Clarity Vision

This directory captures the goals, architecture, and rationale for the next Clarity: a redesigned, independently specified language that compiles to Wasm. It is the first source language for a new chain that executes Wasm.

These documents are working drafts intended to seed discussion. They are expected to eventually move to a dedicated language repository.

All of these documents are published together as a single PDF and web page, rebuilt on every push to `main`. To build the PDF locally, run `make pdf`.

Documents

Document	Purpose
01-why-clarity.md	Why a decidable language on a Wasm chain: what the VM provides, what only the language can provide, the honest gaps, and why not just compile a general-purpose language
02-goals.md	Goals, non-goals, and success criteria
03-architecture.md	The Clarity toolchain, the module format and chain ABI, the guarantees the chain enforces for every module, and the compiler trust path
04-spec-plan.md	Plan for the language specification, the chain ABI it depends on, and the conformance suite
05-roadmap.md	Phased execution plan with milestones, dependencies, and risks
06-issue-tracker.md	Disposition of every open `clarity` issue against the spec draft, and candidate additions
07-chain-abi.md	Proposal for the runtime layer of the chain ABI shared by every source language: host value encoding, module format, interface, host functions, frames, fuel, and outcomes
08-value-interpretation.md	Proposal for the interpretation layer of the chain ABI: declared types and the encoding conventions compilers, wallets, and explorers rely on
spec/	The specification working draft: a minimal core language plus a standard library written in the language itself

Terminology

These terms have one meaning across every document here.

- **Contract**: a deployed program, identified by its principal. A contract has a source and a module.
- **Source**: the Clarity text a contract is written in.
- **Module**: the Wasm module a contract's source compiles to. It is what the chain validates at deploy, stores, and executes. "Module" always means compiled Wasm, never Clarity source. The standard

library is not a separate module: the compiler links the parts a contract uses into that contract's module.

- **The chain**: any implementation of the chain ABI that executes modules.
- **Chain ABI**: the contract between compilers and the chain, shared by every source language. Its **runtime layer** (07-chain-abi.md) is what the chain enforces; its **interpretation layer** (08-value-interpretation.md) is the conventions compilers, wallets, and explorers agree on.
- **Host function**: a function the chain provides to modules through the chain ABI. Everything a module does beyond pure computation goes through one.
- **Fuel**: the chain's unit of execution cost, consumed by a module's Wasm instructions and charged by host functions.
- **Build inputs**: everything a compilation depends on: the source, the compiler version, the deployed interfaces of the contracts the source refers to, and the deploying principal (03-architecture.md section 2).
- **Provenance section**: the custom section of a module that records the compiler version, the source hash, and the hashes of the callee interfaces it was compiled against, so that a module names its own build inputs.
- **Verified triple**: a (source, compiler version, module) triple for which rebuilding the source with that compiler version, and the other build inputs the module's provenance section records, produces the module byte for byte. Analyses, cost bounds, and proofs are stated for a verified triple. "Triple" means only this.
- **Legacy**: Clarity 1–6, which run on the existing interpreter.
- **vNEXT**: the working name of the new Clarity language.

One-paragraph summary

The new chain executes Wasm modules, meters them in fuel, and enforces a set of runtime guarantees for every module regardless of source language (03-architecture.md section 5). Wasm cannot give the guarantees that made Clarity worth having, because Wasm is Turing-complete: fuel makes execution *stop*, but it does not make the worst case *known in advance*. Clarity adds that. Every Clarity program terminates by construction, and every loop and value is bounded by its type, so a contract's worst-case cost is computable before it runs, whole-program analysis is complete, and safety proofs are tractable. This effort (1) specifies Clarity spec-first as a small core language plus a standard library written in it, (2) builds a deterministic compiler to Wasm held to the spec by a conformance suite, (3) defines, with the chain, an ABI that every source language shares, and (4) builds the toolchain that turns decidability into products: source verification, a static cost analyzer, and a property checker.

Clarity 1–6 contracts on the existing chain keep running on the interpreter, and the interpreter and the Wasm engine can call each other. That arrangement is outside the scope of these documents.

Why Clarity on a Wasm Chain

The new chain executes Wasm, and any language that compiles to Wasm can target it. So the question is no longer "why not switch VMs". It is "why a dedicated contract language, instead of Rust or another general-purpose language compiled to the same Wasm". This document makes the case, and states honestly what the Wasm-first design costs.

1. What the VM gives and what only the language can give

Wasm gives the chain a mature, sandboxed, portable execution engine, fuel metering, good performance, and a path to more than one source language. The chain adds runtime guarantees on top that hold for every module (section 1.4).

Wasm cannot give **decidability**. A Wasm module can loop without bound; fuel guarantees that execution stops, not that anyone can know beforehand how much it will cost or what it can do. Those properties have to come from the source language.

1.1 Decidability

Clarity is deliberately **not Turing-complete**:

- There is no recursion (direct or mutual) and no unbounded loop construct. Iteration exists only as `map`, `filter`, `fold-left`, and `fold-right` over sequences whose maximum length is fixed by their type.
- Every type has a statically known maximum size. Lists, buffers, and strings all carry length bounds; there are no unbounded data structures.
- Call depth is bounded and contracts cannot be re-entered.

Consequence: **every Clarity program provably terminates**, and every execution path is statically enumerable in principle. The halting problem, which makes complete static analysis impossible for general Wasm, does not apply to a module compiled from Clarity.

1.2 What decidability enables

These capabilities justify the language, and this effort commits to building them:

- **Sound static cost bounds.** Because all loops and data are bounded by types, the worst-case fuel of a function's own code is computable before deployment without simulation, as a function of its arguments' sizes (a list's actual length, not only its type bound, so the bound for a one-element list is not the bound for a thousand). A wallet can display a true maximum fee; a block producer can pack against guaranteed bounds; there is no "estimate by simulating and hope state doesn't change" pattern. The bound composes across `contract-call?` only as far as the callees are themselves Clarity with verified source. A call through a trait, or into a module written in another language, can

consume the whole remaining budget, and the analyzer reports it as such (03-architecture.md section 5).
- **Complete static analysis.** Whole-program analyses (taint tracking, asset flow, authorization checks) terminate and can be precise, not heuristic. An analyzer can enumerate every path by which a contract's assets can move.
- **Tractable safety proofs.** All functions are total, state access is explicit, and there is no reentrancy, so verification conditions stay inside decidable or well-behaved SMT fragments. Proving invariants such as supply conservation, access control, and "no path locks funds" is an engineering task, not a research program.

1.3 The deployed artifact is Wasm

The chain stores and executes Wasm. It never sees Clarity source. This is the price of a chain that more than one language can target, and it changes one of legacy Clarity's claims: legacy Clarity had no compiler between the author and the chain, and new Clarity does.

A compiler in the trust path means a compiler bug can make the deployed code differ from the source everyone audited. The plan pays that down directly:

- **A specification and conformance suite** define what a correct compiler produces, independent of any one implementation.
- **Deterministic, reproducible builds.** The same source and compiler version produce a byte-identical module, and each module records which compiler built it and the hash of its source. Anyone can rebuild and compare.
- **Source verification as a product.** Analyses, cost bounds, and proofs are computed on source and attach to a verified (source, compiler version, module) triple. They apply to the deployed code exactly when that triple checks.
- **A compiler that is small by design.** The core language is about a dozen forms and about seventy primitives with no bespoke evaluation rules, and the standard library is compiled like user code. Validating the translation of a first-order, loop-bounded language is far more tractable than for a general-purpose one, which keeps verified compilation a realistic long-term goal.

1.4 Safety by construction

Two layers enforce safety, and it matters which one enforces what.

The **chain** enforces these for every module, whatever its source language, so they hold even when a Clarity contract calls one written in something else (the normative table is 03-architecture.md section 5):

- **No reentrancy**, the largest exploit class in EVM history, and bounded call depth.
- **Read-only frames**: a write attempted inside a read-only call is rejected at runtime.
- **Rollback on error**: a public function that returns `err` has its state changes discarded.
- **Post-conditions and asset allowances**: the sender bounds what a transaction may do to their assets, and `as-contract?` bounds what a contract may do with its own, regardless of the called code.

The **language** adds, for Clarity contracts:

- **No undefined behavior**: arithmetic overflow aborts deterministically; there are no null references.
- **Errors must be handled**: ignoring a `response` is a type error.
- **Well-typed boundaries**: a Clarity contract validates every value it receives, its arguments on entry and the result of every call it makes, against its declared types, so it can rely on its types whatever language the other side is written in. The chain passes values through as bytes and checks none of them.
- **Decidability**, with everything in 1.2.

1.5 Bitcoin-native

Clarity contracts read Bitcoin chain state directly (burn block info, Bitcoin transaction outputs, Merkle inclusion proofs). The chain ABI exposes that state to every language; Clarity makes it first-class in the type system and standard library.

2. Honest ledger

Gap	Reality	Plan item
No specification yet	The design exists as a working draft (spec/)	Spec-first design (04-spec-plan.md)
Compiler in the trust path	Deployed code is only as trustworthy as the compiler that produced it	Conformance suite, reproducible builds, source verification (03-architecture.md)
Guarantees are per-language	A module from another language has the chain's runtime guarantees but not Clarity's static ones	Verified-source labeling in wallets and explorers, so users know which they are looking at
Properties unexploited	No static cost analyzer or prover exists yet	Flagship toolchain deliverables (05-roadmap.md)
Not source-compatible with Clarity 1–6	Builders must port contracts	Disposition table for every legacy construct (spec/04-standard-library.md)
Mixed ecosystem for years	Tokens, PoX, and sBTC are legacy contracts; a new contract that calls them crosses a type mapping between legacy analysis types and the chain ABI	Named as a chain-owned deliverable with the mismatches listed (03-architecture.md section 7)
Cost bounds stop at foreign code	A call through a trait or into a non-Clarity module has no static bound	Bounds are stated per verified-Clarity call graph, and the analyzer marks the boundary (05-roadmap.md risk 7)
Developer experience	Tooling, docs, and error messages behind mainstream ecosystems	A stable spec and toolchain crates as the foundation for tools

Each gap is an engineering problem. None is a fundamental limitation. The reverse does not hold for general-purpose languages: their deficits relative to Clarity are fundamental.

3. Why not a general-purpose language on the same Wasm

Option	What you gain	What you permanently lose
Rust (or C, Go, AssemblyScript) to Wasm	Mature toolchains, large developer pool, raw expressiveness	Decidability; static cost bounds; complete analysis; tractable proofs. Contract safety reduces to library discipline, and the language's compiler and runtime become a large trust surface
Solidity or an EVM-style language to Wasm	Familiarity for EVM developers	All of the above, plus a design built around a reentrancy-permitting call model
Move-style resource language	Linear resource types (genuinely good)	Still Turing-complete: termination, exact costs, and complete analysis remain impossible

These are not either/or. The chain is multi-language by design, and nothing stops other languages from targeting it. Clarity's role is to be the language in which the properties above are guaranteed, and the one that consensus-critical and high-value contracts are written in.

The pitch, in one sentence:

> **The contract language where the worst case is a theorem, not a hope.** Every contract's maximum cost is known before it runs, its safety properties are machine-checkable, and anyone can verify that the code on chain was built from the code they read.

4. The commitment this implies

A decidable language only earns its place if we ship what decidability makes possible. Concretely (see 05-roadmap.md):

1. A **specification**, designed spec-first.
2. A **deterministic compiler** to Wasm, pinned to the spec by a **conformance suite**.
3. **Reproducible builds and source verification**, so analyses of source apply to deployed code.
4. A **static cost analyzer** producing sound worst-case fuel bounds per function.
5. **Proof-friendly semantics** and a verification toolchain path.

Goals and Non-Goals

Goals

G1 — Clarity is a specified language compiled to Wasm

- Clarity is designed **spec-first**: grammar, type system, static and dynamic semantics, compilation to Wasm, and the standard library, each normative and each backed by conformance tests.
- The reference compiler implements the spec and is validated against it, never the reverse. Any other Clarity compiler is held to the same suite.
- The toolchain lives in its own repository, published as crates with their own CI, release cadence, and versioning policy.

G2 — One chain ABI, shared by every source language

- The chain defines a single ABI, in two layers: a runtime layer the chain implements (the module format, interface, host functions, call semantics, and fuel schedule; 07-chain-abi.md), and an interpretation layer every compiler and tool shares (declared types and value encoding; 08-value-interpretation.md). Clarity is its first consumer, not its owner.
- Nothing on chain is Clarity-specific. The chain never parses Clarity, and a Clarity contract has no powers on chain that a module from another language lacks.

G3 — Runtime guarantees for every module

The chain enforces a fixed set of guarantees for every module, whatever language produced it: fuel limits, no reentrancy, read-only frames, rollback on `err`, and asset allowances. The chain does not check values against declared types; each module validates what it receives, and the Clarity compiler does so for every Clarity contract. The normative list, with which layer enforces each, is 03-architecture.md section 5 and is not repeated here.

G4 — Cash in decidability

The properties unique to Clarity become shipped products:

- **Static cost analyzer**: sound worst-case fuel bounds (plus host charges) per public function, computed before deployment. Bounds are functions of argument sizes, not only of argument types, so they are tight enough to display as a fee. They cover the contract's own code and, transitively, callees that are verified Clarity; at a trait call or a call into another language the analyzer reports the remaining budget as the bound. The foundation for true max-fee display, simulation-free fee estimation, and block packing.
- **Proof-friendly semantics**: written so that verification conditions land in decidable or well-supported SMT fragments; a contract-property checker (supply conservation, authorization invariants, no fund lockup) is a first-class roadmap item.

G5 — A trustworthy compiler path

- **Deterministic builds**: the same build inputs produce a byte-identical module on every platform. The inputs are the source, the compiler version, the `abi.interface` and `abi.types` of every contract the source calls by literal or whose traits it uses, and the deploying principal when deployer-relative names are resolved at compile time (03-architecture.md section 2).
- **Provenance**: each module records its compiler version, its source hash, and the hashes of both sections of every callee it was compiled against.
- **Source verification**: anyone can rebuild a deployed module from its claimed source and the recorded inputs and compare. Analyses and proofs attach to a verified (source, compiler version, module) triple.

Non-Goals

- **N1 — Source compatibility with Clarity 1–6.** Rejected. The new language has no compatibility obligation in syntax or semantics. The disposition table in spec/04-standard-library.md is the porting guide. Legacy contracts keep running on the interpreter, with interop handled by the chain outside these documents.
- **N2 — Turing completeness.** Decidability is the product. Any proposal that introduces unbounded loops, recursion, or unbounded data is out of scope by definition.
- **N3 — On-chain compilation.** The chain stores and executes Wasm. It does not parse, type-check, or compile Clarity.
- **N4 — Being the only language.** The chain is multi-language. Clarity competes on guarantees, not on privileges.
- **N5 — A general-purpose language.** Clarity stays a smart contract language: no file I/O, FFI, or off-chain execution.

Success criteria

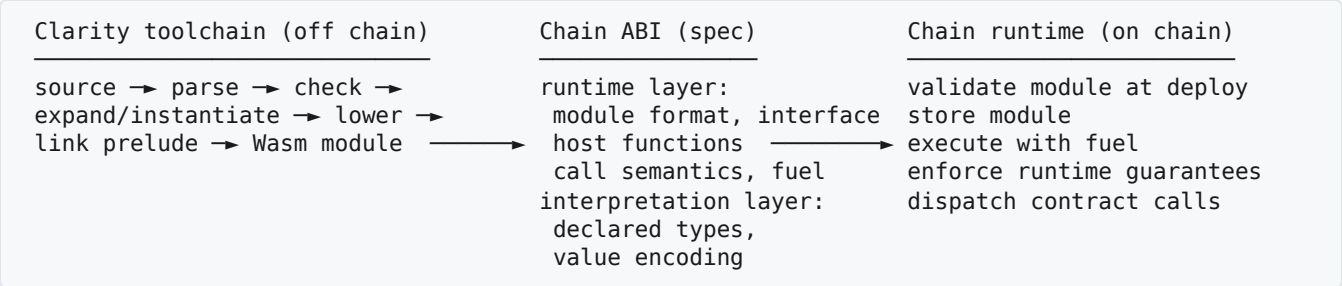
1. A complete, versioned spec for the language, with a conformance suite.
2. A reference compiler that passes the suite and produces byte-identical modules across platforms.
3. A chain ABI specified jointly with the chain team, implemented by the chain, and exercised by Clarity.
4. A static cost analyzer that emits a sound worst-case fuel bound for every public function of any well-typed contract.
5. A source-verification service or tool that rebuilds and verifies any deployed Clarity module from its source.

Architecture: Clarity on a Wasm Chain

This document describes how Clarity source becomes code running on the new chain, what the chain guarantees for every module, and where the compiler sits in the trust path.

1. Shape

There are three parts, and only the middle one is shared between languages:



- The **toolchain** is Clarity-specific and runs off chain. The chain never sees Clarity source.
- The **chain ABI** is the contract between any compiler and the chain. It is specified jointly with the chain and shared by every source language, in two layers: the **runtime layer** (07-chain-abi.md), which the chain implements, and the **interpretation layer** (08-value-interpretation.md), the declared types and value encoding that compilers, wallets, and explorers use. The chain depends only on the runtime layer, except where a legacy contract calls a Wasm one or the reverse (section 7).
- The **chain runtime** executes Wasm and enforces the guarantees in section 5 for every module.

2. The Clarity toolchain

Illustrative crate layout; names are placeholders.

```
clarity-syntax      lexer, parser, source spans
clarity-types       type checker, overload resolution, instantiation (spec ch. 2)
clarity-std         the prelude source (spec/std/prelude.clar) and its tests
clarity-compiler    macro expansion, lowering to Wasm, prelude linking,
                    interface and types sections, provenance section
clarity-analyzer    static cost bounds, asset-flow and authorization analyses
clarity-verify      rebuilds a module from source and compares it
```

Rules that bind every Clarity compiler:

- **Deterministic output.** The same build inputs produce a byte-identical module on every platform. No timestamps, paths, hash-map iteration order, or host-dependent floating point in the output.
- **Build inputs are explicit.** Compilation is not a function of source alone. Type-checking a `contract-call?` with a literal target reads the callee's declared types, its `abi.types` section (spec 2.5.8), and the singleton-contract subtyping rule reads it too (spec 2.3.2). A trait used in the source is read from

the `abi.types` of the contract that defines it. The inputs are therefore the source, the compiler version, the `abi.interface` and `abi.types` of every contract referenced by literal or defining a trait the source uses, and the deploying principal, because deployer-relative `.name` references are resolved at compile time. A module is therefore specific to its deployer, and the chain has no relocation step and no Clarity-specific name resolution. Deployed sections never change, so recording which ones were used is enough to reproduce the build.

- **Provenance.** Each module carries a custom section with the compiler identifier and version, the language version the source declares (spec 2.4.0), a hash of the source, and the hashes of each callee's `abi.interface` and `abi.types` the build read, keyed by contract principal. `clarity-verify` fetches those sections from the chain and checks the hashes before rebuilding.
- **The prelude is linked into each module.** Only the library instances a contract uses are included, compiled like user code. There is no chain-provided library module, so the chain carries no Clarity code. Consequence: a faster library implementation ships in a new compiler release. Contracts already deployed keep the code they were deployed with, and no chain upgrade is needed.
- **Hashes are host functions.** `sha256`, `sha512`, `sha512/256`, `keccak256`, and `ripemd160` are host functions, like the signature schemes (section 4): one audited implementation at a native-speed price, which matters for the Merkle and SPV proofs central to sBTC, shared by every source language (07-chain-abi.md section 12, item 1).
- **A new language version is a compiler release.** It needs the chain only when it depends on a new host function or ABI feature.

3. The module format

A deployed contract is one Wasm module (07-chain-abi.md section 3):

- **Exports**: one function per public and read-only function, each of Wasm type `[] -> []`, and an initializer the chain runs once at deploy (Clarity's top-level expressions and `define-constant` / `define-data-var` initializers).
- **Imports**: host functions only (section 4). A module that imports anything else is rejected.
- **`abi.interface`** (a custom section, validated by the chain): each exported function's name and kind (public or read-only), and the contract's token and event names. This is what the chain dispatches against, and all it reads.
- **`abi.types`** (a custom section the chain does not read, 08-value-interpretation.md section 5): each function's parameter and result types, each NFT's identifier type, each event's data type, the traits the contract defines and claims to implement, and its storage layout. This is what other contracts compile against and what wallets and explorers display values with. Its type grammar includes an empty type (`never` in Clarity, spec 2.3.4) so that a function can declare that it never returns `err`.
- **Provenance** (section 2).

At deploy the chain checks that the module is valid Wasm within the profile and size limits, imports only host functions, and exports exactly the functions its `abi.interface` names. It then runs the initializer. It checks no types and no trait claims.

Values across calls

Arguments and results cross a contract call as bytes the chain does not interpret (07-chain-abi.md section 5.4). By convention they are the canonical encoding (formalizing SIP-005) of values of the declared types (08-value-interpretation.md section 6), which is what wallets encode and clear-sign and explorers decode.

The chain does not validate them, so each module validates what it receives. A Clarity contract decodes its arguments against its declared parameter types on entry, and the result of every call against the declared result type, and aborts on anything that does not decode (spec 2.5.8). A Clarity contract can therefore rely on its own types whoever calls it, because its own compiled code checks them, not the chain. Traits are checked at compile time; at runtime the chain guarantees only that a called function exists and, when the caller asks, that it is read-only.

4. The host interface

Everything a module can do beyond pure computation goes through host functions. The Clarity primitives in spec/03-primitives.md that pass the layering rule's host-access test map onto them.

Group	Host functions
Storage	one key-value store per contract, with raw-byte keys and values; Clarity lays its data variables and maps out on it (spec 3.7)
Assets	fungible and non-fungible token mint/burn/transfer/balance/owner/supply, STX transfer/burn/account; every movement recorded in the transaction's asset map
Context	the fields of `get-current-context` (spec 3.9): sender, caller, sponsor, contract, block heights and time, chain id, network flags, liquid supply
Chain data	Stacks block, tenure, and burn block lookups, one function per field. All total: a height the host cannot serve is reported through the result, never a failure
Calls	contract-call by principal and function name; arguments and results pass through as bytes, validated by the modules (section 3)
Frames	nested state frames: open, commit, discard. Used by `as-contract?` and `restrict-assets?` (with context switch and allowance checks) and by a module's internal calls to its own public functions, which roll back on `err` (spec 2.7.3)
Events	contract events, named in the interface, with data typed in `abi.types`; asset events; debug output (`log`), which the chain charges for and discards
Cryptography	hashes (SHA-256, SHA-512, SHA-512/256, Keccak-256, RIPEMD-160) and signature schemes (secp256k1 recover/verify/decompress, secp256r1, ed25519)

The allowance comparison for `as-contract?` and `restrict-assets?` is host code, not compiled code over a host-provided movement log, so the guarantee holds for every module whatever its source language.

Cost has one dimension, fuel. Wasm execution is metered by the engine, each host function has a fuel charge, and storage reads and writes are charged in fuel at a worst-case price; there are no separate read and write dimensions. A transaction states a fuel limit and a rate (uSTX per million fuel) and pays for the fuel

it uses, so a miner can order transactions by rate and pack a block against one fuel limit. The fuel schedule and charge table are part of the chain ABI (07-chain-abi.md section 9). They may change at epoch boundaries, and a change applies to every deployed module: a module's cost depends only on its code and the schedule, not on its source. Cost bounds are therefore stated per schedule version, and the static cost analyzer's bound is what a wallet proposes as the transaction's fuel limit.

5. Where each guarantee lives

Guarantee	Enforced by	Holds for
Execution stops within the fuel budget	Chain (fuel metering)	Every module
No reentrancy; bounded call depth	Chain (call dispatch)	Every module
No writes inside a read-only call	Chain (host functions reject writes in a read-only frame)	Every module
A public call that returns `err` has its state changes discarded	Chain (frame commit/discard by the result flag)	Every module
Post-conditions; `as-contract?` / `restrict-assets?` allowances	Chain (asset map, frames)	Every module
Termination by construction; worst-case fuel of the module's own code computable in advance	Clarity language, preserved by the compiler	Clarity modules
Worst-case fuel of a whole call graph computable in advance	Static cost analyzer over verified source	Call graphs whose every callee is verified Clarity, reached by literal target. A trait call or a call into another language is bounded only by the remaining budget, and the analyzer reports it as such
Complete static analysis; tractable proofs	Clarity language	Clarity modules with verified source
No undefined behavior; responses must be handled	Clarity type checker and compiler	Clarity modules
Values a contract receives from another match its declared types	Clarity compiler (validation on entry and after each call, section 3)	Clarity modules, whoever calls them

The first five rows are why a Clarity contract can call a module from another language safely. The last five are what Clarity adds over any other language on the chain.

6. The trust path

The compiler is in the trust path: a compiler bug can make the deployed code differ from the audited source. The defenses:

1. **Spec and conformance suite.** A correct compiler is defined by the spec, not by an implementation. Every compiler release passes the suite.
2. **Differential testing and fuzzing against a direct evaluator.** The direct evaluator is an interpreter that runs Clarity source by applying the spec's dynamic-semantics rules one at a time, with no lowering, no Wasm, and no optimization. It is a much smaller program than the compiler and a different kind of program, so the two are unlikely to share a bug. Differential testing runs the same contract and the same calls through both and compares every observable result: return values, events, asset movements, and aborts. Fuzzing supplies those contracts and calls by generating them, well-typed, in volume. Any disagreement is a bug in the compiler, in the evaluator, or in the spec, and the spec adjudicates. The evaluator is only a test oracle: it never runs on chain, is never a fallback execution path, and has no authority over the compiler. A second production implementation would make "whatever the implementation does" the de facto spec again, which is the legacy situation this design leaves behind.
3. **Reproducible builds and source verification.** Source is published in an off-chain registry keyed by the source hash in the module's provenance section; the chain stores only the module. `clarity-verify` rebuilds a module from its claimed source and compiler version and compares bytes. Wallets and explorers show whether a module's source is verified.
4. **Analyses attach to the triple.** A cost bound or proof is stated for a (source, compiler version, module) triple, identified by the source hash, compiler version, and module hash, and is meaningful for a deployed module only when that triple verifies (rebuilds byte for byte). A cost bound also names the fuel-schedule version it was computed for (section 4). Bounds need both halves of the triple: loop and size bounds come from the verified source, because only Clarity source guarantees bounded loops, and fuel comes from the module, because that is what the chain meters. The chain sees only the module, so the ABI carries no bound certificates.

7. Legacy contracts

Clarity 1–6 contracts keep running on the interpreter, and the interpreter and the Wasm engine can call each other. How that works is a chain concern outside the scope of these documents.

It is out of scope, not unimportant. Tokens, PoX, and sBTC are legacy contracts, and a new contract that cannot call them has no ecosystem. The chain owns the mapping from a legacy contract's analysis to an `abi.interface` and an `abi.types` section, and it converts values between the interpreter and Wasm modules by the interpretation layer's encoding, the one place the chain interprets contract values (07-chain-abi.md section 13, question 4). The mapping is not one to one. Known mismatches the chain must decide, with input from the language:

- **Tuples.** Legacy admission lets a tuple with extra fields pass where a smaller one is declared, so a legacy function can return a value with fields its interface does not name. vNEXT requires exact field

sets (spec 2.3.2). The bridge must either strip extra fields on the way out of legacy code or reject them.

- **List union types and trait values.** Legacy lists may carry a union of trait types and contract literals, and legacy trait values carry static type information at runtime. The ABI value model has neither; a trait value is a principal (spec 2.7.1).
- **Read-only marks.** A vNEXT read-only function may call a legacy function only if the legacy interface marks it read-only, so the mapping must carry that mark.
- **Error-code and field-name conventions** (`error_code`, `stx-transfer-memo?`) are the legacy contract's and are not translated.

Specification Plan

Scope decision: two documents, one owner each

- **The Clarity specification** defines the language: syntax, types, semantics, the standard library, and what a correct compiler to Wasm produces. It is owned by the language project. Drafts live in spec/.
- **The chain ABI** defines what any module can rely on, in two layers. The runtime layer (07-chain-abi.md) is what the chain implements: the module format and interface, host functions and their charges, call semantics, runtime guarantees, and the fuel schedule. The interpretation layer (08-value-interpretation.md) is what compilers, wallets, and explorers rely on: declared types and value encoding. It is owned jointly with the chain and shared by every source language. The Clarity spec refers to it and never restates it.

Clarity 1–6 are out of scope. They remain defined by their implementation on the legacy track. The new language owes them nothing in syntax or semantics, and the disposition table in spec/04-standard-library.md serves as the porting guide.

The language is designed **spec-first**: the spec is written and reviewed before the reference compiler, and the compiler is validated against it, never the other way around.

Structure decision: minimal core plus a standard library

The spec is organized as **a small core language and a standard library written in that language**.

- The **core** is the syntax, the type system, about a dozen special forms (binding, `match`, `return`, definitions, calls, context scoping), and about seventy primitive operations. A construct is core if it needs the host, cannot be expressed in the language, is a cryptographic standard, or is a base operation on a type's representation (spec/README.md states the rule). Cost is not a test.
- The **standard library** is everything else: control flow (`if`, `unwrap!`, `try!`, `asserts!`, `default-to`), predicates, derived arithmetic (`max`, `pow`, `sqrti`), sequence helpers (`append`, `index-of?`, `replace-at?`), integer/string/byte conversions, `verify-merkle-proof`, `map-insert?`, and more, about thirty-five definitions. It is ordinary Clarity source (spec/std/prelude.clar), versioned and frozen with the language, compiled and linked into each module like user code, and it has no powers a contract author lacks.
- **Cost is Wasm fuel, not a language concern.** Contracts are metered by the fuel their Wasm consumes, plus the chain's charges for host functions. The language has no cost table, so the "cost table disagrees with behavior" bug class cannot occur. The prelude's source defines results; a later compiler may link a faster implementation, which changes nothing for contracts already deployed.

Spec structure

A versioned document set, one complete spec per language version (the ECMAScript/Rust-edition model; no delta chains to reconstruct):

1. **Design principles**: decidability as an invariant; every construct must preserve statically bounded execution. This chapter is the rejection criterion for future proposals.
2. **Lexical structure and grammar**: complete EBNF for the surface syntax, including the honest edge cases (string escapes, integer literal bounds, principal literals, identifier rules).
3. **Type system**: types, subtyping, trait typing (fully static and decidable), inference rules written formally enough to derive a checker from. Every rule numbered and conformance-tested.
4. **Static semantics**: the analysis passes the language requires: read-only checking, trait conformance, definition ordering. Specified as judgments, not as descriptions of a checker.
5. **Dynamic semantics**: evaluation order, error, abort, and rollback semantics, context scoping (`as-contract?`, `restrict-assets?`), storage visibility, event emission. Written in a proof-friendly style (small-step or big-step operational semantics) so verification tools can consume it. This is the semantics every compiler must preserve.
6. **Compilation to Wasm**: the module layout Clarity uses within the chain ABI, the lowering of each core form and primitive, prelude linking, the `abi.interface`, `abi.types`, and provenance sections, the validation of arguments and call results and the abort codes of runtime errors, and the determinism rules. It also carries the argument that loop bounds guaranteed by the source types survive compilation, which is what makes "worst-case fuel of any function is statically boundable" a theorem. The static cost analyzer is derived from that argument.
7. **Primitives and standard library**: every core primitive with its signature, semantics, and error cases (spec/03-primitives.md); the standard library, specified by its source (spec/std/prelude.clar), plus the Clarity 1–6 disposition table (spec/04-standard-library.md).

The chain ABI is specified separately, in two documents: the runtime layer (07-chain-abi.md: module format and interface, host functions and their charges, call dispatch, runtime guarantees, and the fuel schedule) and the interpretation layer (08-value-interpretation.md: the type grammar, canonical encoding, and the `abi.types` section).

The conformance suite

The suite is what makes the spec real, and it is the highest-leverage artifact of the effort. Without it the spec is a description that implementations resemble to varying degrees; with it the spec is a contract that an implementation either meets or does not.

What a vector is

A **conformance vector** is one test case written as data rather than as code: an input, the outcome the spec requires for that input, and the identifiers of the normative rules it exercises. Because a vector is data, every implementation runs the same vectors through its own runner, so the suite never depends on any one implementation's language or test framework. When a vector fails, it names the rule that was broken.

An illustrative language vector (the concrete format is Phase 0 work in 05-roadmap.md, and rule identifiers are not yet assigned):

```
id:     lang/arith/add-uint-overflow
rules:  03 §3.1 (+ overflows), 02 §2.7.2 (runtime errors abort)
source: (define-public (f)
          (ok (+ u340282366920938463463374607431768211455 u1)))
call:   f
expect: abort, class arithmetic-overflow
```

Four kinds of vector

Kind	Input	Expected outcome	Who must pass
Language	Contract source, plus calls	A compile error of a specified class, or for each call: return value, events, asset movements, and whether it aborts	Every Clarity compiler
ABI	A module, plus chain actions	Deploy validation, rollback, reentrancy and read-only rejection, host function results and charges	Every chain implementation, independent of source language
Encoding	A declared type and bytes	Whether the bytes decode canonically against the type, and to which value; `serialized-size` of the type	Every compiler, every tool that decodes values, and the chain's legacy bridge
Golden build	Source and a compiler version	The exact module bytes	That compiler release, on every supported platform

Language, ABI, and encoding vectors are the suite proper: they pin behavior the specification requires, and they are written with the spec and versioned with it. Golden build vectors are different in kind. They pin nothing about the language; they pin a particular compiler release, so that reproducibility (section 6 of 03-architecture.md) is testable rather than asserted. Each compiler release publishes its own.

What a vector deliberately does not pin

The suite checks what the spec requires and nothing more, so that implementations keep room to improve:

- **Fuel.** A language vector never states how much fuel a call consumes. A better compiler is allowed to produce cheaper code, and a vector that pinned fuel would forbid it. Charges for host functions are pinned by ABI vectors, because the ABI's fuel schedule is normative for the chain.
- **Module bytes.** A language vector never states what Wasm a compiler emits, only how the result behaves. Byte-exactness is the job of golden build vectors, and only for one release at a time.
- **Error text.** A compile-error vector names the class of error (an arity mismatch, an undefined name, a read-only violation), not a message. Messages are a tool's own concern.

How rules and vectors bind each other

Principle P8 (spec/01-principles.md) states the contract: every normative rule cites the vectors that exercise it, and no rule is frozen without vectors. The runner enforces this both ways. A rule that cites no vector is unfrozen and shows up in a coverage report. A vector that cites no rule is testing something the spec does not say, which is a spec bug to fix before the vector is accepted.

Which side is the oracle depends on where the rule lives:

- For the core language, the prose rule is normative. If a compiler and a vector disagree, one of them has a bug, and the disagreement is adjudicated explicitly and recorded (P8).
- For the standard library, the prelude source is normative (spec/04-standard-library.md). Library vectors are derived from the source, and if a vector and the source disagree, the vector is wrong. The same vectors are what hold a hand-written or better-compiled replacement, linked by a later compiler in place of the source, to identical results, events, and errors.
- For the chain, the runtime layer is normative, and ABI vectors are its suite. For values, the interpretation layer is normative, and encoding vectors are its suite.

Where the suite sits among the other defenses

The suite is the floor, not the ceiling. It is a fixed set of cases, so it catches regressions and divergence on behavior someone thought to write down. Differential testing against the direct evaluator and fuzzing (section 6 of 03-architecture.md) search above that floor for behavior nobody wrote down. Every bug they find becomes a vector, so the floor only rises.

Language design opportunities

Candidates for the RFC process, listed to scope ambition, not to commit:

- Consistent naming and calling conventions across the standard library (the `unwrap!` / `unwrap-panic` / `try!` family, `?` / `!` suffix discipline).
- A trait system that is fully static, decidable, and pleasant, checked entirely at compile time against declared signatures that every language shares (08-value-interpretation.md section 5.2).
- Cost transparency in the language: contracts and functions can declare fuel bounds that the analyzer verifies; callers can rely on them.
- Richer (still bounded) iteration and data manipulation, so common patterns stop requiring fold contortions. Every addition passes the chapter-1 decidability test.
- First-class annotations for verification (invariants, pre/postconditions) consumed by the proof toolchain.
- Remove the footguns of Clarity 1–6.

Process and governance

- Spec development runs as an open RFC process in the language repo. Each chapter and each language feature is an RFC with a decidability analysis as a required section.
- A new language version is a spec release plus a compiler release. It needs the chain only when it depends on a new host function or ABI feature, and those changes follow the chain's own governance.
- The spec versions with the language: each version gets its own complete spec. The chain ABI versions separately and is additive once frozen.

Findings from drafting the prelude

Writing the prelude is the test of the core design. Findings so far:

- An earlier draft returned the conversion, search, and Merkle functions to the core because their library versions cost more than a native under a per-operation cost table. With fuel as the metric and optimization done by a compiler linking faster code, that argument is gone, and they are library again. The core is smaller and there is no per-construct cost table.
- Template checking of generic bodies carries more weight than planned. It lets `index-of?` and the string parsers be one definition across sequence kinds instead of four, at the price of errors reported inside the prelude (spec 2.6.2).
- Per-kind constants (digit tables, the minus sign) need small overloaded helpers, because a generic body cannot write a literal of an unknown kind.
- `replace-at?` exposed a gap in the size rules: an intermediate `concat` has a larger bound than any value it can hold. Intermediate expressions are checked against a separate, larger limit (spec 2.3.1).
- Rest parameters are strict: the extra arguments are built into a list before the body runs. That is right for `max`, `+`, and `concat`, but it means n-ary short-circuiting cannot be written in the language, so `and` and `or` moved from the library into the core (spec 2.5.3).
- `never` as the type of `(return ...)` and `(panic)` makes every early-exit form typeable with ordinary lub rules, with no special cases.
- Macros cannot be passed to `filter`, which forces predicates to be private functions (spec 2.6.3).
- Folds whose initial value has a smaller type than the step's result (`std-reverse-16` starts from `0x`, `std-render-decimal` from `""`, `index-of?` from `none`) rely on instantiation finding the accumulator type by iteration (spec 2.6.2). Each closes in two steps; a fold that needs a wider start states it with `as` (2.5.10).
- Overloading and static reference kinds are each justified by a small group of prelude functions (`unwrap!`/`try!` and the integer families; `map-insert?`).

Roadmap

Phased plan for the new Clarity on the Wasm chain. Phases overlap where the dependency summary allows. The legacy track (running Clarity 1–6 on the interpreter) proceeds separately and is not covered here.

Phase 0 — Foundations

- Stand up the language repository and RFC process.
- Carry the working draft in spec/ into RFCs: principles, core language, primitives, standard library.
- Open the chain ABI draft jointly with the chain team: value model, module format, host function list, call semantics.
- Define the conformance-vector formats (language, ABI, encoding, golden build) and build the vector runner early.

Phase 1 — Spec and ABI

- Settle the open questions in each spec chapter; freeze the core forms and primitive list.
- Write chapter 6 (compilation to Wasm) alongside the ABI, so the lowering and the host interface are designed against each other.
- Write language vectors as each chapter lands, not after.
- Reach ABI v1: enough to deploy and call a module, store data, move assets, and call another contract.

Phase 2 — Reference compiler

- Front end: parser, type checker, overload resolution, instantiation, macro expansion.
- Back end: lowering to Wasm, prelude linking, the `abi.interface`, `abi.types`, and provenance sections, and argument and result validation.
- Determinism from the first commit: golden build vectors in CI on every supported platform.
- A direct evaluator: an interpreter that runs Clarity source by applying the spec's semantic rules one at a time, with no Wasm in the path. It exists only as a test oracle. Differential testing runs the same contracts and calls through the evaluator and through the compiler and compares the results, and fuzzing generates those contracts and calls in volume (03-architecture.md section 6). The evaluator never runs on chain.
- Exit criterion: the compiler passes the language vectors.

Phase 3 — Chain integration

- Chain devnet executing Clarity-compiled modules through ABI v1.

- Host functions, fuel metering, and every runtime guarantee in 03-architecture.md section 5, each covered by ABI vectors.

Phase 4 — Trust path

- `clarity-verify`: rebuild a module from source and compiler version and compare bytes.
- A source registry keyed by the provenance hash, and verified-source display in at least one wallet and explorer.
- Compiler hardening: sustained fuzzing, differential testing against the oracle, external audit.

Phase 5 — Cash in decidability

Flagship deliverables, in value order:

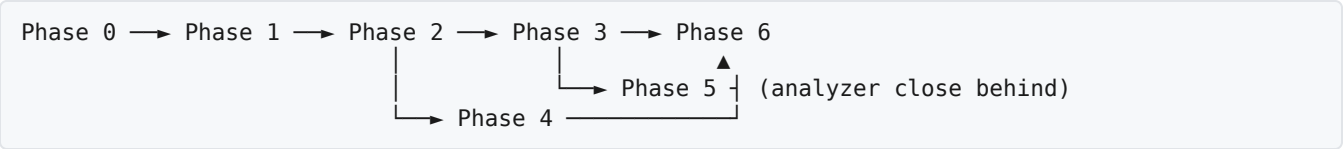
1. **Static cost analyzer**: sound worst-case fuel bounds per public function, from source, the verified module, and the chain's host-charge table. Surfaces: pre-deploy reports, wallet max-fee display, fee estimation without simulation, block packing.
2. **Contract property checker**: SMT-backed verification of standard invariants (supply conservation, authorization, no fund lockup) using the spec's verification annotations; a path toward machine-checked contracts as a normal deployment artifact.
3. **Whole-ecosystem analysis**: precise asset-flow and interaction-graph tooling over verified source.

These are ongoing products, not a phase that completes. Item 1 should land close behind launch, because it is the concrete answer to "why a dedicated language".

Phase 6 — Launch

- Freeze the language spec for the first version and ABI v1.
- Publish the porting guide from Clarity 1–6, built on the disposition table in spec/04-standard-library.md.

Dependency summary



Phase 4 needs only the compiler. Phase 5 needs the chain's fuel schedule and host-charge table from Phase 3, so the analyzer's design can start with Phase 2 but its bounds cannot be stated until ABI v1 fixes the schedule.

Top risks

1. **ABI frozen too early**: a wrong surface forever, for every language. Gate the freeze on Clarity shipping against it, and review the ABI for Clarity-shaped assumptions before freezing.
2. **Compiler bugs**: deployed code differs from audited source. Conformance vectors, differential testing against the oracle, fuzzing, reproducible builds, and an external audit before launch.
3. **Spec drifts from the compiler**: the ECMAScript 4 failure mode. Vectors are written with each chapter, and the compiler implements the spec, never the reverse silently.
4. **Static bounds lost in compilation**: if the lowering obscures loop bounds, the analyzer cannot prove them from the module. The analyzer therefore takes loop bounds from verified source and fuel from the module, and chapter 6 must carry the preservation argument that the module stays within the bound its source implies.
5. **Justification pressure**: if Phase 5 slips indefinitely, "just use Rust on the Wasm chain" wins by default. Treat the static cost analyzer as a headline deliverable with the same priority as the compiler.
6. **Fuel schedule churn**: schedule changes happen at epoch boundaries and re-price every deployed contract, so a published bound holds only for the schedule version it names. Agree with the chain team on how often the schedule may change before launch.
7. **Bounds stop at foreign code**: a `contract-call?` through a trait, or into a module from another language, has no static bound, and most real contracts make such calls. If the analyzer can only bound leaf contracts, the max-fee product does not ship. Mitigations: bounds compose over verified-Clarity call graphs; the analyzer reports the remaining budget at each unbounded call rather than refusing; and the language design opportunity in 04-spec-plan.md (declared, checked fuel bounds that callers can rely on) is the long-term answer for trait calls.

Issue Tracker Disposition

Every open issue labeled `clarity` in stacks-core (as of 2026-09-09), with what the specification draft in spec/ does about it. Issues are grouped by disposition. "Addressed" means the draft's rules make the reported problem impossible or the request part of the language; the referenced section is where. Section numbers refer to the specification chapters in spec/.

Addressed by the draft

Issue	Summary	Where
#7312 (and #7147)	Integer to fixed-width bytes, abort on overflow	`uint-to-buff-be/le`, `int-to-buff-be/le` primitives, 3.4
#6831	A trait defined in the contract cannot be used at the contract's top level	2.4.2: local definitions are fully usable in top-level expressions
#6485	`concretize` mishandles lists mixing trait values and contract literals	No `ListUnionType`. A contract literal has a singleton type that is a subtype of every trait it implements, and list literals take the lub (2.3.2, 2.5.6)
#5128	`get-block-info? miner-address u0` makes the transaction unmineable	3.9: chain-data lookups are total and return `none`
#4751	`(is-eq trait-value .contract)` passes the checker and fails at runtime	`(trait X) <: principal`, so the lub is a principal and the comparison is well-typed; runtime values are plain principals (2.3.2, 2.7.1)
#4622	Checker rejects a list of distinct contract literals	Lub of singleton contract types is `principal` (2.3.2)
#4431	Duplicate parameter names and wrong argument counts accepted	2.4.1: parameter names distinct; 2.6.2 and 2.6.5: declared arity, checked statically
#3566	Cannot call a trait extracted from a tuple	2.5.8: any trait-typed expression is a call target
#3176	Binding a reserved keyword passes analysis, fails at runtime	2.5.2: no shadowing of any in-scope name, checked statically. The forward-compatibility concern raised in the thread is moot under frozen language versions: a new keyword only exists in a new version, and old contracts keep theirs
#1981	Read-only functions cannot call trait functions	2.4.1 and 2.7.4: traits may mark functions `read-only`; implementations must be read-only; read-only callers may call them through the trait

Issue	Summary	Where
#1638	Constants cannot be used where literals are required	2.4.1: static constants (built from literals) are usable as size terms and `contract-call?` targets
#1171	Type aliases	`define-type`, 2.3.6
#3056 (referenced by #7312)	Variadic `concat`	`concat` has a rest parameter (3.3); rest parameters are a user feature (2.6.5)

Candidate additions

Feature requests that fit the language's positioning but are not yet in the draft because each deserves a short RFC (layer, host interface impact, exact signature).

Issue	Request	Draft position
#3568	Schnorr (BIP-340) signature verification over secp256k1	Strong fit for a Bitcoin-native language (Taproot, FROST). Proposed: `secp256k1-schnorr-verify : ((buff 32) (buff 64) (buff 32)) -> bool`, `false` on malformed input, placed in 3.6. Recommend including in vNEXT
#3975	256-bit integer arithmetic for Schnorr, ZK, and curve math	RFC. A `(buff 32)`-based `wide-add`/`wide-mul`/`wide-mod` family is expressible in the language, so under the layering rule it is library code, with a Wasm replacement later if it is hot. A new 256-bit integer type would instead be a core change. The RFC chooses between the two. If #3568 ships as a primitive, the most pressing motivation goes away
#6320	`code-body-of?` / per-definition code hashes	`contract-hash?` is in the draft (3.5). The thread converges on per-top-level-definition hashes (`has-definition-value?` or a MAST) as the safe version of introspection; returning source text invites contracts to parse Clarity in Clarity. The chain holds only the module, so an on-chain per-definition hash would be over compiled code and would depend on the compiler. The RFC decides between that and per-definition source hashes declared in the module's `abi.types` section or its provenance

Out of scope for the language spec

Issue	Why
#4487, #4288	Signer-voting and StackerDB boot-contract API design
#3977, #3878, #3877	sBTC and PoX contract migration checklists (2023)
#4030	Boot-contract test infrastructure
#3350, #3302	Diagnostic quality. Implementation concerns for the reference compiler, though #3350's root cause (a `uint` expression where a literal is required) is removed by static constants
#3849	`tx-fee` context value. Under fuel pricing the fee is the fuel consumed times the rate, which is not known while the transaction runs (chain ABI 9.1)

Patterns worth noting

Three themes recur across the tracker and shaped rules in chapter 2:

- **Checker and runtime disagree** (#4751, #4431, #3176, #6485). The draft's answer is structural: one meaning per name, no shadowing, declared arity, and a value model with no static type information at runtime, so there is nothing for the two to disagree about.
- **Literals where expressions should do** (#1638, #3350, #6831). Static constants and the "any trait-typed expression" rule remove the special cases.
- **Traits are second-class** (#3566, #1981, #6485, #4622, #4751). Making `(trait X)` an ordinary type with a subtyping relation to `principal` resolves every one of these without a dedicated rule.

Chain ABI Proposal

This is a proposal for the chain ABI: the contract between any compiler and the chain, shared by every source language (02-goals.md G2). It fills in what 03-architecture.md sections 3–5 sketch. It is written to seed the joint draft with the chain team (05-roadmap.md Phase 0), so it takes a position on every question instead of listing options; the positions that change an earlier document are collected in section 12, and the questions it cannot settle alone are in section 13.

The ABI has two layers, specified in two documents:

- **The runtime layer**, this document: what the chain parses, executes, and enforces. That is the module format, the host interface, calls and frames, state, fuel, and outcomes, and the encoding of principals, the one structured value the host reads and writes.
- **The interpretation layer**, 08-value-interpretation.md: the types a contract declares for its functions and the conventions compilers, wallets, and explorers use to encode and decode the values it exchanges.

The chain depends only on the runtime layer. Arguments, results, storage, and event data are bytes to it (R1), and a module that follows no interpretation convention still deploys and runs. Tools depend on both layers.

This document draws on two prior systems:

- **The legacy interpreter** in stacks-core, which defines today's value model, wire format (SIP-005), storage layout, asset map, and call semantics.
- **clarity-wasm**, the earlier compiler from Clarity to Wasm and its host in stacks-core (`feat/clarity-wasm-develop`). Its host interface was built to match the interpreter, not designed as an ABI, and most of its problems trace to that. Section 1 lists the lessons.

1. Design rules

Each rule answers a problem found in one of the two prior systems.

Rule	Problem it answers
R1. The host never interprets a contract's values. Arguments, results, storage, and events cross the boundary as bytes; the host decodes only the values it acts on itself, in one canonical encoding.	clarity-wasm passed values in a Clarity-type-directed layout, so the host kept a copy of Clarity's type checker and re-parsed textual type strings at runtime (`print`, early returns). Unknown response arms changed Wasm signatures and needed a "duck typing" conversion layer.
R2. Every encoding has exactly one valid byte string per value. Decoding rejects anything else.	Legacy decoding accepts unsorted tuple fields, duplicate fields before epoch 4.1, and two contract-name length limits, and "sanitizes" extra tuple fields in some paths but not others.

Rule	Problem it answers
R3. Types are complete. No partially known or inference-only types.	`NoType`, `ListUnionType`, and `TraitReferenceType` in legacy, and the epoch-dependent admission rules built around them.
R4. The chain enforces every guarantee itself. No guarantee depends on a compiler emitting instrumentation or on a call being made.	clarity-wasm metering was optional codegen, and the node deploy path compiled without it. Host-function cost charges were commented out.
R5. A module has no state except chain state. Each call gets a fresh instance.	Without this, linear memory and globals could carry data between calls in one transaction, which is state that no rollback covers.
R6. Every contract-caused failure is an includable transaction outcome.	Legacy "rejectable" errors (`Unreachable`, some memory and asset-map errors) make a transaction unmineable rather than failed, which desynchronizes the sender's nonce.
R7. Semantics never change under a module. A module binds to an ABI version, and that version's behavior is frozen; only the fuel schedule moves.	Legacy reinterprets `block-height`, `get-block-info?` heights, `secp256r1-verify` hashing, and argument sanitizing by epoch, under contracts already deployed.
R8. The ABI is a single formal definition. Host and every compiler derive from it.	clarity-wasm's host interface was two hand-synchronized Rust files and a `.wat` import list.

2. Host values

The host reads and writes three kinds of value:

- **principals**, in the encoding of 2.2;
- **unsigned 128-bit integers**, as fixed-width words (6.1);
- **raw bytes**, whose length and meaning each function states.

Nothing with more structure crosses the host interface. A block's fields, an account's balances, and an allowance's fields are each passed on their own (6.2, 7.2), so the host has no record layout to decode or produce.

The values a contract exchanges are not host values: arguments, results, storage, and event data are bytes the host passes through or stores (5.4, 4.2, 6.2). The interpretation layer defines a type grammar and an encoding for them (08-value-interpretation.md sections 2–3), whose encoding of a principal is the one in 2.2. Nothing in this document depends on that layer, except a call between a legacy and a Wasm contract (section 13, question 4).

2.1 Names

Kind	Grammar	Length
Function, token, and event names	ASCII letters, digits, and `- _ ! ? + < > = / * ^`, not starting with a digit	1–128 bytes

Kind	Grammar	Length
Contract names	an ASCII letter, then letters, digits, `-`, `_`	1–40 bytes

Every source language shares these rules. One contract-name limit applies everywhere: in deploys, in principals, and on the wire. Legacy has 40 at deploy and 128 inside values before epoch 4.1.

2.2 Principals

Prefix	Principal	Payload
`0x05`	standard	version (1 byte, < 32), hash160 (20 bytes)
`0x06`	contract	version, hash160, u8 name length, name

This is the SIP-005 encoding of a principal, so a principal has the same bytes here as in legacy values, transaction post-conditions, and the interpretation layer. Decoding accepts exactly this form (R2): the prefix is `0x05` or `0x06`, the version is below 32, a contract name is valid (2.1), and no bytes remain.

3. Module format

A deployed contract is one Wasm module. The chain stores the module bytes and nothing else; any compiled or cached form of it is node-local and never consensus data.

3.1 Wasm profile

Allowed: Wasm 1.0 plus mutable globals, sign-extension operators, multi-value, and bulk memory (`memory.copy`, `memory.fill`, and the passive-segment and table operators).

Rejected at deploy:

- all floating-point types and instructions (no NaN canonicalization questions);
- SIMD, threads, reference types (`externref`), exceptions, multiple memories, memory64, tail calls, and any later proposal not listed above;
- a `start` function (initialization is an explicit export, section 3.3);
- a table with more initial entries than the table limit (section 9.3). Wasm 1.0 allows at most one table, of `funcref`, and gives the module no way to grow it.

A later ABI version may allow more; a module states the version it targets (section 11). The likely direction is the deterministic profile of the Wasm specification, which fixes the results the base specification leaves nondeterministic, such as NaN bit patterns, and would replace this hand-picked subset with a standard one (section 13).

3.2 Imports and memory

- Every import is a function from module `stacks-abi-v1` (section 11) whose name and Wasm type match the host interface exactly (section 6). Anything else is rejected.

- The module defines exactly one memory, exports it as `memory`, and declares a maximum no larger than the instance memory limit (section 9.3).

3.3 Exports

- `memory`.
- `init`, of type `[] -> []`, if the contract has deploy-time code.
- One export per interface function (section 4), under the function's name, of type `[] -> []`.

No other exports. Every entry point has the same Wasm type: arguments arrive and results leave through host functions (section 5.2), so no flattening convention is part of the ABI and no unused response arm can change a signature.

3.4 Custom sections

Section	Required	Validated by the chain
`abi.interface`	yes	yes (section 4)
`abi.types`	no	only for size; contents are defined by the interpretation layer (08-value-interpretation.md section 5)
`abi.provenance`	no	only for size; contents are owned by the toolchain that produced the module (03-architecture.md section 2)

Any other custom section, including `name`, is rejected. The module is the unit that `contract-hash?` hashes and that `clarity-verify` compares, so it carries nothing that is not executed, validated, or metadata for tools.

3.5 Deploy

The chain, in order:

1. Checks the module size limit (section 9.3), parses the module, and applies sections 3.1–3.4.
2. Decodes and validates the interface (section 4) and checks that the exports match it.
3. Instantiates the module and runs `init`, if present, in a public frame (section 7) with `sender` and `caller` set to the deployer. The new contract has an active frame, so it cannot be called during its own `init` (section 7.3).

A runtime error in `init` aborts the deploy; nothing is stored. Deploy is charged for module bytes, validation, and compilation before step 1, and for the fuel `init` consumes (9.2). A module rejected in steps 1–2 is a failed deploy that pays its charges.

4. Interface

4.1 Contents

The `abi.interface` section records what the chain acts on: which functions exist, how each may be called, which tokens the contract defines, and which events it emits. Its bytes are, in order:

Field	Encoding
`abi-version`	u32 BE, the ABI version the module targets
`functions`	u32 BE count, then per function: name, kind byte (`0` public, `1` read-only)
`fts`	u32 BE count, then per fungible token: name
`nfts`	u32 BE count, then per non-fungible token: name
`events`	u32 BE count, then per event: name

A name is a u8 length and the name's bytes (2.1). No bytes may follow `events`.

Rules:

- Names are unique within each list, and lists are sorted in strictly ascending byte order of name, so an interface has exactly one encoding (R2) and one hash: `sha512/256` of the section bytes.
- The interface carries no types and no traits. A function's arguments and result are bytes to the chain (5.2), and their declared types, the type of each NFT's identifiers, the type of each event's data, and the contract's traits are in the `abi.types` section, which the interpretation layer defines (08-value-interpretation.md section 5) and the chain does not read.
- A token or an event is addressed by its index in its sorted list. The index is a module-local handle; the chain keys state by name (section 8.1) and records events by name (section 8.3), so the name, not the index, is the stable identity.

4.2 Storage is untyped

Storage is not part of the interface. Each contract has one key-value store whose keys and values are bytes (6.2), and the host never decodes either. A source language lays its own variables and maps out over that store, for example by prefixing each key with the variable or map name. The chain does not need storage types, since it never acts on a stored value, and leaving them out removes a decode from every write and keeps the host independent of how any language structures its state. A contract makes its state readable by tools by declaring its layout (08-value-interpretation.md section 7.2).

5. Calls

5.1 Entry points

Entry	Frame	`sender / caller`
Contract-call transaction	public frame on a `public` function	origin / origin
Deploy	public frame on `init`	deployer / deployer
`call` host function	per the callee's kind	unchanged / calling contract
Read-only query (RPC)	read-only frame	as the request states

A contract-call transaction that names a missing or `read-only` function fails before any code runs. It is included and pays its intrinsic charge (9.2), with the class `no-such-function`. Its arguments are not checked (5.4); a transaction whose arguments exceed the arguments limit (9.3) is invalid.

The node's read-only query runs in an explicit read-only frame, so a write is a `read-only-violation`, not a zeroed write budget surfacing as a cost error, as in legacy. It runs under the node's own fuel limit (9.1). Simulating a public function is a separate query that runs a public frame and discards it.

5.2 Arguments and results

Inside an entry point:

- `args_size()` returns the byte length of the arguments, and `args_read(dst, offset, len)` copies `len` bytes of them, starting at `offset`, into memory, so a module can read them whole or piecewise. The arguments are the bytes the caller passed, unchanged: a contract-call transaction and a read-only query carry them as one byte string, and `call` takes them as `args`. The chain does not parse them; how a caller lays them out is the interpretation layer's convention (08-value-interpretation.md section 6.1).
- `result_set(ok, result)` sets the result: a flag, `1` for ok and `0` for err, and the result's bytes. It does not return: the host ends the export there, as `abort` does, so no host function runs after the result is set and a result cannot be set twice. For a `public` function, the flag closes the frame: ok commits, err discards (7.1). For a `read-only` function, the flag is passed to the caller and recorded, and has no other effect. An export that returns without calling `result_set` is a `no-result` runtime error.

The module chooses where the bytes go; the ABI defines no allocator and no exported buffer.

5.3 The `call` host function

```
call(target: principal, fn: bytes, args: bytes, read_only: i32, out: buf,
     ok: out<1>) -> len
```

In the notation of 6.1, this is the Wasm type `(param i32 i32 i32 i32 i32 i32 i32 i32 i32 i32) (result i32)`.

- `target` is the callee, a principal.
- `fn` is the function name's bytes (2.1), not an encoded value.
- `args` is the arguments as in 5.2, passed to the callee unchanged.
- `read_only` is `1` to require the callee to be `read-only`, and `0` otherwise; any other value is `bad-host-argument`. A caller sets it when it relies on the callee not writing, as when it calls a function that a trait marks `read-only`, so the host enforces that in any frame.
- `ok` receives the callee's result flag (5.2), as one byte.

The host, in order:

1. Resolves the target. A standard principal, a missing contract, or a missing function is a runtime error (`no-such-function`).
2. If the current frame is read-only, or `read_only` is `1`, requires `fn` to be `read-only` (section 7.4).
3. Applies the reentrancy and depth rules (section 7.3).
4. Opens the callee's frame, instantiates a fresh instance, runs the export, and closes the frame: for a public function, commit if the result flag is ok and discard if it is err.
5. Writes the result flag to `ok`, writes the result to `out`, and returns its length. If `cap` is too small, it writes the flag but not the result, keeps the result, and returns the negated length; `result_read(dst, offset, len)` then copies the kept result, whole or in pieces. A kept result lasts until the next `call`; `result_read` with no kept result is a `bad-host-argument` runtime error. The callee's frame has already closed either way. A module that bounds the result in advance, as a Clarity module does from the declared result type (08-value-interpretation.md section 4.1), sizes `out` for that bound and treats a longer result as one that fails validation (5.4), so it never needs the second step.

Steps 1–3 failing, and any runtime error in the callee, abort the transaction. There is no catch.

5.4 Values at a call boundary

The chain does not validate the values that cross a call boundary: the arguments of a transaction or query (5.1), the arguments of a `call` (5.3), and every result (5.2) pass through as bytes. A module validates what it receives. A callee validates its arguments, and a caller validates the result of every call, since nothing guarantees that either matches the declared types. A module cannot read another contract's declared types, so it cannot check at runtime that a principal it receives conforms to a trait. What it can rely on at a call through that principal is that the function exists (5.3, step 1) and, when it sets `read_only`, that the function is `read-only` (step 2), and it validates the result. Validation is module code, charged as execution (9.2) like any other work.

Declared types and the encoding conventions that wallets, explorers, and compilers rely on are the interpretation layer's (08-value-interpretation.md section 6). Nothing in this section depends on a module following them.

Keeping validation out of the host keeps every language's type checker out of the host and the mempool. A transaction with malformed arguments is included, fails in the module, and pays for the check. What the chain still guarantees at a call boundary is the frame: commit or rollback by the result flag, read-only enforcement, reentrancy, and depth (section 7). A value that fails validation is a runtime error of the source language, reported through `abort` (10.2).

6. Host interface

6.1 Conventions

Signatures in 6.2 are written with typed parameters. Each parameter kind expands to a fixed sequence of Wasm parameters, in the order written, and the import's Wasm type is the concatenation:

Kind	Wasm parameters	Meaning
`i32`	`i32`	an integer, as the function describes
`u128`	`i64 lo`, `i64 hi`	an unsigned 128-bit integer, low 64 bits first
`principal`	`i32 ptr`, `i32 len`	a principal's encoding (2.2); a failed decode is `bad-host-argument`
`bytes`	`i32 ptr`, `i32 len`	raw bytes, not an encoded value
`bytes<N>`	`i32 ptr`	exactly `N` raw bytes
`out<N>`	`i32 ptr`	the host writes exactly `N` bytes; a `u128` result is written as 16 bytes, little-endian
`buf`	`i32 ptr`, `i32 cap`	the host writes at most `cap` bytes and returns the length; what it writes (an encoded principal, or raw bytes) is stated per function

Result	Wasm result	Meaning
none	none	
`len`	`i32`	the number of bytes written to the `buf`, with the short-buffer rule of 5.3; `-1` means "none" where the function says so
`status`	`i32`	`0` for success, otherwise one of the codes listed with the function
`bool`	`i32`	`1` for true, `0` for false

For example, `stx_burn(amount: u128, from: principal) -> status` is imported with Wasm type `(param i64 i64 i32 i32) (result i32)`.

- Lengths and capacities are unsigned. Every range a call names (`ptr` to `ptr + len`, `ptr + N`, or `ptr + cap`) must lie inside the module's memory; otherwise the call is a `bad-host-argument` runtime error, raised before the call has any effect.
- The host reads every input before it writes any output, so an output may overlap an input.
- An index argument (`token`, `event`, `kind`) outside its range is a `bad-host-argument` runtime error.
- Every host function has a charge (section 9.2).

6.2 Functions

Each function is given by its signature in the notation of 6.1, then its behavior. Status codes are listed where a function has them; `0` is always success.

Boundary and control

`args_size`

```
args_size() -> i32
```

Returns the byte length of the current entry point's arguments (5.2).

`args_read`

```
args_read(dst: i32, offset: i32, len: i32)
```

Writes bytes `offset` to `offset + len` of the arguments (5.2) at `dst`. A range that extends past `args_size()` is `bad-host-argument`.

`result_set`

```
result_set(ok: i32, result: bytes)
```

Sets the result of the current entry point and its flag (5.2): `ok` is `1` for ok and `0` for err; any other value is `bad-host-argument`. A `result` longer than the result limit (section 9.3) is `bad-host-argument`. Does not return. `init` has no result; it ends by returning, and calling `result_set` in `init` is `bad-host-argument`.

`abort`

```
abort(code: i32, msg: bytes)
```

Aborts the transaction with a module-defined code and message, both recorded in the receipt as given (section 10). `msg` is at most 128 bytes; a longer one is `bad-host-argument`. Does not return.

Context

Each piece of transaction and block context has its own function, and new context arrives as a new function in a new ABI version (section 11). The frame-scoped functions return the value for the current frame; the others return the same value in every frame of a transaction.

`sender`

```
sender(out: buf) -> len
```

Writes the encoded transaction sender, or the current contract inside an `as-contract` frame (7.2). Frame-scoped.

`caller`

```
caller(out: buf) -> len
```

Writes the encoded caller: the transaction sender for the entry call, the calling contract under `call`, or the current contract inside an `as-contract` frame. Frame-scoped.

`sponsor`

```
sponsor(out: buf) -> len
```

Writes the encoded sponsor, or returns `-1` if the transaction is not sponsored.

`contract`

```
contract(out: buf) -> len
```

Writes the encoded principal of the contract being executed. Frame-scoped.

`stacks_height`

```
stacks_height(out: out<16>)
```

Writes the height of the current Stacks block as a `u128`.

`burn_height`

```
burn_height(out: out<16>)
```

Writes the height of the burn block the current Stacks block is anchored to, as a `u128`.

`tenure_height`

```
tenure_height(out: out<16>)
```

Writes the number of tenures elapsed as a `u128`.

`time`

```
time(out: out<16>)
```

Writes the timestamp of the current Stacks block, in seconds, as a `u128`.

`liquid_supply`

```
liquid_supply(out: out<16>)
```

Writes the total liquid uSTX as a `u128`.

`chain_id`

```
chain_id(out: out<16>)
```

Writes the chain identifier as a `u128`.

`is_mainnet`

```
is_mainnet() -> bool
```

Whether the chain is mainnet.

`is_regtest`

```
is_regtest() -> bool
```

Whether the chain is a regtest environment.

Block data

Lookups of Stacks blocks, tenures, and burn blocks by height. They are total: each reports a height the host cannot serve through its result, as status `1` or as `-1` for a function that writes a `buf`, and no height is ever a runtime error or a reason to drop the transaction. Like the context values, each field is its own function, so a later ABI version can add one without changing a layout.

The Stacks block lookups serve heights below `stacks_height`. The current block's hashes do not exist until it is sealed, after this transaction runs; its time is what `time` returns.

`stacks_block_id_header_hash`

```
stacks_block_id_header_hash(height: u128, out: out<32>) -> status
```

Writes the index block hash of the Stacks block at `height`.

`stacks_block_header_hash`

```
stacks_block_header_hash(height: u128, out: out<32>) -> status
```

Writes the header hash of the Stacks block at `height`.

`stacks_block_time`

```
stacks_block_time(height: u128, out: out<16>) -> status
```

Writes the timestamp of the Stacks block at `height`, in seconds, as a `u128`.

The tenure lookups serve every tenure up to and including `tenure_height`, since every field is fixed at sortition.

`tenure_burnchain_header_hash`

```
tenure_burnchain_header_hash(height: u128, out: out<32>) -> status
```

Writes the header hash of the burn block associated with the tenure at `height`.

`tenure_miner_address`

```
tenure_miner_address(height: u128, out: buf) -> len
```

Writes the encoded principal of the miner of the tenure at `height`.

`tenure_time`

```
tenure_time(height: u128, out: out<16>) -> status
```

Writes the timestamp of the tenure at `height`, in seconds, as a `u128`.

`tenure_vrf_seed`

```
tenure_vrf_seed(height: u128, out: out<32>) -> status
```

Writes the VRF seed of the tenure at `height`.

`tenure_block_reward`

```
tenure_block_reward(height: u128, out: out<16>) -> status
```

Writes the block reward of the tenure at `height`, in uSTX, as a `u128`. Maturity is defined by the host in terms of the tenure height at which the reward was recorded.

- `1` height not served
- `2` the reward has not matured

`tenure_miner_spend_total`

```
tenure_miner_spend_total(height: u128, out: out<16>) -> status
```

Writes the total spent by all miners in the sortition of the tenure at `height`, as a `u128`.

`tenure_miner_spend_winner`

```
tenure_miner_spend_winner(height: u128, out: out<16>) -> status
```

Writes the amount spent by the winning miner in the sortition of the tenure at `height`, as a `u128`.

The burn block lookups serve every burn block up to and including `burn_height`.

`burn_block_header_hash`

```
burn_block_header_hash(height: u128, out: out<32>) -> status
```

Writes the header hash of the burn block at `height`.

`burn_block_pox_addr`

```
burn_block_pox_addr(height: u128, index: i32, out: buf) -> len
```

Writes PoX reward address `index` of the burn block at `height`: its version byte, then its hash bytes (at most 32), as raw bytes. Returns `-1` if the height is not served or the block has no address at `index`. A block has at most two addresses, so an `index` above 1 is `bad-host-argument`.

`burn_block_pox_payout`

```
burn_block_pox_payout(height: u128, out: out<16>) -> status
```

Writes the amount paid to each PoX reward address of the burn block at `height`, as a `u128`.

Storage

The current contract's key-value store (4.2). Keys and values are raw bytes, compared byte for byte. A key longer than the storage key limit, or a value longer than the storage value limit (section 9.3), is `bad-host-argument`.

`kv_get`

```
kv_get(key: bytes, out: buf) -> len
```

Writes the value stored under `key`, or returns `-1` if there is no entry.

`kv_has`

```
kv_has(key: bytes) -> bool
```

Whether an entry exists under `key`. Charged as a read without the value's length.

`kv_set`

```
kv_set(key: bytes, value: bytes)
```

Inserts or overwrites the entry under `key`.

`kv_delete`

```
kv_delete(key: bytes) -> bool
```

Removes the entry under `key`; returns whether it existed.

Assets

`token` is an index into `fts` or `nfts` (4.1). A contract can only operate its own tokens, because it can only name its own indices. An NFT identifier `id` is raw bytes in an encoding the token's contract chooses; the chain compares identifiers byte for byte. An identifier longer than the NFT identifier limit (section 9.3) is `bad-host-argument`.

Every successful mutation records an entry in the asset map (8.2) and emits an asset event. STX functions require `from` to be the frame's `sender`. Token functions do not check the sender: every movement is in the asset map, so post-conditions and allowance frames (7.2) protect the holder, and a token's contract decides its own transfer rules.

`ft_set_cap`

```
ft_set_cap(token: i32, cap: u128)
```

Sets the token's supply cap. Valid only during `init`, at most once per token; otherwise a runtime error. The cap is a host call rather than interface data because deploy-time code computes it.

`ft_mint`

```
ft_mint(token: i32, amount: u128, to: principal) -> status
```

Mints `amount` to `to`. Exceeding the supply cap is a runtime error.

- `1` amount is zero

`ft_burn`

```
ft_burn(token: i32, amount: u128, from: principal) -> status
```

Burns `amount` from `from`.

- `1` insufficient balance, or amount is zero

`ft_transfer`

```
ft_transfer(token: i32, amount: u128, from: principal, to: principal) -> status
```

Moves `amount` from `from` to `to`.

- `1` insufficient balance
- `2` `from` is `to`
- `3` amount is zero

`ft_balance`

```
ft_balance(token: i32, owner: principal, out: out<16>)
```

Writes `owner`'s balance as a `u128`.

`ft_supply`

```
ft_supply(token: i32, out: out<16>)
```

Writes the circulating supply as a `u128`.

`nft_mint`

```
nft_mint(token: i32, id: bytes, to: principal) -> status
```

Mints the identifier `id` to `to`.

- `1` already exists

`nft_burn`

```
nft_burn(token: i32, id: bytes, from: principal) -> status
```

Burns `id`, which `from` must own.

- `1` not owned by `from`
- `3` does not exist

`nft_transfer`

```
nft_transfer(token: i32, id: bytes, from: principal, to: principal) -> status
```

Changes the owner of `id` from `from` to `to`.

- `1` not owned by `from`
- `2` `from` is `to`
- `3` does not exist

`nft_owner`

```
nft_owner(token: i32, id: bytes, out: buf) -> len
```

Writes the encoded owner of `id`, a `principal`, or returns `-1` if `id` does not exist.

`stx_transfer`

```
stx_transfer(amount: u128, from: principal, to: principal, memo: bytes) -> status
```

Moves `amount` uSTX from `from` to `to`, recording `memo` in the event. `memo` is at most 34 bytes, and a longer one is `bad-host-argument`; an empty memo is recorded as no memo.

- `1` insufficient balance
- `2` `from` is `to`
- `3` amount is zero
- `4` `from` is not the frame's `sender`

`stx_burn`

```
stx_burn(amount: u128, from: principal) -> status
```

Burns `amount` uSTX from `from`.

- `1` insufficient balance
- `3` amount is zero
- `4` `from` is not the frame's `sender`

`stx_locked`

```
stx_locked(owner: principal, out: out<16>)
```

Writes the uSTX `owner` has locked for staking, as a `u128`.

`stx_unlock_height`

```
stx_unlock_height(owner: principal, out: out<16>)
```

Writes the burn height at which `owner`'s locked uSTX unlock, as a `u128`, or `0` if none are locked.

`stx_unlocked`

```
stx_unlocked(owner: principal, out: out<16>)
```

Writes `owner`'s unlocked uSTX balance, as a `u128`.

Frames

`frame_open`

```
frame_open(kind: i32, desc: bytes)
```

Opens a frame (section 7): `0` plain, `1` as-contract, `2` restrict. `desc` is empty for a plain frame and the allowance descriptor of 7.2 otherwise.

`frame_close`

```
frame_close(commit: i32) -> i32
```

Closes the innermost module-opened frame. `commit` is `0` to discard or `1` to commit after the allowance check. Returns `-1`, or the index of the violated allowance (7.2).

Calls and principals

`call`

```
call(target: principal, fn: bytes, args: bytes, read_only: i32, out: buf,
     ok: out<1>) -> len
```

Calls a function of another contract and writes its result flag and result (5.3). An `args` longer than the arguments limit (section 9.3) is `bad-host-argument`.

`result_read`

```
result_read(dst: i32, offset: i32, len: i32)
```

Writes bytes `offset` to `offset + len` of the result kept by the last `call` that returned a negated length (5.3) at `dst`. A range that extends past the kept result is `bad-host-argument`.

`contract_hash`

```
contract_hash(contract: principal, out: out<32>) -> status
```

Writes the `sha512/256` of the contract's module.

- `1` the principal is a standard principal
- `2` no such contract

`principal_to_ascii`

```
principal_to_ascii(p: principal, out: buf) -> len
```

Writes the c32check text form of `p` as raw ASCII bytes, not an encoded value. A contract principal is the address, `.`, and the name.

Events

`emit`

```
emit(event: i32, data: bytes)
```

Emits a contract event. `event` is an index into `events` (4.1), so a contract can emit only the events its interface names. The event records the emitting contract, the event's name, and `data` (8.3). `data` is at most the event data limit (section 9.3); a longer one is `bad-host-argument`. It also counts toward the transaction's event bytes (section 8.3), and passing their limit (section 9.3) is an `events-exceeded` runtime error. The chain does not interpret `data`: by convention it is the canonical encoding of a value of the event's declared type (08-value-interpretation.md section 7.1), which is what indexers and explorers decode. Asset events are emitted by the asset functions themselves.

`log`

```
log(data: bytes)
```

Records `data` as debug output, which the chain discards. It is not an event: it is in no receipt and does not count toward the transaction's event bytes. No limit applies to `data` beyond the instance memory that holds it: the host copies nothing on a node that discards it. The call is charged like any other host function (9.2), so a call costs and fails the same whether or not anything shows the output. A node running a simulation or a

development network MAY show `data` to the developer. The chain does not interpret it; a source language passes whatever helps debugging, and Clarity passes a value's canonical encoding (spec section 3.10).

Cryptography

Formats: an ECDSA signature is `r` then `s`, each 32 bytes big-endian; a recoverable signature appends one recovery-id byte, `0`–`3`. ECDSA public keys are SEC 1 compressed points, 33 bytes; the verify functions accept no other form, and `secp256k1_decompress` is the only function that produces the 65-byte uncompressed form. Ed25519 keys and signatures are as in RFC 8032.

An ECDSA signature is canonical only if its `S` is in the lower half of the curve order. The verify functions return `0`, and `secp256k1_recover` returns `1`, for a non-canonical signature; accepting both `S` and `n - S` would let anyone replay a signature-gated authorization with a different encoding.

`sha256`

```
sha256(data: bytes, out: out<32>)
```

Writes the SHA-256 hash of `data`, as defined in FIPS 180-4.

`sha512`

```
sha512(data: bytes, out: out<64>)
```

Writes the SHA-512 hash of `data`, as defined in FIPS 180-4.

`sha512_256`

```
sha512_256(data: bytes, out: out<32>)
```

Writes the SHA-512/256 hash of `data`, as defined in FIPS 180-4, which starts from its own initial hash value. It is not the first 32 bytes of SHA-512.

`keccak256`

```
keccak256(data: bytes, out: out<32>)
```

Writes the Keccak-256 hash of `data`, as defined in the Keccak submission to the SHA-3 competition (version 3), with its original padding. It is not FIPS 202 SHA3-256, which pads differently and gives different results.

`ripemd160`

```
ripemd160(data: bytes, out: out<20>)
```

Writes the RIPEMD-160 hash of `data`, as defined in ISO/IEC 10118-3.

`secp256k1_recover`

```
secp256k1_recover(hash: bytes<32>, sig: bytes<65>, out: out<33>) -> status
```

Writes the compressed public key that produced `sig` over `hash`.

- `1` malformed or non-canonical signature
- `2` recovery failed

`secp256k1_verify`

```
secp256k1_verify(hash: bytes<32>, sig: bytes<64>, key: bytes<33>) -> bool
```

Whether `sig` is a valid, canonical secp256k1 signature of `hash` by `key`. `hash` is used as given, not hashed again.

`secp256r1_verify`

```
secp256r1_verify(hash: bytes<32>, sig: bytes<64>, key: bytes<33>) -> bool
```

Whether `sig` is a valid, canonical P-256 signature of `hash` by `key`. `hash` is used as given, not hashed again.

`secp256k1_decompress`

```
secp256k1_decompress(key: bytes<33>, out: out<65>) -> status
```

Writes the uncompressed form of `key`.

- `1` not a valid key

`ed25519_verify`

```
ed25519_verify(msg: bytes, sig: bytes<64>, key: bytes<32>) -> bool
```

Whether `sig` is a valid Ed25519 signature (RFC 8032) of `msg`, which may be any length.

7. Frames and runtime guarantees

7.1 Frames

A frame buffers state changes: storage writes, token and STX state, asset-map entries, and events. Frames nest. Closing a frame either commits its changes into the parent or discards them.

Frame	Opened by	Closed
public	the host, for every call to a `public` function and for `init`	by the host: by the result flag (5.2); `init` commits when it returns
read-only	the host, for every call to a `read-only` function	by the host: commit (there are no writes, only events)
plain	the module, `frame_open(0, …)`	by the module
as-contract	the module, `frame_open(1, allowances)`	by the module, with the allowance check
restrict	the module, `frame_open(2, owner + allowances)`	by the module, with the allowance check

Plain frames are how a module calls its own public function internally with rollback. Module-opened frames are well nested within one call: closing a frame the module did not open, or ending an export (by returning or by `result_set`) with a frame still open, is an `unbalanced-frame` runtime error.

Events of a read-only frame are kept and commit with the caller. Legacy drops them because it rolls the frame back; nothing in a read-only frame needs rolling back.

7.2 Allowance frames

`frame_open(1 | 2, desc)` takes an allowance descriptor. For a `restrict` frame, `desc` starts with the owner's encoding (2.2). Then, for both kinds:

- a u8 count of allowances, at most 128;
- per allowance, a kind byte, then its fields in the order of the table below: a `principal` as its encoding (2.2), a `name` as a u8 length and the name's bytes (2.1), and a `u128` as 16 bytes, little-endian (6.1);
- for `nft`, the `ids` field is a u32 BE count, then per identifier a u32 BE length and the identifier's bytes (6.2).

No bytes may follow the last allowance. A descriptor that does not decode this way is a `bad-host-argument` runtime error. The allowance kinds are ABI data:

Byte	Kind	Fields	Covers
`0`	`stx`	`amount: u128`	the owner's uSTX transferred plus burned
`1`	`ft`	`contract: principal`, `token: name`, `amount: u128`	the owner's movements of that token, summed

Byte	Kind	Fields	Covers
`2`	`nft`	`contract: principal`, `token: name`, `ids`: identifiers	each moved identifier must be listed, compared by bytes
`3`	`all`	none	everything; only in an `as-contract` frame
`4`	`staking`	`amount: u128`	the uSTX the owner locks for staking, summed
`5`	`pox`	none	any position-altering PoX action by the owner

`as-contract` also sets the frame's `sender` and `caller` to the current contract. `frame_close(1)` on an allowance frame compares the owner's outgoing movements, stakes, and PoX actions in the frame's asset map (8.2) against the list. If all are covered, it commits and returns `-1`. Otherwise it discards and returns the index of the first allowance violated, in list order, or `128` (the allowance limit) for a movement that no allowance names. `frame_close(1)` on a plain frame commits and returns `-1`. `frame_close(0)` discards any frame without a check and returns `-1`.

7.3 Depth and reentrancy

- A call into a contract that already has an active frame in the transaction is a `reentrancy` runtime error. The unit is the contract. Legacy checks per (contract, function), so `A.f → B.g → A.h` passes there.
- Contract-call depth is at most 64. This counts frames the host opens for calls, not module-internal function calls, which the chain cannot see. Legacy's 128 counts every function application, builtins included, so the two numbers are not comparable. A source language may bound its own internal call depth separately.
- Wasm stack depth is bounded separately and deterministically (section 9.3).

7.4 Read-only

In a read-only frame, every storage write, asset mutation, and `frame_open` of an allowance kind is a `read-only-violation` runtime error, and `call` accepts only `read-only` callees. A `call` with `read_only` set (5.3) applies the same callee check from any frame. Legacy checks storage writes at runtime but relies on analysis for asset functions; here every mutation is checked.

8. State and effects

8.1 State layout

Contract storage is keyed by contract and the key's bytes, and stores the value's bytes as given (4.2). Token state is keyed by contract, token name, and, for NFTs, the identifier's bytes. Legacy uses serde JSON for principals in token-balance keys and for persisted type metadata; nothing here depends on a language's in-memory representation. The concrete key strings and the encoding of balances and supplies are the chain's choice, made once.

8.2 Asset map

Every asset operation records an entry: kind (transfer, mint, burn), asset (STX, or contract and token name), from, to, and amount or identifier. Post-conditions and allowance frames are evaluated over it.

Staking adds two more kinds of entry, recorded by the host when a module calls the chain's PoX contract:

- **Stake**: the staker and the uSTX newly locked, recorded by every PoX function that locks uSTX. A `staking` allowance covers their sum within the frame.
- **PoX action**: the staker, recorded by every PoX function that otherwise changes or ends the staker's position. A `pox` allowance covers it.

The staker is the frame's `sender` at the call. The ABI fixes what each kind of entry means; the PoX contract specifies which of its functions record which entry, so a later PoX contract can define its own. Legacy records only outflows, merges burns into transfers for tokens, and does not record mints; recording everything costs little and lets post-conditions and allowances say what they mean.

8.3 Events

Events are contract events and asset events, each tagged with the emitting contract. A contract event also carries its name, from the contract's interface (4.1), and its data bytes (6.2), so an observer can select events by contract and name without decoding any data. An asset event is an entry of the asset map (8.2), of any kind, including stake and PoX-action entries, which the PoX contract emits. Events live in the frame that emitted them and commit or discard with it.

An event's **size** is a byte count fixed by the ABI, regardless of the encoding a node uses for its receipts. Every event counts its kind as 1 byte and its emitting contract as a principal, and then its own fields:

Event	Fields counted
Contract event (`emit`)	name, data
STX transfer	from, to, amount, memo
STX burn	from, amount
Fungible-token transfer	token, from, to, amount
Fungible-token mint	token, to, amount
Fungible-token burn	token, from, amount
NFT transfer	token, from, to, identifier
NFT mint	token, to, identifier
NFT burn	token, from, identifier
Stake	staker, amount
PoX action	staker

A principal counts the length of its SIP-005 encoding (2.2). An amount counts 16 bytes. A name, data, an identifier, and a memo count their length in bytes. A token counts only its name, since a module emits events only for its own tokens and its contract is the emitting one. The transaction's **event bytes** are the total size of its live events: those recorded in a frame that has not rolled back. Recording an event adds its size, and a frame that rolls back subtracts the sizes of the events it discards, so they stop counting. A host function whose event would pass the limit (9.3) is an `events-exceeded` runtime error. The limit therefore bounds the receipt, not the work of recording events, which fuel already charges (9.2).

8.4 Post-conditions

A transaction carries a post-condition mode and a list of post-conditions. Their wire layout belongs to the transaction format; their meaning is defined here, over the asset map (8.2), so they mean the same thing whatever module runs. The kinds, codes, and modes are legacy's, as extended by SIP-040 (originator mode, `may-send`) and SIP-045 (staking and PoX post-conditions).

When. The host evaluates post-conditions once, when the entry point sets an ok result or `init` returns, over the asset map the transaction is about to commit. Entries from discarded frames are already gone, so post-conditions see exactly what would commit. A transaction whose entry point sets an err result commits nothing and its post-conditions are not evaluated.

Kinds. Each post-condition names a principal, either explicitly or as the transaction's origin account, and constrains that principal's outgoing entries: transfers from it, burns from it, its stakes, and its PoX actions. The kinds mirror the allowance kinds (7.2):

Kind	Fields	Holds when
`stx`	principal, comparator, amount	the principal's uSTX transferred plus burned compares to `amount`
`ft`	principal, contract, token, comparator, amount	the principal's transfers and burns of that token, summed, compare
`nft`	principal, contract, token, identifier, code	the identifier's movement from the principal matches the code
`staking`	principal, comparator, amount	the uSTX the principal newly locked, summed, compares to `amount`
`pox`	principal, code	whether the principal recorded a PoX action matches the code

A comparator is one of `=`, `<`, `<=`, `>`, `>=`, applied as "actual comparator amount". Identifiers compare byte for byte, as in 7.2. The codes are:

Kind	Code	Holds when
`nft`	`sent`	the identifier left the principal (transfer or burn)
`nft`	`not-sent`	the identifier did not leave the principal
`nft`	`may-send`	always; it exists to cover the identifier (below)

Kind	Code	Holds when
`pox`	`performed`	the principal recorded at least one PoX action
`pox`	`not-performed`	the principal recorded no PoX action
`pox`	`may-perform`	always; it exists to cover the principal's PoX actions

Coverage. A post-condition covers the outgoing entries it constrains, whatever its comparator or code: an `stx` post-condition covers the principal's STX entries, an `ft` the principal's entries for that token, an `nft` the principal's entries for that one identifier, a `staking` the principal's stakes, and a `pox` the principal's PoX actions.

Modes.

- **Allow**: every listed post-condition must hold.
- **Deny**: every listed post-condition must hold, and every outgoing entry in the asset map must be covered.
- **Originator**: every listed post-condition must hold, and every outgoing entry of the origin account must be covered. Other principals' entries need not be; a contract the user does not trust protects its own assets with allowance frames (7.2).

The **origin account** is the account of the transaction's origin spending condition. It is fixed for the transaction: it is not the frame's `sender`, does not change in an `as-contract` frame, and in a sponsored transaction is the origin, not the sponsor.

A mint is not an outgoing entry for any principal, so recording mints (8.2) never fails a deny-mode or originator-mode transaction. A transfer is outgoing for its sender only.

A post-condition that does not hold makes the transaction's outcome `post-condition failure` (10.1): state discarded, fee paid, no events.

9. Fuel, charges, and limits

9.1 Budget and fee

Cost has one dimension, **fuel**. Wasm execution, host functions, storage reads and writes, instantiation, and the transaction's own processing are all charged in fuel against one budget. Legacy's five dimensions (`runtime`, `read_count`, `read_length`, `write_count`, `write_length`) do not exist here.

A transaction carries two fee fields:

Field	Meaning
`fuel-limit`	the most fuel the transaction may consume
`fuel-rate`	the price, in uSTX per million fuel

Its maximum fee is `fuel-limit × fuel-rate / 10^6`, and the payer (the sponsor, if there is one) must hold that much for the transaction to be valid. The fee charged is the fuel consumed times the rate, rounded up to a whole uSTX; the rest is never taken. A transaction that exceeds its limit is aborted with `budget-exceeded` and pays for its whole limit. The whole fee goes to the miner; there is no protocol base fee.

The rate is per million fuel because one fuel is about one Wasm instruction, so any realistic price per fuel is far below 1 uSTX.

The two fields give a miner what it needs to build a block: it orders transactions by rate and knows each one's worst case from its limit. A block has one fuel limit (9.3), and a transaction whose execution would take the block past it is not included.

Read-only queries pay no fee. Each node sets its own fuel limit for them and may choose not to serve them at all.

9.2 Charges

Every charge is a worst-case price. A node may be faster in the common case, for example by caching state, but the charge does not depend on it.

- **Intrinsic.** Before anything runs, the transaction is charged for its bytes and its signature check, and a deploy is charged for module bytes, validation, and compilation. A transaction whose intrinsic charge alone exceeds its `fuel-limit` is invalid and not included. Once the signature checks out, the transaction is included and pays, even if it then fails before executing (a missing function): the miner has done the work, and an unpaid failure would be a free way to make it do more.
- **Execution.** Wasm execution is metered by the engine: each operator consumes the fuel the schedule assigns it, and the engine traps when the budget runs out. Bulk-memory and table operators (3.1) also consume fuel per byte or element they copy, fill, or initialize. Growing memory is charged per page.
- **Host functions.** Each has a base charge plus a per-byte charge on its inputs and outputs.
- **Storage.** Reads and writes are charged per access and per byte, priced so that a block spent entirely on them stays within what nodes can read, write, and keep. Writes are priced well above computation for that reason: there is no separate limit on state growth.
- **Instantiation.** Each call is charged per byte of the callee module, in place of legacy `LoadContract`.

The fuel schedule (these prices and the fuel of each Wasm operator) is versioned, may change at an epoch, and applies to every module (03-architecture.md section 4). This is the one thing that moves under a deployed module (R7). Because execution is metered by the engine, the engine's fuel accounting is part of the schedule: an engine upgrade that changes how much fuel any execution consumes is a consensus change, and the chain pins the engine's behavior to the schedule. The engine is wasmi, whose fuel costs and stack limits are configurable; the schedule and 9.3 fix them, and no node sets its own.

9.3 Limits

Limit	Value (provisional)	Notes
Contract-call depth	64	7.3
Instance memory	16 MiB	bounds every intermediate value a module holds
Table entries	TBD	bounds the allocation at instantiation
Transaction memory	64 MiB	sum over live instances; caps depth × instance memory
Wasm stack	TBD	in schedule-defined units, enforced by deploy-time instrumentation
Module size	TBD	the only bound on a contract's size
NFT identifier	1 MiB	bytes per identifier
Storage key	1 KiB	bytes per key
Storage value	1 MiB	bytes per value
Arguments	1 MiB	bytes per transaction, query, or `call`
Result	1 MiB	bytes per `result_set`
Event data	1 MiB	bytes per `emit`
Allowances per frame	128	
Fuel per block	TBD	9.1
Fuel per read-only query	node's choice	9.1
Event bytes per transaction	TBD	live events only (8.3); `events-exceeded` (10.2)

Two of these need care beyond a number:

- **Stack.** wasmi's stack limits (value stack height and recursion depth) are in the engine's own units, which depend on how it translates code and may change between engine versions, so a stack overflow it detects would not be stable across upgrades. The chain bounds stack height in its own units, computed per function at deploy, and sets the engine's limits well above the maximum.
- **Memory.** Without a transaction cap, 64 nested instances at 16 MiB each is 1 GiB. Growth is charged in fuel, and the cap is a runtime error.

10. Outcomes and errors

10.1 Transaction outcomes

Outcome	Included	State	Events in receipt
`ok`	yes	committed	yes

Outcome	Included	State	Events in receipt
`err`	yes	discarded; fee paid	no
aborted (runtime error)	yes	discarded; fee paid	no
post-condition failure	yes	discarded; fee paid	no
failed before execution (5.1)	yes	none; fee paid	no
invalid (9.1, 9.2)	no		
does not fit the block (9.1)	no		

A transaction is **invalid** when it cannot be charged: a bad signature or nonce, a payer who cannot cover the maximum fee, an intrinsic charge above its `fuel-limit`, or arguments over the arguments limit (9.3). Everything else a miner does work on is included and paid for.

No other outcome exists (R6). In particular, the receipt of a discarded transaction carries no events from the discarded execution, and a failed deploy does not report `(ok true)`; both happen in legacy.

10.2 Runtime error classes

The receipt of an aborted transaction names one class:

Class	Cause
`budget-exceeded`	the transaction's `fuel-limit` (9.1)
`memory-exceeded`	instance or transaction memory
`stack-exceeded`	Wasm stack limit
`depth-exceeded`	contract-call depth
`events-exceeded`	the transaction's event bytes (8.3, 9.3)
`reentrancy`	7.3
`read-only-violation`	7.4
`no-such-function`	5.1; 5.3, step 1
`no-result`	an entry point returned without calling `result_set` (5.2)
`bad-host-argument`	6.1
`unbalanced-frame`	7.1
`trap`	any Wasm trap
`module-abort`	`abort(code, msg)`; the receipt also carries the code and message

`module-abort` codes belong to the source language. Clarity's compiler maps its runtime errors (overflow, division by zero, `panic`, failed conversion, and a value that fails validation at a call boundary, 5.4) to codes, which is how a language vector's "abort, class arithmetic-overflow" is checked. Legacy and clarity-wasm signalled these through exported globals and `unreachable`.

11. Versioning

- The import module name carries the ABI version: `stacks-abi-v1`. The interface section's `abi-version` must match.
- A frozen version never changes: its host functions, validation rules, and semantics are fixed for every module that targets it (R7). New functions and rules arrive as a new version, and the chain runs every frozen version it has ever accepted.
- The fuel schedule is versioned separately and applies to all modules (9.2).
- A module may target only one ABI version.
- Modules of different versions call each other freely; the call semantics of section 5 apply whichever version each side targets. Arguments and results are bytes the chain does not interpret (5.4), and interfaces carry no types (4.1), so the declared types are versioned with the interpretation layer (08-value-interpretation.md section 8), not here, and never limit a call.

12. Changes to the earlier documents

Adopting this proposal changes the following. None of the changes is made yet.

1. **Hashes become host functions.** 03-architecture.md section 2 compiles them into each module. Host wins on three counts: one audited implementation instead of one per compiler, a native-speed price for Merkle and SPV proofs that are central to sBTC, and every other source language needs the same functions (G2). The cost is a chain upgrade to speed up a hash, which the fuel schedule already covers.
2. **Exports have one Wasm type.** 03-architecture.md section 3 requires "matching Wasm signatures". Under section 3.3 every entry is `[] -> []` and the check is that the export exists.
3. **Storage is an untyped key-value store** (4.2). The Clarity compiler lays out `define-data-var` and `define-map` in it by the declared layout of 08-value-interpretation.md section 7.2 and declares every variable and map in `abi.types`, so tools can read a Clarity contract's state; the Clarity spec must fix the prefixes it uses. 03-architecture.md section 4 lists data-var and map host functions; they become one key-value store.
4. **A fungible token's cap is set by a host call during `init`** (6.2).
5. **Events are declared and typed** (4.1, 6.2). Legacy `print` emits any value, in SIP-005 encoding, with nothing to say which shape of event it is, so indexers match on conventions such as a `topic` field. Under this ABI a contract names its events in its interface and declares each one's data type in `abi.types`, like Solidity's event ABI, and `emit` takes the event's index and the data's bytes. The Clarity compiler encodes the data canonically, so a Clarity event's data has the same bytes legacy `print` emits for the same value, and one decoder serves both. Clarity replaces `print` with `define-event` and `emit` (spec sections 2.4.1 and 3.10), and its debugging use with `log`, whose output the chain discards (6.2).
6. **NFT identifiers are bytes** (6.2). The identifier type of `define-non-fungible-token` stays in the Clarity spec and in the `abi.types` section, for tools, but leaves the interface the chain reads. The Clarity compiler encodes identifiers canonically (08-value-interpretation.md section 3), so a Clarity token's identifiers have the same bytes as in legacy and in transaction post-conditions.

7. **Modules validate values; the chain does not** (5.4). The earlier documents have the chain validate every argument and result: 02-goals.md (the guarantees list), 03-architecture.md sections 3, 4, 5, and 7, 04-spec-plan.md (the ABI chapter and the ABI vectors), and spec sections 2.3.4, 2.4.4, and 2.5.8. Under this ABI the Clarity compiler emits argument validation in every function it exports and result validation after every contract call, and the spec defines the abort codes for a failure, which replace the chain classes `bad-arguments`, `trait-mismatch`, and `bad-result`.
8. **Contract-call transactions carry their arguments as one byte string** (5.2). The legacy payload is a count and a list of encoded values, which the node decodes. Under this ABI the node passes the bytes to the entry point without parsing them, so the transaction format needs a payload that carries them opaquely. Wallets still build the bytes from canonical encodings (08-value-interpretation.md section 6.1).

9. **The chain has no traits** (08-value-interpretation.md section 5.2). Trait definitions, `impl-trait` claims, and trait-typed parameters leave the interface the chain reads, which 03-architecture.md section 3 lists, and the chain checks conformance nowhere except in the legacy bridge (section 13, question 4). They move to `abi.types`: a compiler reads a trait's definition from the defining contract's section, checks conformance at compile time by the rule of 08-value-interpretation.md section 5.2, and validates the result of every call through a trait against the trait's declared result type (spec sections 2.4.4 and 2.5.8). A call to a function the trait marks `read-only` sets `call`'s `read_only` flag (5.3), so the host enforces the kind; the full signature is checked only at compile time. Spec section 2.3.2 refers to "chain ABI trait rules" for when a contract implements a trait; that becomes the same rule.

10. **Typed signatures leave the chain's interface** (4.1). The earlier documents put each function's parameter and result types in interface metadata that the chain validates: 02-goals.md (the single ABI), 03-architecture.md section 3, and 04-spec-plan.md (the ABI chapter). Under this ABI they are in `abi.types`, defined by 08-value-interpretation.md, which the chain does not read. Provenance (03-architecture.md section 2) records, for each callee, the hash of both sections, since a build types its calls against `abi.types`.

13. Open questions

1. **Legacy costs in a fuel block.** Legacy transactions are costed in five dimensions, but a block has one fuel limit. Legacy costs need a defined conversion to fuel, and that conversion should deliberately overcharge legacy execution relative to Wasm, to encourage migration. How much, and whether the premium grows over time, are open.
2. **Wasm profile.** Section 3.1 hand-picks a subset of Wasm. The alternative is to adopt the deterministic profile of the Wasm specification, which would admit floating point and other features whose results it pins down. Open are which features that admits beyond 3.1, whether wasmi implements the profile exactly, and how each admitted operator is metered.
3. **Limit and price values** in 9.2 and 9.3, from benchmarks on wasmi, including the fuel per block and the storage prices.
4. **Legacy interop.** The mapping from legacy interfaces (03-architecture.md section 7) produces both an interface (4.1) and typed signatures (08-value-interpretation.md section 5), and the type mapping's

known mismatches (extra tuple fields, list union types, trait values carrying types) must each resolve to a rejection or a defined conversion. Legacy traits and `impl-trait` claims map to the `traits` and `implements` of `abi.types`. A call between a legacy and a Wasm contract is the one place the chain must interpret contract values, because the interpreter works on typed values: the host decodes the bytes coming from the Wasm side against the declared types, by canonical decoding (08-value-interpretation.md section 3), and encodes the values coming from the legacy side the same way. That makes the interpretation layer's encoding consensus-critical, and the bridge the one part of this document that depends on it. For a Wasm contract calling a legacy one, the declared types are the legacy callee's own. For a legacy contract calling a Wasm one, they come from the callee's `abi.types`, so the chain must read and validate that section for any contract a legacy contract can call. When that call is through a trait, the bridge also checks, as the interpreter does for a legacy callee, that the callee conforms to the trait by the rule of 08-value-interpretation.md section 5.2. The mapping must also be deterministic and frozen, and produce canonical bytes for both sections: tokens, PoX, and sBTC are legacy contracts, so most Clarity builds read a legacy interface, and provenance records its hashes (02-goals.md G5). If the mapping changed, those builds would no longer reproduce.

5. **Bounded and recoverable calls.** Two possible additions to `call` (5.3), both needed only for callees the caller cannot vouch for, such as a contract passed in as an argument or a module in another language: - A **fuel cap** passed into the callee's frame, as an additional check like an allowance. A callee that exceeds it stops having consumed at most the cap, so the static cost analyzer can bound such a call by its cap instead of the remaining budget (05-roadmap.md risk 7). Without the second addition, exceeding the cap aborts the transaction. - **Catching a runtime error** from the callee: the callee's frame is discarded and the caller receives the error class instead of the transaction aborting. This relaxes "there is no catch" (5.3) and needs its own analysis against R6 and the language's abort semantics (spec 2.7.2).
6. **Formal definition** (R8). The host interface in section 6 should be published as one machine-readable file (WIT or a smaller purpose-made format) from which the host bindings, each compiler's import declarations, and the ABI vectors' function list are generated.
7. **Converting values in Wasm.** Every entry point decodes its arguments from canonical encoding into the module's memory layout, validating them on the way (5.4), and encodes its result back; every caller does the same for the result of each call. Whether that is fast enough as compiled Wasm, or whether it needs host functions that convert between canonical encoding and a defined memory layout, is open. Such functions would put one memory layout into the ABI, against R1, so the related question is whether canonical encoding can be brought closer to a natural Wasm layout (fixed-width little-endian integers, aligned fields) so that conversion is cheap or unnecessary, at the cost of SIP-005 compatibility (08-value-interpretation.md section 3).

Value Interpretation Proposal

This is the interpretation layer of the chain ABI, the companion to 07-chain-abi.md, which is the runtime layer. The runtime layer passes a contract's arguments, results, storage, and event data through as bytes and never interprets them (chain ABI R1). This document says what those bytes mean: the types a contract declares, how values of those types are encoded, and what compilers, wallets, explorers, and indexers may rely on.

The chain enforces nothing here, with one exception: a call between a legacy and a Wasm contract converts values by the encoding of section 3 (chain ABI 13, question 4). A module that ignores this document deploys and runs, but tools can then describe its functions only as taking and returning bytes. References of the form "chain ABI 5.2" are to sections of 07-chain-abi.md.

1. The two layers

Value	Chain ABI (runtime)	This document
Interface	reads function names and kinds, and token and event names (4.1)	typed signatures and traits (5)
Arguments	passes the bytes to the callee unchanged (5.2)	their types (5) and layout (6.1, 6.4)
Results	passes the bytes and the result flag; the flag closes a public frame (5.2)	their types (5) and encoding (6.2, 6.4)
Storage	stores bytes under byte keys (4.2)	its declared layout (7.2)
Event data	records the bytes with the contract and event name (6.2, 8.3)	their declared type (7.1)
NFT identifiers	stores and compares bytes (6.2)	their declared type (7.3)
Host values: principals, `u128` words, and raw bytes	defines them and acts on them (2)	nothing

The host's one structured value is a principal, and its encoding (chain ABI 2.2) is the same bytes as a `principal` value here, so a principal reads the same in both layers.

2. Types

```
type ::= int | uint | bool | principal | never
       | (buff N) | (string-ascii N) | (string-utf8 N)
       | (list N type) | (optional type) | (response type type)
       | { name: type, ... }
       | (trait trait-ref)
trait-ref ::= contract-principal "." name
```

Type	Values
`int`	signed 128-bit integers
`uint`	unsigned 128-bit integers
`bool`	`true`, `false`
`principal`	standard or contract principals
`never`	none
`(buff N)`	byte sequences of length at most `N`
`(string-ascii N)`	at most `N` bytes from the ASCII set of section 3
`(string-utf8 N)`	UTF-8 strings of at most `N` code points
`(list N T)`	sequences of at most `N` values of `T`
`(optional T)`	`none`, or `(some v)` with `v` of `T`
`(response T E)`	`(ok v)` with `v` of `T`, or `(err e)` with `e` of `E`
`{ f: T, ... }`	records with exactly the listed fields
`(trait X)`	contract principals, meant to conform to trait `X` (5.2)

A contract declares the type of each parameter and result of its functions, of each function of its traits, of each NFT's identifiers, and of each event's data in this grammar (section 5). A source language maps its own types onto it.

- Every type is closed: no type variables, no unknown parts (chain ABI R3).
- `never` may appear anywhere a type may. A position of type `never` can hold nothing, so a function can declare, for example, that it never returns `err`.
- Tuples have exact field sets (chain ABI R2). Zero-field tuples are allowed.
- Parameter and tuple-field names follow the rules for function names (chain ABI 2.1).

A `(trait X)` names trait `X` by its defining contract and its name (5.2). Its values are contract principals; that a value conforms to `X` is the declaring contract's expectation, and nothing checks it at runtime. The chain does not read declared types, and a module cannot read another contract's `abi.types`. A compiler checks conformance where it can see both contracts, at compile time. At runtime a call through a trait relies on the host for what it checks: that the function exists, and, when the caller sets `call`'s `read_only` flag for a function the trait marks `read-only`, that the callee is `read-only` (chain ABI 5.3). The caller validates the result against the trait's declared result type, as it validates any result (chain ABI 5.4).

Subtyping on declared types is the smallest reflexive, transitive relation such that:

- `never <: T` for every `T`.
- `(buff N) <: (buff M)`, `(string-ascii N) <: (string-ascii M)`, and `(string-utf8 N) <: (string-utf8 M)` when `N <= M`.
- `(list N S) <: (list M T)` when `N <= M` and `S <: T`.
- `(optional S) <: (optional T)` when `S <: T`.
- `(response S1 S2) <: (response T1 T2)` when `S1 <: T1` and `S2 <: T2`.

- `{ f1: S1, ... } <: { f1: T1, ... }` when both have exactly the same field names and `Si <: Ti` for every field.
- `(trait X) <: (trait Y)` when every function of `Y` is a function of `X` with an identical signature (structural sub-trait).
- `(trait X) <: principal`.

It is used for trait conformance (5.2). A language may have more rules for its own source types, such as Clarity's contract literal types (spec 2.3.2), but a declared type is in this grammar, and these rules relate declared types.

3. Canonical encoding

The encoding is SIP-005, byte for byte, for every value that exists in legacy, so values pass between legacy and Wasm contracts and between the chain and existing tooling without translation. What changes from legacy is decoding, which accepts only the canonical form.

Prefix	Value	Payload
`0x00`	`int`	16 bytes, two's complement, big-endian
`0x01`	`uint`	16 bytes, big-endian
`0x02`	`buff`	u32 BE length, bytes
`0x03`	`true`	
`0x04`	`false`	
`0x05`	standard principal	as chain ABI 2.2
`0x06`	contract principal	as chain ABI 2.2
`0x07`	`(ok v)`	encoding of `v`
`0x08`	`(err v)`	encoding of `v`
`0x09`	`none`	
`0x0A`	`(some v)`	encoding of `v`
`0x0B`	list	u32 BE element count, elements
`0x0C`	tuple	u32 BE field count, then per field: u8 name length, name, value
`0x0D`	`string-ascii`	u32 BE byte length, bytes
`0x0E`	`string-utf8`	u32 BE byte length, UTF-8 bytes

Canonical decoding of bytes `b` against type `T` succeeds if and only if `b` is exactly the encoding of some value of `T`:

- the prefix is one that `T` admits; `principal` admits `0x05` and `0x06`, and `(trait X)` admits only `0x06`;
- lengths and counts are within `T`'s bounds; `string-utf8` bounds count code points;

- tuple fields are exactly `T`'s fields, in strictly ascending byte order of name, with no duplicates;
- `string-ascii` bytes are 0x20–0x7E, tab, line feed, or carriage return (legacy also admits form feed), and `string-utf8` bytes are valid UTF-8 (no surrogates, no overlong forms);
- principals decode as chain ABI 2.2 requires;
- nesting depth is at most 32, counted the same way everywhere;
- no bytes remain after the value.

There is no sanitizing and no untyped decoding. Self-describing prefixes are kept for tooling, but every decode that a contract or the chain relies on is against a type. A tool may still read a value by its prefixes alone to show it, as for `log` output (spec 3.10); such a reading is for display and admits no value into a contract.

4. Sizes and type encoding

4.1 Sizes

`serialized-size(T)` is the largest canonical encoding of any value of `T`:

Type	`serialized-size`
`int`, `uint`	17
`bool`	1
`principal`, `(trait X)`	63 (1 + 1 + 20 + 1 + 40)
`never`	0
`(buff N)`, `(string-ascii N)`	5 + N
`(string-utf8 N)`	5 + 4N
`(list N T)`	5 + N × size(T)
`(optional T)`	1 + size(T)
`(response T E)`	1 + max(size(T), size(E))
`{ f1: T1, ... }`	5 + Σ (1 + len(fi) + size(Ti))

The **value size limit**, 1 MiB, is a limit on `serialized-size`. It matches the chain's byte limits on arguments, results, event data, and NFT identifiers (chain ABI 9.3), so every value of a declared type fits them. Legacy measures a separate abstract "size" that disagrees with the bytes (a principal is 148 there and up to 151 on the wire); this layer has one measure.

A caller can size its buffer for a callee's result from `serialized-size` of the declared result type (chain ABI 5.3).

4.2 Type encoding

Types are encoded in binary, reusing the value prefixes:

Tag	Type	Payload
`0x00`	`int`	
`0x01`	`uint`	
`0x02`	`(buff N)`	u32 BE `N`
`0x03`	`bool`	
`0x05`	`principal`	
`0x07`	`(response T E)`	`T`, `E`
`0x0A`	`(optional T)`	`T`
`0x0B`	`(list N T)`	u32 BE `N`, `T`
`0x0C`	tuple	u32 BE count, per field: name, type; sorted
`0x0D`	`(string-ascii N)`	u32 BE `N`
`0x0E`	`(string-utf8 N)`	u32 BE `N`
`0x11`	`never`	
`0x12`	`(trait X)`	trait reference

Names are a u8 length and the bytes. A trait reference is the defining contract's principal (chain ABI 2.2, prefix `0x06`), then the trait's name. Like a value, a type has exactly one encoding.

5. The `abi.types` section

5.1 Contents

A module declares its types in the `abi.types` custom section (chain ABI 3.4), one canonically encoded record:

Field	Contents
`version`	the version of this format (section 8)
`functions`	per function: name, parameters (name, type), result type
`nfts`	per non-fungible token: name, identifier type
`events`	per event: name, data type (7.1)
`traits`	per trait the contract defines: name, then per function: name, kind, parameter types, result type
`implements`	the trait references of the traits the contract claims to conform to
`storage`	per variable or map (not per value): name, kind, key prefix, key type (maps only), value type (7.2)

Rules:

- Types are in the encoding of 4.2. Names are unique within each list, and lists are sorted by name, or `implements` by encoded bytes, so the section has one hash: `sha512/256` of its bytes. Provenance records it for each callee, alongside the hash of the callee's `abi.interface` (03-architecture.md section 2).
- `functions`, `nfts`, and `events` name functions, NFTs, and events of the module's `abi.interface` (chain ABI 4.1), not necessarily all of them. A function, event, or NFT left out is undeclared (6.4, 7.1, 7.3).
- A `public` function's result type is a `response`, whose arms describe the result for each value of the flag that `result_set` takes (chain ABI 5.2). A `read-only` function's result type is any type. The same holds for the functions of a trait, by the kind the trait gives them.
- At most 256 parameters per function, in `functions` and in traits, as legacy.
- No declared type's `serialized-size` exceeds the value size limit (4.1), and no type nests deeper than 32.
- `storage` follows the rules of 7.2.

5.2 Traits

A trait is a named set of function signatures: per function, a name, a kind (`public` or `read-only`), parameter types, and a result type. A contract defines traits in `traits`, and a declared type names one with `(trait X)`, where `X` is the defining contract's principal and the trait's name. A contract can name its own traits.

A contract **conforms** to trait `X` if, for every function `f` of `X`, its `abi.types` declares a function `f` with the same number of parameters, such that:

- if `X` marks `f` `read-only`, the contract's `f` is `read-only` (chain ABI 4.1); otherwise it may be either kind;
- each of `X`'s parameter types is a subtype (section 2) of the contract's parameter type in the same position;
- the contract's result type is a subtype of `X`'s.

This is the rule of Clarity's `impl-trait` (spec 2.4.1), stated over declared types so that every language and tool applies the same one. Conformance is structural: `implements` is a claim that a tool can check against this rule, not a condition of conforming.

A compiler reads a trait's definition from the defining contract's `abi.types`, and provenance records that section's hash (5.1), so a build that uses another contract's trait reproduces. A wallet can label a trait-typed argument with its trait and check that the principal it names conforms.

5.3 Validity and trust

Outside the legacy bridge (chain ABI 13, question 4), the chain checks only that the section fits the module size limit; it does not decode it. A section that breaks the rules of 5.1 is malformed, and tools treat it as absent: every function, event, and NFT of a module without a usable section is undeclared. A compiler refuses to compile a typed call to an undeclared function, but an untyped call (6.4) needs no declaration.

Nothing ties a declaration to what the module does. The chain does not check values against declared types (chain ABI 5.4), so a module can declare one type and accept or return another, or declare a function that conforms to a trait in signature but not in behavior. A declaration is the author's claim, as a source file is; provenance and `clarity-verify` (03-architecture.md section 2) are how a tool checks that a module was compiled from a source that declares those types.

6. Arguments and results

Sections 6.1 to 6.3 apply to declared functions, and 6.4 to undeclared ones.

6.1 Arguments

A caller passes a declared function's arguments as the canonical encoding of each argument against its declared parameter type, concatenated in parameter order. There is no count and no framing: a decoder walks the parameter list, decoding each value where the previous one ended, and the arguments are well-formed only if every value decodes and no bytes remain.

This is the layout of a contract-call transaction's argument bytes (chain ABI 12, item 8), of a read-only query's, and of the `args` of a `call` (chain ABI 5.3).

6.2 Results

For a declared `public` function, the result bytes are the canonical encoding of a value of the arm of the declared `response` that the result flag selects: `T` for ok, `E` for err. A tool shows them as `(ok v)` or `(err v)`. For a `read-only` function, they are the canonical encoding of a value of the declared result type.

6.3 Who relies on this

Wallets encode arguments and clear-sign transactions from the declared types, explorers decode results with them, and compilers type calls against them.

Because the chain does not check values (chain ABI 5.4), a tool can meet bytes that do not decode against their declared type: the arguments of a transaction that then failed validation in the module, or the result of a module that does not follow its declaration. A tool decodes with canonical decoding against the declared type and, where that fails, shows the raw bytes as such rather than presenting a guess as a typed value.

6.4 Undeclared functions

A function with no entry in `functions` is **undeclared**. Its arguments are one raw byte string and its result is a raw byte string, with the result flag as for any function (chain ABI 5.2). Neither carries a prefix or a length header, and the callee's language decides what the bytes mean.

A caller passes the bytes it has as `args` and reads the result as bytes, bounding its length itself (chain ABI 5.3). A typed language exposes this as an untyped call whose argument and result are byte buffers, which

its compiler cannot check against the callee. A wallet can send a call to an undeclared function built from raw bytes but cannot clear-sign it, and an explorer shows its arguments and result as hex.

This differs from declaring `(buff N)`. A declared `(buff N)` parameter or result is canonically encoded, with its prefix and length (section 3), and gives tools a signature to show. A language with no use for structured values can do either; leaving its functions undeclared costs it clear-signing.

7. Events, storage, and identifiers

7.1 Events

The chain records a contract event as the emitting contract, the event's name, and its data bytes, without interpreting the data (chain ABI 6.2, 8.3). The data bytes are the canonical encoding of a value of the event's declared data type (section 5), so an indexer selects events by contract and name and decodes each one against its declared type, as it decodes a result. The Clarity compiler encodes event data this way (chain ABI 12, item 5).

Legacy `print` events carry the printed value in the same encoding, so one decoder serves both. The difference is that a legacy event has no name and no declared type: a tool decodes it by its self-describing prefixes and matches its shape by convention.

An event the interface names but `abi.types` does not declare is undeclared: its data are raw bytes, which a tool shows as hex. A language that wants free-form text events declares the event's type as a `(string-utf8 N)`, and a tool shows the text.

7.2 Storage

Contract storage is an untyped key-value store (chain ABI 4.2), so state is not self-describing: an explorer, a wallet, or the node's state read endpoint returns raw bytes. The `storage` field of `abi.types` (5.1) makes it readable without knowing the contract's language. It holds one declaration per variable or map. Each declaration has a key prefix, a non-empty byte string the language chooses, and fixes where its values live in the store:

Declaration	Store key	Store value
variable	the prefix	canonical encoding of the value
map	per entry: the prefix, then the canonical encoding of the key	canonical encoding of the entry's value

A map has an entry for a key exactly when the store has a value under the store key for that key; a missing entry is the absence of that store key, not a stored marker. Rules:

- No declaration's prefix is a prefix of another's, so every store key belongs to at most one declaration.
- The prefix plus the `serialized-size` of the key type fits the storage key limit, and the value type's `serialized-size` fits the storage value limit (chain ABI 9.3).

A store key that belongs to no declaration is undeclared, and a tool shows it and its value as raw bytes. A language whose layout differs from this one leaves its storage undeclared, as it can leave functions undeclared (6.4); the Clarity compiler declares every data variable and map (chain ABI 12, item 3). Like any declaration, `storage` is a claim: nothing stops a module from writing other bytes under a declared key, and a tool that cannot decode a store value shows it raw (6.3).

7.3 NFT identifiers

The chain stores and compares NFT identifiers as bytes (chain ABI 6.2). The identifier bytes are the canonical encoding of a value of the token's declared identifier type (section 5), so a tool displays identifiers by that type, and a post-condition names an identifier by the same bytes. The Clarity compiler encodes identifiers this way, so a Clarity token's identifiers have the same bytes as in legacy (chain ABI 12, item 6). An undeclared NFT's identifiers are raw bytes, which a tool shows as hex.

8. Versioning

The chain does not read the declared types outside the legacy bridge, so this layer is versioned on its own. The `abi.types` section's `version` field names its format; the first is `1`. A new version may add fields or types without a new ABI version (chain ABI 11), and a tool supports every version it may meet. A new type gets a new prefix. The encoding of an existing type never changes: the legacy bridge converts values by it (chain ABI 13, question 4), and so does every existing tool.

9. Open questions

1. **Whether the chain validates `abi.types` for every contract.** The legacy bridge already needs the section validated for any contract a legacy contract can call (chain ABI 13, question 4), and any Wasm contract can be passed to a legacy contract as a trait-typed argument. Validating every section at deploy would let every tool trust that it is well-formed and matches the interface, instead of each checking for itself. Against it: deploy gains a type decoder for a section the chain otherwise ignores, and a valid declaration still says nothing about the values a module accepts (section 5).
2. **Indexed event fields.** Solidity lets an event mark up to three parameters `indexed`, which the chain records as separate topics that nodes can filter on without decoding the data. Here an observer can select by contract and event name (7.1) but must decode the data to filter on a field, such as every transfer to one recipient. A declaration could mark fields of a tuple data type as indexed, and `emit` could take their encodings as separate byte strings that the chain records and indexes. Open are whether node-side filtering is worth a second argument to `emit` and a larger receipt, and how many fields an event may index.

Clarity vNEXT Specification (working draft)

This directory holds the working draft of the specification for the new Clarity language (working name **vNEXT**), which compiles to Wasm for the new chain. It follows the scope decision in ../04-spec-plan.md: Clarity 1–6 are not specified here, and the chain ABI that every source language shares is specified separately, in ../07-chain-abi.md and ../08-value-interpretation.md, which this spec refers to.

> [!IMPORTANT] Everything in this directory is a draft intended to seed the RFC process. Nothing here is normative yet. A chapter with open questions lists them at the end; an open question tagged `RFC` needs its own design RFC before the chapter can be frozen.

The two-layer design

vNEXT is specified as **a small core language plus a standard library written in that core language**. This is the single most important structural decision in the spec, and every chapter is organized around it:

Layer	Contents	Where specified
Core language	Syntax, types, a fixed set of *special forms* (binding, branching, early exit, definitions, constructors, tuple access, type ascription, calls, context scoping) and a fixed set of *primitive operations* (host access, cryptography, and the base operations on each type's representation)	02-core-language.md, 03-primitives.md
Standard library	Every other construct a contract author uses: `if`, `unwrap!`, `try!`, `asserts!`, `default-to`, `max`, `pow`, `append`, `index-of?`, `map-insert?`, integer/string conversions, `verify-merkle-proof`, and so on. Written in vNEXT and linked into every module by the compiler	04-standard-library.md, std/prelude.clar

The layering rule

A construct belongs in the core **only if** it passes at least one of these tests; otherwise it belongs in the standard library:

1. **Host access.** It reads or writes contract state, moves assets, reads chain data, reads the transaction context, emits events, or calls another contract. (Examples: `map-get?`, `ft-transfer?`, `get-current-context`, `emit`, `contract-call?`.)
2. **Not expressible.** It cannot be written in the language at all because it introduces binding, branching, early exit, definitions, generics, or type-directed behavior, takes a type or size argument, or needs a function argument that the language cannot close over. (Examples: `let`, `match`, `return`, `define-map`, `is-eq`, `as-max-len?`, `from-consensus-buff?`, `uint-to-buff-be`, `map`, `filter`.)
3. **Cryptographic standard.** It is an operation that an external standard defines for a hash function or signature scheme: hashing, signature verification, or public-key operations such as recovery and decompression. These are implemented once, by audited host libraries, not reimplemented in the language, even where one operation of a scheme could be derived from another. Constructions built

from them are library code. (Examples: `sha256`, `secp256k1-verify`, `secp256k1-recover?`, `secp256k1-decompress?`; but `hash160`, `verify-merkle-proof`, and `principal-of?` are library.)

4. **Representation base.** It is part of a built-in type's ordinary operator vocabulary: the integer arithmetic operators `+ - * / mod`, the bitwise operators (`bit-and`, `bit-or`, `bit-xor`, `bit-not`, and the shifts), the comparisons (`<`, `<=`, `>`, `>=`), the base conversions (`to-int`, `to-uint`, `buff-to-uint-be`), and sequence length, concatenation, indexing, and slicing. Named functions over that vocabulary are library (`max`, `abs`, `pow`, `sqrti`, `buff-to-int-le`, `int-to-ascii`). The list is a decision, not a derivation: some members can be written in terms of others, and the choice assumes nothing about how a compiler represents values or what an operation costs. Adding to it is a core change. (The `*-to-buff-*` family is core by test 2, not this one: each takes a size argument.)

Two primitives fail every test and stay core only until their library definitions are written (chapter 3.12).

Cost is not a test. Contracts are metered by the fuel their compiled Wasm consumes, so a library function costs what its code costs to run, and a function that turns out to be hot is optimized by a compiler that links a faster implementation, not by moving it into the core (chapter 4.1).

The standard library has **no special powers**. It is ordinary vNEXT source that uses only features available to every contract. If the library needs a feature (generics, macros, overloading, static references), that feature is a user feature and is specified as such.

Why this shape

- **The spec shrinks where it matters.** The legacy special forms with bespoke lazy-evaluation and typing rules (`if`, `unwrap!`, `try!`, `asserts!`, `default-to`, ...) and the derived arithmetic, sequence, and conversion natives become library definitions whose semantics are *defined by their source*. There is nothing to specify about `unwrap!` beyond the four lines that implement it. The remaining natives become about 70 primitives with uniform typing rules.
- **The language has no cost table.** Execution is metered in Wasm fuel, and the only other charges are the chain ABI's charges for host functions. There is no per-construct cost table to keep in sync with behavior, and nothing about cost to specify for the library.
- **Compilers cannot disagree about the library.** Any compiler that implements the core correctly gets the library for free.
- **Design pressure goes the right way.** Each proposed addition must either pass the layering rule (and pay the full price of a core change) or be written as library code, where it is cheap, reviewable, and provably bounded.

Chapters

Chapter	Status	Maps to plan chapter
01-principles.md	Draft	1 (design principles)
02-core-language.md	Draft	2, 3, 4, 5 (grammar, types, static and dynamic semantics)

Chapter	Status	Maps to plan chapter
03-primitives.md	Draft	7 (native reference, reduced to primitives)
04-standard-library.md	Draft	7 (library), plus the legacy-to-vNEXT disposition table
std/prelude.clar	Draft	The library itself
Compilation to Wasm	Not started	6 (lowering, prelude linking, determinism, bound preservation)
Chain ABI	Proposal, shared by every source language: ../07-chain-abi.md (runtime layer) and ../08-value-interpretation.md (interpretation layer)	ABI (module format, host functions, frames, fuel; declared types and value encoding)
Conformance vectors	Not started	Conformance suite

Conventions

- **MUST**, **MUST NOT**, **SHOULD**, **MAY** carry their RFC 2119 meanings.
- **Rule identifiers.** While a chapter is a draft, a normative rule is identified by the section that states it (for example 2.6.5). When a chapter's conformance vectors are written, each rule in it gains the identifier `<section>-<n>`, numbered in order of appearance within the section, and vectors cite that identifier. Identifiers are assigned at that point and not before, so that a moving draft is not renumbered.
- **Open questions.** Each chapter ends with its open questions. One tagged `RFC` needs its own design RFC before the chapter can be frozen.
- Type signatures use the notation `name : (arg-types) -> result-type`. Uppercase single letters (`T`, `E`, `K`, `V`) are type variables; `N`, `M` are size (length-bound) variables. A signature written "for `I` in {`int`, `uint`}" is a family of signatures, one per listed type.
- Source examples are written in vNEXT syntax, which differs from Clarity 1–6 where noted.
- "Legacy" means Clarity 1–6. It appears only in comparisons and in the porting table of chapter 4.
- "The chain" means any chain implementation of the chain ABI, and "host function" means a function the chain provides to modules through it.

1. Design Principles

These principles are the rejection criteria for every construct in vNEXT, present and future. A proposal that violates one of them is out of scope regardless of its merit on other axes.

P1. Decidability is the invariant

Every vNEXT program terminates, and the worst-case cost of every function is a computable function of its parameter types. Concretely, the language MUST keep all of the following true:

- Every type has a statically known maximum size. Lists, buffers, and strings carry length bounds. There are no unbounded data structures.
- There is no recursion, direct or mutual, and no unbounded loop. Iteration exists only over sequences whose length is bounded by their type.
- Macro expansion and generic instantiation terminate, because the definition graph is acyclic.
- Call depth is bounded, and a contract cannot be re-entered while a call into it is active.

Any construct that cannot be shown to preserve these properties is rejected. This is the property that makes exact static cost bounds (in fuel, P4), complete static analysis, and tractable proofs possible (../01-why-clarity.md).

P2. Minimal core

The core language contains only what passes the layering rule in README.md: host access, constructs that cannot be expressed in the language, hash functions and signature schemes defined by external standards, and the base operations on each type's representation. Everything else is standard library.

The core is therefore small enough to hold in one head: about a dozen top-level definition forms, about a dozen expression forms, and about seventy primitive operations with uniform typing rules. The library holds everything else a contract author uses, about thirty-five definitions.

P3. The standard library has no special powers

The standard library is ordinary vNEXT source. It uses only features that every contract can use, it is type-checked by the same rules, and it is compiled and metered like any other code. Anything the library can do, a contract author can do. If the library needs a language feature, that feature is designed and specified as a user feature.

Corollary: a proposal to "just add a native" for something expressible in the language is rejected, unless it belongs in a type's operator vocabulary under the layering rule's representation-base test, which is a core change judged on that ground alone. Otherwise the right fix is a library function, or if the library cannot express it well, a general language feature that lets it. Speed is never a reason to add a native; see P4.

P4. Cost is not a language concern

The language assigns no cost to any construct. Execution is metered in the fuel the chain's Wasm engine consumes running a contract's module, plus the chain ABI's charges for host functions (storage I/O, asset operations, chain-data lookups, contract calls, cryptography). Both belong to the chain ABI, not to this specification.

Consequences:

- There is no per-construct cost table, so there is no class of bugs where a cost table disagrees with behavior, and no cost chapter that grows with the language.
- A library function costs the fuel of its compiled code, like user code.
- The static bound still exists. By P1, every loop and every value is bounded by its type, so the worst-case fuel of every function is computable from the source, its compiled module, and the chain's charge table (chapter 6 of the plan carries the argument that compilation preserves the bounds).

P5. The deployed Wasm is law; the source is its specification

The chain stores and executes Wasm modules and never sees Clarity source. What a deployed contract does is what its module does.

This specification defines what a module compiled from a given source MUST do: its results, state changes, events, and aborts. A compiler that produces a module whose behavior differs from the source's meaning has a bug. The language keeps its guarantees on chain only through compilers that meet this spec, which is why compilers are held to the conformance suite and to deterministic, verifiable builds (../03-architecture.md).

The standard library is versioned with the language and frozen when the language version is frozen; its source defines its results. The compiler links the library code a contract uses into that contract's module. A later compiler MAY link a faster implementation, which MUST produce identical results; contracts already deployed keep the code they were deployed with. A contract never observes a library change.

P6. One way to do each thing

- No aliases. Legacy pairs such as `element-at` / `element-at?`, `index-of` / `index-of?`, and `xor` / `bit-xor` collapse to one name.
- No overlapping forms. `as-contract` is gone; `as-contract?` remains. `stx-transfer-memo?` is folded into `stx-transfer?`. `(tuple ...)` is gone; `{...}` remains. `begin` is sugar for `let` with no bindings.
- Naming discipline, applied uniformly to core and library:
- `name?` returns an `optional` or a `response`.
- `name!` may return early from the enclosing function.
- `is-name` is a predicate returning `bool`.

- Names are lowercase words separated by `-`. Tuple fields follow the same rule (`error-code`, not `error_code`).
- Declared arity. Every call's arity is fixed by its definition: exactly the declared parameters, or, for a definition ending in a rest parameter `(name T...)`, the fixed parameters plus one or more arguments collected into a list (2.6.5). `(+ a b c)` works because `+` has a rest parameter, not because `+` is special; a user function can have one the same way. There are no optional parameters and no per-arity typing rules.

P7. No undefined behavior, no silent failure

- Arithmetic overflow, division by zero, out-of-range conversions, and invalid shift amounts abort the transaction deterministically.
- Errors are values (`response`, `optional`) or aborts. There is no third kind of outcome, and a `response` cannot be discarded without handling it.
- There are no null values, no implicit conversions, and no reentrancy.

P8. Spec first, then implementation

The reference compiler implements this specification. Where they disagree, one of them has a bug, and the disagreement is adjudicated explicitly and recorded. Every normative rule cites the conformance vectors that exercise it (../04-spec-plan.md describes the suite), and no rule is frozen without vectors. Rules are given identifiers for that purpose when their chapter's vectors are written (README.md, conventions).

What vNEXT is not

- **Not backwards compatible.** vNEXT contracts are written for vNEXT. Chapter 4's disposition table maps every Clarity 1–6 construct for porting.
- **Not Turing-complete.** See P1.
- **Not general-purpose.** No I/O, no foreign functions, no off-chain execution.
- **Not privileged.** Clarity is the first language on the chain, not a special one. Everything a Clarity contract does on chain goes through the same chain ABI as any other module.

Footguns removed

This list is the concrete payoff of having no compatibility obligation. Each item is a legacy behavior that vNEXT does not have.

Legacy behavior	vNEXT
`as-contract` grants unrestricted asset access	Only `as-contract?` with explicit allowances
`at-block` (already disabled in legacy at epoch 3.4)	Absent

Legacy behavior	vNEXT
`bit-shift-left` / `bit-shift-right` silently reduce the shift amount modulo 128	Shift amount of 128 or more aborts
Variadic `is-eq`, `and`, `or`, `+`, ... with subtly different typing per arity	Arity is declared: a rest parameter collects extra arguments into a list, typed by the ordinary rules; `and` / `or` are core short-circuit forms; `is-eq` and comparisons are binary only
Hashing an `int` / `uint` hashes an unspecified 16-byte encoding	Hash primitives accept buffers only
`buff-to-int-be` / `buff-to-int-le` zero-pad a short buffer, so `0xff` is `255` but the same bytes in a 16-byte buffer are `-1`	Signed conversions sign-extend from the buffer's actual length; `(buff-to-int-be 0xff)` is `-1`
Trait types written `<name>` require `use-trait` aliasing with a distinct lexical form	Trait types are ordinary type expressions `(trait id)`; aliases use `define-type`
`principal-construct?` returns a tuple with an `error_code` field	`error-code`
Two ways to write tuples, two names for several functions	One way
`(with-all-assets-unsafe)` accepted at runtime inside `restrict-assets?` but rejected by the type checker	Rejected by the grammar; valid only in `as-contract?`, and only as the sole allowance
`secp256r1-verify` accepts high-S (malleable) signatures	All ECDSA primitives require low-S
Tuple lub admits a smaller tuple for a larger one, creating hidden fields that must be sanitized at boundaries	Tuple subtyping requires identical field sets
Function arguments may shadow constants, and the checker and evaluator resolve them differently	No shadowing of any in-scope name
`get-tenure-info? block-reward` aborts the transaction during the maturity gap	Returns `none`; maturity defined by coinbase height
Any chain-data lookup the host cannot serve makes the transaction "problematic" and unmineable	All chain-data lookups are total and return `none`
Binding a reserved keyword as a local name passes analysis and fails at runtime	No shadowing, checked at analysis time
A trait defined in the current contract cannot be used in the contract's own top-level expressions	Local definitions are fully usable at the top level
Duplicate parameter names are accepted	Static error
A read-only function cannot call a trait function even when every implementation is read-only	Traits may mark functions `read-only`; read-only callers may call those

Legacy behavior	vNEXT
Constants cannot be used where a literal is required (`as-max-len?` bounds, `contract-call?` targets)	Static constants are usable anywhere a literal is
Trait-typed values cannot be extracted from tuples and called directly	Any trait-typed expression is a valid `contract-call?` target
`is-standard` is true for a contract principal on the current network and false for a standard principal on the other network	`is-current-network` for the network test; `is-standard` is true exactly for standard principals
`print` emits a value of any shape, so indexers recognize events by convention (a `topic` field)	Events are declared with `define-event`: each has a name and a data type, and indexers select by name and decode by type

2. Core Language

This chapter defines the vNEXT core: lexical structure, types, top-level forms, expression forms, functions, and dynamic semantics. Primitive operations are enumerated in 03-primitives.md; everything else a contract author uses is defined in the standard library (04-standard-library.md).

The presentation here is semi-formal. Typing rules are given as prose with signatures. The formal judgment-style rules and the full EBNF are follow-up work once the design questions at the end of this chapter are settled.

2.1 Notation

- `name : (A B) -> R` is a signature: `name` takes an `A` and a `B` and produces an `R`.
- `T`, `E`, `K`, `V`, `A`, `U` are type variables. `N`, `M` are size variables (natural numbers bounding sequence lengths).
- `S <: T` means `S` is a subtype of `T` (2.3.2). `lub(S, T)` is their least upper bound when it exists.
- `T[σ]` is `T` with the substitution `σ` applied: each type or size variable in `T` is replaced by the type or size `σ` assigns it (2.6.2).
- `never` is the empty type (2.3.3).

2.2 Lexical structure

vNEXT source is a sequence of S-expressions.

- **Comments** begin with `;;` and run to end of line.
- **Identifiers** are non-empty sequences of ASCII letters, digits, and the characters `- _ ! ? + < > = / * ^`, not starting with a digit, at most 128 bytes long. Identifiers are case-sensitive.
- **Integer literals**: `uint` literals are `u` followed by decimal digits (`u42`); `int` literals are optional `-` followed by decimal digits (`42`, `-7`). Values outside the 128-bit range are a parse error.
- **Boolean literals**: `true`, `false`.
- **Buffer literals**: `0x` followed by an even number of hex digits (`0x`, `0xdeadbeef`).
- **ASCII string literals**: `"..."` with escapes `\\`, `\"`, `\n`, `\t`, `\r`. The **`string-ascii` character set** is the printable ASCII characters 0x20 through 0x7E plus tab (0x09), line feed (0x0A), and carriage return (0x0D). A literal may contain only these characters, the three control characters only through their escapes, and no value of a `string-ascii` type ever holds any other byte.
- **UTF-8 string literals**: `u"..."` with the same escapes plus `\u{XXXX}` code point escapes.
- **Principal literals**: standard principals `'SP...` / `'ST...` (c32check encoded); contract principals `'SP....name` or the deployer-relative `.name`; trait identifiers `'SP....contract.trait`, `.contract.trait`, or a bare local trait `name` where a trait identifier is expected. A contract name is 1 to 40 characters, starts with an ASCII letter, and continues with letters, digits, `-`, or `_`. The same limit applies wherever a contract name appears, including `principal-to-ascii` (3.5) and the library's

`principal-construct?` and `principal-destruct?`. There are no reserved or special-cased contract names (legacy admitted `__transient`).

- **Tuple literal syntax**: `{ name: expr, ... }`. This is the only tuple constructor syntax.
- `none` is the literal of type `(optional never)`.

Parentheses delimit lists; `{`, `}`, `:`, `,` are used only by tuple syntax.

Keywords and reserved names. Two kinds of identifier cannot be bound by a contract:

- A **keyword** is an identifier whose meaning is fixed by the grammar. It is never a name of anything and cannot appear where a name is expected: the literals `true`, `false`, `none`; the heads of special forms (2.5) and definition forms (2.4.1); the allowance forms (2.5.9); the pattern and constructor heads `some`, `ok`, `err`, `list`; the type names and reference kinds of 2.3.1 and 2.3.7; and the `read-only` marker in trait definitions. Type names are keywords in value position too: `(let ((int u5)) ...)` is a static error.
- A **reserved name** is a definition that is in scope in every contract: every primitive (03-primitives.md) and every prelude definition (chapter 4). It behaves like a top-level definition of the contract itself and so cannot be rebound (2.5.2). Reserved names are fixed per language version, because the prelude is part of the language version.

2.3 Types

2.3.1 Type grammar

```
type      ::= int | uint | bool | principal | never
            | (buff size) | (string-ascii size) | (string-utf8 size)
            | (list size type)
            | (optional type)
            | (response type type)
            | { name: type, ... }              ;; tuple; zero or more fields
            | (trait trait-identifier)
            | alias-name                       ;; introduced by define-type
            | type-variable                    ;; see 2.3.5
            | contract-literal-type            ;; internal; see below

size      ::= uint-literal | size-variable | static-constant-name
```

Meaning of each type:

Type	Values
`int`	signed 128-bit integers
`uint`	unsigned 128-bit integers
`bool`	`true`, `false`
`principal`	standard or contract principals
`never`	no values. The type of `return` and `panic`, and the side of `ok`, `err`, `none`, and `(list)` that no value reaches (2.3.3)

Type	Values
`(buff N)`	byte sequences of length at most `N`
`(string-ascii N)`	strings of at most `N` bytes from the `string-ascii` character set (2.2)
`(string-utf8 N)`	UTF-8 strings of at most `N` code points
`(list N T)`	sequences of at most `N` values of type `T`
`(optional T)`	`none` or `(some v)` with `v : T`
`(response T E)`	`(ok v)` with `v : T` or `(err e)` with `e : E`
`{ f: T, ... }`	records with exactly the listed fields
`(trait X)`	contract principals that conform to trait `X` (2.3.2); checked at compile time, not at runtime (2.5.8)
contract literal	the one contract principal named by a literal `'SP...123.c` or `.c`, or by a static constant holding one (2.4.1). Internal: it cannot be written in source, is never an interface or storable type, and exists so that a literal can be used wherever a trait type is expected (2.3.2) and as a `contract-call?` target (2.5.8)

`(buff N)`, `(string-ascii N)`, `(string-utf8 N)`, and `(list N T)` are the four **sequence types**. Their **element types** are `(buff 1)`, `(string-ascii 1)`, `(string-utf8 1)`, and `T` respectively.

Every type has a statically known maximum serialized size, `serialized-size` of the interpretation layer (../08-value-interpretation.md section 4.1). Two limits (2.8) apply to it, depending on where the type occurs:

- The **value size limit** applies to the type of every name (a `let` or `match` binding, or a parameter of a function, not of a macro), every function result, every type written in source, and every value written to storage, emitted in an event, or passed to `contract-call?`.
- The **intermediate size limit** applies to the type of every other expression: a primitive's argument, a list or tuple under construction, or a macro's argument before it is bound. Macro arguments are checked after expansion, at the position where they land.

A type that exceeds the limit for its position is a type error. So no value larger than the value size limit can be bound, returned, or stored, and a value that lives only during the evaluation of one expression is bounded by the intermediate limit. `replace-at?`'s `(concat (concat prefix (list item)) suffix)` has bound `2N + 1` before `as-max-len?` brings it back to `N`, which fits the intermediate limit even when `N` is near the value limit. Type nesting depth MUST NOT exceed the type depth limit.

2.3.2 Subtyping

Subtyping formalizes what legacy Clarity calls "admission". It is the smallest reflexive, transitive relation such that:

- `never <: T` for every `T`.
- `(buff N) <: (buff M)`, `(string-ascii N) <: (string-ascii M)`, `(string-utf8 N) <: (string-utf8 M)` when `N <= M`.
- `(list N S) <: (list M T)` when `N <= M` and `S <: T`.
- `(optional S) <: (optional T)` when `S <: T`.
- `(response S1 S2) <: (response T1 T2)` when `S1 <: T1` and `S2 <: T2`.
- `{ f1: S1, ... } <: { f1: T1, ... }` when both have exactly the same field names and `Si <: Ti` for every field.
- The contract literal type of `c` (2.3.1) `<: (trait X)` when `c` conforms to `X`: its declared signatures in `abi.types` (2.4.4) satisfy the rule of `impl-trait` (2.4.1), which is the interpretation layer's conformance rule (../08-value-interpretation.md section 5.2). It is `<: principal` always.
- `(trait X) <: (trait Y)` when every function required by `Y` is present in `X` with an identical signature (structural sub-trait).
- `(trait X) <: principal`.

Two distinct traits that require the same functions with the same signatures are **equivalent**: each is a subtype of the other, but they remain distinct types, named by different trait identifiers. A contract that declares its own copy of a standard trait can therefore pass a value of its copy's type where the standard trait is expected, and the reverse. Subtyping is thus a preorder on trait types rather than a partial order.

`lub(S, T)` is defined pointwise by the same rules (taking the larger bound, the lub of components, and so on) and is undefined when `S` and `T` have different shapes (for example `int` and `uint`, or tuples with different field sets). An undefined lub where one is required is a type error.

`lub(never, T) = T` for every `T`, since `never <: T`. This is how the empty side of a constructor is filled in: `lub((response int never), (response never uint))` is `(response int uint)`, and `lub((list 0 never), (list 1 int))` is `(list 1 int)`.

Among the principal-like types the join is always defined, and it is the most specific type the rules above name, falling back to `principal`:

- `lub(S, S) = S`.
- `lub(S, (trait X)) = (trait X)` when `S <: (trait X)` and `S` is not the type of a trait equivalent to `X`; symmetrically for the other argument.
- Otherwise `lub(S, T) = principal`, including two different contract literals, two trait types neither of which is a sub-trait of the other, two distinct equivalent trait types, and anything with `principal`.

Two distinct equivalent traits join to `principal`, as unrelated traits do, because no single trait identifier is the answer. Either would be a least upper bound, and choosing one would put an arbitrary trait identifier into the contract interface (2.4.4). The author can state the intended trait with `as` (2.5.10): with `x : (trait X)` and `y : (trait Y)`, `(list x (as (trait X) y))` has type `(list 2 (trait X))`.

`lub(T1, ..., Tn)` is taken over the whole set, not as a pairwise fold. Among principal-like types it is the member of the set that every other member is a subtype of, and it is `principal` if no member qualifies or if two distinct equivalent traits both qualify. Other types combine pointwise as above. A pairwise fold would depend on the order of the arguments, because the fallback to `principal` loses information: if trait `p`

requires `f` and `g`, trait `q` requires `f` and `h`, and trait `r` requires only `f`, then with `x : (trait p)`, `y : (trait q)`, and `z : (trait r)`, folding `(list x y z)` would give `lub(principal, (trait r)) = principal`, while folding `(list x z y)` would give `(trait r)`. Over the whole set both have type `(list 3 (trait r))`. Likewise, if traits `a` and `b` are equivalent and trait `c` requires a subset of their functions, then with `x : (trait a)`, `y : (trait b)`, and `z : (trait c)`, `(list x y)` has type `(list 2 principal)`, while `(list x y z)` and `(list x z y)` both have type `(list 3 (trait c))`. As these examples show, adding an element can make the inferred type more specific: the common supertype of `p` and `q` that requires only `f` has no name unless some member names it.

The join of two unrelated trait types is `principal` rather than an anonymous trait of their common functions, because such a trait has no name, could not be written in an interface, and would give a value a type no one declared. A list of contract literals meant as trait values therefore states the trait once: `(list (as (trait X) .a) .b .c)` has type `(list 3 (trait X))` when all three implement `X`.

Two types **meet** when some type that does not mention `never` is a subtype of both. Outside the principal-like types this is exactly the condition under which `lub` is defined: types of the same shape meet, and types of different shapes do not. Among the principal-like types, `principal` meets every one of them, a contract literal meets itself and every trait it conforms to, two different contract literals do not meet, and two trait types meet when no function is required by both with different signatures, since a contract could then implement both. Overloads of a name are required not to meet (2.6.2).

Because tuple subtyping requires identical field sets, every runtime tuple has exactly the fields of its static type. There are no hidden fields, so no value ever needs to be re-shaped ("sanitized") when it crosses a function or contract boundary.

2.3.3 The `never` type

`never` is the type with no values, the bottom of the subtype order (2.3.2). It arises in two ways:

- As the type of an expression that does not produce a value: `(return e)` (2.5.4), `(panic)` (3.10), and the forms built on them.
- As the side of a type that no value reaches: `(ok e)` has type `(response T never)`, `(err e)` has type `(response never E)`, `none` has type `(optional never)`, and `(list)` has type `(list 0 never)` (2.5.6). The lub fills in the other side when an `ok` meets an `err` (2.6.4) or a fold widens its accumulator (2.6.2), and `never` remains only where nothing ever flows. A function with no `err` path therefore has the result type `(response T never)`, which its interface declares (2.3.4, 2.6.4).

`never` may be written in source wherever a type may, and may appear in storable and interface types (2.3.4). A position of type `never` holds nothing, which is harmless, and argument and result validation (2.5.8) admits no value for it.

`never` has no typing rules of its own. It is a type like any other, with `never <: T` for every `T` as its only distinction, and every rule of this chapter applies to it unchanged:

- An expression of type `never` is admitted wherever a value of some type `T` is expected, by subtyping, and the enclosing form has the type its rule gives it: `(+ 1 (return u0))` has type `int`, `(some`

`(panic))` has type `(optional never)`, and `(+ "a" (panic))` is a type error. A name bound at type `never` by `let`, `match`, or a parameter is an ordinary name of that type.

- A form that selects something by the type of an operand requires that operand to have a type of the shape it selects on, and `never` has no shape. A `match` scrutinee (2.5.3), an operand of `get` or `merge` (2.5.7), and a trait-typed `contract-call?` target (2.5.8) of type `never` are type errors.
- An argument of type `never` fits every candidate of an overloaded name, so a call that more than one candidate admits is an ambiguity error (2.6.2): `(unwrap! (panic) e)` is one. A type variable that only such an argument would fix is instantiated at `never`, the least type (2.6.2), but a sequence kind (3.3) has no least member, so `(len (panic))` is a type error. The author states the intended type with `as` (2.5.10).
- A type variable may be instantiated at `never`, or at a type that mentions it, and the instance body is type-checked under that substitution like any other (2.6.2). An error in such a body is reported at the call site, as any instance error is, and `as` on the argument that fixed the variable resolves it. If `show` matches on a parameter of type `(optional T)` and calls the overloaded `int-to-ascii` on the payload, `(show none)` is an ambiguity error in the instance at `T = never`, and `(show (as (optional int) none))` is not.
- Every arm, binding, and body expression is type-checked, whether or not it can be reached. `(if (panic) 1 "hello")` is a type error because its arms have no lub, as it would be with any other condition. An arm whose binding has type `never` contributes its type to the `match` like any other arm (2.5.3), and a `return` whose operand has type `never` contributes `never` to the result type (2.6.4), which the lub absorbs.

Unreachable code. When an expression of type `never` is the initializer of a `let` binding, an argument to a function that is not a macro, or a body expression other than the last, the code that follows it cannot be reached. An argument to a macro is exempt, because it is evaluated only where the body references it (2.6.3): `(asserts! ok (panic))` aborts only when `ok` is false.

2.3.4 Storable types and interface types

A type is **storable** if it contains no trait type and no type or size variable. Only storable types may appear in `define-data-var`, `define-map`, `define-non-fungible-token`, and in `from-consensus-buff?`. A storable type may contain `never`; such a position can hold no value, which is harmless.

Excluding trait types keeps every call target traceable to the transaction: a trait-typed value can only come from a contract literal or a public function's argument, so the contracts a transaction can reach are named in its payload or in the code, never in chain state. A registry therefore stores principals and has the caller name the target:

```
(define-map extensions uint principal)

(define-public (run (id uint) (ext (trait extension)))
  (begin
    (asserts! (is-eq (map-get? extensions id) (some ext)) (err u1))
    (contract-call? ext execute)))
```

`(trait extension) <: principal` (2.3.2), so the comparison needs no conversion. The restriction is not needed for termination or cost bounds (a call through a trait parameter already has a statically unknown callee). It is kept because lifting it later is backward compatible and restoring it would not be.

A type is an **interface type** if it contains no type or size variable. Trait types are allowed. Only interface types may appear in the parameter and result types of public and read-only functions, in trait definitions, and in event definitions. They are the types the compiler declares in `abi.types` (2.4.4), and each is a type of the interpretation layer's grammar (../08-value-interpretation.md section 2).

An interface type may contain `never`. A public function whose body is `(ok true)` has the result type `(response bool never)`, which states that the function cannot return `err`. The interpretation layer's grammar includes `never`, and argument and result validation (2.5.8) admits no value for it. An author who wants a wider type in the interface states it with `as` (2.5.10): `(as (response bool uint) (ok true))`.

2.3.5 Type variables and size variables

An identifier in type position that is not a built-in type name, a `define-type` alias, or a trait identifier denotes a **type variable**. An identifier in size position that is not a literal denotes a **size variable**. Both are implicitly universally quantified over the definition in which they appear, and each MUST occur in at least one parameter of that definition, in its type or in its reference kind (2.3.7).

Type and size variables may appear only in `define-private` and `define-macro` definitions. Public and read-only functions, traits, storage definitions, and constants are monomorphic.

> [!NOTE] Size arithmetic (`N + M`, `N + 1`, `K × M`) appears only in primitive signatures in 03-primitives.md. Contract authors never write it, because function result types are inferred, not declared.

A **static constant** (2.4.1) of type `uint` may be used as a size, and a static constant of contract principal type may be used wherever a contract principal literal is required.

2.3.6 Type aliases

`(define-type name type)` introduces a transparent alias. The alias is interchangeable with its definition everywhere. Aliases may not be recursive (2.4.3) and may not contain type or size variables.

This form replaces legacy `use-trait`: `(define-type sip010 (trait 'SP....sip-010-trait.sip-010-trait))`.

2.3.7 Static reference kinds

Some parameters do not hold runtime values but refer statically to a definition in the current contract. These **reference kinds** are not types: they cannot appear in `let` bindings, storage, interfaces, tuples, or sequences.

```
ref-kind  ::= (var type)
            | (map type type)
            | (fungible-token)
            | (non-fungible-token type)
            | (event type)
            | (function (type ...) type)
```

An argument in a reference-kind position MUST be either the name of a definition of the matching kind in the current contract, or a parameter of the same kind in the enclosing function. How the argument's types must relate to the parameter's depends on the kind (2.6.2):

- `(var T)`, `(map K V)`, `(non-fungible-token T)`: identical types, not merely related ones. A variable or map is both read and written, so its types may be neither widened nor narrowed; an NFT's identifier type is matched the same way.
- `(event T)`: the event's declared type may be a supertype of `T`. The reference is only emitted to, and every value that fits `T` fits the type the event declares in `abi.types`.
- `(function (type ...) type)`: by the rule of 2.6.2.
- `(fungible-token)`: no types to match.

References are resolved at instantiation time (2.6.2) and have no runtime representation.

Reference kinds let one helper serve every map (or variable, token, or event) of a given shape; each distinct reference is its own instance, so which storage a function touches stays statically known. Reference kinds may appear only in `define-private` and `define-macro` parameters. `(function ...)` arguments may name any function defined in the current contract or the prelude, but not a macro, or any primitive that is not a special form.

2.4 Contracts and top-level forms

A contract is a version declaration followed by a sequence of top-level forms. A definition may refer to any other definition in the contract, earlier or later, as long as the definition graph is acyclic (2.4.3). Source order matters only for code evaluated at deploy (2.4.2).

Top-level names share one namespace, whatever the kind of definition, including `define-type` aliases and traits, although those are resolved only in type position (2.3.5). Two definitions MUST NOT have the same name, except overloads of a function or of a macro (2.6.2); a name's overloads are all functions or all macros. This means, for example, that a contract cannot have an event and a map both named `listed`, or a constant and a function both named `fee`.

2.4.0 Version declaration

```
(clarity-version version)
```

The first form of every contract MUST be a version declaration naming the language version the contract is written for. There is no default: a contract without one is a syntax error, so a contract is never silently reinterpreted under a later version's prelude or rules. A compiler MUST reject a contract whose declared

version it does not implement, and a compiler that implements several versions uses the declared one for the whole contract. The declared version is what the module's provenance section records as the language version (../03-architecture.md section 2).

Nothing else in the language varies by version. The declaration selects one complete specification, with its prelude and reserved names (2.2), for the whole contract.

2.4.1 Definition forms

Form	Introduces	Rules
`(define-constant name expr)`	A named value	`expr` is deploy-time code (2.4.2), evaluated once; its type is inferred and MUST satisfy the value size limit (2.3.1). The type may contain `never`, so `(define-constant ERR-NOT-FOUND (err u404))` has type `(response never uint)` and is usable in any function whose error type is `uint`. If `expr` is a literal, or is built only from literals and other static constants using primitives that read neither state nor context (arithmetic, sequence, conversion, and hash primitives), the constant is **static**: its value is known at analysis time and it may be used wherever a literal is required (size terms, `contract-call?` targets, `as-max-len?` bounds)
`(define-type name type)`	A type alias	2.3.6
`(define-data-var name type expr)`	A persistent variable	`type` storable; `expr : T <: type` is deploy-time code (2.4.2)
`(define-map name key-type value-type)`	A persistent map	both types storable; the key prefix plus `serialized-size(key-type)` MUST fit the storage key limit (2.8, 3.7)
`(define-fungible-token name)` and `(define-fungible-token name expr)`	A fungible token, optionally with a total supply cap `expr : uint`	
`(define-non-fungible-token name type)`	A non-fungible token with identifier type `type`	`type` storable
`(define-event name type)`	A contract event whose data has type `type`, emitted with `emit` (3.10)	`type` is an interface type (2.3.4) whose `serialized-size` MUST satisfy the value size limit (2.8). The name is in the contract's interface and the type in `abi.types` (2.4.4)

Form	Introduces	Rules
`(define-trait name ((fn-name (type ...) type) ...))` and `(define-trait name ((fn-name (type ...) type read-only) ...))`	A trait: a set of function names with interface-typed signatures whose result types are `response` types. A function marked `read-only` may only be implemented by a read-only function, and may be called through the trait from read-only code (2.7.4)	
`(impl-trait trait-identifier)`	A conformance claim, checked statically: every function in the trait MUST be defined as a public or read-only function with a compatible signature (parameters contravariant, result covariant), and a function the trait marks `read-only` MUST be defined with `define-read-only`	
`(define-public (name (param type) ...) expr ...)`	An externally callable function	parameter types are interface types; result type MUST be a `response`
`(define-read-only (name (param type) ...) expr ...)`	An externally callable function that MUST NOT mutate state (2.7.4)	parameter types are interface types
`(define-private (name (param type-or-ref-kind) ...) expr ...)`	An internal function called by value	may be generic and overloaded, and may end with a rest parameter (2.6)
`(define-macro (name (param type-or-ref-kind) ...) expr ...)`	A macro: an internal definition expanded at each call site, with call-by-name parameters (2.6.3)	may be generic and overloaded

Parameter names within one definition MUST be distinct (a duplicate is a static error, not a runtime one). A function body is a non-empty sequence of expressions evaluated in order; the value of the last is the value of the body. Result types are inferred (2.6.4).

Intermediate responses MUST be handled. In any expression sequence (a function body, a `let` or `begin` body, or a `match` arm), every expression other than the last MUST NOT have a `response` type. A `response` is consumed by returning it, matching on it, or passing it to a function; it cannot be dropped (P7). This is a static error.

2.4.2 Top-level expressions

Deploy-time code is every `define-constant` initializer, `define-data-var` initializer, `define-fungible-token` supply cap, and top-level form that is not a definition. It is evaluated once, at deploy, in source order. Deploy-time code is ordinary code: it may read and write state, call other contracts, and emit events, subject only to the `return` restriction below.

Definitions used at deploy must come first. Deploy-time code MUST NOT use, directly or through the definitions it calls (2.4.3), a definition that has an initializer and appears later in the source: a constant that is not static (2.4.1), a data variable, or a fungible token with a supply cap. This is a static error. Every other kind of definition (functions, types, traits, maps, tokens without a cap, and static constants) may be used from any position, because nothing about it is evaluated at deploy: a static constant's value is substituted at analysis time, so its position in the file is irrelevant.

The contract's own definitions, including its traits, are fully usable in top-level expressions: a trait defined in the contract is resolved against the contract's own analysis, not against a deployed copy of the contract, so `(f .other-contract)` with `f` taking a locally defined trait type is valid at the top level. `return` MUST NOT appear in a top-level expression, so the early-exit library functions (`unwrap!`, `try!`, `asserts!`, `unwrap-err!`), which expand to `return`, are unusable in top-level expressions and in `define-constant` and `define-data-var` initializers; `unwrap-panic` and `match` serve there. A runtime error in any top-level expression aborts the deploy.

2.4.3 The definition graph is acyclic

Definition `a` depends on `b` if the body, type, or initializer of `a` mentions the name `b` anywhere: in call position, in type position, or as a reference argument. The dependency graph over all definition names in a contract MUST be acyclic. This single rule excludes recursion (direct, mutual, and through function references), recursive types, and recursive constants.

The graph is over names, not instances, and is computed from the source alone, before overload resolution and instantiation (2.6.2). All definitions of an overloaded name are one node, so one overload of a name MUST NOT use another; the shared part lives in a helper, as the prelude's `std-sqrti-uint` does for the two `sqrti` overloads. Because the graph is acyclic before instantiation begins, every chain of instantiations and macro expansions is finite (P1), and the deploy-order check of 2.4.2 can be made on the source.

2.4.4 Contract interface

The compiler emits two sections describing the contract into its module (../07-chain-abi.md section 3.4):

- **`abi.interface`** (../07-chain-abi.md section 4.1) names the public and read-only functions with their kinds, the fungible and non-fungible tokens, and the events. The chain validates it against the module at deploy and acts on nothing else.
- **`abi.types`** (../08-value-interpretation.md section 5) declares the parameter and result types of every public and read-only function, the identifier type of every non-fungible token, the data type of every event, the traits the contract defines, the traits it claims with `impl-trait`, and the layout of its data

variables and maps (3.7). A Clarity contract declares all of them, so none of its functions, tokens, events, or storage is undeclared.

The chain does not read `abi.types`. Other contracts, whatever their source language, compile typed calls against it (2.5.8), and wallets and explorers display values with it. Nothing at runtime checks that a module honors its declarations, which is why a Clarity contract validates every value it receives from another contract (2.5.8).

2.4.5 Standard library names

The standard library prelude (std/prelude.clar) is in scope in every contract as if it were part of the contract's own top-level forms. A contract MUST NOT define a top-level name that the prelude defines.

2.5 Expression forms

Beyond literals, names, and function application, the core has the following special forms. Anything not listed here is either a primitive (03-primitives.md) or a library definition.

2.5.1 Names

A bare identifier denotes a `let`- or `match`-bound variable, a function parameter, or a `define-constant`; because there is no shadowing (2.5.2), at most one of these applies at any point. Function, map, variable, token, and event names are not values and may appear only in application or reference-argument position. There are no other bare names: the execution context is read with `get-current-context` (3.9).

2.5.2 `let`

```
(let ((name expr) ...) body-expr ...)
```

Bindings are evaluated left to right; each binding is in scope for the following bindings and the body. The type is the type of the last body expression.

No shadowing. A `let` binding, a `match` binding, or a function parameter MUST NOT reuse a name that is already in scope: a local, a parameter, a top-level definition of the contract, or a reserved name (2.2). A keyword (2.2) cannot be bound at all. Every name therefore has exactly one meaning at each point in the program, and the type checker and the evaluator cannot disagree about which one applies.

Legacy is equally strict about locals, top-level definitions, and builtin names. Legacy source that uses a name reserved only in vNEXT (such as `max`, `min`, `abs`, or `fold-left`) or a keyword new in vNEXT (such as `as`) as a local must rename it when ported.

`(begin expr ...)` is sugar for `(let () expr ...)`.

2.5.3 `match`, `and`, `or`

```
(match expr arm arm)
arm     ::= (pattern body-expr ...)
pattern ::= (some name) | none | (ok name) | (err name) | true | false
```

`match` is the core's branching construct. The scrutinee type determines the required patterns, each of which MUST appear exactly once:

Scrutinee type	Required patterns	Bindings
`(optional T)`	`(some x)`, `none`	`x : T`
`(response T E)`	`(ok x)`, `(err e)`	`x : T`, `e : E`
`bool`	`true`, `false`	none

The scrutinee MUST have a type in this table. A scrutinee of type `never` is a type error (2.3.3), including in an instance of a generic function whose type variable is instantiated at `never` (2.6.2).

The type of the `match` is the lub of the arm types, and exactly one arm is evaluated.

An arm whose pattern binds a name of type `never` (`(some x)` on `(optional never)`, `(ok x)` on `(response never E)`, `(err e)` on `(response T never)`) cannot be reached, but it still MUST appear, and its body is type-checked and contributes to the lub like any other arm. On `r : (response int never)`, `(match r ((ok v) v) ((err e) e))` has type `int`, since `lub(int, never) = int`, and so does `(match r ((ok v) v) ((err e) (panic)))`. The convention is `(panic)` in such an arm.

`if` is a library function defined over `match` on `bool`. A macro body is type-checked with each parameter at its declared type (2.6.3), so the scrutinee of that `match` is a `bool` whatever the argument's type: `(if (panic) a b)` and `(if (not (panic)) a b)` both have type `lub(A, B)`.

```
(and expr expr ...)
(or expr expr ...)
```

`and` and `or` take two or more `bool` expressions and evaluate them left to right, stopping at the first `false` (`and`) or the first `true` (`or`). They are core forms, not library functions, because short-circuiting over any number of arguments cannot be written in the language: a macro has a fixed number of by-name parameters, and a rest parameter (2.6.5) is a list value whose elements are all evaluated before the body runs. `(and a b c)` means exactly `(match a (true (match b (true c) (false false))) (false false))`. The operands are typed as the arguments of a macro are (2.6.3): each MUST be a `bool`, which an expression of type `never` is by subtyping, so `(and (panic) b)` has type `bool`.

2.5.4 `return`

```
(return expr)
```

Evaluates `expr`, then terminates the innermost enclosing function body (after macro expansion, 2.6.3) with that value. `return` has type `never`. The type of `expr` contributes to the enclosing function's inferred result type (2.6.4). `return` MUST NOT appear in a top-level expression.

When a `return` crosses an `as-contract?` or `restrict-assets?` boundary, the rules in 2.5.8 apply.

`unwrap!`, `try!`, `asserts!`, and the rest of the early-exit family are library functions defined over `match` and `return`.

2.5.5 Application

```
(f arg ...)
```

`f` is a primitive, a function defined in the current contract or the prelude, or a `(function ...)` parameter. Arguments are evaluated left to right (except for macros, 2.6.3). Typing and overload resolution are defined in 2.6.2.

2.5.6 Constructors

`(some e)`, `(ok e)`, `(err e)`, `(list e ...)`, and `{ f: e, ... }` construct values.

Form	Type
`(some e)` with `e : T`	`(optional T)`
`(ok e)` with `e : T`	`(response T never)`
`(err e)` with `e : E`	`(response never E)`
`(list e1 ... en)`	`(list n lub(T1, ..., Tn))`; `(list)` is `(list 0 never)`
`{ f1: e1, ... }`	`{ f1: T1, ... }`; field names MUST be distinct

`list`, `and`, and `or` are the only variadic forms. Functions take a variable number of arguments only through a rest parameter (2.6.5).

2.5.7 Tuple access

- `(get f e)` with `e : { f: T, ... }` has type `T`. `f` MUST be a field of the tuple type.
- `(merge e1 e2)` with tuple types `R1`, `R2` has the tuple type whose fields are the union of both, taking the type from `R2` for fields present in both. At runtime, fields of `e2` overwrite fields of `e1`.
- The operand of `get` and both operands of `merge` MUST have tuple types.

2.5.8 Cross-contract calls

```
(contract-call? contract function-name arg ...)
```

`contract` is a contract principal literal, a static constant (2.4.1) of contract principal type, or an expression of a trait type `(trait X)`; in the trait case `function-name` MUST be a function of `X`. Any expression of trait

type is acceptable, including a tuple field or a `let`-bound name. A `contract` of type `never` is a type error, since it names no trait (2.3.3). Arguments are type-checked against the callee's declared signature in its `abi.types` (2.4.4) in the literal case, or against the trait's signature in the trait case, read from the `abi.types` of the contract that defines the trait. Either way the section read is an input to compilation, and the compiler records its hash in the module's provenance (../03-architecture.md section 2). In the literal case the callee's function MUST be declared; a function with no declared signature can only be called with `contract-call-raw?`. The result type is the declared result type. The callee MUST be a public or read-only function and MUST NOT be in the current contract.

The call is dispatched by the chain, which enforces the depth limit and the no-reentrancy rule (2.7.5). The callee may be written in any source language. The arguments are passed in canonical encoding against the declared parameter types (../08-value-interpretation.md section 6.1). When the caller relies on the callee being read-only, the compiler sets the call's `read_only` flag (../07-chain-abi.md section 5.3) and the chain rejects a callee that is not: the call is from read-only code (2.7.4), the trait marks the function `read-only`, or the literal callee declares it read-only.

Values are validated by the module. The chain does not check values against declared types (../07-chain-abi.md section 5.4), so the compiled module does:

- Every public and read-only function decodes its arguments, by canonical decoding against its declared parameter types (../08-value-interpretation.md section 3), before its body runs.
- Every `contract-call?` decodes the result against the declared result type, in the trait case the trait's: the `ok` arm's type or the `err` arm's, whichever the result flag selects.

A value that fails to decode is a runtime error (2.7.2). A Clarity contract can therefore rely on its parameter types whoever calls it, and on the result type of every call it makes, whatever language the other side is written in.

Traits are checked at compile time. Conformance (2.3.2, `impl-trait` in 2.4.1) is checked when a contract is compiled, against declared signatures. A principal passed in a trait-typed parameter is not checked against the trait at runtime, because a module cannot read another contract's declarations. A call through it is still safe: it aborts if the function does not exist or, for a function the trait marks `read-only`, is not read-only, and its result is validated like any other, so a principal that does not conform can make the call fail but cannot make it return a value of the wrong type.

```
(contract-call-raw? contract function-name args size)
```

Calls a function without a declared signature, such as a function of a contract written in a language that does not declare its types (../08-value-interpretation.md section 6.4). `contract` and `function-name` are as for `contract-call?`, except that `function-name` is not checked against any declaration. `args : (buff N)` is passed to the callee as the raw argument bytes. `size` is a size term `M` (2.3.1), and the form has type `(response (buff M) (buff M))`: the result bytes, in the `ok` or `err` arm as the result flag selects. A result longer than `M` bytes is a runtime error. From read-only code the call sets the `read_only` flag, so the callee MUST be read-only; otherwise the callee may be public or read-only. The rollback rules of 2.7.3 apply to the callee as to any call.

2.5.9 Context scoping and asset allowances

```
(as-contract? (allowance ...) body-expr ...)
(as-contract? ((with-all-assets-unsafe)) body-expr ...)
(restrict-assets? owner-expr (allowance ...) body-expr ...)

allowance ::= (with-stx amount)
            | (with-ft contract token-name amount)
            | (with-nft contract token-name identifiers)
            | (with-staking amount)
            | (with-pox)
```

`(with-all-assets-unsafe)` waives the check for the current contract's assets. Only `as-contract?` accepts it, because in `restrict-assets?` it would make the form a no-op. It MUST be the only allowance in the list, because any other allowance beside it would have no effect. The grammar above encodes both rules, so writing it in `restrict-assets?` or alongside another allowance is a syntax error.

Every allowance argument is an ordinary expression, so a contract or token can come from a parameter, a variable, or a map entry, not only from a literal:

Argument	Type	Meaning
`amount`	`uint`	the most uSTX (`with-stx`), token units (`with-ft`), or uSTX locked for staking (`with-staking`) the body may move from the owner
`contract`	`principal`	the contract that defines the token; a non-contract principal, or one that defines no token of that name, matches no movement
`token-name`	`(string-ascii N)`	the name the token was defined with (`define-fungible-token` or `define-non-fungible-token`); `N` is at most the identifier limit (2.8)
`identifiers`	`(list N T)`	the non-fungible token identifiers the body may move; `T` is any storable type, and an identifier matches a movement when their serialized forms (3.4) are equal

Both forms evaluate their body in a nested state frame and, on completion, compare the asset movements of a designated owner against the allowance list.

- `as-contract?` sets the context's `sender` and `caller` fields (3.9) to the current contract for the body; the designated owner is the current contract.
- `restrict-assets?` leaves the context unchanged; the designated owner is the value of `owner-expr : principal`.

Both have type `(response T uint)` where `T` is the body type. Outcome:

1. If the body completes normally and every movement of the owner's assets is covered by an allowance, the frame is committed and the value is `(ok body-value)`.
2. If any movement is not covered, the frame is rolled back and the value is `(err i)`, where `i` is the zero-based index in the allowance list of the violated allowance. The allowance comparison is host code (../03-architecture.md section 4), so the chain ABI defines which index is reported when several allowances are violated or a movement matches no allowance at all.

3. If the body is exited by a `return` that crosses the form, the allowance check still runs. On violation, the frame is rolled back, the `return` is cancelled, and the form's value is `(err i)`. Otherwise the frame is committed and the `return` proceeds.
4. A runtime error in the body aborts the transaction as usual.

Allowance sub-forms are valid only in allowance position. Allowance arguments are evaluated left to right, allowance by allowance, before the body, in the enclosing context (for `as-contract?`, with the caller's `sender`, not the contract's). The number of allowances is bounded (2.8).

2.5.10 Type ascription

```
(as type expr)
```

`type` is a type expression (2.3.1), and may use the type and size variables in scope. `expr`'s type MUST be a subtype of `type` (2.3.2); the form has type `type` and evaluates to the value of `expr`. `as` only widens: it never narrows, converts, or checks anything at runtime. It exists to give an initial value the type a fold needs (2.6.2), and to give a type error a place to point: `(as (list 10 uint) (list))`, `(as (optional principal) none)`, `(as (string-ascii N) "")`.

2.6 Functions

2.6.1 Visibility

Kind	Callable from other contracts	Callable internally	May mutate state	May be generic or overloaded	Evaluation
`define-public`	yes	yes	yes	no	by value
`define-read-only`	yes	yes	no	no	by value
`define-private`	no	yes	yes	yes	by value
`define-macro`	no	yes	yes	yes	expanded at call site, by name

Every invocation of a public function has the rollback semantics of 2.7.3, whether it is called from another contract, as the entry point of a transaction, or internally from the same contract. A public function's behavior on `err` therefore does not depend on who called it.

2.6.2 Instantiation and overload resolution

Let a call be `(f a1 ... an)`, and let the candidate definitions of `f` be those that accept `n` arguments: exactly `n` parameters, or, for a rest-parameter definition, fewer than `n` fixed parameters (2.6.5). A call whose last argument is a spread `e...` is accepted only by rest-parameter definitions with exactly `n - 1`

fixed parameters. For a candidate with parameter types `P1 ... Pn` (which may contain type variables, size variables, and reference kinds) and free variables `X`, an **instantiation** is a substitution `σ` over `X`, assigning a type to each type variable and a size to each size variable, such that:

- for each value parameter, `type(ai) <: Pi[σ]`;
- for each reference-kind parameter, `ai` is a name (or reference parameter) of the matching kind (2.3.7), and:
- for `(var T)`, `(map K V)`, and `(non-fungible-token T)`, its declared types are identical to the types in `Pi[σ]`, with no subtyping in either direction;
- for `(event T)`, its declared type `D` is a supertype of `T[σ]`. This bounds the variables in `T` only from above, so a variable that occurs nowhere but in event references is not left to the least-instantiation rule below, which would put it at `never` or size 0. It is matched as in `(var T)`: it takes the type or size that `D` has at its position, so `(event X)` is instantiated at the declared type and `(event (string-ascii N))` at the declared size. The declared types MUST agree at every position of such a variable: `N` in two `(event (buff N))` parameters given events of `(buff 10)` and `(buff 20)` has no instantiation, and separate variables give each its own. A variable that also occurs outside event references is instantiated by those occurrences, and `D` only bounds the result: emitting `0x42` through `(event (buff N))` and `(x (buff N))` to an event of `(buff 10)` instantiates `N` at 1, and to an event of `(buff 0)` has no instantiation;
- for `(function (Q ...) R)`, the named function admits an instantiation whose parameter types are supertypes of `Q[σ]...` and whose result type is a subtype of `R[σ]`. A rest-parameter definition with `k` fixed parameters is named at a function type with `k + 1` parameters, and the last one fills its rest parameter as a one-element list (2.6.5).

The candidate **admits** the call if an instantiation exists. Among the instantiations that exist, the **least** one (pointwise under `<:`, and smallest size values) is chosen; if no least instantiation exists the candidate does not admit the call. This is the declarative form of the familiar "unify and take the least upper bound" algorithm.

Equivalent traits (2.3.2) are each a subtype of the other, so "least" is read the way `lub` is, not as any minimal type in the preorder. A type variable whose lower bounds are the argument types `S1 ... Sk` is instantiated to `lub(S1, ..., Sk)`. One argument of type `(trait X)` instantiates it to `(trait X)`, never to some other trait equivalent to `X`. Arguments of two distinct equivalent trait types instantiate it to `principal`, so `(if c x y)` with `x : (trait a)` and `y : (trait b)`, where `a` and `b` are equivalent traits, has type `principal`, just as the corresponding `match` does.

When a type variable occurs in both a parameter and the result of a `(function ...)` parameter, as the accumulator `A` does in `fold-left`, the least instantiation is a least fixpoint rather than a unification: the initial value's type is a lower bound for `A`, and the named function's result type at that `A` is another, so `A` is the least type closed under both. The prelude relies on this: `(fold-left std-prepend-byte 0x bytes)` starts from `(buff 0)` and settles at `(buff 16)`, and `index-of?` starts from `none` and settles at `(optional uint)`. Every type in the language has finitely many supertypes below the size limit, so the iteration terminates, but a step function that grows its accumulator would take as many steps as the size limit allows before failing. The iteration is therefore capped: starting from `A0`, the initial value's type, each step computes `A(i+1) = lub(A(i), R(A(i)))`, where `R` is the named function's result type, and if `A(i+1) =`

`A(i)` is not reached within the fixpoint step limit (2.8), the candidate does not admit the call. Every prelude fold closes in two steps. A fold that needs a wider starting type states it with `as` (2.5.10): `(fold-left f (as (list 10 uint) (list)) xs)`.

Overload resolution: exactly one candidate MUST admit the call. Zero is a type error; more than one is an ambiguity error. Overloading is by arity and by parameter types; there is no ranking. Because `never <: T` for every `T`, an argument of type `never` is admitted by every candidate, so with an overloaded name it is an ambiguity error: `(unwrap! (panic) e)` is one. A type variable that only such an argument would fix is instantiated at `never`, the least type. A sequence kind (3.3) has no least member, so a primitive whose kind only such an argument would fix has no least instantiation and does not admit the call: `(len (panic))` is a type error. Otherwise the call is typed as any other, so `(+ 1 (return u0))` has type `int` (2.3.3). An author who means a particular candidate states the type with `as` (2.5.10).

Every other ambiguity is an error where the definitions are written rather than at the call: the definitions of a name MUST NOT **overlap**. Two definitions overlap when, under some instantiation of each, their parameters meet (2.3.2) at every position. Reference-kind parameters meet when they are of the same kind with the same declared types, except that two `(event ...)` kinds meet when `lub` of their types is defined, since an event declared at that type is admitted by both, and two `(function ...)` kinds meet when their parameter and result types meet pointwise. The check is on the definitions, whether or not the name is called:

```
(define-private (pad (b (buff 4))) (concat b 0x00))
(define-private (pad (b (buff 8))) (concat b 0x0000))    ;; error: (buff 4) is below both

(define-private (first (xs (list 5 (buff 4)))) ...)
(define-private (first (xs (list 3 (buff 8)))) ...)      ;; error: (list 3 (buff 4)) is below
both

(define-private (show (x T)) ...)
(define-private (show (x uint)) ...)                     ;; error: T at uint

(define-private (note (e (event (buff 4)))) ...)
(define-private (note (e (event (buff 8)))) ...)         ;; error: an event of (buff 8) fits
both
```

Neither parameter type of `first` is a subtype of the other, but the two still meet. Definitions whose parameters differ in shape at some position, such as `int` and `uint`, `optional` and `response`, or a list and a `buff`, never overlap; the prelude's overloads are all of this kind. Definitions that do not overlap can still both admit an argument whose type mentions `never`: one on `(list 5 (buff 4))` and one on `(list 3 uint)` both admit `(list)`, and such a call is an ambiguity error as above.

The same rule applies to a name passed as a `(function (Q1 ... Qk) R)` argument. Its candidates are the definitions of that name that can be named at `k` parameters: those with exactly `k` parameters, and rest-parameter definitions with `k - 1` fixed parameters. An instantiation of the enclosing candidate fixes one of them, and exactly one MUST appear in some instantiation; zero is a type error and more than one is an ambiguity error. Usually only one fits: `(map int-to-ascii xs)` with `xs : (list 5 int)` uses the `int` definition. Because definitions do not overlap, more than one fits only when some `Qi` mentions `never`, where a direct call would be ambiguous too:

```
(map int-to-ascii (list)) ;; ambiguity error: (list) is (list 0 never)
```

In a fold, the referenced definition is fixed for the whole fixpoint, since a single function is called on every element: each definition is tried with its own fixpoint, it appears in an instantiation if that fixpoint closes, and exactly one MUST. With the definition fixed, admitting the final accumulator type implies admitting every earlier one, since each step's type is a subtype of the next.

The result type of the call is the candidate's inferred result type under `σ`, with size arithmetic in primitive signatures evaluated on the concrete sizes. A resulting size that exceeds the limit for the call's position (2.3.1) is a type error.

Each distinct combination of candidate, `σ`, and reference arguments is a distinct **instance**. Instances are what the compiler operates on; the definition graph (2.4.3) is over names and is checked before any instance exists. Instance bodies are type-checked with the concrete substitution applied ("template" semantics): a generic body that is never instantiated is parsed and scope-checked but not type-checked. An instance in which a type variable is instantiated at `never`, as `f` is in `(map f (list))`, is type-checked like any other (2.3.3).

A type error found while checking an instance body is reported at the call site that created the instance. The report MUST name the called definition and the instantiation (for example, `max` at `bool`), and MAY include the location inside the body as secondary information. When the instance was created from inside another instance body, the report MUST include the chain of call sites back to the user's code.

Template semantics give functions over "any tuple with these fields" without a separate construct:

```
(define-private (credit (acct T) (amt uint))
  (merge acct { balance: (+ (get balance acct) amt) }))
```

`credit` admits any tuple with a `uint` field `balance` and returns a tuple of the same shape. Each tuple shape is its own instance, so no value carries fields its static type lacks (2.3.2). A call with a tuple that has no `balance` field is reported as `credit` at that tuple type, at the call site.

2.6.3 Macros

A macro (`define-macro`) is a typed, hygienic definition that is expanded at each call site.

Expansion is defined as follows:

1. The call is resolved and the body is type-checked exactly as for a private function (2.6.2): the arguments select the candidate and fix the instantiation, and the body is checked with each parameter at its declared type under that instantiation, not at the argument expression's own type. An argument whose type is a strict subtype of the parameter's, such as `(panic)` at a `bool` parameter, is seen by the body at the parameter's type, so `(if (panic) a b)` and `(if (not (panic)) a b)` are typed alike (2.5.3).
2. Evaluation of the call is evaluation of the body with each parameter reference replaced by its argument expression, evaluated in the caller's context at the point of reference (call by name). An

argument whose parameter is never referenced on the taken path is not evaluated. Because evaluation happens at the point of reference, an argument referenced inside an `as-contract?` or `restrict-assets?` form in the body is evaluated inside that frame, with the frame's `sender` and `caller`, and the order in which arguments are evaluated is the order of their references in the body, not their order at the call site. Prelude macros reference their parameters outside any such frame. Prelude functions whose names read as ordinary calls (`default-to`, `map-insert?`) are private, so their arguments are evaluated eagerly as in legacy; laziness is confined to control-flow forms, and the library documents each place where it changes legacy behavior (`unwrap!`, `unwrap-err!`).

3. Names bound inside the body (`let`, `match`) are fresh and cannot capture names in argument expressions. Free names in the body resolve in the defining contract's top-level scope.
4. `return` inside the body returns from the caller's innermost enclosing function body, not from the macro.

To make call by name safe, each **value** parameter of a macro MUST be referenced at most once in the body (syntactically). A body that needs a parameter twice binds it with `let` first. Reference-kind parameters have no evaluation and may be referenced any number of times.

Macros cannot be passed as `(function ...)` arguments. Macro expansion is finite because the definition graph is acyclic (2.4.3).

> [!NOTE] **When to use `define-macro`.** A macro is the only way to write a helper that returns from its caller or evaluates an argument only on some paths. Use it for exactly those two cases, and use `define-private` for everything else. For example, a helper that replaces an error code and exits in one step cannot be private:
>
> ``` (define-macro (try-with! (value (response T E)) (code F)) (match value ((ok inner) inner) ((err ignored) (return (err code)))))
>
> (define-public (withdraw (amount uint)) (let ((balance (try-with! (get-balance (get sender (get-current-context))) ERR-NO-ACCOUNT))) ...)) ```
>
> A private function is the better default because it behaves like every other call: each argument is evaluated exactly once, left to right, before the body runs. It can be passed to `map`, `filter`, and the folds. It is compiled once per instance instead of once per call site. Its body may use a parameter any number of times.
>
> A macro gives those up. At a call site, an argument whose parameter is not used on the path taken is never evaluated, so its side effects (a `contract-call?`, a `var-set`, an abort) do not happen. Arguments are evaluated in the order the body uses them, not the order they are written. An argument used inside an `as-contract?` or `restrict-assets?` form in the body sees that frame's `sender`. A function whose name reads as an ordinary call should not be a macro unless its documentation states which arguments may go unevaluated.

2.6.4 Result type inference

The result type of a function instance is the lub of the type of its last body expression and the types of the operands of every `return` in its body (after macro expansion), whether or not that `return` can be reached. A `return` whose operand has type `never` contributes `never`, which the lub absorbs (2.3.3). If the lub is undefined, it is a type error. For `define-public`, the result type MUST be a `response` type; `never` is not one, so a public function whose every path aborts is a type error. Such a function states its interface with `as` (2.5.10): `(as (response bool uint) (panic))`. This is a rule about the declared interface, not about `never`; a read-only function, whose result need not be a `response`, may have the result type `never`.

The result type is not written in source, and may contain `never` (2.3.4). A function with no `err` path has an error type of `never`; a function with no `ok` path has a value type of `never`. Either is a valid interface type and is what the compiler declares in `abi.types` (2.4.4).

2.6.5 Rest parameters

The last parameter of a `define-private` definition MAY be a **rest parameter**, written with `...` after its type:

```
(define-private (name (first A) (second B) (rest T...)) body)
```

A call supplies the fixed parameters in order and **one or more** further arguments for the rest parameter. Inside the body, the rest parameter is a list: `rest : (list N T)`, where `N` is the number of rest arguments at that call site. The call `(name a b t1 t2 t3 t4)` is the call of the definition with `a` and `b` for the fixed parameters and `(list t1 t2 t3 t4)` for the rest, so arguments are evaluated left to right and the list is typed by 2.5.6. Each call site is its own instance (2.6.2), with `N` bound to the number of rest arguments.

A function that takes "two or more" is one definition. Its binary case is simply the call with one rest argument:

```
(define-private (std-max-pair (left T) (right T))
  (if (> left right) left right))

(define-private (max (left T) (right T...))
  (fold-left std-max-pair left right))

;; (max a b)     is (fold-left std-max-pair a (list b))
;; (max a b c d) is (fold-left std-max-pair a (list b c d))
;; (max a)       is an arity error: a rest parameter needs an argument
```

`fold-left` passes the accumulator first (3.3), so this pattern is the left fold of the helper whether or not the helper is commutative.

An argument that is already a list is not spread implicitly: in `(max a xs)` with `xs : (list 5 int)`, `xs` is a single rest argument, so `right : (list 1 (list 5 int))` and the body fails to type-check. To pass an existing list as the rest parameter, **spread** it by writing `...` after the argument:

```
(max a xs...)                            ;; right is xs
(concat "0x" (map std-hex-byte bytes)...)  ;; one string from many
```

A spread argument `e...` with `e : (list N T)` MUST be the only argument in the rest position: the call supplies the fixed parameters and then `e...`, with no other rest arguments. The rest parameter is the value of `e`, typed `(list N T)`, and the instance binds `N` to that bound. `e` is evaluated once, in its position, like any other argument. Unlike a call without a spread, the rest list's runtime length may be less than `N`, including zero, so `(max a (list)...)` is `a`. Only a rest parameter admits a spread argument: a definition with a fixed parameter in that position does not accept the call. Spread is not available for special forms (`and`, `or`, `let`, `match`).

Rules:

- **All fixed or all rest.** If any definition of a name has a rest parameter, every definition of that name MUST have one, with the same number of fixed parameters. Overloads then differ only by type, MUST NOT overlap (2.6.2), and overload resolution needs no rule about arity.
- **Passing a rest function.** A rest-parameter function may be passed as a `(function ...)` argument at the type with one more parameter than its fixed parameters; the last argument becomes a one-element rest list (2.6.2). So `(fold-left max 0 xs)` and `(fold-left + 0 xs)` work.
- **No self-reference.** A rest-parameter body cannot fold itself, since that is a cycle (2.4.3). The binary operation lives in a helper, as `std-max-pair` above, or in a primitive.
- **Private only.** Public and read-only functions MUST NOT have rest parameters, because their declared signatures have a fixed arity (2.4.4). Macros MUST NOT have them either: rest arguments are evaluated as a list before the body runs, which would silently make a macro strict in those arguments.
- **Primitives** whose chapter 3 signature ends in `I...` or `S...` follow the same rules, including spread. Their result type may be tighter than a user definition could state; `concat`'s bound, for example, is the exact sum of its arguments' bounds, or `N + K × M` when a `(list K S[M])` is spread after a first argument of bound `N`.

A compiler instantiates each call site with its own `N`, so it can compile the one-argument case of a rest function without building a list. The spec does not require it.

`is-eq` and the comparisons have no rest parameter: `(is-eq a b c)` has no single obvious meaning, and "all equal" is `(and (is-eq a b) (is-eq a c))`.

2.7 Dynamic semantics

2.7.1 Evaluation

Evaluation is strict and left to right, except that macro parameters are evaluated by name (2.6.3), `match` evaluates exactly one arm, and `and` and `or` stop at the first argument that decides the result (2.5.3). Every expression evaluates to a value of its static type or aborts. A runtime value carries no static type information of its own; its size and shape are determined entirely by the value, and a value passed to a callee is exactly the value the caller computed.

2.7.2 Runtime errors

A runtime error (arithmetic overflow, division by zero, failed conversion, `panic`, cost or memory limit exceeded, a value that fails validation at a call boundary (2.5.8), and the other conditions listed per primitive) aborts the entire transaction. There is no mechanism to catch a runtime error. All state changes of the transaction are discarded; the transaction still pays for the cost it consumed. A runtime error the module detects is reported through the chain ABI's `abort` with a code for its class (../07-chain-abi.md section 10.2); the codes are defined with compilation to Wasm.

2.7.3 Public function results and rollback

Every invocation of a public function runs in its own state frame: its state changes (variables, maps, token balances, events) are buffered. If it returns `(ok v)`, they are committed to the caller's frame. If it returns `(err e)`, they are discarded and `(err e)` is returned to the caller, which continues executing. A `return` of an `err` value has the same effect as evaluating to it.

For a call from another contract or a transaction entry point, the chain opens the frame and commits or discards it by the result flag the callee sets (../07-chain-abi.md section 5.2), so the rule holds for calls between modules of any source language. A Clarity public function sets the flag from the arm of its response and passes the arm's value as the result (../08-value-interpretation.md section 6.2). A read-only function also runs in a frame the chain opens, which carries its context (3.9) but always commits, since it holds no writes (../07-chain-abi.md section 7.1). The flag decides nothing there, so a read-only function always sets it to ok and passes its whole result. For an internal call to a public function, the compiled module opens and closes the frame itself through the chain ABI's frame facility, the one `as-contract?` and `restrict-assets?` use (2.5.9).

An internal call to a public function therefore costs a frame, which a call to a private function does not. Where that matters, the convention is to put the body in a private function and have the public function call it: internal callers use the private function and pay nothing, and external callers get the rollback through the public wrapper.

2.7.4 Read-only enforcement

A read-only function MUST NOT, directly or through functions in the same contract, execute `var-set`, `map-set`, `map-delete`, any `*-mint?`, `*-burn?`, or `*-transfer?` primitive. A `contract-call?` from read-only code is permitted only when the callee is statically known to be read-only: a literal or static constant target whose interface marks the function read-only, or a trait target whose trait marks the function `read-only`. All of this is checked at compile time, so a violation is reported before deploy. The chain independently rejects any write attempted inside a read-only call frame, which is what makes the guarantee hold when read-only code calls a module from another language.

2.7.5 Depth and reentrancy

The chain rejects a `contract-call?` into a contract that already has an active call frame in the current transaction, and rejects any `contract-call?` that would exceed the chain ABI's contract-call depth limit. Both are runtime errors. The function call depth limit (2.8) is the language's own and bounds calls within one

contract's module; the compiler enforces it statically, since without recursion every call chain is visible in the definition graph.

2.7.6 Events

`emit` and every asset primitive emit events into the transaction's event batch. An `emit` event carries the event's name and the canonical encoding of its value (3.10). Events of a rolled-back frame are discarded with it and no longer count toward the transaction's event bytes (../07-chain-abi.md section 8.3). `log` emits no event (3.10).

2.7.7 Cost

The language defines no costs (P4). A contract runs as the Wasm module its source compiles to, and its execution cost is the fuel that module consumes plus the chain's charges for the host functions it calls. Library functions are compiled and metered like any other code. Exceeding the transaction or block budget is a runtime error.

Fuel replaces a per-construct runtime cost table, and it is the only cost dimension: storage reads and writes are charged in fuel like everything else, at prices the chain ABI defines. Compilation happens off chain, so its cost is the deployer's. What the chain charges for is the module: its size and its validation at deploy, as defined by the chain ABI.

The following rules are about resource safety, not pricing:

- Memory for a decoded value is reserved against the size bound of its type before decoding begins, so a hostile input cannot allocate more than its declared type admits. This covers every decode of bytes the module did not produce: `from-consensus-buff?`, which reserves `serialized-size(T)`; the arguments a public or read-only function decodes on entry, which reserve the sum of `serialized-size` over its parameter types; and the result of every `contract-call?`, which reserves `serialized-size` of the declared result type (2.5.8). Input longer than that bound cannot be a valid encoding and is rejected before any decoding: as a validation failure (2.7.2) for arguments and call results, and as `none` for `from-consensus-buff?`.
- A trait-typed value is a principal and is sized as one.

2.8 Limits

These are language-level limits. Hosts MAY impose stricter limits on contract source size and transaction budgets.

Limit	Value	Inherited from
Maximum serialized value size	1,048,576 bytes	`MAX_VALUE_SIZE`; also the chain ABI's event data limit, which bounds `emit` (3.10)
Maximum intermediate value size (2.3.1)	8,388,608 bytes	new, provisional; MUST stay below the chain's Wasm memory limit

Limit	Value	Inherited from
Fixpoint step limit for fold accumulators (2.6.2)	8	new, provisional
Maximum type nesting depth	32	`MAX_TYPE_DEPTH`
Maximum identifier length	128 bytes	`MAX_STRING_LEN`; string literals are bounded only by the value size limit, as every other value is
Maximum function call depth within a contract (2.7.5)	128	`MAX_CALL_STACK_DEPTH`; contract-call depth is the chain ABI's
Maximum allowances per `as-contract?` / `restrict-assets?`	128	`MAX_ALLOWANCES`
Maximum storage key: a map's key prefix plus `serialized-size` of its key type (3.7)	1,024 bytes	new; the chain ABI's storage key limit

There is no limit on the size of a contract after macro expansion and instantiation, which can grow exponentially (definitions that each call the next twice, or instantiate it at two types). The only limit is the chain's limit on module size. A compiler MUST report a contract whose module would exceed it as a compile error, and SHOULD stop expanding as soon as the limit is passed. A language-level cap would not make validity compiler-independent, since near the limit whether a contract fits already depends on code generation (P5).

2.9 Open questions

An item tagged **RFC** needs its own design RFC, with a decidability analysis, before this chapter can be frozen. The others are decisions this draft can take once the dependency named in each is settled.

1. **Version numbering** (2.4.0). Whether `clarity-version` takes a single integer or a major and minor pair depends on how language releases are cut: any change to the prelude or to a rule is a new version for compatibility purposes, which argues for one number.
2. **RFC: explicit size parameters.** A reference kind such as `(width size)` that accepts a size term at the call site, with the rule that a size variable is usable as a `uint` value in the body. It would make `uint-to-buff-be` a single `(uint) -> (buff 16)` primitive and move the four fixed-width `*-to-buff-*` functions (3.4) into the prelude.
3. **RFC: static closures.** Function templates that capture only reference-kind parameters, never runtime values, whose instances are ordinary named functions resolved at instantiation. They would make a generic `fold-right`, a multi-sequence `map`, and `flip`-style adapters writable in the prelude without function values at runtime or in the ABI.
4. **RFC: bounded iteration.** The prelude drives several loops with constant lists used as counters (`std-decimal-iterations`, `std-exponent-bits`). A form that iterates a statically bounded number of times without a list, and that preserves P1, would remove the pattern. This is the "richer bounded iteration" item of the specification plan.

5. **RFC: host bindings in the prelude.** A `define-host` declaration form would let the prelude declare every host-calling primitive (storage, assets, context, chain data, signature schemes, `emit`, `log`) by name and signature, with semantics cited from the chain ABI rather than restated in chapter 3. The same form could cover compiled primitives with ordinary signatures, leaving chapter 3 with only the primitives whose typing cannot be written as a signature. Parked; it requires an admission rule on the ABI side.
6. **Static memory bound.** The intermediate size limit only has to stay below the chain's instance memory limit, so whether a well-typed contract's live values fit in memory depends on code generation, and exceeding it is a runtime error. The compiler could instead compute each function's peak memory and reject a contract that cannot fit, as it already does for module size (2.8). The chain's transaction memory cap, which depends on the callers' memory, would remain a runtime error.
7. **Formal presentation.** The judgment-style typing rules and the full EBNF are pending the questions above.

3. Primitive Operations

This chapter enumerates every primitive in the vNEXT core. A primitive is an operation that passes the layering rule in README.md: it needs the host, it cannot be expressed in the language, it is a cryptographic standard, or it is a base operation on a type's representation. Everything a contract author uses that is not listed here or in the special forms of 02-core-language.md is a standard-library definition.

Every primitive has a declared arity (P6). A signature whose last parameter is written `I...` or `S...` has a rest parameter (2.6.5): the primitive takes its fixed arguments plus one or more further arguments, or a single spread list (2.6.5), and the result is the left fold of the operation over them, so `(- a b c)` is `(- (- a b) c)`, `(- a b)` is ordinary subtraction, and `(+ 0 xs...)` is the sum of `xs`. Passed as a function argument, such a primitive is its two-argument operation: `(fold-left + 0 xs)`.

Notation otherwise follows 2.1: uppercase letters are type or size variables; "for `I` in {...}" denotes a family with one signature per listed type. Subtyping (2.3.2) applies to arguments, so a `(buff 4)` is accepted where `(buff 16)` is written.

Each entry gives the signature, semantics, and error conditions. Primitives have no cost entries (P4). A primitive the compiler lowers to Wasm costs the fuel of that code; a primitive that calls a host function is charged by the chain ABI. Which primitives are host functions is decided by the chain ABI and the compilation chapter, not here.

3.1 Arithmetic

For `I` in {`int`, `uint`}:

Primitive	Signature	Semantics	Runtime errors
`+`	`(I I...) -> I`	addition	overflow
`-`	`(I I...) -> I`	subtraction	overflow/underflow
`*`	`(I I...) -> I`	multiplication	overflow
`/`	`(I I...) -> I`	division truncating toward zero	divisor is zero; overflow (the most negative `int` divided by `-1`)
`mod`	`(I I) -> I`	remainder of `/`, with the sign of the dividend	as `/`
`to-int`	`(uint) -> int`	conversion	argument at least 2^127
`to-uint`	`(int) -> uint`	conversion	argument negative

Primitive	Signature	Semantics	Runtime errors
`bit-and`	`(I I...) -> I`	bitwise and	
`bit-or`	`(I I...) -> I`	bitwise or	
`bit-xor`	`(I I...) -> I`	bitwise exclusive or	
`bit-not`	`(I) -> I`	bitwise complement of the 128-bit pattern	
`bit-shift-left`	`(I uint) -> I`	shift left, discarding bits shifted out; the result reinterprets the 128-bit pattern as `I`	shift amount 128 or more
`bit-shift-right`	`(I uint) -> I`	shift right; arithmetic for `int`, logical for `uint`	shift amount 128 or more

Library, not primitive: `pow`, `sqrti`, `log2`, `max`, `min`, `abs`. Their error conditions are the same as legacy and are defined by their source.

3.2 Comparison and equality

For `C` in {`int`, `uint`, `(buff N)`, `(string-ascii N)`, `(string-utf8 N)`}:

Primitive	Signature	Semantics
`<`	`(C C) -> bool`	numeric order for integers; lexicographic element order for sequences
`<=`	`(C C) -> bool`	numeric order for integers; lexicographic element order for sequences
`>`	`(C C) -> bool`	numeric order for integers; lexicographic element order for sequences
`>=`	`(C C) -> bool`	numeric order for integers; lexicographic element order for sequences

Sequences compare element by element from the start; the first unequal pair decides, and a proper prefix is less than the longer sequence. Elements are ordered by unsigned byte value for `buff` and `string-ascii`, and by Unicode code point for `string-utf8`. Code point order is the order of the strings' UTF-8 encodings compared bytewise, so the result does not depend on how a host stores strings. No Unicode normalization is applied: `u"\u{E9}"` and `u"e\u{301}"` are unequal, and the first is greater.

Primitive	Signature	Semantics
`is-eq`	`(T T) -> bool`	structural equality. Both arguments are admitted at `lub` of their types; an undefined lub is a type error

3.3 Sequences

`S` ranges over the four sequence kinds; `S[N]` is that kind with bound `N`; `elem(S)` is its element type (2.3.1). All sequence primitives are kind-preserving: a `concat` of two buffers is a buffer.

Primitive	Signature	Semantics	Runtime errors
`len`	`(S[N]) -> uint`	current length (code points for `string-utf8`)	
`concat`	`(S[N] S[M]...) -> S[N + M1 + ... + Mk]`	concatenation, left to right; with rest arguments of bounds `M1 ... Mk`, the bound is their exact sum plus `N`. With a spread `(list K S[M])`, the bound is `N + K × M`	
`element-at?`	`(S[N] uint) -> (optional elem(S))`	element at zero-based index, or `none` if out of range	
`slice?`	`(S[N] uint uint) -> (optional S[N])`	elements in `[left, right)`; `none` if `left > right` or `right > len`	
`as-max-len?`	`(S[N] size) -> (optional S[M])`	the second argument is a size term `M` (2.3.1); `(some s)` if `len(s) <= M`, else `none`	
`map`	`((function (elem(S)) U) S[N]) -> (list N U)`	applies the function to each element in order	
`filter`	`((function (elem(S)) bool) S[N]) -> S[N]`	keeps elements for which the function returns `true`, preserving order	
`fold-left`	`((function (A elem(S)) A) A S[N]) -> A`	`(f (f (f init x1) x2) ... xn)`: elements left to right, accumulator first	
`fold-right`	`((function (elem(S) A) A) S[N] A) -> A`	`(f x1 (f x2 ... (f xn init)))`: elements right to left, accumulator second	

In both folds the initial value sits on the same side as the accumulator in the callback, so the argument order of the call mirrors the order of the callback's parameters. Neither fold can be written from the other: that needs a wrapper that swaps `f`'s arguments, and a function cannot be closed over. Legacy `fold` (left to right, element first) is removed; it is `fold-left` with the callback's parameters swapped.

Library, not primitive: `append`, `index-of?`, `replace-at?`. Legacy multi-sequence `map` (`(map f l1 l2)`) is dropped. It cannot be written in the library, because a fold would have to carry `f` in its accumulator. `filter` stays primitive for the same reason: a fold's step function cannot call the predicate, so the predicate's results would have to travel in the accumulator beside the kept elements. That accumulator is a parameter, held to the value size limit (2.3.1), and is larger than the result, so a library `filter` would reject inputs near the limit that the primitive accepts.

3.4 Conversions and serialization

Primitive	Signature	Semantics	Runtime errors
`buff-to-uint-be`	`((buff 16)) -> uint`	big-endian unsigned interpretation; the empty buffer is `u0`	

Primitive	Signature	Semantics	Runtime errors
`uint-to-buff-be`	`(uint size) -> (buff W)`	the second argument is a size term `W` (2.3.1), at most 16; big-endian encoding in exactly `W` bytes	value does not fit in `W` bytes
`uint-to-buff-le`	`(uint size) -> (buff W)`	little-endian counterpart	value does not fit
`int-to-buff-be`	`(int size) -> (buff W)`	two's-complement big-endian encoding in exactly `W` bytes	value does not fit in `W` signed bytes
`int-to-buff-le`	`(int size) -> (buff W)`	little-endian counterpart	value does not fit
`to-consensus-buff`	`(T) -> (buff serialized-size(T))`	canonical encoding of the value (SIP-005; see below). Total: the size limits apply to the result through its position (2.3.1), so a serialization that is only hashed or compared is bounded by the intermediate limit, and one that is bound, returned, or stored by the value limit, both statically. `T` MUST be storable	
`from-consensus-buff?`	`(type (buff N)) -> (optional T)`	the first argument is a type expression `T`, which MUST be storable. Deserializes; `none` if the bytes are not a canonical encoding of a value of type `T` (exact typed-tuple fields, no trailing bytes). Memory for the result is reserved against `serialized-size(T)` before decoding (2.7.7)	

Canonical encoding and `serialized-size(T)`, the largest encoding of any value of `T`, are defined by the interpretation layer (../08-value-interpretation.md sections 3 and 4.1). The compiler uses the same encoding for call arguments and results (2.5.8), storage (3.7), and NFT identifiers (3.8).

Bitcoin transaction parsing

Primitive	Signature	Semantics
`get-bitcoin-tx-output?`	`((buff N) uint) -> (response { script: (buff 1024), amount: uint, txid: (buff 32) } uint)`	parses raw Bitcoin transaction bytes (SegWit or not) and returns output `vout`, with `txid` in internal byte order; `err u1` the bytes do not deserialize as a transaction, `err u2` `vout` is out of range, `err u3` the output script exceeds 1024 bytes

This is pure parsing of a byte string, not a chain lookup, which is why it sits with the conversions. It is expressible in the language, so under the layering rule it is a candidate for the library; it stays primitive in this draft only because the parser (variable-length integers, SegWit markers, witness data) has not been written yet.

`buff-to-uint-be` is the base conversion from bytes to integers. The `*-to-buff-*` family, `uint-to-buff-be` included, stays primitive because each takes a size argument, which a library function cannot. Library, not primitive: `buff-to-uint-le`, `buff-to-int-be`, `buff-to-int-le`, `int-to-ascii`, `int-to-utf8`,

`string-to-int?`, `string-to-uint?`, and `to-ascii?`, with the same signatures as legacy. All but the signed buffer conversions also have the same results. `buff-to-int-be` and `buff-to-int-le` sign-extend from the buffer's actual length, so `(buff-to-int-be 0xff)` is `-1`; legacy zero-pads a short buffer and returns `255`, which makes the signed conversion of a short buffer disagree with the same bytes in a 16-byte buffer. This is a deliberate change, listed in the footguns table of chapter 1 and the disposition table of chapter 4.

The four `*-to-buff-*` primitives have no legacy counterpart. They are the write direction of the `buff-to-*` family, requested for hashing structured data (Merkle leaves, transcripts, commitments); legacy contracts slice the type-tagged output of `to-consensus-buff` instead, which is fragile and aliases values that share low bytes. Rejecting values that do not fit closes that aliasing class at the language level.

3.5 Principals

Primitive	Signature	Semantics
`principal-to-ascii`	`(principal) -> (string-ascii 82)`	the principal's text form: the c32check address of a standard principal, or the address, `.`, and the name of a contract principal (41 for the address, 1 for the dot, 40 for the name)
`contract-hash?`	`(principal) -> (response (buff 32) uint)`	SHA-512/256 of the deployed contract's module; `err u1` if the principal is a standard principal, `err u2` if no such contract is deployed. The chain never holds source; the source hash is available from the module's provenance section, off chain

Library, not primitive: `principal-construct?`, `principal-destruct?`, `is-current-network`, `is-standard`, `principal-of?`.

`principal-to-ascii` is new. It is the principal case of `to-ascii?`, which is otherwise library (3.4). It is expressible, so under the layering rule it is a candidate for the library; it stays primitive in this draft only because the c32check encoder has not been written yet (3.12).

3.6 Cryptography

Hash functions accept buffers only (P6, footguns table). Library, not primitive: `hash160` (`ripemd160` of `sha256`) and `verify-merkle-proof` (a composition of `sha256`), both in chapter 4.

`ripemd160` is new. Legacy exposes RIPEMD-160 only inside `hash160`; exposing it on its own lets `hash160` be library code like every other composition.

Primitive	Signature	Semantics
`ripemd160`	`((buff N)) -> (buff 20)`	RIPEMD-160
`sha256`	`((buff N)) -> (buff 32)`	SHA-256

Primitive	Signature	Semantics
`sha512`	`((buff N)) -> (buff 64)`	SHA-512
`sha512/256`	`((buff N)) -> (buff 32)`	SHA-512/256 (distinct IV; not a truncation of SHA-512)
`keccak256`	`((buff N)) -> (buff 32)`	Keccak-256
`secp256k1-recover?`	`((buff 32) (buff 65)) -> (response (buff 33) uint)`	recovers the compressed public key from a message hash and a recoverable signature; `err u1` malformed or non-canonical signature, `err u2` recovery failed
`secp256k1-verify`	`((buff 32) (buff 64) (buff 33)) -> bool`	verifies a 64-byte signature against a compressed public key; `false` for a signature that is not exactly 64 bytes or is non-canonical. A recoverable signature is verified by dropping its recovery byte with `slice?`
`secp256k1-decompress?`	`((buff 33)) -> (response (buff 65) uint)`	uncompressed form of a compressed public key
`secp256r1-verify`	`((buff 32) (buff 64) (buff 33)) -> bool`	P-256 signature verification; `false` for a non-canonical signature
`ed25519-verify`	`((buff N) (buff 64) (buff 32)) -> bool`	Ed25519 signature verification over an arbitrary-length message

Canonical signatures. An ECDSA signature (secp256k1 or P-256) is canonical only if its `S` component is in the lower half of the curve order (low-S). The verify primitives MUST return `false`, and `secp256k1-recover?` MUST return `err u1`, for a non-canonical signature. Accepting both `S` and `n - S` would let anyone replay a signature-gated authorization with a different signature encoding.

3.7 Contract storage

Primitive	Signature	Semantics
`var-get`	`((var T)) -> T`	current value
`var-set`	`((var T) T) -> bool`	stores the value; returns `true`
`map-get?`	`((map K V) K) -> (optional V)`	lookup
`map-set`	`((map K V) K V) -> bool`	inserts or overwrites; returns `true`
`map-delete`	`((map K V) K) -> bool`	removes the entry; returns `true` if it existed

Library, not primitive: `map-insert?` (`map-get?` then `map-set`). A membership test is `(is-some (map-get? m k))`; a compiler MAY lower it to an existence check that does not read the value, since the result is identical (P5).

Layout. The chain's contract storage is an untyped key-value store (../07-chain-abi.md section 4.2). The compiler lays each data variable and map out in it by the interpretation layer's storage layout (../08-value-interpretation.md section 7.2), and declares each one in the `storage` field of `abi.types` (2.4.4), so a tool can read a Clarity contract's state without its source. A definition's key prefix is a kind byte, `0x00` for a data variable and `0x01` for a map, then the name's length as one byte, then the name. The length byte keeps any prefix from being a prefix of another, as the layout requires.

Definition	Store key	Store value
data var	the prefix	canonical encoding of the value
map	the prefix, then the canonical encoding of the key	canonical encoding of the entry

Canonical encoding is the interpretation layer's (../08-value-interpretation.md section 3), the same as `to-consensus-buff` (3.4). A map entry that does not exist has no store key, so `map-delete` removes the store key and `map-get?` returns `none` when there is none. A data variable is set at deploy (2.4.1) and always has a value.

3.8 Assets

All asset primitives record movements in the transaction's asset map (for post-conditions and allowances) and emit events.

Primitive	Signature	Semantics	`err` codes
`ft-mint?`	`((fungible-token) uint principal) -> (response bool uint)`	mints `amount` to the recipient; exceeding the supply cap is a runtime error, as in legacy	`u1` amount is zero
`ft-burn?`	`((fungible-token) uint principal) -> (response bool uint)`	burns `amount` from the owner	`u1` insufficient balance or amount is zero
`ft-transfer?`	`((fungible-token) uint principal principal) -> (response bool uint)`	moves `amount` from sender to recipient	`u1` insufficient balance; `u2` sender is recipient; `u3` amount is zero
`ft-get-balance`	`((fungible-token) principal) -> uint`	balance	
`ft-get-supply`	`((fungible-token)) -> uint`	circulating supply	
`nft-mint?`	`((non-fungible-token T) T principal) -> (response bool uint)`	mints the identifier to the recipient	`u1` already exists
`nft-burn?`	`((non-fungible-token T) T principal) -> (response bool uint)`	burns the identifier owned by the owner	`u1` not owned by owner; `u3` does not exist

Primitive	Signature	Semantics	`err` codes
`nft-transfer?`	`((non-fungible-token T) T principal principal) -> (response bool uint)`	changes the owner	`u1` not owned by sender; `u2` sender is recipient; `u3` does not exist
`nft-get-owner?`	`((non-fungible-token T) T) -> (optional principal)`	owner	
`stx-transfer?`	`(uint principal principal (optional (buff 34))) -> (response bool uint)`	moves uSTX with an optional memo (legacy `stx-transfer?` and `stx-transfer-memo?` merged)	`u1` insufficient balance; `u2` sender is recipient; `u3` amount is zero; `u4` sender is not the context's `sender`
`stx-burn?`	`(uint principal) -> (response bool uint)`	burns uSTX	`u1` insufficient balance; `u3` amount is zero; `u4` sender is not the context's `sender`
`stx-account`	`(principal) -> { locked: uint, unlock-height: uint, unlocked: uint }`	STX account state	

`stx-transfer?` and `stx-burn?` require the `sender` argument to be the context's `sender` field (3.9), returning `err u4` otherwise. The `ft-*` and `nft-*` primitives do not check the sender; every movement is recorded in the asset map and is subject to transaction post-conditions and to any enclosing `as-contract?` or `restrict-assets?` allowance frame. The asymmetry is deliberate: STX is the chain's own asset and keeps the strict check, while a token's contract decides its own transfer rules, with post-conditions and allowances as the holder's protection.

The chain stores and compares non-fungible token identifiers as bytes (../07-chain-abi.md section 6.2). The compiler passes an identifier's canonical encoding (3.4) and declares the token's identifier type in `abi.types` (2.4.4), so a Clarity token's identifiers have the same bytes as in legacy, in transaction post-conditions, and in `with-nft` allowances (2.5.9), and tools display them by their type.

The chain records a `stx-transfer?` memo as bytes and treats an empty memo as no memo (../07-chain-abi.md section 6.2), so `(some 0x)` and `none` produce the same event.

3.9 Transaction and chain context

The execution context is one record, read with a single primitive. Legacy's twelve bare context names (`tx-sender`, `stacks-block-height`, and so on) are its fields; `(get sender (get-current-context))` replaces `tx-sender`.

Primitive	Signature	Semantics
`get-current-context`	`() -> context`	the context of the current call frame; the fields are below. A compiler MAY lower `(get f (get-current-context))` to a read of that field alone

```
context = { sender: principal, caller: principal, sponsor: (optional principal),
            contract: principal,
            stacks-height: uint, burn-height: uint, tenure-height: uint,
            time: uint, liquid-supply: uint,
            chain-id: uint, is-mainnet: bool, is-regtest: bool }
```

Field	Value
`sender`	the transaction sender, or the current contract inside `as-contract?`
`caller`	the principal that made the current call: the sender for the entry call, the calling contract for `contract-call?`, the current contract inside `as-contract?`
`sponsor`	the sponsor, if the transaction is sponsored
`contract`	the contract being executed
`stacks-height`	height of the current Stacks block
`burn-height`	height of the burnchain block the current Stacks block is anchored to
`tenure-height`	number of tenures elapsed
`time`	timestamp of the current Stacks block (seconds)
`liquid-supply`	total liquid uSTX
`chain-id`	chain identifier
`is-mainnet`	true on mainnet
`is-regtest`	true in regtest environments

`sender`, `caller`, and `contract` are frame-scoped: `as-contract?` (2.5.9) and `contract-call?` change them for the frames they create. The other fields are the same in every frame of a transaction.

Context and chain data are records rather than properties selected by name, so their typing is uniform and there is no property-name mini-language.

Lookups are total. Every chain-data lookup returns `none` for any height or input the host cannot serve, including heights before the chain's first block and data that has not yet matured. A chain-data lookup MUST NOT raise a runtime error and MUST NOT make the transaction unmineable. (Legacy treated some such lookups as "problematic" transactions that miners drop, which desynchronizes the sender's nonce.)

Primitive	Signature	Semantics
`get-stacks-block-info?`	`(uint) -> (optional { id-header-hash: (buff 32), header-hash: (buff 32), time: uint })`	`none` if the height is not less than the current height: the current block's hashes do not exist until it is sealed, after this transaction runs. The current block's time is the context's `time` field

Primitive	Signature	Semantics
`get-tenure-info?`	`(uint) -> (optional { burnchain-header-hash: (buff 32), miner-address: principal, time: uint, vrf-seed: (buff 32), block-reward: (optional uint), miner-spend-total: uint, miner-spend-winner: uint })`	serves every tenure up to and including the current one (the context's `tenure-height`), since all fields are fixed at sortition. `block-reward` is `none` until the reward has matured. Maturity is defined by the host in terms of the coinbase (tenure) height at which the reward was recorded, so the lookup is total: it never fails for a height the host has no row for
`get-burn-block-info?`	`(uint) -> (optional { header-hash: (buff 32), pox-addrs: { addrs: (list 2 { hash-bytes: (buff 32), version: (buff 1) }), payout: uint } })`	serves every burn block up to and including the current one (the context's `burn-height`); `none` if the burn height is outside the known range. The `hashbytes` field of legacy is `hash-bytes` (P6)

3.10 Events and control

Primitive	Signature	Semantics
`emit`	`((event T) T) -> bool`	emits the value as an event of the named definition (2.4.1); returns `true`
`log`	`(T) -> T`	records the value as debug output, which the chain discards; returns the value
`panic`	`() -> never`	aborts the transaction with a runtime error. The library's `unwrap-panic` and `unwrap-err-panic` are defined with it

`emit`'s first argument names a `define-event` of the current contract, or an `(event T)` reference parameter (2.3.7), and its second has a type that is a subtype of `T`: the event's declared type when the first argument names a definition, or the parameter's type when it names a parameter, which may be narrower (2.3.7). The chain records the event as the emitting contract, the event's name, and the value's canonical encoding (../07-chain-abi.md sections 6.2 and 8.3), which an indexer decodes against the event's type in `abi.types` (../08-value-interpretation.md section 7.1). The declared type's `serialized-size` is within the value size limit (2.8), which is also the chain's limit on one event's data, so no single `emit` is too large. The chain also caps the event bytes of a whole transaction, asset events included, and an `emit` or asset operation that passes that cap aborts the transaction with an `events-exceeded` runtime error (../07-chain-abi.md sections 8.3 and 10.2).

```
(define-map listings uint { seller: principal, price: uint })
(define-event listed { id: uint, seller: principal, price: uint })

(define-public (list-item (id uint) (price uint))
  (let ((seller (get sender (get-current-context))))
    (asserts! (is-none (map-get? listings id)) (err u1))
    (map-set listings id { seller: seller, price: price })
    (emit listed { id: id, seller: seller, price: price })
    (ok true)))
```

`emit` replaces legacy `print`. A legacy `print` event carries the value's SIP-005 encoding with no name or declared type; an event's data are the same bytes for the same value, so one decoder reads both.

`log` is for debugging, which legacy `print` also served. It takes a value of any type, bounded like any primitive's argument by the intermediate size limit (2.3.1), and passes the value's canonical encoding to the chain's `log` host function (../07-chain-abi.md section 6.2). The chain charges for the call and discards the output: it is not an event, is in no receipt, and does not count toward the transaction's event bytes. A simulator or development network may show it, decoded by its self-describing encoding. Since `log` returns its argument, it can wrap any subexpression: `(+ (log (get price listing)) fee)`.

3.11 Count

Group	Primitives
Arithmetic	13 (each in two integer families)
Comparison and equality	5
Sequences	9
Conversions	8
Principals	2
Cryptography	10
Storage	5
Assets	12
Context	4
Events and control	3
Total	**71**

For comparison, legacy Clarity 6 has 126 native functions and 16 native variables, most with bespoke typing rules.

3.12 Open questions

An item tagged **RFC** needs its own design RFC before this chapter can be frozen. Issue numbers refer to the stacks-core tracker; the dispositions are in ../06-issue-tracker.md.

1. **RFC: Schnorr verification** (#3568). BIP-340 over secp256k1, proposed as `secp256k1-schnorr-verify : ((buff 32) (buff 64) (buff 32)) -> bool`, `false` on malformed input, in 3.6. It passes the cryptographic-standard test and is the natural fit for Taproot and FROST. Recommended for vNEXT.
2. **RFC: 256-bit arithmetic** (#3975). Either a library family over `(buff 32)` (expressible, so library under the layering rule, replaceable by faster Wasm later) or a new integer type, which is a core change. If Schnorr ships as a primitive, the main motivation goes away.

3. **RFC: per-definition code hashes** (#6320). `contract-hash?` covers the whole module. A per-definition hash would be over compiled code, and so compiler-dependent, unless the module declares per-definition source hashes in its `abi.types` section or its provenance. The RFC chooses.
4. **Bitcoin transaction parser in the prelude** (3.4). `get-bitcoin-tx-output?` is expressible as a byte-at-a-time fold and belongs in the library under the layering rule. It stays primitive until the parser is written and has its own conformance vectors.
5. **c32check encoder in the prelude** (3.5). `principal-to-ascii` is expressible with `principal-destruct?`, `sha256`, and base-32 regrouping of the hash and checksum, and belongs in the library under the layering rule. It stays primitive until the encoder is written and has its own conformance vectors.

4. Standard Library

The standard library is vNEXT source, linked into modules by the compiler as a prelude that is in scope in every contract (2.4.5). Its normative definition is its source: std/prelude.clar. This chapter explains how it is delivered, what it contains and why, and how every legacy Clarity construct maps onto vNEXT.

4.1 Delivery and status

- The prelude is part of the language version. It is frozen when the language version is frozen and never changes for that version.
- Prelude names are reserved: a contract MUST NOT define a top-level name that the prelude defines. This includes the `std-*` helper definitions.
- The prelude uses only features available to every contract (P3). It is type-checked, instantiated, and compiled by the same rules as user code.
- **Results are defined by the source.** Conformance vectors for library functions are derived from the prelude source. If a vector and the source disagree, the vector is wrong.
- **Linked into each module.** The compiler includes the library instances a contract uses in that contract's module. A library call costs the fuel of that code (P4, P5). The chain holds no library code of its own.
- **Optimization replaces the implementation, not the definition.** A later compiler MAY link a hand-written or better compiled Wasm implementation of a library function. It MUST produce identical results, events, and errors to the source for every input, which the conformance suite checks. Contracts already deployed keep the code they were deployed with, so a library change never alters them and no chain upgrade is involved. (Their cost can still change at an epoch boundary if the chain's fuel schedule does; see ../03-architecture.md section 4.)

The source is written for correctness and readability, not speed. Where a definition is slow (a digit-by-digit conversion, a bitwise square root), that is a candidate for a Wasm replacement, not a reason to move it into the core.

4.2 What belongs in the library

Everything that can be written in the language. The layering rule (README.md) keeps in the core only host access, what cannot be expressed, cryptographic standards, and the base operations on each type's representation. Every other construct a contract author uses is here.

Section	Definitions	Built from
Branching	`if`, `not`	`match` on `bool`
Early exit	`asserts!`, `unwrap!`, `unwrap-err!`, `try!`, `unwrap-panic`, `unwrap-err-panic`, `default-to`	`match`, `return`, `panic`

Section	Definitions	Built from
Predicates	`is-some`, `is-none`, `is-ok`, `is-err`	`match`
Arithmetic	`max`, `min` (rest parameters), `abs`	comparison, `-`, `fold-left`
Arithmetic	`pow`, `sqrti`, `log2`	`fold-left` over a constant list of bit positions, with shifts
Sequences	`append`, `replace-at?`	`concat`, `slice?`, `as-max-len?`
Sequences	`index-of?`	`fold-left` with the target in the accumulator
Conversions	`buff-to-uint-le`, `buff-to-int-be`, `buff-to-int-le`	`buff-to-uint-be`, a byte-reversing `fold-left`, two's-complement arithmetic sign-extended from the actual length
Conversions	`int-to-ascii`, `int-to-utf8`, `string-to-int?`, `string-to-uint?`	digit tables, `element-at?`, `index-of?`, a 39-step `fold-left`
Conversions	`to-ascii?`	`int-to-ascii`, `principal-to-ascii`, `map` with a hex or character table, `concat` with a spread
Principals	`principal-construct?`, `principal-destruct?`	`to-consensus-buff`, `from-consensus-buff?`, `slice?`, `get-current-context`
Principals	`is-current-network`, `is-standard`, `principal-of?`	`principal-destruct?`, `principal-construct?`, `hash160`
Cryptography	`hash160`	`ripemd160`, `sha256`
Cryptography	`verify-merkle-proof`	`sha256`, `log2`, `fold-left` over the proof
Storage	`map-insert?`	`map-get?`, `map-set`

About thirty-five public definitions, plus `std-*` helpers. Two conventions decide whether a prelude function is `define-macro` or `define-private`:

- Anything that needs laziness or early exit is a macro: `if` and the early-exit family.
- Everything else is private. Private functions are compiled once per instance and can be passed to `map`, `filter`, `fold-left`, and `fold-right`.

Notes on specific definitions

- **Families over integer or sequence kinds.** `pow`, `sqrti`, `log2`, `int-to-ascii`, and `int-to-utf8` are overloaded on `int` and `uint`. `replace-at?` is overloaded on the four sequence kinds. `max`, `min`, `index-of?`, `string-to-int?`, and `string-to-uint?` are single generic definitions checked per instance (2.6.2); per-kind constants such as digit tables are selected by small overloaded helpers.
- **`pow` does not overflow early.** It squares and multiplies from the most significant exponent bit down, so every intermediate is the base raised to a prefix of the exponent and no step overflows unless the result does.
- **`verify-merkle-proof`** takes a leaf hash, a root hash, the leaf index, the transaction count, and up to 24 sibling hashes from leaf to root, all in internal byte order, and uses double SHA-256 with Bitcoin's

odd-row duplication rule. It returns `false` for any proof inconsistent with the canonical tree shape implied by the count: an index not below the count, a path length other than the tree depth, or a sibling that must equal its partner on an odd row but does not. This contract is enforced by the source; it is security-sensitive and needs the densest conformance vectors in the library.

Kept primitive although expressible

The representation-base test names a chosen set, not a minimal one, so some of its members could be written in terms of the others or of other primitives: `len` and `slice?` as folds, `element-at?` through `slice?`, `buff-to-uint-be` as a fold over the bytes, the shifts as multiplication and division by powers of two, `to-int` and `to-uint` through the `*-to-buff-*` primitives and deserialization, comparison on sequences as a fold, `<`, `>`, and `>=` over `<=` and `not`, `bit-xor`, `bit-or`, and `bit-not` over `bit-and`, `+`, and `-`, and `mod` over `/`, `*`, and `-`. They stay core because the spec chooses them as the vocabulary the library is written in, not because of how a compiler implements them. Everything else written over that vocabulary is library, such as `max`, `min`, `abs`, and the little-endian conversions over `buff-to-uint-be`.

`secp256k1-verify` can be written with `secp256k1-recover?`, by trying the four recovery ids against the key. Both are operations the scheme's standard defines, so both are core by the cryptographic-standard test, which keeps the language from reimplementing either one in terms of the other.

`principal-construct?` and `principal-destruct?` are library. They build and slice the principal's encoding (chain ABI 2.2) with `to-consensus-buff` and `from-consensus-buff?`, whose decoding checks the version byte and the contract name, and they read the network from `get-current-context` against the prelude's version-byte constants, as `principal-of?` does. `principal-to-ascii` is also expressible, since c32check is base-32 text with a double-SHA-256 checksum, and passes none of the four tests, so it belongs here. Like `get-bitcoin-tx-output?`, it stays primitive only until it is written (chapter 3.12).

4.3 Disposition of every legacy construct

These tables cover every native function, define form, and native variable in Clarity 6. Constructs with no legacy counterpart are listed in 4.4. The vNEXT column says what each became:

- **primitive**: stays in the core, with the signature in 03-primitives.md.
- **form**: a special form of the core language, in 02-core-language.md. **Allowance forms** are the allowance syntax of `as-contract?` and `restrict-assets?` (2.5.9).
- **sugar**: a fixed rewrite into other syntax, with no definition of its own; `begin` is `let` with no bindings.
- **literal**: a literal of the lexical grammar (2.2).
- **library**: defined in the prelude, in vNEXT source.
- **removed**: vNEXT has no equivalent by that name; the notes say what to use.

In the notes, **private** and **macro** say how the prelude defines a library function, not who can call it: every prelude function is in scope in every contract (2.4.5), and none is callable by `contract-call?`. A private function is an ordinary function (`define-private`), so all of its arguments are evaluated once, before the call, as in legacy. A macro (`define-macro`, 2.6.3) is expanded at the call site and evaluates an argument only where its body uses it.

Arithmetic and logic

Legacy	vNEXT	Notes
`+`, `-`, `*`, `/`	primitive	rest parameter (2.6.5); at least two arguments, so unary `-` is written `(- 0 x)`
`mod`	primitive	
`pow`, `sqrti`, `log2`	library	same results as legacy
`<=`	primitive	
`>=`, `<`, `>`	primitive	
`to-int`, `to-uint`	primitive	
`xor`	removed	use `bit-xor`
`bit-and`	primitive	rest parameter
`bit-xor`, `bit-or`	primitive	rest parameter
`bit-not`	primitive	
`bit-shift-left`, `bit-shift-right`	primitive	shift of 128 or more aborts instead of wrapping
`and`, `or`	form	two or more arguments, short-circuit
`not`	library	private
`is-eq`	primitive	binary only; `(is-eq a b c)` is written `(and (is-eq a b) (is-eq a c))`
`if`	library	macro over `match`

Binding, sequencing, branching, early exit

Legacy	vNEXT	Notes
`let`	form	
`begin`	sugar	`(let () ...)`
`match`	form	constructor patterns `((some x) ...)`, `(none ...)`, `((ok x) ...)`, `((err e) ...)`, `(true ...)`, `(false ...)`
`asserts!`, `unwrap!`, `unwrap-err!`, `try!`	library	macro over `match` and `return`; `thrown` is evaluated only on failure, which for `unwrap!` and `unwrap-err!` is a change from legacy (legacy evaluates it first); not usable in top-level expressions or constant initializers, where `return` is forbidden (2.4.2)
`unwrap-panic`, `unwrap-err-panic`	library	macro over `match` and `panic`
`default-to`	library	private; both arguments evaluated, as in legacy

Legacy	vNEXT	Notes
`is-ok`, `is-err`, `is-some`, `is-none`	library	private
`ok`, `err`, `some`	form	constructors
`none`, `true`, `false`	literal	

Sequences

Legacy	vNEXT	Notes
`list`	form	variadic, like `and` and `or`; functions use rest parameters
`len`, `concat`, `slice?`, `as-max-len?`	primitive	`concat` has a rest parameter
`element-at?`	primitive	
`index-of?`, `replace-at?`	library	
`element-at`, `index-of`	removed	aliases
`append`	library	`(concat items (list item))`
`map`	primitive	single sequence only; multi-sequence form dropped (chapter 3.3)
`filter`	primitive	
`fold`	removed	`fold-left` with the callback's parameters swapped (accumulator first)

Tuples

Legacy	vNEXT	Notes
`tuple`	removed	use `{ f: v, ... }`
`get`, `merge`	form	

Conversions and serialization

Legacy	vNEXT	Notes
`buff-to-uint-be`	primitive	
`buff-to-uint-le`	library	over `buff-to-uint-be`
`buff-to-int-be`, `buff-to-int-le`	library	over `buff-to-uint-be`; a short buffer is sign-extended from its actual length, where legacy zero-padded it (chapter 3.4)

Legacy	vNEXT	Notes
`int-to-ascii`, `int-to-utf8`	library	
`string-to-int?`, `string-to-uint?`	library	
`to-consensus-buff?`	primitive	renamed `to-consensus-buff`, total: legacy's `none` for a result over the value size limit is replaced by the static size limits on the result's position (2.3.1)
`from-consensus-buff?`	primitive	
`to-ascii?`	library	six overloads (`int`, `uint`, `bool`, `principal`, `buff`, `string-utf8`) with legacy's text and bounds, except that the principal bound is 82 where legacy's is 170, because contract names are limited to 40 characters (2.2); the principal case uses the new `principal-to-ascii` primitive (chapter 3.5). A `string-utf8` containing U+000C (form feed) is now `err u1`: legacy accepted it, but a vNEXT `string-ascii` cannot hold it (2.2)

Principals

Legacy	vNEXT	Notes
`principal-construct?`	library	single signature with `(optional (string-ascii 40))` name; `error_code` and `value` fields renamed `error-code` and `principal`; builds the encoding and decodes it with `from-consensus-buff?`
`principal-destruct?`	library	slices `to-consensus-buff`
`is-standard`	library	renamed `is-current-network`, since it tests the version byte against the current network and is true for contract principals too; vNEXT's `is-standard` tests for a standard (not contract) principal
`principal-of?`	library	`hash160` plus `principal-construct?` with the network's version byte
`contract-of`	removed	`(trait X) <: principal` (2.3.2), so a trait value is already a principal wherever one is expected; `(as principal t)` states it explicitly
`contract-hash?`	primitive	

Cryptography

Legacy	vNEXT	Notes
`sha256`, `sha512`, `sha512/256`, `keccak256`	primitive	buffers only; integer inputs removed

Legacy	vNEXT	Notes
`hash160`	library	`ripemd160` of `sha256`, over buffers only; integer inputs removed
`secp256k1-recover?`, `secp256k1-verify`, `secp256k1-decompress?`	primitive	low-S required; `secp256k1-verify` takes a `(buff 64)` signature only, where legacy also accepted 65 bytes and ignored the recovery byte
`secp256r1-verify`, `ed25519-verify`	primitive	low-S required for ECDSA
`verify-merkle-proof`	library	over `sha256`

Storage

Legacy	vNEXT	Notes
`var-get`, `var-set`	primitive	
`map-get?`, `map-set`, `map-delete`	primitive	
`map-insert`	library	renamed `map-insert?`; returns `(optional V)`: the existing value if the key was present (nothing written), `none` if inserted, where legacy returned `bool`; private, so both arguments are evaluated; written with a `(map K V)` reference parameter (2.3.7)

Assets

Legacy	vNEXT	Notes
`ft-mint?`, `ft-burn?`, `ft-transfer?`, `ft-get-balance`, `ft-get-supply`	primitive	
`nft-mint?`, `nft-burn?`, `nft-transfer?`, `nft-get-owner?`	primitive	
`stx-transfer?`	primitive	takes `(optional (buff 34))` memo
`stx-transfer-memo?`	removed	merged into `stx-transfer?`
`stx-burn?`, `stx-account`	primitive	
`stx-get-balance`	removed	`(get unlocked (stx-account p))`; a one-line alias (P6)

Calls, context, and allowances

Legacy	vNEXT	Notes
`contract-call?`	form	

Legacy	vNEXT	Notes
`as-contract?`	form	
`as-contract`	removed	unrestricted asset access
`restrict-assets?`	form	takes `allowance` forms only; `with-all-assets-unsafe` is accepted by `as-contract?` alone, as its only allowance (2.5.9)
`with-stx`, `with-ft`, `with-nft`, `with-staking`, `with-pox`, `with-all-assets-unsafe`	allowance forms	
`with-stacking`	removed	renamed `with-staking` in Clarity 6
`at-block`	removed	disabled in legacy since epoch 3.4
`print`	removed	for events, declare the event with `define-event` and emit it with `emit` (3.10); its data are the same SIP-005 bytes `print` emitted. For debugging, `log` (3.10)

Chain data

Legacy	vNEXT	Notes
`get-stacks-block-info?`, `get-tenure-info?`, `get-burn-block-info?`	primitive	return the whole record; property-name argument removed; `get-tenure-info?` and `get-burn-block-info?` serve the current tenure and burn block; the `hashbytes` field is `hash-bytes`
`get-block-info?`	removed	already removed in Clarity 3
`get-bitcoin-tx-output?`	primitive	

Define forms

Legacy	vNEXT	Notes
`define-constant`, `define-data-var`, `define-map`	form	
`define-fungible-token`, `define-non-fungible-token`	form	
`define-public`, `define-read-only`, `define-private`	form	private may be generic and overloaded
`define-trait`, `impl-trait`	form	trait identifiers replace `<name>` syntax
`use-trait`	removed	`(define-type name (trait id))`

Native variables

Legacy	vNEXT	Notes
`tx-sender`, `contract-caller`, `tx-sponsor?`, `current-contract`	removed	fields `sender`, `caller`, `sponsor`, `contract` of `(get-current-context)` (3.9); `(get sender (get-current-context))` replaces `tx-sender`

Legacy	vNEXT	Notes
`burn-block-height`, `stacks-block-height`, `tenure-height`, `stacks-block-time`	removed	fields `burn-height`, `stacks-height`, `tenure-height`, `time` of `(get-current-context)`
`chain-id`, `is-in-mainnet`, `is-in-regtest`, `stx-liquid-supply`	removed	fields `chain-id`, `is-mainnet`, `is-regtest`, `liquid-supply` of `(get-current-context)`
`block-height`	removed	already removed in Clarity 3
`none`, `true`, `false`	literal	

4.4 New in vNEXT

Constructs with no legacy counterpart, with the kinds of 4.3:

Construct	Kind	Notes
`clarity-version`	form	required first form of every contract (2.4.0); replaces the `clarity_version` field of the legacy deploy transaction
`define-type`	form	transparent type aliases (2.3.6); replaces `use-trait`
`define-macro`	form	call-by-name, expanded at each call site (2.6.3)
`define-event`	form	a named contract event with a declared data type (2.4.1); replaces ad hoc `print` events
rest parameters `(name T...)`	form	private functions only; the rest arguments are bound as a list (2.6.5)
`return`	form	exits the enclosing function with a value (2.5.4); the library's early-exit functions are defined with it
`as`	form	type ascription; widens, never narrows (2.5.10)
reference kinds `(var ...)`, `(map ...)`, `(fungible-token)`, `(non-fungible-token ...)`, `(event ...)`, `(function ...)`	form	parameters that name a definition of the contract rather than hold a value (2.3.7)
`emit`	primitive	emits a declared event (3.10); replaces `print` for events
`log`	primitive	debug output of a value of any type, which the chain discards (3.10); replaces `print` for debugging
`panic`	primitive	aborts the transaction (3.10); the library's `unwrap-panic` and `unwrap-err-panic` are defined with it
`fold-left`, `fold-right`	primitive	replace legacy `fold` (3.3)

Construct	Kind	Notes
`uint-to-buff-be`, `uint-to-buff-le`, `int-to-buff-be`, `int-to-buff-le`	primitive	fixed width, abort if the value does not fit (3.4, issue #7312)
`principal-to-ascii`	primitive	the text form of a principal (3.5)
`get-current-context`	primitive	one record for the whole execution context, replacing the legacy native variables (3.9); the twelve bare names are no longer reserved
`max`, `min`, `abs`	library	common user-written helpers; `max` and `min` take rest parameters

Further candidates from the issue tracker are collected in ../06-issue-tracker.md.

4.5 Open questions

1. **User overloads of prelude names.** 2.4.5 forbids a contract from defining any name the prelude defines, which also forbids adding an overload, such as `to-ascii?` for a contract's own tuple type. Allowing overloads of prelude names, while still forbidding redefinition of an existing overload, would be backward compatible to add later and is left out of this draft.

Appendix: `spec/std/prelude.clar`

```
;; Clarity vNEXT standard library prelude (working draft)
;;
;; This file is ordinary vNEXT source. It uses only features available to
;; every contract: generics (implicit type variables), macros with
;; call-by-name parameters, overloading, static references, `match`,
;; `return`, and the primitives in ../03-primitives.md.
;;
;; Scope: everything a contract author uses that can be written in the
;; language (layering rule, README.md). Definitions are written for clarity
;; and correctness, not speed. Cost is the Wasm fuel of the compiled code, so
;; a faster Wasm implementation can replace any definition later without
;; changing its results (chapter 4.1).
;;
;; Conventions:
;;   define-macro   for anything needing laziness or early exit
;;   define-private for everything else
;;   fold-left      calls (step accumulator element), so helpers take acc first
;;   (name T...)    a rest parameter: the extra arguments as a list (2.6.5)
;;   std-*          helper definitions; reserved like every prelude name
;;
;; Generic bodies are checked per instance (template semantics, chapter
;; 2.6.2). Where a comment says "`T` is instantiated at ...", any other
;; instantiation is a type error at the call site.
;;
;; Each value parameter of a macro is referenced at most once in
;; its body; bodies that need a value twice bind it with `let` first.

;; ---------------------------------------------------------------------------
;; Branching
;; ---------------------------------------------------------------------------

;; `if` is sugar over `match` on bool. `and` and `or` are core forms, since
;; short-circuiting over any number of arguments cannot be written here
;; (chapter 2.5.3).
(define-macro (if (condition bool) (then T) (else T))
  (match condition
    (true then)
    (false else)))

;; Private so it can be passed to `filter`.
(define-private (not (value bool))
  (if value false true))

;; ---------------------------------------------------------------------------
;; Early exit
;; ---------------------------------------------------------------------------

(define-macro (asserts! (condition bool) (thrown E))
  (if condition true (return thrown)))

;; `thrown` is evaluated only on failure. For `unwrap!` and `unwrap-err!`
;; this is a change from legacy, which evaluated it before the check;
;; `asserts!` was already lazy.
(define-macro (unwrap! (value (optional T)) (thrown E))
  (match value
    ((some inner) inner)
```

```
    (none (return thrown))))

(define-macro (unwrap! (value (response T E)) (thrown F))
  (match value
    ((ok inner) inner)
    ((err ignored) (return thrown))))

(define-macro (unwrap-err! (value (response T E)) (thrown F))
  (match value
    ((ok ignored) (return thrown))
    ((err inner) inner)))

(define-macro (try! (value (optional T)))
  (match value
    ((some inner) inner)
    (none (return none))))

(define-macro (try! (value (response T E)))
  (match value
    ((ok inner) inner)
    ((err inner) (return (err inner)))))

(define-macro (unwrap-panic (value (optional T)))
  (match value
    ((some inner) inner)
    (none (panic))))

(define-macro (unwrap-panic (value (response T E)))
  (match value
    ((ok inner) inner)
    ((err ignored) (panic))))

(define-macro (unwrap-err-panic (value (response T E)))
  (match value
    ((ok ignored) (panic))
    ((err inner) inner)))

;; Private, so both arguments are evaluated before the call, as in legacy.
(define-private (default-to (default T) (value (optional T)))
  (match value
    ((some inner) inner)
    (none default)))

;; ---------------------------------------------------------------------------
;; Predicates (private so they work with `filter`)
;; ---------------------------------------------------------------------------

(define-private (is-some (value (optional T)))
  (match value ((some ignored) true) (none false)))

(define-private (is-none (value (optional T)))
  (match value ((some ignored) false) (none true)))

(define-private (is-ok (value (response T E)))
  (match value ((ok ignored) true) ((err ignored) false)))

(define-private (is-err (value (response T E)))
  (match value ((ok ignored) false) ((err ignored) true)))

;; ---------------------------------------------------------------------------
;; Arithmetic
;; ---------------------------------------------------------------------------
```

```
;; `T` is any type `<=` accepts: int, uint, or a buffer or string type, which
;; compare lexicographically (chapter 3.2: bytes for buffers and ASCII
;; strings, code points for UTF-8 strings). (max a b c) folds the binary helper
;; over (list b c) starting from a.
(define-private (std-max-pair (left T) (right T))
  (if (> left right) left right))

(define-private (max (left T) (right T...))
  (fold-left std-max-pair left right))

(define-private (std-min-pair (left T) (right T))
  (if (< left right) left right))

(define-private (min (left T) (right T...))
  (fold-left std-min-pair left right))

;; Aborts on the most negative int, as `(- 0 x)` overflows there.
(define-private (abs (value int))
  (if (< value 0) (- 0 value) value))

;; Exponent bit positions, most significant first.
(define-constant std-exponent-bits
  (list u31 u30 u29 u28 u27 u26 u25 u24 u23 u22 u21 u20 u19 u18 u17 u16
        u15 u14 u13 u12 u11 u10 u9 u8 u7 u6 u5 u4 u3 u2 u1 u0))

;; Square-and-multiply, most significant bit first. Every intermediate value
;; is `base` raised to a prefix of the exponent's bits, so no step overflows
;; unless the final result does. `T` is int or uint.
(define-private (std-pow-step (acc { base: T, exponent: uint, result: T })
                              (shift uint))
  (let ((squared (* (get result acc) (get result acc))))
    (merge acc
      { result: (if (is-eq (bit-and (bit-shift-right (get exponent acc) shift) u1) u1)
                  (* squared (get base acc))
                  squared) })))

;; Aborts if the exponent is negative or does not fit in 32 bits, or on
;; overflow.
(define-private (pow (base int) (exponent int))
  (if (or (< exponent 0) (> exponent 4294967295))
    (panic)
    (get result (fold-left std-pow-step
                  { base: base, exponent: (to-uint exponent), result: 1 }
                  std-exponent-bits))))

(define-private (pow (base uint) (exponent uint))
  (if (> exponent u4294967295)
    (panic)
    (get result (fold-left std-pow-step
                  { base: base, exponent: exponent, result: u1 }
                  std-exponent-bits))))

;; Even bit positions from 126 down to 0: the powers of four tried by the
;; bitwise square root.
(define-constant std-sqrti-shifts
  (list u126 u124 u122 u120 u118 u116 u114 u112 u110 u108 u106 u104 u102
        u100 u98 u96 u94 u92 u90 u88 u86 u84 u82 u80 u78 u76 u74 u72 u70
        u68 u66 u64 u62 u60 u58 u56 u54 u52 u50 u48 u46 u44 u42 u40 u38
        u36 u34 u32 u30 u28 u26 u24 u22 u20 u18 u16 u14 u12 u10 u8 u6 u4
        u2 u0))
```

```
(define-private (std-sqrti-step (acc { remainder: uint, root: uint })
                                (shift uint))
  (let ((bit (bit-shift-left u1 shift))
        (candidate (+ (get root acc) bit)))
    (if (>= (get remainder acc) candidate)
      { remainder: (- (get remainder acc) candidate),
        root: (+ (bit-shift-right (get root acc) u1) bit) }
      { remainder: (get remainder acc),
        root: (bit-shift-right (get root acc) u1) })))

(define-private (std-sqrti-uint (value uint))
  (get root (fold-left std-sqrti-step { remainder: value, root: u0 }
             std-sqrti-shifts)))

(define-private (sqrti (value uint))
  (std-sqrti-uint value))

;; Aborts on a negative argument.
(define-private (sqrti (value int))
  (if (< value 0)
    (panic)
    (to-int (std-sqrti-uint (to-uint value)))))

;; Binary search for the highest set bit.
(define-private (std-log2-step (acc { rest: uint, result: uint }) (shift uint))
  (let ((shifted (bit-shift-right (get rest acc) shift)))
    (if (is-eq shifted u0)
      acc
      { rest: shifted, result: (+ (get result acc) shift) })))

;; Aborts on zero.
(define-private (std-log2-uint (value uint))
  (if (is-eq value u0)
    (panic)
    (get result (fold-left std-log2-step { rest: value, result: u0 }
                 (list u64 u32 u16 u8 u4 u2 u1)))))

(define-private (log2 (value uint))
  (std-log2-uint value))

;; Aborts on an argument that is not positive.
(define-private (log2 (value int))
  (if (<= value 0)
    (panic)
    (to-int (std-log2-uint (to-uint value)))))

;; ---------------------------------------------------------------------------
;; Sequences
;; ---------------------------------------------------------------------------

;; (list N T) -> (list (+ N 1) T)
(define-private (append (items (list N T)) (item T))
  (concat items (list item)))

(define-private (std-index-of-step (acc { found: (optional uint), index: uint, target: T })
                                   (element T))
  (match (get found acc)
    ((some ignored) acc)
    (none (if (is-eq element (get target acc))
            (merge acc { found: (some (get index acc)) })
            (merge acc { index: (+ (get index acc) u1) })))))
```

```
;; `S` is instantiated at a sequence type and `E` at its element type.
(define-private (index-of? (items S) (item E))
  (get found (fold-left std-index-of-step
               { found: none, index: u0, target: item }
               items)))

;; One definition per sequence kind: a list takes `(list item)` where the
;; other kinds take the one-element sequence itself.
(define-private (replace-at? (items (list N T)) (index uint) (item T))
  (if (>= index (len items))
    none
    (as-max-len? (concat (concat (unwrap-panic (slice? items u0 index)) (list item))
                         (unwrap-panic (slice? items (+ index u1) (len items))))
                 N)))

(define-private (replace-at? (items (buff N)) (index uint) (item (buff 1)))
  (if (>= index (len items))
    none
    (as-max-len? (concat (concat (unwrap-panic (slice? items u0 index)) item)
                         (unwrap-panic (slice? items (+ index u1) (len items))))
                 N)))

(define-private (replace-at? (items (string-ascii N)) (index uint) (item (string-ascii 1)))
  (if (>= index (len items))
    none
    (as-max-len? (concat (concat (unwrap-panic (slice? items u0 index)) item)
                         (unwrap-panic (slice? items (+ index u1) (len items))))
                 N)))

(define-private (replace-at? (items (string-utf8 N)) (index uint) (item (string-utf8 1)))
  (if (>= index (len items))
    none
    (as-max-len? (concat (concat (unwrap-panic (slice? items u0 index)) item)
                         (unwrap-panic (slice? items (+ index u1) (len items))))
                 N)))

;; ---------------------------------------------------------------------------
;; Conversions
;; ---------------------------------------------------------------------------

;; The core has one conversion per pair of representations (`to-int`,
;; `to-uint`, `buff-to-uint-be`, `uint-to-buff-be`). Everything here is
;; derived from those.

(define-private (std-prepend-byte (acc (buff 16)) (byte (buff 1)))
  (unwrap-panic (as-max-len? (concat byte acc) u16)))

(define-private (std-reverse-16 (bytes (buff 16)))
  (fold-left std-prepend-byte 0x bytes))

(define-private (buff-to-uint-le (bytes (buff 16)))
  (buff-to-uint-be (std-reverse-16 bytes)))

;; Two's complement, sign-extended from the buffer's actual length, so 0xff
;; is -1 (legacy zero-padded a short buffer and gave 255; chapter 3.4). The
;; negative branch adds -2^(8L-1), computed without overflowing when L is 16.
(define-private (buff-to-int-be (bytes (buff 16)))
  (let ((magnitude (buff-to-uint-be bytes))
        (width (len bytes)))
    (if (is-eq width u0)
      0
      (let ((half (bit-shift-left u1 (- (* width u8) u1))))
```

```
        (if (< magnitude half)
          (to-int magnitude)
          (+ (to-int (- magnitude half))
             (- (- 0 (to-int (- half u1))) 1)))))))

(define-private (buff-to-int-le (bytes (buff 16)))
  (buff-to-int-be (std-reverse-16 bytes)))

;; |value| as a uint, including for the most negative int.
(define-private (std-magnitude (value int))
  (if (< value 0)
    (+ (to-uint (- -1 value)) u1)
    (to-uint value)))

;; Per-kind digit tables, selected by the text type.
(define-private (std-prepend-digit (digit uint) (text (string-ascii 40)))
  (unwrap-panic (as-max-len? (concat (unwrap-panic (element-at? "0123456789" digit)) text) u40)))

(define-private (std-prepend-digit (digit uint) (text (string-utf8 40)))
  (unwrap-panic (as-max-len? (concat (unwrap-panic (element-at? u"0123456789" digit)) text) u40)))

;; u340282366920938463463374607431768211455 has 39 digits.
(define-constant std-decimal-iterations
  (list true true true true true true true true true true true true true true
        true true true true true true true true true true true true true true
        true true true true true true true true true true true true true true))

;; Emits one digit per step until the rest is zero; the first step always
;; emits, so zero renders as "0".
(define-private (std-decimal-step (acc { rest: uint, text: X }) (ignored bool))
  (if (and (is-eq (get rest acc) u0) (> (len (get text acc)) u0))
    acc
    { rest: (/ (get rest acc) u10),
      text: (std-prepend-digit (mod (get rest acc) u10) (get text acc)) }))

;; `X` is instantiated at the type of the empty string of the result kind,
;; `(string-ascii 0)` or `(string-utf8 0)`, and the fold widens it to 40.
(define-private (std-render-decimal (value uint) (empty X))
  (get text (fold-left std-decimal-step { rest: value, text: empty }
              std-decimal-iterations)))

(define-private (std-render-signed (value int) (empty X) (minus Y))
  (if (< value 0)
    (unwrap-panic (as-max-len? (concat minus (std-render-decimal (std-magnitude value) empty)) u40))
    (std-render-decimal (to-uint value) empty)))

(define-private (int-to-ascii (value uint))
  (std-render-decimal value ""))

(define-private (int-to-ascii (value int))
  (std-render-signed value "" "-"))

(define-private (int-to-utf8 (value uint))
  (std-render-decimal value u""))

(define-private (int-to-utf8 (value int))
  (std-render-signed value u"" u"-"))

(define-private (std-digit-value (char (string-ascii 1)))
  (index-of? "0123456789" char))

(define-private (std-digit-value (char (string-utf8 1)))
```

```
  (index-of? u"0123456789" char))

(define-private (std-is-minus (char (string-ascii 1)))
  (is-eq char "-"))

(define-private (std-is-minus (char (string-utf8 1)))
  (is-eq char u"-"))

;; total * 10 + digit, or none if that exceeds the largest uint.
(define-private (std-push-digit (total uint) (digit uint))
  (if (or (> total u34028236692093846346337460743176821145)
          (and (is-eq total u34028236692093846346337460743176821145) (> digit u5)))
    none
    (some (+ (* total u10) digit))))

(define-private (std-parse-digit (acc (optional uint)) (char C))
  (match acc
    ((some total)
      (match (std-digit-value char)
        ((some digit) (std-push-digit total digit))
        (none none)))
    (none none)))

;; `S` is instantiated at string-ascii or string-utf8.
(define-private (string-to-uint? (text S))
  (if (is-eq (len text) u0)
    none
    (fold-left std-parse-digit (some u0) text)))

;; -magnitude, or none below the most negative int.
(define-private (std-negate-magnitude (magnitude uint))
  (if (is-eq magnitude u0)
    (some 0)
    (if (> magnitude u170141183460469231731687303715884105728)
      none
      (some (- (- 0 (to-int (- magnitude u1))) 1)))))

;; `S` is instantiated at string-ascii or string-utf8.
(define-private (string-to-int? (text S))
  (match (element-at? text u0)
    ((some first)
      (if (std-is-minus first)
        (match (string-to-uint? (unwrap-panic (slice? text u1 (len text))))
          ((some magnitude) (std-negate-magnitude magnitude))
          (none none))
        (match (string-to-uint? text)
          ((some magnitude)
            (if (> magnitude u170141183460469231731687303715884105727)
              none
              (some (to-int magnitude))))
          (none none))))
    (none none)))

;; `to-ascii?` renders the six kinds legacy accepts, with legacy's text and
;; result bounds, except that a principal's bound is 82, not 170, because
;; contract names are at most 40 characters. Only the string-utf8 overload
;; can fail. Principals go through the `principal-to-ascii` primitive
;; until the c32check encoder is written here (chapter 3.12).

(define-private (to-ascii? (value int))
  (ok (int-to-ascii value)))
```

```
;; "u" and at most 39 digits.
(define-private (to-ascii? (value uint))
  (ok (unwrap-panic (as-max-len? (concat "u" (int-to-ascii value)) u40))))

(define-private (to-ascii? (value bool))
  (ok (if value "true" "false")))

(define-private (to-ascii? (value principal))
  (ok (principal-to-ascii value)))

;; Two lowercase hex digits.
(define-private (std-hex-byte (byte (buff 1)))
  (let ((number (buff-to-uint-be byte)))
    (concat (unwrap-panic (element-at? "0123456789abcdef" (/ number u16)))
            (unwrap-panic (element-at? "0123456789abcdef" (mod number u16))))))

;; "0x" and two digits per byte: the spread gives the bound 2 + N * 2.
(define-private (to-ascii? (value (buff N)))
  (ok (concat "0x" (map std-hex-byte value)...)))

;; The string-ascii character set (chapter 2.2), in the same order in both
;; kinds: 0x20 through 0x7E, then tab, line feed, carriage return.
(define-constant std-ascii-chars
  " !\"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~\t\n\r")
(define-constant std-ascii-chars-utf8
  u" !\"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~\t\n\r")

(define-private (std-char-to-ascii (char (string-utf8 1)))
  (match (index-of? std-ascii-chars-utf8 char)
    ((some index) (element-at? std-ascii-chars index))
    (none none)))

(define-private (std-char-or-empty (char (optional (string-ascii 1))))
  (default-to "" char))

;; (err u1) if any character has no string-ascii form. The spread gives the
;; bound N * 1.
(define-private (to-ascii? (value (string-utf8 N)))
  (let ((chars (map std-char-to-ascii value)))
    (if (is-eq (len (filter is-some chars)) (len chars))
      (ok (concat "" (map std-char-or-empty chars)...))
      (err u1))))

;; ------------------------------------------------------------------------
;; Principals
;; ------------------------------------------------------------------------

;; Version bytes for single-signature and multisig standard principals.
(define-constant std-mainnet-p2pkh-version 0x16)
(define-constant std-mainnet-p2sh-version 0x14)
(define-constant std-testnet-p2pkh-version 0x1a)
(define-constant std-testnet-p2sh-version 0x15)

(define-private (std-current-network-version? (version (buff 1)))
  (if (get is-mainnet (get-current-context))
    (or (is-eq version std-mainnet-p2pkh-version)
        (is-eq version std-mainnet-p2sh-version))
    (or (is-eq version std-testnet-p2pkh-version)
        (is-eq version std-testnet-p2sh-version))))
```

```
;; A principal's encoding (chain ABI 2.2) is a prefix byte, the version byte,
;; the 20 hash bytes, and for a contract principal the name's u8 length and the
;; name. A name is at most 40 bytes, so its `string-ascii` encoding is 0x0d,
;; three zero bytes, and then those same length and name bytes.
(define-private (principal-destruct? (who principal))
  (let ((bytes (to-consensus-buff who))
        (version (unwrap-panic (as-max-len? (unwrap-panic (slice? bytes u1 u2)) u1)))
        (parts { hash-bytes: (unwrap-panic (as-max-len? (unwrap-panic (slice? bytes u2 u22)) u20)),
                 name: (if (is-eq (len bytes) u22)
                         none
                         (from-consensus-buff? (string-ascii 40)
                           (concat 0x0d000000 (unwrap-panic (slice? bytes u22 (len bytes))))))
                 version: version }))
    (if (std-current-network-version? version) (ok parts) (err parts))))

;; Builds the encoding and decodes it, so the chain ABI's decoding rules decide
;; which names are valid. Error codes: u0 for a version byte that is not the
;; current network's, with the principal; u1 for a version byte of 32 or more,
;; or a version or hash buffer shorter than its bound; u2 for an invalid name.
(define-private (principal-construct? (version (buff 1)) (hash-bytes (buff 20))
                                      (name (optional (string-ascii 40))))
  (if (not (and (is-eq (len version) u1)
                (is-eq (len hash-bytes) u20)
                (< (buff-to-uint-be version) u32)))
    (err { error-code: u1, principal: none })
    (match (from-consensus-buff? principal
             (match name
               ((some contract-name)
                 (let ((name-bytes (to-consensus-buff contract-name)))
                   (concat 0x06 version hash-bytes
                           (unwrap-panic (slice? name-bytes u4 (len name-bytes))))))
               (none (concat 0x05 version hash-bytes))))
      ((some who) (if (std-current-network-version? version)
                    (ok who)
                    (err { error-code: u0, principal: (some who) })))
      (none (err { error-code: u2, principal: none })))))

;; True if the principal's version byte is for the current network, whether it
;; is a standard or a contract principal. This is what legacy `is-standard`
;; tested, under a name that said otherwise.
(define-private (is-current-network (who principal))
  (is-ok (principal-destruct? who)))

;; True if the principal is a standard principal, not a contract principal,
;; on any network.
(define-private (is-standard (who principal))
  (match (principal-destruct? who)
    ((ok parts) (is-none (get name parts)))
    ((err parts) (is-none (get name parts)))))

;; Derive the standard principal for a compressed public key on the current
;; network. Returns (err u1) if the key is not 33 bytes.
(define-private (principal-of? (public-key (buff 33)))
  (if (not (is-eq (len public-key) u33))
    (err u1)
    (match (principal-construct?
             (if (get is-mainnet (get-current-context))
               std-mainnet-p2pkh-version
               std-testnet-p2pkh-version)
             (hash160 public-key)
             none)
      ((ok who) (ok who))
```

```
      ((err ignored) (err u1)))))

;; ---------------------------------------------------------------------------
;; Cryptography
;; ---------------------------------------------------------------------------

;; Bitcoin's HASH160.
(define-private (hash160 (bytes (buff N)))
  (ripemd160 (sha256 bytes)))

(define-private (std-double-sha256 (bytes (buff 64)))
  (sha256 (sha256 bytes)))

;; Depth of a Bitcoin Merkle tree over `count` leaves (count >= 1).
(define-private (std-merkle-depth (count uint))
  (if (<= count u1)
    u0
    (+ (log2 (- count u1)) u1)))

;; One level up the tree. `width` is the number of nodes in the current row.
;; A left node that is last in its row is paired with itself, so its sibling
;; must equal it.
(define-private (std-merkle-step (acc { hash: (buff 32), index: uint, valid: bool, width: uint })
                                 (sibling (buff 32)))
  (let ((current (get hash acc))
        (position (get index acc))
        (right-child (is-eq (mod position u2) u1)))
    { hash: (if right-child
              (std-double-sha256 (concat sibling current))
              (std-double-sha256 (concat current sibling))),
      index: (/ position u2),
      valid: (and (get valid acc)
                  (or right-child
                      (not (is-eq (+ position u1) (get width acc)))
                      (is-eq sibling current))),
      width: (+ (/ (get width acc) u2) (mod (get width acc) u2)) }))

;; Bitcoin-style inclusion proof; see chapter 4.2 for the full contract. `false` for
;; an index not below the count, a path length other than the tree depth, a
;; failed duplication check, or a root mismatch.
(define-private (verify-merkle-proof (leaf (buff 32)) (root (buff 32))
                                     (index uint) (count uint)
                                     (proof (list 24 (buff 32))))
  (if (or (>= index count) (not (is-eq (len proof) (std-merkle-depth count))))
    false
    (let ((final (fold-left std-merkle-step
                  { hash: leaf, index: index, valid: true, width: count }
                  proof)))
      (and (get valid final) (is-eq (get hash final) root)))))

;; ---------------------------------------------------------------------------
;; Storage
;; ---------------------------------------------------------------------------

;; Insert only if absent. Returns the existing value, and writes nothing, if
;; the key already has one; `none` if the value was inserted. Private, so
;; `key` and `value` are both evaluated before the call.
(define-private (map-insert? (target (map K V)) (key K) (value V))
  (match (map-get? target key)
    ((some existing) (some existing))
    (none (begin (map-set target key value) none))))
```