TECHNICAL REPORT

Canonical Arithmetization of JSON Documents

with Authenticated Stores and Zero-Knowledge Path Queries

Tomoya Nagasawa

November 2026

ABSTRACT

Arithmetic circuits see only vectors of prime-field elements, while most application data is JSON: nested, schemaless and typed at run time. We define a canonical encoding of JSON values into elements of the BN254 scalar field $\mathbb{F}_p$ and prove that it is injective on a precisely stated domain. A value is flattened into (path, leaf) rows, where empty arrays and empty objects are leaves of their own. Objects are canonical finite maps, so the rows come out sorted in path order (keys in shortlex order). The rows are written as type-tagged integers. Each integer becomes length-prefixed decimal tokens, and runs of one-digit tokens are compressed. The resulting digit stream is cut into blocks of 75 digits, each stored behind a guard digit. The main theorem states that the encoding is injective, and we prove it in general by giving every stage a left inverse. Decoding the encoding of any value in the domain, with any amount of zero padding, returns the value. Every element e satisfies $1 \le e < 10^{76} < p$, and a digit decomposition over $\mathbb{F}_p$ of up to 76 digits is unique. On this encoding we define path queries as relations over a commitment, both for "the value at path π is ℓ" and for "π does not exist". We prove that a walk over the token grammar decides both exactly on committed canonical encodings. We state what a circuit must constrain for soundness, and give an analytic constraint estimate, labelled as such. Finally we specify a two-level sparse Merkle store whose leaf indices come from a guarded radix map, which we prove injective with tight capacity bounds, together with membership, non-membership and batched-transition relations whose security reduces to stated standard assumptions. All encoding results except the nesting induction of Corollary 1 are machine-checked in Lean 4 (core library only). The full development is reproduced in the appendix.

Keywords: JSON, arithmetization, canonical encoding, prefix-free codes, injectivity, zero-knowledge proofs, sparse Merkle trees, authenticated data structures, Lean 4

1 Introduction

A verifier that can afford to store very little, such as a light client or a program with metered storage, can hold one short commitment to a database and accept statements about it only together with a succinct proof. Two kinds of statement matter. *Queries* say, for example, "in the committed database, document `Bob` of collection 0 has `age` equal to 10". *Transitions* say "the database moved from root R to root R' by a batch of valid writes". Succinct arguments over Merkle commitments handle both in principle. The obstacle is the data model. Application data is mostly JSON [17]: nested, schemaless, variable-length and typed at run time. An arithmetic circuit has no strings, maps or data-dependent control flow. It sees a fixed number of elements of a prime field, wired into a fixed graph of additions and multiplications.

Two workarounds are common, and both have a cost. A per-schema circuit, with one field element per known field, must be rebuilt and set up again whenever the schema changes. Parsing raw JSON text inside the circuit, as systems that prove claims about TLS transcripts or identity tokens do [18, 19], is general, but the circuit then pays for a byte-level lexer. This report develops a middle path. The document is *arithmetized once*: a canonical encoding turns it into field elements whose digit grammar a fixed circuit can walk, and it is committed in that form. A single circuit, fixed only by size bounds, can then address any path of any committed document within those bounds.

For a commitment to identify a document, the encoding must be canonical, so that equal documents

give equal vectors, and injective, so that different documents give different vectors. The encoding also has to fit the field without wrapping around. These are mathematical properties, and this report proves them. Our contributions are:

- **A canonical encoding** of JSON values into $\mathbb{F}_p$, on an explicit domain. The domain consists of decimal numbers in shortest form, objects as finite maps with unique keys, and empty containers as leaves (§3–4).
- **The main theorem** (§5): the encoding is injective. The proof composes per-stage left inverses: flattening, integer coding, tokenization, run compression and packing. It is machine-checked in general, not by testing cases.
- **Field-fit results** (§6): every element lies in $[1, 10^{76}) \subset [0, p)$, the payload capacity is exact, and digit decomposition over $\mathbb{F}_p$ is unique.
- **Query relations** (§7) for path–value statements and for non-existence. We prove that a walk over the token grammar decides both exactly. We show by a mechanized example that the walk must be anchored at the first digit, list what a sound circuit must constrain, and give an analytic constraint estimate.
- **A two-level authenticated store** (§8) whose leaf index map is injective with a tight bound, with membership, non-membership and batched-transition relations. Their security reduces to stated assumptions: collision resistance, infeasibility of hashing a document to the empty leaf value 0, and knowledge soundness of the argument system.

Table 1 separates what is machine-checked from what rests on paper proofs or on cryptographic assumptions.

result	status
injectivity, round trip (Thms 1–2)	Lean
per-stage left inverses (Lemmas 1–5)	Lean
finite-map canonicity, one level (Prop. 2)	Lean
finite-map canonicity, nested (Cor. 1)	Paper, from Lean parts
field bound, digit uniqueness (Thm 3, Prop. 4)	Lean
walk decides queries (Thm 4)	Lean
anchoring necessity (Prop. 5)	Lean (example)
id map injective, bounds, capacity (Thm 6)	Lean
query and transition soundness (Thms 5, 7)	Paper, under assumptions
constraint counts (§7.4, §8.5)	analytic estimate
float → shortest decimal (§3.1)	preprocessing, outside Lean

Table 1. Status of each result. "Lean" means the statement is proved in the appendix in general. "Paper" means the statement is proved in the text. "Assumption" means a standard cryptographic assumption, stated and not proved.

2 Preliminaries

The field. Pairing-based arguments on the BN254 curve work over the scalar field $\mathbb{F}_p$, where

p = 21888242871839275222246405745257275088548364400416034343698204186575808495617.

p has 77 decimal digits. The following bounds are proved by evaluation in Lean (`p_bounds`, `p_bits`):

$$10^{76} < p < 10^{77},\ 2^{253} < p < 2^{254}.$$

Numerically, $p/10^{76} \approx 2.189$ and $\log_2 p \approx 253.597$. Every integer below 10^{76} is therefore its own representative in $\mathbb{F}_p$.

Constraint systems and arguments. A rank-1 constraint system (R1CS) is a set of equations $\langle \mathbf{a}_i, \mathbf{w} \rangle \cdot \langle \mathbf{b}_i, \mathbf{w} \rangle = \langle \mathbf{c}_i, \mathbf{w} \rangle$ over a witness vector $\mathbf{w}$ that contains the public inputs. QAP-based arguments [8, 9], in particular Groth16 [10], prove knowledge of a satisfying $\mathbf{w}$ with a constant-size proof. Universal-setup systems such as PLONK [11] serve the same role. Throughout, a circuit *is sound for* a relation $\mathscr{R}$ if every satisfying assignment, restricted to the statement and the extracted witness, lies in $\mathscr{R}$. A value that the prover computes but no constraint mentions

can be chosen freely by a dishonest prover. This distinction drives §7.3.

Arithmetization-friendly hashing. Poseidon [12] is a sponge over $\mathbb{F}_p$ whose permutation uses x^5 S-boxes. With width t, R_F full rounds and R_P partial rounds, an R1CS circuit for it needs 3 constraints per S-box, hence $3(t{\cdot}R_F + R_P)$ constraints per permutation. We write P_k for Poseidon on k inputs (width $t = k + 1$).

Sparse Merkle trees. A sparse Merkle tree (SMT) of depth d has 2^d leaf slots, indexed by d-bit integers, almost all of which hold a default value [3, 4]. An empty subtree of height j hashes to a constant Z_j, so only occupied paths are stored. An opening of slot k is the list of d siblings on the path that the bits of k select.

Notation. Lists are written $[a, b, \ldots]$, $\|$ is concatenation, $|s|$ is length, and 0^z is a list of z zeros. For digit lists, $[\![s]\!] = \Sigma_i\, s_i{\cdot}10^{|s|-1-i}$ is the Horner value (`ofD`). *Shortlex* order on sequences compares length first and then compares lexicographically.

3 JSON values and the domain

3.1 Values

Definition 1 (values). The set $\mathscr{J}$ of JSON values is generated by: null; true and false; numbers num(σ, d, m) with $\sigma \in \{+, -\}$ and $d, m \in \mathbb{N}$, denoting $(-1)^{[\sigma=-]}{\cdot}m/10^d$; strings, which are finite sequences of natural numbers (code units); arrays $[v_0, \ldots, v_{n-1}]$; and objects, which are finite lists of (key, value) pairs whose keys are strings.

The theory never interprets code units, so any fixed convention works, for example Unicode scalar values or UTF-16 code units. It only has to be applied consistently.

Numbers. The domain of numbers is the decimal rationals. A JSON number literal denotes one exactly. A binary floating-point value enters through a fixed function: the shortest decimal string that rounds back to it. Because that string rounds back to the same binary value, the map is injective on finite binary values. It is a preprocessing step, outside the encoding and outside the mechanization.

Definition 2 (canonical numbers). A triple (σ, d, m) is canonical if (i) $d > 0$ implies $10 \nmid m$, and (ii) $m = 0$ implies $\sigma = +$ and $d = 0$.

Proposition 1 (numbers). Two canonical triples that denote the same rational number are equal. (`num_canon_unique`)

Condition (ii) identifies -0 with 0. Every decimal rational has exactly one canonical triple, which is obtained by stripping trailing zeros of m while $d > 0$.

3.2 The domain

Definition 3 (domain $\mathscr{W}$). A value is *well formed* (`WF`) if every number in it is canonical and the keys of every object in it are *strictly increasing* in shortlex order. Strict increase rules out duplicate keys.

JSON objects with unique keys are finite maps, and the order in which keys were written carries no meaning. Definition 3 picks one representative of each finite map. The next proposition shows that the choice is well defined. Here isort is insertion sort by key.

Proposition 2 (finite maps). (a) Shortlex order is a strict total order on strings (`slt_irrefl`, `slt_trans`, `slt_total`). (b) Two association lists that are both sorted by strictly increasing key and have the same entries are equal (`sorted_unique`). (c) If l_1 and l_2 both have pairwise distinct keys and the same set of entries, then isort(l_1) = isort(l_2), and the result is strictly sorted (`canonical_rep`, `isort_sorted`).

Let canon : $\mathscr{J} \to \mathscr{J}$ replace every number by its canonical triple and every object by isort of its canonicalized entries, recursively. On values whose objects have distinct keys, canon(x) $\in \mathscr{W}$.

Two JSON values are *equal as JSON*, written $x \equiv y$, if they agree as trees in which arrays are sequences, objects are finite maps and numbers are rationals. Duplicate keys are outside the domain. This exclusion is necessary: `{"a":{"x":1},"a":{"y":2}}` has exactly the rows of `{"a":{"x":1,"y":2}}`, so no row-based encoding separates the two.

3.3 Paths, rows and flattening

Definition 4 (rows). A path component is key(k) for a string k, or idx(i) for $i \in \mathbb{N}$. A path is a list of components. A leaf is a primitive (null, a boolean, a number or a string), or one of two markers: $\varnothing_{\text{arr}}$ for the empty array and $\varnothing_{\text{obj}}$ for the empty object. A row

is a pair (π, ℓ) of a path and a leaf. flatten : $\mathscr{J} \to \text{rows}^\star$ is defined by:

$$\text{flatten}(v) = [([], v)] \text{ for a primitive } v,$$
$$\text{flatten}([]) = [([], \varnothing_{\text{arr}})], \text{flatten}(\{\}) = [([], \varnothing_{\text{obj}})],$$
$$\text{flatten}([v_0,\ldots,v_{n-1}]) = \|_i \, \text{idx}(i)\cdot\text{flatten}(v_i),$$
$$\text{flatten}(\{(k_1,v_1),\ldots\}) = \|_j \, \text{key}(k_j)\cdot\text{flatten}(v_j),$$

where $c\cdot R$ prefixes component c to the path of every row in R.

The markers are needed. If empty containers contributed no row, `{"a":1,"b":[]}`, `{"a":1,"b":{}}` and `{"a":1}` would all have the single row (a, 1). With markers, the leaf set determines the tree, because every node lies on some row's path.

Definition 5 (path order). Components are ordered by $\text{idx}(i) \prec \text{idx}(j)$ iff $i < j$, $\text{idx}(\cdot) \prec \text{key}(\cdot)$, and $\text{key}(a) \prec \text{key}(b)$ iff a precedes b in shortlex order. Paths are compared component by component, and a proper prefix comes first.

Proposition 3 (order). (a) For components a, b: $a \prec b$ iff the integer code of a (§4.1) precedes that of b lexicographically, with a proper prefix first (`cLt_eq_code`). (b) For every $v \in \mathscr{W}$, the rows of flatten(v) are strictly increasing in path order (`flatten_sorted`).

By (b), on the domain the row list is the unique $\prec$-sorted enumeration of the row set. A separate sorting stage would be the identity. Arrays keep index order, and object entries appear in shortlex key order. By (a), comparing integer codes compares components, so an order check needs no decoding.

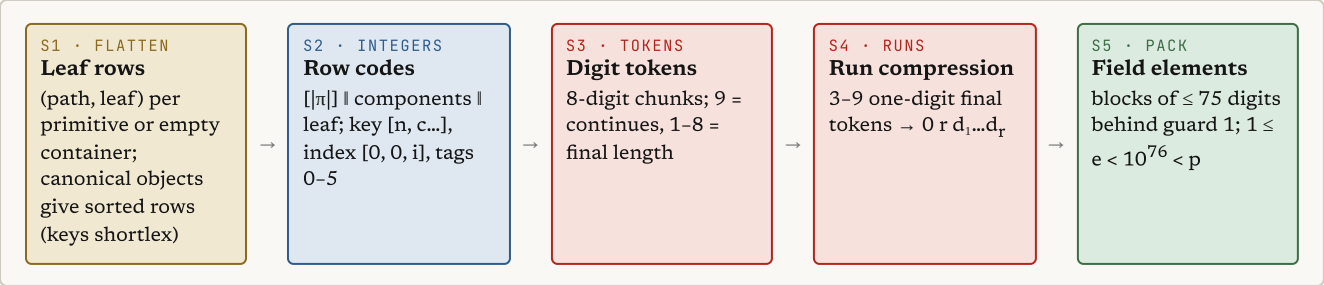


Figure 1. The five stages of the encoding. Every stage has a left inverse on its image (Lemmas 1–5). S1 needs the domain $\mathscr{W}$. The other stages are inverted unconditionally.

4 The pipeline

4.1 Integer coding

Definition 6 (integer codes). Components are coded as follows. A key with code units $c_1 \ldots c_n$, $n \geq 1$, becomes $[n, c_1, \ldots, c_n]$. The empty key becomes [0, 1], and idx(i) becomes [0, 0, i]. Leaves are coded by Table 2. A row (π, ℓ) is coded as $[|\pi|] \,\|\, \text{code}(\pi_1) \,\|\, \cdots \,\|\, \text{code}(\pi_{|\pi|}) \,\|\, \text{code}(\ell)$. ints($R$) concatenates the row codes of a row list.

leaf	code	example
null	[0]	null → [0]
boolean	[1, b]	false → [1, 0]
number	[2, s, d, m]	3.14 → [2, 1, 2, 314]
string	[3, n, $c_1 \ldots c_n$]	"four" → [3, 4, 102, 111, 117, 114]
empty array $\varnothing_{\text{arr}}$	[4]	[] → [4]
empty object $\varnothing_{\text{obj}}$	[5]	{} → [5]

Table 2. Leaf codes. For a number, $s = 0$ if $\sigma = -$ and $s = 1$ otherwise. Each tag determines how many integers follow it, so the leaf code is prefix-free.

The row code is prefix-free. Its first integer gives the number of components. A component code starting with $n > 0$ is followed by n units, and one starting with 0 is followed by 1 (the empty key) or by 0 and an index.

The leaf tag then fixes the rest. For `{"a":1}` the integers are [1, 1, 97, 2, 1, 0, 1].

4.2 Digit tokens

> **Definition 7 (tokens).** Let $\delta(n)$ be the decimal digits of n, most significant first, with $\delta(0) = [0]$. Cut $\delta(n)$ from the left into chunks $c_1, \ldots, c_k$ with $|c_i| = 8$ for $i < k$ and $1 \le |c_k| \le 8$. Each c_i with $i < k$ becomes a *continuation* token, written $9 \parallel c_i$. The last chunk becomes a *final* token, written $|c_k| \parallel c_k$. toks($n_1, \ldots$) concatenates the tokens of each integer.

For example, 123 → `3123`; 1234567890 → `912345678 290`; 10^8 → `910000000 10`; and 123456789 → `912345678 19`. A leading digit is never 0, which frees 0 as an escape for runs.

4.3 Run compression and the one-digit-tail rule

Most integers in the stream are single digits: tags, signs, counts and indices. Each costs two digits as a token.

> **Definition 8 (units and compression).** A *single* is any final token with exactly one digit, $1d$. This includes the one-digit tail of an integer with $8j + 1$ digits (the *one-digit-tail rule*). A *unit* is a plain token or a run $0 \parallel r \parallel d_1 \ldots d_r$, which stands for the r singles $1d_1, \ldots, 1d_r$. compress scans the tokens left to right and collects consecutive singles. A run is closed when it reaches 9 singles. When a non-single token or the end of the stream arrives, the open run is emitted as a run if it has at least 3 singles, and as plain singles otherwise. stream(ts) is the digit string of compress(ts).

A run of $r \ge 3$ singles costs $r + 2$ digits instead of $2r$. The tail rule means that a run may begin while an integer is still incomplete. The decoder expands every run back into singles before it groups tokens into integers, so the first single of the run completes the pending integer (Fig. 2). Lemma 4 proves that the expanded token list is recovered exactly, whatever integers the singles belong to.

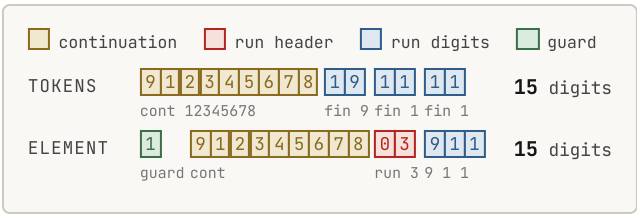


Figure 2. The integers [123456789, 1, 1]. The tail `19` of 123456789 is a single and opens a run of three. Decoding expands the run, and its first digit completes 12345678‖9. The stream is `9 12345678 03 911`, and the element is 191234567803911.

4.4 Packing

> **Definition 9 (packing).** Let $W = 75$. Cut a digit stream s into consecutive blocks $b_1, \ldots, b_k$ with $|b_j| = W$ for $j < k$ and $1 \le |b_k| \le W$. Each block becomes the element elem(b) = $10^{|b|} + [\![b]\!]$, whose decimal form is the guard digit 1 followed by b, leading zeros included. A vector of N elements is padded with zeros. The *encoding* is
>
> $$encode(v) = pack(stream(toks(ints(flatten(v))))).$$

The guard keeps leading zeros, so a block that starts with the run escape 0 survives being read as a number. It also makes every element non-zero, so zero elements are unambiguous padding. Tokens and runs may straddle a block boundary. Remark 1 shows that any other cut decodes the same way.

4.5 Worked examples

For `{"a":1}` the tokens are `11 11 297 12 11 10 11`. The first two singles stay plain. The last four become the run `04 2101`. The encoding is the single element 11111297042101 (Fig. 3, top). For `{"a":1,"b":[]}` the integers are [1, 1, 97, 2, 1, 0, 1, 1, 1, 98, 4], and the encoding is 111112970621011129814. It differs from that of `{"a":1}`, as Theorem 2 requires (`empty_marker_distinguishes`). Replacing [] by {} changes only the final tag: 111112970621011129815.

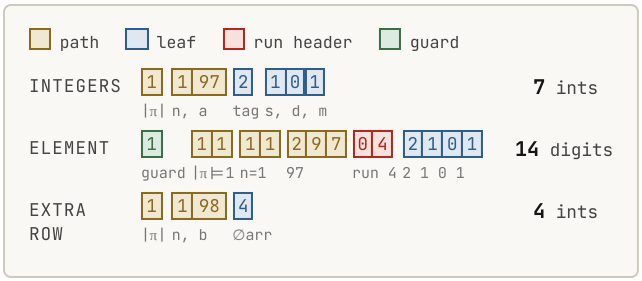


Figure 3. `{"a":1}` as integers and as its one element. The bottom row is the extra row that `"b":[]` contributes.

The 66-character running example

```
{"a":1,"c":false,"b":
{"e":null,"d":"four"},"f":3.14,"ghi":[5,6,7]}
```

is first canonicalized. Its keys are listed in shortlex order, a, b, c, f, ghi, and inside b the order is d, e. It has eight rows and 72 integers. As plain tokens these take 182 digits, and run compression reduces them to 155. Packing gives three elements of 76, 76 and 6 digits, all below p:

```
11111297062101212981131001314310231113117311412112981131010301
12990410113102
10321233141213310331043105090002105233103310431050900121062331
03103104310507 00
122107
```

The last row, ghi[2] with value 7, ends in seven singles (0, 0, 2 from the path and 2, 1, 0, 7 from the leaf), which form the run `07 0022107`. The second element ends with the run header and the first two run digits, and the third element holds the remaining five digits `22107`. Decoding concatenates the blocks before parsing, so the split has no effect. The integers, token counts and elements above were computed by evaluating the Lean definitions in the appendix.

5 Main theorem: injectivity via left inverses

Define decode : $\mathbb{N}^* \to \mathscr{J} \cup \{\bot\}$ as the composition of the stage inverses. unpack drops zero elements, strips each guard and concatenates the blocks. decodeTok parses units and expands runs. tokInts completes integers. parseRows parses row codes. unflatten rebuilds the tree.

Theorem 1 (round trip). For every $v \in \mathscr{W}$ and every $z \in \mathbb{N}$, decode(encode(v) ‖ 0^z) = v. (`decode_encode`)

Theorem 2 (injectivity). For $v, w \in \mathscr{W}$, encode(v) = encode(w) implies $v = w$. More generally, encode(v) ‖ 0^z = encode(w) ‖ $0^{z'}$ implies $v = w$ for all z, z', so the same holds after zero padding to any common length N. (`encode_injective`, `encode_injective_pad`)

Theorem 2 follows from Theorem 1, because a map with a left inverse is injective. Padding is harmless because decode ignores zero elements and encode never produces one. Theorem 1 is the composition of five round-trip lemmas, one per stage.

Lemma 1 (S1). For $v \in \mathscr{W}$, unflatten(flatten(v)) = v. (`unflatten_flatten`)

Lemma 2 (S2). For every row list R, parseRows(ints(R)) = R. (`parseRows_ints'`)

Lemma 3 (S3). For every integer list ns, tokInts(toks(ns)) = ns. (`tokInts_toks`)

Lemma 4 (S4). For every list ts of valid tokens (lengths as in Definition 7, digits below 10), decodeTok(stream(ts)) = ts. More generally, every list of valid units, whether its runs are maximal or split arbitrarily, decodes to its expansion. Also expand(compress(ts)) = ts. (`decodeTok_stream`, `decodeTok_units`, `expand_compress`)

Lemma 5 (S5). For every digit stream s and every z, unpack(pack(s) ‖ 0^z) = s. (`unpack_pack`)

Proof sketch. S5. Each element has a guard, so it is nonzero, and trailing zeros are exactly the padding. elem(b) has $|b| + 1$ decimal digits, so taking the last $|b|$ digits returns b, leading zeros included (`unElem_elem`). *S4.* A unit is identified by its first digit. 0 introduces a run whose length is the next digit, 9 a continuation of 8 digits, and 1–8 a final token of that length. So the digit grammar is prefix-free and parses in exactly one way (`parseU_digits`). Expanding runs reverses compression, because compress only regroups singles. *S3.* Continuation tokens accumulate digits until a final token completes the integer. Since the decoder works on expanded tokens, a single that came from a run completes a pending integer like any other final token. Then $[\![\delta(n)]\!] = n$. *S2.* The row code is prefix-free (§4.1), and each parser consumes exactly its code (`parseRow_code`). *S1.* unflatten groups consecutive rows by their first component and recurses. This recovers the children because a canonical object has pairwise distinct adjacent keys (so groups do not merge), arrays index from 0 in order, every child contributes at least one row (`flatten_ne_nil`, which is where the markers

enter), and a lone row with the empty path is a leaf. Composing the five lemmas gives Theorem 1. □

Corollary 1 (finite-map semantics). Let x, y be JSON values whose objects have pairwise distinct keys. Then encode(canon(x)) = encode(canon(y)) if and only if $x \equiv y$.

Proof. (⇐) By structural induction on $x \equiv y$. Numbers have the same canonical triple (Proposition 1), arrays agree elementwise, and the canonicalized entries of two equal objects form association lists with distinct keys and equal entry sets, so Proposition 2(c) gives equal sorted lists. Hence canon(x) = canon(y). (⇒) Both sides lie in $\mathscr{W}$, so Theorem 2 gives canon(x) = canon(y), and canon(x) $\equiv x$ because canon only permutes entries and rewrites numbers into equal ones. □ The one-level statement (c) is mechanized. The induction over nesting is the paper step.

Each domain condition is necessary. Without the markers, {a:1, b:[]} and {a:1} collide (§3.3). Duplicate keys collide (§3.2). Non-canonical triples break canonicity: (+, 1, 10) and (+, 0, 1) both denote 1 but have different codes. Without the guard, a block starting with 0 would lose its leading digits. The tail rule is consistent only because tokInts runs on expanded tokens. A grouping that read integers off unexpanded units would split a 9-digit integer followed by $r \geq 2$ singles into its first 8 digits and separate integers. Lemmas 3–4 give exact recovery (`one_digit_tail_roundtrip`).

6 Field packing bound and capacity

Theorem 3 (field fit). For every value v, each element e of encode(v) satisfies $1 \leq e < 10^{76} < p$. In general, a block b of at most 75 digits has $10^{|b|} \leq \text{elem}(b) < 10^{|b|+1} \leq 10^{76}$. (`encode_lt_p`, `elem_lt_p`, `elem_bounds`)

No element is ever reduced mod p, so the integer encoding and the field encoding coincide. Theorem 3 does not require $v \in \mathscr{W}$. A circuit also relies on the converse direction: when it decomposes a field element into digits, the digits must be determined.

Proposition 4 (unique decomposition in $\mathbb{F}_p$). Let a, b be digit lists of equal length at most 76. If $[\![a]\!] \equiv [\![b]\!] \pmod p$, then $a = b$. (`digits_mod_p_unique`)

Proof. Both values are below $10^{76} < p$ (`ofD_lt`), so they are equal as integers, and fixed-length decimal representations are unique (`ofD_inj`). □

Capacity. A stream of at most $75N$ digits packs into at most N elements (`encode_length`). With $N = 256$ the payload is 19,200 digits (`capacity_256_elements`). Each element carries 75 payload digits, that is $75 \cdot \log_2 10 \approx 249.145$ bits, and $2^{249} < 10^{75} < 2^{250}$ (`payload_bits`). That is 98.24% of $\log_2 p$. A 256-element vector carries about 63,781 payload bits. A 76-digit element fits a 256-bit machine word.

Remark 1 (any segmentation). Lemma 5 is a special case of a stronger statement. For *any* list of blocks, each of at most 75 digits, unpack returns their concatenation (`unpack_segments`). Together with `decodeTok_units`, this shows that a packer may choose its own cuts, for example keeping every token inside one element and splitting runs at element boundaries. The same decoder inverts it, provided the concatenated blocks form a valid unit list with the same expansion. Different packers give different but equally decodable vectors. A commitment must fix one packer, and we fix Definition 9.

7 Path queries

7.1 The relations

Fix N and a hash $H : \mathbb{F}_p^N \to \mathbb{F}_p$. For $v \in \mathscr{W}$ whose encoding fits, let $x_v = \text{encode}(v) \parallel 0^{N-|\text{encode}(v)|}$. The document commitment is $h = H(x_v)$. The value at a path, get(v, π), follows keys and indices from the root and is undefined if a step is missing.

Definition 10 (query relations). For a path π and a leaf ℓ,

$$\mathscr{R}_q = \{\, ((h, \pi, \ell); x) : H(x) = h,\ \exists v \in \mathscr{W}.\ x = x_v \wedge \text{get}(v, \pi) = \ell \,\},$$

$$\mathscr{R}_{\nexists} = \{\, ((h, \pi); x) : H(x) = h,\ \exists v \in \mathscr{W}.\ x = x_v \wedge \text{get}(v, \pi)\ \text{undefined} \,\}.$$

Here a leaf ℓ stands for the value it denotes, and $\varnothing_{\text{arr}}$, $\varnothing_{\text{obj}}$ stand for [] and {}. By Theorem 2, v is determined by x, so $\exists v$ could equally be read as "the v".

Predicates such as "the number at π is at least t" are relations of the same shape, with "get(v, π) = ℓ" replaced

by "get(v, π) is a leaf satisfying φ". The statement (h, π, ℓ or φ) is public. Only x is witness.

7.2 The walk decides queries

Let walk(x) be the row list that the stage inverses S5 → S2 read from x, that is, decode without the final unflatten (`walkRows`).

> **Theorem 4 (walk soundness and completeness).** Let $v \in \mathscr{W}$ and x = encode(v) ‖ 0^z. Then walk(x) = flatten(v) (`walkRows_encode`), and for every path π and leaf ℓ:
>
> (a) (π, ℓ) ∈ walk(x) ⟺ get(v, π) = ℓ (`query_rows`, from `row_iff_get`);
>
> (b) get(v, π) is undefined ⟺ no row of walk(x) has a path that extends π (`absent_rows`, from `path_exists_iff`).

The theorem makes no hash assumption. It says that on committed canonical encodings, the row walk is exactly the query semantics. A non-existence proof must check every row, but the order of Proposition 3 permits a shortcut. The rows whose paths extend π form a contiguous interval of the sorted list: if $r \prec s \prec t$ and both r and t extend π, then s agrees with them on the first |π| components and so extends π as well. Non-existence can therefore be shown by two adjacent rows $r \prec \pi \prec r'$ where r' does not extend π, or by $\pi \prec r_1$ where the first row r_1 does not extend π, or by $r_{\text{last}} \prec \pi$ for the last row r_{last}. Every extension e of π satisfies $\pi \preccurlyeq e$, and also $e \prec r'$ whenever $\pi \prec r'$ and r' does not extend π, because r' differs from π at some position below |π| and is larger there. So no extension of π can sit between r and r', before r_1, or after r_{last} (which precedes π, and π precedes all its extensions). The walk that locates the two rows must still be constrained from the start.

> **Theorem 5 (query soundness).** Suppose H is collision resistant (A2 below), the argument system is knowledge sound (A4), and the circuit enforces $H(x) = h$ and "(π, ℓ) ∈ walk(x)", or for non-existence "no row of walk(x) extends π". Let $v \in \mathscr{W}$ be fixed in advance, or output by the party that creates the commitment, and let $h = H(x_v)$. Then for every such v, no efficient prover, even one given v, makes the verifier accept (h, π, ℓ) unless get(v, π) = ℓ, or accept (h, π) as non-existence unless get(v, π) is undefined, except with negligible probability.

Proof. The reduction is given v, so it can compute x_v. Knowledge soundness yields a witness x' with $H(x') = h$ that passes the walk check. If $x' \neq x_v$, the reduction outputs (x', x_v), a collision for H. So $x' = x_v$, and Theorem 4 applies. □

The hypothesis "h commits to a canonical encoding" is a property of whoever created h. In the store of §8 it becomes an invariant that every accepted write must establish (§8.3).

7.3 What a sound circuit must constrain

Theorem 5 holds only if the circuit actually enforces the walk. Each of the following must be a constraint, not merely a value computed off-circuit.

1. **Digits.** Each element is decomposed into 76 digits in [0, 9]. By Proposition 4 the decomposition is unique.
2. **Guards and padding.** Each non-zero element has exactly one guard 1 at position |b|, with zeros above it. Zero elements are padding.
3. **Unit automaton, anchored.** The parse of §5 (S4) runs from the first digit of the first element across all element boundaries, and determines which positions are unit headers.
4. **Integer and row automaton.** The parse expands runs, completes pending continuations with the next final token or single (S3), and tracks row structure: row length, component kinds and lengths, leaf tags and string lengths (S2).
5. **Match.** The match compares, at a row start marked by item 4, the expanded tokens of the row with toks(rowCode(π, ℓ)). For non-existence, it checks every row, or two adjacent rows that bracket π. Comparing tokens is equivalent to comparing integers, since toks has a left inverse (Lemma 3).
6. **Commitment.** $H(x) = h$ must be enforced on the same x that is walked.

Item 3 cannot be relaxed to "some position where the query row appears". The code is prefix-free but not self-synchronizing.

> **Proposition 5 (anchoring is necessary).** Let v = {"x": s} with s the string of code units [1, 1, 97, 2, 1, 0, 1]. Then ints(flatten(v)) = [1, 1, 120, 3, 7] ‖ ints([(a, 1)]), so the integer stream of v contains the complete row code of {"a": 1}, although get(v, [a]) is undefined. (`spoof_ints`, `spoof_absent`)

A circuit that lets the prover choose where a row starts would therefore prove "a = 1" about a document that has no key a. Since strings may contain arbitrary code units, any row code can be embedded this way.

Item 4 matches a query on expanded tokens, not on raw digits. The digits of one row depend on its neighbours: a run may span two rows, and a block boundary may cut any unit (§4.5). The expanded token sequence of a row depends only on the row.

7.4 Analytic cost estimate

We estimate R1CS constraints for a circuit that enforces items 1–6 by a fixed, unrolled scan over all $D = 76N$ digit positions. For $N = 256$, $D = 19{,}456$. The counts below follow from the construction sketched for each item. They are ESTIMATED, not measured, and a concrete circuit may differ by constant factors.

item	construction	low	high
1 digits	4 bits, 4 booleanity, 2 for ≤ 9	6	6
2 guard	guard and payload flags	3	3
3 unit automaton	one-hot of 4 modes, counter, 0/9 tests	12	20
4 row automaton	≈ 12 modes, 2 counters, zero tests	15	30
5 match	multiplexer over Q tokens, equality, flag	$Q + 4$	$Q + 4$
total per position		$40 + Q$	$63 + Q$

Table 3. Per-position constraint estimate ESTIMATED. Q is the number of tokens in toks(rowCode(π, ℓ)). All counters stay below $75N$, so they fit in a single final token and in $\mathbb{F}_p$: a length or count never exceeds the number of digits.

For item 6, a two-level hash of 256 elements uses 16 calls of P_{16} on the chunks and one on their digests, 17 calls in all. With the round numbers recommended in [12] for this field and S-box at the 128-bit level ($R_F = 8$, $R_P = 68$ for $t = 17$), each call costs $3(17{\cdot}8 + 68) = 612$ constraints, so the hash costs 10,404 constraints.

Q	low	high
8	944,292	1,391,780
32	1,411,236	1,858,724

Table 4. Totals for $N = 256$ ESTIMATED, computed as D·(per position) + 10,404. Both sizes fit a 2^{21} evaluation domain. The low end for $Q = 8$ fits 2^{20}.

The scan dominates: the commitment is at most about 1.1% of the total. Cost is linear in N. Most of the scan is selection logic, so argument systems with lookup tables or custom gates should reduce it. We do not quantify that here.

8 Authenticated two-level store

8.1 Structure

A database is a two-level tree (Fig. 4). The *collection tree* of collection c is an SMT of depth d_2. Its default leaf is 0, and the slot of a document with id s is $\iota(s)$ (§8.2). An occupied slot holds $h = H(x_v)$, or $H(x_v, \rho)$ with a secret salt ρ when low-entropy documents must stay hidden. The *database tree* is an SMT of depth d_1 over collection indices, whose slot c holds the root C_c of collection c. Its default leaf is the empty-collection root Z_{d_2}, so a collection that was never written is the empty collection. Internal nodes are $P_2(l, r)$. A verifier stores only the database root R.

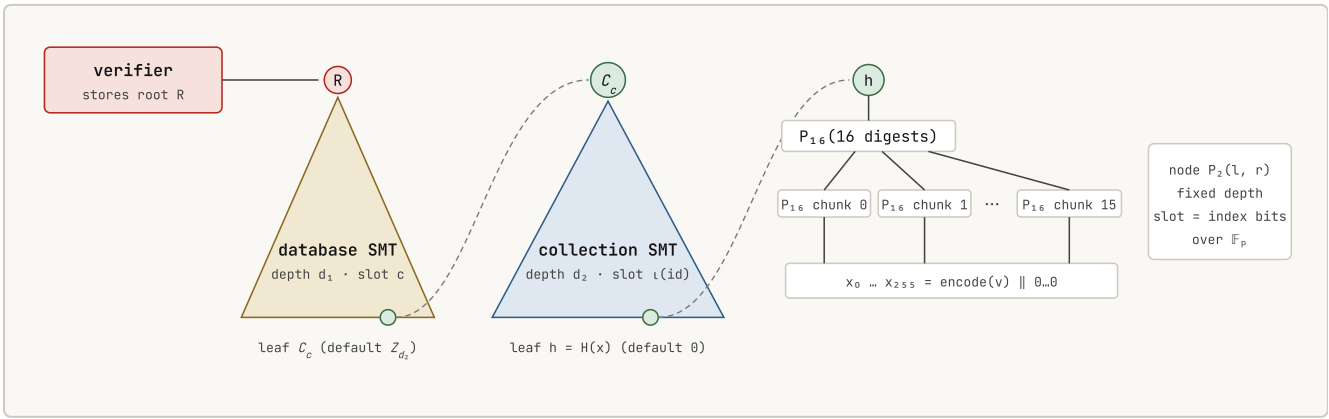


Figure 4. The authenticated store for $N = 256$. A document vector is hashed in two levels of P_{16} into h, which is the leaf of its collection tree. The collection root is the leaf of the database tree. The verifier keeps only R.

8.2 Ids to leaf indices

Leaves are addressed by an injective, unhashed map, so distinct ids never collide, and this holds unconditionally rather than under a hash assumption. The cost is that the depth bounds the id length.

Definition 11 (guarded radix index). For an alphabet of size $b \geq 2$ and an id $s = s_1 \ldots s_n$ with symbols $s_i < b$, let $\iota_b(s)$ be the integer whose base-b digits are 1, s_1, ..., s_n. That is, $\iota_b(s) = b^n + \Sigma_i\, s_i b^{n-i}$ (`idIndex`).

Theorem 6 (id map). Let $b \geq 2$. (a) ι_b has a left inverse and is therefore injective on all ids, including ids with leading zero symbols (`unIndex_idIndex`, `idIndex_injective`). (b) $b^n \leq \iota_b(s) < 2b^n$ (`idIndex_bounds`), and the upper bound is attained: the id $(b-1)^n$ maps to $2b^n - 1$ (`idIndex_max`). (c) Hence every id of length n has its own slot in a depth-d tree if and only if $2b^n \leq 2^d$. If moreover $d \leq 253$, the index is below p (`idIndex_fits`, `idIndex_lt_p`).

The guard digit is essential. Reading the id directly as a base-b number would identify s with $0 \| s$. Table 5 gives the largest admissible length for common alphabets.

alphabet	$d = 32$	100	168	184	253
$b = 64$	5	16	27	30	42
$b = 256$ (bytes)	3	12	20	22	31
$b = 100$	4	14	25	27	37

Table 5. Longest id length $n = n_{\max}(b, d)$ with $2b^n \leq 2^d$. Each entry is proved tight by `decide` (`capacity_64`, `capacity_256`, `capacity_100`). The column $b = 100$ covers any alphabet whose symbols are written as two decimal digits.

For $b = 2^k$ the condition reads $kn + 1 \leq d$, so $n = \lfloor (d - 1)/k \rfloor$. Ids longer than n are outside the domain of ι for that depth. The alternative is to hash ids, which turns collision freedom into an assumption. The map is invertible, so a slot index reveals its id.

8.3 Relations

Write $\mathrm{Open}_d(R, k, u; \sigma)$ for the statement that the siblings $\sigma = (\sigma_1, \ldots, \sigma_d)$, hashed with leaf value u along the bits of k, give root R. Write $\mathrm{Root}_d(k, u; \sigma)$ for the root this computes.

Definition 12 (store relations). Fix an alphabet size b and write $\iota = \iota_b$ and $k = \iota(s)$. A pair (c, s) is *admissible*, $\mathrm{Adm}(c, s)$, if $s_i < b$ for all i, $\iota(s) < 2^{d_2}$ (which holds whenever $|s| \leq n_{\max}(b, d_2)$ of Table 5), and $c < 2^{d_1}$.

Membership $\mathscr{R}_\in = \{((R, c, s, h); C, \sigma, \tau) : \mathrm{Adm}(c, s) \wedge \mathrm{Open}_{d_1}(R, c, C; \tau) \wedge \mathrm{Open}_{d_2}(C, k, h; \sigma) \wedge h \neq 0\}$.

Non-membership $\mathscr{R}_\notin$ is the same relation, including $\mathrm{Adm}(c, s)$, with $h = 0$.

Database query $\mathscr{R}_{\mathrm{dq}}$ joins $\mathscr{R}_\in$ with $\mathscr{R}_{\mathrm{q}}$ or $\mathscr{R}_\nexists$ of §7 on the same h, and so includes $\mathrm{Adm}(c, s)$. Its statement is (R, c, s, π, ℓ), and h becomes a witness.

Write. The write relation $\mathscr{R}_{\mathrm{w}}((R, R'); c, s, x, h_{\mathrm{old}}, C, \sigma, \tau)$ holds if: (o) $\mathrm{Adm}(c, s)$; (i) $\mathrm{Open}_{d_1}(R, c, C; \tau)$ and $\mathrm{Open}_{d_2}(C, k, h_{\mathrm{old}}; \sigma)$; (ii) $x = x_v$ for some $v \in \mathscr{W}$ (canonicity check); (iii) $C' = \mathrm{Root}_{d_2}(k, H(x); \sigma)$; (iv) $R' = \mathrm{Root}_{d_1}(c, C'; \tau)$. A deletion uses leaf 0 in place of $H(x)$.

Batch $\mathscr{R}_{\mathrm{B}}((R_0, R_k); (R_1, \ldots, R_{k-1}), w_{1..k})$ holds if, for each i, $\mathscr{R}_{\mathrm{w}}((R_{i-1}, R_i); w_i)$ or $(w_i = \bot$ and $R_i = R_{i-1})$. The second case is a no-op slot.

Two features of these relations are essential. Both endpoints R_0 and R_k of a batch are *public*, and the verifier accepts a batch only if R_0 equals its stored root. In $\mathscr{R}_{\mathrm{w}}$, the database leaf that is opened in (i) is the *same* C under which the document is opened, and (iii)–(iv) reuse the same siblings. A circuit that omits either link proves a weaker relation than $\mathscr{R}_{\mathrm{B}}$. Condition (ii) maintains the invariant that Theorem 5 assumes: every non-zero document leaf commits to a canonical encoding. One way to enforce it is to check $\mathrm{walk}(x) = \mathrm{flatten}(v)$ for a decoded $v \in \mathscr{W}$ together with re-encoding. We do not give a constraint-level characterization of the image of encode here.

8.4 Assumptions and security

- **A1** P_2 is collision resistant.
- **A2** H (the document hash) is collision resistant.
- **A3** It is infeasible to find x with $H(x) = 0$, the empty leaf value.
- **A4** The argument system is knowledge sound for the relations above. For Groth16 [10] this holds in the generic group model with an honestly generated, circuit-specific setup.

Proposition 6 (position binding). Under A1, no efficient adversary can output R, k and valid openings $\mathrm{Open}_d(R, k, u; \sigma)$ and $\mathrm{Open}_d(R, k, u'; \sigma')$ with $u \neq u'$.

Proof. The two paths start at different leaves and end at the same root. At the first level where the computed nodes coincide, the two hash inputs differ, which is a collision for P_2. □ Consequently, under A1 and A3, a document cannot be shown both present and absent. Distinct ids never share a slot when both are admissible: by Theorem 6(a) their indices differ, and both are below 2^{d_2}, so their d_2 bits differ. Without Adm the bits of k would be k mod 2^{d_2}, and distinct ids could share a slot.

Theorem 7 (transition soundness). Assume A1–A4. Let $\mathscr{S}_0$ be an abstract store (for each collection index $c < 2^{d_1}$, a finite map from admissible ids to canonical values), known to the extractor, whose two-level root is R_0. For every efficient prover there is an efficient extractor such that, whenever the verifier accepts the prover's batch proof for (R_0, R_k), the extractor outputs operations $w_1, \ldots, w_k$, each a write of a value in $\mathscr{W}$ to an admissible (c, s), a deletion, or $\perp$, such that, except with negligible probability, R_k is the root of $\mathscr{S}_k = w_k(\cdots w_1(\mathscr{S}_0))$ and every document in $\mathscr{S}_k$ is canonical.

Proof sketch. A4 gives an extractor for $\mathscr{R}_{\mathrm{B}}$; from the extracted witness it reads off each w_i as $(c, s,$ decode$(x))$, a deletion, or $\perp$. Induct on i, assuming R_{i-1} is the root of $\mathscr{S}_{i-1}$. By Proposition 6, the openings in (i) agree with $\mathscr{S}_{i-1}$ except with negligible probability: C is the true root of collection c, and h_{old} is the true leaf. By Adm, slot k belongs to s alone. Recomputing the root along the same siblings with a new leaf gives the root of the tree in which only that leaf changed. By A3 the new leaf $H(x)$ of a write is non-zero, so it is not confused with the empty leaf of a deletion. So R_i is the root of $\mathscr{S}_i = w_i(\mathscr{S}_{i-1})$. Condition (ii) and Theorem 1 make the written value decode$(x) \in \mathscr{W}$ unique, and A2 binds it to its leaf. A no-op leaves the root and the store unchanged. □

What statements reveal. Zero knowledge hides the witness, not the statement. A database query reveals R, c, the id (ι is invertible), π, and ℓ or the predicate operand. Without a salt, h is a deterministic function of the document, so documents from a small space can be recovered by enumeration. Salting the leaf, $h = H(x, \rho)$, removes this. A2 must then hold for the salted hash.

8.5 Analytic cost of openings

With P_2 at width 3 and the round numbers of [12] ($R_{\mathrm{F}} = 8$, $R_{\mathrm{P}} = 57$), each level costs $3(3{\cdot}8 + 57) = 243$ constraints for the hash, plus about 3 for the path bit and the left/right selection. That is about 246 per level ESTIMATED. Membership at both levels costs about $246(d_1 + d_2)$. A write recomputes both paths with the same siblings, about $2{\cdot}246(d_1 + d_2)$, plus the document hash (10,404 for $N = 256$) and the canonicity check of (ii). For $d_1 = 8$ and $d_2 = 168$, membership costs about 43,296 constraints. A write costs about 86,592 for the two path recomputations, plus 10,404 for the document hash (96,996 in all), plus the canonicity check of (ii), which is not estimated here. A query therefore pays more for the walk of §7.4 than for the openings.

9 Related work

Hash trees and SMTs. Merkle introduced tree authentication [1, 2]. Laurie and Kasper proposed sparse trees indexed by hash values, to prove the presence or absence of revoked certificates [3]. Dahlberg, Pulls and Peeters gave recursive definitions, caching strategies and secure (non-)membership proofs [4]. The store of §8 uses a standard fixed-depth SMT. It differs only in indexing leaves by an injective, unhashed map, which trades a hard length bound (Table 5) for collision freedom by construction.

Authenticated data structures and verifiable databases. The architecture fits Tamassia's three-party model [5]: a source, an untrusted responder, and a verifier that holds a digest. IntegriDB [6] and vSQL [7] verify much richer SQL queries over outsourced, updatable databases. Our query language is a path lookup with simple predicates, but its proofs are constant-size and checked against a single stored root.

Arguments and hashing. Proofs come from the QAP line [8, 9] through Groth16 [10]. PLONK [11] replaces the circuit-specific setup with a universal one, which matters here because every change to N, Q or the depths changes the circuit. Poseidon [12] is the standard arithmetization-friendly hash, and our cost formulas use its round structure.

Canonical JSON. RFC 8259 [17] fixes the data model that our tags cover. RFC 8785 [13] canonicalizes JSON as *text*, for byte-oriented hashing and signing. Our encoding canonicalizes into *field elements*, with a grammar that a fixed circuit can walk, and injectivity and one-level finite-map canonicity are mechanized (nested canonicity follows by a paper induction).

Proving statements about structured data. DECO [18] proves statements about data received from a TLS server, and zkLogin [19] proves claims from OpenID Connect tokens inside a SNARK. Both parse JSON text in the circuit and target specific fields. Cinderella [20] proves policies over parsed X.509 certificates in a SNARK. We prove path–value statements about documents that were arithmetized and committed in advance. This avoids an in-circuit lexer, but documents must be written in the encoding first. Verifiable credentials [21] can use signature schemes with selective disclosure. Path proofs against a hash commitment need no special issuer signature, but they need salting to hide low-entropy fields. Systems that advance a stored root with validity proofs follow the rollup pattern surveyed in [16]. Vector-commitment trees [15] reduce opening size and could replace the SMT.

10 Conclusion

On the domain $\mathscr{W}$ (decimal numbers in shortest form, objects as finite maps with unique keys, empty containers as explicit leaves), flattening, type-tagged integer coding, length-prefixed tokens with run compression under the one-digit-tail rule, and guarded 75-digit packing compose into an encoding into $\mathbb{F}_p$ with a proved left inverse. The encoding is therefore injective, canonical on finite maps, and below p element by element, with unique in-field digit decomposition. These facts are machine-checked in general, except that canonicity for nested finite maps (Corollary 1) combines mechanized one-level lemmas with a paper induction. On top of the encoding, path queries and non-existence are exactly decided by an anchored walk over the token grammar. The soundness of a query circuit is a separate property: it depends on whether the circuit constrains that walk. We have listed the six things such a circuit must constrain and estimated their cost at roughly 10^6 constraints for 256 elements. The two-level store adds an injective guarded index map with tight capacity bounds and transition relations whose security reduces to collision resistance, infeasibility of hashing to the empty leaf value, and knowledge soundness. Two parts remain open: a constraint-level characterization of the image of the encoder, for an efficient canonicity check at write time, and a circuit construction that matches the estimate of §7.4.

References

1. R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. Ph.D. dissertation, Stanford University, 1979.
2. R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology – CRYPTO '87*, LNCS 293, Springer, 1988, pp. 369–378.
3. B. Laurie, E. Kasper. *Revocation Transparency*. Technical report, September 2012.
4. R. Dahlberg, T. Pulls, R. Peeters. Efficient sparse Merkle trees: caching strategies and secure (non-)membership proofs. In *NordSec 2016*, LNCS 10014, Springer, 2016, pp. 199–215.
5. R. Tamassia. Authenticated data structures. In *Algorithms – ESA 2003*, LNCS 2832, Springer, 2003, pp. 2–5.
6. Y. Zhang, J. Katz, C. Papamanthou. IntegriDB: verifiable SQL for outsourced databases. In *Proc. ACM CCS 2015*, pp. 1480–1491.
7. Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, C. Papamanthou. vSQL: verifying arbitrary SQL queries over dynamic outsourced databases. In *Proc. IEEE S&P 2017*.
8. R. Gennaro, C. Gentry, B. Parno, M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT 2013*, LNCS 7881, Springer, 2013.
9. B. Parno, J. Howell, C. Gentry, M. Raykova. Pinocchio: nearly practical verifiable computation. In *Proc. IEEE S&P 2013*, pp. 238–252.
10. J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016, Part II*, LNCS 9666, Springer, 2016, pp. 305–326.
11. A. Gabizon, Z. J. Williamson, O. Ciobotaru. PLONK: permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. IACR ePrint 2019/953, 2019.
12. L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, M. Schofnegger. Poseidon: a new hash function for zero-knowledge proof systems. In *Proc. 30th USENIX Security Symposium*, 2021, pp. 519–535.
13. A. Rundgren, B. Jordan, S. Erdtman. *JSON Canonicalization Scheme (JCS)*. RFC 8785, IETF, 2020.
14. L. de Moura, S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, LNCS 12699, Springer, 2021, pp. 625–635.

15. J. Kuszmaul. *Verkle Trees*. MIT PRIMES research paper, 2018.
16. L. T. Thibault, T. Sarry, A. S. Hafid. Blockchain scaling using rollups: a comprehensive survey. *IEEE Access* 10, 2022.
17. T. Bray (Ed.). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259, IETF, 2017.
18. F. Zhang, D. Maram, H. Malvai, S. Goldfeder, A. Juels. DECO: liberating web data using decentralized oracles for TLS. In *Proc. ACM CCS 2020*.
19. F. Baldimtsi et al. zkLogin: privacy-preserving blockchain authentication with existing credentials. In *Proc. ACM CCS 2024*.
20. A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, B. Parno. Cinderella: turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *Proc. IEEE S&P 2016*, pp. 235–254.
21. W3C. *Verifiable Credentials Data Model v2.0*. W3C Recommendation, 15 May 2025.

A Appendix: the Lean 4 development

The listing below is one file, `Canon.lean` (1,905 lines). It uses the Lean 4 [14] core library only, with no Mathlib, and compiles with Lean 4.16.0 with no errors. The only messages are three unused-variable lint warnings and the axiom report. It contains no `sorry`, `admit` or `axiom`. `native_decide` appears only in two concrete `examples`, the element of `{"a":1}` and the stream of [123456789, 1, 1], and no theorem depends on them. Table 6 maps the results in the text to their Lean names and gives the axioms reported by `#print axioms`. "std" means [propext, Quot.sound], which are Lean's standard axioms, and "+choice" adds Classical.choice. Both are part of Lean's standard foundations.

result	Lean name	axioms
Theorem 1	`decode_encode`	std + choice
Theorem 2	`encode_injective`, `encode_injective_pad`	std + choice
Lemma 1	`unflatten_flatten`	std + choice
Lemma 2	`parseRows_ints'`	std
Lemma 3	`tokInts_toks`	std
Lemma 4	`decodeTok_stream`, `decodeTok_units`	std
Lemma 5, Remark 1	`unpack_pack`, `unpack_segments`	std + choice
Theorem 3	`encode_lt_p`, `elem_lt_p`	std + choice
capacity	`encode_length`, `payload_bits`	std; none
p bounds	`p_bounds`, `p_bits`	none
Proposition 1	`num_canon_unique`	std
Proposition 2	`slt_*`, `sorted_unique`, `canonical_rep`	std (+ choice)
Proposition 3	`cLt_eq_code`, `flatten_sorted`	std (+ choice)
Proposition 4	`digits_mod_p_unique`	std
Theorem 4	`walkRows_encode`, `query_rows`, `absent_rows`	std + choice
Proposition 5	`spoof_ints`, `spoof_absent`	none
Theorem 6	`unIndex_idIndex`, `idIndex_bounds`, `idIndex_max`, `idIndex_lt_p`	std
Table 5	`capacity_64`, `capacity_256`, `capacity_100`	none
§4.5 example	`empty_marker_distinguishes`	std + choice
Fig. 2	`one_digit_tail_roundtrip`	std

Table 6. Results and their Lean statements.

```
/-
  Canon.lean — canonical arithmetization of JSON into BN254 field elements.

  Lean 4.16.0, core library only.  Compile:  lean Canon.lean

  Pipeline (each stage has a proved left inverse on its image):
    J --flatten--> rows --ints--> ℕ* --toks--> tokens --compress--> units
      --digits--> decimal stream --pack--> field elements (guard 1 + ≤ 75 digits)

  Main results:
    decode_encode       decode (encode v ++ 0^z) = some v        for every v with WF v
    encode_injective    WF v → WF w → encode v = encode w → v = w
    encode_lt_p         every element e satisfies 1 ≤ e < 10^76 < p
    decodeTok_stream    run-compression round trip
    unIndex_idIndex     id → index left inverse; idIndex_injective
-/
namespace Canon
```

```
/-- Reverse (snoc) induction on lists. -/
theorem snocInd {α : Type} {P : List α → Prop} (h0 : P [])
    (hs : ∀ l d, P l → P (l ++ [d])) : ∀ l, P l := by
  intro l
  rw [← List.reverse_reverse l]
  induction l.reverse with
  | nil => exact h0
  | cons d r ih => rw [List.reverse_cons]; exact hs _ _ ih

/-! ## Decimal digits -/

/-- Horner reading of a most-significant-first digit list. -/
def ofD (l : List Nat) : Nat := l.foldl (fun a d => 10 * a + d) 0

theorem ofD_snoc (l : List Nat) (d : Nat) : ofD (l ++ [d]) = 10 * ofD l + d := by
  simp [ofD, List.foldl_append]

/-- Least-significant-first decimal digits of `n` (`[0]` for `0`). -/
def dLE (n : Nat) : List Nat := if n < 10 then [n] else n % 10 :: dLE (n / 10)
decreasing_by omega

/-- Most-significant-first decimal digits. -/
def digits (n : Nat) : List Nat := (dLE n).reverse

theorem ofD_digits (n : Nat) : ofD (digits n) = n := by
  induction n using Nat.strongRecOn with
  | ind n ih =>
    unfold digits
    rw [dLE]
    by_cases h : n < 10
    · simp [h, ofD]
    · simp only [h, if_false, List.reverse_cons]
      rw [ofD_snoc]
      have := ih (n / 10) (by omega)
      unfold digits at this
      rw [this]; omega

theorem digits_ne_nil (n : Nat) : digits n ≠ [] := by
  unfold digits; rw [dLE]; split <;> simp

theorem digits_lt (n : Nat) : ∀ x ∈ digits n, x < 10 := by
  induction n using Nat.strongRecOn with
  | ind n ih =>
    unfold digits; rw [dLE]
    by_cases h : n < 10
    · simp [h]
    · simp only [h, if_false, List.reverse_cons, List.mem_append, List.mem_singleton]
      intro x hx
      rcases hx with hx | hx
      · exact ih (n / 10) (by omega) x hx
      · omega

theorem ofD_lt (l : List Nat) (hl : ∀ x ∈ l, x < 10) : ofD l < 10 ^ l.length := by
  revert hl
  refine snocInd (P := fun l => (∀ x ∈ l, x < 10) → ofD l < 10 ^ l.length) ?_ ?_ l
  · intro; simp [ofD]
  · intro l d ih hl
    rw [ofD_snoc]
    have h1 := ih (fun x hx => hl x (by simp [hx]))
    have h2 := hl d (by simp)
    simp only [List.length_append, List.length_singleton, Nat.pow_succ]
    omega

/-- `k` decimal digits of `v`, most significant first (i.e. of `v mod 10^k`). -/
def fixD : Nat → Nat → List Nat
  | 0, _ => []
  | k + 1, v => fixD k (v / 10) ++ [v % 10]

theorem fixD_guard (l : List Nat) (hl : ∀ x ∈ l, x < 10) (a : Nat) :
    fixD l.length (a * 10 ^ l.length + ofD l) = l := by
  revert hl a
  refine snocInd (P := fun l => (∀ x ∈ l, x < 10) → ∀ a,
    fixD l.length (a * 10 ^ l.length + ofD l) = l) ?_ ?_ l
```

```
  · intros; simp [fixD]
  · intro l d ih hl a
    have hd := hl d (by simp)
    have ih' := ih (fun x hx => hl x (by simp [hx])) a
    simp only [List.length_append, List.length_singleton, fixD, ofD_snoc, Nat.pow_succ]
    have e : a * (10 ^ l.length * 10) + (10 * ofD l + d)
        = d + (a * 10 ^ l.length + ofD l) * 10 := by
      rw [Nat.add_mul, ← Nat.mul_assoc]; omega
    rw [e, Nat.add_mul_div_right _ _ (by decide), Nat.add_mul_mod_self_right,
        Nat.div_eq_of_lt hd, Nat.mod_eq_of_lt hd, Nat.zero_add, ih']

/-- Number of decimal digits. -/
def nd (n : Nat) : Nat := if n < 10 then 1 else nd (n / 10) + 1
decreasing_by omega

theorem nd_eq (k n : Nat) (h1 : 10 ^ k ≤ n) (h2 : n < 10 ^ (k + 1)) : nd n = k + 1 := by
  induction k generalizing n with
  | zero => rw [nd]; simp at h2; simp [h2]
  | succ k ih =>
    rw [nd]
    have hp : 10 ≤ 10 ^ (k + 1) := by
      have := Nat.one_le_two_pow (n := k)
      have : 1 ≤ 10 ^ k := Nat.pos_pow_of_pos k (by decide)
      rw [Nat.pow_succ]; omega
    have hn : ¬ n < 10 := by omega
    simp only [hn, if_false]
    rw [ih (n / 10)]
    · exact (Nat.le_div_iff_mul_le (by decide)).2 (by rw [← Nat.pow_succ]; exact h1)
    · exact (Nat.div_lt_iff_lt_mul (by decide)).2 (by rw [← Nat.pow_succ]; exact h2)


/-! ## Guarded packing into field elements -/

/-- The BN254 scalar field modulus. -/
def p : Nat := 21888242871839275222246405745257275088548364400416034343698204186575808495617

theorem p_bounds : 10 ^ 76 < p ∧ p < 10 ^ 77 := by decide
theorem p_bits : 2 ^ 253 < p ∧ p < 2 ^ 254 := by decide
theorem payload_bits : 2 ^ 249 < 10 ^ 75 ∧ 10 ^ 75 < 2 ^ 250 := by decide

/-- Payload digits per element. -/
def W : Nat := 75

/-- Cut a digit stream into consecutive blocks of `W` digits (the last may be shorter). -/
def blocks (s : List Nat) : List (List Nat) :=
  if h : s = [] then [] else s.take W :: blocks (s.drop W)
termination_by s.length
decreasing_by
  simp only [List.length_drop, W]
  have := List.length_pos.2 h; omega

/-- A block becomes the integer whose decimal form is `1` followed by the block. -/
def elem (b : List Nat) : Nat := 10 ^ b.length + ofD b

def pack (s : List Nat) : List Nat := (blocks s).map elem

/-- Read an element back: drop the guard digit, keep the rest with leading zeros. -/
def unElem (e : Nat) : List Nat := fixD (nd e - 1) e

/-- Zero elements are padding and are ignored. -/
def unpack (es : List Nat) : List Nat := ((es.filter (· != 0)).map unElem).flatten

theorem elem_pos (b : List Nat) : 0 < elem b := by
  unfold elem; have := Nat.pos_pow_of_pos b.length (show 0 < 10 by decide); omega

theorem elem_bounds (b : List Nat) (hb : ∀ x ∈ b, x < 10) :
    10 ^ b.length ≤ elem b ∧ elem b < 10 ^ (b.length + 1) := by
  have := ofD_lt b hb
  unfold elem; rw [Nat.pow_succ]; omega

theorem unElem_elem (b : List Nat) (hb : ∀ x ∈ b, x < 10) : unElem (elem b) = b := by
  have ⟨h1, h2⟩ := elem_bounds b hb
  unfold unElem
  rw [nd_eq _ _ h1 h2, Nat.add_sub_cancel]
```

```
  have := fixD_guard b hb 1
  rw [Nat.one_mul] at this
  exact this

/-- Field bound: an element with at most `W` payload digits is below `10^76 < p`. -/
theorem elem_lt_p (b : List Nat) (hb : ∀ x ∈ b, x < 10) (hl : b.length ≤ W) :
    elem b < 10 ^ 76 ∧ elem b < p := by
  have h := (elem_bounds b hb).2
  have : 10 ^ (b.length + 1) ≤ 10 ^ 76 :=
    Nat.pow_le_pow_right (by decide) (by unfold W at hl; omega)
  exact ⟨by omega, by have := p_bounds.1; omega⟩

/-- Segmentation-independent round trip: for ANY split of a digit stream into blocks,
    reading back the guarded elements (with any zero padding) recovers the stream. -/
theorem unpack_segments (bs : List (List Nat)) (hb : ∀ b ∈ bs, ∀ x ∈ b, x < 10) (z : Nat) :
    unpack (bs.map elem ++ List.replicate z 0) = bs.flatten := by
  unfold unpack
  rw [List.filter_append]
  have hz : (List.replicate z 0).filter (· != 0) = [] := by
    induction z with
    | zero => rfl
    | succ z ih => simp [List.replicate_succ]
  rw [hz, List.append_nil]
  induction bs with
  | nil => rfl
  | cons b bs ih =>
    have hpos := elem_pos b
    have : (elem b != 0) = true := by simp; omega
    simp only [List.map_cons, List.filter_cons, this, if_true, List.flatten_cons]
    rw [unElem_elem b (hb b (by simp)), ih (fun b' h' => hb b' (by simp [h']))]

theorem blocks_spec (s : List Nat) :
    (blocks s).flatten = s ∧ ∀ b ∈ blocks s, b.length ≤ W ∧ ∀ x ∈ b, x ∈ s := by
  induction s using (measure List.length).wf.induction with
  | h s ih =>
    rw [blocks]
    by_cases h : s = []
    · simp [h]
    · simp only [h, dif_neg, not_false_eq_true]
      have hl := List.length_pos.2 h
      have ⟨ih1, ih2⟩ := ih (s.drop W) (show (s.drop W).length < s.length by simp [W]; omega)
      refine ⟨by rw [List.flatten_cons, ih1, List.take_append_drop], ?_⟩
      intro b hb
      simp only [List.mem_cons] at hb
      rcases hb with rfl | hb
      · exact ⟨by simp [W]; exact Nat.min_le_left _ _, fun x hx => List.mem_of_mem_take hx⟩
      · have ⟨a1, a2⟩ := ih2 b hb
        exact ⟨a1, fun x hx => List.mem_of_mem_drop (a2 x hx)⟩

theorem blocks_length (s : List Nat) (N : Nat) (h : s.length ≤ W * N) :
    (blocks s).length ≤ N := by
  induction s using (measure List.length).wf.induction generalizing N with
  | h s ih =>
    rw [blocks]
    by_cases hs : s = []
    · simp [hs]
    · simp only [hs, dif_neg, not_false_eq_true, List.length_cons]
      have hl := List.length_pos.2 hs
      cases N with
      | zero => simp only [W, Nat.mul_zero] at h; omega
      | succ N =>
        have := ih (s.drop W) (show (s.drop W).length < s.length by simp [W]; omega) N
          (by simp only [W, Nat.mul_succ, List.length_drop] at h ⊢; omega)
        omega

/-- Packing round trip with padding to any length. -/
theorem unpack_pack (s : List Nat) (hs : ∀ x ∈ s, x < 10) (z : Nat) :
    unpack (pack s ++ List.replicate z 0) = s := by
  have ⟨h1, h2⟩ := blocks_spec s
  unfold pack
  rw [unpack_segments _ (fun b hb x hx => hs x ((h2 b hb).2 x hx)), h1]

theorem pack_lt_p (s : List Nat) (hs : ∀ x ∈ s, x < 10) :
    ∀ e ∈ pack s, 10 ^ 0 ≤ e ∧ e < 10 ^ 76 ∧ e < p := by
```

```
  intro e he
  unfold pack at he
  simp only [List.mem_map] at he
  obtain ⟨b, hb, rfl⟩ := he
  have ⟨h1, h2⟩ := (blocks_spec s).2 b hb
  have := elem_lt_p b (fun x hx => hs x (h2 x hx)) h1
  exact ⟨by have := elem_pos b; simp; omega, this⟩


/-! ## Digit tokens: 8-digit chunks with a length or continuation digit -/

inductive Tok where
  | cont (ds : List Nat)   -- written `9 d₁ … d₈`: eight digits, the integer continues
  | fin (ds : List Nat)    -- written `ℓ d₁ … d_ℓ`, `1 ≤ ℓ ≤ 8`: the integer ends
  deriving DecidableEq, Repr

def tokOK : Tok → Prop
  | .cont ds => ds.length = 8 ∧ ∀ x ∈ ds, x < 10
  | .fin ds => 1 ≤ ds.length ∧ ds.length ≤ 8 ∧ ∀ x ∈ ds, x < 10

/-- Cut a digit string into 8-digit chunks from the left. -/
def chunks (ds : List Nat) : List Tok :=
  if h : ds.length ≤ 8 then [.fin ds] else .cont (ds.take 8) :: chunks (ds.drop 8)
termination_by ds.length
decreasing_by simp; omega

def intToks (n : Nat) : List Tok := chunks (digits n)
def toks (ns : List Nat) : List Tok := ns.flatMap intToks

/-- Token decoder: continuation chunks accumulate until a final chunk completes the integer. -/
def tokInts : List Nat → List Tok → Option (List Nat)
  | acc, [] => if acc = [] then some [] else none
  | acc, .cont ds :: ts => tokInts (acc ++ ds) ts
  | acc, .fin ds :: ts => (tokInts [] ts).map (ofD (acc ++ ds) :: ·)

theorem tokInts_chunks (ds : List Nat) (h : ds ≠ []) (acc : List Nat) (ts : List Tok) :
    tokInts acc (chunks ds ++ ts) = (tokInts [] ts).map (ofD (acc ++ ds) :: ·) := by
  induction ds using (measure List.length).wf.induction generalizing acc with
  | h ds ih =>
    rw [chunks]
    by_cases hl : ds.length ≤ 8
    · simp only [hl, dif_pos, List.singleton_append, tokInts]
    · simp only [hl, dif_neg, not_false_eq_true, List.cons_append, tokInts]
      have hne : ds.drop 8 ≠ [] := by
        intro e; have := congrArg List.length e; simp at this; omega
      rw [ih (ds.drop 8) (show (ds.drop 8).length < ds.length by simp; omega) hne,
          List.append_assoc, List.take_append_drop]

theorem tokInts_toks (ns : List Nat) : tokInts [] (toks ns) = some ns := by
  induction ns with
  | nil => rfl
  | cons n ns ih =>
    simp only [toks, List.flatMap_cons] at ih ⊢
    rw [intToks, tokInts_chunks _ (digits_ne_nil n), ih, List.nil_append, ofD_digits]
    rfl

theorem chunks_ok (ds : List Nat) (h : ds ≠ []) (hd : ∀ x ∈ ds, x < 10) :
    ∀ t ∈ chunks ds, tokOK t := by
  induction ds using (measure List.length).wf.induction with
  | h ds ih =>
    rw [chunks]
    have hp := List.length_pos.2 h
    by_cases hl : ds.length ≤ 8
    · simp only [hl, dif_pos, List.mem_singleton]
      rintro t rfl; exact ⟨by omega, hl, hd⟩
    · simp only [hl, dif_neg, not_false_eq_true, List.mem_cons]
      rintro t (rfl | ht)
      · exact ⟨by simp; omega, fun x hx => hd x (List.mem_of_mem_take hx)⟩
      · have hne : ds.drop 8 ≠ [] := by
          intro e; have := congrArg List.length e; simp at this; omega
        exact ih (ds.drop 8) (show (ds.drop 8).length < ds.length by simp; omega) hne
          (fun x hx => hd x (List.mem_of_mem_drop hx)) t ht

theorem toks_ok (ns : List Nat) : ∀ t ∈ toks ns, tokOK t := by
```

```
  intro t ht
  simp only [toks, List.mem_flatMap] at ht
  obtain ⟨n, -, ht⟩ := ht
  exact chunks_ok _ (digits_ne_nil n) (digits_lt n) t ht

/-! ## Run compression -/

/-- A unit of the digit stream: a plain token, or a run `0 r d₁ … d_r` of `r` one-digit
    final tokens. -/
inductive U where
  | tok (t : Tok)
  | run (ds : List Nat)
  deriving DecidableEq, Repr

def uOK : U → Prop
  | .tok t => tokOK t
  | .run ds => 1 ≤ ds.length ∧ ds.length ≤ 9 ∧ ∀ x ∈ ds, x < 10

def uDigits : U → List Nat
  | .tok (.cont ds) => 9 :: ds
  | .tok (.fin ds) => ds.length :: ds
  | .run ds => 0 :: ds.length :: ds

def single (d : Nat) : Tok := .fin [d]

def expandU : U → List Tok
  | .tok t => [t]
  | .run ds => ds.map single

def expand (us : List U) : List Tok := us.flatMap expandU

def flush (acc : List Nat) : List U :=
  if 3 ≤ acc.length then [.run acc] else acc.map (fun d => .tok (single d))

def isSingle : Tok → Option Nat
  | .fin [d] => some d
  | _ => none

theorem isSingle_some {t : Tok} {d : Nat} (h : isSingle t = some d) : t = single d := by
  unfold isSingle at h; split at h <;> simp_all [single]

/-- Greedy compression: maximal runs of one-digit final tokens (whatever integer they end,
    so a one-digit tail of a long integer may join a run), closed at length 9; runs shorter
    than 3 stay plain. -/
def comp : List Nat → List Tok → List U
  | acc, [] => flush acc
  | acc, t :: ts =>
    match isSingle t with
    | some d => if acc.length + 1 = 9 then .run (acc ++ [d]) :: comp [] ts
                else comp (acc ++ [d]) ts
    | none => flush acc ++ .tok t :: comp [] ts

def compress (ts : List Tok) : List U := comp [] ts

theorem flatMap_single {α β : Type} (f : α → β) (l : List α) :
    (l.flatMap fun a => [f a]) = l.map f := by
  induction l with
  | nil => rfl
  | cons a l ih => simp [ih]

theorem expand_flush (acc : List Nat) : expand (flush acc) = acc.map single := by
  unfold flush; split
  · simp [expand, expandU]
  · simp only [expand, List.flatMap_map, expandU]; exact flatMap_single _ _

theorem expand_comp (acc : List Nat) (ts : List Tok) :
    expand (comp acc ts) = acc.map single ++ ts := by
  induction ts generalizing acc with
  | nil => simp [comp, expand_flush]
  | cons t ts ih =>
    unfold comp
    split
    · next d hd =>
      have := isSingle_some hd; subst this
```

```
      split
      · simp [expand, expandU] at ih ⊢; rw [ih]; simp
      · rw [ih]; simp
    · simp only [expand, List.flatMap_append, List.flatMap_cons, expandU] at ih ⊢
      have ef := expand_flush acc
      unfold expand at ef
      rw [ih, ef]
      simp


theorem flush_ok (acc : List Nat) (h9 : acc.length < 9) (hd : ∀ x ∈ acc, x < 10) :
    ∀ u ∈ flush acc, uOK u := by
  unfold flush
  by_cases h : 3 ≤ acc.length
  · simp only [h, if_true, List.mem_singleton]; intro u hu; subst hu; exact ⟨by omega, by omega, hd⟩
  · simp only [h, if_false, List.mem_map]
    rintro u ⟨d, hd', rfl⟩
    exact ⟨by simp, by simp, by simp [hd d hd']⟩

theorem comp_ok (acc : List Nat) (ts : List Tok) (h9 : acc.length < 9)
    (hd : ∀ x ∈ acc, x < 10) (ht : ∀ t ∈ ts, tokOK t) : ∀ u ∈ comp acc ts, uOK u := by
  induction ts generalizing acc with
  | nil => exact flush_ok acc h9 hd
  | cons t ts ih =>
    have ht' : ∀ t ∈ ts, tokOK t := fun t' h => ht t' (by simp [h])
    unfold comp
    split
    · next d hs =>
      have e := isSingle_some hs; subst e
      have hd10 : d < 10 := by
        have := ht (single d) (by simp); simp [tokOK, single] at this; exact this
      have hd' : ∀ x ∈ acc ++ [d], x < 10 := by
        intro x hx; simp at hx; rcases hx with hx | rfl; exact hd x hx; exact hd10
      split
      · next h =>
        simp only [List.mem_cons]
        rintro u (rfl | hu)
        · exact ⟨by simp, by simp; omega, hd'⟩
        · exact ih [] (by simp) (by simp) ht' u hu
      · next h => exact ih _ (by simp; omega) hd' ht'
    · intro u hu
      rw [List.mem_append, List.mem_cons] at hu
      rcases hu with hu | hu | hu
      · exact flush_ok acc h9 hd u hu
      · subst hu; exact ht t (by simp)
      · exact ih [] (by simp) (by simp) ht' u hu

theorem compress_ok (ts : List Tok) (ht : ∀ t ∈ ts, tokOK t) : ∀ u ∈ compress ts, uOK u :=
  comp_ok [] ts (by simp) (by simp) ht

theorem expand_compress (ts : List Tok) : expand (compress ts) = ts := by
  simp [compress, expand_comp]

/-! ## Parsing the digit stream back into units -/

def takeN (n : Nat) (l : List Nat) : Option (List Nat × List Nat) :=
  if n ≤ l.length then some (l.take n, l.drop n) else none

theorem takeN_append (ds rest : List Nat) : takeN ds.length (ds ++ rest) = some (ds, rest) := by
  simp [takeN]

def parseU : List Nat → Option (U × List Nat)
  | [] => none
  | t :: rest =>
    if t = 0 then
      match rest with
      | r :: rest' => (takeN r rest').map fun x => (.run x.1, x.2)
      | [] => none
    else if t = 9 then (takeN 8 rest).map fun x => (.tok (.cont x.1), x.2)
    else if t ≤ 8 then (takeN t rest).map fun x => (.tok (.fin x.1), x.2)
    else none

theorem parseU_digits (u : U) (hu : uOK u) (rest : List Nat) :
    parseU (uDigits u ++ rest) = some (u, rest) := by
```

```
  match u, hu with
  | .run ds, ⟨h1, h2, _⟩ => simp [uDigits, parseU, takeN_append]
  | .tok (.cont ds), ⟨h1, _⟩ =>
    simp only [uDigits, List.cons_append, parseU]
    rw [← h1, takeN_append]; simp
  | .tok (.fin ds), ⟨h1, h2, _⟩ =>
    simp only [uDigits, List.cons_append, parseU, takeN_append]
    have a : ds.length ≠ 0 := by omega
    have b : ds.length ≠ 9 := by omega
    simp [a, b, h2]

def parseUs : Nat → List Nat → Option (List U)
  | _, [] => some []
  | 0, _ :: _ => none
  | n + 1, l@(_ :: _) =>
    match parseU l with
    | some (u, r) => (parseUs n r).map (u :: ·)
    | none => none

theorem uDigits_ne (u : U) : uDigits u ≠ [] := by
  match u with
  | .run _ => simp [uDigits]
  | .tok (.cont _) => simp [uDigits]
  | .tok (.fin _) => simp [uDigits]

theorem parseUs_digits (us : List U) (hu : ∀ u ∈ us, uOK u) (n : Nat) (hn : us.length ≤ n) :
    parseUs n (us.flatMap uDigits) = some us := by
  induction us generalizing n with
  | nil => simp [parseUs]
  | cons u us ih =>
    cases n with
    | zero => simp at hn
    | succ n =>
      simp only [List.flatMap_cons]
      have hne := uDigits_ne u
      obtain ⟨a, as, ha⟩ : ∃ a as, uDigits u = a :: as := by
        cases h : uDigits u with
        | nil => exact absurd h hne
        | cons a as => exact ⟨a, as, rfl⟩
      have hp := parseU_digits u (hu u (by simp)) (us.flatMap uDigits)
      rw [ha] at hp ⊢
      simp only [List.cons_append, parseUs]
      simp only [List.cons_append] at hp
      rw [hp]
      simp only
      rw [ih (fun u h => hu u (by simp [h])) n (by simp at hn; omega)]
      rfl

theorem length_le_flatMap (us : List U) : us.length ≤ (us.flatMap uDigits).length := by
  induction us with
  | nil => simp
  | cons u us ih =>
    have := List.length_pos.2 (uDigits_ne u)
    simp only [List.flatMap_cons, List.length_append, List.length_cons]; omega

/-- Token-level decoder of a digit stream: parse units, then expand runs. -/
def decodeTok (s : List Nat) : Option (List Tok) := (parseUs s.length s).map expand

/-- General unit round trip: any valid unit sequence (in particular any way of splitting
    runs) decodes to its expansion. -/
theorem decodeTok_units (us : List U) (hu : ∀ u ∈ us, uOK u) :
    decodeTok (us.flatMap uDigits) = some (expand us) := by
  unfold decodeTok
  rw [parseUs_digits us hu _ (length_le_flatMap us)]; rfl

/-- The compressed digit stream of a token list. -/
def stream (ts : List Tok) : List Nat := (compress ts).flatMap uDigits

/-- Run-compression round trip. -/
theorem decodeTok_stream (ts : List Tok) (ht : ∀ t ∈ ts, tokOK t) :
    decodeTok (stream ts) = some ts := by
  unfold stream; rw [decodeTok_units _ (compress_ok ts ht), expand_compress]

theorem uDigits_lt (u : U) (hu : uOK u) : ∀ x ∈ uDigits u, x < 10 := by
```

```
  match u, hu with
  | .run ds, ⟨_, h2, h3⟩ =>
    simp only [uDigits, List.mem_cons]; rintro x (rfl | rfl | hx) <;> first | omega | exact h3 x hx
  | .tok (.cont ds), ⟨_, h3⟩ =>
    simp only [uDigits, List.mem_cons]; rintro x (rfl | hx) <;> first | omega | exact h3 x hx
  | .tok (.fin ds), ⟨_, h2, h3⟩ =>
    simp only [uDigits, List.mem_cons]; rintro x (rfl | hx) <;> first | omega | exact h3 x hx

theorem stream_lt (ts : List Tok) (ht : ∀ t ∈ ts, tokOK t) : ∀ x ∈ stream ts, x < 10 := by
  intro x hx
  simp only [stream, List.mem_flatMap] at hx
  obtain ⟨u, hu, hx⟩ := hx
  exact uDigits_lt u (compress_ok ts ht u hu) x hx


/-! ## JSON values, leaf rows, flattening -/

/-- JSON values. Numbers are decimal triples: `num neg d m` denotes `(-1)^neg · m / 10^d`.
    Strings and keys are sequences of code units. Objects are association lists. -/
inductive J where
  | null
  | bool (b : Bool)
  | num (neg : Bool) (d m : Nat)
  | str (s : List Nat)
  | arr (xs : List J)
  | obj (kvs : List (List Nat × J))

inductive Comp where
  | key (k : List Nat)
  | idx (i : Nat)
  deriving DecidableEq, Repr

/-- Leaves: primitives and the two empty-container markers. -/
inductive Leaf where
  | null
  | bool (b : Bool)
  | num (neg : Bool) (d m : Nat)
  | str (s : List Nat)
  | earr
  | eobj
  deriving DecidableEq, Repr

abbrev Row := List Comp × Leaf

def pre (c : Comp) (r : Row) : Row := (c :: r.1, r.2)

mutual
def flatten : J → List Row
  | .null => [([], .null)]
  | .bool b => [([], .bool b)]
  | .num n d m => [([], .num n d m)]
  | .str s => [([], .str s)]
  | .arr [] => [([], .earr)]
  | .arr (x :: xs) => flatA 0 (x :: xs)
  | .obj [] => [([], .eobj)]
  | .obj (kv :: kvs) => flatO (kv :: kvs)
def flatA : Nat → List J → List Row
  | _, [] => []
  | i, x :: xs => (flatten x).map (pre (.idx i)) ++ flatA (i + 1) xs
def flatO : List (List Nat × J) → List Row
  | [] => []
  | (k, x) :: kvs => (flatten x).map (pre (.key k)) ++ flatO kvs
end


/-! ## The domain: canonical objects (keys strictly ascending in shortlex order) -/

/-- Lexicographic strict order on equal-length code-unit sequences. -/
def lexLt : List Nat → List Nat → Bool
  | a :: as, b :: bs => a < b || (a == b && lexLt as bs)
  | _, _ => false

/-- Shortlex: shorter first, then lexicographic. -/
def slt (a b : List Nat) : Bool := a.length < b.length || (a.length == b.length && lexLt a b)
```

```
theorem lexLt_irrefl (a : List Nat) : lexLt a a = false := by
  induction a with
  | nil => rfl
  | cons x as ih => simp [lexLt, ih]

theorem slt_irrefl (a : List Nat) : slt a a = false := by simp [slt, lexLt_irrefl]

theorem slt_ne {a b : List Nat} (h : slt a b = true) : a ≠ b := by
  intro e; subst e; rw [slt_irrefl] at h; contradiction

def keysAsc : List (List Nat) → Prop
  | a :: b :: r => slt a b = true ∧ keysAsc (b :: r)
  | _ => True

/-- Shortest decimal form: no trailing zero after the point; zero is `+0` with `d = 0`. -/
def numOK (neg : Bool) (d m : Nat) : Prop := (0 < d → m % 10 ≠ 0) ∧ (m = 0 → neg = false ∧ d = 0)

mutual
def WF : J → Prop
  | .num neg d m => numOK neg d m
  | .arr xs => WFA xs
  | .obj kvs => keysAsc (kvs.map Prod.fst) ∧ WFO kvs
  | _ => True
def WFA : List J → Prop
  | [] => True
  | x :: xs => WF x ∧ WFA xs
def WFO : List (List Nat × J) → Prop
  | [] => True
  | (_, x) :: kvs => WF x ∧ WFO kvs
end

/-! ## Unflattening: the left inverse of `flatten` -/

/-- Take the maximal prefix of rows whose path starts with `c`, stripping `c`. -/
def strip (c : Comp) : List Row → List Row × List Row
  | (c' :: p, l) :: rs =>
    if c' = c then ((p, l) :: (strip c rs).1, (strip c rs).2) else ([], (c' :: p, l) :: rs)
  | rs => ([], rs)

/-- Group consecutive rows by their first path component. -/
def group : Nat → List Row → Option (List (Comp × List Row))
  | _, [] => some []
  | 0, _ :: _ => none
  | n + 1, (c :: p, l) :: rs => (group n (strip c rs).2).map ((c, (p, l) :: (strip c rs).1) :: ·)
  | _ + 1, ([], _) :: _ => none

def isLeaf : List Row → Option Leaf
  | [([], l)] => some l
  | _ => none

def ofLeaf : Leaf → J
  | .null => .null
  | .bool b => .bool b
  | .num n d m => .num n d m
  | .str s => .str s
  | .earr => .arr []
  | .eobj => .obj []

def optMap {α β : Type} (f : α → Option β) : List α → Option (List β)
  | [] => some []
  | a :: as => match f a with
    | some b => (optMap f as).map (b :: ·)
    | none => none

def unKey (f : List Row → Option J) (g : Comp × List Row) : Option (List Nat × J) :=
  match g.1 with
  | .key k => (f g.2).map (k, ·)
  | .idx _ => none

def unflat : Nat → List Row → Option J
  | 0, _ => none
  | n + 1, rows =>
    match isLeaf rows with
```

```
    | some l => some (ofLeaf l)
    | none =>
      match group rows.length rows with
      | some ((.idx i, r) :: gs) => (optMap (fun g => unflat n g.2) ((.idx i, r) :: gs)).map .arr
      | some ((.key k, r) :: gs) => (optMap (unKey (unflat n)) ((.key k, r) :: gs)).map .obj
      | _ => none

def maxLen (rows : List Row) : Nat := rows.foldr (fun r m => max r.1.length m) 0

def unflatten (rows : List Row) : Option J := unflat (maxLen rows + 1) rows

/-! ### Lemmas -/

theorem maxLen_append (a b : List Row) : maxLen (a ++ b) = max (maxLen a) (maxLen b) := by
  induction a with
  | nil => simp [maxLen]
  | cons r a ih => simp only [maxLen, List.cons_append, List.foldr_cons] at ih ⊢; rw [ih]; omega

theorem maxLen_pre (c : Comp) (rows : List Row) (h : rows ≠ []) :
    maxLen (rows.map (pre c)) = maxLen rows + 1 := by
  induction rows with
  | nil => exact absurd rfl h
  | cons r rs ih =>
    cases rs with
    | nil => simp [maxLen, pre]
    | cons r' rs' =>
      have := ih (by simp)
      simp only [maxLen, List.map_cons, List.foldr_cons] at this ⊢
      rw [this]; simp [pre]; omega

def headNot (c : Comp) : List Row → Prop
  | (c' :: _, _) :: _ => c' ≠ c
  | _ => True

theorem strip_pre (c : Comp) (rows rest : List Row) (h : headNot c rest) :
    strip c (rows.map (pre c) ++ rest) = (rows, rest) := by
  induction rows with
  | nil =>
    match rest, h with
    | [], _ => rfl
    | ([], l) :: rs, _ => rfl
    | (c' :: p, l) :: rs, h => simp only [headNot] at h; simp [strip, h]
  | cons r rows ih =>
    simp only [List.map_cons, List.cons_append, pre, strip, if_true]
    simp only [pre] at ih
    rw [ih]

mutual
theorem flatten_ne_nil : (v : J) → flatten v ≠ []
  | .null | .bool _ | .num _ _ _ | .str _ | .arr [] | .obj [] => by simp [flatten]
  | .arr (x :: xs) => by
    simp only [flatten, flatA]
    have := flatten_ne_nil x
    intro h; simp at h; exact this h.1
  | .obj ((k, x) :: kvs) => by
    simp only [flatten, flatO]
    have := flatten_ne_nil x
    intro h; simp at h; exact this h.1
end

def gA : Nat → List J → List (Comp × List Row)
  | _, [] => []
  | i, x :: xs => (.idx i, flatten x) :: gA (i + 1) xs

def gO : List (List Nat × J) → List (Comp × List Row)
  | [] => []
  | (k, x) :: kvs => (.key k, flatten x) :: gO kvs

theorem pre_cons_map (c : Comp) (rows : List Row) (h : rows ≠ []) :
    ∃ p l rs, rows.map (pre c) = (c :: p, l) :: rs.map (pre c) ∧ rows = (p, l) :: rs := by
  match rows, h with
  | (p, l) :: rs, _ => exact ⟨p, l, rs, rfl, rfl⟩

theorem headNot_flatA (i : Nat) (xs : List J) : headNot (.idx i) (flatA (i + 1) xs) := by
```

```
  match xs with
  | [] => simp [flatA, headNot]
  | x :: xs =>
    obtain ⟨p, l, rs, e, -⟩ := pre_cons_map (.idx (i + 1)) (flatten x) (flatten_ne_nil x)
    simp only [flatA, e, List.cons_append, headNot]
    simp

theorem group_flatA (xs : List J) (i n : Nat) (hn : xs.length ≤ n) :
    group n (flatA i xs) = some (gA i xs) := by
  induction xs generalizing i n with
  | nil => cases n <;> rfl
  | cons x xs ih =>
    cases n with
    | zero => simp at hn
    | succ n =>
      obtain ⟨p, l, rs, e, e2⟩ := pre_cons_map (.idx i) (flatten x) (flatten_ne_nil x)
      simp only [flatA, e, List.cons_append, group]
      rw [strip_pre _ _ _ (headNot_flatA i xs)]
      simp only
      rw [ih (i + 1) n (by simp at hn; omega)]
      simp [gA, e2]

theorem headNot_flatO (k : List Nat) (kvs : List (List Nat × J))
    (h : keysAsc (k :: kvs.map Prod.fst)) : headNot (.key k) (flatO kvs) := by
  match kvs, h with
  | [], _ => simp [flatO, headNot]
  | (k', x) :: kvs, h =>
    obtain ⟨p, l, rs, e, -⟩ := pre_cons_map (.key k') (flatten x) (flatten_ne_nil x)
    simp only [flatO, e, List.cons_append, headNot]
    simp only [List.map_cons, keysAsc] at h
    have := slt_ne h.1
    intro e'; injection e' with e'; exact this e'.symm

theorem keysAsc_tail (k : List Nat) (ks : List (List Nat)) (h : keysAsc (k :: ks)) :
    keysAsc ks := by
  match ks, h with
  | [], _ => trivial
  | _ :: _, h => exact h.2

theorem group_flatO (kvs : List (List Nat × J)) (n : Nat) (hn : kvs.length ≤ n)
    (hk : keysAsc (kvs.map Prod.fst)) : group n (flatO kvs) = some (gO kvs) := by
  induction kvs generalizing n with
  | nil => cases n <;> rfl
  | cons kx kvs ih =>
    obtain ⟨k, x⟩ := kx
    cases n with
    | zero => simp at hn
    | succ n =>
      obtain ⟨p, l, rs, e, e2⟩ := pre_cons_map (.key k) (flatten x) (flatten_ne_nil x)
      simp only [flatO, e, List.cons_append, group]
      rw [strip_pre _ _ _ (headNot_flatO k kvs hk)]
      simp only
      rw [ih n (by simp at hn; omega) (keysAsc_tail _ _ hk)]
      simp [gO, e2]

theorem length_flatA (xs : List J) (i : Nat) : xs.length ≤ (flatA i xs).length := by
  induction xs generalizing i with
  | nil => simp [flatA]
  | cons x xs ih =>
    have := List.length_pos.2 (flatten_ne_nil x)
    have := ih (i + 1)
    simp only [flatA, List.length_append, List.length_map, List.length_cons]; omega

theorem length_flatO (kvs : List (List Nat × J)) : kvs.length ≤ (flatO kvs).length := by
  induction kvs with
  | nil => simp [flatO]
  | cons kx kvs ih =>
    obtain ⟨k, x⟩ := kx
    have := List.length_pos.2 (flatten_ne_nil x)
    simp only [flatO, List.length_append, List.length_map, List.length_cons]; omega

theorem isLeaf_pre (c : Comp) (rows rest : List Row) (h : rows ≠ []) :
    isLeaf (rows.map (pre c) ++ rest) = none := by
  obtain ⟨p, l, rs, e, -⟩ := pre_cons_map c rows h
```

```
  rw [e]
  cases rs <;> cases rest <;> rfl

mutual
theorem unflat_flatten : (v : J) → WF v → ∀ n, maxLen (flatten v) < n →
    unflat n (flatten v) = some v
  | .null, _, n + 1, _ => rfl
  | .bool _, _, n + 1, _ => rfl
  | .num _ _ _, _, n + 1, _ => rfl
  | .str _, _, n + 1, _ => rfl
  | .arr [], _, n + 1, _ => rfl
  | .obj [], _, n + 1, _ => rfl
  | .arr (x :: xs), hw, n + 1, hn => by
    have hl := List.length_pos.2 (flatten_ne_nil x)
    simp only [flatten] at hn ⊢
    simp only [unflat]
    rw [show flatA 0 (x :: xs) = (flatten x).map (pre (.idx 0)) ++ flatA 1 xs from rfl,
        isLeaf_pre _ _ _ (flatten_ne_nil x)]
    simp only
    rw [← show flatA 0 (x :: xs) = (flatten x).map (pre (.idx 0)) ++ flatA 1 xs from rfl,
        group_flatA _ _ _ (length_flatA _ _)]
    simp only [gA]
    rw [← show gA 0 (x :: xs) = (.idx 0, flatten x) :: gA 1 xs from rfl,
        unflat_gA (x :: xs) 0 hw n (by omega)]
    rfl
  | .obj ((k, x) :: kvs), hw, n + 1, hn => by
    simp only [flatten] at hn ⊢
    simp only [unflat]
    rw [show flatO ((k, x) :: kvs) = (flatten x).map (pre (.key k)) ++ flatO kvs from rfl,
        isLeaf_pre _ _ _ (flatten_ne_nil x)]
    simp only
    rw [← show flatO ((k, x) :: kvs) = (flatten x).map (pre (.key k)) ++ flatO kvs from rfl,
        group_flatO _ _ (length_flatO _) hw.1]
    simp only [gO]
    rw [← show gO ((k, x) :: kvs) = (.key k, flatten x) :: gO kvs from rfl,
        unflat_gO ((k, x) :: kvs) hw.2 n (by omega)]
    rfl
theorem unflat_gA : (xs : List J) → (i : Nat) → WFA xs → ∀ n, maxLen (flatA i xs) ≤ n →
    optMap (fun g => unflat n g.2) (gA i xs) = some xs
  | [], _, _, _, _ => rfl
  | x :: xs, i, hw, n, hn => by
    simp only [flatA, maxLen_append, maxLen_pre _ _ (flatten_ne_nil x)] at hn
    simp only [gA, optMap]
    rw [unflat_flatten x hw.1 n (by omega), unflat_gA xs (i + 1) hw.2 n (by omega)]
    rfl
theorem unflat_gO : (kvs : List (List Nat × J)) → WFO kvs → ∀ n, maxLen (flatO kvs) ≤ n →
    optMap (unKey (unflat n)) (gO kvs) = some kvs
  | [], _, _, _ => rfl
  | (k, x) :: kvs, hw, n, hn => by
    simp only [flatO, maxLen_append, maxLen_pre _ _ (flatten_ne_nil x)] at hn
    simp only [gO, optMap, unKey]
    rw [unflat_flatten x hw.1 n (by omega), unflat_gO kvs hw.2 n (by omega)]
    rfl
end

/-- Stage 1 round trip: `unflatten` is a left inverse of `flatten` on the domain. -/
theorem unflatten_flatten (v : J) (hv : WF v) : unflatten (flatten v) = some v :=
  unflat_flatten v hv _ (by omega)


/-! ## Type-tagged integer coding of rows -/

/-- Path components: non-empty key `[ℓ, c₁ … c_ℓ]`, empty key `[0, 1]`, index `[0, 0, i]`. -/
def compCode : Comp → List Nat
  | .key [] => [0, 1]
  | .key (c :: cs) => (cs.length + 1) :: c :: cs
  | .idx i => [0, 0, i]

/-- Leaf tags: null 0, boolean 1, number 2, string 3, empty array 4, empty object 5. -/
def leafCode : Leaf → List Nat
  | .null => [0]
  | .bool b => [1, if b then 1 else 0]
  | .num neg d m => [2, if neg then 0 else 1, d, m]
  | .str s => 3 :: s.length :: s
```

```
  | .earr => [4]
  | .eobj => [5]

/-- A row: path length, components, leaf. -/
def rowCode (r : Row) : List Nat := r.1.length :: (r.1.flatMap compCode ++ leafCode r.2)

def ints (rows : List Row) : List Nat := rows.flatMap rowCode

def parseComp : List Nat → Option (Comp × List Nat)
  | [] => none
  | l :: rest =>
    if l = 0 then
      match rest with
      | 1 :: r => some (.key [], r)
      | 0 :: i :: r => some (.idx i, r)
      | _ => none
    else (takeN l rest).map fun x => (.key x.1, x.2)

def parseComps : Nat → List Nat → Option (List Comp × List Nat)
  | 0, l => some ([], l)
  | n + 1, l =>
    match parseComp l with
    | some (c, r) => (parseComps n r).map fun x => (c :: x.1, x.2)
    | none => none

def parseLeaf : List Nat → Option (Leaf × List Nat)
  | [] => none
  | t :: rest =>
    if t = 0 then some (.null, rest)
    else if t = 1 then
      match rest with
      | b :: r => some (.bool (b != 0), r)
      | [] => none
    else if t = 2 then
      match rest with
      | s :: d :: m :: r => some (.num (s == 0) d m, r)
      | _ => none
    else if t = 3 then
      match rest with
      | l :: r => (takeN l r).map fun x => (.str x.1, x.2)
      | [] => none
    else if t = 4 then some (.earr, rest)
    else if t = 5 then some (.eobj, rest)
    else none

def parseRow : List Nat → Option (Row × List Nat)
  | [] => none
  | n :: rest =>
    match parseComps n rest with
    | some (p, r) =>
      match parseLeaf r with
      | some (l, r') => some ((p, l), r')
      | none => none
    | none => none

def parseRows : Nat → List Nat → Option (List Row)
  | _, [] => some []
  | 0, _ :: _ => none
  | n + 1, l@(_ :: _) =>
    match parseRow l with
    | some (r, rest) => (parseRows n rest).map (r :: ·)
    | none => none

theorem parseComp_code (c : Comp) (rest : List Nat) :
    parseComp (compCode c ++ rest) = some (c, rest) := by
  match c with
  | .key [] => rfl
  | .idx i => rfl
  | .key (a :: as) =>
    simp only [compCode, List.cons_append, parseComp]
    have := takeN_append (a :: as) rest
    simp only [List.length_cons, List.cons_append] at this
    simp [this]
```

```
theorem parseComps_code (cs : List Comp) (rest : List Nat) :
    parseComps cs.length (cs.flatMap compCode ++ rest) = some (cs, rest) := by
  induction cs with
  | nil => rfl
  | cons c cs ih =>
    simp only [List.length_cons, List.flatMap_cons, List.append_assoc, parseComps,
      parseComp_code, ih]
    rfl

theorem parseLeaf_code (l : Leaf) (rest : List Nat) :
    parseLeaf (leafCode l ++ rest) = some (l, rest) := by
  match l with
  | .null => rfl
  | .bool b => cases b <;> rfl
  | .num neg d m => cases neg <;> rfl
  | .str s =>
    simp only [leafCode, List.cons_append, parseLeaf, takeN_append]
    simp
  | .earr => rfl
  | .eobj => rfl

theorem parseRow_code (r : Row) (rest : List Nat) :
    parseRow (rowCode r ++ rest) = some (r, rest) := by
  obtain ⟨p, l⟩ := r
  simp only [rowCode, List.cons_append, parseRow, List.append_assoc, parseComps_code,
    parseLeaf_code]

theorem parseRows_ints (rows : List Row) (n : Nat) (hn : rows.length ≤ n) :
    parseRows n (ints rows) = some rows := by
  induction rows generalizing n with
  | nil => cases n <;> rfl
  | cons r rows ih =>
    cases n with
    | zero => simp at hn
    | succ n =>
      have hp := parseRow_code r (ints rows)
      simp only [ints, List.flatMap_cons] at hp ⊢
      simp only [rowCode, List.cons_append] at hp ⊢
      simp only [parseRows, hp]
      have := ih n (by simp at hn; omega)
      simp only [ints] at this
      rw [this]
      rfl

theorem length_ints (rows : List Row) : rows.length ≤ (ints rows).length := by
  induction rows with
  | nil => simp [ints]
  | cons r rows ih =>
    simp only [ints, List.flatMap_cons, List.length_append, List.length_cons] at ih ⊢
    simp only [rowCode, List.length_cons]; omega

/-- Stage 2 round trip. -/
theorem parseRows_ints' (rows : List Row) :
    parseRows (ints rows).length (ints rows) = some rows :=
  parseRows_ints rows _ (length_ints rows)

/-! ## The whole encoder and decoder -/

def encode (v : J) : List Nat := pack (stream (toks (ints (flatten v))))

def decode (es : List Nat) : Option J :=
  match decodeTok (unpack es) with
  | none => none
  | some ts =>
    match tokInts [] ts with
    | none => none
    | some ns =>
      match parseRows ns.length ns with
      | none => none
      | some rows => unflatten rows

/-- MAIN THEOREM (round trip): on the domain, `decode` inverts `encode`,
    also after padding with any number of zero elements. -/
theorem decode_encode (v : J) (hv : WF v) (z : Nat) :
```

```
    decode (encode v ++ List.replicate z 0) = some v := by
  have hok := toks_ok (ints (flatten v))
  unfold decode encode
  rw [unpack_pack _ (stream_lt _ hok), decodeTok_stream _ hok]
  simp only
  rw [tokInts_toks]
  simp only
  rw [parseRows_ints']
  exact unflatten_flatten v hv

/-- MAIN THEOREM (injectivity on the domain). -/
theorem encode_injective (v w : J) (hv : WF v) (hw : WF w) (h : encode v = encode w) :
    v = w := by
  have a := decode_encode v hv 0
  have b := decode_encode w hw 0
  rw [h, b] at a
  exact Option.some.inj a.symm

/-- Injectivity after zero padding, with independent padding lengths. -/
theorem encode_injective_pad (v w : J) (hv : WF v) (hw : WF w) (z z' : Nat)
    (h : encode v ++ List.replicate z 0 = encode w ++ List.replicate z' 0) : v = w := by
  have a := decode_encode v hv z
  have b := decode_encode w hw z'
  rw [h, b] at a
  exact Option.some.inj a.symm

/-- Every element is the guard digit followed by ≤ 75 digits: `1 ≤ e < 10^76 < p`. -/
theorem encode_lt_p (v : J) : ∀ e ∈ encode v, 1 ≤ e ∧ e < 10 ^ 76 ∧ e < p := by
  intro e he
  have := pack_lt_p _ (stream_lt _ (toks_ok (ints (flatten v)))) e he
  simpa using this

/-- Capacity: a stream of at most `75·N` digits fits in `N` elements. -/
theorem encode_length (v : J) (N : Nat) (h : (stream (toks (ints (flatten v)))).length ≤ W * N) :
    (encode v).length ≤ N := by
  unfold encode pack; rw [List.length_map]; exact blocks_length _ N h


/-! ## Decimal numbers: canonical triples are unique -/

/-- `num n₁ d₁ m₁` and `num n₂ d₂ m₂` denote the same rational number. -/
def numEq (n₁ : Bool) (d₁ m₁ : Nat) (n₂ : Bool) (d₂ m₂ : Nat) : Prop :=
  m₁ * 10 ^ d₂ = m₂ * 10 ^ d₁ ∧ (m₁ = 0 ∨ n₁ = n₂)

theorem num_canon_aux (d₁ d₂ m₁ m₂ : Nat) (hle : d₁ ≤ d₂) (h2 : 0 < d₂ → m₂ % 10 ≠ 0)
    (he : m₁ * 10 ^ d₂ = m₂ * 10 ^ d₁) : d₁ = d₂ ∧ m₁ = m₂ := by
  obtain ⟨k, rfl⟩ : ∃ k, d₂ = d₁ + k := ⟨d₂ - d₁, by omega⟩
  have hp : 0 < 10 ^ d₁ := Nat.pos_pow_of_pos _ (by decide)
  rw [Nat.pow_add, ← Nat.mul_assoc, Nat.mul_comm (m₁ * 10 ^ d₁), ← Nat.mul_assoc] at he
  have he' : 10 ^ k * m₁ = m₂ := Nat.eq_of_mul_eq_mul_right hp (by rw [he, Nat.mul_comm])
  cases k with
  | zero => simp at he'; exact ⟨by omega, he'⟩
  | succ k =>
    exfalso
    apply h2 (by omega)
    rw [← he', Nat.pow_succ, Nat.mul_comm (10 ^ k) 10, Nat.mul_assoc]
    exact Nat.mul_mod_right 10 _

/-- Canonical (shortest) decimal triples are in bijection with the decimal numbers they denote. -/
theorem num_canon_unique (n₁ : Bool) (d₁ m₁ : Nat) (n₂ : Bool) (d₂ m₂ : Nat)
    (h₁ : numOK n₁ d₁ m₁) (h₂ : numOK n₂ d₂ m₂) (he : numEq n₁ d₁ m₁ n₂ d₂ m₂) :
    n₁ = n₂ ∧ d₁ = d₂ ∧ m₁ = m₂ := by
  obtain ⟨he, hs⟩ := he
  by_cases z : m₁ = 0
  · subst z
    have hp : 0 < 10 ^ d₁ := Nat.pos_pow_of_pos _ (by decide)
    have : m₂ = 0 := by
      rw [Nat.zero_mul] at he
      exact (Nat.mul_eq_zero.1 he.symm).resolve_right (by omega)
    subst this
    have a := h₁.2 rfl; have b := h₂.2 rfl
    exact ⟨by rw [a.1, b.1], by rw [a.2, b.2], rfl⟩
  · have hs := hs.resolve_left z
    refine ⟨hs, ?_⟩
```

```
    by_cases hle : d₁ ≤ d₂
    · exact num_canon_aux d₁ d₂ m₁ m₂ hle h₂.1 he
    · have := num_canon_aux d₂ d₁ m₂ m₁ (by omega) h₁.1 he.symm
      exact ⟨this.1.symm, this.2.symm⟩

/-! ## Document ids to leaf indices: guarded radix map -/

/-- Index of an id over a `b`-symbol alphabet: a leading guard `1`, then one base-`b`
    digit per symbol. -/
def idIndex (b : Nat) (s : List Nat) : Nat := s.foldl (fun a c => a * b + c) 1

def unIndex (b : Nat) (hb : 2 ≤ b) (n : Nat) : List Nat :=
  if h : n < b then [] else unIndex b hb (n / b) ++ [n % b]
termination_by n
decreasing_by exact Nat.div_lt_self (by omega) (by omega)

theorem idIndex_snoc (b : Nat) (s : List Nat) (c : Nat) :
    idIndex b (s ++ [c]) = idIndex b s * b + c := by
  simp [idIndex, List.foldl_append]

theorem idIndex_pos (b : Nat) (hb : 1 ≤ b) (s : List Nat) : 1 ≤ idIndex b s := by
  refine snocInd (P := fun s => 1 ≤ idIndex b s) (by simp [idIndex]) ?_ s
  intro s c ih
  rw [idIndex_snoc]
  have := Nat.le_mul_of_pos_right (idIndex b s) hb; omega

/-- Left inverse, hence injectivity, of the id → index map (no assumption on leading symbols). -/
theorem unIndex_idIndex (b : Nat) (hb : 2 ≤ b) (s : List Nat) (hs : ∀ c ∈ s, c < b) :
    unIndex b hb (idIndex b s) = s := by
  revert hs
  refine snocInd (P := fun s => (∀ c ∈ s, c < b) → unIndex b hb (idIndex b s) = s) ?_ ?_ s
  · intro; rw [unIndex]; simp [idIndex]; omega
  · intro s c ih hs
    have hc := hs c (by simp)
    have ih := ih (fun x hx => hs x (by simp [hx]))
    have h1 := idIndex_pos b (by omega) s
    rw [idIndex_snoc, unIndex]
    have : ¬ idIndex b s * b + c < b := by
      have := Nat.le_mul_of_pos_left b h1; omega
    simp only [this, dif_neg, not_false_eq_true]
    rw [show idIndex b s * b + c = c + idIndex b s * b by omega,
        Nat.add_mul_div_right _ _ (by omega), Nat.add_mul_mod_self_right,
        Nat.div_eq_of_lt hc, Nat.mod_eq_of_lt hc, Nat.zero_add, ih]

theorem idIndex_injective (b : Nat) (hb : 2 ≤ b) (s t : List Nat)
    (hs : ∀ c ∈ s, c < b) (ht : ∀ c ∈ t, c < b) (h : idIndex b s = idIndex b t) : s = t := by
  rw [← unIndex_idIndex b hb s hs, ← unIndex_idIndex b hb t ht, h]

theorem idIndex_bounds (b : Nat) (s : List Nat) (hs : ∀ c ∈ s, c < b) :
    b ^ s.length ≤ idIndex b s ∧ idIndex b s < 2 * b ^ s.length := by
  revert hs
  refine snocInd (P := fun s => (∀ c ∈ s, c < b) →
    b ^ s.length ≤ idIndex b s ∧ idIndex b s < 2 * b ^ s.length) ?_ ?_ s
  · intro; simp [idIndex]
  · intro s c ih hs
    have hc := hs c (by simp)
    have ⟨i1, i2⟩ := ih (fun x hx => hs x (by simp [hx]))
    rw [idIndex_snoc, List.length_append, List.length_singleton, Nat.pow_succ]
    constructor
    · have := Nat.mul_le_mul_right b i1; omega
    · have := Nat.mul_le_mul_right b (show idIndex b s + 1 ≤ 2 * b ^ s.length by omega)
      rw [Nat.add_mul, Nat.one_mul, Nat.mul_assoc] at this; omega

/-- The largest index of an id of length `n` is `2·b^n - 1` (all symbols `b - 1`), so
    "every id of length `n` fits in a depth-`d` tree" is exactly `2·b^n ≤ 2^d`. -/
theorem idIndex_max (b n a : Nat) (hb : 1 ≤ b) :
    (List.replicate n (b - 1)).foldl (fun a c => a * b + c) a + 1 = (a + 1) * b ^ n := by
  induction n generalizing a with
  | zero => simp
  | succ n ih =>
    rw [List.replicate_succ, List.foldl_cons, ih, Nat.pow_succ]
    have : a * b + (b - 1) + 1 = (a + 1) * b := by rw [Nat.add_mul]; omega
    rw [this, Nat.mul_assoc, Nat.mul_comm b]
```

```
theorem idIndex_fits (b d : Nat) (s : List Nat) (hs : ∀ c ∈ s, c < b)
    (h : 2 * b ^ s.length ≤ 2 ^ d) : idIndex b s < 2 ^ d :=
  Nat.lt_of_lt_of_le (idIndex_bounds b s hs).2 h

theorem idIndex_lt_p (b d : Nat) (s : List Nat) (hs : ∀ c ∈ s, c < b)
    (h : 2 * b ^ s.length ≤ 2 ^ d) (hd : d ≤ 253) : idIndex b s < p := by
  have := idIndex_fits b d s hs h
  have : 2 ^ d ≤ 2 ^ 253 := Nat.pow_le_pow_right (by decide) hd
  have := p_bits.1; omega

/-- Capacity table: for each depth `d`, the maximal id length `n` with `2·b^n ≤ 2^d`. -/
theorem capacity_64 :
    (2 * 64 ^ 5 ≤ 2 ^ 32 ∧ ¬ 2 * 64 ^ 6 ≤ 2 ^ 32) ∧
    (2 * 64 ^ 16 ≤ 2 ^ 100 ∧ ¬ 2 * 64 ^ 17 ≤ 2 ^ 100) ∧
    (2 * 64 ^ 27 ≤ 2 ^ 168 ∧ ¬ 2 * 64 ^ 28 ≤ 2 ^ 168) ∧
    (2 * 64 ^ 30 ≤ 2 ^ 184 ∧ ¬ 2 * 64 ^ 31 ≤ 2 ^ 184) ∧
    (2 * 64 ^ 42 ≤ 2 ^ 253 ∧ ¬ 2 * 64 ^ 43 ≤ 2 ^ 253) := by decide

theorem capacity_256 :
    (2 * 256 ^ 3 ≤ 2 ^ 32 ∧ ¬ 2 * 256 ^ 4 ≤ 2 ^ 32) ∧
    (2 * 256 ^ 12 ≤ 2 ^ 100 ∧ ¬ 2 * 256 ^ 13 ≤ 2 ^ 100) ∧
    (2 * 256 ^ 20 ≤ 2 ^ 168 ∧ ¬ 2 * 256 ^ 21 ≤ 2 ^ 168) ∧
    (2 * 256 ^ 22 ≤ 2 ^ 184 ∧ ¬ 2 * 256 ^ 23 ≤ 2 ^ 184) ∧
    (2 * 256 ^ 31 ≤ 2 ^ 253 ∧ ¬ 2 * 256 ^ 32 ≤ 2 ^ 253) := by decide

theorem capacity_100 :
    (2 * 100 ^ 4 ≤ 2 ^ 32 ∧ ¬ 2 * 100 ^ 5 ≤ 2 ^ 32) ∧
    (2 * 100 ^ 14 ≤ 2 ^ 100 ∧ ¬ 2 * 100 ^ 15 ≤ 2 ^ 100) ∧
    (2 * 100 ^ 25 ≤ 2 ^ 168 ∧ ¬ 2 * 100 ^ 26 ≤ 2 ^ 168) ∧
    (2 * 100 ^ 27 ≤ 2 ^ 184 ∧ ¬ 2 * 100 ^ 28 ≤ 2 ^ 184) ∧
    (2 * 100 ^ 37 ≤ 2 ^ 253 ∧ ¬ 2 * 100 ^ 38 ≤ 2 ^ 253) := by decide

/-! ## Shortlex is a strict total order; objects as finite maps -/

theorem lexLt_trans : ∀ a b c : List Nat, lexLt a b = true → lexLt b c = true → lexLt a c = true
  | x :: as, y :: bs, z :: cs, h1, h2 => by
    simp only [lexLt, Bool.or_eq_true, Bool.and_eq_true, decide_eq_true_eq, beq_iff_eq] at h1 h2 ⊢
    rcases h1 with h1 | ⟨rfl, h1⟩ <;> rcases h2 with h2 | ⟨rfl, h2⟩
    · left; omega
    · left; omega
    · left; omega
    · right; exact ⟨rfl, lexLt_trans as bs cs h1 h2⟩
  | [], _, _, h, _ => by simp [lexLt] at h
  | _ :: _, [], _, h, _ => by simp [lexLt] at h
  | _ :: _, _ :: _, [], _, h => by simp [lexLt] at h

theorem lexLt_total : ∀ a b : List Nat, a.length = b.length → a ≠ b →
    lexLt a b = true ∨ lexLt b a = true
  | [], [], _, h => absurd rfl h
  | x :: as, y :: bs, hl, h => by
    simp only [lexLt, Bool.or_eq_true, Bool.and_eq_true, decide_eq_true_eq, beq_iff_eq]
    by_cases e : x = y
    · subst e
      have : as ≠ bs := fun e' => h (by rw [e'])
      rcases lexLt_total as bs (by simpa using hl) this with h' | h'
      · left; right; exact ⟨rfl, h'⟩
      · right; right; exact ⟨rfl, h'⟩
    · omega
  | [], _ :: _, hl, _ => by simp at hl
  | _ :: _, [], hl, _ => by simp at hl

theorem slt_trans (a b c : List Nat) (h1 : slt a b = true) (h2 : slt b c = true) :
    slt a c = true := by
  simp only [slt, Bool.or_eq_true, Bool.and_eq_true, decide_eq_true_eq, beq_iff_eq] at h1 h2 ⊢
  rcases h1 with h1 | ⟨e1, h1⟩ <;> rcases h2 with h2 | ⟨e2, h2⟩
  · left; omega
  · left; omega
  · left; omega
  · right; exact ⟨by omega, lexLt_trans a b c h1 h2⟩

theorem slt_total (a b : List Nat) (h : a ≠ b) : slt a b = true ∨ slt b a = true := by
  simp only [slt, Bool.or_eq_true, Bool.and_eq_true, decide_eq_true_eq, beq_iff_eq]
  by_cases hl : a.length = b.length
  · rcases lexLt_total a b hl h with h' | h'
```

```
    · left; right; exact ⟨hl, h'⟩
    · right; right; exact ⟨hl.symm, h'⟩
  · omega

theorem slt_asymm (a b : List Nat) (h : slt a b = true) : slt b a = false := by
  cases h' : slt b a
  · rfl
  · have := slt_trans a b a h h'; rw [slt_irrefl] at this; contradiction

theorem keysAsc_iff (k : List Nat) (ks : List (List Nat)) :
    keysAsc (k :: ks) ↔ (∀ k' ∈ ks, slt k k' = true) ∧ keysAsc ks := by
  induction ks generalizing k with
  | nil => simp [keysAsc]
  | cons k₁ ks ih =>
    simp only [keysAsc, List.mem_cons]
    constructor
    · rintro ⟨h1, h2⟩
      refine ⟨?_, h2⟩
      rintro k' (rfl | hk')
      · exact h1
      · exact slt_trans _ _ _ h1 (((ih k₁).1 h2).1 k' hk')
    · rintro ⟨h1, h2⟩; exact ⟨h1 k₁ (Or.inl rfl), h2⟩

/-- A finite map has at most one key-sorted association list. -/
theorem sorted_unique {V : Type} : ∀ (l₁ l₂ : List (List Nat × V)),
    keysAsc (l₁.map Prod.fst) → keysAsc (l₂.map Prod.fst) → (∀ x, x ∈ l₁ ↔ x ∈ l₂) → l₁ = l₂
  | [], [], _, _, _ => rfl
  | [], b :: _, _, _, h => by have := (h b).2 (by simp); simp at this
  | a :: _, [], _, _, h => by have := (h a).1 (by simp); simp at this
  | a :: as, b :: bs, h₁, h₂, h => by
    simp only [List.map_cons, keysAsc_iff, List.mem_map] at h₁ h₂
    have hab : a = b := by
      have ha := (h a).1 (by simp)
      have hb := (h b).2 (by simp)
      simp only [List.mem_cons] at ha hb
      rcases ha with ha | ha
      · exact ha
      rcases hb with hb | hb
      · exact hb.symm
      have x1 := h₂.1 a.1 ⟨a, ha, rfl⟩
      have x2 := h₁.1 b.1 ⟨b, hb, rfl⟩
      rw [slt_asymm _ _ x1] at x2; contradiction
    subst hab
    have : as = bs := by
      apply sorted_unique as bs h₁.2 h₂.2
      intro x
      have hx := h x
      simp only [List.mem_cons] at hx
      constructor
      · intro hx'
        rcases hx.1 (Or.inr hx') with e | e
        · subst e; have := h₁.1 x.1 ⟨x, hx', rfl⟩; rw [slt_irrefl] at this; contradiction
        · exact e
      · intro hx'
        rcases hx.2 (Or.inr hx') with e | e
        · subst e; have := h₂.1 x.1 ⟨x, hx', rfl⟩; rw [slt_irrefl] at this; contradiction
        · exact e
    rw [this]

/-- Insertion by key. -/
def ins {V : Type} (x : List Nat × V) : List (List Nat × V) → List (List Nat × V)
  | [] => [x]
  | y :: ys => if slt x.1 y.1 then x :: y :: ys else y :: ins x ys

def isort {V : Type} (l : List (List Nat × V)) : List (List Nat × V) := l.foldr ins []

theorem mem_ins {V : Type} (x z : List Nat × V) (l : List (List Nat × V)) :
    z ∈ ins x l ↔ z = x ∨ z ∈ l := by
  induction l with
  | nil => simp [ins]
  | cons y ys ih =>
    unfold ins; split
    · simp
    · simp only [List.mem_cons, ih]; constructor <;> (intro h; rcases h with h | h | h <;> simp [h])
```

```
theorem ins_sorted {V : Type} (x : List Nat × V) (l : List (List Nat × V))
    (hl : keysAsc (l.map Prod.fst)) (hx : ∀ y ∈ l, y.1 ≠ x.1) :
    keysAsc ((ins x l).map Prod.fst) := by
  induction l with
  | nil => simp [ins, keysAsc]
  | cons y ys ih =>
    simp only [List.map_cons, keysAsc_iff, List.mem_map] at hl
    unfold ins; split
    · next h =>
      simp only [List.map_cons, keysAsc_iff, List.mem_map, List.mem_cons]
      refine ⟨?_, ?_, hl.2⟩
      · intro k' hk
        rcases hk with rfl | ⟨z, hz, rfl⟩
        · exact h
        · exact slt_trans _ _ _ h (hl.1 z.1 ⟨z, hz, rfl⟩)
      · intro k' ⟨z, hz, e⟩; subst e; exact hl.1 z.1 ⟨z, hz, rfl⟩
    · next h =>
      have hyx : slt y.1 x.1 = true := by
        rcases slt_total x.1 y.1 (fun e => hx y (by simp) e.symm) with h' | h'
        · exact absurd h' h
        · exact h'
      simp only [List.map_cons, keysAsc_iff, List.mem_map]
      refine ⟨?_, ih hl.2 (fun z hz => hx z (by simp [hz]))⟩
      rintro k' ⟨z, hz, rfl⟩
      rcases (mem_ins x z ys).1 hz with rfl | hz
      · exact hyx
      · exact hl.1 z.1 ⟨z, hz, rfl⟩

theorem mem_isort {V : Type} (l : List (List Nat × V)) (z : List Nat × V) :
    z ∈ isort l ↔ z ∈ l := by
  induction l with
  | nil => simp [isort]
  | cons a l ih =>
    simp only [isort, List.foldr_cons] at ih ⊢
    rw [mem_ins, ih]; simp

theorem isort_sorted {V : Type} (l : List (List Nat × V))
    (hd : l.Pairwise (fun a b => a.1 ≠ b.1)) : keysAsc ((isort l).map Prod.fst) := by
  induction l with
  | nil => simp [isort, keysAsc]
  | cons a l ih =>
    rw [List.pairwise_cons] at hd
    simp only [isort, List.foldr_cons]
    apply ins_sorted _ _ (ih hd.2)
    intro y hy
    have := hd.1 y ((mem_isort l y).1 hy)
    exact fun e => this e.symm

/-- Objects as finite maps: every association list with distinct keys has a key-sorted
    representative with the same entries, and it depends only on the set of entries. -/
theorem canonical_rep {V : Type} (l₁ l₂ : List (List Nat × V))
    (h₁ : l₁.Pairwise (fun a b => a.1 ≠ b.1)) (h₂ : l₂.Pairwise (fun a b => a.1 ≠ b.1))
    (h : ∀ x, x ∈ l₁ ↔ x ∈ l₂) : isort l₁ = isort l₂ :=
  sorted_unique _ _ (isort_sorted l₁ h₁) (isort_sorted l₂ h₂)
    (fun x => by rw [mem_isort, mem_isort]; exact h x)

/-! ## Path lookup and its row characterization (what a query circuit checks) -/

def lookupK (k : List Nat) : List (List Nat × J) → Option J
  | [] => none
  | (k', x) :: kvs => if k' = k then some x else lookupK k kvs

/-- The value of `v` at path `π`, if any. -/
def get : J → List Comp → Option J
  | v, [] => some v
  | .arr xs, .idx i :: q =>
    match xs[i]? with
    | some x => get x q
    | none => none
  | .obj kvs, .key k :: q =>
    match lookupK k kvs with
    | some x => get x q
    | none => none
```

```
  | _, _ :: _ => none

theorem mem_flatA (r : Row) : ∀ (xs : List J) (j : Nat),
    r ∈ flatA j xs ↔ ∃ i x, xs[i]? = some x ∧ ∃ r' ∈ flatten x, r = pre (.idx (j + i)) r'
  | [], j => by simp [flatA]
  | x :: xs, j => by
    rw [flatA, List.mem_append, mem_flatA r xs (j + 1), List.mem_map]
    constructor
    · rintro (⟨r', h1, rfl⟩ | ⟨i, y, h1, r', h2, rfl⟩)
      · exact ⟨0, x, by simp, r', h1, rfl⟩
      · exact ⟨i + 1, y, by simpa using h1, r', h2, by simp [Nat.add_assoc, Nat.add_comm 1 i]⟩
    · rintro ⟨i, y, h1, r', h2, rfl⟩
      cases i with
      | zero => left; simp at h1; subst h1; exact ⟨r', h2, rfl⟩
      | succ i =>
        right; simp at h1
        exact ⟨i, y, h1, r', h2, by simp [Nat.add_assoc, Nat.add_comm 1 i]⟩

theorem mem_flatO (r : Row) : ∀ (kvs : List (List Nat × J)),
    r ∈ flatO kvs ↔ ∃ k x, (k, x) ∈ kvs ∧ ∃ r' ∈ flatten x, r = pre (.key k) r'
  | [] => by simp [flatO]
  | (k, x) :: kvs => by
    rw [flatO, List.mem_append, mem_flatO r kvs, List.mem_map]
    constructor
    · rintro (⟨r', h1, rfl⟩ | ⟨k', y, h1, r', h2, rfl⟩)
      · exact ⟨k, x, by simp, r', h1, rfl⟩
      · exact ⟨k', y, by simp [h1], r', h2, rfl⟩
    · rintro ⟨k', y, h1, r', h2, rfl⟩
      simp only [List.mem_cons, Prod.mk.injEq] at h1
      rcases h1 with ⟨rfl, rfl⟩ | h1
      · left; exact ⟨r', h2, rfl⟩
      · right; exact ⟨k', y, h1, r', h2, rfl⟩

theorem WFA_get (xs : List J) (h : WFA xs) : ∀ (i : Nat) (x : J), xs[i]? = some x → WF x := by
  induction xs with
  | nil => simp
  | cons y ys ih =>
    intro i x hx
    cases i with
    | zero => simp at hx; subst hx; exact h.1
    | succ i => simp at hx; exact ih h.2 i x hx

theorem WFO_mem (kvs : List (List Nat × J)) (h : WFO kvs) : ∀ k x, (k, x) ∈ kvs → WF x := by
  induction kvs with
  | nil => simp
  | cons kx kvs ih =>
    obtain ⟨k', y⟩ := kx
    intro k x hx
    simp only [List.mem_cons, Prod.mk.injEq] at hx
    rcases hx with ⟨rfl, rfl⟩ | hx
    · exact h.1
    · exact ih h.2 k x hx

theorem lookupK_iff (kvs : List (List Nat × J)) (hk : keysAsc (kvs.map Prod.fst))
    (k : List Nat) (x : J) : lookupK k kvs = some x ↔ (k, x) ∈ kvs := by
  induction kvs with
  | nil => simp [lookupK]
  | cons kx kvs ih =>
    obtain ⟨k', y⟩ := kx
    simp only [List.map_cons, keysAsc_iff, List.mem_map] at hk
    simp only [lookupK, List.mem_cons, Prod.mk.injEq]
    by_cases e : k' = k
    · subst e
      simp only [if_true, true_and, Option.some.injEq]
      constructor
      · intro h; left; exact h.symm
      · rintro (h | h)
        · exact h.symm
        · have := hk.1 k' ⟨(k', x), h, rfl⟩; rw [slt_irrefl] at this; contradiction
    · simp only [e, if_false, ih hk.2]
      constructor
      · exact Or.inr
      · rintro (⟨h, -⟩ | h)
        · exact absurd h.symm e
```

```
        · exact h

/-- Rows characterize lookup: on the domain, `(π, ℓ)` is a row of `flatten v` exactly when
    the value at `π` is the leaf `ℓ` (a primitive or an empty container). -/
theorem row_iff_get (π : List Comp) : ∀ (v : J) (l : Leaf), WF v →
    ((π, l) ∈ flatten v ↔ get v π = some (ofLeaf l)) := by
  induction π with
  | nil =>
    intro v l _
    simp only [get, Option.some.injEq]
    match v with
    | .null | .bool _ | .num _ _ _ | .str _ | .arr [] | .obj [] =>
      cases l <;> simp [flatten, ofLeaf] <;> constructor <;> intro h <;> simp_all
    | .arr (x :: xs) =>
      simp only [flatten]
      constructor
      · intro h
        obtain ⟨_, _, _, _, _, e⟩ := (mem_flatA _ _ _).1 h
        simp [pre] at e
      · intro h; cases l <;> simp [ofLeaf] at h
    | .obj (kx :: kvs) =>
      simp only [flatten]
      constructor
      · intro h
        obtain ⟨_, _, _, _, _, e⟩ := (mem_flatO _ _).1 h
        simp [pre] at e
      · intro h; cases l <;> simp [ofLeaf] at h
  | cons c q ih =>
    intro v l hv
    match v, c with
    | .null, _ | .bool _, _ | .num _ _ _, _ | .str _, _ | .arr [], .key _ | .obj [], .idx _
    | .arr [], .idx _ | .obj [], .key _ =>
      simp [flatten, get, lookupK]
    | .arr (x :: xs), .key k =>
      simp only [flatten, get]
      constructor
      · intro h
        obtain ⟨_, _, _, _, _, e⟩ := (mem_flatA _ _ _).1 h
        simp [pre] at e
      · intro h; simp at h
    | .obj (kx :: kvs), .idx i =>
      simp only [flatten, get]
      constructor
      · intro h
        obtain ⟨_, _, _, _, _, e⟩ := (mem_flatO _ _).1 h
        simp [pre] at e
      · intro h; simp at h
    | .arr (x :: xs), .idx i =>
      simp only [flatten, get]
      rw [mem_flatA]
      constructor
      · rintro ⟨i', y, h1, ⟨r1, r2⟩, h2, e⟩
        simp only [pre, Nat.zero_add, Prod.mk.injEq, List.cons.injEq, Comp.idx.injEq] at e
        obtain ⟨⟨e1, e2⟩, e3⟩ := e
        subst e1 e2 e3
        rw [h1]
        exact (ih y l (WFA_get _ hv _ y h1)).1 h2
      · intro h
        split at h
        · next y hy =>
          refine ⟨i, y, hy, (q, l), (ih y l (WFA_get _ hv i y hy)).2 h, ?_⟩
          simp [pre]
        · contradiction
    | .obj (kx :: kvs), .key k =>
      simp only [flatten, get]
      rw [mem_flatO]
      constructor
      · rintro ⟨k', y, h1, ⟨r1, r2⟩, h2, e⟩
        simp only [pre, Prod.mk.injEq, List.cons.injEq, Comp.key.injEq] at e
        obtain ⟨⟨e1, e2⟩, e3⟩ := e
        subst e1 e2 e3
        rw [(lookupK_iff _ hv.1 _ _).2 h1]
        exact (ih y l (WFO_mem _ hv.2 _ y h1)).1 h2
      · intro h
```

```
        split at h
        · next y hy =>
          have hm := (lookupK_iff _ hv.1 _ _).1 hy
          refine ⟨k, y, hm, (q, l), (ih y l (WFO_mem _ hv.2 _ y hm)).2 h, ?_⟩
          simp [pre]
        · contradiction

/-- Existence (and hence non-existence) of a path: on the domain, `π` addresses some
    sub-value of `v` exactly when some row's path extends `π`. -/
theorem path_exists_iff (π : List Comp) : ∀ (v : J), WF v →
    ((∃ w, get v π = some w) ↔ ∃ r ∈ flatten v, ∃ t, r.1 = π ++ t) := by
  induction π with
  | nil =>
    intro v _
    simp only [get, List.nil_append]
    constructor
    · intro _
      obtain ⟨r, rs, e⟩ : ∃ r rs, flatten v = r :: rs := by
        cases h : flatten v with
        | nil => exact absurd h (flatten_ne_nil v)
        | cons r rs => exact ⟨r, rs, rfl⟩
      exact ⟨r, by simp [e], r.1, rfl⟩
    · intro _; exact ⟨v, rfl⟩
  | cons c q ih =>
    intro v hv
    match v, c with
    | .null, _ | .bool _, _ | .num _ _ _, _ | .str _, _ | .arr [], .key _ | .obj [], .idx _
    | .arr [], .idx _ | .obj [], .key _ =>
      simp [flatten, get, lookupK]
    | .arr (x :: xs), .key k =>
      simp only [flatten, get]
      constructor
      · rintro ⟨_, h⟩; simp at h
      · rintro ⟨r, h, t, e⟩
        obtain ⟨_, _, _, r', _, rfl⟩ := (mem_flatA _ _ _).1 h
        simp [pre] at e
    | .obj (kx :: kvs), .idx i =>
      simp only [flatten, get]
      constructor
      · rintro ⟨_, h⟩; simp at h
      · rintro ⟨r, h, t, e⟩
        obtain ⟨_, _, _, r', _, rfl⟩ := (mem_flatO _ _).1 h
        simp [pre] at e
    | .arr (x :: xs), .idx i =>
      simp only [flatten, get]
      constructor
      · rintro ⟨w, h⟩
        split at h
        · next y hy =>
          obtain ⟨r, hr, t, e⟩ := (ih y (WFA_get _ hv i y hy)).1 ⟨w, h⟩
          exact ⟨pre (.idx i) r, (mem_flatA _ _ _).2 ⟨i, y, hy, r, hr, by simp⟩, t,
            by simp [pre, e]⟩
        · contradiction
      · rintro ⟨r, h, t, e⟩
        obtain ⟨i', y, h1, ⟨r1, r2⟩, h2, rfl⟩ := (mem_flatA _ _ _).1 h
        simp only [pre, Nat.zero_add, List.cons_append, List.cons.injEq, Comp.idx.injEq] at e
        obtain ⟨e1, e2⟩ := e
        subst e1
        rw [h1]
        exact (ih y (WFA_get _ hv _ y h1)).2 ⟨(r1, r2), h2, t, e2⟩
    | .obj (kx :: kvs), .key k =>
      simp only [flatten, get]
      constructor
      · rintro ⟨w, h⟩
        split at h
        · next y hy =>
          have hm := (lookupK_iff _ hv.1 _ _).1 hy
          obtain ⟨r, hr, t, e⟩ := (ih y (WFO_mem _ hv.2 _ y hm)).1 ⟨w, h⟩
          exact ⟨pre (.key k) r, (mem_flatO _ _).2 ⟨k, y, hm, r, hr, rfl⟩, t,
            by simp [pre, e]⟩
        · contradiction
      · rintro ⟨r, h, t, e⟩
        obtain ⟨k', y, h1, ⟨r1, r2⟩, h2, rfl⟩ := (mem_flatO _ _).1 h
        simp only [pre, List.cons_append, List.cons.injEq, Comp.key.injEq] at e
```

```
        obtain ⟨e1, e2⟩ := e
        subst e1
        rw [(lookupK_iff _ hv.1 _ _).2 h1]
        exact (ih y (WFO_mem _ hv.2 _ y h1)).2 ⟨(r1, r2), h2, t, e2⟩

/-! ## Canonical row order: flattening a canonical value lists rows already sorted -/

/-- Lexicographic order on integer arrays, a proper prefix first. -/
def arrLt : List Nat → List Nat → Bool
  | [], _ :: _ => true
  | a :: as, b :: bs => a < b || (a == b && arrLt as bs)
  | _, _ => false

/-- Component order: indices by value, every index before every key, keys in shortlex. -/
def cLt : Comp → Comp → Bool
  | .idx i, .idx j => i < j
  | .idx _, .key _ => true
  | .key _, .idx _ => false
  | .key a, .key b => slt a b

/-- Path order: component by component, a proper prefix first. -/
def pLt : List Comp → List Comp → Bool
  | [], _ :: _ => true
  | a :: p, b :: q => cLt a b || (a == b && pLt p q)
  | _, _ => false

theorem arrLt_eq_lexLt : ∀ a b : List Nat, a.length = b.length → arrLt a b = lexLt a b
  | [], [], _ => rfl
  | x :: as, y :: bs, h => by
    simp only [arrLt, lexLt, arrLt_eq_lexLt as bs (by simpa using h)]
  | [], _ :: _, h => by simp at h
  | _ :: _, [], h => by simp at h

/-- The component order is exactly the lexicographic order of the components' integer codes. -/
theorem cLt_eq_code (a b : Comp) : cLt a b = arrLt (compCode a) (compCode b) := by
  match a, b with
  | .idx i, .idx j => simp [cLt, compCode, arrLt]
  | .idx i, .key [] => rfl
  | .idx i, .key (c :: cs) => simp [cLt, compCode, arrLt]
  | .key [], .idx j => rfl
  | .key (c :: cs), .idx j => simp [cLt, compCode, arrLt]
  | .key [], .key [] => rfl
  | .key [], .key (c :: cs) => simp [cLt, compCode, arrLt, slt]
  | .key (c :: cs), .key [] => simp [cLt, compCode, arrLt, slt]
  | .key (c :: cs), .key (d :: ds) =>
    simp only [cLt, compCode, arrLt, slt, List.length_cons]
    by_cases h : cs.length = ds.length
    · rw [arrLt_eq_lexLt cs ds h]; simp [h, lexLt]
    · have : (cs.length + 1 == ds.length + 1) = false := by simp; omega
      simp [this]

theorem cLt_irrefl (c : Comp) : cLt c c = false := by
  cases c <;> simp [cLt, slt_irrefl]

theorem pLt_cons_same (c : Comp) (p q : List Comp) : pLt (c :: p) (c :: q) = pLt p q := by
  simp [pLt, cLt_irrefl]

theorem pLt_cons_lt (a b : Comp) (p q : List Comp) (h : cLt a b = true) :
    pLt (a :: p) (b :: q) = true := by
  simp [pLt, h]

def RowLt (r s : Row) : Prop := pLt r.1 s.1 = true

theorem pairwise_pre (c : Comp) (rows : List Row) (h : rows.Pairwise RowLt) :
    (rows.map (pre c)).Pairwise RowLt := by
  rw [List.pairwise_map]
  exact h.imp (fun {a b} hab => by simp only [RowLt, pre, pLt_cons_same]; exact hab)

mutual
theorem flatten_sorted : (v : J) → WF v → (flatten v).Pairwise RowLt
  | .null, _ | .bool _, _ | .num _ _ _, _ | .str _, _ | .arr [], _ | .obj [], _ => by
    simp [flatten]
  | .arr (x :: xs), hw => by
    simp only [flatten]; exact flatA_sorted (x :: xs) 0 hw
```

```
  | .obj (kx :: kvs), hw => by
    simp only [flatten]; exact flatO_sorted (kx :: kvs) hw.2 hw.1
theorem flatA_sorted : (xs : List J) → (j : Nat) → WFA xs → (flatA j xs).Pairwise RowLt
  | [], _, _ => by simp [flatA]
  | x :: xs, j, hw => by
    simp only [flatA]
    rw [List.pairwise_append]
    refine ⟨pairwise_pre _ _ (flatten_sorted x hw.1), flatA_sorted xs (j + 1) hw.2, ?_⟩
    intro a ha b hb
    obtain ⟨a', _, rfl⟩ := List.mem_map.1 ha
    obtain ⟨i, _, _, b', _, rfl⟩ := (mem_flatA _ _ _).1 hb
    exact pLt_cons_lt _ _ _ _ (by simp [cLt]; omega)
theorem flatO_sorted : (kvs : List (List Nat × J)) → WFO kvs → keysAsc (kvs.map Prod.fst) →
    (flatO kvs).Pairwise RowLt
  | [], _, _ => by simp [flatO]
  | (k, x) :: kvs, hw, hk => by
    simp only [flatO]
    rw [List.pairwise_append]
    simp only [List.map_cons, keysAsc_iff] at hk
    refine ⟨pairwise_pre _ _ (flatten_sorted x hw.1), flatO_sorted kvs hw.2 hk.2, ?_⟩
    intro a ha b hb
    obtain ⟨a', _, rfl⟩ := List.mem_map.1 ha
    obtain ⟨k', y, hm, b', _, rfl⟩ := (mem_flatO _ _).1 hb
    exact pLt_cons_lt _ _ _ _ (by simp only [cLt]; exact hk.1 k' (List.mem_map.2 ⟨(k', y), hm, rfl⟩))
end

/-! ## Concrete checks and consequences -/

/-- Empty-container markers matter: `{"a":1,"b":[]}` and `{"a":1}` get different encodings. -/
theorem empty_marker_distinguishes :
    encode (.obj [([97], .num false 0 1), ([98], .arr [])]) ≠ encode (.obj [([97], .num false 0 1)]) := by
  intro h
  have := encode_injective _ _
    (by simp [WF, WFO, WFA, keysAsc, numOK, slt, lexLt])
    (by simp [WF, WFO, keysAsc, numOK]) h
  injection this with h'
  simp at h'

/-- `{"a":1}` is the single element 1‖11‖11‖297‖04 2101 (concrete check). -/
example : encode (.obj [([97], .num false 0 1)]) = [11111297042101] := by native_decide

/-- A one-digit tail joins a run, and decoding still completes the pending integer
    (instance of `decodeTok_stream` and `tokInts_toks`, checked concretely). -/
example : stream (toks [123456789, 1, 1]) = [9, 1, 2, 3, 4, 5, 6, 7, 8, 0, 3, 9, 1, 1] := by
  native_decide

theorem ints_tok_roundtrip (ns : List Nat) :
    (decodeTok (stream (toks ns))).bind (tokInts []) = some ns := by
  rw [decodeTok_stream _ (toks_ok _)]; exact tokInts_toks _

theorem one_digit_tail_roundtrip :
    (decodeTok (stream (toks [123456789, 1, 1]))).bind (tokInts []) = some [123456789, 1, 1] :=
  ints_tok_roundtrip _

theorem capacity_256_elements : W * 256 = 19200 := by decide

/-! ## In-field digit decomposition is unique -/

theorem ofD_inj (a b : List Nat) (ha : ∀ x ∈ a, x < 10) (hb : ∀ x ∈ b, x < 10)
    (hl : a.length = b.length) (h : ofD a = ofD b) : a = b := by
  have e1 := fixD_guard a ha 0
  have e2 := fixD_guard b hb 0
  rw [Nat.zero_mul, Nat.zero_add] at e1 e2
  rw [← e1, ← e2, hl, h]

/-- A vector of at most 76 decimal digits is determined by its residue mod `p`: digit
    decomposition constraints over $\mathbb{F}_p$ have a unique solution. -/
theorem digits_mod_p_unique (a b : List Nat) (ha : ∀ x ∈ a, x < 10) (hb : ∀ x ∈ b, x < 10)
    (hl : a.length = b.length) (h76 : a.length ≤ 76) (h : ofD a % p = ofD b % p) : a = b := by
  have h1 := ofD_lt a ha
  have h2 := ofD_lt b hb
  have h3 : 10 ^ a.length ≤ 10 ^ 76 := Nat.pow_le_pow_right (by decide) h76
  have hp := p_bounds.1
  rw [← hl] at h2
```

```
  rw [Nat.mod_eq_of_lt (by omega), Nat.mod_eq_of_lt (by omega)] at h
  exact ofD_inj a b ha hb hl h

/-! ## What a walk over the committed elements recovers -/

/-- The rows read back from an element vector: unpack, parse units, expand runs, complete
    integers, parse rows. This is the walk a query circuit must constrain. -/
def walkRows (es : List Nat) : Option (List Row) :=
  match decodeTok (unpack es) with
  | none => none
  | some ts =>
    match tokInts [] ts with
    | none => none
    | some ns => parseRows ns.length ns

theorem walkRows_encode (v : J) (z : Nat) :
    walkRows (encode v ++ List.replicate z 0) = some (flatten v) := by
  have hok := toks_ok (ints (flatten v))
  unfold walkRows encode
  rw [unpack_pack _ (stream_lt _ hok), decodeTok_stream _ hok]
  simp only
  rw [tokInts_toks]
  simp only
  exact parseRows_ints' _

/-- Soundness of a leaf query: on a committed canonical value, the walk yields a row
    `(π, ℓ)` exactly when the value at `π` is the leaf `ℓ`. -/
theorem query_rows (v : J) (hv : WF v) (z : Nat) (π : List Comp) (l : Leaf) (rs : List Row)
    (h : walkRows (encode v ++ List.replicate z 0) = some rs) :
    (π, l) ∈ rs ↔ get v π = some (ofLeaf l) := by
  rw [walkRows_encode] at h
  injection h with h
  subst h
  exact row_iff_get π v l hv

/-- Soundness of a non-existence query: `π` addresses nothing exactly when no row's
    path extends `π`. -/
theorem absent_rows (v : J) (hv : WF v) (z : Nat) (π : List Comp) (rs : List Row)
    (h : walkRows (encode v ++ List.replicate z 0) = some rs) :
    get v π = none ↔ ∀ r ∈ rs, ∀ t, r.1 ≠ π ++ t := by
  rw [walkRows_encode] at h
  injection h with h
  subst h
  have e := path_exists_iff π v hv
  constructor
  · intro hn r hr t ht
    obtain ⟨w, hw⟩ := e.2 ⟨r, hr, t, ht⟩
    rw [hn] at hw; contradiction
  · intro hall
    cases hg : get v π with
    | none => rfl
    | some w =>
      obtain ⟨r, hr, t, ht⟩ := e.1 ⟨w, hg⟩
      exact absurd ht (hall r hr t)

/-- The code is prefix-free but not self-synchronizing. The integers of
    `{"x": [1,1,97,2,1,0,1] as code units}` contain, from position 5, exactly the row
    of `{"a": 1}`, although the path `a` does not exist. A walk must therefore start at
    the first digit; a row cannot be located by pattern search. -/
def spoof : J := .obj [([120], .str [1, 1, 97, 2, 1, 0, 1])]

theorem spoof_ints : ints (flatten spoof) = [1, 1, 120, 3, 7] ++ ints [([.key [97]], .num false 0 1)] := rfl

theorem spoof_absent : get spoof [.key [97]] = none := rfl

end Canon

#print axioms Canon.decode_encode
#print axioms Canon.encode_injective
#print axioms Canon.encode_injective_pad
#print axioms Canon.encode_lt_p
#print axioms Canon.encode_length
#print axioms Canon.p_bounds
#print axioms Canon.unpack_segments
```

```
#print axioms Canon.decodeTok_stream
#print axioms Canon.decodeTok_units
#print axioms Canon.unflatten_flatten
#print axioms Canon.parseRows_ints'
#print axioms Canon.tokInts_toks
#print axioms Canon.num_canon_unique
#print axioms Canon.canonical_rep
#print axioms Canon.flatten_sorted
#print axioms Canon.cLt_eq_code
#print axioms Canon.row_iff_get
#print axioms Canon.path_exists_iff
#print axioms Canon.unIndex_idIndex
#print axioms Canon.idIndex_injective
#print axioms Canon.idIndex_bounds
#print axioms Canon.idIndex_max
#print axioms Canon.idIndex_lt_p
#print axioms Canon.capacity_64
#print axioms Canon.empty_marker_distinguishes
#print axioms Canon.one_digit_tail_roundtrip
#print axioms Canon.digits_mod_p_unique
#print axioms Canon.walkRows_encode
#print axioms Canon.query_rows
#print axioms Canon.absent_rows
#print axioms Canon.spoof_absent
#print axioms Canon.p_bits
#print axioms Canon.payload_bits
#print axioms Canon.capacity_256
#print axioms Canon.capacity_100
#print axioms Canon.spoof_ints
#print axioms Canon.unpack_pack
#print axioms Canon.elem_lt_p
#print axioms Canon.sorted_unique
#print axioms Canon.slt_total
#print axioms Canon.slt_trans
#print axioms Canon.isort_sorted
#print axioms Canon.idIndex_fits
#print axioms Canon.expand_compress
#print axioms Canon.capacity_256_elements
```