Types

Clarity Type System

The type system contains the following types:

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

Functions

* (multiply)

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:
(* 2 3) ;; Returns 6

(* 5 2) ;; Returns 10

(* 2 2 2) ;; Returns 8

+ (add)

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:
(+ 1 2 3) ;; Returns 6

- (subtract)

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:
(- 2 1 1) ;; Returns 0

(- 0 3) ;; Returns -3

/ (divide)

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:
(/ 2 3) ;; Returns 0

(/ 5 2) ;; Returns 2

(/ 4 2 2) ;; Returns 1

< (less than)

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:

(< 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)

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:

(<= 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)

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:
(> 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)

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:
(>= 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

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:
(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

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

as-contract

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

as-max-len?

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:
(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!

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

at-block

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:
(define-data-var data int 1)

(at-block 0x0000000000000000000000000000000000000000000000000000000000000000 block-height) ;; Returns u0

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

begin

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

bit-and

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:
(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

input: `int | uint`

output: `int | uint`

signature: `(bit-not i1)`

description:

Returns the one's compliement (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 ì1`will be`0`in the result. Conversely, every bit that is`0`in`i1`will be`1` in the result.

example:
(bit-not 3) ;; Returns -4

(bit-not u128) ;; Returns u340282366920938463463374607431768211327

(bit-not 128) ;; Returns -129

(bit-not -128) ;; Returns 127

bit-or

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:
(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

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:
(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

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:

(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

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:
(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

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

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

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

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

buff-to-int-le

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

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

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

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

buff-to-uint-be

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

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

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

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

buff-to-uint-le

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

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

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

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

concat

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:
(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?

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

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

contract-of

input: `Trait`

output: `principal`

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

description:

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

example:
(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

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:
(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-existant"))))) ;; Returns 0

define-constant

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

(+ 4 four) ;; Returns 8

define-data-var

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:
(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

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:

(define-fungible-token stacks)

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

define-map

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:

(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

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:

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

define-private

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:

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

 (if (> i1 i2)

   i1

   i2))

(max-of 4 6) ;; Returns 6
```

define-public

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

 (begin

   (print (+ 2 input))

   (ok input)))

define-read-only

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

 (* 10 10))

define-trait

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:
(define-trait token-trait

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

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

element-at

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:
(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?

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:
(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

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:
(err true) ;; Returns (err true)

filter

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:
(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

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:
(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?

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:
(from-consensus-buff? int 0x0000000000000000000000000000000001) ;; Returns (some 1)

(from-consensus-buff? uint 0x0000000000000000000000000000000001) ;; 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)

(from-consensus-buff? { abc: int, def: int }
0x0c0000000203616263000000000000000000000000000000000303646566000000000000000000000000000000000
0000000000000000000004) ;; Returns (some (tuple (abc 3) (def 4)))

ft-burn?

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* transfered 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:
(define-fungible-token stackaroo)

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

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

ft-get-balance

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:
(define-fungible-token stackaroo)

(ft-mint? stackaroo u100 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)

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

ft-get-supply

input: `TokenName`

output: `uint`

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

description:

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

example:
(define-fungible-token stackaroo)

(ft-mint? stackaroo u100 'SZ2J6ZY48GV1EZ5V2V5RB9MP66SW86PYKKQ9H6DPR)

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

ft-mint?

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* transfered from another principal, but rather minted.

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

example:

(define-fungible-token stackaroo)

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

ft-transfer?

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:
(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

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:
(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?

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:
(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?

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

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

hash160

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:
(hash160 0) ;; Returns 0xe4352f72356db555721651aa612e00379167b30f

if

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:
(if true 1 2) ;; Returns 1

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

impl-trait

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:
(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

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:
(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?

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:
(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

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

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

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

int-to-utf8

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

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

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

is-eq

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:
(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

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

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

is-none

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:
(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-existant" }))) ;; Returns true

is-ok

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

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

is-some

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:
(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-existant" }))) ;; Returns false

is-standard

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:
(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

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:

(keccak256 0) ;; Returns
0xf490de2920c8a35fabeb13208852aa28c76f9be9b03a4dd2b3c075f7a26923b4

len

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:
(len "blockstack") ;; Returns u10

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

(len 0x010203) ;; Returns u3

let

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:

(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

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

log2

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:
(log2 u8) ;; Returns u3

(log2 8) ;; Returns 3

(log2 u1) ;; Returns u0

(log2 1000) ;; Returns 9

map

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:
(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

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:
(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?

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:
(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

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:
(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

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:
(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

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:

```
(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

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:
(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

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:
(mod 2 3) ;; Returns 2

(mod 5 2) ;; Returns 1

(mod 7 1) ;; Returns 0

nft-burn?

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:
(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?

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:
(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?

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 successfuly mint, it returns `(ok true)`.

example:
(define-non-fungible-token stackaroo (string-ascii 40))

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

nft-transfer?

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:
(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

input: `bool`

output: `bool`

signature: `(not b1)`

description:

Returns the inverse of the boolean input.

example:
(not true) ;; Returns false

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

ok

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:
(ok 1) ;; Returns (ok 1)

or

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:
(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

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:
(pow 2 3) ;; Returns 8

(pow 2 2) ;; Returns 4

(pow 7 1) ;; Returns 7

principal-construct?

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:
(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?

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:
(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?

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:
(principal-of? 0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns (ok ST1AW6EKPGT61SQ9FNVDS17RKNWT8ZP582VF9HSCP)

print

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

replace-at?

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:
(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?

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:
(secp256k1-recover?
0xde5b9eb9e7c5592930eb2e30a01369c36586d872082ed8181ee83d2a0ec20f04
0x8738487ebe69b93d8e51583be8eee50bb4213fc49c767d329632730cc193b873554428fc936c
a3569afc15f1c9365f6591d6251a89fee9c9ac661116824d3a1301) ;; Returns (ok
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110)

secp256k1-verify

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:
(secp256k1-verify
0xde5b9eb9e7c5592930eb2e30a01369c36586d872082ed8181ee83d2a0ec20f04
0x8738487ebe69b93d8e51583be8eee50bb4213fc49c767d329632730cc193b873554428fc936c
a3569afc15f1c9365f6591d6251a89fee9c9ac661116824d3a1301
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns
true

(secp256k1-verify
0xde5b9eb9e7c5592930eb2e30a01369c36586d872082ed8181ee83d2a0ec20f04
0x8738487ebe69b93d8e51583be8eee50bb4213fc49c767d329632730cc193b873554428fc936c
a3569afc15f1c9365f6591d6251a89fee9c9ac661116824d3a13
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns
true

(secp256k1-verify
0x0000000000000000000000000000000000000000000000000000000000000000
0x00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
0x03adb8de4bfb65db2cfd6120d55c6526ae9c52e675db7e47308636534ba7786110) ;; Returns
false

sha256

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:
(sha256 0) ;; Returns
0x374708fff7719dd5979ec875d56cd2286f6d3cf7ec317a3b25632aab28ec37bb

sha512

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:
(sha512 1) ;; Returns
0x6fcee9a7b7a7b821d241c03c82377928bc6882e7a08c78a4221199bfa220cdc55212273018ee
613317c8293bb8d1ce08d1e017508e94e06ab85a734c99c7cc34

sha512/256

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:
(sha512/256 1) ;; Returns
0x515a7e92e7c60522db968d81ff70b80818fc17aeabbec36baf0dda2812e94a86

slice?

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:
(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

input: `A`

output: `(optional A)`

signature: `(some value)`

description:

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

example:
(some 1) ;; Returns (some 1)

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

sqrti

input: `int | uint`

output: `int | uint`

signature: `(sqrti n)`

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

Fails on a negative numbers.

example:
(sqrti u11) ;; Returns u3

(sqrti 1000000) ;; Returns 1000

(sqrti u1) ;; Returns u1

(sqrti 0) ;; Returns 0

string-to-int?

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

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

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

string-to-uint?

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

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

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

stx-account

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:
(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?

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

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

stx-get-balance

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

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

stx-transfer-memo?

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

stx-transfer?

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:
(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?

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:
(to-consensus-buff? 1) ;; Returns (some 0x0000000000000000000000000000000001)

(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)

(to-consensus-buff? { abc: 3, def: 4 }) ;; Returns (some
0x0c00000002036162630000000000000000000000000000000003036465660000000000000000
000000000000000004)

to-int

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:
(to-int u238) ;; Returns 238

to-uint

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:
(to-uint 238) ;; Returns u238

try!

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:
(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

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:
(tuple (name "blockstack")

(id 1337)) ;; using tuple

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

unwrap!

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:
(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-existant") ;; Returns (err 1)

unwrap-err!

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

unwrap-err-panic

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

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

unwrap-panic

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:
(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-existant" })) ;; Throws a runtime exception

use-trait

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:
(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

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:
(define-data-var cursor int 6)

(var-get cursor) ;; Returns 6

var-set

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:
(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

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:
(xor 1 2) ;; Returns 3

(xor 120 280) ;; Returns 352

Keywords

block-height

output: **`uint`**

description:

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

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

burn-block-height

output: `uint`

description:

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

example:
(> burn-block-height 1000) ;; returns true if the current height of the underlying burn blockchain has passed 1000 blocks.

chain-id

output: `uint`

description:

Returns the 32-bit chain ID of the blockchain running this transaction

example:
(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

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:
(print contract-caller) ;; Will print out a Stacks address of the transaction sender

false

output: `bool`

description:

Boolean false constant.

example:
(and true false) ;; Evaluates to false

(or false true) ;; Evaluates to true

is-in-mainnet

output: `bool`

description:

Returns a boolean indicating whether or not the code is running on the mainnet

example:
(print is-in-mainnet) ;; Will print 'true' if the code is running on the mainnet

is-in-regtest

output: `bool`

description:

Returns whether or not the code is running in a regression test

example:
(print is-in-regtest) ;; Will print 'true' if the code is running in a regression test

none

output: `(optional ?)`

description:

Represents the *none* option indicating no value for a given optional (analogous to a null value).

example:
```
(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

stx-liquid-supply

output: `uint`

description:

Returns the total number of micro-STX (uSTX) that are liquid in the system as of this block.

example:

(print stx-liquid-supply) ;; Will print out the total number of liquid uSTX

true

output: `bool`

description:

Boolean true constant.

example:
(and true false) ;; Evaluates to false

(or false true) ;; Evaluates to true

tx-sender

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:
(print tx-sender) ;; Will print out a Stacks address of the transaction sender

tx-sponsor?

output: `optional principal`

description:

Returns the sponsor of the current transaction if there is a sponsor, otherwise returns None.

example:
(print tx-sponsor?) ;; Will print out an optional value containing the Stacks address of the transaction sponsor

Smart contract Example code:

- counter.clar:

```
;; Multiplayer Counter contract

(define-map counters principal uint)

(define-read-only (get-count (who principal))
	(default-to u0 (map-get? counters who))
)

(define-public (count-up)
	(ok (map-set counters tx-sender (+ (get-count tx-sender) u1)))
)
```

- multisig-vault.clar:

```
;; multisig-vault
;; A simple multisig vault that allows members to vote on who should receive the STX contents.

(define-constant contract-owner tx-sender)
(define-constant err-owner-only (err u100))
(define-constant err-already-locked (err u101))
(define-constant err-more-votes-than-members-required (err u102))
(define-constant err-not-a-member (err u103))
(define-constant err-votes-required-not-met (err u104))

(define-data-var members (list 100 principal) (list))
(define-data-var votes-required uint u1)
(define-map votes {member: principal, recipient: principal} {decision: bool})

(define-public (start (new-members (list 100 principal)) (new-votes-required uint))
	(begin
		(asserts! (is-eq tx-sender contract-owner) err-owner-only)
		(asserts! (is-eq (len (var-get members)) u0) err-already-locked)
		(asserts! (>= (len new-members) new-votes-required)
err-more-votes-than-members-required)
		(var-set members new-members)
		(var-set votes-required new-votes-required)
		(ok true)
	)
)

(define-public (vote (recipient principal) (decision bool))
	(begin
		(asserts! (is-some (index-of (var-get members) tx-sender)) err-not-a-member)
		(ok (map-set votes {member: tx-sender, recipient: recipient} {decision: decision}))
	)
)

(define-read-only (get-vote (member principal) (recipient principal))
	(default-to false (get decision (map-get? votes {member: member, recipient: recipient})))
)

(define-private (tally (member principal) (accumulator uint))
	(if (get-vote member tx-sender) (+ accumulator u1) accumulator)
)

(define-read-only (tally-votes)
	(fold tally (var-get members) u0)
)

(define-public (withdraw)
	(let
		(
			(recipient tx-sender)
			(total-votes (tally-votes))
		)
		(asserts! (>= total-votes (var-get votes-required)) err-votes-required-not-met)
		(try! (as-contract (stx-transfer? (stx-get-balance tx-sender) tx-sender recipient)))
		(ok total-votes)
	)
)

(define-public (deposit (amount uint))
	(stx-transfer? amount tx-sender (as-contract tx-sender))
)
```

- stacksies.clar:

```
(impl-trait 'SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait.nft-trait)

(define-constant contract-owner tx-sender)
(define-constant err-owner-only (err u100))
(define-constant err-not-token-owner (err u101))

(define-non-fungible-token stacksies uint)

(define-data-var last-token-id uint u0)

(define-read-only (get-last-token-id)
	(ok (var-get last-token-id))
)

(define-read-only (get-token-uri (token-id uint))
	(ok none)
)

(define-read-only (get-owner (token-id uint))
	(ok (nft-get-owner? stacksies token-id))
)

(define-public (transfer (token-id uint) (sender principal) (recipient principal))
	(begin
		(asserts! (is-eq tx-sender sender) err-not-token-owner)
		(nft-transfer? stacksies token-id sender recipient)
	)
)

(define-public (mint (recipient principal))
	(let
		(
			(token-id (+ (var-get last-token-id) u1))
		)
		(asserts! (is-eq tx-sender contract-owner) err-owner-only)
		(try! (nft-mint? stacksies token-id recipient))
		(var-set last-token-id token-id)
		(ok token-id)
	)
)
```

- clarity-coin.clar:

```
(impl-trait
'SP3FBR2AGK5H9QBDH3EEN6DF8EK8JY7RX8QJ5SVTE.sip-010-trait-ft-standard.sip-010-trai
t)

(define-constant contract-owner tx-sender)
(define-constant err-owner-only (err u100))
(define-constant err-not-token-owner (err u101))

;; No maximum supply!
(define-fungible-token clarity-coin)

(define-public (transfer (amount uint) (sender principal) (recipient principal) (memo (optional
(buff 34))))
	(begin
		(asserts! (is-eq tx-sender sender) err-not-token-owner)
		(try! (ft-transfer? clarity-coin amount sender recipient))
		(match memo to-print (print to-print) 0x)
		(ok true)
	)
)

(define-read-only (get-name)
	(ok "Clarity Coin")
)

(define-read-only (get-symbol)
	(ok "CC")
)

(define-read-only (get-decimals)
	(ok u0)
)

(define-read-only (get-balance (who principal))
	(ok (ft-get-balance clarity-coin who))
)

(define-read-only (get-total-supply)
	(ok (ft-get-supply clarity-coin))
)
```

```
(define-read-only (get-token-uri)
	(ok none)
)

(define-public (mint (amount uint) (recipient principal))
	(begin
		(asserts! (is-eq tx-sender contract-owner) err-owner-only)
		(ft-mint? clarity-coin amount recipient)
	)
)
```

- smart-claimant.clar

```
;; smart-claimant
;; An ad-hoc smart contract to specifically function as the beneficiary
;; of a timelocked-wallet contract.

(define-public (claim)
	(begin
		(try! (as-contract (contract-call? .timelocked-wallet claim)))
		(let
			(
				(total-balance (as-contract (stx-get-balance tx-sender)))
				(share (/ total-balance u4))
			)
			(try! (as-contract (stx-transfer? share tx-sender
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)))
			(try! (as-contract (stx-transfer? share tx-sender
'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2)))
			(try! (as-contract (stx-transfer? share tx-sender
'ST21HMSJATHZ888PD0S0SSTWP4J61TCRJYEVQ0STB)))
			(try! (as-contract (stx-transfer? (stx-get-balance tx-sender) tx-sender
'ST2QXSK64YQX3CQPC530K79XWQ98XFAM9W3XKEH3N)))
			(ok true)
		)
	)
)
```

- timelocked-wallet.clar

```
;; timelocked-wallet
;; A time-locked vault contract that becomes eligible to claim by the beneficiary after a certain
block-height has been reached.
```

```
;; Owner
(define-constant contract-owner tx-sender)

;; Errors
(define-constant err-owner-only (err u100))
(define-constant err-already-locked (err u101))
(define-constant err-unlock-in-past (err u102))
(define-constant err-no-value (err u103))
(define-constant err-beneficiary-only (err u104))
(define-constant err-unlock-height-not-reached (err u105))

;; Data
(define-data-var beneficiary (optional principal) none)
(define-data-var unlock-height uint u0)

(define-public (lock (new-beneficiary principal) (unlock-at uint) (amount uint))
	(begin
		(asserts! (is-eq tx-sender contract-owner) err-owner-only)
		(asserts! (is-none (var-get beneficiary)) err-already-locked)
		(asserts! (> unlock-at block-height) err-unlock-in-past)
		(asserts! (> amount u0) err-no-value)
		(try! (stx-transfer? amount tx-sender (as-contract tx-sender)))
		(var-set beneficiary (some new-beneficiary))
		(var-set unlock-height unlock-at)
		(ok true)
	)
)

(define-public (bestow (new-beneficiary principal))
	(begin
		(asserts! (is-eq (some tx-sender) (var-get beneficiary)) err-beneficiary-only)
		(var-set beneficiary (some new-beneficiary))
		(ok true)
	)
)

(define-public (claim)
	(begin
		(asserts! (is-eq (some tx-sender) (var-get beneficiary)) err-beneficiary-only)
		(asserts! (>= block-height (var-get unlock-height)) err-unlock-height-not-reached)
		(as-contract (stx-transfer? (stx-get-balance tx-sender) tx-sender (unwrap-panic
(var-get beneficiary))))
	)
)
```

- sip009-nft.clar:

```
;; sip009-nft
;; A SIP009-compliant NFT with a mint function.

(impl-trait 'SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait.nft-trait)

(define-constant contract-owner tx-sender)

(define-constant err-owner-only (err u100))
(define-constant err-token-id-failure (err u101))
(define-constant err-not-token-owner (err u102))

(define-non-fungible-token stacksies uint)
(define-data-var token-id-nonce uint u0)

(define-read-only (get-last-token-id)
        (ok (var-get token-id-nonce))
)

(define-read-only (get-token-uri (token-id uint))
        (ok none)
)

(define-read-only (get-owner (token-id uint))
        (ok (nft-get-owner? stacksies token-id))
)

(define-public (transfer (token-id uint) (sender principal) (recipient principal))
        (begin
                (asserts! (is-eq tx-sender sender) err-not-token-owner)
                (nft-transfer? stacksies token-id sender recipient)
        )
)

(define-public (mint (recipient principal))
        (let ((token-id (+ (var-get token-id-nonce) u1)))
                (asserts! (is-eq tx-sender contract-owner) err-owner-only)
                (try! (nft-mint? stacksies token-id recipient))
                (asserts! (var-set token-id-nonce token-id) err-token-id-failure)
                (ok token-id)
        )
)
```

- sip010-token.clar:

```
;; sip010-token
;; A SIP010-compliant fungible token with a mint function.

(impl-trait
'SP3FBR2AGK5H9QBDH3EEN6DF8EK8JY7RX8QJ5SVTE.sip-010-trait-ft-standard.sip-010-trai
t)

(define-constant contract-owner tx-sender)

(define-fungible-token amazing-coin u100000000)

(define-constant err-owner-only (err u100))
(define-constant err-not-token-owner (err u102))

(define-public (transfer (amount uint) (sender principal) (recipient principal) (memo (optional
(buff 34))))
	(begin
		(asserts! (is-eq tx-sender sender) err-not-token-owner)
		(ft-transfer? amazing-coin amount sender recipient)
	)
)

(define-read-only (get-name)
	(ok "Amazing Coin")
)

(define-read-only (get-symbol)
	(ok "AC")
)

(define-read-only (get-decimals)
	(ok u6)
)

(define-read-only (get-balance (who principal))
	(ok (ft-get-balance amazing-coin who))
)

(define-read-only (get-total-supply)
	(ok (ft-get-supply amazing-coin))
)

(define-read-only (get-token-uri)
	(ok none)
)

(define-public (mint (amount uint) (recipient principal))
	(begin
		(asserts! (is-eq tx-sender contract-owner) err-owner-only)
		(ft-mint? amazing-coin amount recipient)
	)
)
```

- tiny-market.clar

```
;; tiny-market
;; A tiny NFT marketplace that allows users to list NFT for sale. They can specify the following:
;; - The NFT token to sell.
;; - Listing expiry in block height.
;; - The payment asset, either STX or a SIP010 fungible token.
;; - The NFT price in said payment asset.
;; - An optional intended taker. If set, only that principal will be able to fulfil the listing.

(use-trait nft-trait 'SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait.nft-trait)
(use-trait ft-trait
'SP3FBR2AGK5H9QBDH3EEN6DF8EK8JY7RX8QJ5SVTE.sip-010-trait-ft-standard.sip-010-trai
t)

(define-constant contract-owner tx-sender)

;; listing errors
(define-constant err-expiry-in-past (err u1000))
(define-constant err-price-zero (err u1001))

;; cancelling and fulfiling errors
(define-constant err-unknown-listing (err u2000))
(define-constant err-unauthorised (err u2001))
(define-constant err-listing-expired (err u2002))
(define-constant err-nft-asset-mismatch (err u2003))
(define-constant err-payment-asset-mismatch (err u2004))
(define-constant err-maker-taker-equal (err u2005))
(define-constant err-unintended-taker (err u2006))
```

```
(define-constant err-asset-contract-not-whitelisted (err u2007))
(define-constant err-payment-contract-not-whitelisted (err u2008))

(define-map listings
	uint
	{
		maker: principal,
		taker: (optional principal),
		token-id: uint,
		nft-asset-contract: principal,
		expiry: uint,
		price: uint,
		payment-asset-contract: (optional principal)
	}
)

(define-data-var listing-nonce uint u0)

(define-map whitelisted-asset-contracts principal bool)

(define-read-only (is-whitelisted (asset-contract principal))
	(default-to false (map-get? whitelisted-asset-contracts asset-contract))
)

(define-public (set-whitelisted (asset-contract principal) (whitelisted bool))
	(begin
		(asserts! (is-eq contract-owner tx-sender) err-unauthorised)
		(ok (map-set whitelisted-asset-contracts asset-contract whitelisted))
	)
)

(define-private (transfer-nft (token-contract <nft-trait>) (token-id uint) (sender principal) (recipient
principal))
	(contract-call? token-contract transfer token-id sender recipient)
)

(define-private (transfer-ft (token-contract <ft-trait>) (amount uint) (sender principal) (recipient
principal))
	(contract-call? token-contract transfer amount sender recipient none)
)

(define-public (list-asset (nft-asset-contract <nft-trait>) (nft-asset {taker: (optional principal),
token-id: uint, expiry: uint, price: uint, payment-asset-contract: (optional principal)}))
	(let ((listing-id (var-get listing-nonce)))
		(asserts! (is-whitelisted (contract-of nft-asset-contract))
err-asset-contract-not-whitelisted)
		(asserts! (> (get expiry nft-asset) block-height) err-expiry-in-past)
		(asserts! (> (get price nft-asset) u0) err-price-zero)
		(asserts! (match (get payment-asset-contract nft-asset) payment-asset
(is-whitelisted payment-asset) true) err-payment-contract-not-whitelisted)
		(try! (transfer-nft nft-asset-contract (get token-id nft-asset) tx-sender (as-contract
tx-sender)))
		(map-set listings listing-id (merge {maker: tx-sender, nft-asset-contract:
(contract-of nft-asset-contract)} nft-asset))
		(var-set listing-nonce (+ listing-id u1))
		(ok listing-id)
	)
)

(define-read-only (get-listing (listing-id uint))
	(map-get? listings listing-id)
)

(define-public (cancel-listing (listing-id uint) (nft-asset-contract <nft-trait>))
	(let (
		(listing (unwrap! (map-get? listings listing-id) err-unknown-listing))
		(maker (get maker listing))
		)
		(asserts! (is-eq maker tx-sender) err-unauthorised)
		(asserts! (is-eq (get nft-asset-contract listing) (contract-of nft-asset-contract))
err-nft-asset-mismatch)
		(map-delete listings listing-id)
		(as-contract (transfer-nft nft-asset-contract (get token-id listing) tx-sender maker))
	)
)

(define-private (assert-can-fulfil (nft-asset-contract principal) (payment-asset-contract (optional
principal)) (listing {maker: principal, taker: (optional principal), token-id: uint, nft-asset-contract:
principal, expiry: uint, price: uint, payment-asset-contract: (optional principal)}))
	(begin
		(asserts! (not (is-eq (get maker listing) tx-sender)) err-maker-taker-equal)
		(asserts! (match (get taker listing) intended-taker (is-eq intended-taker tx-sender)
true) err-unintended-taker)
		(asserts! (< block-height (get expiry listing)) err-listing-expired)
		(asserts! (is-eq (get nft-asset-contract listing) nft-asset-contract)
err-nft-asset-mismatch)
		(asserts! (is-eq (get payment-asset-contract listing) payment-asset-contract)
err-payment-asset-mismatch)
```

```
			(ok true)
		)
)

(define-public (fulfil-listing-stx (listing-id uint) (nft-asset-contract <nft-trait>))
		(let (
			(listing (unwrap! (map-get? listings listing-id) err-unknown-listing))
			(taker tx-sender)
			)
			(try! (assert-can-fulfil (contract-of nft-asset-contract) none listing))
			(try! (as-contract (transfer-nft nft-asset-contract (get token-id listing) tx-sender
taker)))
			(try! (stx-transfer? (get price listing) taker (get maker listing)))
			(map-delete listings listing-id)
			(ok listing-id)
		)
)

(define-public (fulfil-listing-ft (listing-id uint) (nft-asset-contract <nft-trait>)
(payment-asset-contract <ft-trait>))
		(let (
			(listing (unwrap! (map-get? listings listing-id) err-unknown-listing))
			(taker tx-sender)
			)
			(try! (assert-can-fulfil (contract-of nft-asset-contract) (some (contract-of
payment-asset-contract)) listing))
			(try! (as-contract (transfer-nft nft-asset-contract (get token-id listing) tx-sender
taker)))
			(try! (transfer-ft payment-asset-contract (get price listing) taker (get maker
listing)))
			(map-delete listings listing-id)
			(ok listing-id)
		)
)
```

Stacking

Stacking is implemented as a smart contract using Clarity. You can always find the Stacking contract identifier using the Stacks Blockchain API `v2/pox` endpoint.

Currently stacking uses the pox-4 contract. The deployed pox-4 contract can be viewed in the explorer.

In this walkthrough, we'll cover the entire stacking contract from start to finish, including descriptions of the various functions and errors, and when you might use/encounter them.

Rather than walking through the contract line by line, which you can do by simply reading the contract code and the comments, we'll instead explore it from the perspective of conducting stacking operations, incuding solo stacking, delegating, and running a pool.

At the bottom you will find a list of some errors you may run into and their explanations.

There are a few utilities that make interacting with this contract easier including Lockstacks as a UI and the @stacks/stacking package for a JS library.

Hiro has a detailed guide available for stacking using this library as well as a Nakamoto guide specifically for the additions made to work with `pox-4`.
Full Stacking Contract Source Code w/ comments

Prerequisites

If you are not familiar with stacking as a concept, it will be useful to familiarize yourself with that first before diving into the contract.

Solo Stacking

Solo stacking is the simplest option, and begins by calling the `stack-stx` function.

stack-stx

This function locks up the given amount of STX for the given lock period (number of reward cycles) for the `tx-sender`.

Here's the full code for that function, then we'll dive into how it works below that.

```
(define-public (stack-stx (amount-ustx uint)

                    (pox-addr (tuple (version (buff 1)) (hashbytes (buff 32))))

                    (start-burn-ht uint)

                    (lock-period uint)

                    (signer-sig (optional (buff 65)))

                    (signer-key (buff 33))

                    (max-amount uint)
                    (auth-id uint))

;; this stacker's first reward cycle is the _next_ reward cycle

(let ((first-reward-cycle (+ u1 (current-pox-reward-cycle)))

      (specified-reward-cycle (+ u1 (burn-height-to-reward-cycle start-burn-ht))))

  ;; the start-burn-ht must result in the next reward cycle, do not allow stackers

  ;;  to "post-date" their `stack-stx` transaction

  (asserts! (is-eq first-reward-cycle specified-reward-cycle)

            (err ERR_INVALID_START_BURN_HEIGHT))


  ;; must be called directly by the tx-sender or by an allowed contract-caller

  (asserts! (check-caller-allowed)

            (err ERR_STACKING_PERMISSION_DENIED))


  ;; tx-sender principal must not be stacking

  (asserts! (is-none (get-stacker-info tx-sender))

    (err ERR_STACKING_ALREADY_STACKED))


  ;; tx-sender must not be delegating

  (asserts! (is-none (get-check-delegation tx-sender))

    (err ERR_STACKING_ALREADY_DELEGATED))


  ;; the Stacker must have sufficient unlocked funds
```

```
    (asserts! (>= (stx-get-balance tx-sender) amount-ustx)
      (err ERR_STACKING_INSUFFICIENT_FUNDS))

    ;; Validate ownership of the given signer key
    (try! (consume-signer-key-authorization pox-addr (- first-reward-cycle u1) "stack-stx"
lock-period signer-sig signer-key amount-ustx max-amount auth-id))

    ;; ensure that stacking can be performed
    (try! (can-stack-stx pox-addr amount-ustx first-reward-cycle lock-period))

    ;; register the PoX address with the amount stacked
    (let ((reward-set-indexes (try! (add-pox-addr-to-reward-cycles pox-addr first-reward-cycle
lock-period amount-ustx tx-sender signer-key))))
        ;; add stacker record
       (map-set stacking-state
         { stacker: tx-sender }
         { pox-addr: pox-addr,
           reward-set-indexes: reward-set-indexes,
           first-reward-cycle: first-reward-cycle,
           lock-period: lock-period,
           delegated-to: none })

        ;; return the lock-up information, so the node can actually carry out the lock.
```

(ok { stacker: tx-sender, lock-amount: amount-ustx, signer-key: signer-key, unlock-burn-height: (reward-cycle-to-burn-height (+ first-reward-cycle lock-period)) }))))

First let's cover the needed parameters.

`amount-ustx` is the amount of STX you would like to lock, denoted in micro-STX, or uSTX (1 STX = 1,000,000 uSTX)
`pox-addr` is a tuple that encodes the Bitcoin address to be used for the PoX rewards, details below
`start-burn-ht` is the Bitcoin block height you would like to begin stacking. You will receive rewards in the reward cycle following `start-burn-ht`. Importantly, `start-burn-ht` may not be further into the future than the next reward cycle, and in most cases should be set to the current burn block height.
`lock-period` sets the number of reward cycles you would like you lock your STX for, this can be 1-12
`signer-sig` is a unique generatedf signature that proves ownership of this signer. Further details for its role and how to generate it can be found in the How to Stack doc
`signer-key` is the public key of your signer, more details in the How to Run a Signer doc
`max-amount` sets the maximum amount allowed to be stacked during the provided stacking period
`auth-id` is a unique string to prevent re-use of this stacking transaction

It's important to make sure that these fields match what you pass in to the signer signature generation. If they don't, you will likely get error 35 (`ERR_INVALID_SIGNATURE_PUBKEY`) when trying to submit this transaction as the signer signature will not be valid.

Supported Reward Address Types
For the `pox-addr` field, the `version` buffer must represent what kind of bitcoin address is being submitted. These are all the possible values you can pass here depending on your address type:

(define-constant ADDRESS_VERSION_P2PKH 0x00)

(define-constant ADDRESS_VERSION_P2SH 0x01)

(define-constant ADDRESS_VERSION_P2WPKH 0x02)

```
(define-constant ADDRESS_VERSION_P2WSH 0x03)

(define-constant ADDRESS_VERSION_NATIVE_P2WPKH 0x04)

(define-constant ADDRESS_VERSION_NATIVE_P2WSH 0x05)

(define-constant ADDRESS_VERSION_NATIVE_P2TR 0x06)
```

The `hashbytes` are the 20 hash bytes of the bitcoin address. You can obtain that from a bitcoin library, for instance using `bitcoinjs-lib`:

```
const btc = require("bitcoinjs-lib");

console.log(

 "0x" +

   btc.address

     .fromBase58Check("1C56LYirKa3PFXFsvhSESgDy2acEHVAEt6")

     .hash.toString("hex")

);
```

The first thing the `stack-stx` function will do perform several checks including:

The `start-burn-ht` results in the next reward cycle
The function is being called by the `tx-sender` or an allowed contract caller
The `tx-sender` is not currently stacking or delegating
The `tx-sender` has enough funds
The given `signer-key` is valid, proving ownership
Stacking can be performed. This is determined through additional checks including if the amount meets the minimum stacking threshold and if the lock period and provided Bitcoin address are valid

You can explore the private functions that handle these checks to see exactly how they do so.

Next we register the provided PoX address for the next reward cycle, assign its specific reward slot, and add it to the `stacking-state` map, which keeps track of all current stackers per reward cycle.

Finally we return the lock up information so the node can carry out the lock by reading this information. This step is what actually locks the STX and prevents the stacker from using them on-chain.

From here, the locked STX tokens will be unlocked automatically at the end of the lock period.

The other option is that the stacker can call the `stack-increase` or `stack-extend` functions to either increase the amount of STX they have locked or increase the amount of time to lock them, respectively.

Delegated Stacking

Delegated stacking has a few additional steps to it. It is essentially a three-step process:

Delegator delegates their STX to a pool operator

Pool operator stacks their delegated STX

Pool operator commits an aggregate of all STX delegated to them

Each of these steps has a corresponding function. Let's dig into them.

delegate-stx

This function is called by the individual stacker delegating their STX to a pool operator. An individual stacker choosing to delegate does not need to run their own signer.

This function does not actually lock the STX, but just allows the pool operator to issue the lock.

```
(define-public (delegate-stx (amount-ustx uint)

                             (delegate-to principal)

                             (until-burn-ht (optional uint))

                             (pox-addr (optional { version: (buff 1), hashbytes: (buff 32) })))



  (begin

    ;; must be called directly by the tx-sender or by an allowed contract-caller
```

```
(asserts! (check-caller-allowed)
          (err ERR_STACKING_PERMISSION_DENIED))

;; delegate-stx no longer requires the delegator to not currently
;; be stacking.
;; delegate-stack-* functions assert that
;; 1. users can't swim in two pools at the same time.
;; 2. users can't switch pools without cool down cycle.
;;    Other pool admins can't increase or extend.
;; 3. users can't join a pool while already directly stacking.

;; pox-addr, if given, must be valid
(match pox-addr
  address
    (asserts! (check-pox-addr-version (get version address))
        (err ERR_STACKING_INVALID_POX_ADDRESS))
  true)

;; tx-sender must not be delegating
(asserts! (is-none (get-check-delegation tx-sender))
  (err ERR_STACKING_ALREADY_DELEGATED))
```

```
    ;; add delegation record
    (map-set delegation-state
      { stacker: tx-sender }
      { amount-ustx: amount-ustx,
        delegated-to: delegate-to,
        until-burn-ht: until-burn-ht,
        pox-addr: pox-addr })


    (ok true)))
```

This function takes a few parameters:

- `amount-ustx` is the amount the stacker is delegating denoted in uSTX
- `delegate-to` is the Stacks address of the pool operator (or delegate) being delegated to
- `until-burn-ht` is an optional parameter that describes when this delegation expires
- `pox-addr` is an optional Bitcoin address. If this is provided, the pool operator must send rewards to this address. If it is not provided, it is up to the discretion of the pool operator where to send the rewards.

It first runs through a few checks to ensure that the function caller is allowed, the provided `pox-addr` is valid, and that the stacker is not already delegating.

Finally it updates the `delegation-state` of the contract with the details.

At this point, no STX have been locked. The pool operator next needs to call the `delegate-stack-stx` function in order to partially lock the STX.

delegate-stack-stx

```
;; As a delegate, stack the given principal's STX using partial-stacked-by-cycle

;; Once the delegate has stacked > minimum, the delegate should call
stack-aggregation-commit
```

```
(define-public (delegate-stack-stx (stacker principal)

                                  (amount-ustx uint)

                                  (pox-addr { version: (buff 1), hashbytes: (buff 32) })

                                  (start-burn-ht uint)

                                  (lock-period uint))

  ;; this stacker's first reward cycle is the _next_ reward cycle

  (let ((first-reward-cycle (+ u1 (current-pox-reward-cycle)))

        (specified-reward-cycle (+ u1 (burn-height-to-reward-cycle start-burn-ht)))

        (unlock-burn-height (reward-cycle-to-burn-height (+ (current-pox-reward-cycle) u1
lock-period))))

    ;; the start-burn-ht must result in the next reward cycle, do not allow stackers

    ;;  to "post-date" their `stack-stx` transaction

    (asserts! (is-eq first-reward-cycle specified-reward-cycle)

              (err ERR_INVALID_START_BURN_HEIGHT))



    ;; must be called directly by the tx-sender or by an allowed contract-caller

    (asserts! (check-caller-allowed)

      (err ERR_STACKING_PERMISSION_DENIED))



    ;; stacker must have delegated to the caller

    (let ((delegation-info (unwrap! (get-check-delegation stacker) (err
ERR_STACKING_PERMISSION_DENIED))))

      ;; must have delegated to tx-sender
      (asserts! (is-eq (get delegated-to delegation-info) tx-sender)
            (err ERR_STACKING_PERMISSION_DENIED))
      ;; must have delegated enough stx
      (asserts! (>= (get amount-ustx delegation-info) amount-ustx)
            (err ERR_DELEGATION_TOO_MUCH_LOCKED))
      ;; if pox-addr is set, must be equal to pox-addr
      (asserts! (match (get pox-addr delegation-info)
                specified-pox-addr (is-eq pox-addr specified-pox-addr)
                true)
            (err ERR_DELEGATION_POX_ADDR_REQUIRED))
      ;; delegation must not expire before lock period
      (asserts! (match (get until-burn-ht delegation-info)
                until-burn-ht (>= until-burn-ht
                            unlock-burn-height)
              true)
            (err ERR_DELEGATION_EXPIRES_DURING_LOCK))
      )

    ;; stacker principal must not be stacking
    (asserts! (is-none (get-stacker-info stacker))
      (err ERR_STACKING_ALREADY_STACKED))
```

```
;; the Stacker must have sufficient unlocked funds

(asserts! (>= (stx-get-balance stacker) amount-ustx)

  (err ERR_STACKING_INSUFFICIENT_FUNDS))


;; ensure that stacking can be performed

(try! (minimal-can-stack-stx pox-addr amount-ustx first-reward-cycle lock-period))


;; register the PoX address with the amount stacked via partial stacking

;;   before it can be included in the reward set, this must be committed!

(add-pox-partial-stacked pox-addr first-reward-cycle lock-period amount-ustx)


;; add stacker record

(map-set stacking-state

  { stacker: stacker }

  { pox-addr: pox-addr,

    first-reward-cycle: first-reward-cycle,

    reward-set-indexes: (list),

    lock-period: lock-period,

    delegated-to: (some tx-sender) })


;; return the lock-up information, so the node can actually carry out the lock.

(ok { stacker: stacker,
```

lock-amount: amount-ustx,

unlock-burn-height: unlock-burn-height })))

At this point, the stacker has given their permission to the pool operator to delegate their STX on their behalf.

The delegation is now in the hands of the pool operator to stack these delegated STX using the `delegate-stack-stx` function.

This function can only be called after a stacker has called their respective `delegate-stx` function and must be called by the pool operator for each instance of a stacker calling `delegate-stx`.

Like the other functions, this function takes in several parameters and runs through several checks before updating the contract state.

`stacker` is the principal of the stacker who has delegated their STX

`amount-ustx` is the amount they have delegated in uSTX

`pox-addr` is the Bitcoin address the rewards will be sent to. If the stacker passed this field in to their `delegate-stx` function, this must be the same value

`start-burn-ht` corresponds to the field passed in by the stacker

`lock-period` corresponds to the same field that the stacker passed in

Now we assign a few variables using `let` before running several checks.

`first-reward-cycle` is set to the next reward cycle automatically

`specified-reward-cycle` is the reward cycle that the passed-in `start-burn-ht` parameter falls within

`unlock-burn-height` is the Bitcoin block height at which the stackers STX will be unlocked

Now we proceed to run through some checks including:

The first reward cycle is the same as the specified reward cycle

The function is being called by the `tx-sender` or an approved contract

That the stacker actually delegated to the contract caller

Then we get the information from the delegation and make sure the information we are passing here matches it

That the stacker is not currently stacking

They have sufficient unlocked funds

Run the `minimal-can-stack-stx` to run a few additional checks to make sure stacking can occur

Next we call a function called `add-pox-partial-stacked` which will add this stacker to a `partial-stacked-by-cycle` map.

After those checks and partial stacking, we update the `stacking-state` map and return the information for the node to process.

At this point this stacker's STX are considered partially stacked. We still need to perform one more step as the pool operator in order to officially lock them.

stack-aggregation-commit

The `stack-aggregation-commit` function is just a wrapper for the private `inner-stack-aggregation-commit` function, so that is the source code included here.

;; Commit partially stacked STX and allocate a new PoX reward address slot.

;;   This allows a stacker/delegate to lock fewer STX than the minimal threshold in multiple transactions,

;;   so long as: 1. The pox-addr is the same.

;;   2. This "commit" transaction is called _before_ the PoX anchor block.

;;   This ensures that each entry in the reward set returned to the stacks-node is greater than the threshold,

;;   but does not require it be all locked up within a single transaction

;;

;; Returns (ok uint) on success, where the given uint is the reward address's index in the list of reward

;; addresses allocated in this reward cycle. This index can then be passed to `stack-aggregation-increase`

;; to later increment the STX this PoX address represents, in amounts less than the stacking minimum.

```
;;

;; *New in Stacks 2.1.*

(define-private (inner-stack-aggregation-commit (pox-addr { version: (buff 1), hashbytes: (buff
32) })

                                                (reward-cycle uint)

                                                (signer-sig (optional (buff 65)))

                                                (signer-key (buff 33))

                                                (max-amount uint)

                                                (auth-id uint))

 (let ((partial-stacked

        ;; fetch the partial commitments

        (unwrap! (map-get? partial-stacked-by-cycle { pox-addr: pox-addr, sender: tx-sender,
reward-cycle: reward-cycle })

                (err ERR_STACKING_NO_SUCH_PRINCIPAL))))

   ;; must be called directly by the tx-sender or by an allowed contract-caller

   (asserts! (check-caller-allowed)

            (err ERR_STACKING_PERMISSION_DENIED))

   (let ((amount-ustx (get stacked-amount partial-stacked)))

     (try! (consume-signer-key-authorization pox-addr reward-cycle "agg-commit" u1 signer-sig
signer-key amount-ustx max-amount auth-id))

     (try! (can-stack-stx pox-addr amount-ustx reward-cycle u1))

     ;; Add the pox addr to the reward cycle, and extract the index of the PoX address

     ;; so the delegator can later use it to call stack-aggregation-increase.
```

```
  (let ((add-pox-addr-info
          (add-pox-addr-to-ith-reward-cycle
            u0
            { pox-addr: pox-addr,
              first-reward-cycle: reward-cycle,
              num-cycles: u1,
              reward-set-indexes: (list),
              stacker: none,
              signer: signer-key,
              amount-ustx: amount-ustx,
              i: u0 }))
       (pox-addr-index (unwrap-panic
          (element-at (get reward-set-indexes add-pox-addr-info) u0))))

    ;; don't update the stacking-state map,
    ;;  because it _already has_ this stacker's state
    ;; don't lock the STX, because the STX is already locked
    ;;
    ;; clear the partial-stacked state, and log it
    (map-delete partial-stacked-by-cycle { pox-addr: pox-addr, sender: tx-sender, reward-cycle:
reward-cycle })
    (map-set logged-partial-stacked-by-cycle { pox-addr: pox-addr, sender: tx-sender,
reward-cycle: reward-cycle } partial-stacked)
```

(ok pox-addr-index)))))

This function contains many of the same parameters as the `stack-stx` function, with a few changes:

`pox-addr` is a tuple that encodes the Bitcoin address to be used for the PoX rewards, details below

`reward-cycle` is the reward cycle that these delegated STX will be stacked in

`signer-sig` is a unique generated signature that proves ownership of this signer. Further details for its role and how to generate it can be found in the [How to Stack]() doc

`signer-key` is the public key of your signer, more details in the [How to Run a Signer]() doc

`max-amount` sets the maximum amount allowed to be stacked during the provided stacking period

`auth-id` is a unique string to prevent re-use of this stacking transaction

It's important to make sure that these fields match what you pass in to the signer signature generation. If they don't, you will likely get error 35 (`ERR_INVALID_SIGNATURE_PUBKEY`) when trying to submit this transaction as the signer signature will not be valid.

The first thing we do is pull the partially stacked STX that have been added to the contract state via our `delegate-stack-stx` calls.

Next we run through many of the same checks as in previous functions:

Making sure the contract caller is allowed

Making sure the signer signature is valid

Making sure we can actually perform this stacking operation

Now we take all of the delegated STX and actually add it to the reward cycle. At this point the pool operator has a reward slot in the chosen cycle.

We don't need to update the stacking state because we already did that in the `delegate-stack-stx` function.

All we need to do is delete and log the partially stacked STX state.

How Stacking Reward Distribution Works

All of the above stacking functions take in a `pox-reward` field that corresponds to a Bitcoin address where BTC rewards will be sent. It's important to understand how these addresses are used and how reward distribution is handled in general.

How Bitcoin rewards are distributed is primarily up to the discretion of the pool operator. Since PoX reward distributions are handled using Bitcoin transactions, there is currently not an effective way to automate their distribution to individual delegated stackers.

Let's go over the role of `pox-addr` in each function and how it should be used.

stack-stx

This is the simplest option and simply corresponds to the Bitcoin address that the stacker would like to receive their rewards.

delegate-stx

In this function, which is the one that the delegator will be calling to give permission to the pool operator to stack on their behalf, the `pox-addr` argument is optional.

If no `pox-addr` argument is not passed in, the pool operator determines where this delegator's rewards are sent.

If a `pox-addr` is passed in, then rewards must be distributed to that address. **However, if this is passed in, the delegator must have enough STX to meet the minimum stacking amount.**

The reason is because there are a finite amount of reward slots (4,000) and each `pox-addr` takes up one of these reward slots.

Stackers need to be able to stack the minimum in order to be eligible for one of these reward slots. A delegator may choose to delegate to a pool (even if they meet the minimum stacking requirement) if they do not want to handle the infrastructure of running a signer or the actual stacking operations, which is why this option exists.

delegate-stack-stx and stack-aggregation-commit

In both of these functions, `pox-addr` corresponds to the address where the pool operator would like the rewards to be sent.

At this point, it is up to the pool operator to determine how to allocate rewards. In most cases, a pool operator will use a wrapper contract in order to transparently track this information on-chain, and manually send rewards out to participants according to the proportion that they delegated.

Errors

You may encounter several errors when trying to perform stacking operations. We won't cover them all in detail here, as you can see the error in the failed transaction and trace the source code to find it.

But we will cover some of the more common errors you may encounter, what they mean, and how to resolve them.

Error 35 - `ERR_INVALID_SIGNATURE_PUBKEY`

This is likely the most common error you will encounter, and you'll usually see it in a failed `stack-stx` or `stack-aggregation-commit` transaction.

This error actually occurs in the `consume-signer-key-authorization` which is called any time we pass in a signer signature.

This means one of two things:

- The public key you used to generate the signer signature is not the same as the one you are passing in to the `signer-key` field
- One of the fields you passed in to generate your signer signature does not match the field you passed in to either the `stack-stx` or `stack-aggregation-commit` function

To fix this, check all of the data you passed in to see where the mismatch is.

Error 4 - ERR_STACKING_NO_SUCH_PRINCIPAL

The stacking contract looks up partially stacked stx (after you have called `delegate-stack-stx`) with the lookup key `(pox-addr, stx-address, reward-cycle`. This error means that either when you generated your signature or called the `stack-aggregation-commit` function, you passed in the wrong parameter for one of these. More information in the stacking guide.

BNS

The Bitcoin Name System (BNS) is implemented as a smart contract using Clarity.

Below is a list of public and read-only functions as well as error codes that can be returned by those methods:

- Public functions
- Read-only functions
- Error codes

Public functions

name-import

Input: `(buff 20), (buff 48), principal, (buff 20)`

Output: `(response bool int)`

Signature: `(name-import namespace name beneficiary zonefile-hash)`

Description:

Imports name to a revealed namespace. Each imported name is given both an owner and some off-chain state.

name-preorder

Input: `(buff 20), uint`

Output: `(response uint int)`

Signature: `(name-preorder hashed-salted-fqn stx-to-burn)`

Description:

Preorders a name by telling all BNS nodes the salted hash of the BNS name. It pays the registration fee to the namespace owner's designated address.

name-register

Input: `(buff 20), (buff 48), (buff 20), (buff 20)`

Output: `(response bool int)`

Signature: `(name-register namespace name salt zonefile-hash)`

Description:

Reveals the salt and the name to all BNS nodes, and assigns the name an initial public key hash and zone file hash.

name-renewal

Input: `(buff 20), (buff 48), uint, (optional principal), (optional (buff 20))`

Output: `(response bool int)`

Signature: `(name-renewal namespace name stx-to-burn new-owner zonefile-hash)`

Description:

Depending in the namespace rules, a name can expire. For example, names in the .id namespace expire after 2 years. You need to send a name renewal every so often to keep your name.

You will pay the registration cost of your name to the namespace's designated burn address when you renew it. When a name expires, it enters a "grace period". The period is set to 5000 blocks (a month) but can be configured for each namespace.

It will stop resolving in the grace period, and all of the above operations will cease to be honored by the BNS consensus rules. You may, however, send a NAME_RENEWAL during this grace period to preserve your name. After the grace period, everybody can register that name again. If your name is in a namespace where names do not expire, then you never need to use this transaction.

name-revoke

Input: `(buff 20), (buff 48)`

Output: `(response bool int)`

Signature: `(name-revoke namespace name)`

Description:

Makes a name unresolvable. The BNS consensus rules stipulate that once a name is revoked, no one can change its public key hash or its zone file hash. The name's zone file hash is set to null to prevent it from resolving. You should only do this if your private key is compromised, or if you want to render your name unusable for whatever reason.

name-transfer

Input: `(buff 20), (buff 48), principal, (optional (buff 20))`

Output: `(response bool int)`

Signature: `(name-transfer namespace name new-owner zonefile-hash)`

Description:

Changes the name's public key hash. You would send a name transfer transaction if you wanted to:

Change your private key

Send the name to someone else or

Update your zone file

When transferring a name, you have the option to also clear the name's zone file hash (i.e. set it to null). This is useful for when you send the name to someone else, so the recipient's name does not resolve to your zone file.

name-update

Input: `(buff 20), (buff 48), (buff 20)`

Output: `(response bool int)`

Signature: `(name-update namespace name zonefile-hash)`

Description:

Changes the name's zone file hash. You would send a name update transaction if you wanted to change the name's zone file contents. For example, you would do this if you want to deploy your own Gaia hub and want other people to read from it.

namespace-preorder

Input: `(buff 20), uint`

Output: `(response uint int)`

Signature: `(namespace-preorder hashed-salted-namespace stx-to-burn)`

Description:

Registers the salted hash of the namespace with BNS nodes, and burns the requisite amount of cryptocurrency. Additionally, this step proves to the BNS nodes that user has honored the BNS consensus rules by including a recent consensus hash in the transaction. Returns pre-order's expiration date (in blocks).

namespace-ready

Input: `(buff 20)`

Output: `(response bool int)`

Signature: `(namespace-ready namespace)`

Description:

Launches the namespace and makes it available to the public. Once a namespace is launched, anyone can register a name in it if they pay the appropriate amount of cryptocurrency.

namespace-reveal

Input: `(buff 20), (buff 20), uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, uint, principal`

Output: `(response bool int)`

Signature: `(namespace-reveal namespace namespace-salt p-func-base p-func-coeff p-func-b1 p-func-b2 p-func-b3 p-func-b4 p-func-b5 p-func-b6 p-func-b7 p-func-b8 p-func-b9 p-func-b10 p-func-b11 p-func-b12 p-func-b13 p-func-b14 p-func-b15 p-func-b16 p-func-non-alpha-discount p-func-no-vowel-discount lifetime namespace-import)`

Description:

Reveals the salt and the namespace ID (after a namespace preorder). It reveals how long names last in this namespace before they expire or must be renewed, and it sets a price function for the namespace that determines how cheap or expensive names its will be.All of the parameters prefixed by `p` make up the `price function`. These parameters govern the pricing and lifetime of names in the namespace.

The rules for a namespace are as follows:

a name can fall into one of 16 buckets, measured by length. Bucket 16 incorporates all names at least 16 characters long.
the pricing structure applies a multiplicative penalty for having numeric characters, or punctuation characters.
the price of a name in a bucket is `((coeff) * (base) ^ (bucket exponent)) / ((numeric discount multiplier) * (punctuation discount multiplier))`

Example:

base = 10
coeff = 2
nonalpha discount: 10
no-vowel discount: 10

buckets 1, 2: 9

buckets 3, 4, 5, 6: 8

buckets 7, 8, 9, 10, 11, 12, 13, 14: 7

buckets 15, 16+:

Read-only functions

can-name-be-registered

Input: `(buff 20), (buff 48)`

Output: `(response bool int)`

Signature: `(can-name-be-registered namespace name)`

Description:

Returns true if the provided name can be registered.

can-namespace-be-registered

Input: `(buff 20)`

Output: `(response bool UnknownType)`

Signature: `(can-namespace-be-registered namespace)`

Description:

Returns true if the provided namespace is available.

can-receive-name

Input: `principal`

Output: `(response bool int)`

Signature: `(can-receive-name owner)`

Description:

Returns true if the provided name can be received. That is, if it is not currently owned, a previous lease is expired, and the name wasn't revoked.

get-name-price

Input: `(buff 20), (buff 48)`

Output: `(response uint int)`

Signature: `(get-name-price namespace name)`

Description:

Gets the price for a name.

get-namespace-price

Input: `(buff 20)`

Output: `(response uint int)`

Signature: `(get-namespace-price namespace)`

Description:

Gets the price for a namespace.

get-namespace-properties

Input: `(buff 20)`

Output: `(response (tuple (namespace (buff 20)) (properties (tuple (can-update-price-function bool) (launched-at (optional uint)) (lifetime uint) (namespace-import principal) (price-function (tuple (base uint) (buckets (list 16 uint)) (coeff uint) (no-vowel-discount uint) (nonalpha-discount uint))) (revealed-at uint)))) int)`

Signature: `(get-namespace-properties namespace)`

Description:

Get namespace properties.

is-name-lease-expired

Input: `(buff 20), (buff 48)`

Output: `(response bool int)`

Signature: `(is-name-lease-expired namespace name)`

Description:

Return true if the provided name lease is expired.

name-resolve

Input: `(buff 20), (buff 48)`

Output: `(response (tuple (lease-ending-at (optional uint)) (lease-started-at uint) (owner principal) (zonefile-hash (buff 20))) int)`

Signature: `(name-resolve namespace name)`

Description:

Get name registration details.

resolve-principal

Input: `principal`

Output: `(response (tuple (name (buff 48)) (namespace (buff 20))) (tuple (code int) (name (optional (tuple (name (buff 48)) (namespace (buff 20)))))))`

Signature: `(resolve-principal owner)`

Description:

Returns the registered name that a principal owns if there is one. A principal can only own one name at a time.

Error codes

ERR_INSUFFICIENT_FUNDS

type: `int`

value: `4001`

ERR_NAMESPACE_ALREADY_EXISTS

type: `int`

value: `1006`

ERR_NAMESPACE_ALREADY_LAUNCHED

type: `int`

value: `1014`

ERR_NAMESPACE_BLANK

type: `int`

value: `1013`

ERR_NAMESPACE_CHARSET_INVALID

type: `int`

value: `1016`

ERR_NAMESPACE_HASH_MALFORMED

type: `int`

value: `1015`

ERR_NAMESPACE_NOT_FOUND

type: `int`

value: `1005`

ERR_NAMESPACE_NOT_LAUNCHED

type: `int`

value: `1007`

ERR_NAMESPACE_OPERATION_UNAUTHORIZED

type: `int`

value: `1011`

ERR_NAMESPACE_PREORDER_ALREADY_EXISTS

type: `int`

value: `1003`

ERR_NAMESPACE_PREORDER_CLAIMABILITY_EXPIRED

type: `int`

value: `1009`

ERR_NAMESPACE_PREORDER_EXPIRED

type: `int`

value: `1002`

ERR_NAMESPACE_PREORDER_LAUNCHABILITY_EXPIRED

type: `int`

value: `1010`

ERR_NAMESPACE_PREORDER_NOT_FOUND

type: `int`

value: `1001`

ERR_NAMESPACE_PRICE_FUNCTION_INVALID

type: `int`

value: `1008`

ERR_NAMESPACE_STX_BURNT_INSUFFICIENT

type: `int`

value: `1012`

ERR_NAMESPACE_UNAVAILABLE

type: `int`

value: `1004`

ERR_NAME_ALREADY_CLAIMED

type: `int`

value: `2011`

ERR_NAME_BLANK

type: `int`

value: `2010`

ERR_NAME_CHARSET_INVALID

type: `int`

value: `2022`

ERR_NAME_CLAIMABILITY_EXPIRED

type: `int`

value: `2012`

ERR_NAME_COULD_NOT_BE_MINTED

type: `int`

value: `2020`

ERR_NAME_COULD_NOT_BE_TRANSFERRED

type: `int`

value: `2021`

ERR_NAME_EXPIRED

type: `int`

value: `2008`

ERR_NAME_GRACE_PERIOD

type: `int`

value: `2009`

ERR_NAME_HASH_MALFORMED

type: `int`

value: `2017`

ERR_NAME_NOT_FOUND

type: `int`

value: `2013`

ERR_NAME_NOT_RESOLVABLE

type: `int`

value: `2019`

ERR_NAME_OPERATION_UNAUTHORIZED

type: `int`

value: `2006`

ERR_NAME_PREORDERED_BEFORE_NAMESPACE_LAUNCH

type: `int`

value: `2018`

ERR_NAME_PREORDER_ALREADY_EXISTS

type: `int`

value: `2016`

ERR_NAME_PREORDER_EXPIRED

type: `int`

value: `2002`

ERR_NAME_PREORDER_FUNDS_INSUFFICIENT

type: `int`

value: `2003`

ERR_NAME_PREORDER_NOT_FOUND

type: `int`

value: `2001`

ERR_NAME_REVOKED

type: `int`

value: `2014`

ERR_NAME_STX_BURNT_INSUFFICIENT

type: `int`

value: `2007`

ERR_NAME_TRANSFER_FAILED

type: `int`

value: `2015`

ERR_NAME_UNAVAILABLE

type: `int`

value: `2004`

ERR_PANIC

type: `int`

value: `0`

ERR_PRINCIPAL_ALREADY_ASSOCIATED

type: `int`

value: `3001`

Multi Send

Multi send is a very simple but highly useful utility contract for executing multiple STX transfers in a single transaction.

It takes in a list of addresses and amounts and folds through them to execute a STX transfer for each one.

Mainnet contract:
https://explorer.hiro.so/txid/0x59665b756dc0fa9efb3fca9e05a28f572c9b14ca894c115fd3e7d81a563e14f8?chain=mainnet

```
;; send-many

(define-private (send-stx (recipient { to: principal, ustx: uint }))

 (stx-transfer? (get ustx recipient) tx-sender (get to recipient)))

(define-private (check-err (result (response bool uint))

                          (prior (response bool uint)))

 (match prior ok-value result

              err-value (err err-value)))

(define-public (send-many (recipients (list 200 { to: principal, ustx: uint })))

 (fold check-err

   (map send-stx recipients)

   (ok true)))
```