

API

Introduction

The Stacks 2.0 Blockchain API allows you to query the Stacks 2.0 blockchain and interact with smart contracts. It was built to maintain pageable materialized views of the Stacks 2.0 Blockchain.

Note that the [Stacks Node RPC API](#) and the [Hiro Stacks API](#) are two different things. The Hiro API is a centralized service run by Hiro, a developer tooling company, that makes it easy to get onboarded and begin interacting with the Stacks blockchain in a RESTful way. You can also [run your own API server](#)

The Hiro Stacks API is a proxy for the Stacks Node API that makes it a bit easier to work with by providing additional functionality.

The RPC API is generated by every Stacks node and allows developers to self-host their own node and API for a more decentralized architecture.

This documentation only covers endpoints that are exposed on a Stacks node, referred to as the RPC API. For full documentation on the RESTful API, check out the [Hiro's API reference](#).

The RPC API can be used without any authorization. The basepath for the API is:

Copy

for mainnet, replace `testnet` with `mainnet`

`https://api.testnet.hiro.so/`

If you run a local node, it exposes an HTTP server on port `20443`. The info endpoint would be `localhost:20443/v2/info`.

Stacks Node RPC API endpoints

The Stacks 2.0 Blockchain API (Hiro's API) is centrally hosted. However, every running Stacks node exposes an RPC API, which allows you to interact with the underlying blockchain. Instead of using a centrally hosted API, you can directly access the RPC API of a locally hosted Node.

Broadcast raw transaction

Broadcast raw transactions on the network. You can use the [@stacks/transactions](#) project to generate a raw transaction payload.

`POST http://localhost:20443/v2/transactions`

Body

`application/octet-stream`

Transaction id of successful post of a raw tx to the node's mempool

text/plain

string

Example: `"\"e161978626f216b2141b156ade10501207ae535fa365a13ef5d7a7c9310a09f2\""`

Request

JavaScriptCurlPython

Copy

```
const response = await fetch('http://localhost:20443/v2/transactions', {
```

method: 'POST',

```
headers: {
```

```
"Content-Type": "application/octet-stream"
```

},

```
body: JSON.stringify("binary format of
```

[illegible] $\}) ;$

```
const data = await response.json();
```

Test it

Response

200

Copy

```
"e161978626f216b2141b156ade10501207ae535fa365a13ef5d7a7c9310a09f2"
```

Get contract interface

Get contract interface using a `ContractName` and

GEThttp://localhost:20443/v2/contracts/interface/{contract address}/{contract name}

Path parameters

```
contract address*string
```

Stacks address

contract_name*string

Contract name

Query parameters

Response

200

Contract interface

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await
```

```
fetch('http://localhost:20443/v2/contracts/interface/{contract_address}/{contract_name}', {
```

```
  method: 'GET',
```

```
  headers: {},
```

```
});
```

```
const data = await response.json();
```

Test it

Get specific data-map inside a contract

Attempt to fetch data from a contract data map. The contract is identified with [Stacks Address] and [Contract Name] in the URL path. The map is identified with [Map Name].

The key to lookup in the map is supplied via the POST body. This should be supplied as the hex string serialization of the key (which should be a Clarity value). Note, this is a JSON string atom.

In the response, `data` is the hex serialization of the map response. Note that map responses are Clarity option types, for non-existent values, this is a serialized none, and for all other responses, it is a serialized (some ...) object.

```
POSThttp://localhost:20443/v2/map_entry/{contract_address}/{contract_name}/{map_name}
```

Path parameters

contract_address*string

Stacks address

contract_name*string

Contract name

map_name*string

Map name

Query parameters

Body

application/json

Hex string serialization of the lookup key (which should be a Clarity value)

string

Response

200400

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

const response = await

fetch('http://localhost:20443/v2/map_entry/{contract_address}/{contract_name}/{map_name}', {

method: 'POST',

headers: {

"Content-Type": "application/json"

},

body: JSON.stringify("text"),

});

const data = await response.json();

Test it

Get contract source

Returns the Clarity source code of a given contract, along with the block height it was published

in, and the MARF proof for the data

GEThttp://localhost:20443/v2/contracts/source/{contract_address}/{contract_name}

Path parameters

contract_address*string

Stacks address

contract_name*string

Contract name

Query parameters

Response

200

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await
```

```
fetch('http://localhost:20443/v2/contracts/source/{contract_address}/{contract_name}', {
```

```
  method: 'GET',
```

```
  headers: {},
```

```
});
```

```
const data = await response.json();
```

Test it

Call read-only function

Call a read-only public function on a given smart contract.

The smart contract and function are specified using the URL path. The arguments and the simulated tx-sender are supplied via the POST body in the following JSON format:

```
POSThttp://localhost:20443/v2/contracts/call-read/{contract_address}/{contract_name}/{function_name}
```

Path parameters

contract_address*string

Stacks address

contract_name*string

Contract name

function_name*string

Function name

Query parameters

Body

application/json

map of arguments and the simulated tx-sender where sender is either a Contract identifier or a normal Stacks address, and arguments is an array of hex serialized Clarity values.

any

Response

200

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await
```

```
fetch('http://localhost:20443/v2/contracts/call-read/{contract_address}/{contract_name}/{function_name}', {
```

```
  method: 'POST',
```

```
  headers: {
```

```
    "Content-Type": "application/json"
```

```
  },
```

```
});
```

```
const data = await response.json();
```

Test it

Get account info

Get the account data for the provided principal

Where balance is the hex encoding of a unsigned 128-bit integer (big-endian), nonce is a unsigned 64-bit integer, and the proofs are provided as hex strings.

For non-existent accounts, this does not 404, rather it returns an object with balance and nonce of 0.

GET `http://localhost:20443/v2/accounts/{principal}`

Path parameters

principal*string

Stacks address or a Contract identifier (e.g.

)

Query parameters

Response

200

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await fetch('http://localhost:20443/v2/accounts/{principal}', {
```

```
  method: 'GET',
```

```
  headers: {},
```

```
});
```

```
const data = await response.json();
```

Test it

Get approximate fees for the given transaction

Get an estimated fee for the supplied transaction. This estimates the execution cost of the transaction, the current fee rate of the network, and returns estimates for fee amounts.

- `txid` is a hex-encoded serialization of the TransactionPayload for the transaction.
- `length` is an optional argument that provides the endpoint with an estimation of the final length (in bytes) of the transaction, including any post-conditions and signatures

If the node cannot provide an estimate for the transaction (e.g., if the node has never seen a contract-call for the given contract and function) or if estimation is not configured on this node, a 400 response is returned. The 400 response will be a JSON error containing a `message` field which can be one of the following:

- - this Stacks node has had an internal database error while trying to estimate the costs of the supplied transaction.
- - this Stacks node has not seen this kind of contract-call before, and it cannot provide an estimate yet.
- - this Stacks node does not perform fee or cost estimation, and it cannot respond on this endpoint.

The 200 response contains the following data:

- - the estimated multi-dimensional cost of executing the Clarity VM on the provided transaction.
- - a unitless integer that the Stacks node uses to compare how much of the block limit is consumed by different transactions. This value incorporates the estimated length of the transaction and the estimated execution cost of the transaction. The range of this integer may vary between different Stacks nodes. In order to compute an estimate of total fee amount for the transaction, this value is multiplied by the same Stacks node's estimated fee rate.
- - a float value that indicates how much the value would increase for every additional byte in the final transaction.
- - an array of estimated fee rates and total fees to pay in microSTX for the transaction. This array provides a range of estimates (default: 3) that may be used. Each element of the array contains the following fields:
 - - the estimated value for the current fee rates in the network
 - - the estimated value for the total fee in microSTX that the given transaction should pay. These values are the result of computing: $\text{estimated_cost} \times \text{estimated_fee_rate}$. If the estimated fees are less than the minimum relay fee, then that minimum relay fee will be returned here instead.

Note: If the final transaction's byte size is larger than supplied to `POST /fees/transaction`, then applications should increase this fee amount by:

$$\text{estimated_cost} \times (\text{actual_size} - \text{supplied_size}) \times \text{estimated_fee_rate}$$

POST `http://localhost:20443/v2/fees/transaction`
 Body
 application/json
 any
 Response

200

Estimated fees for the transaction

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await fetch('http://localhost:20443/v2/fees/transaction', {
```

```
  method: 'POST',
```

```
  headers: {
```

```
    "Content-Type": "application/json"
```

```
  },
```

```
});
```

```
const data = await response.json();
```

Test it

Get estimated fee

Get an estimated fee rate for STX transfer transactions. This a a fee rate / byte, and is returned as a JSON integer

GEThttp://localhost:20443/v2/fees/transfer

Response

200

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await fetch('http://localhost:20443/v2/fees/transfer', {
```

```
  method: 'GET',
```

```
  headers: {},
```

```
});
```

```
const data = await response.json();
```

Test it

Get Core API info

Get Core API information

GET <http://localhost:20443/v2/info>

Response

200

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await fetch('http://localhost:20443/v2/info', {
```

```
  method: 'GET',
```

```
  headers: {},
```

```
});
```

```
const data = await response.json();
```

Test it

Get PoX details

Get Proof of Transfer (PoX) information. Can be used for Stacking.

GET <http://localhost:20443/v2/pox>

Query parameters

Response

200

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await fetch('http://localhost:20443/v2/pox', {
```

```

    method: 'GET',

    headers: {},

});

const data = await response.json();
Test it

```

Get trait implementation details

Determine whether or not a specified trait is implemented (either explicitly or implicitly) within a given contract.

GET `http://localhost:20443/v2/traits/{contract_address}/{contract_name}/{trait_contract_address}/{trait_contract_name}/{trait_name}`

Path parameters

`contract_address`*string

Stacks address

`contract_name`*string

Contract name

`trait_contract_address`*string

Trait Stacks address

`trait_contract_name`*string

Trait contract name

`trait_name`*string

Trait name

Query parameters

Response

200

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```
const response = await
```

```
fetch('http://localhost:20443/v2/traits/{contract_address}/{contract_name}/{trait_contract_address}/{trait_contract_name}/{trait_name}', {
```

```

method: 'GET',

headers: {},

});

const data = await response.json();
Test it

```

Get the value of a constant inside a contract

Attempt to fetch the value of a constant inside a contract. The contract is identified with [Stacks Address] and [Contract Name] in the URL path. The constant is identified with [Constant Name].

In the response, is the hex serialization of the constant value.

```

POSThttp://localhost:20443/v2/constant_val/{contract_address}/{contract_name}/{constant_name}
}

```

Path parameters

contract_address*string

Stacks address

contract_name*string

Contract name

constant_name*string

Constant name

Query parameters

Response

200400

Success

Body

application/json

any

Request

JavaScriptCurlPython

Copy

```

const response = await

```

```

fetch('http://localhost:20443/v2/constant_val/{contract_address}/{contract_name}/{constant_name}', {

```

```

method: 'POST',

```

```

headers: {},

```

```
});
```

```
const data = await response.json();
```

Clarity Types

Clarity Type System

The type system contains the following types:

Types	Notes
<code>int</code>	signed 128-bit integer
<code>uint</code>	unsigned 128-bit integer
<code>bool</code>	boolean value (<code>true</code> or <code>false</code>)
<code>principal</code>	object representing a principal (whether a contract principal or standard principal)
<code>(buff max-len)</code>	byte buffer of maximum length <code>max-len</code> .
<code>(string-ascii max-len)</code>	ASCII string of maximum length <code>max-len</code>

`(string-utf8 max-len)`

UTF-8 string of maximum length `max-len` (u"A smiley face emoji
\\u{1F600} as a utf8 string")

`(list max-len
entry-type)`

list of maximum length `max-len`, with entries of type `entry-type`

`{label-0: value-type-0,
label-1: value-type-1,
...}`

tuple, group of data values with named fields

`(optional some-type)`

an option type for objects that can either be `(some value)` or
`none`

`(response ok-type
err-type)`

object used by public functions to commit their changes or
abort. May be returned or used by other functions as well,
however, only public functions have the commit/abort behavior.

Clarity Functions

* (multiply)

Introduced in: **Clarity 1**

input: `int, ... | uint, ...`

output: `int | uint`

signature: `(* i1 i2...)`

description:

Multiplies a variable number of integer inputs and returns the result. In the event of an *overflow*, throws a runtime error.

example:

Copy

(* 2 3) ;; Returns 6

(* 5 2) ;; Returns 10

(* 2 2 2) ;; Returns 8

+ (add)

Introduced in: **Clarity 1**

input: `int, ... | uint, ...`

output: `int | uint`

signature: `(+ i1 i2...)`

description:

Adds a variable number of integer inputs and returns the result. In the event of an *overflow*, throws a runtime error.

example:

Copy

(+ 1 2 3) ;; Returns 6

- (subtract)

Introduced in: **Clarity 1**

input: `int, ... | uint, ...`

output: `int | uint`

signature: `(- i1 i2...)`

description:

Subtracts a variable number of integer inputs and returns the result. In the event of an *underflow*, throws a runtime error.

example:

Copy

(- 2 1 1) ;; Returns 0

(- 0 3) ;; Returns -3

/ (divide)

Introduced in: **Clarity 1**

input: `int, ... | uint, ...`

output: `int | uint`

signature: `(/ i1 i2...)`

description:

Integer divides a variable number of integer inputs and returns the result. In the event of division by zero, throws a runtime error.

example:

Copy

(/ 2 3) ;; Returns 0

(/ 5 2) ;; Returns 2

(/ 4 2 2) ;; Returns 1

< (less than)

Introduced in: **Clarity 1**

input: `int, int | uint, uint | string-ascii, string-ascii | string-utf8, string-utf8 | buff, buff`

output: `bool`

signature: `(< i1 i2)`

description:

Compares two integers, returning `true` if `i1` is less than `i2` and `false` otherwise. `i1` and `i2` must be of the same type. Starting with Stacks 1.0, the `<`-comparable types are `int` and `uint`. Starting with Stacks 2.1, the `<`-comparable types are expanded to include `string-ascii`, `string-utf8` and `buff`.

example:

Copy

```
(< 1 2) ;; Returns true
```

```
(< 5 2) ;; Returns false
```

```
(< "aaa" "baa") ;; Returns true
```

```
(< "aa" "aaa") ;; Returns true
```

```
(< 0x01 0x02) ;; Returns true
```

```
(< 5 u2) ;; Throws type error
```

<= (less than or equal)

Introduced in: **Clarity 1**

input: `int, int | uint, uint | string-ascii, string-ascii | string-utf8, string-utf8 | buff, buff`

output: `bool`

signature: `(<= i1 i2)`

description:

Compares two integers, returning `true` if `i1` is less than or equal to `i2` and `false` otherwise. `i1` and `i2` must be of the same type. Starting with Stacks 1.0, the `<=`-comparable types are `int` and `uint`. Starting with Stacks 2.1, the `<=`-comparable types are expanded to include `string-ascii`, `string-utf8` and `buff`.

example:

Copy

```
(<= 1 1) ;; Returns true
```

```
(<= 5 2) ;; Returns false
```

`(<= "aaa" "baa") ;; Returns true`

`(<= "aa" "aaa") ;; Returns true`

`(<= 0x01 0x02) ;; Returns true`

`(<= 5 u2) ;; Throws type error`

> (greater than)

Introduced in: **Clarity 1**

input: `int, int | uint, uint | string-ascii, string-ascii | string-utf8, string-utf8 | buff, buff`

output: `bool`

signature: `(> i1 i2)`

description:

Compares two integers, returning `true` if `i1` is greater than `i2` and false otherwise. `i1` and `i2` must be of the same type. Starting with Stacks 1.0, the `>`-comparable types are `int` and `uint`. Starting with Stacks 2.1, the `>`-comparable types are expanded to include `string-ascii`, `string-utf8` and `buff`.

example:

Copy

`(> 1 2) ;; Returns false`

`(> 5 2) ;; Returns true`

`(> "baa" "aaa") ;; Returns true`

`(> "aaa" "aa") ;; Returns true`

`(> 0x02 0x01) ;; Returns true`

`(> 5 u2) ;; Throws type error`

>= (greater than or equal)

Introduced in: **Clarity 1**

input: `int, int | uint, uint | string-ascii, string-ascii | string-utf8, string-utf8 | buff, buff`

output: `bool`

signature: `(>= i1 i2)`

description:

Compares two integers, returning `true` if `i1` is greater than or equal to `i2` and `false` otherwise. `i1` and `i2` must be of the same type. Starting with Stacks 1.0, the `>=`-comparable types are `int` and `uint`. Starting with Stacks 2.1, the `>=`-comparable types are expanded to include `string-ascii`, `string-utf8` and `buff`.

example:

Copy

`(>= 1 1) ;;` Returns true

`(>= 5 2) ;;` Returns true

`(>= "baa" "aaa") ;;` Returns true

`(>= "aaa" "aa") ;;` Returns true

`(>= 0x02 0x01) ;;` Returns true

`(>= 5 u2) ;;` Throws type error

and

Introduced in: **Clarity 1**

input: `bool, ...`

output: `bool`

signature: `(and b1 b2 ...)`

description:

Returns `true` if all boolean inputs are `true`. Importantly, the supplied arguments are evaluated in-order and lazily. Lazy evaluation means that if one of the arguments returns `false`, the function short-circuits, and no subsequent arguments are evaluated.

example:

Copy

`(and true false) ;; Returns false`

`(and (is-eq (+ 1 2) 1) (is-eq 4 4)) ;; Returns false`

`(and (is-eq (+ 1 2) 3) (is-eq 4 4)) ;; Returns true`

append

Introduced in: **Clarity 1**

input: `list A, A`

output: `list`

signature: `(append (list 1 2 3 4) 5)`

description:

The `append` function takes a list and another value with the same entry type, and outputs a list of the same type with `max_len += 1`.

example:

Copy

`(append (list 1 2 3 4) 5) ;; Returns (1 2 3 4 5)`

as-contract

Introduced in: **Clarity 1**

input: `A`

output: `A`

signature: `(as-contract expr)`

description:

The `as-contract` function switches the current context's `tx-sender` value to the *contract's* principal and executes `expr` with that context. It returns the resulting value of `expr`.

example:

Copy

`(as-contract tx-sender) ;; Returns S1G2081040G2081040G2081040G208105NK8PE5.docs-test`

as-max-len?

Introduced in: **Clarity 1**

input: `sequence_A`, `uint`

output: `(optional sequence_A)`

signature: `(as-max-len? sequence max_length)`

description:

The `as-max-len?` function takes a sequence argument and a uint-valued, literal length argument. The function returns an optional type. If the input sequence length is less than or equal to the supplied `max_length`, this returns `(some sequence)`, otherwise it returns `none`. Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`.

example:

Copy

`(as-max-len? (list 2 2 2) u3) ;; Returns (some (2 2 2))`

`(as-max-len? (list 1 2 3) u2) ;; Returns none`

`(as-max-len? "hello" u10) ;; Returns (some "hello")`

`(as-max-len? 0x010203 u10) ;; Returns (some 0x010203)`

asserts!

Introduced in: **Clarity 1**

input: `bool`, `C`

output: `bool`

signature: (asserts! bool-expr thrown-value)

description:

The `asserts!` function admits a boolean argument and asserts its evaluation: if `bool-expr` is `true`, `asserts!` returns `true` and proceeds in the program execution. If the supplied argument is returning a false value, `asserts!` *returns thrown-value* and exits the current control-flow.

example:

Copy

```
(asserts! (is-eq 1 1) (err 1)) ;; Returns true
```

at-block

Introduced in: **Clarity 1**

input: (buff 32), A

output: A

signature: (at-block id-block-hash expr)

description:

The `at-block` function evaluates the expression `expr` as if it were evaluated at the end of the block indicated by the `block-hash` argument. The `expr` closure must be read-only.

Note: The block identifying hash must be a hash returned by the `id-header-hash` block information property. This hash uniquely identifies Stacks blocks and is unique across Stacks forks. While the hash returned by `header-hash` is unique within the context of a single fork, it is not unique across Stacks forks.

The function returns the result of evaluating `expr`.

example:

Copy

```
(define-data-var data int 1)
```

[illegible]

(at-block (get-block-info? id-header-hash 0) (var-get data)) ;; Throws NoSuchDataVariable because `data` wasn't initialized at block height 0

begin

Introduced in: **Clarity 1**

input: `AnyType, ... A`

output: `A`

signature: `(begin expr1 expr2 expr3 ... expr-last)`

description:

The `begin` function evaluates each of its input expressions, returning the return value of the last such expression. Note: intermediary statements returning a response type must be checked.

example:

Copy

`(begin (+ 1 2) 4 5) ;; Returns 5`

bit-and

Introduced in: **Clarity 2**

input: `int, ... | uint, ...`

output: `int | uint`

signature: `(bit-and i1 i2...)`

description:

Returns the result of bitwise and'ing a variable number of integer inputs.

example:

Copy

`(bit-and 24 16) ;; Returns 16`

`(bit-and 28 24 -1) ;; Returns 24`

`(bit-and u24 u16) ;; Returns u16`

`(bit-and -128 -64) ;; Returns -128`

(bit-and 28 24 -1) ;; Returns 24

bit-not

Introduced in: **Clarity 2**

input: `int | uint`

output: `int | uint`

signature: `(bit-not i1)`

description:

Returns the one's complement (sometimes also called the bitwise compliment or not operator) of `i1`, effectively reversing the bits in `i1`. In other words, every bit that is `1` in `i1` will be `0` in the result. Conversely, every bit that is `0` in `i1` will be `1` in the result.

example:

Copy

(bit-not 3) ;; Returns -4

(bit-not u128) ;; Returns u340282366920938463463374607431768211327

(bit-not 128) ;; Returns -129

(bit-not -128) ;; Returns 127

bit-or

Introduced in: **Clarity 2**

input: `int, ... | uint, ...`

output: `int | uint`

signature: `(bit-or i1 i2...)`

description:

Returns the result of bitwise inclusive or'ing a variable number of integer inputs.

example:

Copy

(bit-or 4 8) ;; Returns 12

(bit-or 1 2 4) ;; Returns 7

(bit-or 64 -32 -16) ;; Returns -16

(bit-or u2 u4 u32) ;; Returns u38

bit-shift-left

Introduced in: **Clarity 2**

input: `int, uint | uint, uint`

output: `int | uint`

signature: `(bit-shift-left i1 shamt)`

description:

Shifts all the bits in `i1` to the left by the number of places specified in `shamt` modulo 128 (the bit width of Clarity integers).

Note that there is a deliberate choice made to ignore arithmetic overflow for this operation. In use cases where overflow should be detected, developers should use `*`, `/`, and `pow` instead of the shift operators.

example:

Copy

(bit-shift-left 2 u1) ;; Returns 4

(bit-shift-left 16 u2) ;; Returns 64

(bit-shift-left -64 u1) ;; Returns -128

(bit-shift-left u4 u2) ;; Returns u16

(bit-shift-left 123 u9999999999) ;; Returns -170141183460469231731687303715884105728

(bit-shift-left u123 u9999999999) ;; Returns u170141183460469231731687303715884105728

(bit-shift-left -1 u7) ;; Returns -128

(bit-shift-left -1 u128) ;; Returns -1

bit-shift-right

Introduced in: **Clarity 2**

input: `int`, `uint` | `uint`, `uint`

output: `int` | `uint`

signature: `(bit-shift-right i1 shamt)`

description:

Shifts all the bits in `i1` to the right by the number of places specified in `shamt` modulo 128 (the bit width of Clarity integers). When `i1` is a `uint` (unsigned), new bits are filled with zeros. When `i1` is an `int` (signed), the sign is preserved, meaning that new bits are filled with the value of the previous sign-bit.

Note that there is a deliberate choice made to ignore arithmetic overflow for this operation. In use cases where overflow should be detected, developers should use `*`, `/`, and `pow` instead of the shift operators.

example:

Copy

(bit-shift-right 2 u1) ;; Returns 1

(bit-shift-right 128 u2) ;; Returns 32

(bit-shift-right -64 u1) ;; Returns -32

(bit-shift-right u128 u2) ;; Returns u32

(bit-shift-right 123 u9999999999) ;; Returns 0

(bit-shift-right u123 u9999999999) ;; Returns u0

(bit-shift-right -128 u7) ;; Returns -1

(bit-shift-right -256 u1) ;; Returns -128

(bit-shift-right 5 u2) ;; Returns 1

(bit-shift-right -5 u2) ;; Returns -2

bit-xor

Introduced in: **Clarity 2**

input: `int, ... | uint, ...`

output: `int | uint`

signature: `(bit-xor i1 i2...)`

description:

Returns the result of bitwise exclusive or'ing a variable number of integer inputs.

example:

Copy

(bit-xor 1 2) ;; Returns 3

(bit-xor 120 280) ;; Returns 352

(bit-xor -128 64) ;; Returns -64

(bit-xor u24 u4) ;; Returns u28

(bit-xor 1 2 4 -1) ;; Returns -8

buff-to-int-be

Introduced in: **Clarity 2**

input: `(buff 16)`

output: `int`

signature: `(buff-to-int-be (buff 16))`

description:

Converts a byte buffer to a signed integer use a big-endian encoding. The byte buffer can be up to 16 bytes in length. If there are fewer than 16 bytes, as this function uses a big-endian encoding, the input behaves as if it is zero-padded on the *left*.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

(buff-to-int-be 0x01) ;; Returns 1

(buff-to-int-be 0x00000000000000000000000000000001) ;; Returns 1

(buff-to-int-be 0xffffffffffffffffffffffff) ;; Returns -1

(buff-to-int-be 0x) ;; Returns 0

buff-to-int-le

Introduced in: **Clarity 2**

input: (buff 16)

output: int

signature: (buff-to-int-le (buff 16))

description:

Converts a byte buffer to a signed integer use a little-endian encoding. The byte buffer can be up to 16 bytes in length. If there are fewer than 16 bytes, as this function uses a little-endian encoding, the input behaves as if it is zero-padded on the *right*.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

(buff-to-int-le 0x01) ;; Returns 1

(buff-to-int-le 0x01000000000000000000000000000000) ;; Returns 1

(buff-to-int-le 0xffffffffffffffffffffffff) ;; Returns -1

(buff-to-int-le 0x) ;; Returns 0

buff-to-uint-be

Introduced in: **Clarity 2**

input: (buff 16)

output: uint

signature: (buff-to-uint-be (buff 16))

description:

Converts a byte buffer to an unsigned integer use a big-endian encoding. The byte buffer can be up to 16 bytes in length. If there are fewer than 16 bytes, as this function uses a big-endian encoding, the input behaves as if it is zero-padded on the *left*.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

(buff-to-uint-be 0x01) ;; Returns u1

(buff-to-uint-be 0x00000000000000000000000000000001) ;; Returns u1

(buff-to-uint-be 0xffffffffffffffffffffffffffff) ;; Returns
u340282366920938463463374607431768211455

(buff-to-uint-be 0x) ;; Returns u0

buff-to-uint-le

Introduced in: **Clarity 2**

input: (buff 16)

output: uint

signature: (buff-to-uint-le (buff 16))

description:

Converts a byte buffer to an unsigned integer use a little-endian encoding.. The byte buffer can be up to 16 bytes in length. If there are fewer than 16 bytes, as this function uses a little-endian encoding, the input behaves as if it is zero-padded on the *right*.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

(buff-to-uint-le 0x01) ;; Returns u1

(buff-to-uint-le 0x01000000000000000000000000000000) ;; Returns u1

(buff-to-uint-le 0xffffffffffffffffffffffff) ;; Returns
u340282366920938463463374607431768211455

(buff-to-uint-le 0x) ;; Returns u0

concat

Introduced in: **Clarity 1**

input: `sequence_A`, `sequence_A`

output: `sequence_A`

signature: (`concat` `sequence1` `sequence2`)

description:

The `concat` function takes two sequences of the same type, and returns a concatenated sequence of the same type, with the resulting `sequence_len` = `sequence1_len` + `sequence2_len`. Applicable sequence types are (`list A`), `buff`, `string-ascii` and `string-utf8`.

example:

Copy

(concat (list 1 2) (list 3 4)) ;; Returns (1 2 3 4)

(concat "hello " "world") ;; Returns "hello world"

(concat 0x0102 0x0304) ;; Returns 0x01020304

contract-call?

Introduced in: **Clarity 1**

input: `ContractName, PublicFunctionName, Arg0, ...`

output: `(response A B)`

signature: `(contract-call? .contract-name function-name arg0 arg1 ...)`

description:

The `contract-call?` function executes the given public function of the given contract. You *may not* use this function to call a public function defined in the current contract. If the public function returns *err*, any database changes resulting from calling `contract-call?` are aborted. If the function returns *ok*, database changes occurred.

example:

Copy

`:: instantiate the sample-contracts/tokens.clar contract first`

`(as-contract (contract-call? .tokens mint! u19)) ;; Returns (ok u19)`

contract-of

Introduced in: **Clarity 1**

input: `Trait`

output: `principal`

signature: `(contract-of .contract-name)`

description:

The `contract-of` function returns the principal of the contract implementing the trait.

example:

Copy

`(use-trait token-a-trait`

`'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF.token-a.token-trait)`

```
(define-public (forward-get-balance (user principal) (contract <token-a-trait>))  
  
  (begin  
  
    (ok (contract-of contract)))) ;; returns the principal of the contract implementing <token-a-trait>
```

default-to

Introduced in: **Clarity 1**

input: `A`, `(optional A)`

output: `A`

signature: `(default-to default-value option-value)`

description:

The `default-to` function attempts to 'unpack' the second argument: if the argument is a `(some ...)` option, it returns the inner value of the option. If the second argument is a `(none)` value, `default-to` it returns the value of `default-value`.

example:

Copy

```
(define-map names-map { name: (string-ascii 12) } { id: int })
```

```
(map-set names-map { name: "blockstack" } { id: 1337 })
```

```
(default-to 0 (get id (map-get? names-map (tuple (name "blockstack"))))) ;; Returns 1337
```

```
(default-to 0 (get id (map-get? names-map (tuple (name "non-existent"))))) ;; Returns 0
```

define-constant

Introduced in: **Clarity 1**

input: `MethodSignature`, `MethodBody`

output: `Not Applicable`

signature: `(define-constant name expression)`

description:

`define-constant` is used to define a private constant value in a smart contract. The expression passed into the definition is evaluated at contract launch, in the order that it is supplied in the contract. This can lead to undefined function or undefined variable errors in the event that a function or variable used in the expression has not been defined before the constant.

Like other kinds of definition statements, `define-constant` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

example:

Copy

```
(define-constant four (+ 2 2))
```

```
(+ 4 four) ;; Returns 8
```

define-data-var

Introduced in: **Clarity 1**

input: `VarName, TypeDefinition, Value`

output: `Not Applicable`

signature: `(define-data-var var-name type value)`

description:

`define-data-var` is used to define a new persisted variable for use in a smart contract. Such variable are only modifiable by the current smart contract.

Persisted variable are defined with a type and a value.

Like other kinds of definition statements, `define-data-var` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

example:

Copy

```
(define-data-var size int 0)
```

```
(define-private (set-size (value int))
```

(var-set size value))

(set-size 1)

(set-size 2)

define-fungible-token

Introduced in: **Clarity 1**

input: `TokenName`, `<uint>`

output: `Not Applicable`

signature: `(define-fungible-token token-name <total-supply>)`

description:

`define-fungible-token` is used to define a new fungible token class for use in the current contract.

The second argument, if supplied, defines the total supply of the fungible token. This ensures that all calls to the `ft-mint?` function will never be able to create more than `total-supply` tokens. If any such call were to increase the total supply of tokens passed that amount, that invocation of `ft-mint?` will result in a runtime error and abort.

Like other kinds of definition statements, `define-fungible-token` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

Tokens defined using `define-fungible-token` may be used in `ft-transfer?`, `ft-mint?`, and `ft-get-balance` functions

example:

Copy

```
(define-fungible-token stacks)
```

```
(define-fungible-token limited-supply-stacks u100)
```

define-map

Introduced in: **Clarity 1**

input: `MapName, TypeDefinition, TypeDefinition`

output: `Not Applicable`

signature: `(define-map map-name key-type value-type)`

description:

`define-map` is used to define a new datamap for use in a smart contract. Such maps are only modifiable by the current smart contract.

Maps are defined with a key type and value type, often these types are tuple types.

Like other kinds of definition statements, `define-map` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

example:

Copy

```
(define-map squares { x: int } { square: int })
```

```
(define-private (add-entry (x int))
```

```
  (map-insert squares { x: 2 } { square: (* x x) })))
```

```
(add-entry 1)
```

```
(add-entry 2)
```

```
(add-entry 3)
```

```
(add-entry 4)
```

```
(add-entry 5)
```

define-non-fungible-token

Introduced in: **Clarity 1**

input: `AssetName, TypeSignature`

output: Not Applicable

signature: (define-non-fungible-token asset-name asset-identifier-type)

description:

`define-non-fungible-token` is used to define a new non-fungible token class for use in the current contract. Individual assets are identified by their asset identifier, which must be of the type `asset-identifier-type`. Asset identifiers are *unique* identifiers.

Like other kinds of definition statements, `define-non-fungible-token` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

Assets defined using `define-non-fungible-token` may be used in `nft-transfer?`, `nft-mint?`, and `nft-get-owner?` functions

example:

Copy

```
(define-non-fungible-token names (buff 50))
```

define-private

Introduced in: **Clarity 1**

input: MethodSignature, MethodBody

output: Not Applicable

signature: (define-private (function-name (arg-name-0 arg-type-0) (arg-name-1 arg-type-1) ...) function-body)

description:

`define-private` is used to define *private* functions for a smart contract. Private functions may not be called from other smart contracts, nor may they be invoked directly by users. Instead, these functions may only be invoked by other functions defined in the same smart contract.

Like other kinds of definition statements, `define-private` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

Private functions may return any type.

example:

Copy

```
(define-private (max-of (i1 int) (i2 int))
```

```
(if (> i1 i2)
```

```
  i1
```

```
  i2))
```

(max-of 4 6) ;; Returns 6

define-public

Introduced in: **Clarity 1**

input: `MethodSignature`, `MethodBody`

output: `Not Applicable`

signature: `(define-public (function-name (arg-name-0 arg-type-0) (arg-name-1 arg-type-1) ...) function-body)`

description:

`define-public` is used to define a *public* function and transaction for a smart contract. Public functions are callable from other smart contracts and may be invoked directly by users by submitting a transaction to the Stacks blockchain.

Like other kinds of definition statements, `define-public` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

Public functions *must* return a `ResponseType` (using either `ok` or `err`). Any datamap modifications performed by a public function is aborted if the function returns an `err` type. Public functions may be invoked by other contracts via `contract-call?`.

example:

Copy

```
(define-public (hello-world (input int))
```

```
(begin
```

```
(print (+ 2 input))
```

```
(ok input)))
```

define-read-only

Introduced in: **Clarity 1**

input: `MethodSignature`, `MethodBody`

output: `Not Applicable`

signature: `(define-read-only (function-name (arg-name-0 arg-type-0) (arg-name-1 arg-type-1) ...) function-body)`

description:

`define-read-only` is used to define a *public read-only* function for a smart contract. Such functions are callable from other smart contracts.

Like other kinds of definition statements, `define-read-only` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

Read-only functions may return any type. However, read-only functions may not perform any datamap modifications, or call any functions which perform such modifications. This is enforced both during type checks and during the execution of the function. Public read-only functions may be invoked by other contracts via `contract-call?`.

example:

Copy

```
(define-read-only (just-return-one-hundred)
```

```
(* 10 10))
```

define-trait

Introduced in: **Clarity 1**

input: `VarName`, `[MethodSignature]`

output: `Not Applicable`

signature: `(define-trait trait-name ((func1-name (arg1-type arg2-type ...) (return-type))))`

description:

`define-trait` is used to define a new trait definition for use in a smart contract. Other contracts can implement a given trait and then have their contract identifier being passed as a function argument in order to be called dynamically with `contract-call?`.

Traits are defined with a name, and a list functions, defined with a name, a list of argument types, and return type.

In Clarity 1, a trait type can be used to specify the type of a function parameter. A parameter with a trait type can be used as the target of a dynamic `contract-call?`. A principal literal (e.g. `ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.foo`) may be passed as a trait parameter if the specified contract implements all of the functions specified by the trait. A trait value (originating from a parameter with trait type) may also be passed as a trait parameter if the types are the same.

Beginning in Clarity 2, a trait can be used in all of the same ways that a built-in type can be used, except that it cannot be stored in a data var or map, since this would inhibit static analysis. This means that a trait type can be embedded in a compound type (e.g. `(optional <my-trait>)` or `(list 4 <my-trait>)`) and a trait value can be bound to a variable in a `let` or `match` expression. In addition to the principal literal and trait value with matching type allowed in Clarity 1, Clarity 2 also supports implicit casting from a compatible trait, meaning that a value of type `trait-a` may be passed to a parameter with type `trait-b` if `trait-a` includes all of the requirements of `trait-b` (and optionally additional functions).

Like other kinds of definition statements, `define-trait` may only be used at the top level of a smart contract definition (i.e., you cannot put a define statement in the middle of a function body).

example:

Copy

```
(define-trait token-trait
```

```
((transfer? (principal principal uint) (response uint uint))
```

```
(get-balance (principal) (response uint uint))))
```

element-at

Introduced in: **Clarity 1**

input: `sequence_A, uint`

output: `(optional A)`

signature: `(element-at? sequence index)`

description:

The `element-at?` function returns the element at `index` in the provided sequence. Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`, for which the corresponding element types are, respectively, `A`, `(buff 1)`, `(string-ascii 1)` and `(string-utf8 1)`. In Clarity1, `element-at` must be used (without the `?`). The `?` is added in Clarity2 for consistency -- built-ins that return responses or optionals end in `?`. The Clarity1 spelling is left as an alias in Clarity2 for backwards compatibility.

example:

Copy

`(element-at? "blockstack" u5) ;; Returns (some "s")`

`(element-at? (list 1 2 3 4 5) u5) ;; Returns none`

`(element-at? (list 1 2 3 4 5) (+ u1 u2)) ;; Returns (some 4)`

`(element-at? "abcd" u1) ;; Returns (some "b")`

`(element-at? 0xfb01 u1) ;; Returns (some 0x01)`

element-at?

Introduced in: **Clarity 2**

input: `sequence_A, uint`

output: `(optional A)`

signature: `(element-at? sequence index)`

description:

The `element-at?` function returns the element at `index` in the provided sequence. Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`, for which the corresponding element types are, respectively, `A`, `(buff 1)`, `(string-ascii 1)` and `(string-utf8 1)`. In Clarity1, `element-at` must be used (without the `?`). The `?` is added in

Clarity2 for consistency -- built-ins that return responses or optionals end in `?`. The Clarity1 spelling is left as an alias in Clarity2 for backwards compatibility.

example:

Copy

`(element-at? "blockstack" u5) ;; Returns (some "s")`

`(element-at? (list 1 2 3 4 5) u5) ;; Returns none`

`(element-at? (list 1 2 3 4 5) (+ u1 u2)) ;; Returns (some 4)`

`(element-at? "abcd" u1) ;; Returns (some "b")`

`(element-at? 0xfb01 u1) ;; Returns (some 0x01)`

err

Introduced in: **Clarity 1**

input: `A`

output: `(response A B)`

signature: `(err value)`

description:

The `err` function constructs a response type from the input value. Use `err` for creating return values in public functions. An `err` value indicates that any database changes during the processing of the function should be rolled back.

example:

Copy

`(err true) ;; Returns (err true)`

filter

Introduced in: **Clarity 1**

input: `Function(A) -> bool, sequence_A`

output: `sequence_A`

signature: (`filter func sequence`)

description:

The `filter` function applies the input function `func` to each element of the input sequence, and returns the same sequence with any elements removed for which `func` returned `false`.

Applicable sequence types are (`list A`), `buff`, `string-ascii` and `string-utf8`, for which the corresponding element types are, respectively, `A`, (`buff 1`), (`string-ascii 1`) and (`string-utf8 1`). The `func` argument must be a literal function name.

example:

Copy

(`filter not (list true false true false)`) ;; Returns (`false false`)

(`define-private (is-a (char (string-utf8 1)))`

(`is-eq char u"a"`)

(`filter is-a u"acabd"`) ;; Returns `u"aa"`

(`define-private (is-zero (char (buff 1)))`

(`is-eq char 0x00`)

(`filter is-zero 0x00010002`) ;; Returns `0x0000`

fold

Introduced in: **Clarity 1**

input: `Function(A, B) -> B, sequence_A, B`

output: `B`

signature: (`fold func sequence_A initial_B`)

description:

The `fold` function condenses `sequence_A` into a value of type `B` by recursively applies the function `func` to each element of the input sequence *and* the output of a previous application of `func`.

`fold` uses `initial_B` in the initial application of `func`, along with the first element of `sequence_A`. The resulting value of type `B` is used for the next application of `func`, along with the next element of `sequence_A` and so on. `fold` returns the last value of type `B` returned by these successive applications `func`.

Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`, for which the corresponding element types are, respectively, `A`, `(buff 1)`, `(string-ascii 1)` and `(string-utf8 1)`. The `func` argument must be a literal function name.

example:

Copy

```
(fold * (list 2 2 2) 1) ;; Returns 8
```

```
(fold * (list 2 2 2) 0) ;; Returns 0
```

```
;; calculates (- 11 (- 7 (- 3 2)))
```

```
(fold - (list 3 7 11) 2) ;; Returns 5
```

```
(define-private (concat-string (a (string-ascii 20)) (b (string-ascii 20)))
```

```
  (unwrap-panic (as-max-len? (concat a b) u20)))
```

```
(fold concat-string "cdef" "ab") ;; Returns "fedcab"
```

```
(fold concat-string (list "cd" "ef") "ab") ;; Returns "efcdab"
```

```
(define-private (concat-buff (a (buff 20)) (b (buff 20)))
```

```
  (unwrap-panic (as-max-len? (concat a b) u20)))
```

```
(fold concat-buff 0x03040506 0x0102) ;; Returns 0x060504030102
```

from-consensus-buff?

Introduced in: **Clarity 2**

input: `type-signature(t)`, `buff`

output: `(optional t)`

signature: `(from-consensus-buff? type-signature buffer)`

description:

`from-consensus-buff?` is a special function that will deserialize a buffer into a Clarity value, using the SIP-005 serialization of the Clarity value. The type that `from-consensus-buff?` tries to deserialize into is provided by the first parameter to the function. If it fails to deserialize the type, the method returns `none`.

example:

Copy

```
(from-consensus-buff? int 0x00000000000000000000000000000001) ;; Returns (some 1)
```

```
(from-consensus-buff? uint 0x00000000000000000000000000000000) ;; Returns none
```

```
(from-consensus-buff? uint 0x0100000000000000000000000000000001) ;; Returns (some u1)
```

```
(from-consensus-buff? bool 0x0000000000000000000000000000000001) ;; Returns none
```

```
(from-consensus-buff? bool 0x03) ;; Returns (some true)
```

(from-consensus-buff? bool 0x04) ;; Returns (some false)

(from-consensus-buff? principal 0x051fa46ff88886c2ef9762d970b4d2c63678835bd39d) ;;
Returns (some SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)

[illegible]

ft-burn?

Introduced in: **Clarity 1**

input: TokenName, uint, principal

output: (response bool uint)

signature: (ft-burn? token-name amount sender)

description:

`ft-burn?` is used to decrease the token balance for the `sender` principal for a token type defined using `define-fungible-token`. The decreased token balance is *not* transferred to another principal, but rather destroyed, reducing the circulating supply.

On a successful burn, it returns (`ok true`). In the event of an unsuccessful burn it returns one of the following error codes:

(`err u1`) -- `sender` does not have enough balance to burn this amount or the amount specified is not positive

example:

Copy

```
(define-fungible-token stackaroo)
```

```
(ft-mint? stackaroo u100 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns  
(ok true)
```

```
(ft-burn? stackaroo u50 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns  
(ok true)
```

ft-get-balance

Introduced in: **Clarity 1**

input: `TokenName`, `principal`

output: `uint`

signature: (`ft-get-balance token-name principal`)

description:

`ft-get-balance` returns `token-name` balance of the principal `principal`. The token type must have been defined using `define-fungible-token`.

example:

Copy

```
(define-fungible-token stackaroo)
```

```
(ft-mint? stackaroo u100 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)
```

```
(ft-get-balance stackaroo 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR) ;;  
Returns u100
```

ft-get-supply

Introduced in: **Clarity 1**

input: `TokenName`

output: `uint`

signature: `(ft-get-supply token-name)`

description:

`ft-get-supply` returns `token-name` circulating supply. The token type must have been defined using `define-fungible-token`.

example:

Copy

```
(define-fungible-token stackaroo)
```

```
(ft-mint? stackaroo u100 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)
```

```
(ft-get-supply stackaroo) ;; Returns u100
```

ft-mint?

Introduced in: **Clarity 1**

input: `TokenName, uint, principal`

output: `(response bool uint)`

signature: `(ft-mint? token-name amount recipient)`

description:

`ft-mint?` is used to increase the token balance for the `recipient` principal for a token type defined using `define-fungible-token`. The increased token balance is *not* transferred from another principal, but rather minted.

If a non-positive amount is provided to mint, this function returns `(err 1)`. Otherwise, on successfully mint, it returns `(ok true)`.

example:

Copy

```
(define-fungible-token stackaroo)
```

(ft-mint? stackaroo u100 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns (ok true)

ft-transfer?

Introduced in: **Clarity 1**

input: `TokenName, uint, principal, principal`

output: `(response bool uint)`

signature: `(ft-transfer? token-name amount sender recipient)`

description:

`ft-transfer?` is used to increase the token balance for the `recipient` principal for a token type defined using `define-fungible-token` by debiting the `sender` principal. In contrast to `stx-transfer?`, any user can transfer the assets. When used, relevant guards need to be added.

This function returns (ok true) if the transfer is successful. In the event of an unsuccessful transfer it returns one of the following error codes:

`(err u1)` -- `sender` does not have enough balance to transfer `(err u2)` -- `sender` and `recipient` are the same principal `(err u3)` -- amount to send is non-positive

example:

Copy

(define-fungible-token stackaroo)

(ft-mint? stackaroo u100 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)

(ft-transfer? stackaroo u50 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns (ok true)

(ft-transfer? stackaroo u60 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns (err u1)

get

Introduced in: **Clarity 1**

input: `KeyName, (tuple) | (optional (tuple))`

output: `A`

signature: `(get key-name tuple)`

description:

The `get` function fetches the value associated with a given key from the supplied typed tuple. If an `Optional` value is supplied as the inputted tuple, `get` returns an `Optional` type of the specified key in the tuple. If the supplied option is a `(none)` option, `get` returns `(none)`.

example:

Copy

```
(define-map names-map { name: (string-ascii 12) } { id: int })
```

```
(map-insert names-map { name: "blockstack" } { id: 1337 }) ;; Returns true
```

```
(get id (tuple (name "blockstack") (id 1337))) ;; Returns 1337
```

```
(get id (map-get? names-map (tuple (name "blockstack")))) ;; Returns (some 1337)
```

```
(get id (map-get? names-map (tuple (name "non-existent")))) ;; Returns none
```

get-block-info?

Introduced in: **Clarity 1**

input: `BlockInfoPropertyName, uint`

output: `(optional buff) | (optional uint)`

signature: `(get-block-info? prop-name block-height)`

description:

The `get-block-info?` function fetches data for a block of the given *Stacks* block height. The value and type returned are determined by the specified `BlockInfoPropertyName`. If the provided `block-height` does not correspond to an existing block prior to the current block, the function returns `none`. The currently available property names are as follows:

burnchain-header-hash: This property returns a `(buff 32)` value containing the header hash of the burnchain (Bitcoin) block that selected the Stacks block at the given Stacks chain height.

id-header-hash: This property returns a (buff 32) value containing the *index block hash* of a Stacks block. This hash is globally unique, and is derived from the block hash and the history of accepted PoX operations. This is also the block hash value you would pass into (at-block).

header-hash: This property returns a (buff 32) value containing the header hash of a Stacks block, given a Stacks chain height. **WARNING* this hash is not guaranteed to be globally unique, since the same Stacks block can be mined in different PoX forks. If you need global uniqueness, you should use **id-header-hash**.

miner-address: This property returns a **principal** value corresponding to the miner of the given block. **WARNING** In Stacks 2.1, this is not guaranteed to be the same **principal** that received the block reward, since Stacks 2.1 supports coinbase transactions that pay the reward to a contract address. This is merely the address of the **principal** that produced the block.

time: This property returns a **uint** value of the block header time field. This is a Unix epoch timestamp in seconds which roughly corresponds to when the block was mined. **Note:** this does not increase monotonically with each block and block times are accurate only to within two hours. See [BIP113](#) for more information.

New in Stacks 2.1:

block-reward: This property returns a **uint** value for the total block reward of the indicated Stacks block. This value is only available once the reward for the block matures. That is, the latest **block-reward** value available is at least 101 Stacks blocks in the past (on mainnet). The reward includes the coinbase, the anchored block's transaction fees, and the shares of the confirmed and produced microblock transaction fees earned by this block's miner. Note that this value may be smaller than the Stacks coinbase at this height, because the miner may have been punished with a valid **PoisonMicroblock** transaction in the event that the miner published two or more microblock stream forks.

miner-spend-total: This property returns a **uint** value for the total number of burnchain tokens (i.e. satoshis) spent by all miners trying to win this block.

miner-spend-winner: This property returns a **uint** value for the number of burnchain tokens (i.e. satoshis) spent by the winning miner for this Stacks block. Note that this value is less than or equal to the value for **miner-spend-total** at the same block height.

example:

Copy

(get-block-info? time u0) ;; Returns (some u1557860301)

(get-block-info? header-hash u0) ;; Returns (some
0x374708fff7719dd5979ec875d56cd2286f6d3cf7ec317a3b25632aab28ec37bb)

(get-block-info? vrf-seed u0) ;; Returns (some 0xf490de2920c8a35fabeb13208852aa28c76f9be9b03a4dd2b3c075f7a26923b4)

get-burn-block-info?

Introduced in: **Clarity 2**

input: `BurnBlockInfoPropertyName`, `uint`

output: `(optional buff) | (optional (tuple (addrs (list 2 (tuple (hashbytes (buff 32)) (version (buff 1))))) (payout uint)))`

signature: `(get-burn-block-info? prop-name block-height)`

description:

The `get-burn-block-info?` function fetches data for a block of the given *burnchain* block height. The value and type returned are determined by the specified `BlockInfoPropertyName`. Valid values for `block-height` only include heights between the burnchain height at the time the Stacks chain was launched, and the last-processed burnchain block. If the `block-height` argument falls outside of this range, then `none` shall be returned.

The following `BlockInfoPropertyName` values are defined:

The `header-hash` property returns a 32-byte buffer representing the header hash of the burnchain block at burnchain height `block-height`.

The `pox-addrs` property returns a tuple with two items: a list of up to two PoX addresses that received a PoX payout at that block height, and the amount of burnchain tokens paid to each address (note that per the blockchain consensus rules, each PoX payout will be the same for each address in the block-commit transaction). The list will include burn addresses -- that is, the unspendable addresses that miners pay to when there are no PoX addresses left to be paid. During the prepare phase, there will be exactly one burn address reported. During the reward phase, up to two burn addresses may be reported in the event that some PoX reward slots are not claimed.

The `addrs` list contains the same PoX address values passed into the PoX smart contract:

They each have type signature `(tuple (hashbytes (buff 32)) (version (buff 1)))`

The `version` field can be any of the following:

`0x00` means this is a p2pkh address, and `hashbytes` is the 20-byte hash160 of a single public key

`0x01` means this is a p2sh address, and `hashbytes` is the 20-byte hash160 of a redeemScript script

`0x02` means this is a p2wpkh-p2sh address, and `hashbytes` is the 20-byte hash160 of a p2wpkh witness script

`0x03` means this is a p2wsh-p2sh address, and `hashbytes` is the 20-byte hash160 of a p2wsh witness script

`0x04` means this is a p2wpkh address, and `hashbytes` is the 20-byte hash160 of the witness script

`0x05` means this is a p2wsh address, and `hashbytes` is the 32-byte sha256 of the witness script

`0x06` means this is a p2tr address, and `hashbytes` is the 32-byte sha256 of the witness script

example:

Copy

```
(get-burn-block-info? header-hash u677050) ;; Returns (some  
0xe67141016c88a7f1203eca0b4312f2ed141531f59303a1c267d7d83ab6b977d8)
```

```
(get-burn-block-info? pox-addr u677050) ;; Returns (some (tuple (addr ( (tuple (hashbytes  
0x395f3643cea07ec4eec73b4d9a973dcce56b9bf1) (version 0x00)) (tuple (hashbytes  
0x7c6775e20e3e938d2d7e9d79ac310108ba501ddb) (version 0x01)))) (payout u123)))
```

hash160

Introduced in: **Clarity 1**

input: `buff|uint|int`

output: `(buff 20)`

signature: `(hash160 value)`

description:

The `hash160` function computes `RIPEMD160(SHA256(x))` of the inputted value. If an integer (128 bit) is supplied the hash is computed over the little-endian representation of the integer.

example:

Copy

(hash160 0) ;; Returns 0xe4352f72356db555721651aa612e00379167b30f

if

Introduced in: **Clarity 1**

input: `bool`, `A`, `A`

output: `A`

signature: `(if bool1 expr1 expr2)`

description:

The `if` function admits a boolean argument and two expressions which must return the same type. In the case that the boolean input is `true`, the `if` function evaluates and returns `expr1`. If the boolean input is `false`, the `if` function evaluates and returns `expr2`.

example:

Copy

(if true 1 2) ;; Returns 1

(if (> 1 2) 1 2) ;; Returns 2

impl-trait

Introduced in: **Clarity 1**

input: `TraitIdentifier`

output: `Not Applicable`

signature: `(impl-trait trait-identifier)`

description:

`impl-trait` can be use for asserting that a contract is fully implementing a given trait. Additional checks are being performed when the contract is being published, rejecting the deployment if the contract is violating the trait specification.

Trait identifiers can either be using the sugared syntax (`.token-a.token-trait`), or be fully qualified (`'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF.token-a.token-trait`).

Like other kinds of definition statements, `impl-trait` may only be used at the top level of a smart contract definition (i.e., you cannot put such a statement in the middle of a function body).

example:

Copy

```
(impl-trait 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF.token-a.token-trait)
```

```
(define-public (get-balance (account principal))
```

```
  (ok u0))
```

```
(define-public (transfer? (from principal) (to principal) (amount uint))
```

```
  (ok u0))
```

index-of

Introduced in: **Clarity 1**

input: `sequence_A`, `A`

output: `(optional uint)`

signature: `(index-of? sequence item)`

description:

The `index-of?` function returns the first index at which `item` can be found, using `is-eq` checks, in the provided sequence. Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`, for which the corresponding element types are, respectively, `A`, `(buff 1)`, `(string-ascii 1)` and `(string-utf8 1)`. If the target item is not found in the sequence (or if an empty string or buffer is supplied), this function returns `none`. In Clarity1, `index-of` must be used (without the `?`). The `?` is added in Clarity2 for consistency -- built-ins that return responses or optionals end in `?`. The Clarity1 spelling is left as an alias in Clarity2 for backwards compatibility.

example:

Copy

```
(index-of? "blockstack" "b") ;; Returns (some u0)
```

```
(index-of? "blockstack" "k") ;; Returns (some u4)
```

`(index-of? "blockstack" "")` ;; Returns none

`(index-of? (list 1 2 3 4 5) 6)` ;; Returns none

`(index-of? 0xfb01 0x01)` ;; Returns (some u1)

index-of?

Introduced in: **Clarity 2**

input: `sequence_A, A`

output: `(optional uint)`

signature: `(index-of? sequence item)`

description:

The `index-of?` function returns the first index at which `item` can be found, using `is-eq` checks, in the provided sequence. Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`, for which the corresponding element types are, respectively, `A`, `(buff 1)`, `(string-ascii 1)` and `(string-utf8 1)`. If the target item is not found in the sequence (or if an empty string or buffer is supplied), this function returns `none`. In Clarity1, `index-of` must be used (without the `?`). The `?` is added in Clarity2 for consistency -- built-ins that return responses or optionals end in `?`. The Clarity1 spelling is left as an alias in Clarity2 for backwards compatibility.

example:

Copy

`(index-of? "blockstack" "b")` ;; Returns (some u0)

`(index-of? "blockstack" "k")` ;; Returns (some u4)

`(index-of? "blockstack" "")` ;; Returns none

`(index-of? (list 1 2 3 4 5) 6)` ;; Returns none

`(index-of? 0xfb01 0x01)` ;; Returns (some u1)

int-to-ascii

Introduced in: **Clarity 2**

input: `int | uint`

output: `(string-ascii 40)`

signature: `(int-to-ascii (int|uint))`

description:

Converts an integer, either `int` or `uint`, to a `string-ascii` string-value representation.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

`(int-to-ascii 1) ;; Returns "1"`

`(int-to-ascii u1) ;; Returns "1"`

`(int-to-ascii -1) ;; Returns "-1"`

int-to-utf8

Introduced in: **Clarity 2**

input: `int | uint`

output: `(string-utf8 40)`

signature: `(int-to-utf8 (int|uint))`

description:

Converts an integer, either `int` or `uint`, to a `string-utf8` string-value representation.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

`(int-to-utf8 1) ;; Returns u"1"`

`(int-to-utf8 u1) ;; Returns u"1"`

`(int-to-utf8 -1) ;; Returns u"-1"`

is-eq

Introduced in: **Clarity 1**

input: `A, A, ...`

output: `bool`

signature: `(is-eq v1 v2...)`

description:

Compares the inputted values, returning `true` if they are all equal. Note that *unlike* the `(and ...)` function, `(is-eq ...)` will *not* short-circuit. All values supplied to `is-eq` *must* be the same type.

example:

Copy

`(is-eq 1 1) ;; Returns true`

`(is-eq true false) ;; Returns false`

`(is-eq "abc" 234 234) ;; Throws type error`

`(is-eq "abc" "abc") ;; Returns true`

`(is-eq 0x0102 0x0102) ;; Returns true`

is-err

Introduced in: **Clarity 1**

input: `(response A B)`

output: `bool`

signature: `(is-err value)`

description:

`is-err` tests a supplied response value, returning `true` if the response was an `err`, and `false` if it was an `ok`.

example:

Copy

```
(is-err (ok 1)) ;; Returns false
```

```
(is-err (err 1)) ;; Returns true
```

is-none

Introduced in: **Clarity 1**

input: (optional A)

output: bool

signature: (is-none value)

description:

`is-none` tests a supplied option value, returning `true` if the option value is `(none)`, and `false` if it is a `(some ...)`.

example:

Copy

```
(define-map names-map { name: (string-ascii 12) } { id: int })
```

```
(map-set names-map { name: "blockstack" } { id: 1337 })
```

```
(is-none (get id (map-get? names-map { name: "blockstack" }))) ;; Returns false
```

```
(is-none (get id (map-get? names-map { name: "non-existent" }))) ;; Returns true
```

is-ok

Introduced in: **Clarity 1**

input: (response A B)

output: bool

signature: (is-ok value)

description:

`is-ok` tests a supplied response value, returning `true` if the response was `ok`, and `false` if it was an `err`.

example:

Copy

```
(is-ok (ok 1)) ;; Returns true
```

```
(is-ok (err 1)) ;; Returns false
```

is-some

Introduced in: **Clarity 1**

input: `(optional A)`

output: `bool`

signature: `(is-some value)`

description:

`is-some` tests a supplied option value, returning `true` if the option value is `(some ...)`, and `false` if it is a `none`.

example:

Copy

```
(define-map names-map { name: (string-ascii 12) } { id: int })
```

```
(map-set names-map { name: "blockstack" } { id: 1337 })
```

```
(is-some (get id (map-get? names-map { name: "blockstack" }))) ;; Returns true
```

```
(is-some (get id (map-get? names-map { name: "non-existent" }))) ;; Returns false
```

is-standard

Introduced in: **Clarity 2**

input: `principal`

output: `bool`

signature: (is-standard standard-or-contract-principal)

description:

Tests whether `standard-or-contract-principal` *matches* the current network type, and therefore represents a principal that can spend tokens on the current network type. That is, the network is either of type `mainnet`, or `testnet`. Only `SPxxxx` and `SMxxxx` *c32check form* addresses can spend tokens on a mainnet, whereas only `STxxxx` and `SNxxxx` *c32check forms* addresses can spend tokens on a testnet. All addresses can *receive* tokens, but only principal *c32check form* addresses that match the network type can *spend* tokens on the network. This method will return `true` if and only if the principal matches the network type, and false otherwise.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

(is-standard 'STB44HYPYAT2BB2QE513NSP81HTMYWBJP02HPGK6) ;; returns true on testnet and false on mainnet

(is-standard 'STB44HYPYAT2BB2QE513NSP81HTMYWBJP02HPGK6.foo) ;; returns true on testnet and false on mainnet

(is-standard 'SP3X6QWWETNBZWGBK6DRGTR1KX50S74D3433WDGJY) ;; returns true on mainnet and false on testnet

(is-standard 'SP3X6QWWETNBZWGBK6DRGTR1KX50S74D3433WDGJY.foo) ;; returns true on mainnet and false on testnet

(is-standard 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR) ;; returns false on both mainnet and testnet

keccak256

Introduced in: **Clarity 1**

input: `buff|uint|int`

output: (`buff` 32)

signature: (`keccak256` `value`)

description:

The `keccak256` function computes `KECCAK256(value)` of the inputted value. Note that this differs from the `NIST SHA-3` (that is, FIPS 202) standard. If an integer (128 bit) is supplied the hash is computed over the little-endian representation of the integer.

example:

Copy

`(keccak256 0) ;; Returns`

`0xf490de2920c8a35fabeb13208852aa28c76f9be9b03a4dd2b3c075f7a26923b4`

len

Introduced in: **Clarity 1**

input: `sequence_A`

output: `uint`

signature: `(len sequence)`

description:

The `len` function returns the length of a given sequence. Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`.

example:

Copy

`(len "blockstack") ;; Returns u10`

`(len (list 1 2 3 4 5)) ;; Returns u5`

`(len 0x010203) ;; Returns u3`

let

Introduced in: **Clarity 1**

input: `((name1 AnyType) (name2 AnyType) ...), AnyType, ... A`

output: `A`

signature: `(let ((name1 expr1) (name2 expr2) ...) expr-body1 expr-body2 ...
expr-body-last)`

description:

The `let` function accepts a list of `variable name` and `expression` pairs, evaluating each expression and *binding* it to the corresponding variable name. `let` bindings are sequential: when a `let` binding is evaluated, it may refer to prior binding. The *context* created by this set of bindings is used for evaluating its body expressions. The `let` expression returns the value of the last such body expression. Note: intermediary statements returning a response type must be checked

example:

Copy

```
(let
```

```
  ( (a 2) (b (+ 5 6 7)))
```

```
  (print a)
```

```
  (print b)
```

```
  (+ a b)) ;; Returns 20
```

```
(let
```

```
  ( (a 5) (c (+ a 1)) (d (+ c 1)) (b (+ a c d)))
```

```
  (print a)
```

```
  (print b)
```

```
  (+ a b)) ;; Returns 23
```

list

Introduced in: **Clarity 1**

input: `A, ...`

output: `(list A)`

signature: `(list expr1 expr2 expr3 ...)`

description:

The `list` function constructs a list composed of the inputted values. Each supplied value must be of the same type.

example:

Copy

(list (+ 1 2) 4 5) ;; Returns (3 4 5)

log2

Introduced in: **Clarity 1**

input: `int | uint`

output: `int | uint`

signature: `(log2 n)`

description:

Returns the power to which the number 2 must be raised to to obtain the value `n`, rounded down to the nearest integer. Fails on a negative numbers.

example:

Copy

(log2 u8) ;; Returns u3

(log2 8) ;; Returns 3

(log2 u1) ;; Returns u0

(log2 1000) ;; Returns 9

map

Introduced in: **Clarity 1**

input: `Function(A, B, ..., N) -> X, sequence_A, sequence_B, ..., sequence_N`

output: `(list X)`

signature: `(map func sequence_A sequence_B ... sequence_N)`

description:

The `map` function applies the function `func` to each corresponding element of the input sequences, and outputs a *list* of the same type containing the outputs from those function applications. Applicable sequence types are `(list A)`, `buff`, `string-ascii` and `string-utf8`, for which the corresponding element types are, respectively, `A`, `(buff 1)`, `(string-ascii 1)` and `(string-utf8 1)`. The `func` argument must be a literal function name. Also, note that, no matter what kind of sequences the inputs are, the output is always a list.

example:

Copy

```
(map not (list true false true false)) ;; Returns (false true false true)
```

```
(map + (list 1 2 3) (list 1 2 3) (list 1 2 3)) ;; Returns (3 6 9)
```

```
(define-private (a-or-b (char (string-utf8 1)))
```

```
  (if (is-eq char u"a") u"a" u"b"))
```

```
(map a-or-b u"aca") ;; Returns (u"a" u"b" u"a")
```

```
(define-private (zero-or-one (char (buff 1)))
```

```
  (if (is-eq char 0x00) 0x00 0x01))
```

```
(map zero-or-one 0x000102) ;; Returns (0x00 0x01 0x01)
```

map-delete

Introduced in: **Clarity 1**

input: `MapName`, `tuple`

output: `bool`

signature: `(map-delete map-name key-tuple)`

description:

The `map-delete` function removes the value associated with the input key for the given map. If an item exists and is removed, the function returns `true`. If a value did not exist for this key in the data map, the function returns `false`.

example:

Copy

```
(define-map names-map { name: (string-ascii 10) } { id: int })
```

```
(map-insert names-map { name: "blockstack" } { id: 1337 }) ;; Returns true
```

```
(map-delete names-map { name: "blockstack" }) ;; Returns true
```

```
(map-delete names-map { name: "blockstack" }) ;; Returns false
```

```
(map-delete names-map (tuple (name "blockstack"))) ;; Same command, using a shorthand for constructing the tuple
```

map-get?

Introduced in: **Clarity 1**

input: `MapName`, `tuple`

output: `(optional (tuple))`

signature: `(map-get? map-name key-tuple)`

description:

The `map-get?` function looks up and returns an entry from a contract's data map. The value is looked up using `key-tuple`. If there is no value associated with that key in the data map, the function returns a `none` option. Otherwise, it returns `(some value)`.

example:

Copy

```
(define-map names-map { name: (string-ascii 10) } { id: int })
```

```
(map-set names-map { name: "blockstack" } { id: 1337 })
```

```
(map-get? names-map (tuple (name "blockstack"))) ;; Returns (some (tuple (id 1337)))
```

```
(map-get? names-map { name: "blockstack" }) ;; Same command, using a shorthand for constructing the tuple
```

map-insert

Introduced in: **Clarity 1**

input: `MapName, tuple_A, tuple_B`

output: `bool`

signature: `(map-insert map-name key-tuple value-tuple)`

description:

The `map-insert` function sets the value associated with the input key to the inputted value if and only if there is not already a value associated with the key in the map. If an insert occurs, the function returns `true`. If a value already existed for this key in the data map, the function returns `false`.

Note: the `value-tuple` requires 1 additional byte for storage in the materialized blockchain state, and therefore the maximum size of a value that may be inserted into a map is `MAX_CLARITY_VALUE - 1`.

example:

Copy

```
(define-map names-map { name: (string-ascii 10) } { id: int })
```

```
(map-insert names-map { name: "blockstack" } { id: 1337 }) ;; Returns true
```

```
(map-insert names-map { name: "blockstack" } { id: 1337 }) ;; Returns false
```

```
(map-insert names-map (tuple (name "blockstack")) (tuple (id 1337))) ;; Same command, using  
a shorthand for constructing the tuple
```

map-set

Introduced in: **Clarity 1**

input: `MapName, tuple_A, tuple_B`

output: `bool`

signature: `(map-set map-name key-tuple value-tuple)`

description:

The `map-set` function sets the value associated with the input key to the inputted value. This function performs a *blind* update; whether or not a value is already associated with the key, the function overwrites that existing association.

Note: the `value-tuple` requires 1 additional byte for storage in the materialized blockchain state, and therefore the maximum size of a value that may be inserted into a map is `MAX_CLARITY_VALUE - 1`.

example:

Copy

```
(define-map names-map { name: (string-ascii 10) } { id: int })
```

```
(map-set names-map { name: "blockstack" } { id: 1337 }) ;; Returns true
```

```
(map-set names-map (tuple (name "blockstack")) (tuple (id 1337))) ;; Same command, using a shorthand for constructing the tuple
```

match

Introduced in: **Clarity 1**

input: (optional A) name expression expression | (response A B) name expression
name expression

output: C

signature: (match opt-input some-binding-name some-branch none-branch) |
(match-resp input ok-binding-name ok-branch err-binding-name err-branch)

description:

The `match` function is used to test and destructure optional and response types.

If the `input` is an optional, it tests whether the provided `input` is a `some` or `none` option, and evaluates `some-branch` or `none-branch` in each respective case.

Within the `some-branch`, the *contained value* of the `input` argument is bound to the provided `some-binding-name` name.

Only *one* of the branches will be evaluated (similar to `if` statements).

If the `input` is a response, it tests whether the provided `input` is an `ok` or `err` response type, and evaluates `ok-branch` or `err-branch` in each respective case.

Within the `ok-branch`, the *contained ok value* of the `input` argument is bound to the provided `ok-binding-name` name.

Within the `err-branch`, the *contained err value* of the `input` argument is bound to the provided `err-binding-name` name.

Only *one* of the branches will be evaluated (similar to `if` statements).

Note: Type checking requires that the type of both the `ok` and `err` parts of the response object be determinable. For situations in which one of the parts of a response is untyped, you should use `unwrap-panic` or `unwrap-err-panic` instead of `match`.

example:

Copy

```
(define-private (add-10 (x (optional int)))
```

```
  (match x
    (value (+ 10 value)
      10))
```

```
(add-10 (some 5)) ;; Returns 15
```

```
(add-10 none) ;; Returns 10
```

```
(define-private (add-or-pass-err (x (response int (string-ascii 10))) (to-add int))
```

```
  (match x
    (value (ok (+ to-add value))
      err-value (err err-value)))
```

```
(add-or-pass-err (ok 5) 20) ;; Returns (ok 25)
```

```
(add-or-pass-err (err "ERROR") 20) ;; Returns (err "ERROR")
```

merge

Introduced in: **Clarity 1**

input: `tuple`, `tuple`

output: `tuple`

signature: `(merge tuple { key1: val1 })`

description:

The `merge` function returns a new tuple with the combined fields, without mutating the supplied tuples.

example:

Copy

```
(define-map users { id: int } { name: (string-ascii 12), address: (optional principal) })
```

```
(map-insert users { id: 1337 } { name: "john", address: none }) ;; Returns true
```

```
(let
```

```
  ( (user (unwrap-panic (map-get? users { id: 1337 }))))
```

```
  (merge user { address: (some 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) })) ;;  
Returns (tuple (address (some SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF))  
(name "john"))
```

mod

Introduced in: **Clarity 1**

input: `int, int | uint, uint | string-ascii, string-ascii | string-utf8,
string-utf8 | buff, buff`

output: `int | uint`

signature: `(mod i1 i2)`

description:

Returns the integer remainder from integer dividing `i1` by `i2`. In the event of a division by zero, throws a runtime error.

example:

Copy

```
(mod 2 3) ;; Returns 2
```

(mod 5 2) ;; Returns 1

(mod 7 1) ;; Returns 0

nft-burn?

Introduced in: **Clarity 1**

input: `AssetName, A, principal`

output: `(response bool uint)`

signature: `(nft-burn? asset-class asset-identifier sender)`

description:

`nft-burn?` is used to burn an asset that the `sender` principal owns. The asset must have been defined using `define-non-fungible-token`, and the supplied `asset-identifier` must be of the same type specified in that definition.

On a successful burn, it returns `(ok true)`. In the event of an unsuccessful burn it returns one of the following error codes:

`(err u1)` -- `sender` does not own the specified asset `(err u3)` -- the asset specified by `asset-identifier` does not exist

example:

Copy

```
(define-non-fungible-token stackaroo (string-ascii 40))
```

```
(nft-mint? stackaroo "Roo" 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns  
(ok true)
```

```
(nft-burn? stackaroo "Roo" 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns  
(ok true)
```

nft-get-owner?

Introduced in: **Clarity 1**

input: `AssetName, A`

output: (optional principal)

signature: (nft-get-owner? asset-class asset-identifier)

description:

`nft-get-owner?` returns the owner of an asset, identified by `asset-identifier`, or `none` if the asset does not exist. The asset type must have been defined using `define-non-fungible-token`, and the supplied `asset-identifier` must be of the same type specified in that definition.

example:

Copy

```
(define-non-fungible-token stackaroo (string-ascii 40))
```

```
(nft-mint? stackaroo "Roo" 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF)
```

```
(nft-get-owner? stackaroo "Roo") ;; Returns (some  
SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF)
```

```
(nft-get-owner? stackaroo "Too") ;; Returns none
```

nft-mint?

Introduced in: **Clarity 1**

input: `AssetName`, `A`, `principal`

output: (response bool uint)

signature: (nft-mint? asset-class asset-identifier recipient)

description:

`nft-mint?` is used to instantiate an asset and set that asset's owner to the `recipient` principal. The asset must have been defined using `define-non-fungible-token`, and the supplied `asset-identifier` must be of the same type specified in that definition.

If an asset identified by `asset-identifier` *already exists*, this function will return an error with the following error code:

```
(err u1)
```

Otherwise, on successfully mint, it returns `(ok true)`.

example:

Copy

```
(define-non-fungible-token stackaroo (string-ascii 40))
```

```
(nft-mint? stackaroo "Roo" 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns (ok true)
```

nft-transfer?

Introduced in: **Clarity 1**

input: `AssetName, A, principal, principal`

output: `(response bool uint)`

signature: `(nft-transfer? asset-class asset-identifier sender recipient)`

description:

`nft-transfer?` is used to change the owner of an asset identified by `asset-identifier` from `sender` to `recipient`. The `asset-class` must have been defined by `define-non-fungible-token` and `asset-identifier` must be of the type specified in that definition. In contrast to `stx-transfer?`, any user can transfer the asset. When used, relevant guards need to be added.

This function returns `(ok true)` if the transfer is successful. In the event of an unsuccessful transfer it returns one of the following error codes:

`(err u1)` -- `sender` does not own the asset `(err u2)` -- `sender` and `recipient` are the same principal `(err u3)` -- asset identified by `asset-identifier` does not exist

example:

Copy

```
(define-non-fungible-token stackaroo (string-ascii 40))
```

```
(nft-mint? stackaroo "Roo" 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)
```

```
(nft-transfer? stackaroo "Roo" 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR 'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns (ok true)
```

```
(nft-transfer? stackaroo "Roo" 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR  
'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns (err u1)
```

```
(nft-transfer? stackaroo "Stacka" 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR  
'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF) ;; Returns (err u3)
```

not

Introduced in: **Clarity 1**

input: `bool`

output: `bool`

signature: `(not b1)`

description:

Returns the inverse of the boolean input.

example:

Copy

`(not true) ;; Returns false`

`(not (is-eq 1 2)) ;; Returns true`

ok

Introduced in: **Clarity 1**

input: `A`

output: `(response A B)`

signature: `(ok value)`

description:

The `ok` function constructs a response type from the input value. Use `ok` for creating return values in public functions. An `ok` value indicates that any database changes during the processing of the function should materialize.

example:

Copy

(ok 1) ;; Returns (ok 1)

or

Introduced in: **Clarity 1**

input: `bool, ...`

output: `bool`

signature: `(or b1 b2 ...)`

description:

Returns `true` if any boolean inputs are `true`. Importantly, the supplied arguments are evaluated in-order and lazily. Lazy evaluation means that if one of the arguments returns `true`, the function short-circuits, and no subsequent arguments are evaluated.

example:

Copy

(or true false) ;; Returns true

(or (is-eq (+ 1 2) 1) (is-eq 4 4)) ;; Returns true

(or (is-eq (+ 1 2) 1) (is-eq 3 4)) ;; Returns false

(or (is-eq (+ 1 2) 3) (is-eq 4 4)) ;; Returns true

pow

Introduced in: **Clarity 1**

input: `int, int | uint, uint | string-ascii, string-ascii | string-utf8, string-utf8 | buff, buff`

output: `int | uint`

signature: `(pow i1 i2)`

description:

Returns the result of raising `i1` to the power of `i2`. In the event of an *overflow*, throws a runtime error. Note: Corner cases are handled with the following rules:

- if both `i1` and `i2` are `0`, return `1`
- if `i1` is `1`, return `1`
- if `i1` is `0`, return `0`
- if `i2` is `1`, return `i1`
- if `i2` is negative or greater than `u32::MAX`, throw a runtime error

example:

Copy

(pow 2 3) ;; Returns 8

(pow 2 2) ;; Returns 4

(pow 7 1) ;; Returns 7

principal-construct?

Introduced in: **Clarity 2**

input: (buff 1), (buff 20), [(string-ascii 40)]

output: (response principal { error_code: uint, principal: (option principal) })

signature: (principal-construct? (buff 1) (buff 20) [(string-ascii 40)])

description:

A principal value represents either a set of keys, or a smart contract. The former, called a *standard principal*, is encoded as a (buff 1) *version byte*, indicating the type of account and the type of network that this principal can spend tokens on, and a (buff 20) *public key hash*, characterizing the principal's unique identity. The latter, a *contract principal*, is encoded as a standard principal concatenated with a (string-ascii 40) *contract name* that identifies the code body.

The `principal-construct?` function allows users to create either standard or contract principals, depending on which form is used. To create a standard principal, `principal-construct?` would be called with two arguments: it takes as input a (buff 1) which encodes the principal address's *version-byte*, a (buff 20) which encodes the principal address's *hash-bytes*. To create a contract principal, `principal-construct?` would be called

with three arguments: the (`buff 1`) and (`buff 20`) to represent the standard principal that created the contract, and a (`string-ascii 40`) which encodes the contract's name. On success, this function returns either a standard principal or contract principal, depending on whether or not the third (`string-ascii 40`) argument is given.

This function returns a `Response`. On success, the `ok` value is a `Principal`. The `err` value is a value tuple with the form `{ error_code: uint, value: (optional principal) }`.

If the single-byte `version-byte` is in the valid range `0x00` to `0x1f`, but is not an appropriate version byte for the current network, then the error will be `u0`, and `value` will contain (`some principal`), where the wrapped value is the principal. If the `version-byte` is not in this range, however, then the `value` will be `none`.

If the `version-byte` is a `buff` of length 0, if the single-byte `version-byte` is a value greater than `0x1f`, or the `hash-bytes` is a `buff` of length not equal to 20, then `error_code` will be `u1` and `value` will be `None`.

If a name is given, and the name is either an empty string or contains ASCII characters that are not allowed in contract names, then `error_code` will be `u2`.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

```
(principal-construct? 0x1a 0xfa6bf38ed557fe417333710d6033e9419391a320) ;; Returns (ok ST3X6QWWETNBZWGBK6DRGTR1KX50S74D3425Q1TPK)
```

```
(principal-construct? 0x1a 0xfa6bf38ed557fe417333710d6033e9419391a320 "foo") ;; Returns (ok ST3X6QWWETNBZWGBK6DRGTR1KX50S74D3425Q1TPK.foo)
```

```
(principal-construct? 0x16 0xfa6bf38ed557fe417333710d6033e9419391a320) ;; Returns (err (tuple (error_code u0) (value (some SP3X6QWWETNBZWGBK6DRGTR1KX50S74D3433WDGJY))))
```

```
(principal-construct? 0x16 0xfa6bf38ed557fe417333710d6033e9419391a320 "foo") ;; Returns (err (tuple (error_code u0) (value (some SP3X6QWWETNBZWGBK6DRGTR1KX50S74D3433WDGJY.foo))))
```

```
(principal-construct? 0x 0xfa6bf38ed557fe417333710d6033e9419391a320) ;; Returns (err (tuple (error_code u1) (value none)))
```

```
(principal-construct? 0x16 0xfa6bf38ed557fe417333710d6033e9419391a3) ;; Returns (err (tuple (error_code u1) (value none)))
```

(principal-construct? 0x20 0xfa6bf38ed557fe417333710d6033e9419391a320) ;; Returns (err (tuple (error_code u1) (value none)))

(principal-construct? 0x1a 0xfa6bf38ed557fe417333710d6033e9419391a320 "") ;; Returns (err (tuple (error_code u2) (value none)))

(principal-construct? 0x1a 0xfa6bf38ed557fe417333710d6033e9419391a320 "foo[") ;; Returns (err (tuple (error_code u2) (value none)))

principal-destruct?

Introduced in: **Clarity 2**

input: `principal`

output: (response (tuple (hash-bytes (buff 20)) (name (optional (string-ascii 40))) (version (buff 1))) (tuple (hash-bytes (buff 20)) (name (optional (string-ascii 40))) (version (buff 1)))))

signature: (principal-destruct? principal-address)

description:

A principal value represents either a set of keys, or a smart contract. The former, called a *standard principal*, is encoded as a (buff 1) *version byte*, indicating the type of account and the type of network that this principal can spend tokens on, and a (buff 20) *public key hash*, characterizing the principal's unique identity. The latter, a *contract principal*, is encoded as a standard principal concatenated with a (string-ascii 40) *contract name* that identifies the code body.

`principal-destruct?` will decompose a principal into its component parts: either {`version-byte`, `hash-bytes`} for standard principals, or {`version-byte`, `hash-bytes`, `name`} for contract principals.

This method returns a `Response` that wraps this data as a tuple.

If the version byte of `principal-address` matches the network (see `is-standard`), then this method returns the pair as its `ok` value.

If the version byte of `principal-address` does not match the network, then this method returns the pair as its `err` value.

In both cases, the value itself is a tuple containing three fields: a `version` value as a `(buff 1)`, a `hash-bytes` value as a `(buff 20)`, and a `name` value as an `(optional (string-ascii 40))`. The `name` field will only be `(some ..)` if the principal is a contract principal.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

```
(principal-destruct? 'STB44HYPYAT2BB2QE513NSP81HTMYWBJP02HPGK6) ;; Returns (ok (tuple (hash-bytes 0x164247d6f2b425ac5771423ae6c80c754f7172b0) (name none) (version 0x1a)))
```

```
(principal-destruct? 'STB44HYPYAT2BB2QE513NSP81HTMYWBJP02HPGK6.foo) ;; Returns (ok (tuple (hash-bytes 0x164247d6f2b425ac5771423ae6c80c754f7172b0) (name (some "foo")) (version 0x1a)))
```

```
(principal-destruct? 'SP3X6QWWETNBZWGBK6DRGTR1KX50S74D3433WDGJY) ;; Returns (err (tuple (hash-bytes 0xfa6bf38ed557fe417333710d6033e9419391a320) (name none) (version 0x16)))
```

```
(principal-destruct? 'SP3X6QWWETNBZWGBK6DRGTR1KX50S74D3433WDGJY.foo) ;; Returns (err (tuple (hash-bytes 0xfa6bf38ed557fe417333710d6033e9419391a320) (name (some "foo")) (version 0x16)))
```

principal-of?

Introduced in: **Clarity 1**

input: `(buff 33)`

output: `(response principal uint)`

signature: `(principal-of? public-key)`

description:

The `principal-of?` function returns the principal derived from the provided public key. If the `public-key` is invalid, it will return the error code `(err u1)`..

Note: Before Stacks 2.1, this function has a bug, in that the principal returned would always be a testnet single-signature principal, even if the function were run on the mainnet. Starting with Stacks 2.1, this bug is fixed, so that this function will return a principal suited to the network it is

called on. In particular, if this is called on the mainnet, it will return a single-signature mainnet principal.

example:

Copy

(principal-of?

0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns
(ok ST1AW6EKP6GT61SQ9FNVDS17RKNWT8ZP582VF9HSCP)

print

Introduced in: **Clarity 1**

input: A

output: A

signature: (print expr)

description:

The `print` function evaluates and returns its input expression. On Stacks Core nodes configured for development (as opposed to production mining nodes), this function prints the resulting value to `STDOUT` (standard output).

example:

Copy

(print (+ 1 2 3)) ;; Returns 6

replace-at?

Introduced in: **Clarity 2**

input: sequence_A, uint, A

output: (optional sequence_A)

signature: (replace-at? sequence index element)

description:

The `replace-at?` function takes in a sequence, an index, and an element, and returns a new sequence with the data at the index position replaced with the given element. The given element's type must match the type of the sequence, and must correspond to a single index of the input sequence. The return type on success is the same type as the input sequence.

If the provided index is out of bounds, this functions returns `none`.

example:

Copy

```
(replace-at? u"ab" u1 u"c") ;; Returns (some u"ac")
```

```
(replace-at? 0x00112233 u2 0x44) ;; Returns (some 0x00114433)
```

```
(replace-at? "abcd" u3 "e") ;; Returns (some "abce")
```

```
(replace-at? (list 1) u0 10) ;; Returns (some (10))
```

```
(replace-at? (list (list 1) (list 2)) u0 (list 33)) ;; Returns (some ( (33) (2)))
```

```
(replace-at? (list 1 2) u3 4) ;; Returns none
```

secp256k1-recover?

Introduced in: **Clarity 1**

input: (`buff 32`), (`buff 65`)

output: (`response (buff 33) uint`)

signature: (`secp256k1-recover? message-hash signature`)

description:

The `secp256k1-recover?` function recovers the public key used to sign the message which sha256 is `message-hash` with the provided `signature`. If the signature does not match, it will return the error code (`err u1`).. If the signature is invalid, it will return the error code (`err u2`).. The signature includes 64 bytes plus an additional recovery id (00..03) for a total of 65 bytes.

example:

Copy

```
(secp256k1-recover?
```

```
0xde5b9eb9e7c5592930eb2e30a01369c36586d872082ed8181ee83d2a0ec20f04
```

```
0x8738487ebe69b93d8e51583be8eee50bb4213fc49c767d329632730cc193b873554428fc936c
a3569afc15f1c9365f6591d6251a89fee9c9ac661116824d3a1301) ;; Returns (ok
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110)
```

secp256k1-verify

Introduced in: **Clarity 1**

input: (buff 32), (buff 64) | (buff 65), (buff 33)

output: bool

signature: (secp256k1-verify message-hash signature public-key)

description:

The `secp256k1-verify` function verifies that the provided signature of the message-hash was signed with the private key that generated the public key. The `message-hash` is the `sha256` of the message. The signature includes 64 bytes plus an optional additional recovery id (00..03) for a total of 64 or 65 bytes.

example:

Copy

```
(secp256k1-verify
```

```
0xde5b9eb9e7c5592930eb2e30a01369c36586d872082ed8181ee83d2a0ec20f04
0x8738487ebe69b93d8e51583be8eee50bb4213fc49c767d329632730cc193b873554428fc936c
a3569afc15f1c9365f6591d6251a89fee9c9ac661116824d3a1301
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns
true
```

```
(secp256k1-verify
```

```
0xde5b9eb9e7c5592930eb2e30a01369c36586d872082ed8181ee83d2a0ec20f04
0x8738487ebe69b93d8e51583be8ee50bb4213fc49c767d329632730cc193b873554428fc936c
a3569afc15f1c9365f6591d6251a89fee9c9ac661116824d3a13
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns
true
```

```
(secp256k1-verify
```

```
0x0000000000000000000000000000000000000000000000000000000000000000
0x0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns
false
```


sha256

Introduced in: **Clarity 1**

input: `buff|uint|int`

output: `(buff 32)`

signature: `(sha256 value)`

description:

The `sha256` function computes `SHA256(x)` of the inputted value. If an integer (128 bit) is supplied the hash is computed over the little-endian representation of the integer.

example:

Copy

`(sha256 0) ;; Returns`

`0x374708fff7719dd5979ec875d56cd2286f6d3cf7ec317a3b25632aab28ec37bb`

sha512

Introduced in: **Clarity 1**

input: `buff|uint|int`

output: `(buff 64)`

signature: `(sha512 value)`

description:

The `sha512` function computes `SHA512(x)` of the inputted value. If an integer (128 bit) is supplied the hash is computed over the little-endian representation of the integer.

example:

Copy

`(sha512 1) ;; Returns`

`0x6fcee9a7b7a7b821d241c03c82377928bc6882e7a08c78a4221199bfa220cdc55212273018ee613317c8293bb8d1ce08d1e017508e94e06ab85a734c99c7cc34`

sha512/256

Introduced in: **Clarity 1**

input: `buff|uint|int`

output: `(buff 32)`

signature: `(sha512/256 value)`

description:

The `sha512/256` function computes `SHA512/256(x)` (the SHA512 algorithm with the 512/256 initialization vector, truncated to 256 bits) of the inputted value. If an integer (128 bit) is supplied the hash is computed over the little-endian representation of the integer.

example:

Copy

`(sha512/256 1) ;;` Returns

`0x515a7e92e7c60522db968d81ff70b80818fc17aeabbec36baf0dda2812e94a86`

slice?

Introduced in: **Clarity 2**

input: `sequence_A, uint, uint`

output: `(optional sequence_A)`

signature: `(slice? sequence left-position right-position)`

description:

The `slice?` function attempts to return a sub-sequence of that starts at `left-position` (inclusive), and ends at `right-position` (non-inclusive). If `left_position==right_position`, the function returns an empty sequence. If either `left_position` or `right_position` are out of bounds OR if `right_position` is less than `left_position`, the function returns `none`.

example:

Copy

`(slice? "blockstack" u5 u10) ;;` Returns (some "stack")

`(slice? (list 1 2 3 4 5) u5 u9) ;;` Returns none

`(slice? (list 1 2 3 4 5) u3 u4) ;;` Returns (some (4))

(slice? "abcd" u1 u3) ;; Returns (some "bc")

(slice? "abcd" u2 u2) ;; Returns (some "")

(slice? "abcd" u3 u1) ;; Returns none

some

Introduced in: **Clarity 1**

input: `A`

output: `(optional A)`

signature: `(some value)`

description:

The `some` function constructs a `optional` type from the input value.

example:

Copy

(some 1) ;; Returns (some 1)

(is-none (some 2)) ;; Returns false

sqrtn

Introduced in: **Clarity 1**

input: `int | uint`

output: `int | uint`

signature: `(sqrtn n)`

description:

Returns the largest integer that is less than or equal to the square root of `n`.

Fails on a negative numbers.

example:

Copy

(sqrti u11) ;; Returns u3

(sqrti 1000000) ;; Returns 1000

(sqrti u1) ;; Returns u1

(sqrti 0) ;; Returns 0

string-to-int?

Introduced in: **Clarity 2**

input: (`string-ascii` 1048576) | (`string-utf8` 262144)

output: (`optional int`)

signature: (`string-to-int?` (`string-ascii`|`string-utf8`))

description:

Converts a string, either `string-ascii` or `string-utf8`, to an optional-wrapped signed integer. If the input string does not represent a valid integer, then the function returns `none`. Otherwise it returns an integer wrapped in `some`.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

(string-to-int? "1") ;; Returns (some 1)

(string-to-int? u"-1") ;; Returns (some -1)

(string-to-int? "a") ;; Returns none

string-to-uint?

Introduced in: **Clarity 2**

input: (`string-ascii` 1048576) | (`string-utf8` 262144)

output: `(optional uint)`

signature: `(string-to-uint? (string-ascii|string-utf8))`

description:

Converts a string, either `string-ascii` or `string-utf8`, to an optional-wrapped unsigned integer. If the input string does not represent a valid integer, then the function returns `none`. Otherwise it returns an unsigned integer wrapped in `some`.

Note: This function is only available starting with Stacks 2.1.

example:

Copy

`(string-to-uint? "1") ;; Returns (some u1)`

`(string-to-uint? u"1") ;; Returns (some u1)`

`(string-to-uint? "a") ;; Returns none`

stx-account

Introduced in: **Clarity 2**

input: `principal`

output: `(tuple (locked uint) (unlock-height uint) (unlocked uint))`

signature: `(stx-account owner)`

description:

`stx-account` is used to query the STX account of the `owner` principal.

This function returns a tuple with the canonical account representation for an STX account. This includes the current amount of unlocked STX, the current amount of locked STX, and the unlock height for any locked STX, all denominated in microstacks.

example:

Copy

`(stx-account 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR) ;; Returns (tuple (locked u0) (unlock-height u0) (unlocked u0))`

(stx-account (as-contract tx-sender)) ;; Returns (tuple (locked u0) (unlock-height u0) (unlocked u1000))

stx-burn?

Introduced in: **Clarity 1**

input: `uint`, `principal`

output: `(response bool uint)`

signature: `(stx-burn? amount sender)`

description:

`stx-burn?` decreases the `sender` principal's STX holdings by `amount`, specified in microstacks, by destroying the STX. The `sender` principal *must* be equal to the current context's `tx-sender`.

This function returns (ok true) if the transfer is successful. In the event of an unsuccessful transfer it returns one of the following error codes:

`(err u1)` -- `sender` does not have enough balance to transfer `(err u3)` -- amount to send is non-positive `(err u4)` -- the `sender` principal is not the current `tx-sender`

example:

Copy

(as-contract (stx-burn? u60 tx-sender)) ;; Returns (ok true)

(as-contract (stx-burn? u50 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)) ;; Returns (err u4)

stx-get-balance

Introduced in: **Clarity 1**

input: `principal`

output: `uint`

signature: `(stx-get-balance owner)`

description:

`stx-get-balance` is used to query the STX balance of the `owner` principal.

This function returns the STX balance, in microstacks (1 STX = 1,000,000 microstacks), of the `owner` principal. In the event that the `owner` principal isn't materialized, it returns 0.

example:

Copy

```
(stx-get-balance 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR) ;; Returns u0
```

```
(stx-get-balance (as-contract tx-sender)) ;; Returns u1000
```

stx-transfer-memo?

Introduced in: **Clarity 2**

input: `uint, principal, principal, buff`

output: `(response bool uint)`

signature: `(stx-transfer-memo? amount sender recipient memo)`

description:

`stx-transfer-memo?` is similar to `stx-transfer?`, except that it adds a `memo` field.

This function returns (ok true) if the transfer is successful, or, on an error, returns the same codes as `stx-transfer?`.

example:

Copy

```
(as-contract (stx-transfer-memo? u60 tx-sender  
'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR 0x010203)) ;; Returns (ok true)
```

stx-transfer?

Introduced in: **Clarity 1**

input: `uint, principal, principal, buff`

output: `(response bool uint)`

signature: `(stx-transfer? amount sender recipient)`

description:

`stx-transfer?` is used to increase the STX balance for the `recipient` principal by debiting the `sender` principal by `amount`, specified in microstacks. The `sender` principal *must* be equal to the current context's `tx-sender`.

This function returns (ok true) if the transfer is successful. In the event of an unsuccessful transfer it returns one of the following error codes:

(err u1) -- `sender` does not have enough balance to transfer (err u2) -- `sender` and `recipient` are the same principal (err u3) -- amount to send is non-positive (err u4) -- the `sender` principal is not the current `tx-sender`

example:

Copy

```
(as-contract (stx-transfer? u60 tx-sender  
'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)) ;; Returns (ok true)
```

```
(as-contract (stx-transfer? u60 tx-sender  
'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)) ;; Returns (ok true)
```

```
(as-contract (stx-transfer? u50 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR  
tx-sender)) ;; Returns (err u4)
```

to-consensus-buff?

Introduced in: **Clarity 2**

input: `any`

output: `(optional buff)`

signature: `(to-consensus-buff? value)`

description:

`to-consensus-buff?` is a special function that will serialize any Clarity value into a buffer, using the SIP-005 serialization of the Clarity value. Not all values can be serialized: some value's consensus serialization is too large to fit in a Clarity buffer (this is because of the type prefix in the consensus serialization).

If the value cannot fit as serialized into the maximum buffer size, this returns `none`, otherwise, it will be `(some consensus-serialized-buffer)`. During type checking, the analyzed type of the

result of this method will be the maximum possible consensus buffer length based on the inferred type of the supplied value.

example:

Copy

[illegible]

(to-consensus-buff? u1) ;; Returns (some 0x0100000000000000000000000000000001)

(to-consensus-buff? true) ;; Returns (some 0x03)

(to-consensus-buff? false) ;; Returns (some 0x04)

(to-consensus-buff? none) ;; Returns (some 0x09)

(to-consensus-buff? 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR) ;; Returns
(some 0x051fa46ff88886c2ef9762d970b4d2c63678835bd39d)

[illegible]

to-int

Introduced in: **Clarity 1**

input: uint

output: `int`

signature: (to-int u)

description:

Tries to convert the `uint` argument to an `int`. Will cause a runtime error and abort if the supplied argument is `>= pow(2, 127)`

example:

Copy

(to-int u238) ;; Returns 238

to-uint

Introduced in: **Clarity 1**

input: `int`

output: `uint`

signature: `(to-uint i)`

description:

Tries to convert the `int` argument to a `uint`. Will cause a runtime error and abort if the supplied argument is negative.

example:

Copy

`(to-uint 238) ;; Returns u238`

try!

Introduced in: **Clarity 1**

input: `(optional A) | (response A B)`

output: `A`

signature: `(try! option-input)`

description:

The `try!` function attempts to 'unpack' the first argument: if the argument is an option type, and the argument is a `(some ...)` option, `try!` returns the inner value of the option. If the argument is a response type, and the argument is an `(ok ...)` response, `try!` returns the inner value of the `ok`. If the supplied argument is either an `(err ...)` or a `none` value, `try!` returns either `none` or the `(err ...)` value from the current function and exits the current control-flow.

example:

Copy

`(define-map names-map { name: (string-ascii 12) } { id: int })`

`(map-set names-map { name: "blockstack" } { id: 1337 })`

`(try! (map-get? names-map { name: "blockstack" })) ;; Returns (tuple (id 1337))`

```
(define-private (checked-even (x int))
```

```
(if (is-eq (mod x 2) 0)
```

```
(ok x)
```

```
(err false)))
```

```
(define-private (double-if-even (x int))
```

```
(ok (* 2 (try! (checked-even x)))))
```

```
(double-if-even 10) ;; Returns (ok 20)
```

```
(double-if-even 3) ;; Returns (err false)
```

tuple

Introduced in: **Clarity 1**

input: (key-name A), (key-name-2 B), ...

output: (tuple (key-name A) (key-name-2 B) ...)

signature: (tuple (key0 expr0) (key1 expr1) ...)

description:

The `tuple` special form constructs a typed tuple from the supplied key and expression pairs. A `get` function can use typed tuples as input to select specific values from a given tuple. Key names may not appear multiple times in the same tuple definition. Supplied expressions are evaluated and associated with the expressions' paired key name.

There is a shorthand using curly brackets of the form {key0: expr0, key1: expr, ...}

example:

Copy

```
(tuple (name "blockstack")
```

```
(id 1337)) ;; using tuple
```

```
{name: "blockstack", id: 1337} ;; using curly brackets
```

unwrap!

Introduced in: **Clarity 1**

input: (optional A) | (response A B), C

output: A

signature: (unwrap! option-input thrown-value)

description:

The `unwrap!` function attempts to 'unpack' the first argument: if the argument is an option type, and the argument is a `(some ...)` option, `unwrap!` returns the inner value of the option. If the argument is a response type, and the argument is an `(ok ...)` response, `unwrap!` returns the inner value of the `ok`. If the supplied argument is either an `(err ...)` or a `(none)` value, `unwrap!` *returns thrown-value* from the current function and exits the current control-flow.

example:

Copy

```
(define-map names-map { name: (string-ascii 12) } { id: int })
```

```
(map-set names-map { name: "blockstack" } { id: 1337 })
```

```
(define-private (get-name-or-err (name (string-ascii 12)))
```

```
  (let ( (raw-name (unwrap! (map-get? names-map { name: name }) (err 1))))
```

```
    (ok raw-name)))
```

```
(get-name-or-err "blockstack") ;; Returns (ok (tuple (id 1337)))
```

```
(get-name-or-err "non-existent") ;; Returns (err 1)
```

unwrap-err!

Introduced in: **Clarity 1**

input: (response A B), C

output: B

signature: `(unwrap-err! response-input thrown-value)`

description:

The `unwrap-err!` function attempts to 'unpack' the first argument: if the argument is an `(err ...)` response, `unwrap-err!` returns the inner value of the `err`. If the supplied argument is an `(ok ...)` value, `unwrap-err!` *returns thrown-value* from the current function and exits the current control-flow.

example:

Copy

`(unwrap-err! (err 1) false) ;; Returns 1`

unwrap-err-panic

Introduced in: **Clarity 1**

input: `(response A B)`

output: `B`

signature: `(unwrap-err-panic response-input)`

description:

The `unwrap-err` function attempts to 'unpack' the first argument: if the argument is an `(err ...)` response, `unwrap` returns the inner value of the `err`. If the supplied argument is an `(ok ...)` value, `unwrap-err` throws a runtime error, aborting any further processing of the current transaction.

example:

Copy

`(unwrap-err-panic (err 1)) ;; Returns 1`

`(unwrap-err-panic (ok 1)) ;; Throws a runtime exception`

unwrap-panic

Introduced in: **Clarity 1**

input: `(optional A) | (response A B)`

output: `A`

signature: `(unwrap-panic option-input)`

description:

The `unwrap` function attempts to 'unpack' its argument: if the argument is an option type, and the argument is a `(some ...)` option, this function returns the inner value of the option. If the argument is a response type, and the argument is an `(ok ...)` response, it returns the inner value of the `ok`. If the supplied argument is either an `(err ...)` or a `(none)` value, `unwrap` throws a runtime error, aborting any further processing of the current transaction.

example:

Copy

```
(define-map names-map { name: (string-ascii 12) } { id: int })
```

```
(map-set names-map { name: "blockstack" } { id: 1337 })
```

```
(unwrap-panic (map-get? names-map { name: "blockstack" })) ;; Returns (tuple (id 1337))
```

```
(unwrap-panic (map-get? names-map { name: "non-existent" })) ;; Throws a runtime exception
```

use-trait

Introduced in: **Clarity 1**

input: `VarName, TraitIdentifier`

output: `Not Applicable`

signature: `(use-trait trait-alias trait-identifier)`

description:

`use-trait` is used to bring a trait, defined in another contract, to the current contract. Subsequent references to an imported trait are signaled with the syntax `<trait-alias>`.

Traits import are defined with a name, used as an alias, and a trait identifier. Trait identifiers can either be using the sugared syntax `(.token-a.token-trait)`, or be fully qualified `('SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF.token-a.token-trait)`.

Like other kinds of definition statements, `use-trait` may only be used at the top level of a smart contract definition (i.e., you cannot put such a statement in the middle of a function body).

example:

Copy

(use-trait token-a-trait

'SPAXYA5XS51713FDTQ8H94EJ4V579CXMTRNBZKSF.token-a.token-trait)

(define-public (forward-get-balance (user principal) (contract <token-a-trait>))

(begin (ok 1)))

var-get

Introduced in: **Clarity 1**

input: `VarName`

output: `A`

signature: `(var-get var-name)`

description:

The `var-get` function looks up and returns an entry from a contract's data map. The value is looked up using `var-name`.

example:

Copy

(define-data-var cursor int 6)

(var-get cursor) ;; Returns 6

var-set

Introduced in: **Clarity 1**

input: `VarName`, `AnyType`

output: `bool`

signature: `(var-set var-name expr1)`

description:

The `var-set` function sets the value associated with the input variable to the inputted value. The function always returns `true`.

example:

Copy

```
(define-data-var cursor int 6)
```

```
(var-get cursor) ;; Returns 6
```

```
(var-set cursor (+ (var-get cursor) 1)) ;; Returns true
```

```
(var-get cursor) ;; Returns 7
```

xor

Introduced in: **Clarity 1**

input: `int, int | uint, uint | string-ascii, string-ascii | string-utf8, string-utf8 | buff, buff`

output: `int | uint`

signature: `(xor i1 i2)`

description:

Returns the result of bitwise exclusive or'ing `i1` with `i2`.

example:

Copy

```
(xor 1 2) ;; Returns 3
```

```
(xor 120 280) ;; Returns 352
```

Clarity Keywords

block-height

The Nakamoto hard fork will introduce a few new Clarity keywords. It's important to note that even with the new [block production mechanism](#), the `block-height` keyword behavior will not change. It will simply correspond to the current tenure height. This means any Clarity contracts using this keyword will be backwards compatible after the Nakamoto Upgrade.

Introduced in: Clarity 1

output: `uint`

description:

Returns the current block height of the Stacks blockchain as an uint

example:

Copy

(> block-height 1000) ;; returns true if the current block-height has passed 1000 blocks.

burn-block-height

Introduced in: Clarity 1

output: `uint`

description:

Returns the current block height of the underlying burn blockchain as a uint

example:

Copy

(> burn-block-height 1000) ;; returns true if the current height of the underlying burn blockchain has passed 1000 blocks.

chain-id

Introduced in: Clarity 2

output: `uint`

description:

Returns the 32-bit chain ID of the blockchain running this transaction

example:

Copy

(print chain-id) ;; Will print 'u1' if the code is running on mainnet, and 'u2147483648' on testnet, and other values on different chains.

contract-caller

Introduced in: Clarity 1

output: `principal`

description:

Returns the caller of the current contract context. If this contract is the first one called by a signed transaction, the caller will be equal to the signing principal. If `contract-call?` was used to invoke a function from a new contract, `contract-caller` changes to the *calling* contract's principal. If `as-contract` is used to change the `tx-sender` context, `contract-caller` *also* changes to the same contract principal.

example:

Copy

(print contract-caller) ;; Will print out a Stacks address of the transaction sender

false

Introduced in: Clarity 1

output: `bool`

description:

Boolean false constant.

example:

Copy

(and true false) ;; Evaluates to false

(or false true) ;; Evaluates to true

is-in-mainnet

Introduced in: Clarity 2

output: `bool`

description:

Returns a boolean indicating whether or not the code is running on the mainnet

example:

Copy

(print is-in-mainnet) ;; Will print 'true' if the code is running on the mainnet

is-in-regtest

Introduced in: Clarity 1

output: `bool`

description:

Returns whether or not the code is running in a regression test

example:

Copy

(print is-in-regtest) ;; Will print 'true' if the code is running in a regression test

none

Introduced in: Clarity 1

output: `(optional ?)`

description:

Represents the *none* option indicating no value for a given optional (analogous to a null value).

example:

Copy

(define-public (only-if-positive (a int))

(if (> a 0)

(some a)

none))

(only-if-positive 4) ;; Returns (some 4)

(only-if-positive (- 3)) ;; Returns none

Copy

(print stx-liquid-supply) ;; Will print out the total number of liqui

stacks-block-height

Will be available after the Nakamoto hard fork

Introduced in: Clarity 3

output: `uint`

description:

Returns the current Stacks block height.

example:

Copy

(print stacks-block-height) ;; Will print out the current Stacks block height

stx-liquid-supply

Introduced in: Clarity 1

output: `uint`

description:

Returns the total number of micro-STX (uSTX) that are liquid in the system as of this block.

example:

Copy

(print stx-liquid-supply) ;; Will print out the total number of liquid uSTX

tenure-height

Will be available after Nakamoto hard fork

Introduced in: Clarity 3

output: `uint`

description:

Returns the number of tenures that have passed. When the Nakamoto block-processing starts, this will be equal to the chain length.

example:

Copy

(print tenure-height) ;; Will print out the current tenure height

true

Introduced in: Clarity 1

output: **bool**

description:

Boolean true constant.

example:

Copy

(and true false) ;; Evaluates to false

(or false true) ;; Evaluates to true

tx-sender

Introduced in: Clarity 1

output: **principal**

description:

Returns the original sender of the current transaction, or if **as-contract** was called to modify the sending context, it returns that contract principal.

example:

Copy

(print tx-sender) ;; Will print out a Stacks address of the transaction sender

tx-sponsor?

Introduced in: Clarity 2

output: **optional principal**

description:

Returns the sponsor of the current transaction if there is a sponsor, otherwise returns None.

example:

Copy

(print tx-sponsor?) ;; Will print out an optional value containing the Stacks address of

Stacks Node Configuration

Note that these config fields are for a Stacks follower node. If you are running a signer alongside your Stacks node, you'll want to use the sample file found on the [Signer Configuration](#) page as it contains additional parameters needed for your signer and Stacks node to function properly.

Usage

Copy

stacks-node sub-command [--subcommand-option <value>]

Subcommands

mocknet: start a mocknet instance using defaults

testnet: start a testnet instance using defaults (chainstate is not persistent)

mainnet: start a mainnet instance using defaults (chainstate is not persistent)

start: combined with **--config**, starts an instance with a specified configuration file

version: displays binary version

help: displays the help message

Configuration File Options

The Stacks Blockchain configuration file has multiple sections under which an option may be placed.

node

events_observer

connection_options

burnchain
ustx_balance

For reference, several configuration file examples are [available here](#)

Example mainnet follower configuration

node

Contains various configuration options for the stacks-node binary.

Name	Required	Description
rpc_bind	✓	IPv4 address and port to open for RPC connections
p2p_bind	✓	IPv4 address and port to open for P2P connections
working_dir		Absolute path to the directory where chainstate data will be stored
data_url		IPv4 address and port for incoming RPC connections
p2p_address		IPv4 address and port for incoming P2P connections
bootstrap_node		Public key, IPv4 address, and port to bootstrap the chainstate
wait_time_for_microblocks		The amount of time in ms to wait before trying to mine a block after catching up to the anchored chain tip
seed		The private key to use for mining. Only needed if <code>miner</code> is set to <code>true</code>

local_peer_seed	The private key to use for signing P2P messages in the networking stack
miner	Determines whether the node is running a follower (<code>false</code>) or a miner (<code>true</code>). Defaults to <code>false</code>
mock_miner	Simulates running a miner (typically used for debugging)
mine_microblocks	Determines whether the node will mine microblocks. Will only take effect if <code>miner</code> is set to <code>true</code>
prometheus_bind	Address and port for Prometheus metrics collection.

events_observer

This section is *optional* and not required

However, if this section is added, **all** fields are required.

Contains options for sending events emitted to the [stacks-blockchain-api](#) service.

Name	Required	Description
endpoint	✓	Address and port to a stacks-blockchain-api service
retry_count	✓	Number of times to retry sending events to the endpoint before failing
events_keys	✓	Event keys for which to watch. The emitted node events can be restricted by account, function name and event type. Asterix ("*") can be used to emit all events.

connection_options

This section is *optional* and not required.

Specifies configuration options for others connecting to the stacks node.

Name	Required	Description
public_ip_address		Public IPv4 to advertise to other nodes
download_interval		Time (in seconds) between attempts to download blocks
walk_interval		Time (in seconds) between attempts to walk the list of neighbors
private_neighbors		If false, this node won't announce or accept neighbors that are behind private networks. Defaults to true.
read_only_call_limit_read_length		Total number of bytes allowed to be read by an individual read-only function call
read_only_call_limit_read_count		Total number of independent read operations permitted for an individual read-only function call
read_only_call_limit_runtime		Runtime cost limit for an individual read-only function call

burnchain

This section contains configuration options pertaining to the blockchain the stacks-node binds to on the backend for proof-of-transfer (BTC).

Name	Required	Description
chain	✓	The blockchain stacks-node binds to on the backend for proof-of-transfer. Only value supported: bitcoin

mode	✓	The profile or test phase of which to run stacks-node. Valid values are [<code>mocknet</code> , <code>testnet</code> , <code>xenon</code> , <code>mainnet</code>]
peer_host		FQDN of the host running the backend Bitcoin blockchain
rpc_port		RPC port of <code>peer_host</code>
peer_port		P2P port of <code>peer_host</code>

Mining

Name	Required	Description
burn_fee_cap	✓	Maximum amount (in sats) of "burn commitment" to broadcast for the next block's leader election
satoshis_per_byte	✓	Amount (in sats) per byte - Used to calculate the transaction fees
commit_anchor_block_within		Sets the time period (in milliseconds) for commitments. Only used when <code>mode</code> is set to <code>mocknet</code> .

ustx_balance

This section is only required for the `testnet` and `mocknet` networks.

However, if this section is added, **all** fields are required.

This section contains configuration options allocating microSTX per address in the genesis block

This section can repeat multiple times, but each section can only define a single address.

Name	Required	Description
------	----------	-------------

address	✓	Address which maintains a microSTX balance
amount	✓	The balance of microSTX given to the address at the start of the node

Example Mainnet Follower Configuration

Copy

[node]

working_dir = "/stacks-blockchain"

rpc_bind = "0.0.0.0:30443"

p2p_bind = "0.0.0.0:20444"

bootstrap_node =

"02196f005965cebe6ddc3901b7b1cc1aa7a88f305bb8c5893456b8f9a605923893@seed.mainnet.hiro.so:20444,02539449ad94e6e6392d8c1deb2b4e61f80ae2a18964349bc14336d8b903c46a8c@cet.stacksnodes.org:20444,02ececc8ce79b8adf813f13a0255f8ae58d4357309ba0cedd523d9f1a306fcfb79@sgt.stacksnodes.org:20444,0303144ba518fe7a0fb56a8a7d488f950307a4330f146e1e1458fc63fb33defe96@est.stacksnodes.org:20444"

[burnchain]

chain = "bitcoin"

mode = "mainnet"

peer_host = "localhost"

username = "user"

password = "pass"

rpc_port = 8332

peer_port = 8333

```
[[events_observer]]
```

```
endpoint = "localhost:3700"
```

```
retry_count = 255
```

```
events_keys = ["*"]
```

Signer Configuration

The block for Nakamoto activation has been chosen as Bitcoin block 864,864, which is currently expected on October 9th. This block is subject to change should core developers need additional time for testing or unexpected issues.

Binaries will be provided roughly a week in advance and your normal upgrade procedure should apply here, you'll want to be running the latest node and Signer software. Note that if you do not upgrade ahead of the hard fork, your nodes will be dropped from the network. Ideally, you will have Stacked for Cycle #94 ahead of time, any time now is good.

Current Signer and Stacks Node Versions

If you are a signer, these are the current latest versions you'll want to be running.

Signer - [2.5.0.0.5.2](#)

Stacks Node - [2.5.0.0.7](#)

Signer Configuration File Options

The signer configuration file is a TOML file that contains the configuration options for your signer. Below are the options you can set in the signer configuration file.

Name	Required	Description
------	----------	-------------

node_host	✓	IP:PORT where your Stacks node can be accessed. The port 20443 is the default RPC endpoint for Stacks nodes. Note that you must use an IP address - DNS hosts are not supported at this time.
endpoint	✓	IP:PORT where the signer will expose an RPC endpoint for receiving events from your Stacks node.
stacks_private_key	✓	Hex representation of the signer's Stacks private key used for communicating with the Stacks Node, including writing to the Stacker DB instance.
network	✓	Network to use. One of "mainnet", "testnet" or "mocknet".
auth_password	✓	Authorization token for HTTP requests made from the signer to your Stacks node.
db_path	✓	Path to the signer's database file or :memory: for an in-memory database.
metrics_endpoint		IP:PORT for Prometheus metrics collection.
event_timeout_ms		Time to wait (in milliseconds) for a response from the stacker-db instance.
block_proposal_timeout_ms		Time to wait (in milliseconds) for a block response from miners.

dkg_public_timeout_ms	Timeout in (milliseconds) to gather DkgPublicShares messages.
dkg_private_timeout_ms	Timeout in (milliseconds) to gather DkgPrivateShares messages.
dkg_end_timeout_ms	Timeout in (milliseconds) to gather DkgEnd messages.
nonce_timeout_ms	Timeout in (milliseconds) to gather nonces.
sign_timeout_ms	Timeout in (milliseconds) to gather signature shares.
tx_fee_ustx	The STX tx fee to use in microSTX. If not set, will default to 10000.
max_tx_fee_ustx	The max STX tx fee to use in microSTX when estimating fees. If not set, will use tx_fee_ustx.

Example Configs

Below are sample configuration files for running a Stacks node and signer provided in one place for convenience. You'll need to modify some of these according to the [How to Run a Signer](#) doc.

Testnet Signer

Copy

The IP address and port where your Stacks node can be accessed.

The port 20443 is the default RPC endpoint for Stacks nodes.

Note that you must use an IP address - DNS hosts are not supported at this time.

This should be the IP address accessible via Docker, usually via a network.

```
node_host = "127.0.0.1:20443"
```

```
# This is the location where the signer will expose an RPC endpoint for  
# receiving events from your Stacks node.
```

```
endpoint = "127.0.0.1:30000"
```

```
# Either "testnet" or "mainnet"
```

```
network = "testnet"
```

```
# this is a file path where your signer will persist data. If using Docker,  
# this must be within a volume, so that data can be persisted across restarts
```

```
db_path = "/var/stacks/signer.sqlite"
```

```
# an authentication token that is used for some HTTP requests made from the  
# signer to your Stacks node. You'll need to use this later on when configuring  
# your Stacks node. You create this field yourself, rather than it being generated  
# with your private key.
```

```
auth_password = "$your_http_auth_token"
```

```
# This is the privateKey field from the keys you generated in the  
# previous step.
```

```
stacks_private_key = "$your_stacks_private_key"
```

Nakamoto Testnet Config

This is the configuration you'll need to run a Stacks follower node if you are also running a signer. Be sure to change the commented lines to the appropriate data for your setup. If you are not familiar with the process of setting up a signer, be sure to follow the [How to Run a Signer](#) guide.

An overview of all Stacks node configuration options can be found in the [Stacks Node Configuration](#) doc.

Additions necessary specifically to run a signer are the `[connection_options]` and `[events_observer]` sections and the `stacker = true` line. There are also a few comments detailing other lines that need to change.

Copy

[node]

Set this based on where you downloaded

the chain state archive as described in the How to Run a Signer guide:

`working_dir = "/stacks-blockchain/data"`

`rpc_bind = "0.0.0.0:20443"`

`p2p_bind = "0.0.0.0:20444"`

This is the node that your node will use to begin syncing chain state

`bootstrap_node =`

`"0341b2ff35b545d8e5c5d2fc8821484610ef85ce8e276214caf23d53be16fdcd65@seed.nakamoto.testnet.hiro.so:50444"`

`wait_time_for_microblocks = 0`

`mine_microblocks = false`

Required for nodes attached to signers, optional for other nodes

`stacker = true`

[burnchain]


```
chain = "bitcoin"

mode = "krypton"

magic_bytes = "N3"

poll_time_secs = 30

pox_prepare_length = 100

pox_reward_length = 900

peer_host = "bitcoin.nakamoto.regtest.hiro.so"

username = "hirosystems"

password = "hirosystems"

burnchain_op_tx_fee = 5500

commit_anchor_block_within = 300000

rpc_port = 18543

peer_port = 18544

#satoshis_per_byte = 20

first_burn_block_height = 230

first_burn_block_timestamp = 1714513150

first_burn_block_hash =
"654e1e9f66701d4f8a138b46d4cf0cc26665688175bcbb1700729efbf759e57d"


# Set your auth token, which the signer uses

# This should match the auth_password field of your signer config

[connection_options]

auth_token = "12345"
```

```
# Set your signer as an event observer
```

```
[[events_observer]]
```

```
# This endpoint is where your signer will communicate with your Stacks node
```

```
endpoint = "127.0.0.1:30000"
```

```
retry_count = 255
```

```
include_data_events = false
```

```
events_keys = ["stackerdb", "block_proposal", "burn_blocks"]
```

```
[[burnchain.epochs]]
```

```
epoch_name = "1.0"
```

```
start_height = 0
```

```
[[burnchain.epochs]]
```

```
epoch_name = "2.0"
```

```
start_height = 230
```

```
[[burnchain.epochs]]
```

```
epoch_name = "2.05"
```

```
start_height = 240
```

```
[[burnchain.epochs]]
```

epoch_name = "2.1"

start_height = 240

[[burnchain.epochs]]

epoch_name = "2.2"

start_height = 241

[[burnchain.epochs]]

epoch_name = "2.3"

start_height = 242

[[burnchain.epochs]]

epoch_name = "2.4"

start_height = 243

[[burnchain.epochs]]

epoch_name = "2.5"

start_height = 244

[[burnchain.epochs]]

epoch_name = "3.0"

start_height = 2_000_250

[[ustx_balance]]

address = "ST0DZFQ1XGHC5P1BZ6B7HSWQKQJHM74JBGCSDTNA"

amount = 10000000000000000

[[ustx_balance]]

address = "ST2G2RJR4B5M95D0ZZAGZJP9J4WH090WHP0C5YW0H"

amount = 10000000000000000

[[ustx_balance]]

address = "ST3JCQJE9NZRCAPPE44Q12KR7FH8AY9HTEMWP2G5F"

amount = 10000000000000000

[[ustx_balance]]

address = "STA0EP5GD8FC661T8Q0Z382QW7Z6JXDM3E476MB7"

amount = 17500000000000

[[ustx_balance]]

address = "ST3MNK12DGQF7JN4Q0STK6926VWE5MN21KJ4EGV0E"

amount = 10000000000000000

[[ustx_balance]]

address = "ST484MS3VACPAZ90WHC21XQ7T6XANCV341HJYE0W"

amount = 10000000000000000

[[ustx_balance]]

address = "ST2D1M978SCE52GAV07VXSRC9DQBP69X5WHX0DHN5"

amount = 10000000000000000

[[ustx_balance]]

address = "ST2A68NMMXVZDWDTDZ5GJGA69M86V8KK0JS9X1QQP"

amount = 10000000000000000

[[ustx_balance]]

address = "ST2ME1CR5XR0P332SBTSD90P9HG48F1SK8MZVJ3XW"

amount = 10000000000000000

[[ustx_balance]]

address = "ST19MXV72S9HHRSZCDY10K9DMB11JYPTXVVNYAWPH"

amount = 10000000000000000

[[ustx_balance]]

address = "ST20Q2N56E1NBWE37R4VGSF89X4HHTB3GSMD8GKYW"

amount = 10000000000000000

[[ustx_balance]]

address = "ST2Q6124HQFKVKPJSS5J6156BJR74FD6EC1297HJ1"

amount = 10000000000000000

[[ustx_balance]]

address = "ST1114TBQYGNPGFAVXKWBKZAHP0X7ZGX9K6XYYE4F"

amount = 10000000000000000

[[ustx_balance]]

address = "ST1NCEQ0T4Z32QTYT88BNXJKC9HR3VWYHJ0TB95TP"

amount = 10000000000000000

[[ustx_balance]]

address = "STWF12K119FTA70NDG29MNYWR0CPMF44ZKC2SG2T"

amount = 24378281250000

[[ustx_balance]]

address = "ST36G5CRHH1GJVZGFWPTW4H9GSA8VAVWM0ST7AV82"

amount = 24378281250000

[[ustx_balance]]

address = "ST2KWFMX0SVXFMZ0W7TXZ3MV0C6V276BNAT49XAQW"

amount = 24378281250000

[[ustx_balance]]

address = "ST1ZMVDYKGWF5TFGH46GEFBR273JJ3RRTHEDETKNH"

amount = 24378281250000

[[ustx_balance]]

address = "ST3D0TEK871ZMBFFF0998YY609A1QGM6ZTYCQJJFQ"

amount = 24378281250000

[[ustx_balance]]

address = "ST372ND8K8M3GKESD0KG8ZWJ6EV0GGXWXC5246MJN"

amount = 24378281250000

[[ustx_balance]]

address = "ST33PA4H3TW3DQFHG2RXPGGW1FFG5YQJ704B3DA8M"

amount = 24378281250000

[[ustx_balance]]

address = "STJ737JNPK525J86BGSPAW362SRRAYC4SP6F95HC"

amount = 24378281250000

[[ustx_balance]]

address = "ST21AJANGK9NA2ZED5D5J1VZPTVW8DY05B0ECMFN"

amount = 24378281250000

[[ustx_balance]]

address = "ST30Z74A4S2T8563D844ENSBHBFSVQEVBPV9S0A7E"

amount = 24378281250000

[[ustx_balance]]

address = "ST2FGTGYAGJVXJZQX17NBJNSQAM4J2V5JFDHEEAZQ"

amount = 24378281250000

[[ustx_balance]]

address = "ST16PC3G9BMQH0G37JGAGDGYZPDB5NGNARBDFPWYB"

amount = 24378281250000

[[ustx_balance]]

address = "ST1XJHGBSQPV9B14HFYG98ZBSQGKG8GN0AMB3V2VT"

amount = 24378281250000

[[ustx_balance]]

address = "ST2XDC0R30841X2RRECWW2F9KTANKQEERPS4V3H9D"

amount = 24378281250000

[[ustx_balance]]

address = "ST2HC6JENRNNE6YVATT7WZVZWVR5J26BGYX67W8G7"

amount = 24378281250000

[[ustx_balance]]

address = "STPW2CGZC98EZ95XYC9DE93SFBS5KA2PYYK89VHM"

amount = 24378281250000

[[ustx_balance]]

address = "STNX3E9MYTA2ZDQK53YNMMJ3E7783DC019JZNYZZ"

amount = 24378281250000

[[ustx_balance]]

address = "ST0D135PF2R0S4B6S4G49QZC69KF19MSZ4Z5RDF5"

amount = 24378281250000

Primary Testnet Config

Copy

[node]

rpc_bind = "0.0.0.0:20443"

p2p_bind = "0.0.0.0:20444"

```
bootstrap_node =  
"029266faff4c8e0ca4f934f34996a96af481df94a89b0c9bd515f3536a95682ddc@seed.testnet.hiro.so:30444"
```

```
prometheus_bind = "127.0.0.1:9153"
```

```
working_dir = "/hirosystems/data"
```

```
local_peer_seed = "{{ redacted }}"
```

```
# Required for nodes attached to signers, optional for other nodes
```

```
stacker = true
```

```
[burnchain]
```

```
chain = "bitcoin"
```

```
mode = "krypton"
```

```
peer_host = "bitcoin.regtest.hiro.so"
```

```
username = "hirosystems"
```

```
password = "hirosystems"
```

```
rpc_port = 18443
```

```
peer_port = 18444
```

```
pox_prepare_length = 100
```

```
pox_reward_length = 900
```

```
# Set your auth token, which the signer uses
```

```
# This should match the auth_password field of your signer config
```

```
[connection_options]
```

```
auth_token = "12345"
```

```
# Set your signer as an event observer
```

```
[[events_observer]]
```

```
# This endpoint is where your signer will communicate with your Stacks node
```

```
endpoint = "127.0.0.1:30000"
```

```
retry_count = 255
```

```
include_data_events = false
```

```
events_keys = ["stackerdb", "block_proposal", "burn_blocks"]
```

```
[[ustx_balance]]
```

```
address = "ST2QKZ4FKHAH1NQKYKYAYZPY440FEPK7GZ1R5HBP2"
```

```
amount = 10000000000000000
```

```
[[ustx_balance]]
```

```
address = "ST319CF5WV77KYR1H3GT0GZ7B8Q4AQPY42ETP1VPF"
```

```
amount = 10000000000000000
```

```
[[ustx_balance]]
```

```
address = "ST221Z6TDTTC5E0BYR2V624Q2ST6R0Q71T78WTAX6H"
```

```
amount = 10000000000000000
```

[[ustx_balance]]

address = "ST2TFVBMRPS5SSNP98DQKQ5JNB2B6NZM91C4K3P7B"

amount = 10000000000000000

[fee_estimation]

fee_estimator = "fuzzed_weighted_median_fee_rate"

[[burnchain.epochs]]

epoch_name = "1.0"

start_height = 0

[[burnchain.epochs]]

epoch_name = "2.0"

start_height = 0

[[burnchain.epochs]]

epoch_name = "2.05"

start_height = 1

[[burnchain.epochs]]

epoch_name = "2.1"

start_height = 2

```
[[burnchain.epochs]]
```

```
epoch_name = "2.2"
```

```
start_height = 3
```

```
[[burnchain.epochs]]
```

```
epoch_name = "2.3"
```

```
start_height = 4
```

```
[[burnchain.epochs]]
```

```
epoch_name = "2.4"
```

```
start_height = 5
```

```
[[burnchain.epochs]]
```

```
epoch_name = "2.5"
```

```
start_height = 6
```

```
[[burnchain.epochs]]
```

```
epoch_name = "3.0"
```

```
start_height = 2000701
```

Mainnet Signer

This config is very similar to the testnet config, except the `network` field is changed.

Copy

The IP address and port where your Stacks node can be accessed.

The port 20443 is the default RPC endpoint for Stacks nodes.

Note that you must use an IP address - DNS hosts are not supported at this time.

This should be the IP address accessible via Docker, usually via a network.

node_host = "127.0.0.1:20443"

This is the location where the signer will expose an RPC endpoint for

receiving events from your Stacks node.

endpoint = "127.0.0.1:30000"

Either "testnet" or "mainnet"

network = "mainnet"

this is a file path where your signer will persist data. If using Docker,

this must be within a volume, so that data can be persisted across restarts

db_path = "/var/stacks/signer.sqlite"

an authentication token that is used for some HTTP requests made from the

signer to your Stacks node. You'll need to use this later on when configuring

your Stacks node. You create this field yourself, rather than it being generated

with your private key.

auth_password = "\$your_http_auth_token"

This is the privateKey field from the keys you generated in the

previous step.

stacks_private_key = "\$your_stacks_private_key"

The IP address and port where prometheus metrics can be accessed.

metrics_endpoint = "127.0.0.1:9154"

Mainnet Stacks Node

With a mainnet Stacks node config, you'll need to change the bootstrap node field and the burnchain fields. Other than that, the `ustx_balance` fields are not necessary.

Copy

[node]

Set this based on where you downloaded

the chain state archive as described in the How to Run a Signer guide:

working_dir = "/data-dir-somewhere"

rpc_bind = "0.0.0.0:20443"

p2p_bind = "0.0.0.0:20444"

This is the node that your node will use to begin syncing chain state

bootstrap_node =

"02196f005965cebe6ddc3901b7b1cc1aa7a88f305bb8c5893456b8f9a605923893@seed.mainnet.hiro.so:20444,02539449ad94e6e6392d8c1deb2b4e61f80ae2a18964349bc14336d8b903c46a8c@cet.stacksnodes.org:20444,02eccc8ce79b8adf813f13a0255f8ae58d4357309ba0cedd523d9f1a306fcfb79@sgt.stacksnodes.org:20444,0303144ba518fe7a0fb56a8a7d488f950307a4330f146e1e1458fc63fb33defe96@est.stacksnodes.org:20444"

Required for nodes attached to signers, optional for other nodes

stacker = true

```
[burnchain]
```

```
chain = "bitcoin"
```

```
mode = "mainnet"
```

```
peer_host = "bitcoind.stacks.co"
```

```
username = "blockstack"
```

```
password = "blockstacksystem"
```

```
rpc_port = 8332
```

```
peer_port = 8333
```

```
# Set your auth token, which the signer uses
```

```
# This should match the auth_password field of your signer config
```

```
[connection_options]
```

```
auth_token = "12345"
```

```
# Set your signer as an event observer
```

```
[[events_observer]]
```

```
# This endpoint is where your signer will communicate with your Stacks node
```

```
endpoint = "127.0.0.1:30000"
```

```
retry_count = 255
```

```
include_data_events = false
```

```
events_keys = ["stackerdb", "block_proposal", "burn_blocks"]
```


Stacks Tooling

New to developing on Stacks or looking for a quick reference guide for all the important components and links? You're in the right place.

We'll go over all the building blocks you need to be aware of to build high-quality Stacks dapps. This page exists to serve as a reference to the Stacks developer's tool chest.

Building Blocks

Clarity

[Clarity](#) is the smart contract language on Stacks. If you want to build the next decentralized social network, DeFi protocol, or any other Stacks dapp, you'll need to know Clarity.

Post Conditions

[Post conditions](#) are a cool feature of the Stacks blockchain that allow you to verify the legitimacy of a transaction on the client side before it is executed. This adds an additional layer of defense against malicious smart contracts.

Proof of Transfer

[PoX](#) is the unique consensus mechanism of Stacks that facilitates new block production and also allows Stackers to earn real Bitcoin yield by participating in locking their STX tokens.

Stacking

Speaking of [Stacking](#), it's the mechanism that helps to secure the Stacks chain and allows Stackers to earn real Bitcoin yield transferred by miners.

SIP-009 and SIP-010 Tokens

Fungible and non-fungible tokens in Clarity are defined by [SIP-009](#) and [SIP-010](#) standards. You can learn more about how to work with these tokens in the Clarity book.

Tools

Wallets

Wallets are a key tool in any web3 ecosystem, and Stacks is no different. There are several options available including:

[Leather](#)

[Xverse](#)

[Asigna](#)

Platform

The [Hiro Platform](#) is your all-in-one cloud development environment for Stacks development, and is integrated with most of the tools listed below.

It's by far the easiest way to get up and running quickly. Plus, they have SSH integration

Explorer

Every developer needs a block explorer to see take a look at information about blocks and transactions being submitted to the chain. You have two choices here: the [Hiro Explorer](#) and [STXScan](#).

API

If you want to interact with or read data from the chain, there's a good chance the [Hiro API](#) has an endpoint for that.

Stacks.js

[Stacks.js](#) is the de-facto JavaScript library for the Stacks ecosystem. There are several packages here that will help you build robust frontends for your applications.

Clarinet

All good developer tooling needs a robust, easy-to-use development environment. Enter [Clarinet](#). Clarinet provides everything you need to write, test, and deploy Clarity smart contracts, including a fully-featured local devnet blockchain.

Chainhook

One of the key use cases for Stacks is being able to directly interact with the Bitcoin chain. Hiro's [Chainhook](#) makes this easier by providing an IFTTT system for responding and reacting to events on both the Bitcoin and Stacks chains.

Stacking Tools

The DegenLabs team has created a [suite of tools](#) to make stacking significantly easier including a signer signature generator, a solo stacking dapp to stack without needing to run a signer, and a TypeScript library for mocking stacking functions.

Educational Resources

Docs

These docs you are currently looking at are a great place to get a comprehensive view of all things in the Stacks ecosystem, as well as providing some links out to additional resources you'll find helpful.

Hiro Docs

Hiro is a key player in the Stacks ecosystem, providing several developer tools to make your life easier. They also publish excellent [guides and docs](#) to make using these tools a breeze.

Clarity Book

The [Clarity Book](#) is the go-to resource for learning how to be a Clarity developer. In it you'll not only get the basics of Clarity but go through several practice projects and learn best practices.

LearnWeb3

LearnWeb3 is one of the best education providers in the game. They have recently begun publishing courses as part of their [Stacks Developer Degree](#). LearnWeb3 courses will teach you everything you need to know about building Stacks Dapps.

EasyA

[EasyA](#) is a mobile app with a Stacks course built in. The EasyA app allows you to learn on the go and is a great way to learn the basics of Stacks and Clarity development all directly in their app.

Bitcoin Primer

If you're new to Bitcoin, interested in how it works, and how you can build Stacks dapps that interact with it, the [Bitcoin Primer](#) is a great place to start.