)
```

6.6.14 print

value <a> → `string`

Print a string, mainly to format newlines correctly

6.6.15 rollback-tx

→ `string`

Rollback transaction.

```
(rollback-tx)
```

6.6.16 sig-keyset

→ `keyset`

Convenience to build a keyset from keys present in message signatures, using 'keys-all' as the predicate.

6.6.17 typecheck

module `string` → `string`

module `string` *debug* `bool` → `string`

Typecheck MODULE, optionally enabling DEBUG output.