kaiv — Specification

kaiv.io

version 1.0-draft.15

Contents

Terminology reference. All kaiv terms used in this document (namespace, namepath, field address, canonical form, Compiler, Denormalizer, Validator, integrity check, etc.) are defined in Appendix A, the single source of truth for kaiv terminology.

Status. This document is the working specification, organized around the canonical Level system (Scalars / Trees / Tables / Collation / Corpus-Dependent Features) defined in the Level chapters below and summarized in the Terminology appendix (§A.11). Foundations, Levels 0–4, Compiled Schema (including the map form), Mappings (`.maiv`), Parsing Requirements, the Formal Grammar (ABNF, Levels 0–1), Errors, and File Representation are populated. The built-in unit set is enumerated normatively (§2.7.7), Level 3 collation is pinned to a reference CLDR version and strength (§5.3), and the pipeline materializes defaults and nulls into `.daiv` — the Denormalizer is schema-aware (§2.6.17). Implementation-side background (the build/runtime split, the certification boundary, the performance model) is collected in the non-normative Appendix B. This document's conformance surface is **Levels 0–3**; Level 4 (Corpus-Dependent Features) is defined and reserved here (§6) and specified separately in the experimental corpus specification, on its own version cadence — this document's stability promise does not extend to it. The unit-definition file format (`.faiv`, §2.7.9) and the `.!units` import are populated; constrained-union lowering is specified (§2.6.14); and namespace-schema delegation — the discriminated schema set — is specified (§7.6), with reference-implementation support pending.

Introduction

kaiv is an immutable structural type system for data at rest. It is realized as a family of line-oriented UTF-8 text formats — authored data (`.kaiv`), canonical data (`.raiv`, `.daiv`), schemas (`.saiv`, `.csaiv`), type libraries (`.taiv`), unit definitions (`.faiv`), and schema mappings (`.maiv`) — that share one line grammar, so a single per-line classifier reads every file in the family. Data is validated against compiled schemas by a constant-memory parallel scan, and the fully denormalized canonical form (`.daiv`) is the only artifact a downstream consumer ever needs to trust.

This document specifies the format family: the foundations (data model, line grammar, sigil system, ordered keys), the five conformance Levels — Level 0 (Scalars) through Level 4 (Corpus-Dependent Features; reserved, specified separately) — the compiled schema (`.csaiv`) validation contract, parsing requirements, the formal grammar (ABNF, Levels 0–1), the error taxonomy, and the file representation (extensions, encoding, media types).

The intended audience is implementers — of Lexers, of the Compiler, Denormalizer, and Validator stages, and of tooling that consumes the canonical forms — and authors of kaiv schemas, type libraries, and data.

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in BCP 14 (RFC 2119, RFC 8174) when, and only when, they appear in small capitals, as shown here.

1 Foundations

This section establishes the substrate that every Level builds on: the data model, the line grammar shared by every kaiv file (`caiv`), the sigil system, and the ordered-keys property.

1.1 Scope

This specification defines:

— the kaiv text syntax — the shape of every line in every kaiv file (`.kaiv`, `.daiv`, `.raiv`, `.saiv`, `.csaiv`, `.taiv`, `.faiv`, `.qaiv`, `.maiv`);
— the **Lexer** requirements — what a regular-grammar token producer MUST do to emit a conformant token stream from kaiv text, including the per-line classifier, error conditions, UTF-8 handling, and line-number tracking;
— the **Compiler** / **Denormalizer** / **Validator** pipeline — the three Parser stages that transform `.kaiv` to `.raiv` to `.daiv` and validate against a `.csaiv`;
— the **Compiled Schema** (`.csaiv`) format — the validation contract;
— the type system — `str` as the single primitive, the constraint triple, and the named-type resolution model;
— the unit annotation system — physical and currency units that compose with numeric types, including built-in SI base units, compound-unit grammar, and the `kfaiv.com` registry for custom units;
— the five conformance Levels (Scalars, Trees, Tables, Collation, Corpus-Dependent Features) and what each Level adds — the fifth defined and reserved here, specified separately (§6).

This specification does *not* define:

— parser APIs — implementations may use channels, iterators, callbacks, or any pattern that suits their host language;
— internal data structures — how a parser represents emitted tokens (tuples, structs, objects) is implementation-defined;
— application logic that consumes the validated stream — JSON serialization, ProtoBuf population, configuration loading, format conversion, etc.;
— variable resolution strategy beyond the canonical pipeline rules — applications resolving `$.name` or `$path::field` references at additional layers (environment, host config) define their own resolution order;
— the semantics of any specific named type beyond the `std/core` definitions in this spec — domain types live in their own `.taiv` files.

Conformance is determined by adherence to this specification, not by behavioral equivalence with any particular implementation. Many implementations may coexist (streaming parsers for embedded systems, channel-based parsers in Go or Erlang, iterator-based parsers in Rust or Python), each conformant to the same spec.

1.2 The Data Model

The data model is a **tree**: interior nodes are namespaces, leaf nodes are **literals** (scalar fields and scalar array elements). A **name** is a scope-local identifier (analogous to "key" in JSON). A **namepath** is a globally unique address composed of names separated by `/`. A **namespace** is the container that a name (or namepath) refers to. The `::` operator is the **field projection operator**: it transitions from the tree (namespaces) to a leaf value, selecting one named property from a namespace — either a scalar field (`::host`) or a scalar array element (`/@ports::0`, where the numeric index `0` is the field name).

Arrays are a **special form of namespace** that uses integer (index) fields. The `@` sigil marks an array namespace. Array elements are accessed by the operator immediately after the array name and before the index: `@name::index` projects a scalar element (the index is the field), and `@name/index` descends into a namespace element (the element is an interior node). This is the same operator distinction that governs every other path step — `::` exits the tree to a leaf value, `/` stays in the tree. A **mixed array** uses `::index` for some elements and `/index` for others, per element.

kaiv text is what you author and what the Lexer reads — the raw characters in a `.kaiv` file, before any parsing or validation. A **kaiv document** is the validated abstract tree with the root namespace, all namepaths resolved, and all types checked — the result of the full three-stage pipeline. The **Compiler** produces **relational kaiv text** (`.raiv`) — the relational intermediate where field references are preserved, all variables resolved and elided, and all namepaths fully qualified. The **Denormalizer** then reads `.raiv` — together with the compiled schema (`.csaiv`) when the data declares one — expands `$field` references, materializes absent optional fields (§2.6.13), lifts untyped lines to their fields' retained head types (§7.3), and converts authored same-dimension units into each field's declared unit (§2.7.3) to produce **denormalized kaiv text** (`.daiv`) — the self-contained deployment artifact where all field references are resolved to their values and every schema-declared field is present. The **Validator** checks `.daiv` against the compiled schema (`.csaiv`) and produces pass/fail. `.kaiv` is source code; `.raiv` is the compiled intermediate; `.daiv` is the fully denormalized deployment artifact; only `.daiv` is what safety-critical systems ever see.

1.3 The Universal Line Grammar (`caiv`)

Every file in the kaiv family — `.kaiv`, `.daiv`, `.raiv`, `.saiv`, `.csaiv`, `.taiv`, `.faiv`, `.maiv`, `.qaiv`, and the draft `.msaiv` (reserved — corpus specification, §6) — shares the same line grammar. This shared grammar is called **`caiv`** (core attributed information values, pronounced "cave"). Once you understand `caiv`, you know the shape of every kaiv file. Each chapter then only needs to explain what is *different* about its file type, not what is the same.

`caiv` is a pedagogical anchor — a named invariant that makes the spec easier to read, teach, and discuss. It is not a file extension (no `.caiv` files exist on disk), not an implementation component, and not a certification target.

1.3.1 The Six Rules

For each line in any kaiv file, exactly one of the six rules below applies. Authored `.kaiv`/`.saiv` files add one structural case ahead of rule 5 — **block-delimiter lines** — described in the note after the table; canonical files (`.daiv`/`.raiv`/`.csaiv`) never contain them.

Rule	Test	Classification
1	Line is blank (empty or whitespace only)	Skip
2	Line starts with `#`	Comment
3	Line starts with `//`	Doc comment

Rule	Test	Classification
4	Line starts with `.!` or `.?`	Declaration — see §2.1.1 for the full keyword set
5	Line contains `=`	Content line — split on the first `=` into a **left side** and a **right side**
6	None of the above (no `=`, not declaration/comment/blank)	Metadata annotation — classify by first character (`!`, `?`, `&`; in `.saiv`/`.taiv` also constraint leaders; in `.faiv` a definition-line leader — see below). In authored `.kaiv`, a `$/.` leader is a variable-splat line (§2.5.2)

Block-delimiter lines (authored files). A line whose first character (after leading-whitespace stripping) is `[` or `(` is a **structure line** — a section-block or namespace-block open or close: `[/@servers]`, `[]`, `(/server)`, `()`, including a section-open line carrying a Level 2 table header (§10.7). Structure lines are recognized by their bracket/paren delimiters **before** the rule-5 `=` test, because a table header may itself contain `=` (`[/@servers host=! min=1]`) and would otherwise misclassify as a content line. A line that both opens and closes with brackets (…`[]`) or parens (…`()`) is a structure line; this is what distinguishes it from a range-constraint metadata line (`[0,100] ..num`), which does not end in `]`. Structure lines occur only in authored `.kaiv` and `.saiv` files; canonical files have blocks already expanded to indexed namepaths. The pairing of open/close lines is resolved by the Compiler (§3.1.2), not the line classifier.

Collection-constraint lines (`.csaiv`). A compiled schema carries collection-constraint lines (§5.5) — an array or namespace path followed by bracketed clauses (uniqueness, cardinality, and foreign-key on arrays; map-key and schema-delegation on namespaces): `/@servers [unique::host,port] [min=1] [max=50]`. A cardinality clause may contain `=` (`[min=1]`), which would misclassify under rule 5. In `.csaiv` files, a line whose first character is `/` and that contains no `'` delimiter is a collection-constraint line, recognized **before** the rule-5 `=` test — the same carve-out-ahead-of-rule-5 mechanism as block-delimiter lines in authored files.

Map-key lines (`.saiv`). Inside a map schema block, the key line is a whole pattern in key position (§7.5): `/^[a-z]{2,3}$/=`. The pattern body may itself contain `=`, so — by the same mechanism as the rule-6 priority — a `.saiv` line leading with `/` whose **entire** text parses as a single pattern followed by a terminal `=` is classified as a content line with the pattern as its key, ahead of the rule-5 first-`=` split. The pattern *is* the key grammar, so the bare-name key check (`INVALID_KEY_ERROR`) does not apply to it; the schema compiler rejects the line kind anywhere outside a map schema block.

1.3.2 Classifier Pseudocode

```
for each line:                          # leading whitespace already stripped
```

```
 if blank                      → skip
 if starts with '#'            → comment
 if starts with '//'           → doc comment
 if starts with '.!' or '.?' → declaration
 if starts with '[' or '(' AND ends with ']' or ')'
                               → structure line   # authored .kaiv/.saiv
only:
                                                   # block open/close,
incl. table
                                                   # header; BEFORE the
'=' test.
                                                   # The closing-delimiter
 test is
                                                   # what keeps "[0,100]
..num" a
                                                   # rule-6 constraint
line
 if starts with '/' and has no "'"
                               → collection line  # .csaiv only:
collection
                                                  # constraints; a
cardinality
                                                  # clause may contain '='
 if starts with a metadata leader for this file kind
    and the ENTIRE line parses as rule 6
                               → metadata annotation  # rule-6 priority: a
 pattern
                                                      # or enum item may
contain '='
 if contains '='               → content line: split on first '=' → (left,
 right)
 else                          → metadata annotation: classify by first
char
                                 # .kaiv: !, ?, &, $/. (splat)
                                 # .saiv/.taiv also: / { .. [
```

The classifier is **=-based, not first-character-based for content lines**. The `!` sigil means different things in different file types, which is why first-character classification alone is insufficient.

Rule 4 tests for the two-character declaration sigils `.!` and `.?`, not for a bare leading `.` — a line beginning with `.` followed by any other character is not a declaration. In particular, hidden-variable definitions (`.name=value`, see §2.5) begin with `.` but fall through to rule 5 as ordinary content lines.

Rule-6 priority for metadata-leader lines. A pattern or enumeration item may contain a literal `=` — an annotation like `!str/^a=b$/`, or an enum with `=` in a member. Under naive rule ordering, rule 5 would misclassify these as content lines. So: for a line whose first character is a rule-6 metadata leader for the file kind (`!`, `?`, `&` in data files; additionally `/`, `{`, `[`, `..` in `.saiv`/`.taiv`; in `.faiv` any character that can begin a definition line — `ALPHA`/`DIGIT` opening a unit expression, or `$` for a currency dimension), the classifier first attempts the full rule-6 parse; the parse must consume the **entire line** to classify, otherwise the line

falls through to the rule-5 =-test. Falling through is what keeps every existing content form correct: a canonical line `!str'::host=x` fails the annotation parse at ' and lands in rule 5; a `.taiv` definition `&int=` fails the named-annotation parse at = and lands in rule 5; an authored `/server::host=x` in a schema fails the constraint-line parse and lands in rule 5.

For example: in `.kaiv`, a line like `!int` (no =) is a metadata annotation (rule 6 — a type annotation above the next data line); in `.daiv`, a line like `!int?sensor23'::temperature_f=100` (contains =) is a content line (rule 5). The disambiguation is by the presence or absence of =, refined by the rule-6 priority above for metadata-leader lines whose grammar admits an embedded =.

1.3.3 How Each File Type Interprets the Left Side and Right Side

Each file type defines its own interpretation of the left side and right side of =:

File type	Left side interpretation	Right side interpretation
`.daiv` / `.raiv`	`!type:unit?prov'namepath` — metadata prefix (`!type` plus optional `:unit` and optional `?prov`) followed by ' then a fully-qualified namepath	Value (scalar string)
`.csaiv`	`constraint'namepath` — constraint form followed by ' then a namepath	The compile-time-resolved applicable default (often empty); requiredness is carried by the operator (= required, ?= optional). The Validator ignores the right side
`.kaiv`	Bare key, namespaced key (`/server::host`), array op (`/@ports+=`), variable, field reference, etc.	Value, variable reference, or field reference
`.saiv`	Field definition key — the type annotation sits on the metadata line above	The field's **default value** (a kaiv value is never absent, only empty — an empty right side is the empty-string default); optionality is carried by the operator (= required, ?= optional)
`.taiv`	Named type definition — `&name`	The type's default value (an empty right side is the inert empty-string default), inherited by fields through the default cascade (§2.6.13)
`.faiv`	Unit definition — `&name` (the dimension/factor line sits above)	Empty, or an alias target (`&alias=name`)
`.qaiv`	Query pattern — path expression with optional predicates	Match expression

1.3.4 Metadata Annotations (Rule 6)

Metadata annotations are lines with no = that are not declarations, comments, blank, or structure lines. In **data** files (.kaiv) they are a small, closed set of sigil-prefixed lines:

First character	Meaning
!	Type annotation (!int, !str, !bool, etc.) — annotates the type of the next content line; the elided-type form !:unit asserts only the unit (§2.7.4)
?	Provenance annotation (?id or ?id@timestamp) — annotates the provenance source of the next content line
&	Named type annotation (&port, &datetime) — like !type but references a named type from a library

In .saiv schema files and .taiv type library files, rule-6 lines additionally include **constraint lines** — space-separated constraint items placed above a field or &name= definition (§2.8, §2.6.13). A constraint line may lead with / (pattern), { (enum), .. (span, a leading .), [(range, when followed by further items so the line does not end in]), in addition to !/&. So the rule-6 first-character set for schema and type library files is { ! ? & / { . [} plus lines leading with the reserved re{sep} pattern-literal introducer, not the three-element data-file set. A re{sep}-leading line that fails the full-line parse is a malformed literal (INVALID_CONSTRAINT_ERROR), never a content line. A **length constraint cannot lead** a constraint line: rule 2 fires first and classifies any #-leading line as a comment, so #[2,8] alone is a comment, not a constraint. Authors put another item first (..lex #[2,8]) or use the whitespace-free annotation form (!str#[2,8]), where the # is not line-leading. The mirror-image trap: a [-leading constraint line whose *final* item ends in] ([0,100] #[2,8]) opens and closes with brackets and therefore classifies as a structure line — glue such items into an annotation instead (!str[0,100]#[2,8]). The authoritative production is constraint-line in §10; a line that is wholly …[] or …() is a structure line (§1.3.1), recognized ahead of rule 6.

A metadata annotation binds to the next content line as a whole: when that line **expands** to several canonical lines (;= vector assignment, := struct assignment, +:= array-append struct), the annotation applies to **every** line of the expansion — !int above /limits :=rps=500|burst=200 types both fields. This is what makes single-annotation authoring of homogeneous structs and vectors possible; heterogeneous members need the per-line form.

Annotations of *different* kinds stack: at most one type-designating annotation (!…type or &name) and at most one provenance annotation (?…sourceID) may appear, in either order, above the same content line — this is how a data line carrying both a type and provenance (!int?sensor23'::temperature_f=100 in canonical form) is authored. A second annotation of the *same* kind above one content line raises MetadataWithoutTargetError (§11.2).

Metadata annotations appear only in authored files (.kaiv, .saiv, .taiv). In canonical files (.daiv, .raiv, .csaiv), rule 6 never applies — every content line contains =, and type and provenance information is folded into the left side of = as part of the metadata prefix before ' (.csaiv collection-constraint lines are recognized by their own carve-out ahead of rule 5, see §1.3.1). Metadata annotations as separate lines exist only in authoring.

Once you know `caiv`, you know the shape of every kaiv file. The Six Rules apply universally — across all ten file types, across all conformance Levels, across all tool stages. Each Level only needs to explain what is *different* about its file types: how they interpret the left side and right side of `=`, which comment forms they allow, which declarations they use, and whether they have metadata annotations. The Six Rules themselves never change.

These Foundations tables are the *teaching* presentation of the line grammar. The normative, machine-checkable form is the ABNF in §10, which **wins on any conflict** with the prose here (it says so itself). When implementing, treat that grammar — not these tables — as authoritative, and report any discrepancy.

1.4 The Sigil System

The sigils are **type discriminators** that map onto the fundamental data structures shared by every major interchange format. Every major structured data format supports scalars, arrays, and objects; kaiv's sigils correspond to exactly those three categories, plus declarations.

Sigil	kaiv Construct	JSON	TOML	ProtoBuf	GraphQL	ASN.1
(none)	`key=value` scalar	primitive	value	scalar field	scalar field	primitive type
`@`	array — **a namespace with integer fields**. Scalar array elements are projected via `@name::index`; namespace array elements are descended into via `@name/index`	array	array	`repeated` field	list type	`SEQUENCE OF`
`/`	namespace — **a structural sigil that survives canonicalization**. `/server::host` in authored `.kaiv` becomes `!str'/server::host=localhost` in canonical `.daiv`. The `/` is part of the canonical namepath	object	table	`message` / nested message	type / nested type	`SEQUENCE` / module
`.!`	declaration (`.!kaiv`, `.!schema`, `.!types`, `.!registry`) — format and schema declarations, type library imports, and registry prefix overrides	—	—	package / syntax	—	module header
`.?`	provenance source declaration (`.?id uri`) — document-level declaration that maps a short provenance ID to a full URI	—	—	—	—	—

Sigil	kaiv Construct	JSON	TOML	ProtoBuf	GraphQL	ASN.1
`$`	dereference operator. **Variables** (dot-prefixed): `$.name`, `$@.name`, `$/.name`. **Field references** (no `.` after `$`): `$field`, `$path::field`. Absent in verbatim documents (§2.5.6)	—	—	—	—	—
`!`	type annotation — the **type sigil**: `!int` on a metadata line in authored files, `!type[constraints]: unit` in the canonical metadata prefix before `'`. Survives canonicalization as part of every canonical line	—	—	field type	type	type
`&`	imported library type annotation (`&name` on metadata line) or named type definition (`&name=` in `.taiv`). Authoring-only — resolved to `!library /path/typename` in canonical form	—	—	named type / logical type	—	named type
`?`	provenance annotation/reference — `?id` (or `?id@timestamp` or `?id@timestamp#dpid`) on a metadata line in authored `.kaiv`, or inline in the metadata prefix before `'` in canonical `.daiv`/`.raiv`	—	—	—	—	—

1.4.1 Sigil Survival Rule

The harmonized rule is: **structural sigils (`/`, `@`) survive canonicalization; resolution sigils (`.`, `$`) do not.**

Sigil	Role	`.kaiv`	`.daiv`	Why
`/`	Namespace path	[x] present	[x] present	Structural: self-describing, matches query syntax
`@`	Array path	[x] present	[x] present	Structural: self-describing, matches query syntax
`.`	Variable / hidden name	[x] present	[ ] elided	Resolved by the Compiler — variables have no existence in canonical form

Sigil	Role	`.kaiv`	`.daiv`	Why
`$`	Dereference	[x] present	[ ] elided	`$.var` resolved by the Compiler; `$field` resolved by the Denormalizer — neither appears in canonical form

The DFA dispatch after `'` in canonical form follows directly from these rules:

First char after `'`	Structural meaning
`/`	Namespace path follows — read until `::` for the field name. Array steps appear inside the path as `@`-prefixed steps (`/@servers/0`)
`:`	Expect second `:`, then root field mode

This is one additional character check compared to a design where `/` is stripped. The benefit is that every canonical line is self-describing without context.

1.4.2 The Two Path Operators

The two path operators complement the sigils:

Operator	Formal name	Operation	Transition
`/`	tree descent operator	Navigate from a name to a child name — purely interior node to interior node; never terminal	interior node → interior node
`::`	field projection operator	Select a leaf from a namespace — the field name immediately follows `::` and is always the last element before `=`. Appears in **every** canonical data line	interior node → leaf value

`/` stays within the world of tree nodes — it moves from one namespace to a child namespace. `::` exits the tree: it projects a leaf value out of a namespace. After a projection, the path is terminal — there is nothing left to navigate into. This is why `/` can be chained indefinitely but `::` always ends a path. Every canonical data line has `::` as the tree→leaf boundary.

The same distinction governs arrays: after an array name (`@name`), using `::index` makes that index a scalar field — `!type'/@name::index=value` is a leaf. Using `/index` descends into a namespace element — `!type'/@name/index::field=value` requires further `::` projection to reach the leaf. The `@` sigil always means “array namespace”; the operator after the array name and before the index determines element kind.

1.4.3 Authoring vs. Canonical Sigil Resolution

`&` joins several authoring constructs that canonicalize to a more explicit form. None of these appear in `.daiv` or `.csaiv`:

Authoring form (`.kaiv` /`.saiv`)	Canonical form (`.daiv`/`.csaiv`)	What the Compiler/Denormalizer/Validator does
`&name` type annotation	`!library/path/name` (non-core) or `!core-shorthand` (for `std/core`)	Resolves `&name` against imported type libraries; `std/core` types stay as `!int`/`!bool` etc., all others become `!library/path/name`
`/ns::field=value` (authored namespace path)	`/ns::field=value` (canonical — **unchanged**)	`/` survives canonicalization. It is a structural sigil, not a resolution sigil.
`+=` array append	Indexed lines (`/@name::0=v`, `/@name::1=v`, ...)	Tracks index counter, emits numbered assignments
`;=` vector assignment	Indexed lines (same as `+=` but batch)	Same as `+=` but processes semicolon-separated values
`:=` namespace assignment	Individual field lines (`/path::field=value`, ...)	Splits pipe-delimited pairs into separate `/`-prefixed assignments; `/` is preserved
`+:=` array-append namespace assignment	Indexed namespace-element lines (`/@name/0::host=a`, `/@name/0::port=1`, ...)	Tracks index counter like `+=`, splits pipe-delimited pairs like `:=`, emits `/@name/index::field=value` lines (see §3.1.1)
`$.name` / `$@.name` / `$/.name` variable reference	Inlined value (variable elided from output)	Substitutes value from variable table; elides dot-prefixed definitions
`$field` / `$path::field` field reference	`.raiv`: preserved as-is; `.daiv`: resolved	In `.daiv`, inlines the value from the field table; in `.raiv`, preserved verbatim
`!::unit` elided-type unit annotation	`.raiv`: preserved; `.daiv`: resolved head (or `!float`) with the declared unit, value converted	Denormalizer inherits the governing head, else `float`; converts to the declared unit (§2.7.4)
`?id` (or `?id@timestamp`, etc.) metadata line	provenance list inline before `'` in canonical metadata prefix	Collapses per-line provenance annotations into the canonical metadata prefix. Unlike `&name`, `?id` is **not** resolved away — it survives into canonical form

1.4.4 $ Is Additive (Prepend, Not Replace)

kaiv's `$` is always additive — it never replaces a sigil. The full container type is always visible in the reference:

- `$.name` → scalar (no container sigil)
- `$@.name` → array (`@` is preserved)
- `$/.name` → namespace (`/` is preserved)

The triple character sequence `$@.` is three distinct pieces of information: `$` ("look up and substitute"), `@` ("the thing being looked up is an array"), `.` ("the name is hidden, elided from canonical output"). This contrasts with Perl's `$array[0]`, where `@` is replaced by `$` and the container type is lost in the reference.

The discriminant between hidden variables and visible data fields is the `.` (dot) immediately after `$` (or after `$@`/`$/`):

- `.` present → hidden variable (elided from both `.raiv` and `.daiv`)
- `.` absent → data field reference (preserved in `.raiv`, resolved in `.daiv`)

The `$` operator is always additive. The container sigil (`@`, `/`) is always preserved. The hidden marker (`.`) is always visible. Three orthogonal pieces of information, three separate characters, no ambiguity.

1.5 Ordered Keys

Key ordering is significant in kaiv. This is a deliberate choice of strictness over looseness, with consequences throughout the pipeline.

Variable expansion is well-defined. `$.name` in a value can only refer to names defined on previous lines. There are no forward references, no circular dependencies, and no resolution-order ambiguity. The rules are simple and local. This strict ordering is what makes variable interpolation resolvable in a single left-to-right pass with no dependency-graph construction.

Streaming validation stays single-pass. The compiled schema parallel scan processes entries in document order and never needs to look ahead or revisit prior tokens. This is what makes the O(N) linear validation structure (described under §7 below) possible.

Diffing and merging are meaningful. Two kaiv documents with the same fields in different orders are different documents. This makes kaiv documents suitable for version-controlled storage with meaningful diffs.

Target formats that do not care simply ignore the ordering. Conversion *from* kaiv preserves order; conversion *to* kaiv from an unordered format requires a canonical ordering rule. Acceptable rules include: alphabetical order, schema-definition order, or original-definition order (for formats that have one). The choice of ordering rule is part of the interchange profile for that target format.

2 Level 0: Scalars

Level 0 covers the per-line concerns: declarations, variables, type annotations, provenance, scalar `key=value` lines, and the constraint forms applied within type annotations. Everything in Level 0 is processable by a minimal DFA in constant memory.

2.1 Declarations

Declarations are commands that apply to the entire kaiv text. They MUST be placed at the top of the text, before any content lines. Declarations use the `.!` sigil (or `.?` for provenance source declarations).

2.1.1 Declaration Inventory

The complete set of declaration keywords, across all file types, is:

Keyword	Syntax	File types	Defined in
`.!kaiv`	`.!kaiv [VERSION]` (bare form means version 1)	`.kaiv` (optional — absent means authored `.kaiv` version 1)	§2.1.2
`.!raiv`	`.!raiv [VERSION]` (bare form means version 1)	`.raiv` (always present — emitted by the Compiler; §2.1.2)	§2.1.2
`.!daiv`	`.!daiv [VERSION]` (bare form means version 1)	`.daiv` (always present — emitted by the Denormalizer; §2.1.2)	§2.1.2
`.!saiv`	`.!saiv [VERSION] ID-OR-URL [strict]` (bare form means version 1)	`.saiv`	§2.6.13; strict modifier in §11
`.!csaiv`	`.!csaiv [VERSION] ID-OR-URL [strict]` (bare form means version 1)	`.csaiv` (always present — emitted by the schema compiler; §2.1.2)	§2.1.2
`.!taiv`	`.!taiv [VERSION] LIBRARY-ID` (bare form means version 1)	`.taiv`	§2.6.13
`.!faiv`	`.!faiv [VERSION] LIBRARY-ID` (bare form means version 1)	`.faiv`	§2.7.9

Keyword	Syntax	File types	Defined in
`.!maiv`	`.!maiv [VERSION]` (bare form means version 1; no identity token — a mapping's identity is its endpoint pair, §8.2)	`.maiv`	§8.1
`.!source`	`.!source ID-OR-URL`	`.maiv`	§8.1
`.!target`	`.!target ID-OR-URL`	`.maiv`	§8.1
`.!via`	`.!via MAP-ID`	`.maiv`	§8.6
`.!drop`	`.!drop NAMEPATH`	`.maiv`	§8.1
`.!schema`	`.!schema:ID` / `.!schema ID-OR-URL` / `.!schema:/ns ID-OR-URL` / `.!schema:/@arr ID-OR-URL`	`.kaiv`, `.raiv`, `.daiv`, `.saiv` (inheritance)	§2.1.3; §3.5.6
`.!types`	`.!types LIBRARY-ID`	`.saiv`, `.taiv`, `.kaiv`; resolved at build time — does not survive into canonical output (canonical type names are fully qualified)	§2.6.13
`.!units`	`.!units LIBRARY-ID`	`.kaiv`, `.saiv`, `.taiv`; survives into canonical output (§2.7.10)	§2.7.10
`.!verbatim`	`.!verbatim` (no arguments; at most once)	`.kaiv`, `.raiv`; carried into `.raiv` by the Compiler, discharged by the Denormalizer — does not survive into `.daiv`	§2.5.6
`.!registry`	`.!registry prefix=base_url`	all authored file types; SHOULD survive into `.daiv` (§2.1.4)	§2.1.4
`.!provenance`	`.!provenance:LEVEL`	`.saiv`, `.csaiv`	§2.4.3
`.!ref`	`.!ref:alias schemapath`	reserved — corpus spec	§6
`.!compose`	`.!compose:NAMEPATH SCHEMA JOIN`	reserved — corpus spec	§6

Keyword	Syntax	File types	Defined in
`.!msaiv`	`.!msaiv [VERSION] CORPUS-ID` (bare form means version 1)	reserved — corpus spec	§6
`.!bind`	`.!bind:PATTERN SCHEMA`	reserved — corpus spec	§6
`.!unique`	`.!unique:PATTERN NAMEPATH`	reserved — corpus spec	§6
`.!fk`	`.!fk:PATTERN NAMEPATH TARGET:: NAMEPATH`	reserved — corpus spec	§6
`.?<id>`	`.?id uri`	`.kaiv`, `.raiv`, `.daiv`	§2.4

This table is the authoritative recognizer set: a lexer classifies a line as a declaration iff it begins with `.!` or `.?` (rule 4 of the Six Rules), and a `.!` line whose keyword is not in this table is an `INVALID_DIRECTIVE_ERROR` (see §11.1). Which keywords are *meaningful* varies by file type per the table; a keyword valid in the grammar but out of place for the file type is diagnosed by the consuming stage, not the lexer.

2.1.2 Format Declaration

The **format declaration** names the file's kind and, optionally, the kaiv text version. The declaration keyword mirrors the file extension — every kind in the family is self-describing by its first line, with no filename or out-of-band context needed (`.qaiv`, whose design is unfrozen and outside this document, will pin its declaration when it freezes):

File	Declaration	Presence
`.kaiv`	`.!kaiv [VERSION]`	OPTIONAL (absent means `.kaiv` version 1)
`.raiv`	`.!raiv [VERSION]`	always present (emitted by the Compiler)
`.daiv`	`.!daiv [VERSION]`	always present (emitted by the Denormalizer)
`.saiv`	`.!saiv [VERSION] ID [strict]`	REQUIRED
`.csaiv`	`.!csaiv [VERSION] ID [strict]`	always present (emitted by the schema compiler)
`.taiv`	`.!taiv [VERSION] LIBRARY-ID`	REQUIRED
`.faiv`	`.!faiv [VERSION] LIBRARY-ID`	REQUIRED
`.maiv`	`.!maiv [VERSION]`	REQUIRED (bare form means version 1)
`.msaiv`	`.!msaiv [VERSION] CORPUS-ID`	REQUIRED (reserved — corpus spec)

The version is optional for every kind, and the bare form is canonical: a versioned declaration naming version 1 (`.!kaiv 1`, `.!kaiv 1.0`, `.!kaiv 1.0.0`, `.!saiv 1 acme/x`) is equivalent authored input, and the build pipeline emits the bare form. The version token exists for forward evolution only; a future major revision would be a separate specification in any case, free to decide its own marker. The identity-carrying kinds (`.!saiv`, `.!csaiv`, `.!taiv`, `.!faiv`, `.!msaiv`) carry a library or schema identity after the keyword, and the declaration is split by token shape: a digit-first first token is the version (validated against the version

syntax below); any other first token is the identity, and the version is 1. The two can never collide — a version is digits and dots only, while an identity is a library path whose first segment starts with a letter (`library-path`, §10.1) or an absolute URL. `.!maiv` carries no identity token of its own — a mapping's identity is its endpoint pair (§8.2) — so its bare form is the keyword alone.

The format declaration, when present, MUST be placed on the first line of the kaiv text — or, when an optional shebang line is present (§12.5), on the first line after it.

In authored `.kaiv` data files the format declaration is OPTIONAL: a file that carries no format declaration is read as authored `.kaiv`, version 1, exactly as if it opened with `.!kaiv`. The default is deliberate — it keeps plain `KEY=value` files, including any well-formed `.env` file, valid kaiv documents as-is, with no header required. The one caveat is `$`: a bare dollar in a value is reference syntax at compile time (§2.5.3), so a `.env` whose values carry dollars lexes as-is but compiles only with a `.!verbatim` declaration prepended (§2.5.6).

Canonical files always carry their kind declaration explicitly, and the pipeline rewrites the keyword at each stage: the Compiler MUST emit `.!raiv` as the first line of `.raiv` output, the Denormalizer MUST emit `.!daiv` as the first line of `.daiv` output, and the schema compiler MUST emit `.!csaiv` (preserving the identity and any `strict` modifier from the `.!saiv` header, with a version-1 header normalized to the bare form and an explicit non-1 version preserved) as the first line of `.csaiv` output. Each emits its keyword whether or not the source document carried a declaration of its own. Consumers of canonical files are strict: a stream presented as a canonical kind whose first line is missing the matching declaration — or carries a different kind's — raises `FORMAT_KIND_ERROR` (§11.1). The same gate applies to the REQUIRED-header authored kinds: a `.saiv` / `.taiv` / `.faiv` / `.maiv` stream whose first line is missing its declaration — or carries another kind's — raises the same error in its consuming stage. This is what makes validation context-free: `.kaiv` and `.daiv` can no longer be confused for one another, because only the former may omit the declaration and each names its own kind. Canonical excerpts elsewhere in this document sometimes omit the format declaration when illustrating individual lines or isolated constructs; a complete canonical document always opens with it.

The version, when given, is one to three dot-separated decimal integers — `major`, `major.minor`, or `major.minor.patch` — matching `^[0-9]+(\.[0-9]+){0,2}$`. Omitted components are zero: `1`, `1.0`, and `1.0.0` name the same version, and all three canonicalize to the bare form — which is why the examples in this document write bare declarations throughout. The same version syntax applies to the identity-carrying declarations.

2.1.3 Schema Declaration

The **schema declaration** uses the `.!schema` command followed by the schema reference. Schema references fall into two categories: (1) a kaiv schema registry reference, and (2) a custom URL. The two categories each use a different syntax.

A kaiv schema registry reference appends a schema identifier to the `.!schema` command, separated by a colon. For example:

```
.!schema:acme/test-one
.!schema:acme/api-request
```

A custom URL schema reference is a space-separated URL on the `.!schema` line:

```
.!schema https://example.org/schema.saiv
.!schema https://example.org/schema.csaiv
```

Note that the custom URL schema reference MUST include the file extension: these are concrete *file references*. The kaiv schema registry reference, on the other hand, returns the compiled schema (`.csaiv`) by default; content negotiation with the `kind` media-type parameter selects another kind — `Accept: application/kaiv;kind=saiv` requests the authored schema (§12.4.2).

An **encapsulated schema reference** scopes the hub schema's fields under a sub-namespace rather than merging them at root. The colon after `.!schema` introduces a namespace qualifier using the / namespace prefix:

```
.!schema:/server hub/server-endpoint
.!schema:/auth hub/credentials
```

This places `hub/server-endpoint` fields under the `/server` namespace and `hub/credentials` fields under the `/auth` namespace. The same hub may be encapsulated multiple times under different namespaces:

```
.!schema:/upstream hub/server-endpoint
.!schema:/downstream hub/server-endpoint
```

URL references also accept the namespace qualifier:

```
.!schema:/server https://example.org/schema.csaiv
```

Flat extension (fields at root) and encapsulated extension (fields under a namespace) may be combined freely in the same kaiv text.

2.1.4 Type Registry Resolution

All named type references (`!library/path/typename`) and type library imports (`.!types`) implicitly resolve through `ktaiv.com`. This works for the public ecosystem but does not cover air-gapped environments, IP-protected enterprise types, regulatory data sovereignty, or builds that need to be independent of `ktaiv.com` uptime. The solution is a **layered resolution model**: four resolution layers evaluated in priority order so that the most specific configuration wins.

Layer	Mechanism	Priority	Internet required?
1	`.!registry` declaration in the document	Highest	No
2	Build-time configuration (`kaiv.kaiv` / environment variables)	High	No
3	Registry redirect aliasing (HTTP 301/302 from `ktaiv.com` for types, `ksaiv.com` for schemas)	Low	Yes
4	Default registry (`ktaiv.com` for `.taiv`; `ksaiv.com` for `.saiv`/`.csaiv`/`.maiv`)	Lowest	Yes

Layer 1: `.!registry` declaration. A `.!registry` declaration in the document maps a library-path prefix to an alternative base URL. It is a document-level declaration — placed after the format declaration, before content lines. Declarations pass through to canonical form (the format declaration itself is rewritten to the output kind's keyword, §2.1.2), so a `.daiv` carrying registry-resolved identities reads:

```
.!daiv
.!registry acme=https://types.acme.com
.!registry internal=https://types.internal.corp.net
.!schema:acme/server-config
!acme/ourtypes/customerid'::owner=CUST-001
!internal/auth/token'::session=abc123
```

Rules:

- Syntax: `.!registry prefix=base_url` where `prefix` is matched against the **first path segment** of `!library/path/typename`. When matched, the base URL replaces `ktaiv.com` and resolution becomes `{base_url}/{library/path}.taiv`.
- A base is an absolute `http(s)` URL or a **filesystem path** — absolute, or relative to the directory containing the document — exactly as in `kaiv.kaiv` (Layer 2 below); the relative form is what lets a document vendor its own types offline (conformance vector `valid/045`).
- Multiple `.!registry` declarations are allowed, one per prefix.
- `.!registry` declarations SHOULD survive into `.daiv` (they are resolution metadata, like `.!schema`). They MAY be omitted if all types are baked into the `.csaiv` constraints.
- Layer 1 is the most explicit and auditable mechanism — you can see exactly where types come from. It is also the one layer authored by the document's *producer*; see the trust model (§2.1.4).

Layer 2: Build-time configuration. A `kaiv.kaiv` file (project-level, at the project root) maps prefixes to alternative registries with no impact on the document format. The toolchain's own configuration is a kaiv document:

```
# kaiv.kaiv
.!kaiv

/registries::acme=https://types.acme.com
/registries::internal=https://types.internal.corp.net
/registries::default=https://ktaiv.com
```

The file is deliberately restricted to the **Level 0–1 scalar subset**: flat `key=value` fields under the single `/registries` namespace (namespaces and `::` projection are Level 1 constructs), no schema, no type annotations, no named-type references. This is what makes the bootstrap sound — a kaiv processor parses `kaiv.kaiv` with the core Level 0–1 pipeline *before* any type resolution exists, so the configuration that drives resolution never needs resolution itself. A registry prefix containing characters outside the bare-name grammar (`-` is common in `path-seg` prefixes) is written as a quoted name: `/registries::"acme-corp"=https://types.acme-corp.com`.

A base value is an absolute `http(s)` URL, or a **filesystem path** — absolute, or relative to the directory containing `kaiv.kaiv` — for air-gapped and local-tree resolution. Resolution appends `{library/path}.taiv` to the base either way: `/registries::acme=./types` resolves `!acme/ourtypes/customerid` against `./types/acme/ourtypes.taiv`. The reserved key `default` overrides the Layer 4 default registry for unmatched prefixes. The same map resolves **every** registry-shaped identifier, with the appended extension determined by the consumer: type references and `.!types` imports append `.taiv`, `.!units` imports append `.faiv` (§2.7.10), and `.!schema` IDs append `.saiv` when the schema compiler consumes them (inheritance, §7.3) and `.csaiv` when the Denormalizer/Validator does — the conformance suite exercises all three.

Environment variables override the file:

```
KAIV_REGISTRY_ACME=https://types.acme.com
KAIV_REGISTRY_INTERNAL=https://types.internal.corp.net
KAIV_REGISTRY=https://types.internal.corp.net   # default override for
   unmatched prefixes
```

Naming convention: `KAIV_REGISTRY_{PREFIX}` (prefix uppercased) for per-prefix override; `KAIV_REGISTRY` for the default. Air-gapped environments change configuration without touching any source file. The `.daiv` file is identical regardless of which registry was used — type identity is a logical path, independent of the resolution mechanism.

Layer 3: Registry redirect aliasing. `ksaiv.com` and `ktaiv.com` support HTTP 301/302 redirects. A registered namespace prefix configured by its owner can transparently redirect resolution requests to the owner's own server:

```
!acme/ourtypes/customerid
  → GET ktaiv.com/acme/ourtypes.taiv
  → HTTP 301 Moved Permanently → https://acmekaiv.example.com/types/acme/
   ourtypes.taiv
  → fetches .taiv from owner's server
```

The canonical type path is unchanged — the redirect is transparent to type identity. The registry acts as a **naming authority**: it validates prefix ownership and ensures global uniqueness of library paths. Implementations SHOULD cache resolved `.taiv` files keyed by their canonical URL.

Layer 4: Default registry. The kaiv registries are split by artifact kind: `ktaiv.com` hosts type libraries (`.taiv`); `ksaiv.com` hosts schemas (`.saiv`, `.csaiv`) and mappings (`.maiv`). The file extension discriminates artifact kind and selects the registry.

Artifact	Extension	Resolution path
Type library	`.taiv`	`ktaiv.com/{library/path}.taiv`
Schema	`.saiv`	`ksaiv.com/{schema/path}.saiv`
Compiled schema	`.csaiv`	`ksaiv.com/{schema/path}.csaiv`

For example, `!std/net/port` resolves to `ktaiv.com/std/net.taiv`, and the schema `acme/server-config` to `ksaiv.com/acme/server-config.saiv`.

Implementations MUST support this layer. `std/core` is an exception — implementations SHOULD ship it bundled (embedded) so it is always available, even offline.

Alpha hosting. Through the `1.0-draft` series of this specification the reference hosts are the `kaiv.io` registry subdomains — `t.kaiv.io` (types), `s.kaiv.io` (schemas), `f.kaiv.io` (units) — and *no* eternalink permanence is promised: alpha artifacts (`std/net` included) may still change, and the alpha registries may be reset. The `k*aiv.com` production domains named throughout this specification activate at beta, and the write-once, read-forever contract begins there. The default-host table in a conforming implementation is the single switch point.

Type identity vs. type resolution. Type identity is a logical path; type resolution is a deployment concern. Two `.daiv` files with the same data have the same canonical lines regardless of which resolution layer was used to fetch the source `.taiv`:

```
# Layer 1 resolution (fetches from acme.com):
.!registry acme=https://types.acme.com
```

— or, with no `.!registry` declaration at all, Layer 4 resolution fetches the same library from `ktaiv.com`. Both produce the identical canonical line:

```
!acme/ourtypes/customerid'::owner=CUST-001
```

The certified runtime never resolves type names — type resolution is build-time only. The integrity check reads `.daiv` and `.csaiv` (which carries lowered constraints), not `.taiv`.

A future extension reserves the design space for **DNS-based authority**: if the first path segment contains a `.`, it is treated as a domain that directly serves the type (`!acme.com/ourtypes/customerid`). This is not specified in the current version but is documented to prevent future path patterns from accidentally closing it off.

Trust model and strict resolution. The resolution layers differ in *who authors them*. Layer 2 is the consumer's own configuration; Layers 3–4 are the registry authority. Layer 1 alone is authored by the document's *producer* — and it outranks every other layer. A document can therefore pin a well-known schema identity while redirecting its resolution: one declaring both `.!schema:hub/invoice` and `.!registry hub=https://evil.example` validates against whatever the declared base serves, while appearing to validate against the registry's `hub/invoice`. Identity is unaffected (a logical path, above), but the *content* bound to that identity is chosen by the producer. An untrusted document must not be allowed to choose where its own contract comes from.

Implementations MUST therefore provide a **strict resolution mode**, off by default: when enabled, a `.!registry` declaration that would determine the base of any resolved artifact raises `RegistryStrictError` instead of resolving. Dormant declarations — prefixes no resolved reference uses — are not an error, since declarations survive into canonical form. Layer 2 not only remains in effect under strict mode — it *shadows* Layer 1 for every prefix it covers (exact entry or `default`), so only an uncovered Layer 1 win is refused. Vendoring the affected prefix in `kaiv.kaiv` is thus the sanctioned way to validate such a document: the consumer's configuration decides, and the document's declaration

becomes dormant. Implementations SHOULD also report **resolution provenance**: for each resolved artifact, the reference, the winning layer, and the resolved base and location.

Implementations SHOULD additionally provide a **canonical resolution mode**: Layers 1–2 are ignored entirely and every artifact resolves through the Layer 4 default registry — no document declaration, configuration file, or environment variable can redirect resolution. Where strict mode protects a consumer from a document's overrides, canonical mode asserts that resolution depends on nothing but the reference and the canonical hosts. It subsumes strict mode: a `.!registry` declaration is dormant by construction, so nothing is refused.

The reference toolchain exposes strict mode as `--registry-strict` / `KAIV_REGISTRY_STRICT=1`, canonical mode as `--registry-canonical` / `KAIV_REGISTRY_CANONICAL=1`, and provenance reporting as `-v`.

2.2 Comments

kaiv distinguishes two comment syntaxes with different semantic roles:

Syntax	`.kaiv`	`.saiv`	`.taiv`	`.faiv`	`.maiv`	`.daiv`	`.csaiv`	Semantic role
`#`	[x]	[x]	[x]	[x]	[x]	[ ]	[ ]	Human annotation — no semantic content
`//`	[ ]	[x]	[x]	[x]	[x]	SHOULD	SHOULD	Field/type/unit documentation — schema metadata ([ ]: classified but dropped like `#`; no doc semantics in data)

`#` comments never appear in canonical files. `//` doc strings SHOULD be carried into `.daiv` / `.csaiv` by the build pipeline and MAY be omitted for constrained deployments (§2.2.2).

2.2.1 General Comments (`#`)

`#` comments are allowed in all authored file types — `.kaiv` data files, `.saiv` schema files, `.taiv` type library files, `.faiv` unit definition files, and `.maiv` mapping files. The Lexer emits them as `COMMENT` tokens but filters them out before the Parser stage. They carry no semantic content and have no effect on the AST, schema validation, or canonical output. The pipeline never emits them into `.daiv` or `.csaiv`; a consumer encountering one there (a hand-maintained artifact) classifies and discards it like any comment line — the Six Rules are universal (§1.3.1).

```
# This is a general comment — filtered before the Parser stage
host=localhost
```

2.2.2 Documentation Comments (//)

`//` doc comments are **meaningful only in the definition-bearing file types** — `.saiv` schema files, `.taiv` type library files, `.faiv` unit definition files, `.maiv` mapping files, and reserved `.msaiv` metaschemas — never in `.kaiv` data files. Documentation is a schema/type/unit concern, not a data concern: "what does this field mean?" is answered by the schema, not by the data instance. Every peer format takes the same position: ProtoBuf comments document `.proto` schema fields, Avro `"doc"` is a schema property, GraphQL `"""..."""` descriptions are on SDL, XSD `xs:annotation` is on schema elements, JSON Schema `"description"` is a schema keyword.

In `.kaiv` the six-rule classifier still produces a `DOC` token — the rules are universal (§1.3.1) — but the token carries no documentation semantics there: the Compiler MUST drop it exactly as it drops a `#` comment. Doc strings acquire meaning only in the definition-bearing file types.

The Parser associates each `DOC` token with the immediately following field or type definition line.

```
// The hostname for the primary database connection
host=
```

```
// Port number in the range -065535
/^-?[0-9]+$/ ..num [0,65535]
&port=
```

In `.saiv`, doc comments document field definitions; in `.taiv`, they document type definitions. The documentation string is distinct from the field's value and is not part of the data payload. Avro's `"doc"` property maps directly during Avro schema generation, enabling round-trip fidelity.

Documentation in canonical form. The build pipeline SHOULD include `//` doc strings from the schema in `.daiv` and `.csaiv`. Implementations MAY omit them for constrained deployments (safety-critical ECUs, bandwidth-limited buses). When present, the Validator skips them during validation — they are metadata that has no effect on the parallel scan, type checking, or constraint evaluation. When omitted from `.daiv`, the documentation remains recoverable from the `.saiv`/`.csaiv`/`.taiv` source.

Two deployment profiles are valid:

- **Rich deployment:** `//` doc strings included in `.daiv` and `.csaiv` for tooling, format conversion, and developer inspection.
- **Minimal deployment:** `//` doc strings omitted from `.daiv` for constrained environments. The runtime integrity check never uses `//` lines; documentation remains recoverable from the schema source.

2.3 Quoted Names

kaiv's identifier system rests on a single principled boundary: **bare names are POSIX identifiers; quoted names exist for interchange with formats whose identifier rules differ from POSIX.** No widening of the bare-name alphabet for convenience.

2.3.1 What Gets Quoted

Quoting applies to **individual names** — the atomic identifiers between path operators. Never to operators, never to entire namepaths.

Namepath component	Quotable?	Example
Name (between / operators)	[x]	`"Content-Type"`
Field (after :: operator)	[x]	`::"Accept-Language"`
Array name (after @)	[x]	`@"x-items"`
Path operators (/, ::, @)	[ ] Never	Always bare
Entire namepath	[ ] Never	Always composed of individually-quoted-or-bare segments

2.3.2 When to Quote

A name MUST be quoted if and only if it is not a valid POSIX identifier:

```
bare-name = ( ALPHA / "_" ) *( ALPHA / DIGIT / "_" )
```

If a name matches `bare-name`, it MUST NOT be quoted in canonical form. If it doesn't, it MUST be quoted. This makes quoting **deterministic** — there is exactly one canonical representation for any namepath. No ambiguity, no stylistic choice. **Index segments are outside this rule**: an element index (`/@servers/0`, `::0`) is its own segment kind (`index` in §10), always unquoted; a *quoted* all-digit segment (`"0"`) is an ordinary quoted name — a map key or field name — and never addresses an array element. Authored text MAY quote a bare-able name — necessary when the bare spelling is positionally reserved, e.g. a schema field named `re` (§2.6.11) — and the Compiler normalizes it to the bare spelling on canonicalization.

2.3.3 Quoted Name Rules

1. A quoted name is enclosed in double quotes (").
2. A double quote character *within* a name is represented as `""` (repeated double quote).
3. All other characters are literal — no backslash escapes, no `\n`, no `\t`. This preserves kaiv's no-escape-sequences principle; `""` is not an escape sequence, it's a doubling convention (same mechanism as SQL identifiers and CSV fields).
4. A quoted name MUST contain at least one character (empty quoted names are not allowed).

2.3.4 Examples in Canonical Form

```
!str'::"Content-Type"=application/json

!str'/app/"dark-mode"::enabled=true

!int'/@"x-servers"/0::"retry-count"=3
```

```
!str'::"weird""name"=value with a literal quote in the key
```

The last example: the name is `weird"name` where the `""` represents a single `"` character within the quoted identifier.

2.3.5 DFA Impact

Zero structural change. The Lexer sees `"` at the start of a name position and enters a quoted-name sub-state. Inside that state, `""` produces a literal `"` and a lone `"` terminates the name. The state machine has three states (START → QUOTED → MAYBE_END). No stack, no lookahead beyond one character. The grammar remains regular.

2.4 Provenance

Provenance records where a value came from and when it was observed. It is an optional prefix on data lines, written before the `'` delimiter.

2.4.1 Provenance Syntax

The full provenance prefix grammar is:

```
provenance_list = '?' provenance (',' provenance)*
provenance      = id ('@' instant)? ('#' dpid)?
id              = identifier
instant         = YYYY-MM-DD 'T' HH:mm:SS 'Z'   (20 chars, canonical)
                | YYYYMMDD 'T' HHmmSS 'Z'       (16 chars, deprecated)
dpid            = identifier
```

A single provenance entry carries a mandatory source id and two optional qualifiers:

1. `?sourceID` — the source identifier (a short name declared via `.?id uri` at file top, resolving to a URI, sensor ID, URN, or other opaque string).
2. `@instant` — the temporal qualifier recording *when* the value was observed. The canonical form is dashed extended ISO 8601 UTC, `YYYY-MM-DDTHH:MM:SSZ` (20 characters): `?sensor1@2026-07-28T14:00:00Z`. Producers MUST emit this form.
   The compact basic form `YYYYMMDDTHHmmSSZ` (16 characters) remains an accepted *parse* form through the `1.0-draft` series and is **deprecated**: consumers MUST accept it, producers MUST NOT emit it, and it is removed at 1.0. A document carrying the compact form canonicalizes to the dashed form.

   > The dashed form introduces `:` into the provenance run, where `:` elsewhere in an annotation opens the unit sigil. The two never collide: a provenance run is introduced by `?` and consumes to whitespace or `'`, so every `:` inside it belongs to the instant. This is the same disambiguation-by-introducer rule that lets `#` mean *data point identifier* here and nothing else.

3. `#dpid` — the **data point identifier**: an optional stable identifier assigned to this data point. Any valid identifier is accepted; the identifier is not required to be a UUID. In

manually-authored `.kaiv` files a user may write `#request-42` or `#row-17`; the `chraiv.com` ingestion pipeline auto-assigns UUID values (`#uuid`) at ingest time.

The timestamp and data-point qualifiers are optional and independent; the source id anchors every entry — a qualifier cannot appear without one (the grammar above; §10). Provenance answers *who says so*: the qualifiers refine that attribution, so an attributor-less timestamp is not a provenance form (a document that genuinely has no source names a trivial one). The full canonical metadata prefix when all three are present is:

```
!type?sourceID@timestamp#dpid'namepath=value
```

2.4.2 Disambiguation from Length Constraints

The `#` sigil is used in two distinct positions:

1. **Type-annotation position** (after a type or pattern constraint, before `'`) — denotes a **length constraint**: `#[min,max]` or `#{a,b,c}`. See §2.8.2.
2. **Provenance position** (after `@timestamp` or after `?sourceID` when no timestamp is present, still before `'`) — denotes the **data point identifier**: `#dpid`.

The DFA distinguishes the two uses by position: a `#` encountered while processing the provenance list (after `?` and any `@timestamp`) is always a data point identifier; a `#` encountered while processing a type annotation (after `!type` and any pattern or range constraints, with no intervening `?`) is always a length constraint. No lookahead or disambiguation table is required.

2.4.3 Requiring Provenance in Schemas

By default provenance is unconstrained: any data line MAY carry any subset of the provenance triple. A schema can make provenance part of the validation contract with the `.!provenance` declaration in the `.saiv` header:

```
.!saiv hub/log-entry
.!provenance:required
```

Three levels are supported:

Declaration	Meaning
`.!provenance:required`	Both `?sourceID` and `@timestamp` must be present on every data line
`.!provenance:source`	`?sourceID` required; `@timestamp` optional
`.!provenance:none`	Provenance prohibited (for schemas where it would be noise)

The declaration constrains only the source and timestamp components. The `#dpid` component is always optional and is never required or prohibited by `.!provenance`. A violation is a `ProvenanceSchemaError`, detected by the Validator against the compiled schema, not a lexer error — the lines are syntactically well-formed either way. The `required` and `source`

levels are **incompatible with optional fields**: materialization (§2.6.17) synthesizes a `.daiv` line for every absent optional field, and a synthesized line carries no provenance — the pipeline could never produce a valid artifact. The schema compiler MUST reject the combination at compile time (`ProvenanceSchemaError`), the same static-rejection posture as `SchemaOptionalWithoutDefaultError`. Because the Validator reads the `.csaiv`, not the `.saiv`, the schema compiler propagates the `.!provenance` declaration verbatim into the `.csaiv` header, exactly as it propagates the `strict` modifier (§11). A `.csaiv` with no `.!provenance` header imposes no provenance requirement.

2.5 Variables and Field References

Variable interpolation enables **DRY composition** in authored kaiv text. Variables are temporary named values — scalars, arrays, or namespaces — defined with dot-prefixed hidden names and referenced with the `$` dereference operator in values. They are resolved during Compiler canonicalization and completely elided from the canonical output. In addition to hidden variable references, kaiv supports **field references** that reference previously-defined data fields — distinguishable from variable references by the absence of `.` after `$`. A document can opt out of this machinery wholesale with the `.!verbatim` declaration (§2.5.6), which makes values fully verbatim — every `$` literal, no references.

2.5.1 The Variable Name System

Variable names use a **dot prefix** (`.name`) — following Unix hidden-file convention (`.bashrc` is present but hidden from `ls`). The dot is part of the name itself. Structural sigils (`@`, `/`) come before the dot-name because they describe the container type, not the name's visibility.

On the **definition side** (left of `=`), no `$` — POSIX-aligned (shell: `HOST=value`):

Definition syntax	Reference syntax	Contains	Available at Level
`.name=value`	`$.name`	scalar	Level 0
`@.name+=value` / `@.name;=a;b`	`$@.name`	array	Level 1
`/.name:=host=a|port=1`	`$/.name`	namespace	Level 1

On the **reference side** (right of `=`, inside values), `$` dereferences the hidden name — POSIX-aligned (shell: `$HOST`). The `@` or `/` structural sigil follows `$` and precedes the dot-name to identify the container kind.

The dot distinguishes a hidden name from data. `.host` is a hidden scalar; `@.ports` is an array with a hidden name; `/.base` is a namespace with a hidden name. No `$` on the left side — `$` is purely a value-side dereference operator. The Parser knows at definition time that dot-prefixed names are scaffolding and elides them from canonical output.

2.5.2 Namespace-Variable Splat

Scalar and array variable references are substituted *inside values* — scalars as text; an array variable holds elements, not text, so `$@.name` is a **splice**, legal in exactly one position: as

the **entire right side** of an append (`+=`) or extend (`;=`) line — targeting a visible array (`/@mirrors;=$@.ports`) or another hidden array (`@.all+=$@.ports`). Expansion is **element-wise**: each element of the variable appends one element (one canonical line per element on a visible target). In any other position — mid-value, a scalar assignment, a struct-pair value — the reference has no text representation and is a `VariableContextError` (§11).

A namespace variable holds pairs, not text, so its reference `$/.name` is not a value substitution — it is a **splat**: the variable's pairs are expanded as if they had been written out at the reference point. The splat form appears in exactly two positions:

1. **As the entire right side of a struct assignment** (`:=` or `+:=`): `/server/api:=$/.base` expands to the same lines as writing the variable's pairs inline; the target may itself be a hidden namespace variable (`/.derived:=$/.base`) (`/server/api:=host=localhost|ssl=true`). Subsequent lines may override individual expanded fields (last-write-wins, §2.5.4).
2. **As a standalone line inside an open section or namespace block**: each of the variable's pairs is expanded as if written as a `key=value` line at that point in the block. In the line classifier, such a line has no `=` and is recognized within rule 6 by its `$/.` leader (authored `.kaiv` only).

A namespace-variable reference in any other position — inside a scalar value, mixed with other text, or as a `pair`'s value — is a `VariableContextError` (§11.2): a namespace variable has no text representation to substitute.

2.5.3 Field References

In addition to hidden variable references (`$.name`), kaiv supports **field references** that reference previously-defined **data fields** — visible, schema-defined fields, not hidden variables.

Reference	Syntax	Discriminant from variables
Root-level field	`$field`	No `.` after `$` (variables always have `.` after `$`)
Namespaced field	`$path::field`	Contains `::`

The discriminant is unambiguous: the presence of `.` immediately after `$` (or after `$@`/`$/`) identifies a hidden variable reference. The absence of `.` — a bare name or a path with `::` — identifies a field reference.

Token boundaries. A reference embedded in a value ends deterministically (`field-ref` in §10): the token is the longest run of *reference characters* — `ALPHA`/`DIGIT`/`_`, the separators `/` and `::`, and `@` where a step may begin (token start or after `/`) — with a dangling trailing separator excluded from the token. The first character outside the set ends the reference; the rest of the value is literal text (in `log=$dir/today.txt` the token runs to `$dir/today`, which is not a well-formed field reference — no `::` projection — so the line fails rather than referencing `$dir`; separators bind tighter than intuition suggests, so restructure when in doubt, e.g. define a variable). Note that digits are reference characters,

since indices are addressable: `price is $5 off` reads `$5` as a root-field reference and is always an `UndefinedReferenceError` — a bare name cannot be all digits, so no root field `::5` can exist — **a literal dollar is always written** `$$` (§9.5) — except in a verbatim document (§2.5.6), where every `$` is literal and `price is $5 off` means what it says. A reference without a leading `/` is the same path with one (`$server::host` ≡ `$/server::host`). Quoted names cannot appear in a reference: a field whose namepath requires quoting is not addressable by reference.

Behavior in Each Canonical Form

Construct	In `.raiv`	In `.daiv`
Variable definitions (`.name=`)	Elided	Elided
Variable references (`$.name`)	Resolved	Resolved
Field references (`$path::field`)	**Preserved**	Resolved (inlined)
Data fields	Preserved	Preserved
Type annotations	Fully qualified	Fully qualified

Resolution rules. Same left-to-right, no-forward-references rule as variables. The Compiler maintains a **field table** alongside the variable table. Every data field that is emitted into the output is added to the field table (keyed by fully-qualified namepath). When a field reference `$path::field` is encountered in a value:

- The fully-qualified namepath is constructed (applying any active namespace block prefix).
- The field table is consulted. If the field was previously emitted, its value is available.
- In `.daiv`: the referenced value is inlined (denormalized).
- In `.raiv`: the reference is preserved as `$path::field`, with the **fully-qualified** namepath substituted — a reference authored inside a namespace block (`/server`) as `$host` is emitted as `$server::host`. Block scoping is an authoring construct; it cannot survive into `.raiv`, so the reference is qualified even though the referenced value is not yet inlined.

Forward references and circular references are impossible by construction — the left-to-right rule ensures a field is always defined before it can be referenced.

2.5.4 Resolution Rules

1. **Document-scoped.** Variables are visible from their definition to the end of the kaiv text.
2. **Left-to-right resolution.** A variable can only reference variables defined on previous lines. No forward references, no circular dependencies.
3. **Templates can reference templates.** A `/.` namespace variable can include `$.name` references in its values, which must already be defined. Resolution is sequential substitution, not recursive expansion.

4. **Override semantics.** When a namespace variable is expanded and a subsequent line redefines a field, last-write-wins. This enables template-then-override patterns.
5. **Elision.** All dot-prefixed definitions are removed from the canonical output. Only schema-defined data fields survive canonicalization.

2.5.5 The "Almost Verbatim" Principle

kaiv's original design principle was that values are verbatim: what you write between = and end-of-line is exactly the value, with no escape sequences and no interpretation. Variable interpolation and field references introduce controlled exceptions:

— `$.identifier` (and `$@.identifier`, `$/.identifier`) in a value is expanded to the variable's value.
— `$field` or `$path::field` in a value is a field reference: expanded to the referenced field's value in `.daiv`, preserved verbatim in `.raiv`.
— `$$` produces a literal `$` character. Like `""` in quoted names, `$$` is not an escape sequence but a doubling convention — `\/` in regex patterns remains the sole escape sequence in the kaiv family.

The principle becomes: *values are verbatim except that `$.identifier` (variable) or `$field`/`$path::field` (field reference) is expanded (in `.daiv`) or preserved (in `.raiv`), and `$$` produces a literal `$`.* In `.daiv`, all references are resolved — values are truly verbatim with no special characters.

This section states the *default* mode — and the `.maiv` right-side grammar builds on its doubling convention (§8.3), so the rules above stand on their own. The original unconditional principle is recoverable per document: a `.!verbatim` declaration removes the exceptions along with the reference machinery.

2.5.6 Verbatim Documents (`.!verbatim`)

A document may opt out of the reference machinery wholesale. The `.!verbatim` declaration — no arguments, at most once, placed with the other header declarations (§2.1) — makes a **verbatim document**, whose values are *fully* verbatim:

— `$` is an ordinary literal character in value text.
— `$$` is not a doubling — it is two literal dollars.
— `$.name`, `$@.name`, `$/.name`, `$field`, and `$path::field` have no meaning; no `UndefinedReferenceError` is possible, at either stage.

What stands between = and end-of-line is exactly the value — the original verbatim principle (§2.5.5) without exceptions. A short pipeline run:

```
.!kaiv
.!verbatim
price=$5
pattern=^abc$
```

compiles to the `.raiv`

```
.!raiv
.!verbatim
!str'::price=$5
!str'::pattern=^abc$
```

and denormalizes to the `.daiv`

```
.!daiv
!str'::price=$5
!str'::pattern=^abc$
```

The `.daiv` is byte-identical to that of the default-mode document authoring the same values as `price=$$5` and `pattern=^abc$$`.

No reference machinery. A verbatim document declares that references do not exist, so defining or using the machinery is a contradiction, surfaced as a `VerbatimContextError` (§11.2). The ban is *positional* — it covers exactly the three positions where the machinery is recognized structurally, and never the content of value text:

1. a variable-definition line: `.name=`, `@.name+=` / `@.name;=`, or `/.name:=`;
2. a standalone namespace-variable splat line (`$/.name`, §2.5.2);
3. a compound-form right side that is entirely a splice: a `:=` / `+:=` right side `$/.name`, or a `+=` / `;=` value that is exactly `$@.name`.

A `$` *inside* value text is never this error: in a verbatim document `note=see $@.other` is a scalar whose value contains exactly those bytes. A whole value that would read as class 3 is authored through the standard escape hatch — restructure into the single-value `key=value` form (§9.5).

Survival. The declaration lives exactly as long as the machinery it disables. Field references are preserved in `.raiv` and resolved into `.daiv` (§2.5.3); correspondingly, `.!verbatim` is carried into `.raiv` by the Compiler — emitted within the leading declaration run, so `.raiv` values need no doubling either — and discharged by the Denormalizer: it does not appear in `.daiv`. A `.daiv` is already fully verbatim by kind (every reference resolved, every `$` literal), so the declaration would be inert there — and admitting it would give one data text two `.daiv` forms, which the content-addressed uses of `.daiv` exclude. A `.!verbatim` encountered in `.daiv` — or in any kind other than `.kaiv` / `.raiv` — is diagnosed by the consuming stage (§2.1.1).

Pipeline behavior. The mode switches the resolution sub-passes *off* rather than reconfiguring them: the Compiler emits values untouched and enforces the positional ban; the Denormalizer, seeing the declaration in the `.raiv` header, performs no `$$` collapse and no reference expansion. Both stages remain single-pass — declarations precede content (§2.1), so the mode is fixed before the first value in both `.kaiv` and `.raiv`. The Six Rules and the `'` split are untouched: the mode changes what a value *denotes*, never how a line is classified. When re-authoring `.kaiv` from canonical form, a formatter MAY target verbatim mode (emit `.!verbatim` and raw values) instead of the default `$$` doubling.

The same bytes, two readings. The mode is document-level, so identical authored bytes denote different values under different headers: `price=$$5` is the two bytes `$5` in default mode and the three bytes `$$5` in a verbatim document — and `price=$5`, an `UndefinedReferenceError` in default mode (§2.5.3), is simply `$5` in a verbatim one. Keep the header in view when copying lines between documents.

2.5.7 Worked Example — Variables

Authored kaiv

```
.!kaiv
.!schema:acme/cluster-config

# Template variables (elided from canonical output)
/.base_endpoint:=host=localhost|ssl=true
.default_timeout=30

# Data using templates
/server/api:=$/.base_endpoint
/server/api::port=8080
/server/api::timeout=$.default_timeout

/server/admin:=$/.base_endpoint
/server/admin::port=9090
/server/admin::timeout=$.default_timeout

[/@workers]
$/.base_endpoint
name=worker-1
port=7001
[/@workers]
$/.base_endpoint
name=worker-2
port=7002
[]
```

Canonical Output (`.daiv`)

```
.!daiv
.!schema:acme/cluster-config
!str'/server/api::host=localhost
!str'/server/api::ssl=true
!str'/server/api::port=8080
!str'/server/api::timeout=30
!str'/server/admin::host=localhost
!str'/server/admin::ssl=true
!str'/server/admin::port=9090
!str'/server/admin::timeout=30
!str'/@workers/0::host=localhost
!str'/@workers/0::ssl=true
```

```
!str'/@workers/0::name=worker-1
!str'/@workers/0::port=7001
!str'/@workers/1::host=localhost
!str'/@workers/1::ssl=true
!str'/@workers/1::name=worker-2
!str'/@workers/1::port=7002
```

All `$.name` references are resolved. All dot-prefixed definitions are elided. Overrides are applied (port values differ per namespace). The denormalized form is pure data.

2.5.8 Worked Example — Field References

Authored kaiv

```
.!kaiv
.!schema:acme/cluster-config

# Primary server (real data)
/server/api::host=localhost
/server/api::port=8080
/server/api::timeout=30

# Backup server (references primary via field references)
/server/backup::host=$server/api::host
/server/backup::port=9090
/server/backup::timeout=$server/api::timeout
```

Relational Output (`.raiv`) — Field References Preserved

```
.!raiv
.!schema:acme/cluster-config
!str'/server/api::host=localhost
!str'/server/api::port=8080
!str'/server/api::timeout=30
!str'/server/backup::host=$server/api::host
!str'/server/backup::port=9090
!str'/server/backup::timeout=$server/api::timeout
```

Denormalized Output (`.daiv`) — Field References Resolved

```
.!daiv
.!schema:acme/cluster-config
!str'/server/api::host=localhost
!str'/server/api::port=8080
!str'/server/api::timeout=30
!str'/server/backup::host=localhost
!str'/server/backup::port=9090
!str'/server/backup::timeout=30
```

In `.raiv`, the relational structure is preserved: `$server/api::timeout` makes the derivation relationship visible and auditable. In `.daiv`, the value `30` appears directly — the relationship is gone, but the file is self-contained and trivially validatable.

2.5.9 Architectural Impact

Variable interpolation and field references are an **authoring-layer concern**, with one exception: the Denormalizer's `$path::field` resolution is also an authoring/build-time concern, not a runtime one.

- **Compiler:** Gains variable resolution and field-reference preservation as sub-passes. Still single-pass left-to-right, still bounded memory. Produces `.raiv` (field references preserved).
- **Denormalizer:** Reads `.raiv` and expands `$path::field` references from the field table; when the data declares a schema, it also reads the `.csaiv`, materializes absent optional fields (§2.6.13), and lifts `!str` lines to the field's retained head type (§7.3) — for `!text` heads this is the guarded str→text coercion (§2.6.4) — and converts authored same-dimension units into the declared unit (§2.7.3). Produces `.daiv` (field references resolved, every schema-declared field present, schema types applied).
- **Validator:** Reads `.daiv` (no field references remain) and `.csaiv`. Performs schema constraint checking. Produces pass/fail.
- **Compiled schema (`.csaiv`):** Unchanged. Variables and field references are invisible to the schema — they are resolved before validation.
- **Denormalized form (`.daiv`):** No dot-prefixed definitions, `$.name` references, or `$path::field` references survive. Pure data.
- **Relational form (`.raiv`):** No dot-prefixed definitions or `$.name` references survive. `$path::field` field references are preserved verbatim.
- **Query language (qaiv):** Unchanged. Queries operate on `.daiv`, which contains no variables or field references.
- **Certification:** The variable and field-reference resolution sub-passes are deterministic, bounded, and non-recursive. They are part of the Compiler's scope (build-time only) and do not affect the Validator's parallel scan or the Lexer's certification.
- **Verbatim documents:** A `.!verbatim` declaration (§2.5.6) switches the resolution sub-passes *off* rather than adding work: the Compiler emits values untouched and carries the declaration into `.raiv`; the Denormalizer expands nothing and discharges it. The `.daiv` output is byte-identical to that of the escaped-authored equivalent.

2.5.10 Why `.raiv` Exists

The `.raiv` (relational canonical form) is the intermediate produced by the Compiler before the Denormalizer expands field references:

Property	Detail
`.raiv` → `.daiv`	Straightforward — resolve all `$path::field` references left-to-right and materialize absent optional fields from the `.csaiv` (§2.6.13); one pass, O(N) plus schema-sized memory
`.daiv` → `.raiv`	**Impossible** — the relational information is destroyed by denormalization; there is no way to recover which values were copied and which were original
Runtime consumption	The certified runtime consumes only `.daiv` + `.csaiv` — never `.raiv`
Schema validation	`.raiv` can be validated against a schema (field references are syntactically valid values)

Use cases for `.raiv`: meaningful diffs (a single source-of-truth change shows as one changed line, not many), referential-integrity-by-construction in tooling, schema-evolution tracking, round-trip fidelity for `.kaiv` ↔ `.raiv`, and converters that target relational formats.

2.6 Type System

2.6.1 The Single Primitive: `str`

kaiv has exactly one primitive type: **`str`**. Every value between `=` and end-of-line is a string of characters. There is no binary form at any stage — not at authoring time, not at canonical time, not at schema time, not at validation time.

`str` is the identity type: no pattern constraint (any string matches) and `..lex` span ordering by default. `!str` in a type annotation means "raw string with no additional constraints."

Every other type — including `int`, `float`, `bool`, `null`, `b64` — is a **named type** defined in terms of `str` plus constraints. Named types live in type library files (`.taiv`); the standard library `std/core` is always implicitly imported and supplies `int`, `float`, `bool`, `null`, and `b64`. The `!` prefix marks a type annotation in canonical form (`!int`, `!str`, `!std/net/port`); the `&` prefix marks a named-type reference in authored form (`&int`, `&port`), which the Compiler resolves to the canonical `!library/path/typename` form. For `std/core` types, the short name is preserved in canonical form (`!int`, not `!std/core/int`).

2.6.2 Unannotated Scalars Canonicalize to `!str`

The type annotation the *Compiler* writes reflects the type asserted at *authoring* time — an explicit `!type` or `&name` annotation on the source line. An authored scalar carrying no annotation canonicalizes to **`!str`**, the identity type. Canonicalization performs **no type inference** from a value's lexical shape: `port=8080`, `ssl=true`, and `flag=null` all compile to `!str`…`'::=8080` / `=true` / `=null` unless the source line is explicitly annotated (`!int`, `!bool`, `&null`, …). This follows directly from `str` being the single primitive and values being verbatim (§9.5) — there is no lexer or compiler stage that examines a value to guess a narrower type. `.raiv` therefore always carries the authored type.

Schema-declared types reach data lines by **declaration, never inference** — and only in the deployment artifact. When the document declares a schema, the schema-aware *Denormalizer* lifts each untyped (`!str`) line to the field's retained head type from the `.csaiv` (§7.3), so the `.daiv` line itself carries the schema's statement of what the field is (§2.6.17; for `!text` heads the lift is the guarded str→text coercion, §2.6.4). The lift fills in what the author left unsaid; it never overrides an assertion — a data line whose annotation conflicts with its field's head is a `TypeMismatchError` (§7.1), not a rewrite — and in a schemaless document the `.daiv` line keeps `!str` unchanged.

2.6.3 The `std/core` Standard Library

All "built-in" types are defined in `std/core.taiv` — the standard type library that is always implicitly imported. It is hosted at `ktaiv.com` for implementers to reference, but since it is frozen (never changes after initial publication), implementations SHOULD ship it embedded or bundled rather than downloading it at runtime. It is part of the library, tool, and parser distribution.

```
.!taiv std/core

// Integer — decimal string with numeric ordering
/^-?[0-9]+$/ ..num
&int=

// Floating-point number — decimal string with numeric ordering
/^-?[0-9]*\.?[0-9]+([eE][+-]?[0-9]+)?$/ ..num
&float=

// Boolean — two-valued enumeration
{true,false}
&bool=

// Null — empty value only
/^$/
&null=

// Base64url-encoded binary data (RFC 4648 section 5, unpadded)
/^([A-Za-z0-9_-]{4})*([A-Za-z0-9_-]{2,3})?$/
&b64=

// Multi-line text in readable form — the value is a |:|-separated
// sequence of lines. The separator is interpreted only at the
// application/export layer; within kaiv the value is verbatim.
&text=
```

The `b64` pattern admits exactly the encoded lengths unpadded base64url can produce — $\not\equiv 0, 2, \text{or } 3 \pmod 4$ — so an impossible-length value fails the type itself, consistent with the decoded-byte translation of length constraints (§2.8.2). Trailing-bit canonicality (RFC 4648 §3.5 pad bits) is deliberately *not* checked: rejecting non-canonical pad bits is a decoder MAY in the RFC, and kaiv does not out-strict the ecosystem it interoperates with. The quad-form pattern stays inside the pinned regex dialect and the empty value (zero bytes) still matches.

For `std/core` types, `!int`, `!bool`, etc. are **canonical shorthands** — they remain as `!int`, `!bool` in canonical form (they do NOT expand to `!std/core/int`). For all other named types, `!library/path/typename` is the fully-qualified canonical form that replaces `&name` authoring annotations.

Written (authoring)	Canonical form	Note
`!int`	`!int`	`std/core` shorthand — stays as `!int` in canonical form
`!str`	`!str`	Identity type (no pattern, `..lex` default)
`!bool`	`!bool`	`std/core` shorthand — stays as `!bool`
`!float`	`!float`	`std/core` shorthand — stays as `!float`
`!null`	`!null`	`std/core` shorthand — stays as `!null`. Null values always have an empty payload after `=`: `!null'::field=`. Distinct from `!str'::field=` (empty string). See §2.6.17.
`!b64`	`!b64`	`std/core` shorthand — stays as `!b64`
`!text`	`!text`	`std/core` shorthand — stays as `!text`. Multi-line text in readable form; see §2.6.4.
`&port`	`!std/net/port`	Resolved from `std/net.taiv` — library path `std/net`, type `port`
`&datetime`	`!std/time/datetime`	Resolved from `std/time.taiv` — library path `std/time`, type `datetime`
`&customerid` (from `acme/ourtypes`)	`!acme/ourtypes/customerid`	Resolved from `acme/ourtypes.taiv` — library path `acme/ourtypes`, type `customerid`

2.6.4 The `text` Type

Multi-line text in readable form. A `!text` value is a sequence of lines joined by the fixed separator `|:|` — the value's segments, split on every occurrence of the separator, are the text's lines, joined by newlines:

```
!text
basho=old pond|:|a frog jumps in|:|sound of water
```

canonicalizes to `!text'::basho=old pond|:|a frog jumps in|:|sound of water` and exports — to JSON, YAML, TOML, or any other target — as the three-line string. The rules:

- — N separators encode N newlines. Empty segments are empty lines; a trailing separator is a trailing newline; a value with no separator is single-line text; an empty value is empty text.
- — The separator is **fixed**. A configurable separator (the era-1 `!nl` directive) would smuggle per-document lexical state into the format; the fixed `|:|` keeps `!text` a plain type.
- — Interpretation happens **only at the application/export layer**. Within kaiv the value is verbatim bytes like any other: the Validator byte-compares it, constraints apply to the

joined form, and the certified runtime never splits it. Exporters resolve separators to real newlines; importers join LF-separated content.

- Only LF line breaks are representable. Content carrying a CR, a literal `|:|` of its own, or any other violation of the value rules embeds as `std/enc &plain` instead (§2.6.5) — the fallback that carries anything, at the cost of readability.
- `!text` values are authored on standalone lines. Inside the compound sugars the separator collides with the forms' own delimiters (`|` in `:=/+:=` pair values, `:` in inline map entries) and is rejected by the delimiter-collision rule (§11) — multi-line text has no business in a compact one-liner.
- In schemas, a `!text`-declared field **retains** `!text` as a type item in the compiled schema — unlike the constraint-lowered core types, the annotation itself carries semantics (how the value is interpreted on export). The parallel scan accepts two annotations there: `!text` itself, and — the **str→text coercion** — a plain `!str` (or unannotated) line, *provided its value contains no literal `|:|`*. The coercion is meaning-preserving exactly then; a `!str` value carrying `|:|` means those bytes literally, and retyping would silently reinterpret them as line breaks, so it raises `DelimiterCollisionError` instead. The schema-aware Denormalizer performs the retype at build time: a coerced line is emitted into `.daiv` as `!text` — the schema type wins in the deployment artifact, so downstream consumers get the export semantics without re-reading the schema. (This is one instance of the general head-type lift, §7.3; the `|:|` guard is what `text` adds to it.) The coercion is **asymmetric**: a `!text` line on a `!str`-declared field remains a `TypeMismatchError` — text semantics never sneak into fields the schema promised as plain strings.

2.6.5 The `std/enc` Encoding Library

kaiv values are single-line: readable multi-line text travels as the core `!text` type (§2.6.4), and everything else — binary content, text the `|:|` separator cannot carry, structured payloads — embeds as `!b64`. Plain `!b64` says nothing about what the decoded bytes *are*; the **`std/enc`** library types the payload — each member is a named type derived from `!b64` whose name states the decoded content:

```
.!taiv std/enc

// JSON document
!b64
&json=
```

Members: `bin` (generic binary), `plain` (multi-line UTF-8 text — the embedded fallback for content the `!text` separator cannot carry), `json`, `yaml`, `toml`, `xml`, `html`, `md`, `csv`. Usage:

```
.!kaiv
.!types std/enc

&json
config=eyJvbiI6dHJ1ZX0
```

canonicalizes to `!std/enc/json'::config=eyJvbiI6dHJ1ZX0`. Like `std/core`, implementations SHOULD ship `std/enc` embedded; unlike `std/core` it is **not implicitly imported** —

documents opt in with `.!types std/enc` (only `std/core` gets the implicit import and the short canonical names). Private payload types are ordinary type libraries deriving from `!b64` (`acme/enc/product`), resolved through the registries like any other.

> **Lineage.** This restores the era-1 `!b64 TYPE` encoded-type parameter (`!b64 json`, `!b64 x/ProductDetails`) as ordinary named types — the parameter dissolved into the type system rather than surviving as special grammar. Since every type is a constraint triple, "b64-of-JSON" is just a name for the b64 constraint with a documented payload interpretation; validators check the base64url shape and nothing more (payload interpretation is the application's concern, exactly as era 1 specified).

2.6.6 The `std/time` Time Library

RFC 3339 temporal shapes with **chronological (`..time`) span ordering**, mirroring TOML's four datetime flavors — the library behind the `&datetime` → `!std/time/datetime` resolution example used throughout this document:

```
.!taiv std/time

// Offset date-time: 2026-07-03T21:00:00Z
/^\d{4}-\d{2}-\d{2}[Tt ]\d{2}:\d{2}:\d{2}(\.\d+)?([Zz]|[+-]\d{2}:\d{2})$/
   ..time
&datetime=
```

Members: `datetime` (offset date-time), `localdatetime`, `date`, `time`. Like `std/enc`, implementations SHOULD ship `std/time` embedded, and it is imported explicitly (`.!types std/time`); canonical identity is the full path (`!std/time/datetime`). Range constraints (`[2026-01-01,2026-12-31]`) work at every Level: `..time` compares instants, offset-aware (§2.6.12), so mixed-offset documents validate chronologically rather than bytewise.

2.6.7 The `std/num` Numeric Markers Library

kaiv floats are deliberately **finite**: the core `float` pattern admits only finite decimals, which keeps the `..num` span a total, decidable order — range checks never meet IEEE non-finite semantics. The non-finite markers are therefore **dedicated enum types**, mirroring null-as-a-type:

```
.!taiv std/num

// Signed infinity
{inf,-inf}
&inf=

// Not-a-number
{nan}
&nan=
```

An **extended-real** field is the same idiom as a nullable one — a union whose compiled form carries each alternative's group:

```
!float|std/num/inf        →   !float…(/…float/..num)|std/num/inf({inf,-inf
   })'::x?=
```

The canonical marker spellings are `inf`, `-inf`, `nan` (format converters map TOML's `inf`/`nan` and YAML's `.inf`/`.nan` onto them). Like the other marker libraries, `std/num` SHOULD ship embedded and is imported explicitly (`.!types std/num`).

2.6.8 The `std/net` Network Identifiers Library

Network identifiers are the validation everybody re-implements; `std/net` pins them to their defining documents. Five types, each a pure refinement of `str`:

- `uri` — RFC 3986 generic syntax, exactly: scheme; optional authority with userinfo, host (`reg-name`, IPv4, or an `IP-literal` covering the full IPv6 grammar and `IPvFuture`), and port; the path forms; query; fragment; percent-encoding. The pattern is the ABNF transliterated — no simplification.
- `url` — the same grammar with a **required authority** (`scheme…://`): the everyday hierarchical subset. Every `url` is a `uri`; `mailto:` and `urn:` forms are `uri`-only.
- `email` — the WHATWG HTML valid-e-mail-address pattern (dotted-atom local part over RFC 1123 domain labels): the practical interchange subset, deliberately not full RFC 5322 (whose quoted forms and domain literals no receiver wants).
- `hostname` — RFC 1123: dot-separated labels of letters, digits, and interior hyphens, at most 63 octets per label, with the 253-octet total carried as a length constraint (`#[1,253]`).
- `port` — `int` refined to `[0,65535]`. Port 0 is reserved; schemas tighten to `[1,]` where an assignable port is meant.

The `uri`, `url`, and `email` patterns are the reason the dialect carries `\x27` (§10): apostrophe sits in RFC 3986's `sub-delims` and in the e-mail local-part alphabet, and `\x27` is its only expressible spelling. Schema converters map JSON Schema's `format:` values (`uri`, `email`, `hostname`) and XSD's `xs:anyURI` onto these types instead of dropping them. Like the other libraries, `std/net` SHOULD ship embedded and is imported explicitly (`.!types std/net`).

2.6.9 The `std/math` Mathematical Types Library

`std/math` holds mathematical value types beyond the core scalars; its first member is `complex`:

```
// Complex number — a+bi, both components always present.
/^-?[0-9]*\.?[0-9]+([eE][+-]?[0-9]+)?[+-][0-9]*\.?[0-9]+([eE][+-]?[0-9]+)?i
   $/
&complex=
```

One spelling per value: both components are always present (`3+0i`, `0+1i`, never a bare real or a lone `2i`), the separator sign *is* the imaginary part's sign (`-1.5-0.5i`; `3+-2i` is ill-formed), no whitespace, and the `i` suffix is mandatory. Components use the `float` token grammar — integer spellings included, so Gaussian integers (`3+2i`) are expressible without a dedicated type.

`complex` deliberately carries **no numeric span**: the complex numbers admit no total order compatible with their arithmetic, so ordering stays the default byte order — deterministic for uniqueness checks, meaningless for ranges, which is exactly the mathematical situation. Range constraints on `complex` are therefore inert; constrain components by pattern instead. Like the other libraries, `std/math` SHOULD ship embedded and is imported explicitly (`.!types std/math`).

2.6.10 Standard Library Evolution

Registries are append-only and entries are permanent: a published library file never changes. A standard library therefore evolves by **new name**, never by republication — a revised network library is published as `std/net2` beside the eternal `std/net`, and `.maiv` mappings bridge the two. The old name keeps serving every document that ever referenced it; nothing published against it can break.

`std/core` is the one library this convention cannot reach: it is **unversionable by construction**. Its members resolve to the canonical shorthands (`!int`, `!text`, …) that appear in every canonical document *without* a library path — core is implicitly imported everywhere, so no `core2` could slot into existing documents, and any change to core's membership is a change to canonical form itself. The membership of `std/core` frozen at 1.0 is therefore final in a stronger sense than any other library's; altering it is major-revision territory.

2.6.11 The Constraint Triple

The universal type constructor is the **constraint triple** `(pattern, span, range)`. Every type is `str` narrowed by zero or more of:

— a **pattern** constraint `/regex/` — the value must match the regex. Within the pattern body, `\/` denotes a literal `/`; the closing delimiter is the first `/` not preceded by a backslash. This is the **sole escape sequence in the kaiv family**, scoped to the constraint sub-grammar of schema and type files — it does not exist in data values (§9.5). The backslash is passed through to the regex engine as part of the pattern (`\/` is a valid regex escape for `/`), so no unescaping step is needed: the delimiter scan and the regex body agree byte for byte.
For slash-heavy regexes (URLs, filesystem paths), an **alternative-delimiter form** avoids the escapes: `re{sep}body{sep}`, where `{sep}` is one of `: ; % ~ @ #` — `re~^https://[a-z.]+/.*~` is the same constraint as `/^https:\/\/[a-z.]+\/.*/`. There is **no escaping** inside the body: the literal ends at the first recurrence of the chosen separator, so the body simply may not contain it (pick another separator; the slash form with `\/` remains the universal fallback). The form is authoring-only and always **whitespace-separated** — from a preceding type annotation (`!str re…~~`, never glued) and from neighboring constraint items; after a union it narrows the immediately preceding alternative, like a glued constraint. The compiler lowers it to the canonical slash-delimited form, escaping every `/` in the body as `\/`; `re{sep}` never appears in `.raiv`/`.daiv`/`.csaiv`. Because a line-leading `re{sep}` must classify deterministically against the `:=`/`;=` operators, **`re` is a reserved word in leading name position** in `.saiv`/`.taiv` files: a field literally named `re` uses a quoted name (`"re"=`), mirroring `min`/`max` in table headers; `&re` type definitions are unaffected (the sigil disambiguates),

and data files are unaffected (they carry no constraint items). A pattern body in this form is subject to the same `'` exclusion as `p-char`.

— a **span ordering** `..kind` — declares how range constraints are evaluated.

— a **range** constraint `[min,max]` — the value must fall within the inclusive range under the declared span. Either endpoint may be omitted for a half-open range (`[0,]`, `[,255]`).

For example, `int` (in `std/core`) is defined as `str + /^-?[0-9]+$/ + ..num`: a string matching the integer regex, ordered numerically. A port number type adds a range: the constraint line `&int [0,65535]` above the definition line `&port=`.

Two additional constraint kinds compose with the triple:

— an **enumeration** `{a,b,c}` — the value must be one of the listed alternatives. (`bool` in `std/core` is defined purely as `{true,false}`.)

— a **length** constraint `#[min,max]` or `#{a,b,c}` — restricts character count (for `!str`), element count (for arrays), or decoded byte count (for `!b64`). See §2.8.2 below.

2.6.12 Span Orderings

The span ordering declared on a type determines how range constraints `[min,max]` are evaluated. The built-in spans are a small, fixed, certifiable set:

Span	Meaning
`..num`	Decimal numeric ordering — parse string as number, compare mathematically.
`..lex`	Lexicographic (byte-by-byte) ordering. The default for `str`.
`..lex[locale]`	Locale-aware lexicographic ordering. Level 3 only; BCP 47 language tag selects collation rules.
`..time`	RFC 3339 chronological ordering — instants, offset-aware.
`..ver`	Version ordering — dot-separated components, numeric where both parse.

A range constraint is meaningful only relative to a span: `[1,65535]` on `&int` (`..num`) means numeric comparison; `[1,65535]` on a `..lex` type would compare strings lexicographically. Schema authors normally never have to think about this — the span is set on the named type and inherited by any range constraint that uses it.

The comparison function each span ordering uses:

Span	Comparison	Implementation
`..num`	Numeric parse + mathematical comparison	`parse_int(a) <= parse_int(b)`
`..lex`	Byte-by-byte	`memcmp(a, b)`
`..lex[locale]`	Locale-aware collation	ICU `ucol_strcoll()` or equivalent (CLDR 48, tertiary strength; §5.3)
`..time`	RFC 3339 chronological	Instant comparison, offset-aware (see below)

Span	Comparison	Implementation
`..ver`	Dotted version	Component-wise comparison (see below)

Numeric domain of `..num`. `parse_int`/`parse_float` above is schematic. Normatively: for a value constrained by an `int`-derived type, `..num` comparison is **exact integer** comparison (arbitrary precision — no silent truncation at 2^53); for a `float`-derived type it is IEEE-754 double comparison. A value that is syntactically valid for its type but outside the representable range, or a non-finite token (`NaN`, `inf`), fails the type's own pattern constraint first (e.g. `NaN` does not match the float pattern) and so never reaches the range check. Range endpoints are parsed in the same domain as the value.

Temporal domain of `..time`. `..time` comparison is **chronological over instants**: operands in the RFC 3339 shapes are parsed and compared as instants, with UTC offsets normalized — `2025-12-31T23:00:00+01:00` and `2026-01-01T00:00:00+02:00` are equal, and a range check meeting a differing offset compares the instants, not the bytes. Within a single fixed offset this coincides with byte order. An operand outside the RFC 3339 shapes falls back to byte comparison — a `..time` type's own pattern normally excludes that case before any range check runs. The parse is constant-memory; no calendar library is required. (Pinning string comparison instead was considered and rejected: it would freeze a cross-offset footgun whose later fix would be validation-semantics drift on eternal artifacts.)

Version domain of `..ver`. `..ver` comparison is **component-wise over dot-separated components**, left to right: a component pair that both parse as unsigned integers compares numerically (`1.10` > `1.9`); any other pair compares bytewise; and when one operand exhausts its components first, the shorter orders lower (`1.2` < `1.2.0`). This is deliberately *not* full SemVer — there are no prerelease or build-metadata semantics; a domain needing them defines a named type whose pattern pins the accepted shape.

Performance is identical to a primitive-based approach. The Validator reads the string `"8080"` from the `.daiv` line and validates against the compiled constraints (pattern, span, range) from the `.csaiv`. Implementations MAY — and SHOULD — hardcode optimized validators for `std/core` types as an optimization. The spec defines the semantics; the implementation may recognize `!int` and skip the regex match in favor of a direct integer parse.

2.6.13 Named Types

The Named Type System **is** kaiv's type system. All types except `str` are named types — types defined in type library files (`.taiv`) as `str` plus optional constraints. This includes the "built-in" types (`int`, `float`, `bool`, `null`, `b64`) which are defined in `std/core.taiv`. The design follows the `#include <stdint.h>` model from C, but taken further: even `int` itself is a library type, not a language primitive.

The key principle: **one Lexer, same line grammar everywhere.** Type library files (`.taiv`), schema files (`.saiv`), and data files (`.kaiv` / `.daiv`) are all processed by the same Lexer with the same metadata-above / definition-below line grammar.

Type Library Files (`.taiv`). A type library file (`.taiv` — *type kaiv*, pronounced "tave") defines a set of named types. Each type definition follows the same pattern as every other definition in the kaiv family: **metadata lines above, definition line below**.

The format declaration for a `.taiv` file is `.!taiv`, paralleling `.!saiv` for `.saiv` files. The optional version and the library identifier (`acme/net`) follow the same convention as other format declarations.

Example — a domain type library (`acme/net.taiv`)

The example library below is *fictional* — deliberately distinct from the shipped `std/net` (§2.6.8), whose normative membership and RFC-exact patterns it does not reproduce:

```
.!taiv acme/net

// IPv4 address in dotted-decimal notation (simplified pattern — production
    use would validate 0–255 per octet)
/^(\d{1,3}\.){3}\d{1,3}$/
&ipv4=

// IPv6 address
!str
&ipv6=

// Network port number (inherits &int range semantics via ..num)
&int [0,65535]
&port=

// URI (RFC 3986)
!str
&uri=

// Email address (simplified pattern — production use would follow RFC 5321
    more precisely)
/^[^@]+@[^@.]+\.[^@]+$/
&email=
```

Each type definition consists of:

1. Optional `//` doc comment(s) above
2. Constraint line(s): `/regex/` pattern, `..span` ordering, `{enum}` enumeration, or a base named type reference (`!str`, `&int`, etc.) plus any narrowing constraints — any combination on one or more metadata lines
3. Definition line: `&name=` — the named type being defined, optionally with a default value (`&name=defaultvalue`)

The `&name=` definition line is a `KV` token where the key starts with `&`. It is the same token kind as `field=` in a schema — the `&` prefix distinguishes a type definition from a field definition. A `.taiv` file is a kaiv document: lexed by the same Lexer, same comment syntax, same line structure, same ordered-keys rule. Its content happens to be type definitions rather than data.

The & Sigil. The & sigil identifies **named type references** in authoring files. It appears in two positions:

Position	File type	Syntax	Meaning
Definition position	`.taiv`	`&name=` / `&name= default`	Defines a named type (the definition line, analogous to `field=` in schemas)
Annotation position	`.saiv`, `.kaiv`	`&name` on a metadata line above a field/data line	Annotates a field or value with a named type (same position as `!int`, `!str`, etc.)

& is an **authoring-level sigil only**. It does not appear in canonical form (`.daiv`) or compiled schemas (`.csaiv`). During Compiler canonicalization, every `&name` annotation is resolved to `!library/path/typename` — the fully-qualified canonical form:

- `&port` (from `std/net.taiv`) → `!std/net/port`
- `&datetime` (from `std/time.taiv`) → `!std/time/datetime`
- `&customerid` (from `acme/ourtypes.taiv`) → `!acme/ourtypes/customerid`

The **resolution rule** for `!library/path/typename`:

- The last `/`-separated segment is the type name.
- Everything before is the library path.
- Base URL: determined by Type Registry Resolution (see §2.1.4 above) — `.!registry` declaration overrides; build-time config overrides for matching prefixes; default is `ktaiv.com`.
- File: `{base_url}/{library/path}.taiv` (default: `ktaiv.com/{library/path}.taiv`).
- Definition: `&typename=` in that file.

& joins `+=`, `;=`, `:=`, and dot-prefixed variable definitions as authoring sugar that canonicalizes to a more explicit form.

! vs. & at a Glance

Prefix	Where it appears	Meaning
!	`.daiv` (canonical data), `.csaiv` (compiled schema), `.kaiv`/`.saiv` (as shorthand for `std/core`)	Canonical type reference. `!str` = identity type. `!int`, `!bool`, `!float`, `!null`, `!b64`, `!text` = canonical `std/core` forms (not expanded further). `!library/path/name` = fully-qualified for all other types.

Prefix	Where it appears	Meaning
`&`	`.kaiv` (authored data), `.saiv` (authored schema), `.taiv` (type definitions)	Short-form type reference. Resolved against imported type libraries declared via `.!types`. In canonical form, `&name` → `!library/path/name` (or `!core-shorthand` for `std/core` types). Never survives canonicalization.

Type Library Import (`.!types`). A schema imports type libraries with the `.!types` declaration. It follows the same pattern as `.!schema` declarations: placed after the format declaration, before content lines. Multiple `.!types` declarations are allowed.

```
.!saiv https://example.org/server-config.saiv
.!types std/net
.!types std/time
```

The `.!types` declaration is the kaiv equivalent of `#include <stdint.h>` in C. It tells the schema compiler to load the named type library and make its named types available for `&` annotation resolution.

The resolution of `.!types` library paths follows the same layered resolution as type annotations (see §2.1.4). Type libraries form a one-directional dependency: `.taiv` files reference only `str` (the primitive) or other named types from already-imported libraries. Schemas reference type libraries. Data files reference schemas. No circular imports are possible. `std/core` is implicitly available everywhere — it does not need to be declared with `.!types`.

Default Values. A schema field definition is a complete `caiv` content line, and its right side is the field's **default value**. This mirrors the data layer exactly: in a `.kaiv` file, `key=` assigns the empty string — a kaiv value is never absent, only empty — so in a `.saiv` file, `key=` declares the empty-string default. Every field therefore carries a default; the empty string is the degenerate one, and there is no "no default" state to represent.

Types carry defaults too. A named-type definition is also a complete content line, and its right side is the *type's* default: `&port=443` in a `.taiv` defines a port type defaulting to 443. Every level of the definition chain — the schema field, its type, that type's base, transitively — therefore holds a default slot, and the schema compiler resolves them as a **cascade**:

> The applicable default is the most specific one that satisfies the field's own constraints: the field's, else the type's, else the base chain's, else the (inert) empty string.

Because the empty string fails every constrained type's own pattern, inheritance needs no syntax — a bare `timeout?=` under `&port` inherits 443 precisely because its own `""` default is inert, while an explicit `admin?=9090` wins by being both more specific and applicable. The honest edge: for an *unconstrained* `str` field the empty string is a valid value, so it shadows any type default; a constrained string type (any pattern or length) behaves like the typed cases.

The compiled schema carries the resolved default. The schema compiler runs the cascade at compile time and bakes the winner into the `.csaiv` right side (…`'::listen?=443`), so consumers of the compiled artifact — the form the registries serve — can materialize defaults without fetching the authored sources. The `csaiv-field-line` right side is thus `[ value ]`, empty when the resolved default is the empty string.

Two rules keep defaults orthogonal to validation:

— **A default is applicable only if it satisfies the field's own constraints.** An empty-string default on an `!int` field fails the int pattern and is inert by construction.

— **Defaults are materialized at build time, never at validation time.** Requiredness is the operator's concern (`=`/`?=`). When an optional field is absent from the authored data, the **Denormalizer** materializes the resolved default (or `!null`, §2.6.17) into `.daiv`, so every schema-declared field appears in the deployment artifact and `.daiv` is fully self-contained — no schema lookup, default application, or absent-field branching is ever needed downstream of the build. The **Validator** never applies defaults: it sees the materialized `.daiv` and checks each line against its `.csaiv` constraints. The certified integrity check compares values; it never synthesizes them, and it ignores the right side of `.csaiv` field lines entirely. A consuming application likewise never applies defaults — what is in `.daiv` is the complete truth.

An optional field whose resolved default is inapplicable and whose type does not admit `!null` would leave the Denormalizer with nothing to materialize when the field is absent; the schema compiler rejects such a declaration (`SchemaOptionalWithoutDefaultError`, §11.2).

Named Types in Schemas. In a schema (`.saiv`), `&name` appears on the metadata line above a field definition — the same position as `!int` or `!str`:

```
.!saiv https://example.org/server-config.saiv
.!types acme/net
.!types std/time

// Server hostname
&ipv4
host=

// Server port
&port
port=

// When the server was configured
&datetime
created_at=
```

The `&ipv4` line sits in the **exact same position** as `!str` would — it is a type annotation metadata line that applies to the immediately following field definition line. The schema compiler resolves `&ipv4` against the imported type libraries (`acme/net`) to retrieve the constraints (pattern `/^(\d{1,3}\.){3}\d{1,3}$/`) and lower them to the compiled form in the `.csaiv`.

Named Types in Data Files. In authored data files (`.kaiv`), `&name` appears in the same metadata position as `!type`, above the data line:

```
.!kaiv
.!schema:acme/server-config

&datetime
created_at=2025-01-15T09:30:00Z

&ipv4
host=192.168.1.1

&port
port=8080
```

When the schema already supplies type information, the explicit `&` annotation in data files is optional — the same relationship as `!int` in a schema vs. authored data. When present, the annotation is checked against the compiled field's retained head type (§7.3) under the parallel scan's type-check rules (§7.1): nominally where the head's name is its semantics (a union, `!text`, `!null`, `std/enc/*`, an explicit `!str` head), structurally everywhere else — a chain-compatible annotation such as `!int` under a `!std/net/port` head is accepted, and the head's constraint group governs the value either way. Only a field whose head is the elided identity `str` carries no type item; there the annotation is trusted as carried documentation — the value's conformance is enforced by the constraint check, and downstream converters key on the annotation as written.

In canonical form (`.daiv`), `&name` is resolved: `std/core` types become their canonical shorthand (`!int`, `!bool`, etc.), all other types become `!library/path/typename`. So the above becomes:

```
.!daiv
.!schema:acme/server-config
!std/time/datetime'::created_at=2025-01-15T09:30:00Z
!acme/net/ipv4'::host=192.168.1.1
!acme/net/port'::port=8080
```

No `&` appears in the canonical output.

Constraint Narrowing. A schema can add further constraints on a field that uses a named type, narrowing the base definition:

```
// In std/net.taiv — base definition:
!int [0,65535]
&port=

// In my-schema.saiv — narrowed for this field:
&port [1024,65535]
listen_port=
```

The `&port [1024,65535]` annotation metadata line narrows the base `[0,65535]` range. This is the same constraint-narrowing mechanism as `!int[1024,65535]` narrowing plain `!int` — the named type annotation can carry additional inline constraints on the metadata line that tighten (but never widen) the base definition from the type library.

Anonymous Refinement (Bare Constraint Lines). A `.saiv` metadata line consisting solely of value-constraint items — pattern, span, range, enum, length — refines the **implicit `str`** type of the next field definition:

```
// Equivalent to !str/^[a-z]+$/#[1,8] on one line:
/^[a-z]+$/ #[1,8]
name=
```

This is the `.taiv` definition shape (§2.6.13) applied to a field instead of a `&name=` definition — the natural reading for a kaiv author, since `str` is the universal value type and every constraint already refines it. The compiled form carries no type item: the field lowers to a bare constraint group, which the Validator evaluates without a nominal check. This is the deliberate contrast with an authored `!str` annotation carrying the same items — the *identity declaration*, retained as a nominal head (§7.3): write `!str` to reject asserted types on the field, write the bare constraints to tolerate them. Type-reference items (`!type`, `&name`) are **not** admitted on a bare constraint line — a line carrying a type reference is authored as a type-annotation or named-annotation line, which lead with their own sigils. A schema compiler encountering a rule-6 metadata line it cannot interpret — a bare line with a type-reference item, or any other leader with no `.saiv` meaning — MUST reject it; silently dropping an annotation would weaken the compiled contract relative to the authored one.

Why Named Types Resolve to Fully-Qualified Form in Canonical Output. Named type annotations are resolved to fully-qualified form (`!library/path/typename`) in `.daiv` because:

- **Format converters** need the full library path to emit the correct logical type in Avro (`logicalType: "date-time"`), XSD (`xs:dateTime`), ProtoBuf (`google.protobuf.Timestamp`), etc. `!std/time/datetime` tells a converter exactly which type library and type definition to look up — no import context needed.
- **Greppable by type:** `grep "^!std/time/" config.daiv` finds all time fields; `grep "^!std/net/port'" config.daiv` finds all port fields. The fully-qualified prefix makes every line a self-contained fact.
- **No import context needed at runtime:** The certified runtime reads `!std/time/datetime'/server::created_at=2025-01-15T09:30:00Z` and knows the full type identity from the line alone.

Exporters Resolve Custom Heads. That lookup is specified: a format exporter interpreting a custom head (a library-path type outside the built-in `std` tables) resolves it, in order, through (1) the type registries — the standard Layer 1–4 resolution, so vendored local libraries serve offline devices exactly as they serve validation; and (2) the document's declared `.!schema`, whose compiled form already carries every field's lowered base kind — a device holding the `.csaiv` for validation already holds everything an exporter needs, with no further resolution. A chain whose lowering carries the `..num` span exports as the target format's number; a lowered `{true,false}` enum as its boolean. Interpretation is **best-effort and total**: a head neither source can interpret — and a value that does not fit the resolved kind — exports as the string the value already is, never an error. Resolution

refines; it cannot reject. (Implementations expose the uninterpreted form as an option — the reference CLI's `--no-resolve`.)
The compiled `.csaiv` schema carries the *lowered* form — constraint pattern + span + range/enum — for the Validator's parallel scan. The runtime never sees `&` references; it sees only constraint forms. Type identity (the fully-qualified `!library/path/name`) is carried in the `.daiv` data file for tooling and format conversion; constraint forms (patterns, spans, ranges) are carried in the `.csaiv` for validation.

2.6.14 Tagged Unions (`oneOf`)

Schema syntax. `!int|str` — pipe-separated type alternatives.

Data text MUST include an explicit type annotation choosing one alternative. The type annotation on a data line is the discriminant that determines which constraint set is applied during the Validator's parallel scan validation.

`!null|T` is the nullable pattern: a field that can be either null or a value of type `T` is declared as `!null|T` in the schema. The canonical data line carries the active variant — `!null'::field=` when null, `!T'::field=value` when non-null. See §2.6.17 for the complete null model.

Units on alternatives. An alternative carries its own unit exactly as a non-union annotation does — the unit attaches to the alternative it follows and is part of that alternative's identity:

```
!null|float:km        →   !null(/^$/)|float:km…(/…float/..num)'::dist?=
```

`!null|float:km` is the **nullable-quantity pattern** — the shape the optionality interlock forces for an optional measurement with no applicable default (§2.6.13): the canonical line carries `!float:km'::dist=42` when present and `!null'::dist=` when null. The union itself never carries a unit; each alternative's unit, or its absence, participates in the discriminant (§7.1).

Maps to:

Target	Construct
ProtoBuf	`oneof`
GraphQL	`union`
ASN.1	`CHOICE`
JSON	schema `oneOf`

2.6.15 Schema Composition (`allOf`)

Schema syntax. Multiple `.!schema` declarations in the same kaiv text.

The data MUST satisfy all referenced schemas simultaneously. Composition operates at the schema level, not the data level — each referenced schema contributes field definitions to the compiled schema (`.csaiv`).

Maps to:

Target	Construct
GraphQL	`implements` (interface implementation)
ASN.1	component inclusion
JSON	schema `allOf`

2.6.16 anyOf

No distinct syntax is needed. `anyOf` is subsumable under the constraint system and does not require a first-class language construct. None of ProtoBuf, GraphQL, or ASN.1 has a native `anyOf` concept, so there is no target format to drive a dedicated syntax for it.

2.6.17 Null Semantics

This section formalizes the null model in kaiv: what null means, how it is represented in canonical form, how nullable fields are declared in schemas, and how null interacts with defaults and field absence.

The Canonical Representation: `!null'::field=`. A null value in canonical form is:

```
!null'::field=
```

The type annotation is `!null` and the value after `=` is always empty. The Validator MUST reject `!null'::field=something` — null carries no payload. The `/^$/` pattern constraint on `&null` in `std/core.taiv` enforces this: the value must be the empty string. Inside a `!null|T` **union** the same holds: the `null` alternative's compiled group carries `/^$/`, so a non-empty `!null` payload fails validation (§2.6.14).

Null vs Empty String — The Type Annotation Disambiguates. The distinction between null and empty string is entirely in the type annotation:

Line	Meaning
`!null'::field=`	Null — the field has no value
`!str'::field=`	Empty string — the field has a value, and that value is `""`

Both lines have nothing after `=`. They are semantically different because of the type annotation. This is a direct consequence of kaiv's "every value has an explicit type" principle.

Nullable Fields Require Explicit `!null|T` Declaration. A field is nullable **only** if its schema declaration includes `!null` in a union type. Nullability is explicit and opt-in — never implicit.

```
# In .saiv schema:

# nullable string — can be null or any string (including empty)
!null|str
```

```
name=

# required non-nullable string — must have a string value
!str
host=

# nullable integer — can be null or an integer
!null|int
timeout=
```

A field declared as `!str` **cannot** hold null. Only fields declared with `!null|T` (or `!null|T1|T2|...`) can hold null.

Active-Variant Annotation in Canonical Form. The full union type `!null|str` is a schema concept (it appears in `.saiv` and `.csaiv`). A canonical metadata prefix (§10) has no union form: the data line carries only the **active variant** — the concrete type of the actual value. A union annotation on an *authored* data line is sugar, and the Compiler MUST resolve it per emitted value (a splice's elements may each pick a different alternative) to the **first** alternative — the head type first, then left to right — whose lowered definition (base type plus authored narrowing) the value satisfies; a value satisfying no alternative is `TypeMismatchError` at compile time. Order therefore matters: an unconstrained `str` alternative accepts everything, so it belongs last.

```
# Schema declares: name is !null|str

# When the value is null:
!null'::name=

# When the value is a non-empty string:
!str'::name=hello

# When the value is an empty string (NOT null):
!str'::name=
```

This is consistent with how all union types work in canonical form: the canonical line always carries the concrete type, not the union declaration. The parallel scan validator checks that the data line's type annotation is one of the allowed union alternatives declared in the `.csaiv`.

Materialization of Absent Fields. A field that is absent from the authored `.kaiv` has a schema-dependent outcome, applied by the **Denormalizer** at build time (§2.6.13):

Situation	`.daiv` output
Field present with value	`!type'::field=value`
Field explicitly null (`!null` annotation above an empty-valued line)	`!null'::field=`
Field absent, applicable default (§2.6.13)	`!type'::field=default_value`

Situation	.daiv output
Field absent, optional, no applicable default, nullable (!null|T)	!null'::field=
Field absent, required (=)	Build error — the Denormalizer raises RequiredFieldSchemaError

The Denormalizer emits !null'::field= for absent nullable fields and the resolved default for absent defaulted fields. This preserves the "every schema-declared field appears in .daiv" invariant required by the parallel scan. (The remaining combination — optional, no applicable default, non-nullable — is rejected when the schema itself is compiled: SchemaOptionalWithoutDefaultError, §2.6.13.)

Two consequences of working from the *compiled* schema: the !type token of a materialized line is the .csaiv field's retained head type (§7.3) — !int, !type:unit, !acme/net/port, or the matching union alternative — and !str only for a head-less identity field. The same head is **lifted** onto *present* untyped lines: a !str data line on a typed field is emitted into .daiv under the head type (the schema type wins in the deployment artifact; for !text heads this is the guarded str→text coercion, §2.6.4), so exporters and every other downstream consumer read the type from the line itself, never from the schema. And inside a namespace array, materialization applies **per element**: each element run must present the full schema-declared field sequence for the strict lockstep scan, so an element's absent optional fields are materialized within that element's run.

Explicit null in authored .kaiv may also be written as:

```
!null
name=
```

The !null type annotation above the data line explicitly marks the value as null. The Compiler canonicalizes this to !null'::name=.

Every Schema-Declared Field Appears in .daiv — No Absent Lines. The parallel scan between .daiv and .csaiv requires every schema-declared field to have a corresponding line in .daiv. There are no "absent" lines in canonical form: the Denormalizer materializes every schema-declared field (§2.6.13), in schema-declared order. This preserves the pure lockstep parallel scan — no schema lookup and no branching logic for absent fields is needed at runtime, which is precisely what makes .daiv self-contained for embedded and safety-critical consumers.

Default Values and Nullability Are Independent

Schema declaration	Outcome when the field is absent in .kaiv
!str + field= (required, non-nullable)	Build error — RequiredFieldSchemaError
!str + field?= (optional, unconstrained str)	Empty-string default applies (§2.6.13): !str'::field=

Schema declaration	Outcome when the field is absent in `.kaiv`
`!int + field?=` (optional, constrained, no applicable default)	Schema rejected at compile time — `SchemaOptionalWithoutDefaultError`
`!str + field?=default_value` (has default)	Default materialized: `!str'::field=default_value`
`!null|int + field?=` (nullable, no applicable default)	Null materialized: `!null'::field=`
`!null|str + field?=default_value` (nullable, has default)	Default materialized: `!str'::field=default_value`

2.6.18 Map Type

Maps are collections of key-value pairs where keys are arbitrary strings and values conform to a declared type. Every major interchange format supports maps:

Format	Map construct
Avro	`map`
JSON	object with dynamic keys
ProtoBuf	`map<string, V>`
GraphQL	not a first-class type; represented as custom scalars or key-value list types
TOML	inline table with dynamic keys

In kaiv, maps are represented using the `!map` type annotation. A map field declares its value type in the schema and is populated with arbitrary string keys at the data layer. Unlike the other core-type keywords, `map` is a **structural type constructor**, not a `std/core` named type — it is not defined as `str` plus constraints; its semantics are the namespace-with-arbitrary-fields construction below.

Authored Syntax

```
!map
/config/settings={}
```

Or with inline key-value pairs (key and value separated by `:`, pairs separated by `;`):

```
!map
options=key1:val1;key2:val2
```

Schema Declaration

In a `.saiv` schema, `!map<VALUETYPE>` is a type annotation like any other — a metadata line above the field definition, in the same position `!str` or `&ipv4` would occupy:

```
!map<str>
settings=
```

This declares `settings` as a map whose values are strings.

Canonical form

A map is a **namespace with arbitrary string-named fields** — the same construction as an array, which is a namespace with integer fields. In `.daiv`, each map entry is one canonical line projecting the entry key as a field of the map's namespace:

```
!str'/config/settings::key1=val1
!str'/config/settings::key2=val2
```

Every map-entry line therefore ends with `::key=value` like all other canonical data lines — the universal `::` rule (§1.4.2) has no map exception. A key that does not match the bare-name grammar is quoted per §2.3 (`!str'/config/settings::"weird key"=val`). An empty map (authored `={}`) produces no canonical entry lines.

This keeps the canonical form flat and DFA-walkable, consistent with the treatment of namespaces and arrays. The schema DFA validates each map entry's value against the declared value type. The key is unconstrained (arbitrary string) unless a key-pattern constraint is declared in the schema.

Because `:` separates key from value and `;` separates pairs, neither character can appear literally in an inline map key or value — there are no escape sequences (§9.5). Entries containing these characters are authored as direct namespaced field lines instead (`/config/settings::key1=a:b;c`, with the key quoted per §2.3 if needed) — the value after `=` is verbatim, so any character is representable there.

2.7 Units

Numeric types may carry a **unit** annotation declaring the physical or economic quantity the value represents. The unit attaches to the type sigil with a `:` separator and lives inside the metadata prefix, before any provenance and before the `'` boundary:

```
!float:km'/server::distance=42
!int:s'::timeout=30
!float:km/h'/vehicle::speed=80
!float:~EUR'/order::total=99.50
```

Units are restricted to types whose span ordering is `..num` — `!int`, `!float`, and any named type derived from them. Annotating a `..lex`-, `..time`-, or `..ver`-ordered type (including `!str`, `!bool`, `!null`, `!b64`) with a unit is a compile-time error. In authored annotations the type name may be elided — `!:km` — inheriting the field's schema head, or `float` where none governs (§2.7.4).

2.7.1 Metadata Prefix Order

The full canonical metadata prefix order is:

```
!type[inline-constraints]:unit?provenance'namepath
```

For example, `!int[1,3600]:s?sensor1'/timeout::value=30` declares an integer in seconds, range-constrained to one hour, sourced from `sensor1`. Inline constraints stack with units exactly as they stack with provenance: each component is independently optional, and the order is fixed.

Two examples with every slot populated, showing the fixed DFA ordering end to end:

```
!float[0,100]:km?gps1@2025-01-15T09:30:00Z#p-17'/trip::length=42.5
!str/[a-zA-Z0-9.-]+/#[1,253]?dns1@2025-01-15T09:30:00Z#req-42'/server::host
   =example.com
```

The first line is a numeric type with an inline range, a unit, and the full provenance triple. The second is a string type with a pattern and a length constraint (length constraints sit inside the inline-constraints slot, before `:unit` and `?provenance`), then the full provenance triple. In both, everything before `'` is one whitespace-free token; the `#` in `#[1,253]` is a length constraint because it appears before any `?`, and the `#` in `#p-17` / `#req-42` is a data point identifier because it appears inside the provenance list (§2.4.2).

The `/` inside a compound unit expression is unambiguous because units always precede `'` and the namepath always follows it — the DFA reads from the `:` after the type up to the `?` of a provenance list or, absent one, the `'` delimiter, as a single unit token, with no context-sensitive parsing (`?` cannot occur inside a unit expression).

2.7.2 Validation: Units Do Not Convert

The Validator checks two things about units:

1. The unit string parses as a known unit expression — built-in, or defined by a `.faiv` library imported with `.!units` (§2.7.10).
2. The data line's unit string is byte-identical, in canonical form, to the unit declared by the schema for that field. The unit rides the retained type token in the compiled schema (`!float:km` as the field's first item); a mismatch — or a missing unit — is a `TypeMismatchError` (the unit is part of the type's identity). For a union field the unit rides the *matched alternative* (§2.6.14); an alternative without a unit demands none.

It does **not** convert values across units. `!float:km'::d=42` and `!float:m'::d=42000` are different canonical lines with different literal payloads, and the Validator treats them as such. Conversion belongs to the build's authored-unit conversion (§2.7.3) and to consumer tooling — queries, post-processing, format converters — never to the Validator or the certified runtime.

This keeps unit handling within the constant-memory, no-arithmetic discipline of every other Validator check: parse, compare strings, accept or reject.

2.7.3 Authored-Unit Conversion

The declared unit is the field's contract; the authored unit is the author's convenience. When a data line's annotation carries a **different unit of the same dimension** than the field's declared unit, the schema-aware Denormalizer **converts** the value into the declared

unit at build time — the unit-flavored twin of the head-type lift (§2.6.17): the schema's unit wins in the deployment artifact, and the value is converted so that meaning is preserved.

```
# schema declares !float:m; authored:
!float:km
/trip::dist=42
```

```
# .raiv — authored unit preserved:
!float:km'/trip::dist=42
# .daiv — declared unit, value converted:
!float:m'/trip::dist=42000
```

The rules:

- **Exact decimal arithmetic only.** The converted value is the authored value multiplied by the ratio of the authored unit's conversion factor to the declared unit's. The computation is exact over finite decimals; if the exact result is not a finite decimal (authored `m` against a declared `:yd` field divides by `0.9144`), or violates the field's type (a non-integer result on an int-derived field), the build fails with `UnitConversionError` and the author writes the declared unit instead. Determinism is the point: `.daiv` bytes are reproducible, and no floating-point arithmetic exists anywhere in the pipeline.
- **Same dimension only**, with factors resolved from the built-in tables (§2.7.7) and imported `.faiv` definitions. A cross-dimension unit remains a `TypeMismatchError`. Compound units convert by their derived factors under the same exactness rule.
- **Currencies never convert** — they carry no factors by design (§2.7.12); an authored `:~EUR` against a declared `:~USD` field is a `TypeMismatchError`. The rate-source template remains consumer-side.
- **Build-side only; DFA processing is unaffected.** `.raiv` preserves the authored unit — the relational record of what was written; `.daiv` carries the declared unit. The Validator and the certified runtime byte-compare canonical units exactly as before (§2.7.2): a `.daiv` line whose unit differs from the schema's fails validation, conversion or no conversion.

2.7.4 Elided-Type Unit Annotation (`!:unit`)

Under authored-unit conversion, the only information an annotation contributes on a schema-defined numeric field is the unit — the type is the schema's, and a restated type name is unchecked noise that can drift. The **elided-type unit annotation** says exactly the new thing:

```
!:km
/trip::dist=42
```

`!:unit` is resolved by the schema-aware Denormalizer, exactly like an unannotated line under the head-type lift (§2.6.17):

- **A governing head exists** — the field is schema-defined: the head is inherited, and authored-unit conversion runs toward the declared unit (§2.7.3). A non-numeric governing head is the existing error — units attach only to `..num`-ordered types (§2.7).

— **No governing head** — a schemaless document, or an undefined field under a relaxed schema: the type resolves to **`float`**, the numeric type admitting every `..num` literal (integer spellings included), and the authored unit *stands* as the canonical unit — there is no declared unit to convert toward: `!float:km'/trip::dist=42`.

This is elision, not inference: the default keys off the annotation the author wrote — a unit is definitionally a `..num` property — never off the value's lexical shape (§2.6.2). An author who wants integer semantics writes `!int:km`. The two defaults compose into one rule: *no annotation inherits the head or falls back to `str`; a unit-only annotation inherits the head or falls back to `float`.*

The form is minimal by design. `!::unit` admits **no inline constraints** (`![0,100]:km` is ill-formed — constraints belong to the schema in inherit mode) and no trailing pattern literal; a provenance list composes as usual (`!::km?sensor1`); and it is a type-designating annotation for the stacking rule (§1.3.4). Like `$field` references, the form survives the schema-independent Compiler into `.raiv` (`!::km'/trip::dist=42`) and is resolved by the Denormalizer; it never appears in `.daiv`. DFA processing is unaffected: the six-rule classifier and the `'` split see the same line shapes, and the certified runtime consumes only `.daiv`.

2.7.5 Compound Units

Compound units use SI-style operators on the unit string:

Operator	Meaning	Example
`*`	multiplication	`!float:N*m` (torque)
`/`	division	`!float:m/s` (velocity)
`^`	integer exponent	`!float:m/s^2` (acceleration)

`1` is the **dimensionless unit** — the multiplicative identity. It is the canonical form of any value with no physical dimension (ratios, fractions, counts as dimensionless quantities, fully-cancelled compound expressions). It also serves as the sole numerator for "per X" ratios that have no named SI unit, e.g. `!float:1/s` (frequency expressed as inverse seconds, equivalent to Hz; the format does not auto-substitute named SI units for canonical compound forms).

The grammar is strict:

— No whitespace anywhere inside the unit expression.
— No parentheses.
— Integer exponents only — no fractional powers. Authoring may use negative exponents (`m*s^-1`); canonical form normalizes them into denominator factors (`m/s`), so canonical exponents are always positive integers ≥ 2.
— `/` is left-associative: `a/b/c` means `a/(b*c)`. Each denominator factor uses its own `/` in canonical form (see below).
— Operator side-assignment follows arithmetic, not SI typography: each factor's side is determined by its own introducing operator (`*` → numerator, `/` → denominator), so `a/b*c` means `(a*c)/b`, not `a/(b*c)`. To put several factors in the denominator, give each its own `/` (`a/b/c`).

2.7.6 Canonical Form: ASCII-Sorted Factors

Compound units are canonicalized at parse time so that semantically equivalent expressions produce byte-identical lines. The canonical form is a numerator (one or more `*`-joined factors) followed by zero or more denominator factors, each introduced by its own `/`:

```
factor[*factor]*[/factor[/factor]*]
```

The canonicalization rules:

1. Numerator factors are `*`-joined and sorted by **base unit name** (the letters, with a currency's leading `~` included), compared as ASCII byte strings. The exponent is **not** part of the sort key: after rule 3, each base name occurs at most once per side, so ties cannot arise. Thus `mm^2*mA` → `mA*mm^2` (`mA` < `mm` because `A` `0x41` < `m` `0x6D`), not `mm^2*mA` — the `^2` never enters the comparison.
2. Denominator factors each carry their own `/` and are sorted by base unit name as a list, by the same key as rule 1.
3. Repeated occurrences of the same factor on the same side collapse to positive integer exponents (`m*m` → `m^2`, `m/s/s` → `m/s^2`).
4. Authored negative exponents fold into denominator factors (`m*s^-1` → `m/s`); canonical exponents are always positive integers ≥ 2.
5. Integer exponents of 1 are omitted (`m^1` → `m`).
6. A factor that appears in both numerator and denominator cancels (`m*s/m` → `s`). If cancellation empties the numerator, the canonical numerator becomes `1` (the dimensionless unit) and the denominator survives (`m/m^2` → `1/m`). If cancellation empties both sides, the canonical form is `1` (`m/m` → `1`, `kg*s/kg/s` → `1`).
7. The dimensionless unit `1` is elided from canonical numerator and denominator unless it is the sole factor on that side (`1*m` → `m`, `1/m` is canonical, `1` alone is canonical).

Authored	Canonical
`m*kg/s^2`	`kg*m/s^2`
`s*A`	`A*s`
`m/s/s`	`m/s^2`
`N/s/m^2`	`N/m^2/s`
`m*s^-1`	`m/s`
`kg*m^2/A/s^3`	`kg*m^2/A/s^3` (already canonical)
`mm^2*mA`	`mA*mm^2` (sort by base name; `^2` ignored)
`s^-1`	`1/s`
`m/m`	`1`
`kg*s/kg/s`	`1`

Two unit expressions denote the same unit if and only if their canonical forms are byte-identical. The `.csaiv` and `.daiv` files always carry the canonical form; the `.raiv` likewise — canonicalization happens in the Compiler.

2.7.7 Built-in Units

A **frozen, enumerated set** of units ships embedded with every implementation — the SI base units, the named SI-derived units, the SI decimal prefixes applied to them, and a curated set of non-SI and US/imperial units. Like `std/core` for types, this set is part of every conforming distribution: implementations MUST recognize **exactly** the set enumerated in this section, and no `kfaiv.com` lookup is required to validate any unit in it. Membership is checked at build time — at the *lex* stage when the document declares no `.!units` import (the namespace is closed over the built-in set, §2.7.10), and by the Compiler when imports leave the set open; either way a unit name that is neither built-in nor resolvable is an `INVALID_CONSTRAINT_ERROR`. The certified Validator only byte-compares already-canonical unit strings and never converts (§2.7.2). Because unit names are ASCII (`unit-name` is `1*ALPHA`), non-ASCII symbols use ASCII spellings: `u` for micro (µ), `ohm` for Ω.

SI base units. (dimension symbol; conversion factor 1):

Unit	Quantity	Dimension
`m`	length	L
`kg`	mass	M
`s`	time	T
`A`	electric current	I
`K`	thermodynamic temperature	Θ
`mol`	amount of substance	N
`cd`	luminous intensity	J

Dimension symbols (`L M T I Θ N J`) and unit names occupy **separate namespaces**: a unit string is never parsed as a dimension, so the standard SI overloads (unit `N` newton vs dimension N; unit `T` tesla vs dimension T; unit `J` joule vs dimension J) do not collide.

Named SI-derived units. (coherent; each has conversion factor 1 and the SI expansion shown):

Unit	Quantity	SI expansion
`rad`	plane angle	`1`
`sr`	solid angle	`1`
`Hz`	frequency	`1/s`
`N`	force	`kg*m/s^2`
`Pa`	pressure	`kg/m/s^2`
`J`	energy	`kg*m^2/s^2`
`W`	power	`kg*m^2/s^3`
`C`	electric charge	`A*s`
`V`	electric potential	`kg*m^2/A/s^3`
`F`	capacitance	`A^2*s^4/kg/m^2`
`ohm`	resistance	`kg*m^2/A^2/s^3`
`S`	conductance	`A^2*s^3/kg/m^2`

Unit	Quantity	SI expansion
`Wb`	magnetic flux	`kg*m^2/A/s^2`
`T`	magnetic flux density	`kg/A/s^2`
`H`	inductance	`kg*m^2/A^2/s^2`
`lm`	luminous flux	`cd`
`lx`	illuminance	`cd/m^2`
`Bq`	activity	`1/s`
`Gy`	absorbed dose	`m^2/s^2`
`Sv`	dose equivalent	`m^2/s^2`
`kat`	catalytic activity	`mol/s`

SI decimal prefixes. A single prefix from the table below may be prepended to any SI base unit (for mass, to the gram `g`, not `kg`), any named SI-derived unit above, and the litre `L`, forming a distinct built-in unit whose factor is `10^power` times the base's. The unprefixed gram `g` (`0.001 kg`) is itself a member of the built-in set — it is the prefix-attachment base for mass, even though `kg` is the SI base unit. At most one prefix per unit; prefixes do **not** apply to non-SI/imperial units or to currencies. So `km`, `mm`, `MHz`, `kPa`, `mA`, `ns`, `mL`, `mg` are built-in; `kft`, `k~USD` are not.

Prefix	Power	Prefix	Power
`Y`	24	`d`	−1
`Z`	21	`c`	−2
`E`	18	`m`	−3
`P`	15	`u`	−6
`T`	12	`n`	−9
`G`	9	`p`	−12
`M`	6	`f`	−15
`k`	3	`a`	−18
`h`	2	`z`	−21
`da`	1	`y`	−24

Non-SI and US/imperial units. (curated; exact conversion factors to the SI base of the same dimension):

Unit	Quantity	= SI
`min`	time	`60 s`
`h`	time	`3600 s`
`d`	time	`86400 s`
`t`	mass	`1000 kg` (tonne)
`L`	volume	`0.001 m^3`
`in`	length	`0.0254 m`
`ft`	length	`0.3048 m`
`yd`	length	`0.9144 m`

Unit	Quantity	= SI
`mi`	length	`1609.344 m`
`nmi`	length	`1852 m`
`lb`	mass	`0.45359237 kg`
`oz`	mass	`0.028349523125 kg`
`gal`	volume	`0.003785411784 m^3` (US gallon)

Information units. The bit and the byte are a base dimension of their own — information, underivable from SI — and ship built in (dimension symbol `B`; the byte's factor is 8 relative to the bit):

Unit	Quantity	=
`b`	information	1 bit (base)
`B`	information	`8 b` (byte)

Two prefix families attach to `b` and `B`, and only the *multiplying* members — a millibyte is not a thing:

- **SI decimal** ($\times 1000^n$): `kB MB GB TB PB` and `kb Mb Gb Tb Pb`.
- **IEC binary** ($\times 1024^n$): `KiB MiB GiB TiB PiB` and `Kib Mib Gib Tib Pib`. The IEC prefixes are **restricted to information units**: a kibimeter (`Kim`) is invalid.

`KB` is rejected as ambiguous — at the unit *grammar* level, so no context (not even a custom `.faiv`) can claim the spelling. `K` is not an SI prefix; JEDEC `KB` means 1024 bytes while readers assume 1000, and kaiv refuses to guess. The rejection is a teaching error naming both resolutions: *KB is ambiguous: write kB (1000 B) or KiB (1024 B)* — likewise `Kb` for bits.

Data *rates* are ordinary compound units — `b/s`, `kb/s`, `MiB/s`, `GB/s` — and need no dedicated names. The telecom spellings are accepted as authored input and canonicalize to the decimal-bit compounds: `bps`→`b/s`, `kbps`→`kb/s`, `Mbps`→`Mb/s`, `Gbps`→`Gb/s`, `Tbps`→`Tb/s`. `kbps` is 1000 bits per second everywhere networks are sold — never kibibits.

Offset (affine) units are excluded. The unit model is factor-only — a unit is a pure scale relative to its base — so temperature scales with a non-zero offset (`°C`, `°F`) are **not** built in; kelvin (`K`) is the only built-in temperature unit. Consumers needing degrees Celsius/Fahrenheit apply the offset in application code.

Compound units inherit their dimensions from their factors by the same multiplication/division/exponentiation rules as the unit expression itself: `kg*m/s^2` has dimension `M*L/T^2` — the SI expression of force. The dimensionless unit `1` has the empty dimension product.

Within a dimension, **base units are reference units** (conversion factor 1); every other unit's factor above is relative to that base. Conversion factors are not consulted by the Validator — they exist for the build's authored-unit conversion (§2.7.3) and for consumer tooling that elects to convert between same-dimension values.

2.7.8 Custom units (`kfaiv.com`)

Domain-specific units (astronomical units, specialized SI extensions, industry-specific scales) are defined in unit library files served from `kfaiv.com` — the unit-definition registry, paralleling `ktaiv.com` for types and `ksaiv.com` for schemas. Resolution and override mechanics follow the same layered model as type registry resolution: a registry-override declaration overrides per prefix; build-time configuration overrides for matching prefixes; the default base URL is `kfaiv.com`.

A custom unit definition declares:

1. The unit's name and any aliases.
2. The unit's dimension, expressed by reference to other units.
3. A conversion factor to the dimension's base unit.

2.7.9 Unit Definition Files (`.faiv`)

Unit libraries are **`.faiv`** files (**Factor** Attributed Information Values — a unit definition is fundamentally a conversion-factor declaration). The extension pairs with its registry domain like the rest of the family (`ktaiv.com`→`.taiv`, `ksaiv.com`→`.saiv`, `kfaiv.com`→`.faiv`). A `.faiv` file is `caiv`: a `.!faiv [VERSION] LIBRARY-ID` header, then a **definition line** above each `&name=`:

```
.!faiv astro/units

// Astronomical unit
m 1.495978707e11
&au=
&AU=au

// Custom currency with a rate source
$ @https://rates.example.com/v1?code={code}&at={timestamp}
&~XYZ=
```

The definition line is `DIMENSION [FACTOR | @RATE-URL]`:

— **Dimension by unit reference.** The dimension is a unit expression (`m`, `kg*m/s^2`, `1` for dimensionless) referencing built-in units or units defined *earlier in the same library* — never the seven abstract dimension symbols, which include the non-ASCII `θ`. The dimension of kelvin is written `K`. Currencies use the literal dimension **`$`**.

— **Factor.** A positive decimal (optional fraction and exponent: `1.495978707e11`), the conversion factor to the dimension's **base unit** — required for every physical dimension, **forbidden for `$`** (exchange rates are external and time-varying, §2.7.12).

— **Rate source (currencies only).** In place of a factor, a currency definition `MAY` declare `@` followed by a URL template with the placeholders `{code}` and `{timestamp}` (the `@instant` qualifier of the value's provenance, in the canonical dashed form), letting consumer tooling pull the rate contemporaneous with the value. The Validator never fetches it.

— **Names and aliases.** `&name=` (empty right side) binds the pending definition line; `&alias=name` (right side naming an earlier definition) declares an alias. A currency definition is named by its code: `&~XYZ=`. Aliases are authoring-side spellings: a unit authored by an alias canonicalizes to the primary name, exactly as the telecom aliases do (§2.7.7) — canonical `.daiv`/`.csaiv` lines never carry an alias.

```
faiv-def-line  = dimension [ 1*ws ( factor / rate-source ) ] eol
                 ; factor REQUIRED for physical dimensions,
                 ; absent or a rate-source for "$"
dimension      = unit-expr / "$"
factor         = 1*DIGIT [ "." 1*DIGIT ] [ ( "e" / "E" ) [ "+" / "-" ] 1*
   DIGIT ]
rate-source    = "@" uri              ; "{code}" / "{timestamp}"
   placeholders
faiv-name-line = "&" ( unit-name / currency ) "=" [ unit-name / currency ]
   eol
faiv-decl  = ".!faiv" [ 1*ws version ] 1*ws library-path eol
```

A rate-source URL may contain = (query strings); the rule-6 classification priority (§1.3.1) applies to `.faiv` definition lines exactly as it does to patterns.

2.7.10 Referencing Custom Units: `.!units`

Unit expressions carry **bare names** — a library path inside a unit string is impossible, since `/` already means division (`:astro/units/au` would parse as `astro ÷ units ÷ au`). A document therefore imports the unit libraries it uses with the **`.!units LIBRARY-ID`** declaration, paralleling `.!types`:

```
.!kaiv
.!units astro/units

!float:au
/probe::distance=1.5
```

Rules mirror `.!types`: multiple imports are allowed; a unit name MUST resolve against the built-in set or exactly one imported library (ambiguity across imports is an error, and a custom unit MUST NOT shadow a built-in name); `.!units` is valid in `.kaiv`, `.saiv`, and `.taiv`, and survives into canonical output as resolution metadata. Resolution follows the standard layered model with `{base}/{library/path}.faiv` and `kfaiv.com` as the Layer 4 default. With no `.!units` declaration in the document, the unit namespace is closed over the built-in set and membership is checkable at lex time; with imports present, membership is resolution-dependent and checked at compile time (an unknown name remains an `INVALID_CONSTRAINT_ERROR` condition). Canonicalization is unchanged — custom names sort with built-ins by the same base-name key, and the canonical `.daiv`/`.csaiv` line carries the bare canonical unit string.

2.7.11 Units on Named Types

A named type derived from `!int` or `!float` may carry a unit at its definition site:

```
.!taiv acme/distances

// A non-negative length expressed in kilometres.
!float:km [0,]
&distance_km=
```

The unit is part of the named type's canonical identity. In data files, the unit appears on the canonical line alongside the fully-qualified type name:

```
!acme/distances/distance_km:km'/trip::length=42
```

The unit is **immutable at use sites** — unit rules are deliberately *not* the constraint-narrowing model:

- A unit-carrying named type cannot be re-united or stripped at a use site. A "narrowing" from `:km` to `:m` would leave the accepted value set untouched while rescaling the meaning of every number — and of every inherited range and default with it — which is reinterpretation, not narrowing; and the canonical line would forever carry a type name contradicting its unit (`distance_km:m`). The same shape in another unit is a sibling type (`&distance_m=`) or the inline core form (`!float:m`) — each says what it means.
- A named type *without* a unit MAY be annotated with one at a use site — additive information, no reinterpretation; the field is then unit-carrying under the ordinary rules.
- Authoring-side unit *conversion* is a different, supported operation: a data line may express its value in any same-dimension unit, converted by the build into the field's declared unit (§2.7.3).

2.7.12 Currencies

Currencies are a variant of units distinguished by a tilde prefix on the unit code:

```
!float:~EUR'/order::total=99.50
!float:~USD'::price=12.00
!int:~JPY'::amount=1500
```

A currency code is ~ followed by **exactly three uppercase ASCII letters** — the ISO 4217 shape. Well-formedness (three uppercase letters) is required; **membership** in the ISO 4217 register is **not** checked — a well-formed but unassigned code (`~XYZ`) is accepted, because a currency carries no dimension breakdown or conversion factor to validate against, and the register is external and time-varying. All currency units share a single dimension, written $ (the dimension symbol for monetary value). Compound expressions containing currencies — `!float:~USD/h` for an hourly rate, `!float:~EUR*kg^-1` for a price-per-mass — are formed and canonicalized by the same rules as physical compound units. The tilde stays attached to the currency factor under canonicalization.

Currency definitions deliberately omit conversion factors: exchange rates are time-dependent and external to any unit definition. The Validator therefore performs no cross-currency conversion, and mixing multiple currencies of the same $ dimension within a document is permitted. Converting between currencies is a consumer concern, typically resolved by joining the value with an exchange-rate source keyed off the `@timestamp` qualifier of the value's provenance.

A currency definition on `kfaiv.com` MAY declare a **rate-source URL template** in place of a conversion factor (`$ @https…://?code={code}&at={timestamp}`, §2.7.9), so that consumer tooling can pull the exchange rate contemporaneous with the `@timestamp` qualifier of a value's provenance. The Validator never consults it.

2.8 Constraints

The constraint forms — pattern, span, range, enumeration, length — are introduced in the Type System section above. This section covers their use directly on type annotations (inline, without a named type) and the length constraint in detail.

2.8.1 Inline Constraints on Type Annotations

Any constraint form may be applied directly in a type annotation, immediately following the type sigil:

- `!int[1,65535]` — range, on a type that already supplies a span (`..num` from `&int`).
- `!str/[a-zA-Z0-9.-]+/` — pattern.
- `!str{https,http}` — enumeration.

Multiple constraints of different kinds may appear on the same annotation:

```
!str/[a-zA-Z0-9.-]+/#[1,253]'/server::host=
```

Here the pattern constraint `/[a-zA-Z0-9.-]+/` and the length constraint `#[1,253]` are independent predicates applied to the same field; both must be satisfied.

2.8.2 Length Constraints

A **length constraint** applies a constraint to the *length* of the value rather than to the value itself. It is written as a `#` prefix immediately before the constraint bracket or brace: `#[min,max]` or `#{a,b,c}`.

The length semantics depend on the type:

- For `!str`: length is the character count.
- For arrays (`@name`): length is the element count.
- For `!b64`: length is the decoded byte count. The schema compiler **translates** decoded-byte bounds into encoded-character bounds at lowering time — exact for unpadded base64url (`n` bytes occupy `⌊4n⌋/3 + {0,2,3}[n mod 3]` characters), so `!b64#[16,16]` compiles to `#[22,22]` and `!b64#{16,24,32}` to `#{22,32,43}` — and the Validator's counting stays character-based with no decoding in the certified runtime.

Valid Length Constraint Forms

- `#[min,max]` — length must fall in the range. For example, `!str#[1,253]` constrains the character count of a string to 1–253.
- `#{a,b,c}` — length must be exactly one of the listed values. For example, `!b64#{16,24,32}` constrains the decoded byte count to 16, 24, or 32 (AES key sizes).

The `#/regex/` form is not valid: applying a pattern constraint to a length is not meaningful.

Examples. On strings:

```
!str/[a-zA-Z0-9.-]+/#[1,253]'/server::host=
!str#[8,128]'/user::password=
!str#[2,2]'/address::country_code=
```

On arrays (in a `.saiv` schema — the annotation constrains the element type, the length constraint counts elements; the vector operator `;=` declares the scalar array, §7.3):

```
!str#[1,10]
/@tags;=

!float#[2,2]
/@coords;=
```

On binary:

```
!b64#[32,32]'::hash=
!b64#{16,24,32}'::key=
```

Composability. A field may carry both a value constraint and a length constraint simultaneously. The two are independent predicates:

```
!str/[a-zA-Z0-9.-]+/#[1,253]'/server::host=
```

The `/[a-zA-Z0-9.-]+/` checks the value; the `#[1,253]` checks the character count. Both must be satisfied. Order within the type annotation is insignificant — they are commutative predicates on the same field.

DFA Compatibility. The `#` prefix is a single additional DFA state transition. The DFA sees `#` after the type annotation, enters *length-constraint mode*, then parses the following `[min,max]` or `{a,b,c}` using the same constraint-parsing path as value constraints. At validation time the check is `len(value) ∈ constraint` instead of `value ∈ constraint`. No new parsing machinery is required.

Narrowing. Length constraints follow the same narrowing rules as value constraints: a narrowed constraint must be a subset of the parent's constraint.

```
// Base type: hostname allows up to 253 chars
!str/[a-zA-Z0-9.-]+/#[1,253]
&hostname=

// Narrowed: short hostname (embedded devices)
&hostname #[1,63]
&short_hostname=
```

The `#[1,63]` narrows the length constraint from `[1,253]` to `[1,63]`. The value constraint (`/[a-zA-Z0-9.-]+/`) is inherited unchanged.

3 Level 1: Trees

Level 1 introduces tree-shaped data: arrays (`@`), namespaces and structs (`/`), the field projection operator (`::`), and the assignment operators that build them (`+=`, `;=`, `:=`). Level 1 still parses in constant memory.

3.1 Arrays

An **array** is an ordered collection of **elements**. Arrays use the `@` sigil, and plural form in identifiers is RECOMMENDED: e.g. `/@hosts`, `/@roles`.

There are two syntax variants for defining arrays: value accumulation, and inline assignment.

With value accumulation, the **array appending operator** `+=` is used in order to append a value to an array. For example:

```
/@hosts+=localhost
/@hosts+=example.com
```

The inline syntax variant of array definition uses an **array extending operator** `;=` to extend the array with a list of semicolon-separated values. For example:

```
/@hosts;=localhost;example.com
```

The two variants are *syntactically different*, hence parsers emit different entries. However, they are *semantically equivalent* — applications MUST treat them identically. The two variants can be intermixed in building one and the same array structure:

```
/@numbers+=one
/@numbers;=two;three
/@numbers+=four
```

The extending syntax can be thought of as syntactic sugar for the appending syntax; if the data contains semicolons, simply use the `+=` operator with one element per line instead of the multi-element `;=` operator.

The extending/appending syntax terminology is based on the premise that the array is constructed by accumulation. However, arrays are often used to represent tuples or vectors, defined in one command and never appended or extended, for example:

```
/@numbers;=one;two;three;four
```

The `;=` operator is therefore also called the **vector assignment operator**. The terms *array extending operator* and *vector assignment operator* are synonyms — we use one or the other depending on context.

3.1.1 Arrays Are Namespaces with Integer Fields

The indexed namepath syntax naturally handles **arrays of namespaces** — the construct that all major target formats support. The challenge: how to represent a repeated composite type without introducing syntactic nesting. The answer is that both authoring forms reduce to the same indexed namepath representation in canonical form.

Authored Form — Inline (+:= Operator, Array-Append Namespace-Assignment)

```
/@servers+:=host=a|port=1
/@servers+:=host=b|port=2
```

Authored Form — Section Blocks

```
[/@servers]
host=a
port=1
[/@servers]
host=b
port=2
[]
```

Canonical Form (Both Reduce to the Same Output)

```
!str'/@servers/0::host=a
!str'/@servers/0::port=1
!str'/@servers/1::host=b
!str'/@servers/1::port=2
```

Both the inline `+:=` syntax and the section block `[/@key…][]` syntax are **lexer-level syntactic sugar**. The Compiler canonicalizes both into indexed namepaths. The Application layer sees only the flat canonical stream.

Scalar arrays vs. namespace arrays. The distinction between the two kinds of arrays is visible in canonical form through the operator between the array name and the index. Scalar array elements (`!int'/@ports::0=8080`) use `::` before the index — the index is the field, and `::` projects it as a leaf value. Namespace array elements (`!str'/@servers/0::host=a`) use `/` before the index — the index descends into a child namespace, and the fields within are then projected with `::`. The same operator rule that governs all other path steps (`::` exits the tree, `/` stays in it) determines the array kind.

3.1.2 Section Block Semantics

Section blocks (`[/@path]` … `[]`) and namespace blocks (`(/path)` … `()`) are authoring sugar that the Compiler expands into fully-indexed canonical namepaths. The Lexer only classifies the delimiter lines (§1.3.1); all pairing and indexing is a Compiler concern. The Compiler maintains a **block stack** and, for each array path, a monotonic **element counter**.

Canonical index spelling. An element index has exactly one spelling — the counter's decimal value, with no leading zeros (`index` in §10) — parallel to the single canonical representation of names (§2.3.2). An all-digit namepath segment with a leading zero fails the

Lexer's key check (`INVALID_KEY_ERROR`), so `/@a/0` and `/@a/00` can never denote two elements or alias one. A collection line covers only canonical spellings: a hand-assembled index beyond an implementation's documented magnitude limit (§9.9) is simply not covered — an undefined field, tolerated by a relaxed schema and rejected by a strict one.

Section-open `[/@path]`. Opens a new element of the array at `/@path` (resolved against the enclosing block's prefix). The element index is that array path's next counter value, starting at `0`; content lines that follow are emitted under `/@path/<index>` until the block is closed or superseded. A section-open may carry a Level 2 table header after the path (`[/@servers host=! min=1]`); the array path is the first whitespace-separated token and the header is consumed by the schema layer (§4.2), not by the path logic.

A repeated open advances the element. A `[/@path]` for the array that is currently the innermost open block — with no intervening `[]` — closes the current element and opens the next (counter += 1). This is how `[/@servers…][/@servers…][]` yields elements `0` and `1` (conformance vector `valid/008` in the spec repository).

A new section-open closes the current one. Opening any section block while a *different* section block is the innermost open block closes (abandons) the current one and starts the new array at its own path's next index. Consequently **section blocks do not nest inside one another**: a nested array (a `/@outer` whose elements each contain a `/@inner`) is authored with indexed namepaths or the `+:=` / `;=` forms (§3.1), not by nesting `…[]` inside `…[]`. Namespace blocks, by contrast, *do* nest — a `(/meta)` opened inside a section-block element extends the current prefix (`/@servers/0/meta`) and is closed by `()`.

Close `[]` / `().` `[]` pops the innermost block iff it is a section (array) block; `()` pops it iff it is a namespace block. A close whose kind does not match the innermost open block — or a close with no block open — is a tolerated no-op that leaves the stack unchanged, not an error. Blocks still open at end of input are closed implicitly; no error is raised (implementations MAY warn).

Counters persist per array path. Element counters are keyed by the fully-resolved array path and live for the whole document. Reopening an array path later in the file (after it was closed) continues its index rather than resetting: arrays are append-only, and an array's elements are numbered in first-seen order across every section block and every `+=` / `+:=` line that targets that path.

3.1.3 Mixed Arrays

A **mixed array** contains elements that may be either scalar or namespace, with the operator before the index discriminating per element:

```
!str'/@items::0=hello
!str'/@items/1::name=Alice
!int'/@items::2=42
!int'/@items/3::x=1
```

```
!int'/@items/3::y=2
```

This represents `["hello", {"name": "Alice"}, 42, {"x": 1, "y": 2}]`. Elements 0 and 2 are scalars (indexed with `::`); elements 1 and 3 are namespaces (indexed with `/`). No new syntax is required — the existing `::` / `/` operator distinction naturally handles mixed arrays, enabling kaiv to represent all JSON lists faithfully.

3.1.4 Nesting Depth

With indexed namepaths, nesting depth is no longer an architectural constraint — it becomes a **schema-bounded policy**.

Depth 2 Example

A namespace array `/@matrix`, each element containing a nested scalar array `@values`:

```
!str'/@matrix/0/@values::0=a
!str'/@matrix/0/@values::1=b
!str'/@matrix/1/@values::0=c
```

Depth 3 Example

```
!int'/@cube/0/@planes/0/@points/0::x=1
```

The Lexer does not care about depth — it sees a key string, `=`, and a value. The Parser tracks N index counters for N nesting levels; each counter is a single integer. There is no stack, no recursion, no pushdown automaton. The schema defines the maximum nesting depth for any given field, and the compiled schema (`.csaiv`) declares that maximum. The Validator pre-allocates exactly that many index counters at startup. Memory usage is static and schema-determined.

> **Flat grammar.** kaiv's lexical grammar is flat: every line is a self-contained `(type, key_path, value)` tuple. Structural nesting is encoded in the namepath, not in the syntax. The schema bounds the nesting depth for any given document, and the runtime validator's memory usage is statically determined by the schema.

3.2 Structs

Structs are key-value data structures using the `/` prefix. Structs are assigned to using the **struct assignment operator** `:=`, where the right side is a pipe-separated sequence of key-values: a sequence of scalar and array assignments with EOL replaced with pipes (and final EOL stripped). For example:

```
/server:=host=localhost|port=8080
```

Structs can contain arrays as values:

```
/meta:=category=info|@tags;=doc;manual;syntax|published=true
```

The semantic on the right side is the same as for arrays (defined in the Arrays section above) with the only syntactic difference being EOL vs pipes. Therefore, the same array `@tags` could have been constructed by accumulation:

```
/meta:=category=info|@tags+=doc|@tags+=manual|@tags+=syntax|published=true
```

Note that, as arrays are ordered by definition, this implies that structs are also ordered structures (otherwise the ordering of `@tags` would be lost in the second example).

As recursion is not allowed in kaiv (core principle), structs cannot contain nested structs. However, this functionality is expressed in a different syntactic form using namespaces.

Because the pipe is the field separator, a literal `|` cannot appear in a value inside a struct assignment — there are no escape sequences (§9.5). The same rule as for `;=` applies: if the data contains pipes, do not use the inline sugar. Author the fields as separate namespaced field lines (`/server::motd=a|b`), where the value after `=` is verbatim and may contain any character. The same restriction and the same escape hatch apply to `+:=` (use section blocks instead).

3.3 Fields and Literals

Before we define namespaces, we need to introduce the concepts of **fields** and **literals**.

Keys and *values* are syntactic concepts designating whatever is on the left and right side of the assignment (or appending/extending) operator, respectively. *Fields* and *literals* are similar concepts, but operating on the semantic plane: a field is an identifier that is tied to a literal, regardless of how this relation is established syntactically. A literal can be a scalar or an array value, but *not* a struct value.

With simple string values assigned to simple identifiers, fields coincide with keys, and literals with values:

```
x=1
```

Here, `x` is both a key and a field, and `1` is both a value and a literal.

With structs, however, we encountered a situation where both fields and literals are on the right side of the assignment operator, they are both part of the value:

```
/server:=host=localhost|port=8080
```

Here, the *value* is `host=localhost|port=8080`, where `host` and `port` are *fields*, and `localhost` and `8080` are *literals*.

3.4 Namespaces

A **namespace** is a slash-separated prefix in a key. Namespaces always carry the `/` prefix. An example of a namespace is `/app/db`.

The fact that namespaces and structs use the same prefix is not accidental. As both namespaces and structs add hierarchy to data (namespaces do so on the left side of the assignment operator, and structs on the right), in kaiv they both represent one and the same concept.

3.4.1 Namespaced Structs

An example of a namespaced struct (a struct with a namespace prefix) is:

```
/app/db/server:=host=localhost|port=5432
```

Here `/app/db/server` designates the namespaced struct. (One might think of `app/db` as being the namespace and `server` being the struct, but as we will see soon, the two are so organically tied in kaiv that drawing this line feels artificial — this is why the prefix `/` prefixes the entire identifier, and we don't see `/server` in the key.)

3.4.2 Namespaced Field Keys

With **field keys** — keys designating simple string values or array values, but not struct values — the syntax for applying a namespace is different: for example, applying the namespace `/book/intro` to the field `num_chars` reads:

```
/book/intro::num_chars=1500
```

Here, `::` is the field projection operator (§1.4.2).

Thus we have come full circle: it should make sense now why the struct assignment operator `:=` uses this particular syntax.

```
/book/intro:=num_chars=1500|num_paragraphs=3
```

The struct assignment operator `:=` tells us that the fields are to be defined on the right side of the assignment operator. On the other hand, here:

```
/book/intro::num_chars=1500
```

we first access the field, then assign a simple literal to it.

And so, when keeping our eyes on the fields and literals, the line where namespaces end and structs begin becomes irrelevant.

3.5 Namespace Blocks

Namespace blocks are an **authoring-layer scoping construct** that allows a group of lines to share a common namespace prefix — and optionally a scoped schema for sub-namespace validation. They are syntactically parallel to array section blocks (`[/@key…][]`) and reduce to flat namepath lines in canonical form.

3.5.1 Syntax

```
(/name)
  field1=value1
  field2=value2
()
```

With an optional schema annotation:

```
(/name schema:schema-id)
  field1=value1
  field2=value2
()
```

The opening delimiter `(/name)` or `(/name schema:schema-id)` opens the block. Every line inside the block is prepended with `name/` (accumulating with any outer block prefix). The closing delimiter `()` ends the block.

The block syntax is consistent with the existing block family:

Block syntax	Scope	Effect
`[/@name…][]`	Array element	Each line inside is prepended `/@name/n::`
`(/name)…()`	Namespace	Each line inside is prepended `name/`
`(/name schema:ID)…()`	Namespace with schema	Each line prepended `name/` and contents validated against the named schema (e.g. `schema:crypto/rsa-params`)

3.5.2 Basic Example

Authored kaiv

```
.!kaiv
.!schema:acme/server-config

(/server)
host=localhost
port=8080
()
```

Canonical Output

```
.!daiv
.!schema:acme/server-config
!str'/server::host=localhost
!str'/server::port=8080
```

The namespace block is purely authoring sugar. The canonical output is identical to writing `/server::host=localhost` and `/server::port=8080` directly.

3.5.3 Namespace-Scoped Schemas

When a `schema:` annotation is included, the namespace block's contents are validated against the named member of the schema set the parent schema declares for that namespace (see below); the parent's own scan governs everything outside the block. This is **DFA composition**: the parent schema delegates validation of the block's contents to a sub-DFA, then resumes the parent DFA after the block closes.

```
.!kaiv
.!schema:x509/algorithm-identifier

algorithm=rsa

(/parameters schema:crypto/rsa-params)
modulus=00:ab:cd:...
exponent=65537
()
```

Changing the algorithm and schema:

```
algorithm=ecdsa

(/parameters schema:crypto/ecdsa-params)
curve=P-256
public-key=04:ab:cd:...
()
```

The parent schema (`x509/algorithm-identifier`) declares that `parameters` accepts a **discriminated schema set** — analogous to how a tagged union declares a set of type alternatives. The block's `schema:` annotation selects which member validates its contents. Ordered keys ensure that `algorithm` is resolved before the `(/parameters …)` block is entered, making the selection available at parse time.

Declaring the set (`.saiv`). The parent schema declares a delegated namespace *ostensively*, exactly as the map schema block does (§7.5): the same block the data writes, carrying the *set* of admissible schemas — `schema:` with pipe-separated alternatives — and an empty body:

```
.!saiv x509/algorithm-identifier

{rsa,ecdsa}
algorithm=

(/parameters schema:crypto/rsa-params|crypto/ecdsa-params)
()
```

Rules:

— A data block names exactly **one** schema — the selection; a parent-schema block names **one or more** — the set. Any member validates the block regardless of sibling field values; value-correlated selection (`algorithm=rsa` ⇒ `rsa-params`) is a corpus-level concern outside this document's surface.

— A delegated namespace declares **no inline fields** in the parent: the block body is empty, and a parent schema that both delegates a namespace and declares fields under it — or declares an empty or duplicate-member set — is rejected (`SchemaDelegationError`).
— Members resolve like any other schema reference (§2.1.4). A member may itself delegate; an implementation MUST bound the chain and reject a cycle exactly as for inheritance (`SchemaInheritanceCycleError`).
— Inside the namespace, the *member's* compiled contract governs — its constraints, its `strict` modifier, its `.!provenance` level. The parent's own modifiers govern everything outside.

3.5.4 Nested Namespace Blocks

Namespace blocks compose naturally. The accumulated prefix grows with each nesting level:

```
(/server)
  host=localhost
  port=8080

  (/db schema:db/postgres-config)
  host=db.local
  port=5432
  max_connections=100
  ()

  (/cache schema:cache/redis-config)
  host=redis.local
  ttl=300
  ()
()
```

Canonical Output

```
!str'/server::host=localhost
!str'/server::port=8080
!str'/server/db::host=db.local
!str'/server/db::port=5432
!str'/server/db::max_connections=100
!str'/server/cache::host=redis.local
!str'/server/cache::ttl=300
```

Each nested namespace block scopes its lines with the accumulated path prefix. Each `schema:` annotation validates its block's contents independently. The parent schema validates the top-level structure.

3.5.5 Canonical Form

The namespace block is authoring sugar. In canonical form it expands to flat `!type' namepath=value` lines — just like array blocks. The `schema:` annotation is the one part that must survive: the Compiler MUST emit it as a scoped declaration line —

```
.!schema:/parameters crypto/rsa-params
```

— the standard namespace-qualified form of the `.!schema` declaration (`schema-registry-ref` in §10). The emission is mechanical and schema-free, so the schema-independent Compiler can perform it. When the parent schema *delegates* the namespace (§3.5.3), this declaration is the block's **discriminant**: exactly as a union's data line carries its active variant, the canonical document carries its active schema, so `.daiv` stays self-contained and the Validator selects the member from the artifact alone (§7.6) — a delegated namespace whose declaration is absent fails validation (`DelegationSchemaError`). For a namespace the parent does not delegate, the declaration is a per-document assertion that tooling MAY validate against as an additional contract.

3.5.6 Encapsulated Hub Schema Extension (`.!schema:/ns hub/x`)

The encapsulated extension syntax is a document-level declaration (introduced under §2.1.3 in Level 0) that scopes an entire hub schema's fields under a sub-namespace rather than merging them at root. This is distinct from the namespace block `schema:` annotation — it is a schema **inheritance** declaration (a compiled-in merge of one known contract), not a local validation delegation (the one-of-N case: §3.5.3, compiled per §7.6).

The complete syntax table:

Form	Syntax	Effect
Flat — space-separated	`.!schema hub/x`	Hub fields merged at root
Flat — registry ref	`.!schema:hub/x`	Hub fields merged at root
Encapsulated (namespace)	`.!schema:/ns hub/x`	Hub fields scoped under `/ns`
Encapsulated URL	`.!schema:/ns https://url`	Hub fields scoped under `/ns`
Array-element	`.!schema:/@arr hub/x`	Hub schema applied to each element of `/@arr`
Array-element URL	`.!schema:/@arr https://url`	Hub schema applied to each element of `/@arr`

Given:

```
.!schema:/server hub/server-endpoint
```

where `hub/server-endpoint` declares `::host`, `::port`, `::timeout_ms?`, the fields in the extending schema are addressed as `/server::host`, `/server::port`, `/server::timeout_ms` (optional).

Multiple instantiation of the same hub. The same hub may be encapsulated under different namespaces in the same document — giving two fully independent instances with independent field values:

```
.!schema:/upstream hub/server-endpoint
.!schema:/downstream hub/server-endpoint
```

This yields `/upstream::host`, `/upstream::port`, `/downstream::host`, `/downstream::port` as four independent fields. Flat extension cannot achieve this — a second `.!schema:hub/server-endpoint` would be a duplicate.

Auto-derived mappings. The registry auto-derives the mapping edge at publish time: `.!schema:/ns hub/x` generates the declaration `ns::field → hub/x::field` for every field in `hub/x`. No hand-written `.maiv` file is needed.

Constraint inheritance. Hub field constraints are inherited under the namespace prefix. The namespace prefix does not change constraint semantics — `/server::port` is still constrained to `!std/net/port` (0–65535) if the hub declares it so. The schema author may narrow the constraint (e.g. `[1024,65535]`) but may not widen it.

Compiled merge. In the extending schema's `.csaiv`, the inherited schema's compiled field and collection lines merge at the `.!schema` declaration's position, in the inherited schema's order (their namepaths carrying the namespace or array-element prefix, and any foreign-key target paths re-anchored under it). A field the extending schema redeclares — the narrowing case above — replaces the inherited line *in place*, keeping the inherited field order for the parallel scan. The inherited schema's own header declarations (`.!saiv`, `.!provenance`) do not carry over: the extending schema's header governs. Inheritance chains resolve recursively; an implementation MUST bound the chain depth and report a cycle as a `SchemaInheritanceCycleError` (§11.2).

3.5.7 Array-Element Hub Schema Extension (`.!schema:/@arr hub/x`)

The array-element extension applies a hub schema to **every element** of a named array, rather than to the document root or a single namespace.

```
.!schema:/@servers hub/server-endpoint
```

where `hub/server-endpoint` declares `::host`, `::port`, `::timeout_ms?`, each element of `/@servers` is addressed as `/@servers/n::host`, `/@servers/n::port`, `/@servers/n::timeout_ms` (optional). The Validator applies the hub schema's field rules to every array element independently. Required hub fields MUST be present in every element; optional fields (`?=`) may be absent.

Relationship to table definitions. The `.!schema:/@arr hub/x` declaration complements the table-definition syntax (`[/@arr …][]` — see §4). A table definition declares collection-level constraints (UNIQUE, FOREIGN KEY, cardinality); the array-element schema declaration declares the structural shape each element must conform to. Both can appear together:

```
# element fields from hub/server-endpoint;
# host unique, at least one element required:
.!schema:/@servers hub/server-endpoint
[/@servers host=! min=1]
```

3.5.8 Schema Composition for Namespace Blocks

The key architectural property of namespace-scoped schemas is that they preserve the parallel-scan structure of the Validator's validation:

1. The Validator's scanner reaches the delegation line for the namespace (§7.6).
2. It suspends the parent schema scan and delegates to the member `.csaiv` selected by the document's scoped `.!schema` declaration — the discriminant (§3.5.5).
3. The member's scan validates the namespace's lines in lockstep.
4. When the namepath prefix leaves the namespace, control returns to the parent scan at the next schema line (canonical form has no block delimiters — the prefix change is the boundary).

This is **schema composition, not recursion**. The set of valid sub-schemas is declared in the parent schema and all sub-schema files are pre-loaded at the same time as the parent. No dynamic dispatch, no heap allocation, no recursion. The composed validator retains constant-memory properties.

Composition adds one level of delegation per nesting level; since nesting depth is bounded by the schema, the composed validator's state is also statically bounded.

4 Level 2: Tables

Level 2 introduces **table definitions** — collection-level constraints on arrays of namespaces that cannot be expressed at Level 1. These are kaiv's equivalents of SQL's `UNIQUE`, `FOREIGN KEY`, and row-cardinality constraints. Validation requires a post-scan pass with O(N) memory for uniqueness hash sets, isolated from the Level 0–1 constant-memory parallel scan.

See §B.2 for the certification boundary that Level 2 constraints operate within.

4.1 The Gap: No Collection-Level Constraints

kaiv's Level 1 data model supports **arrays of namespaces** — ordered, indexed sequences of composite records. At the data level, the `[/@arr…][]` section block syntax and the inline `/@key+:=field=val|field=val` form both canonicalize to indexed namepaths (`/@servers/0::host=a`, `/@servers/1::host=b`). This is expressive enough to represent anything from a list of server configs to a product catalog.

At the schema level, however, there is a gap: field definitions inside an array element can declare types, constraints, and required/optional status — but nothing can say "the `host` field must be unique across all elements" or "the `department` field must reference a value that exists in another array" or "there must be at least one element." These are

collection-level constraints — constraints that apply to the array as a whole, not to individual element fields.

This gap means kaiv's DDL, prior to Level 2, has no equivalent for SQL's `UNIQUE`, `FOREIGN KEY`, or `CHECK (COUNT...)` on a table. Table definitions bridge that gap.

4.2 Table Declaration Syntax

At the schema level, a table definition uses the same `[/@name…][]` block syntax as the data-level array section block — but with constraint annotations on the opening line and no values. The block declares both the element field schema and the collection-level constraints in one place.

Full Syntax

```
[/@arrayname field1=!|field2=!,field3=! field4=/@other/*::ref min=N max=M]
  ...element field definitions...
[]
```

Where the table declaration line (`[/@arrayname …]`) contains zero or more of:

Component	Syntax	Meaning
Single unique	`field=!`	Values of `field` must be unique across all elements
Compound unique	`field1=!,field2=!`	The *combination* must be unique across all elements
Independent constraints	`…|…`	Pipe separates independent unique/ref constraint groups
Foreign key reference	`field=/@path`	Field values must appear in the referenced array field
Minimum cardinality	`min=N`	Array must contain at least N elements
Maximum cardinality	`max=M`	Array must contain at most M elements

Minimal Example — Unique Host Within a Server List

```
[/@servers host=!]
!str
host=
!int[1,65535]
port=
[]
```

This is identical to the data-level `[/@servers…][]` block except that the opening line carries constraint annotations. The element field definitions inside follow the same schema syntax as any other field schema.

4.3 Unique Constraints

A `field=!` on the table declaration line declares that the named field must have a distinct value for every element in the array. This is the table-level equivalent of SQL's `UNIQUE` constraint (or `PRIMARY KEY` when combined with `=` inside the element body).

Single-Field Unique Constraint

```
[/@users username=!]
!str
username=
!str
email=
[]
```

Every user must have a distinct `username`.

Compound Unique Constraint

`field1=!,field2=!` declares that the *combination* of those fields must be unique, not each field individually:

```
[/@servers host=!,port=!]
!str
host=
!int[1,65535]
port=
[]
```

The pair `(host, port)` must be unique — two servers may share the same `host` (different ports) or the same `port` (different hosts), but not both.

Multiple Independent Unique Constraints

The pipe separator `|` introduces a second independent uniqueness requirement on the same array:

```
[/@servers id=!|host=!,port=!]
!str
id=
!str
host=
!int[1,65535]
port=
[]
```

This declares two independent constraints: `id` must be globally unique (any two elements must differ in `id`), AND the `(host, port)` combination must also be globally unique. Both constraints must hold simultaneously.

4.4 Foreign Key References

A `field=/@path` on the table declaration line declares that the field's value in every element must appear as a value in another array's field. The path uses qaiv path syntax — but Level 2 needs (and a Level 2 implementation MUST support) only the fixed shape pinned as `fk-path` in §10: `/`-descent to an array, the any-element wildcard `*`, and a single `::field` projection (`/@departments/*::name`). The full qaiv language (predicates, comparison operators, additional wildcards) is a superset defined in QUERY.md, which is an early design document; the `fk-path` subset defined here is normative and stable independently of it.

Example

```
[/@employees department=/@departments/*::name]
!str
name=
!str
department=
[]
```

Every `department` value in the `employees` array must exist as a `name` value in the `departments` array. This is the kaiv DDL equivalent of:

```
FOREIGN KEY (department) REFERENCES departments(name)
```

The qaiv path `/@departments/*::name` reads: "in the `departments` array, any element (`*`), project the `name` field." This reuses the existing query language path syntax — no new syntax is introduced.

Foreign Key Combined with Uniqueness

```
[/@orders id=!|customer=/@customers/*::id]
!str
id=
!str
customer=
[]
```

`id` must be unique within `orders`, and every `customer` value must reference an existing customer `id`.

4.5 Cardinality Constraints

`min=N` and `max=M` on the table declaration line constrain the number of elements the array may contain.

```
[/@servers host=! min=1 max=50]
!str
host=
!int[1,65535]
port=
```

```
[]
```

This declares: at least 1 server must be present, at most 50 servers are allowed, and all `host` values must be distinct.

Cardinality constraints are O(1) to validate — the Pass 1 parallel scan maintains a counter per array and checks bounds on completion. Unlike uniqueness and referential integrity, cardinality does **not** require O(N) memory. The counter is compatible with the constant-memory certification profile; implementations MAY validate cardinality in Pass 1 without promoting to Level 2.

Constraint	Memory	Pass
`min=N` / `max=M`	O(1) — single counter	Pass 1 (parallel scan)
`field=!` (unique)	O(N) — hash set of values	Pass 2 (post-scan)
`field=/@path` (foreign key)	O(M) — hash set of referenced values	Pass 2 (post-scan)

4.6 Compiled Form

Table declarations compile to **collection constraint lines** in the `.csaiv` — a new first-class line kind in the compiled schema. Collection constraint lines immediately precede the element field definitions for the array.

Authored `.saiv`

```
[/@servers id=!|host=!,port=! min=1 max=50]
!str
id=
!str
host=
!int[1,65535]
port=
!int[1,3600]
timeout?=
[]
```

Compiled `.csaiv`

```
/@servers [unique::id]|[unique::host,port] [min=1] [max=50]
!str'/@servers/::id=
!str'/@servers/::host=
!int /^-?[0-9]+$/ ..num [1,65535]'/@servers/::port=
!int /^-?[0-9]+$/ ..num [1,3600]'/@servers/::timeout?=
```

The collection constraint line `/@servers [unique::id]|[unique::host,port] [min=1] [max=50]` is recognized exactly as §1.3.1 classifies it: a line whose first character is `/` and that contains no `'` delimiter — an array path followed by bracketed constraint clauses (in compiled `.csaiv` form the leading `/` is always present). Pass 2 recognizes this line kind

and registers the constraints for post-scan checking. The element field definitions that follow use the single-line ' format.

Foreign Key Compiled Form

```
// authored .saiv:
[/@employees department=/@departments/*::name]
...

// compiled .csaiv:
/@employees [ref::department=/@departments/*::name]
...
```

4.7 Validation

Table constraint validation uses a two-pass approach (the build/runtime placement of the pipeline stages is in Appendix B.1):

Pass 1 — parallel scan (constant memory, Levels 0–1). The existing Validator parallel scan validates all field-level constraints: types, namepaths, ranges, patterns, enumerations, and required/optional. During Pass 1, the Validator also maintains per-array element counters for cardinality checking (`min`/`max`). Cardinality errors are reported at the end of Pass 1 when the array boundary is detected.

Pass 2 — table constraint check (O(N) memory, Level 2). After Pass 1 completes, Pass 2 executes for each array that has a collection constraint line in the `.csaiv`:

- **Unique constraints:** For each `[unique::field]` or `[unique::f1,f2]`, build a hash set of values seen across all elements. Report duplicate values as constraint violations.
- **Foreign key references:** For each `[ref::field=/@path]`, collect the set of values in the referenced array's field (using a hash set), then verify that every value in the constrained field appears in that set. Report missing references as violations.

Pass 2 runs entirely in the Application layer — it is not part of the Lexer or the Validator's parallel scan. A Level 1 runtime terminates after Pass 1 and never executes Pass 2.

Pseudocode

```
// Pass 2 — table constraint check

for each collection_constraint in csaiv:
    if constraint is [unique::fields]:
        seen = new hash_set();
        for each element in array:
            key = element[fields];  // tuple for compound unique
            if key in seen:
                error("uniqueness violation", array, fields, key);
```

```
            seen.add(key);

    if constraint is [ref::field=/@path]:
        ref_values = collect_field_values(/@path);  // hash set
        for each element in array:
            if element[field] not in ref_values:
                error("referential integrity violation", array, field,
    element[field]);
```

Reconstructing elements. `.daiv` is a flat line stream; `for each element in array` means the Validator groups consecutive lines sharing the `/@arr/<i>::` prefix into one element as it scans — the same index-run boundary the Pass 1 array loop uses. An empty array (zero element lines) is valid and contributes nothing to Pass 2: arrays are **exempt** from the Pass 1 "every schema field appears once" invariant, and a minimum element count is enforced only by an explicit `[min=N]` clause (`CardinalityViolationError`).

Omitted fields. Pass 2 operates on the materialized `.daiv`: an element that omits an optional constrained field participates with its **materialized value** — the resolved default, or the empty payload of a materialized `!null` line (§2.6.13, §2.6.17). Two elements that both materialize the same default (or both materialize `!null`) therefore collide under a unique constraint, and an omitted foreign-key field references its materialized value, which must appear in the target field like any other.

Compound-key encoding. For a compound `[unique::f1,f2]`, `element[fields]` MUST be serialized so that distinct field-value tuples never collide — length-prefix each value (or use a delimiter that cannot occur in a value). Plain concatenation is non-conforming: values are verbatim byte sequences, so `(f1="a", f2="bc")` and `(f1="ab", f2="c")` would otherwise hash to the same key.

4.8 Why This Is Level 2

Table definitions are **Level 2** because they break the constant-memory guarantee of Levels 0–1:

- **UNIQUE validation** requires a hash set of seen values — O(N) where N is the number of array elements.
- **FOREIGN KEY validation** requires collecting referenced values — O(M) where M is the size of the referenced array.
- **Cardinality (`min`/`max`)** is O(1) and MAY be validated in Pass 1 without requiring Level 2.

The O(N) memory requirement for uniqueness and referential integrity cannot be eliminated without a fundamentally different algorithm (e.g. requiring sorted input, which the format does not guarantee). For safety-critical environments that require constant-memory certification at Levels 0–1, Level 2 constraints are excluded from the certified runtime.

The Two-Pass Split Preserves the Level 0–1 Certification Boundary

Pass	Memory	Levels	Certification
Pass 1 (Validator parallel scan)	O(1) constant	0–1	MISRA-C certifiable
Cardinality counter in Pass 1	O(1)	2 (MAY be Pass 1)	MISRA-C certifiable
Pass 2 (table constraint check)	O(N) bounded	2	Not constant-memory certified

A deployment that uses table definitions (Level 2) runs both passes. A deployment that requires constant-memory certification uses Levels 0–1 only — table definitions are not declared in the schema, Pass 2 never runs, and the full constant-memory certification profile applies.

The key architectural insight: **the Level 1 constant-memory parallel scan is never compromised.** Level 2 adds a post-scan pass that runs after the certified scan completes. The certified scan output is still produced by a constant-memory DFA + parallel scan. Level 2 constraints are checked against that output in a separate pass that explicitly acknowledges O(N) memory usage.

4.9 Architectural Impact

Component	Impact
Lexer	Recognizes constraint annotations on `[/@name …]` schema block opening lines: `field=!`, `field1=!,field2=!`, `|`, `field=/@path`, `min=N`, `max=M`. All regular-grammar extensions — no change to DFA structure.
Compiler	Compiles `[/@name constraints…][]` schema blocks to collection constraint lines + element field definitions in `.csaiv`. Resolves constraint field references.
Validator (Pass 1)	Recognizes collection constraint lines in `.csaiv` and registers them for Pass 2. Validates cardinality constraints (O(1) counters). Otherwise unchanged from Level 1.
Pass 2 (new — Level 2 only)	Post-scan table constraint check. Builds hash sets for uniqueness and referential integrity. Runs after Pass 1 in the Application layer. Not part of the certified constant-memory runtime.
Schema compiler	Parses `[/@name constraints…][]` in `.saiv` source, emits collection constraint lines in `.csaiv`. Collection constraint lines use bracket-delimited clauses that the schema compiler generates from the authored constraint annotation syntax.

Component	Impact
Compiled schema (.csaiv)	Gains collection constraint lines (`/@name [unique::field] [min=N] [max=M]`). These are a new line kind — first character `/`, no `'` delimiter (§1.3.1): an array path followed by bracket clauses. Element field definitions that follow use the single-line `constraint'/@name/::field=` format.
Canonical form (.daiv)	Unchanged — data files are unaffected by table definitions.
Query language (qaiv)	The `/@path` syntax used in `[ref::field=/@path]` reuses existing qaiv path syntax unchanged. No new query operators are required.
Certification	Level 0–1 certification profiles are entirely unaffected. Pass 1 (the certified parallel scan) gains only a cardinality counter. Pass 2 is an Application-layer addition that is explicitly O(N) and not part of the certified runtime.

4.10 SQLite Comparison

Table definitions (Level 2) close the DDL gap between kaiv and SQLite's `CREATE TABLE` statement. SQLite is the most common embedded database, used as a file format, a data interchange format, and a configuration store. The comparison is precise because SQLite blurs the line between storage and schema — it is schema-driven, file-based, and serverless, making it the closest peer to kaiv in the storage-and-schema space.

SQLite DDL Feature	kaiv Equivalent	Assessment
`CREATE TABLE name …()`	`[/@name constraints…][]` in `.saiv`	Equivalent
`NOT NULL`	`=` required field (the default operator)	Equivalent
`DEFAULT value`	Schema defaults	Equivalent
`CHECK (range)`	Range constraint `[min,max]`	Equivalent
`CHECK (pattern)`	Pattern constraint `/regex/`	kaiv stronger (first-class regex)
`UNIQUE (field)`	`[/@name field=!]`	Equivalent
`UNIQUE (f1, f2)`	`[/@name f1=!,f2=!]`	Equivalent
Primary key (implicit unique + required)	`field=` inside element body + `field=!` on declaration	Equivalent minus auto-increment index
`FOREIGN KEY (f) REFERENCES t(c)`	`[/@name f=/@t/*::c]`	Equivalent
Cardinality (min/max rows)	`[/@name min=N max=M]`	kaiv stronger (SQLite has no row-count constraint)
`INTEGER`, `TEXT`, `REAL` (type affinity)	`!int`, `!str`, `!float` (explicit types)	kaiv stronger (explicit, pattern-validated)
Enum values via `CHECK ( IN …)`	`{A,B,C}` enumeration constraint	Equivalent

SQLite DDL Feature	kaiv Equivalent	Assessment
`CREATE INDEX`	[ ]	SQLite stronger (kaiv is not a database)
SQL joins	[ ]	SQLite stronger (DML, not DDL)
Aggregation (`GROUP BY`, etc.)	[ ]	SQLite stronger (DML, not DDL)
Triggers	[ ]	SQLite stronger (DML, not DDL)
Transactions	[ ]	SQLite stronger (DML, not DDL)
Partial reads (B-tree)	[ ]	SQLite stronger (storage engine)
Concurrent writes (WAL)	[ ]	SQLite stronger (storage engine)

With table definitions (Level 2), kaiv's DDL **covers SQLite's schema definition DDL completely and exceeds it** in several areas. kaiv has stronger constraint expressiveness than SQLite (which uses type affinity — an `INTEGER` column can store text without error) and equivalent structural integrity constraints. The remaining SQLite features (indexes, joins, aggregation, triggers, transactions, WAL) are database engine features — DML and storage concerns, not DDL concerns. See `DDL_COMPARISONS.md` §7 (in the spec repository) for the full comparison.

5 Level 3: Collation

Level 3 introduces **locale-aware string ordering** via `..lex[locale]` — a span ordering that takes a BCP 47 language tag and produces locale-correct lexicographic comparisons. Collation is a property of named types, never of bare `str`, and is the only feature in kaiv where two conforming validators may produce different comparison results (the spec pins a reference CLDR version **and** collation strength for conformance — §5.3).

5.1 The Problem: `..lex` Is Byte Order

`..lex` means byte-by-byte comparison (`memcmp`). This is correct and efficient for ASCII identifiers and English strings — field names, hostnames, port numbers, URIs, and version strings all sort correctly under byte order. But byte order breaks for locale-sensitive data:

- **"Étude" > "zebra"** — É (0xC3 0x89 in UTF-8) sorts *after* every ASCII letter under `memcmp` (0xC3 > 0x7A), so "Étude" lands beyond "zebra" at the end of the list, while French collation places É with `E` — near the front, before "apple".
- **"café" vs "cafe"** — locale-dependent: in some locales, é sorts equivalently to `e`, in others strictly after it.
- **"straße" vs "strasse"** — in German, ß and `ss` are collation-equivalent; byte order treats them as entirely different strings.

For domain-specific named types — product names, city names, personal names, geographic identifiers — `..lex` (byte order) produces incorrect ordering and range constraint evaluation.

5.2 Syntax: `..lex[locale]`

```
..lex[locale]
```

Where `locale` is a **BCP 47 / IETF language tag** — the same identifier system used by ICU, CLDR, HTML `lang=`, and HTTP `Accept-Language`.

Example	Locale
`..lex[fr-CA]`	French Canadian
`..lex[de-DE]`	German (Germany)
`..lex[zh-Hans]`	Simplified Chinese
`..lex[en-US]`	English (United States)
`..lex[ja]`	Japanese

The locale tag is parsed as a bracketed string immediately after `..lex`. The brackets `…[]` are part of the span token — the Lexer recognizes `..lex[tag]` as a single span ordering token. This is a regular-grammar extension: the DFA gains states to consume the bracketed tag after `..lex`, but the overall token classification structure is unchanged.

`..lex` (bare) remains unchanged — byte-by-byte comparison, available at all Levels including Level 0.

5.3 Reference Collation: CLDR Version and Strength

Two conforming Level 3 validators agree only if they resolve `..lex[locale]` against the same collation data **and** the same comparison options. This specification pins both, and Level 3 conformance is defined against them:

— **Reference data.** Collation is defined against the **CLDR 48** root collation and locale tailorings (the Unicode Collation Algorithm, UTS #10). An implementation SHOULD use CLDR 48; one built on a different version is still usable but MUST report the CLDR version it used in its conformance metadata, and its results are conformant only where they agree with CLDR 48. A future revision of this spec MAY advance the pinned version — it is a normative parameter of the Level, not of the implementation.

— **Strength and options.** Comparisons run at **tertiary strength** with **non-ignorable** variable handling and no case-level, case-first, or numeric reordering — the CLDR root defaults — unless the locale tag carries a BCP 47 `-u-` collation extension that overrides them. Recognized overrides: `-u-ks-level1|level2|level3|identic` (strength), `-u-ka-shifted` (variable = shifted), `-u-kc-true` (case level), `-u-co-<name>` (a named collation such as `phonebk` or `pinyin`). Example: `..lex[de-DE-u-co-phonebk-ks-level1]` selects the German phonebook collation at primary strength. Options the tag does not carry take the pinned defaults, so a given tag always yields one ordering under CLDR 48.

Because collation governs equality as well as order, the pinned version and strength apply to `enum` membership and `[lo,hi]` range evaluation for `..lex[locale]` fields exactly as they apply to sorting.

Partial implementations (the honest-partial rule). A Level 3 validator MUST, for each `..lex[locale]` constraint it encounters, either evaluate it exactly as pinned above or reject that constraint with `CollationUnsupportedError`. Producing an ordering or equality result that differs from the pinned semantics is non-conformant. Conformant Level 3 validators may therefore differ in which constraints they *refuse* — never in how they *order*. A lightweight implementation may support the plain locale tailorings and reject the `-u-` overrides, and remains fully Level 3 conformant in doing so.

Unknown and ill-formed input. Nothing about a locale tag degrades silently:

- A tag that fails BCP 47 well-formedness is a constraint-syntax error: the schema compiler MUST reject it with `INVALID_CONSTRAINT_ERROR` — a static, backend-independent check.
- A well-formed tag whose primary language subtag has no CLDR 48 tailoring MUST raise `CollationUnsupportedError` — root fallback is never silent. An author who wants the root collation says so explicitly: `..lex[und]` is legal and selects the CLDR root.
- An unrecognized `-u-` extension key MUST raise `CollationUnsupportedError` — an explicit ordering instruction is never dropped.

5.4 In Type Definitions (`.taiv`)

Collation is a property of **named types**, not of `str`. `str` stays `..lex` (bare, byte order). Domain-specific types carry the collation they need:

```
.!taiv acme/catalog

// French Canadian product name with locale-aware ordering
/^.{1,200}$/ ..lex[fr-CA]
&product_name=

// German city name
/^.{1,100}$/ ..lex[de-DE]
&city=

// Japanese personal name
/^.{1,100}$/ ..lex[ja]
&person_name=
```

Key principle: the collation declaration sits next to the pattern constraint on the type definition line. Any field annotated `&product_name` automatically uses French Canadian collation for range constraints and query comparisons.

5.5 In Compiled Schema (`.csaiv`)

The collation tag appears as part of the constraint on the element field definition line in the compiled schema:

```
// .csaiv output for &product_name field:
!acme/catalog/product_name /^.{1,200}$/ ..lex[fr-CA]'::product_name=
```

The runtime reads `..lex[fr-CA]` (from the part before `'`) and selects the corresponding collation function. A range constraint like `[Arbre,Zèbre]` on a `..lex[fr-CA]` field is evaluated using French Canadian collation rules, not byte order.

5.6 Certification Impact

`..lex[locale]` is Level 3 because it introduces three properties absent from Levels 0–2:

- **External dependency**: ICU/CLDR is ~25M lines of C/C++ with locale data files measured in megabytes. No Level 0–2 feature requires any external library; the entire Levels 0–2 runtime is self-contained.
- **Non-deterministic across versions**: Collation rules change between CLDR versions. A field sorted one way under CLDR 46 may sort differently under CLDR 48. This spec pins **CLDR 48 and tertiary strength** as the conformance reference (§5.3); an implementation built on a different CLDR version MUST report it and is conformant only where it agrees with the pinned version.
- **Platform variance**: Two conforming Level 3 validators may produce different comparison results if they use different CLDR versions. This never happens at Levels 0–2, where the same input always produces the same output on every platform.

The boundary. Levels 0–2 are **platform-independent** (same input → same output everywhere). Level 3 introduces **platform dependency** — the only Level where two conforming validators may disagree.

A Level 0–2 runtime encountering `..lex[locale]` MUST either:

- **Reject**: emit an error "collation not supported at this level" (safe for ASIL-D — strict conformance)
- **Fall back to `..lex`**: validate with byte order and emit a warning (pragmatic — useful for development and migration)

5.7 Query Impact

qaiv predicate comparisons inherit span ordering from the field's type. If a field has `..lex[fr-CA]`, then:

```
/@products[name>café]/*::price
```

The query engine uses French Canadian collation to evaluate `name > café`. The comparison function is selected from the field's span ordering, resolved through the compiled schema.

A Level 0–2 qaiv engine encountering a collation predicate follows the same reject/fallback rule as the Validator.

5.8 Architectural Impact

Component	Impact
Lexer	Recognizes `..lex[tag]` as a span ordering token. The `[tag]` is parsed as a bracketed string after `..lex`. Regular-grammar extension — no DFA structural change.
Compiler	Passes collation tag through to canonical output unchanged.
Validator	Selects comparison function based on span + collation tag. `..lex` → `memcmp`, `..lex[locale]` → collation library.
Schema compiler	Preserves collation tag in `.csaiv` output.
Compiled schema (.csaiv)	Gains `..lex[tag]` syntax on span declarations.
Type libraries (.taiv)	Can declare collation on named types: `..lex[fr-CA]`.
Query language (qaiv)	Comparison operators use the field's span ordering including collation.
Certification	`..lex` (bare) unchanged — available at all Levels. `..lex[locale]` requires collation library — Level 3 only.

6 Level 4: Corpus-Dependent Features (Reserved)

Level 4 collects operations that read or query a *corpus* of `.daiv` files rather than a single document: path identity, the metaschema (authored as `.msaiv`), schema composition (`.!compose`), and cross-schema foreign keys (`.!ref` / `$alias.field`). Corpus-dependent features break the self-containment property Levels 0–3 share — validating or compiling a `.daiv` requires only that `.daiv` plus its `.csaiv` — and they run only in `kaiv db`-class tooling, never in the certified runtime.

Level 4 is defined, numbered, and **reserved** here — and **specified separately**: the corpus specification is an experimental document versioned on its own cadence (the corpus spec draft and the metaschema design notes in the specification repository). Nothing in the corpus specification is part of this document's conformance surface, and this document's stability promise does not extend to it. Levels 3 and 4 remain independent extensions of Level 2 — neither subsumes the other.

Reserved surface in this document. Two reservations keep the corpus specification a conforming *superset* of this document rather than a fork:

- **`.msaiv` is a reserved format kind.** The extension remains registered (§12.1) and its format declaration remains recognized: a Level 0–3 processor handed a `.msaiv` document MUST report it as a recognized-but-unsupported format kind — never as an invalid directive or unknown file type.
- **Five reserved declaration names.** `.!ref`, `.!compose`, `.!bind`, `.!unique`, and `.!fk` remain in the declaration recognizer as *reserved names*: recognized, never reassignable to other meanings, and permitted in no Level 0–3 format kind — their sole home is

the `.msaiv` document of the corpus specification. A Level 4 tool accepting them there is a conforming superset extension; their occurrence in a Level 0–3 document is a declaration-placement error exactly as for any other declaration outside its permitted format kinds.

7 Compiled Schema (`.csaiv`)

The compiled schema (`.csaiv`) is the validation contract — the artifact the Validator reads in lockstep with `.daiv` to produce pass/fail. It is `caiv` where the left side of `=` is `constraint 'namepath` and the operator is `?=` for optional fields or `=` for required fields. Required is the default.

The optional marker is **build-time information**: it tells the Denormalizer which fields it may materialize when absent from the authored data (§2.6.13). The Validator does not branch on it — materialization guarantees every declared field a `.daiv` line, so the parallel scan is a strict lockstep walk. The one exception is **collection element lines** (elided-index namepaths like `/@ports::=`): an empty collection contributes no data lines, so the scan may advance past an unmatched element line; element counts are enforced by the Pass-1 cardinality check, not by presence.

The compiled grammar is a floor. Compiled schemas are published to registries whose entries are immutable and permanent; a `.csaiv` fetched today must remain readable by every conformant validator, forever. Version 1.0 validators are therefore the permanent floor for the compiled-schema grammar: an extension of the clause grammar after 1.0 would mint eternal registry artifacts that 1.0 validators cannot read. The clause grammar of this chapter **freezes at 1.0**; any later addition requires a major revision that gives the compiled schema a new format identity rather than silently widening this one.

7.1 Parallel Scan Validation

The Validator reads the data and `.csaiv` files in parallel. Each line is split on `'` to separate the metadata prefix (type annotation plus optional provenance list) from the namepath. The provenance list, if present, is in the metadata prefix before `'` — it is skipped or optionally validated against the `.?` header table, and does not affect the `'`-split rule. For each pair of lines:

Step	Action
Split	Split each line on `'` to extract (type/constraint, namepath+value) from `.daiv` and (constraint, namepath+optional-marker) from `.csaiv`

Step	Action
Type check	*Nominal only where the name IS the semantics; structural everywhere else.* A data line whose annotation **equals the retained head** always applies the head's constraints. Otherwise, for a **non-union head** other than `str` / `text` / `null` / `std/enc/*`, the check is **structural**: the head's constraint group governs the value whatever the data line's own annotation — the pre-lift `!str` authored form and a chain-compatible name (`!int` under `!std/net/port`) both apply — except that `!null`- and `std/enc`-typed lines never satisfy it. **Nominal cases:** a **union** — the annotation MUST match one alternative (the discriminant, §2.6.14); **`!text`** — accepts `!text`, or `!str` under the `|:|` guard (§2.6.4; a `|:|`-carrying value is `DelimiterCollisionError`), and `!text` data satisfies no other head; **`!null`** — only `!null` data; an **explicit `!str` head** (the identity declaration — retained iff authored, §7.3) — only `str`-typed lines; **`std/enc/*`** — only their own name (embedded payloads never satisfy plain fields, and vice versa). A unit-carrying head byte-compares the canonical unit both directions (§2.7.2); in a union, on the matched alternative. Violations are `TypeMismatchError`. A head-less field line (identity `str`; its lex-saver, when present, is the vacuous pattern `/^/` — a constraint item, not a head) has no nominal requirement at all.
Namepath match	Compare namepath from schema line against data line
Empty-collection skip	If the schema line is a collection element line (elided-index namepath) and the data line does not match it, the collection is empty — advance the schema pointer without consuming the data line. Element counts are checked by Pass-1 cardinality, not presence
Presence check	If a non-element schema line does not match the data line, a schema-declared field is missing or out of order — emit `RequiredFieldSchemaError`. Materialization (§2.6.17) guarantees every declared field a line, so the scan never branches on the optional marker
Duplicate check	A data line whose namepath equals an *exact schema-declared name* that the scan has already seen is `DuplicateKeySchemaError` (scoping per §9.8); the seen set is schema-sized. Data-introduced names (collection entry keys, relaxed-mode undefined fields) are outside this check
Constraint validation	If schema line has constraints, validate data value against them

Step	Action
Array loop	A schema namepath containing `@` is a collection element line. A *scalar* array (a single element line) stays on that line, consuming the indexed data lines until the namepath prefix changes. A *namespace* array is a *group* of element field lines sharing the `/@name/` prefix: for each element index the Validator walks the group in order, then **resets to the group's first line** when the index increments, repeating until the prefix changes. Element fields arrive in group order — materialization (§2.6.17) gives every element the full field sequence — so a skipped or absent group field, optional or not, is `RequiredFieldSchemaError`, and a field repeated within one element is `DuplicateKeySchemaError`

The two array kinds are handled identically by the loop rule:

- **Scalar array** (`!type'/@ports::n=v` data lines) — the compiled element line carries the lowered element constraint with the elided-index namepath (`!int /^-?[0-9]+$/ ..num'/@ports::=`); the Validator loops consuming indexed data lines until the namepath prefix changes.
- **Namespace array** (`!type'/@servers/n::field=v`) — the schema declares each element field once with `/@servers/` prefix; the Validator loops over the indexed element fields until the namepath prefix changes.

Tagged unions. A union field retains its type names in the compiled line as one type item; each alternative carries its **lowered constraint group** in parentheses — and its unit, glued to its name, when it carries one (`float:km…()`) — so the `.csaiv` stays self-contained (the certified runtime never resolves types). The group concatenates the alternative's lowered definition plus any authored narrowing, **without whitespace** — lowered items are self-delimiting (`/pattern/`, `..span`, `[range]`, `{enum}`, `#[length]`), which keeps the whole union one whitespace-free item token. An alternative with no constraints stays bare:

```
// authored .saiv:                          // compiled .csaiv:
!null|int[1,3600]                           !null(/^$/)|int(/^-?[0-9]+$/..num
   [1,3600])'::timeout?=
timeout?=
```

The Validator checks that the data line's type annotation (the part before `'`) is the head type or one of the alternatives (else `TypeMismatchError`); the **matched alternative's group** — including its own span — then governs the value. The discriminant match includes the unit: a data line matches a unit-carrying alternative by name *and* byte-identical canonical unit, and an alternative without a unit matches only a unit-less annotation. A `!null` alternative therefore enforces its `/^$/` empty-payload constraint (§2.6.17). In authored form, an inline constraint attaches to the alternative it textually follows: in `!null|int[1,3600]` the range narrows `int`, not the union head. A bracketed span argument inside a group belongs only to `..lex` (`..lex[fr-CA]`); after any other span, `[` starts the next (range) item.

7.2 Validator Pseudocode

The entire Validator parallel scan (also used by the integrity check at deployment/runtime) is approximately:

```
// split_on_apostrophe: splits on the first unquoted ' (quoted names use
// "" doubling, never '', so the first bare ' is always the metadata-data
// delimiter; a provenance list ?id[,id...] with optional @timestamp and
// optional #dpid sits before it and is skipped or validated against the
// .? header table). d, s are the data-line and schema-line cursors.
d = 0; s = first(csaiv);
while d < ndata:
    (data_type, data_np) = split_on_apostrophe(data[d]);
    if !matches_any_schema_line(data_np):          // undefined field
        if strict: error("undefined field", data_np);
        d += 1; continue;                          // relaxed: skip, keep s
    if is_exact_schema_name(data_np) && !seen_add(data_np):
        error("duplicate key", data_np);           // schema-declared name
                                                   // repeated (§ Errors);
                                                   // seen is schema-sized
    (sc, sreq) = split_on_apostrophe(s);
    while s && is_element_line(sreq) && !namepath_matches(sreq, data_np):
        // empty collection: contributes no data lines; element counts
        // are enforced by the Pass-1 cardinality check, not here
        s = next(s);
        if s: (sc, sreq) = split_on_apostrophe(s);
    if !s || !namepath_matches(sreq, data_np):
        // materialization guarantees presence and order (§ Null Semantics)
    :
        // a mismatch is always a missing or out-of-order declared field
        error("missing required field", sreq);

    if is_namespace_array(sreq):          // a GROUP of element field lines
        group = element_group(s);         // the /@name/ lines, in group
    order
        apre  = array_prefix(sreq);       // "/@name/"
        while d < ndata && starts_with(namepath_of(data[d]), apre):
            idx = element_index(data[d]);
            g = 0;                        // per-element cursor into the
    group
            while d < ndata && starts_with(namepath_of(data[d]), apre)
                            && element_index(data[d]) == idx:
                k = find_in_group(group, element_field(data[d]));  // whole
     group
                if k == NONE:             // not a group field
                    if strict: error("undefined field", data[d]); // else
    skip
                else if k < g:            // already consumed in this
    element
                    error("duplicate key", data[d]);
                else if k > g:            // group[g] was skipped: lockstep
     order
                    error("missing required field", group[g]);
```

```
                else:
                    check_type_and_constraints(group[k], data[d]);
                    g += 1;
                d += 1;
            if g < len(group):            // element ended short of the
    full
                error("missing required field", group[g]);  // field
    sequence
        s = after(group);
        continue;

     // flat field, scalar-array element, or map entry: one schema line
     if !type_matches(sc, data_type): error("type mismatch", sreq);
     validate_constraints(sc, value_of(data_np));
     d += 1;
     if is_element_line(sreq) && d < ndata
                              && namepath_matches(sreq, namepath_of(data[d])
    ):
        continue;                         // stay on s for the next entry/
    index
     s = next(s);

while s:                                  // remaining schema lines: empty
     if !is_element_line(namepath_of(s)):  //   collections are fine,
    anything
        error("missing required field", namepath_of(s));   // else is
    missing
     s = next(s);
```

Splitting rule. Split on the first bare `'` in the line. Since quoted names in namepaths use `""` (double-quote doubling) as their only escape — not `''` — there is no ambiguity: the first `'` in any canonical data line is always the metadata-data delimiter. Any provenance list (`?sourceID`, `?id@timestamp`, `?id@timestamp#dpid`, `?id#dpid`, or `?id1,id2@timestamp`) in the metadata prefix sits before this `'` and does not interfere with the split — no provenance ID or data point ID contains `'`. Collection constraint lines in `.csaiv` (e.g. `/@servers [unique::field] [min=1] [max=50]`) have no `'` and are recognized by their `@`-prefixed structure with bracket clauses.

Undefined fields and the schema pointer. A data line that matches *no* schema line — an undefined field — MUST NOT advance the schema pointer: relaxed schemas MAY interleave undefined fields at any point *outside a namespace-array run* (§11, strict vs. relaxed), and the defined fields that follow still have to find their schema lines in order. A namespace array's data lines are one atomic, contiguous run — canonical output emits each array as a single block of ascending elements, so no conforming producer wedges a foreign line into one. Inside a run the sub-scan consumes only lines under the array prefix: an *unknown field under the prefix* (a newer producer's per-element addition such as `/@servers/0::region`) is skipped under a relaxed schema and is not an interruption, but any line outside the prefix ends the run — an element left short of its field sequence is `RequiredFieldSchemaError`,

and array lines resuming after the break fail the presence check as out-of-order defined fields. `matches_any_schema_line` is a membership check against the resolved schema (an exact-namepath set plus the collection-line prefix forms) — schema-sized memory, which is already resident; no data-sized allocation. Ordering of *defined* fields remains enforced: a defined field appearing out of schema order still fails with `RequiredFieldSchemaError` via the presence check.

Constant memory — a namespace-array element needs one cursor into its group, and the group itself is schema text, already resident — linear time, no data-proportional allocation. The flat, map, and scalar-array path is a dozen lines of C; a namespace array adds the bounded element-group sub-scan above. Certifiable at any ASIL level.

> **Integrity check.** The Validator's parallel scan logic may optionally be re-run on an existing `.daiv` at deployment or runtime (loading `.daiv` and `.csaiv` and performing the same line-by-line scan). This is not a separate pipeline stage — the build pipeline ends at `.daiv`. The integrity check re-runs the same constant-memory validation to verify that the artifact has not been corrupted or tampered with after production.

7.3 The Schema Compiler

The schema compiler is a **compilation-pipeline canonicalizer for `.saiv` → `.csaiv`**. It reads authored schema text, which is a kaiv document using the same Lexer and the same line grammar as data files. The schema compiler:

— Expands namespace blocks in the schema source, resolving all namepaths to fully-qualified form.

— Resolves `.!types` imports — loading named type library files (`.taiv`) and lowering `&name` annotations to their resolved constraint forms (pattern + span + range/enum). This includes `std/core` types, which are resolved from the implementation's embedded/bundled copy of `std/core.taiv` — `!int` is treated exactly like any other named type reference, resolved to `/^-?[0-9]+$/` + `..num`. The `.csaiv` drops every `&`-reference, lowers each type to its value-constraint form (`/pattern/ ..span [range/enum]`), and **retains the authored head type name** as the field's leading item (**head-type retention**): `!int`, `!std/time/datetime`, `!acme/net/port` — the most derived *authored* name, never its lowered base. The retained head is the schema's statement of what the field *is*, carried in the compiled artifact itself: it is what the schema-aware Denormalizer lifts onto untyped data lines (§2.6.17), what keeps exporters schema-independent, and what preserves `.daiv` self-description. Three heads carry semantics beyond their constraints, exactly as before: **`!text`** — the line-separator interpretation (§2.6.4); a **unit-carrying type**, emitted `!type:unit` so the Validator can byte-compare the unit; and a **union**, emitted as its full type item — each alternative carrying its lowered constraint group in parentheses, and its unit glued to its name when it carries one (`!null(/^$/)|int(/^-?[0-9]+$/..num)`) — so the Validator can discriminate the active variant from the data line's annotation and apply that variant's constraints (§2.6.14). The **identity head `str`** is retained exactly when the author declared it: an authored `!str` annotation is the **identity declaration** and survives as a nominal head (§7.1). A field with no authored type — unannotated, or an anonymous refinement (§2.6.13) — carries no type item; where such a line needs a leading item to lex (an items-free field, or lowered

items that would otherwise begin with `#` or `//`), the schema compiler emits the **vacuous pattern** `/^/` — inside the pinned dialect, matching every string — never a type item. Retention is authorial intent, not an emission artifact, and DFA processing is unaffected: the line lexes by the same `'` split either way, and `/^/` is an ordinary pattern item. `&`-references never survive into the `.csaiv`.

— Resolves schema inheritance (`.!schema` declarations in schema files), merging inherited field definitions.

— Outputs canonical schema text — one single-line `constraint'namepath=` field definition per field, all namepaths fully qualified, each field carrying its retained head type item (elided for `str`) followed by its lowered `/pattern/` `..span [range]` / `{enum}` constraint items, joined with the `'` delimiter. A field line may never begin with `#` — rule 2 would classify it as a comment — so when the lowered items would otherwise lead with a length constraint (a `str`-typed, length-only field), the schema compiler emits a leading item first: the retained `!str` when the author declared it (`!str #[2,8]'::code=`), the vacuous pattern otherwise (`/^/ #[2,8]'::code=`).

The schema compiler uses the same Compiler/Denormalizer/Validator pipeline as for data files, adapted for schema syntax. It is not a DFA that produces a DFA — it is a canonicalizer that produces canonical text. For certification purposes: compile-time tools run on the developer's workstation, not on the safety-critical target. Only runtime components — the Lexer and the integrity check parallel scan — run on target hardware and require certification.

7.4 Table Declarations in the Compiled Schema

Table definitions (Level 2) introduce a new kind of line in the `.csaiv`: the **collection constraint line**. It immediately precedes the element field definitions for the array and declares the collection-level constraints that Pass 2 must check.

Authored `.saiv` Table Definition

```
[/@servers host=!,port=! min=1 max=50]
!str
host=
!int[1,65535]
port=
!int[1,3600]
timeout?=
[]
```

Compiled `.csaiv` Output

```
/@servers [unique::host,port] [min=1] [max=50]
!str'/@servers/::host=
!int /^-?[0-9]+$/ ..num [1,65535]'/@servers/::port=
!int /^-?[0-9]+$/ ..num [1,3600]'/@servers/::timeout?=
```

The `/@servers [unique::host,port] [min=1] [max=50]` line is the **collection constraint line** — a new first-class line kind in `.csaiv`. It carries:

Clause	Syntax	Meaning
`[unique::field]`	Single-field unique constraint	All values of this field across all elements must be distinct
`[unique::f1,f2]`	Compound unique constraint	The *combination* of `f1` and `f2` must be distinct across elements
`[unique::f1]|[unique::f2,f3]`	Multiple independent unique constraints	Two separate uniqueness requirements on the same array
`[ref::field=/@path]`	Foreign key reference	Field values must exist in the referenced array field
`[min=N]`	Minimum element count	Array must have at least N elements
`[max=M]`	Maximum element count	Array must have at most M elements

Authored array schemas below Level 2. The element-level compiled lines do not require the Level 2 collection machinery. A **scalar array** is declared with the vector operator in schema position — `!int` above `/@ports;=` compiles to `!int /^-?[0-9]+$/ .. num'/@ports::=` — mirroring `?=`'s role shift from data to schema. A **namespace array**'s element fields are declared with a constraint-free section block (`[/@servers] ... []`; the table-declaration syntax explicitly allows zero constraints), compiling to the `{items}'/@servers/::field=` element lines. What makes a table *Level 2* is the collection constraints (`unique`/`ref`/`min`/`max`) and their O(N) Pass 2 — element-shape validation alone is single-pass and Level 1. Collections are never themselves required: an empty array or map is valid absent an explicit `[min=N]`.

The collection constraint line has no `'` delimiter — it is `/@name [clauses]` without a namepath. The element field definitions that follow use the single-line `'` format and the array namepath prefix (`!str'/@servers/::host=`) to scope them to the array without an explicit index — the `::` immediately following `/` signals that this is an element-level schema line, not a specific indexed element.

Multiple independent unique constraints compile to adjacent `[unique::]` clauses separated by `|` on the collection constraint line:

```
// authored .saiv:
[/@servers id=!|host=!,port=!]
...

// compiled .csaiv:
/@servers [unique::id]|[unique::host,port]
```

Foreign key reference compiles to a `[ref::]` clause:

```
// authored .saiv:
[/@employees department=/@departments/*::name]
```

```
...

// compiled .csaiv:
/@employees [ref::department=/@departments/*::name]
```

7.5 Maps in the Compiled Schema

A map field (§2.6.18) is a namespace with arbitrary string-named entries that all share one value type. Its compiled form parallels the scalar array: where an array declares its element constraint once against an elided integer index (`!str'/@servers/::host=`), a map declares its **entry** constraint once against an elided string key.

Authored `.saiv`

```
!map<int>
/config/settings=
```

Compiled `.csaiv`

```
!int /^-?[0-9]+$/ ..num'/config/settings::=
```

(A root-level map — `settings=` with a bare-name key — compiles the same way with the map's own name as the single namespace step: `/settings::=`.) The **map-entry line** uses the empty-terminal namepath `mapnamespace::` — the same `canonical-steps "::"` form as a scalar-array element (§10.6), distinguished by the absence of `@`: a `::`-terminated schema line whose steps contain an `@` is a scalar-array element (integer keys), and one whose steps contain **no** `@` is a map entry (string keys). The value type is lowered to its constraint form exactly like a scalar field (`!map<str>` → `!str'/config/settings::=`, the identity item).

Validation scan. The map-entry line is consumed by the same variable-run loop as the scalar-array element (§7.1, *Array loop*): the Validator stays on the entry line while consecutive data lines share the map's namepath prefix (`/config/settings::<key>`), validating each entry's value against the value constraint. Zero entries is valid — a map may be empty (authored `={}` emits no entry lines), so a map field imposes no minimum entry count by itself.

Key constraints (optional). By default a map key is any legal name (§2.3). A schema MAY constrain keys with a **key clause** on a collection-style line preceding the entry line, and MAY bound the entry count with the same `[min=N]` / `[max=M]` clauses used for arrays:

```
/config/settings [key::/^[a-z][a-z0-9_]*$/] [max=100]
!int /^-?[0-9]+$/ ..num'/config/settings::=
```

Each entry's key — the terminal after `::` — MUST match the key pattern; a violation raises a `ConstraintViolationError`.

Authored key constraints — the map schema block

The authored surface for key constraints follows kaiv's **ostensive definition** principle: the schema shows a specimen of the data, in the data's own shape. Exactly as a table schema mirrors the data's `[/@name]` block, a keyed map schema mirrors the data's `(/name)` block — with a `/regex/` standing in key position where a data entry would carry its key, and the entry-count bounds riding the block header as `min=` / `max=` clauses, exactly as table headers carry them:

```
(/config/settings max=100)
!int
/^[a-z][a-z0-9_]*$/=
()
```

Inside the block, the type-annotation line applies to entry *values*; the following key line carries the key grammar in key position — "entries look like this" — and takes no default. The block lowers to exactly the compiled form above: the key clause and cardinality clauses onto the collection-style line, the value type onto the map-entry line. The flat `!map <VALUETYPE>` field form remains the authored surface for *unkeyed* maps; the block form is its keyed superset. A *literal* key line inside a map schema block (a required named entry, e.g. `en=`) is **reserved**: the schema compiler MUST reject it in this revision.

> **Implementation status.** The reference pipeline implements both sides: map data lowering (the entry lines, §2.6.18) and `!map<…>` schema-field lowering with the map validation scan (conformance group `schema/005-map`); the ostensive map schema block and key-clause validation are exercised by conformance group `schema/041-map-keys`.

7.6 Delegated Namespaces in the Compiled Schema

A discriminated schema set (§3.5.3) compiles to a **delegation line** — a collection-style line (first character `/`, no `'`) carrying a single bracketed clause with the member references, fully qualified, in authored order:

```
/parameters [schema::crypto/rsa-params|crypto/ecdsa-params]
```

The delegated namespace contributes no field lines of its own to the parent `.csaiv` — the delegation line is its entire compiled presence. Contrast the encapsulated extension (§3.5.6), which *merges* one schema's compiled lines in place: a merge inlines a single known contract; a delegation defers a one-of-N choice to the document.

Validation. The Validator resolves and preloads every member's `.csaiv` at startup — the set is static, so the working set remains schema-sized. When the scan reaches the delegation line, it reads the document's scoped declaration (`.!schema:/parameters crypto /rsa-params` — the block's discriminant, §3.5.5): a missing declaration, or one naming a schema outside the set, is a `DelegationSchemaError`. The lines under the delegated namespace are then walked in lockstep against the selected member — its constraints, its `strict` modifier, and its `.!provenance` level govern within the namespace — and the parent scan resumes at the first line outside the prefix. The sub-scan is the same constant-memory parallel scan (§7.1); composition adds one delegation level per nesting, and the chain is bounded statically by the schema graph (§3.5.3).

Materialization. The schema-aware Denormalizer applies the same selection at build time: the selected member's absent optional fields are materialized within the namespace, in the member's declared order, so the every-declared-field-appears invariant (§2.6.17) holds inside a delegated namespace exactly as outside it — including the head-type lift onto the member's untyped lines (§7.3).

> **Implementation status.** The delegation clause is specified ahead of the reference implementation so that its syntax lands inside the 1.0 compiled-grammar freeze (§7 — the clause grammar is a floor). Conformance vectors follow with the kaiv-rs implementation (REVIEW.md D-10, in the spec repository).

8 Mappings (`.maiv`)

A **mapping** declares a structural correspondence between two schemas: which target field receives which source field. Because kaiv schemas describe pure data — no methods, no computed properties — a mapping is exhaustive and purely structural: a name-to-name rewrite table with no value transformations, no predicates, and no conditionals. Mappings are the edges of the schema graph: published to `ksaiv.com` alongside schemas, they make independently authored schemas mutually convertible, and their composition is a join on namepaths rather than a program-synthesis problem.

A mapping file (`.maiv`) is `caiv`: the same six-rule line classifier, the same declarations mechanism, and — decisively — the same two core constructs that already express "receives from": `=` assignment and the `$` dereference sigil. A `.maiv` line assigns to a *target* namepath (left of `=`, as everywhere in kaiv) a value that is either a `$`-reference into the *source* schema's namespace or a literal constant. No new operators are required.

8.1 Header Declarations

```
.!maiv
.!source acme/server-config
.!target hub/server-endpoint
```

— `.!maiv [VERSION]` is the format declaration. A mapping carries **no identity token of its own**: its identity *is* the `(source, target)` endpoint pair, and its registry address derives from it (§8.2). The version is optional with the bare form canonical, exactly as for the data kinds (§2.1.2).

— `.!source ID-OR-URL` and `.!target ID-OR-URL` name the source and target schemas — registry paths or absolute URLs, resolved exactly like `.!schema` references, and always written fully qualified. Both are required, each exactly once.

— `.!via EDGE` (optional, repeatable) records the provenance trail of a *composed* mapping: one declaration per hop, in application order (§8.6). Each hop is an edge identity — `{source}/{target}` with the same-namespace short form — version-qualified when the hop was retrieved from a registry (§8.2); a hop composed from a local file has no edition and is recorded unversioned.

— `.!drop SOURCE-NAMEPATH` (optional, repeatable) documents a deliberately unmapped source field. Unmapped source fields are skipped either way (§8.4); the declaration

makes the omission machine-readable so publish-time tooling can distinguish intent from oversight.

8.2 Registry Addressing

Mappings publish to `ksaiv.com` — they are edges between schemas, and they live beside them. A published mapping has no chosen name; its address **derives from its endpoints and its edition**:

```
https://s.kaiv.io/{source}/mapto/{target}/{version}.maiv
```

where `{source}` and `{target}` are the fully qualified `{namespace}/{name}` schema references (schema references on the registries are exactly two segments), `mapto` is the **direction marker**, and `{version}` is the mapping's edition: `v1`, `v2`, … The marker makes the address self-evident — it reads as a sentence — and names the publisher: a `mapto` address is the *source* owner's assertion. The rules:

- **The version segment is mandatory** and always last — the first publish is `v1`. Each address is write-once like every registry artifact; a refined mapping is a new edition beside the old, never a replacement. The current edition is discovered by probing upward from `v1` (a registry MAY offer a `latest` convenience alias, which is mutable and therefore explicitly outside the eternalink promise).
- **Same-namespace short form.** When the endpoint after the marker shares the leading namespace, its namespace is omitted — `acme/config-v1/mapto/config-v2/v1.maiv` — and the short form is canonical and mandatory: the redundant spelling is rejected at publish time.
- **Ownership follows the address.** The address begins with the source's namespace, so the ordinary cella ownership rule means *the source owner publishes the edge*. For the inbound case — a third party attaching an edge *from* a schema it does not own (typically a hub) *to* its own — the `mapfrom` marker reverses the spelling and puts the edge under the target's namespace:

  ```
  https://s.kaiv.io/{target}/mapfrom/{source}/{version}.maiv
  ```

  A `mapfrom` address is the *target* owner's assertion — the marker itself records who vouches for the edge. The segment names `mapto` and `mapfrom` are reserved on the registries (no schema or namespace may claim them), so the two address families parse unambiguously.
- **The `mapto` address is canonical.** A consumer resolving the edge `A→B` tries `{A}/mapto/{B…}/` first and falls back to `{B}/mapfrom/{A…}/`. Hub curators can therefore bless a canonical out-edge that supersedes a third party's `mapfrom` edge without deleting anything — write-once holds everywhere.
- The `.maiv` file content is identical at either address family — the marker changes who may publish it, never what it says. `.!via` hops use the canonical `mapto` spelling regardless of which address hosts the hop.

8.3 Mapping Lines

Every mapping line is a rule-5 content line: a target namepath, `=`, and a right side.

```
# Simple field rename: target host receives source hostname
::host=$::hostname

# Namepath restructure
/network::listen_port=$/server::port

# Array field mapping — the /* wildcard maps every element
/@nodes/*::host=$/@servers/*::hostname

# Constant override: source values outside the target constraint
# fall back to the constant (here: TRACE/FATAL -> DEBUG)
::level=$::level|DEBUG

# Null fallback: requires the target field to be nullable
::region=$::legacy_region|!null

# Constant: target field with no source counterpart
::api_version=v2
```

The right-side forms:

Right side	Meaning
`$namepath`	The target field receives the source field's value
`$namepath|constant`	As above; if the source value fails the target field's constraint, the constant is emitted instead
`$namepath|!null`	As above with a null fallback; the target field MUST be declared nullable (`!null|T`)
`literal`	Constant: the target field always receives this literal. Also serves as the default for a target field with no source counterpart

Rules:

- **Target-left.** The target namepath is always on the left of `=` — the same direction as every kaiv assignment. Namepaths use the canonical shapes (§10): `::field` for root fields, `/ns::field` for namespaced fields, with the `/*` wildcard standing for every element index directly after an `@`-step (the same position and meaning as in `fk-path`).
- **$ discriminates source references from constants.** A right side beginning with `$` is a source reference; `$$` begins a literal constant that starts with `$` (the standard doubling, §2.5.5).
- **One optional `|`-override per line, no logic.** The override constant is validated against the target field's constraint at publish time; there are no conditionals and no value transformations. An override constant cannot contain a literal `|` — the same delimiter-collision rule (`DelimiterCollisionError`) as `:=` pair values.
- **No metadata annotations.** Rule 6 never applies in `.maiv` — target types are resolved from the target schema's `.csaiv`, never annotated in the mapping. `#` comments and `//` doc comments are allowed as in other definition-bearing authored files.

8.4 Execution Model

The **mapper** is a single-pass line rewriter — build-time tooling, outside the certified runtime:

```
Input:   source .daiv + source .csaiv + target .csaiv + .maiv
Output:  target .daiv

For each line in source .daiv:
    1. Split on ' -> metadata prefix + namepath=value
    2. Look up the source namepath in the mapping table
    3. Mapped:   rewrite the namepath, resolve the target type from
                 the target .csaiv, and emit; if the value fails the
                 target constraint and the line carries an override,
                 emit the override constant (or !null) instead
    4. Unmapped: skip (dropped or out of scope - no error)

Then emit every constant line (targets with literal right sides)
not produced above, and assemble the output in target schema order.
```

The output is a canonical `.daiv` against the target schema: fully materialized, in target schema order (§2.6.17), ready for the target's Validator with no further resolution. When the target reference is a registry path, the output carries a `.!schema` declaration naming it, so downstream validation is self-describing; source-side declarations do not carry over, except `.?` provenance source declarations, which survive alongside the per-value provenance they anchor. The mapper reads the source in one pass with the mapping table and both compiled schemas resident, collects the produced target lines, and assembles them in target schema order — a mapping may permute fields arbitrarily, so the working set is the schemas *plus the produced output* (output-sized, not merely schema-sized). It performs no corpus I/O.

8.5 Publish-Time Validation

A `.maiv` is validated against both schemas when published (and SHOULD be validated by tooling before use):

- Every target namepath MUST exist in the target schema, and every source namepath in the source schema (`SchemaResolutionError` if either schema cannot be retrieved; `UndefinedReferenceError` for a namepath that resolves to no declared field).
- Every override constant MUST satisfy the target field's constraint — a violation is a `ConstraintViolationError` — and a `|!null` fallback MUST target a nullable field.
- **Completeness:** every *required* target field MUST be produced — by a mapping line, a constant line, or the target schema's own applicable default (§2.6.13). A required target field with none of the three is an `IncompleteMappingError`. Because mappings are purely structural, this check is static: no data is needed to run it.

8.6 Composition

Mappings compose by joining on namepaths: given `B←A` and `C←B`, the composed `C←A` replaces each source namepath of the second mapping with the corresponding source

namepath of the first — string substitution, no synthesis. This is exactly the property that pure structural mapping buys. Overrides are the one lossy spot: a composed line carries a single override, while the two-hop chain distinguishes a value that fails the *intermediate* schema's constraint from one that fails the final one. A composed mapping is therefore a faithful **endpoint** contract — its override fires on the final constraint only — not a replay of the chain's intermediate checks; the `.!via` trail preserves the hops for a consumer that needs the exact chained semantics.

A composed, published mapping records its derivation with `.!via` declarations, one per hop in application order:

```
.!maiv
.!source acme/server-config
.!target helm/values-v1
.!via acme/server-config/mapto/k8s/deployment/v1
.!via k8s/deployment/mapto/helm/values-v1/v2
```

Each named hop is itself a published `.maiv`, so the trail is auditable; the hop endpoints are recoverable from the referenced mappings' own `.!source`/`.!target` headers.

8.7 Auto-Derived Mappings

Schema extension produces mapping edges without a hand-written `.maiv`: when a schema extends a hub schema (§3.5.6), the extending fields are structurally identical to the hub's by declaration, so the registry derives the edge at publish time (`/ns:: field=$::field` for each hub field, in the encapsulated case). Hand-written mappings and auto-derived edges participate in the same graph; only hand-written ones carry overrides.

9 Parsing Requirements

This section specifies what a conformant Lexer (the regular-grammar token producer at the front of the pipeline) MUST and MAY do. The Lexer's responsibility ends with token emission; the Compiler/Denormalizer/Validator stages described under §7 take over from there (their build/runtime placement: Appendix B.1).

9.1 Line Numbers

Lexers MUST track line numbers and include them with every emitted token.

- Line numbers MUST be 1-based.
- Lexers SHOULD include line numbers in error reports.
- If tokens are emitted out of order (e.g. parallel parsing), every token type that may exist (KV, comment, blank, declaration, type annotation, provenance, shebang) MUST be emitted, so the consumer can reconstruct order from the line numbers.

9.2 UTF-8 Processing

Lexers MUST process input as UTF-8. Validation may occur:

— before parsing (whole text), or
— during parsing (line by line), or
— while streaming.

9.2.1 BOM Handling

The UTF-8 Byte Order Mark is not supported. If a kaiv text begins with the bytes `EF BB BF`, the Lexer MUST raise a `BOM_ERROR`. BOM detection MUST occur before any other parsing action (including UTF-8 validation), and the BOM bytes MUST NOT be interpreted as part of the kaiv text.

9.2.2 Forbidden Characters

A kaiv text MUST NOT contain:

— a CR character (U+000D) that is not part of a CRLF sequence;
— a NUL character (U+0000).

A Lexer encountering either MUST raise an `INVALID_CHARACTER_ERROR`. Range U+D800 – U+DFFF (UTF-16 surrogates) is implicitly excluded by valid UTF-8.

9.3 EOL

`EOL` is `LF` (U+000A) or `CRLF` (U+000D U+000A). Intermixing of LF and CRLF within the same kaiv text is tolerated. Canonical emitters write LF only (§12.3).

Every line in a kaiv text MUST be terminated with an EOL, *including the final line*. Without a final EOL, streaming Lexers cannot reliably distinguish incomplete transmissions from valid end-of-input. A Lexer encountering a non-empty final line without an EOL terminator MUST raise a `MISSING_FINAL_EOL_ERROR`.

An empty kaiv text (0 bytes) is valid and yields no tokens.

9.4 Whitespace Handling

In this specification, *whitespace* refers exclusively to:

— U+0020 SPACE
— U+0009 CHARACTER TABULATION

Lexers MUST:

— remove leading whitespace on every line (any whitespace between the start of the line and the first non-whitespace character);
— remove whitespace between keys and assignment operators;
— preserve all whitespace after the assignment operator verbatim — anything after `=` is part of the value.

Whitespace within a key is not optional indentation but a key-character violation; it MUST raise an `INVALID_KEY_ERROR`. The `=` character is the only assignment operator at the lexical level — neither space nor colon is recognized.

9.5 Value Preservation

Lexers MUST treat all kaiv-text values as strings, and MUST preserve them verbatim. Lexers MUST NOT interpret any characters as having special meaning or initiating escape sequences. Specifically:

- quotes MUST be preserved as part of the value;
- backslashes MUST be preserved as literal backslash characters (`\n` is a backslash followed by `n`, not a newline);
- `$` MUST be preserved verbatim — its semantic role for variable / field references is a Compiler concern, not a Lexer concern (and absent altogether in a verbatim document, §2.5.6);
- `#` and `//` inside values are literal characters; comment recognition only applies at the start of a line (after optional leading whitespace).

Values MUST NOT contain EOL characters. All other characters, including any leading or trailing whitespace, MUST be preserved verbatim.

9.5.1 Empty Values

When an assignment operator is immediately followed by EOL, the Lexer MUST emit the value as an empty string. The line `KEY=` is a valid KV token with key `KEY` and empty value.

9.6 Empty Documents

An empty text (0 bytes) is a valid kaiv text and yields no tokens.

9.7 Parsing Models

Lexers implement one of two models:

1. **Eager parsing** — validate the entire text before emitting any tokens.
2. **Streaming parsing** — emit tokens as lines are processed.

Lexers MUST document which model they implement, and MUST document their behavior when an error occurs:

- *Eager parsers*: whether any tokens are emitted in case of error.
- *Streaming parsers*: whether tokens before the first error are emitted, and whether parsing continues after errors.

Lines causing errors SHOULD NOT emit tokens.

9.8 Duplicate Keys

The Lexer MUST NOT detect or process duplicate keys — duplicate-key handling is an application-level concern. Multiple data lines with the same fully-qualified namepath produce multiple tokens, and the application chooses the resolution strategy. Common strategies:

Strategy	Behavior
First wins	Use the first occurrence; ignore subsequent values.
Last wins	Use the last occurrence; override earlier values.
Concatenate	Join all occurrences into a single string.
Preserve all	Keep all values as a list or multiset.
Reject	Treat duplicates as an error.

In the canonical pipeline, the Compiler uses *last-write-wins* for namespace-block field overrides (see §2.5) but does not enforce a global rule. The one schema-level exception is `DuplicateKeySchemaError` (§11.2): a repeated *schema-declared* field is a validation error, while data-introduced names — collection entry keys, fields undeclared in a relaxed schema — remain governed by this section.

9.9 Implementation Limits

This specification does not mandate maximum limits for line length, number of tokens in a document, key length, value length, or total document size. Implementations MAY impose reasonable limits based on available resources, but SHOULD document those limits and provide clear error messages when limits are exceeded.

10 Formal Grammar (Levels 0–1)

This section consolidates the line grammar into ABNF (RFC 5234). It is normative for Levels 0 and 1 plus the Level 0/1 subset of `.saiv`, `.taiv`, and `.csaiv` files. Where a prose section and this grammar disagree, the grammar wins; report the discrepancy. Level 2 adds only the table-declaration and collection-constraint productions given at the end; Level 3 adds only the `..lex[locale]` span argument; the Level 4 declaration keywords appear only as reserved productions with opaque interiors (§6).

Two prose-level rules frame everything below:

1. **Line independence.** The grammar is line-oriented and regular. Every line matches exactly one of the six rules (§1.3.1); no production spans an EOL. Block delimiters (`[/@x]`, `[]`, `(/x)`, `()`) are themselves single lines; the pairing of open/close lines is a Compiler concern, not a lexical one.
2. **Leading whitespace** is stripped before classification on every line (§9.4); the productions below describe lines *after* that stripping. Whitespace around the = split is stripped; everything after = is verbatim.

10.1 Common Productions

```
eol           = LF / (CR LF)
ws            = SP / HTAB
any-char      = <any valid UTF-8 character except NUL, CR, LF>
                ; CR is permitted only as part of CRLF (§ Forbidden
   characters)
```

```
value          = *any-char        ; verbatim — no escape sequences (§ Value
   Preservation)

bare-name      = ( ALPHA / "_" ) *( ALPHA / DIGIT / "_" )
quoted-name    = DQUOTE 1*( qn-char / (DQUOTE DQUOTE) ) DQUOTE
qn-char        = <any-char except DQUOTE>
name           = bare-name / quoted-name

index          = "0" / ( %x31-39 *DIGIT )
                 ; no leading zeros: one canonical spelling per element

path-seg       = ( ALPHA / DIGIT ) *( ALPHA / DIGIT / "_" / "-" )
                 ; library paths and schema IDs (std/net, acme/server-config
   ,
                 ; hub/log-entry). Note: allows "-", unlike bare-name. A "."
    in
                 ; the first segment is reserved for future DNS-based
   authority
                 ; (§ Type identity vs. type resolution) and currently
   invalid.
lib-seg0       = ALPHA *( ALPHA / DIGIT / "_" / "-" )
                 ; alpha-first: a library path can never be mistaken for a
                 ; version token (§ Format Declaration)
library-path   = lib-seg0 *( "/" path-seg )

prov-ident     = ( ALPHA / DIGIT / "_" ) *( ALPHA / DIGIT / "_" / "-" )
                 ; provenance source IDs and data point IDs (sensor1, req
   -42, UUIDs)
```

10.2 Line Classification

```
document         = [ shebang-line ] *line
                   ; the format declaration is optional in authored
                   ; .kaiv (absent means authored .kaiv version 1);
                   ; canonical .raiv/.daiv always open with
                   ; .!raiv/.!daiv (§ Format Declaration)
shebang-line     = "#!" *any-char eol
                   ; physical first line only (§ Shebang Lines);
                   ; recognized before line classification — never
                   ; a comment

line             = blank-line / comment-line / doc-line / declaration-line
                 / content-line / metadata-line          ; rules -16 in
   order
blank-line       = eol                                    ; rule 1
comment-line     = "#" *any-char eol                      ; rule 2
doc-line         = "//" *any-char eol                     ; rule 3
; declaration-line (rule 4): begins ".!" or ".?" — see § Declarations below
; content-line (rule 5): contains "=" — split on the FIRST "="
; metadata-line (rule 6): no "=", first char one of "!", "?", "&"
```

Rule 5's split on the first = is what makes the grammar regular: the left side is everything before the first = (with surrounding whitespace stripped), the right side is everything after (verbatim). One qualification: a line whose left side begins with " enters the quoted-name sub-state (§2.3) first, and an = *inside* the quoted name is part of the name, not the split point — the split is on the first = outside a quoted name. The sub-DFA already implies this; it is stated here because a naive byte scan for = would get `"a=b"=v` wrong. A second qualification: rule 6 has priority over the split for metadata-leader lines whose entire text parses as a metadata/constraint line (§1.3.1, rule-6 priority) — a pattern or enum item may contain = without making the line a content line. All assignment operators are recognized as suffixes of the left side: a left side ending in `+` is `+=`, ending in `;` is `;=`, ending in `+:` is `+:=`, ending in `:` is `:=` (a key can never otherwise end in a single `:` — colons appear in keys only as the `::` projection mid-path), ending in `?` (in schema files) is `?=`. Block-delimiter lines (`[...]` / `[]` / `(...)` / `()`) contain no = at authoring level except inside table clauses, and are recognized before the rule-5 split by their bracket/paren first character; they are given under §10.7 below.

10.3 Declaration Lines (Rule 4)

```
declaration-line = format-decl / saiv-decl / taiv-decl / faiv-decl
                 / maiv-decl / msaiv-decl / schema-decl / types-decl
                 / units-decl / verbatim-decl
                 / registry-decl / provenance-req-decl
                 / ref-decl / compose-decl / bind-decl / unique-decl / fk-
   decl
                 / source-decl / target-decl / via-decl / drop-decl
                 / source-id-decl

version          = 1*DIGIT [ "." 1*DIGIT [ "." 1*DIGIT ] ]
                   ; omitted components are zero (§ Format Declaration)

format-decl      = ( ".!kaiv" / ".!raiv" / ".!daiv" )
                   [ 1*ws version ] eol
                   ; the keyword mirrors the file kind; bare form
                   ; means version 1 and is canonical
                   ; (§ Format Declaration)
saiv-decl  = ( ".!saiv" / ".!csaiv" ) [ 1*ws version ] 1*ws
                   ( library-path / url ) [ 1*ws "strict" ] eol
                   ; .!saiv authored, .!csaiv emitted by the
                   ; schema compiler; bare form means version 1 —
                   ; version and identity are distinguished by
                   ; shape (§ Format Declaration)
taiv-decl    = ".!taiv" [ 1*ws version ] 1*ws library-path eol
                   ; faiv-decl (.faiv header) in § Unit Definition Files
maiv-decl     = ".!maiv" [ 1*ws version ] eol
                   ; bare form means version 1; no identity token
                   ; (§ Mappings — identity is the endpoint pair)

; .maiv header declarations (§ Mappings):
source-decl      = ".!source" 1*ws ( library-path / url ) eol
target-decl      = ".!target" 1*ws ( library-path / url ) eol
```

```
via-decl         = ".!via" 1*ws library-path eol
drop-decl        = ".!drop" 1*ws maiv-namepath eol

schema-decl      = ".!schema" ( ":" schema-registry-ref
                              / 1*ws ( library-path / url ) ) eol
schema-registry-ref = [ ns-path 1*ws ] ( library-path / url )
                    ; .!schema:acme/x  |  .!schema hub/x
                    ; |  .!schema:/server hub/x
                    ; |  .!schema:/@arr hub/x  |  .!schema URL
                    ; the flat space-separated and colon forms are
                    ; equivalent for an unscoped ID; ns-path is
                    ; defined under § Content lines

types-decl       = ".!types" 1*ws library-path eol
units-decl       = ".!units" 1*ws library-path eol
verbatim-decl    = ".!verbatim" eol
                    ; no arguments; .kaiv and .raiv only — values are
                    ; fully verbatim ("$" is a literal character; no
                    ; doubling, no references); carried into .raiv by
                    ; the Compiler, discharged by the Denormalizer
                    ; (§ Verbatim Documents)
registry-decl    = ".!registry" 1*ws path-seg "=" registry-base eol
registry-base    = url / fs-path
fs-path          = <a filesystem path — absolute, or relative to the
                    document's directory (§ Type Registry Resolution)>
provenance-req-decl = ".!provenance:" ( "required" / "source" / "none" )
   eol
source-id-decl   = ".?" prov-ident 1*ws uri eol

ref-decl         = ".!ref:" bare-name 1*ws library-path eol           ;
   reserved (corpus spec)
compose-decl     = ".!compose:" *any-char eol                         ;
   reserved (corpus spec)
msaiv-decl = ".!msaiv" [ 1*ws version ] 1*ws library-path eol
                                                                      ;
   reserved (corpus spec)
bind-decl        = ".!bind:" *any-char eol                            ;
   reserved (corpus spec)
unique-decl      = ".!unique:" *any-char eol                          ;
   reserved (corpus spec)
fk-decl          = ".!fk:" *any-char eol                              ;
   reserved (corpus spec)

url              = <an absolute http(s) URI per RFC 3986>
uri              = <a URI per RFC 3986, or an opaque non-whitespace
   identifier>
```

The reserved productions are recognized by the Lexer (keyword membership in §2.1.1) with their interiors deliberately opaque: their normative grammar belongs to the corpus specification (§6), and this document only pins that the keywords can never be reassigned.

10.4 Type References, Constraints, Units, Provenance

```
core-type     = "int" / "float" / "bool" / "null" / "b64" / "text" / "str"
   / "map"
                ; "map" is a structural type constructor (§ Map Type), not
   a
                ; std/core named type — unlike the others it is not defined
                ; as str + constraints. Bare "!map" annotates authored map
                ; assignments; "map<T>" (map-type) is the schema form.
type-ref      = core-type / library-path "/" path-seg
                ; !int  |  !std/net/port — last segment is the type name
type-name     = bare-name                     ; the &name= definition in .
   taiv
union-alt     = type-ref *inline-constraint [ ":" unit-expr ]
                                              ; constraints and the unit
                                              ; attach to the alternative
                                              ; they follow (§ Tagged
   Unions)
union-type    = union-alt *( "|" union-alt )  ; schema position; on an
   authored
                                              ; data line it is sugar the
   Compiler
                                              ; resolves to the active
   variant
                                              ; (§ Null Semantics) —
   canonical
                                              ; metadata prefixes carry no
   union
csaiv-alt     = type-ref [ ":" unit-expr ] [ "(" *lowered-item ")" ]
                ; compiled form: each alternative's lowered definition +
                ; narrowing, concatenated whitespace-free (items are
                ; self-delimiting); bare when the group is empty. The
                ; unit rides the alternative, never the union — an
                ; alternative without a unit demands none
csaiv-union   = csaiv-alt *( "|" csaiv-alt )
lowered-item  = pattern / span / range / enum / length
map-type      = "map<" type-ref ">"           ; schema annotation position
   only

pattern       = "/" pattern-body "/"
              / re-literal                        ; authoring-only
   alternative
pattern-body  = *( ( "\" any-char ) / p-char )
re-literal    = "re" re-sep *re-char re-sep       ; no escaping: re-char is
    any
re-sep        = ":" / ";" / "%" / "~" / "@" / "#" ; character except the
   line's
                                                  ; opening re-sep, "'", CR
   , LF;
                                                  ; lowered to the "/" form
p-char        = <any-char except "/", "\", and "'">
```

```
                ; the closing delimiter is the first "/" not preceded by
    "\";
                ; "\/" inside the body is passed to the regex engine
    verbatim
                ; (§ The Constraint Triple). Sole escape in the kaiv family
    .
                ; "'" is excluded — like ep-char/em-char — so the metadata
    /
                ; namepath delimiter is always the first "'" on a canonical
                ; line, keeping the split a single memchr (§ Parallel Scan
                ; Validation). A literal apostrophe in matched data is
                ; expressible as \x27 — the hex escape, below.

range           = "[" [ endpoint ] "," [ endpoint ] "]"
endpoint        = 1*ep-char
ep-char         = <any-char except ",", "]", "'", SP, HTAB>
enum            = "{" enum-member *( "," enum-member ) "}"
enum-member     = 1*em-char
em-char         = <any-char except ",", "}", "'", SP, HTAB>
length          = "#" ( range / enum )

span            = "..num" / "..lex" / "..lex[" locale "]" / "..time" / "..ver
    "
locale          = <a BCP 47 language tag>       ; Level 3 only

inline-constraint = pattern / range / enum / length
                ; concatenated, whitespace-free, at most one of each kind;
                ; kinds are commutative predicates (§ Composability), but
    the
                ; canonical order emitted by tooling is pattern, range,
    enum,
                ; length

unit-expr       = unit-term *( ( "*" / "/" ) unit-term )
unit-term       = unit-factor [ "^" exponent ]
unit-factor     = "1" / unit-name / currency
unit-name       = 1*ALPHA                       ; a built-in (base/derived/
    prefixed)
                                                ; or kfaiv.com unit; ASCII
    letters,
                                                ; "u"=micro, "ohmΩ"= (§ Built
    -in units)
currency        = "~" 3( %x41-5A )              ; ISO 4217 shape: 3 uppercase
     letters
                                                ; (well-formed; membership
    unchecked)
exponent        = [ "-" ] %x31-39 *DIGIT
                ; negative exponents authoring-only; canonical exponents
    are
                ; positive integers >= 2 (§ Canonical form: ASCII-sorted
    factors)

provenance-list = provenance *( "," provenance )
```

```
                ; a value may carry several sources; canonical form
   preserves
                ; the authored (first-seen) order of the list -- it is not
   sorted
provenance    = prov-ident [ "@" timestamp ] [ "#" prov-ident ]
instant       = instant-ext / instant-basic
instant-ext   = 4DIGIT "-" 2DIGIT "-" 2DIGIT "T"
                2DIGIT ":" 2DIGIT ":" 2DIGIT "Z"
                ; 2026-07-28T14:00:00Z, 20 chars — CANONICAL.
                ; The ":" here is unambiguous: a provenance run
                ; opens with "?" and runs to whitespace or "'",
                ; so it never reaches the unit sigil's branch.
instant-basic = 8DIGIT "T" 6DIGIT "Z"
                ; YYYYMMDDTHHmmSSZ, 16 chars — DEPRECATED.
                ; Accepted through the 1.0-draft series, never
                ; emitted, removed at 1.0.
```

Regex dialect. Pattern bodies use a deliberately restricted dialect chosen to keep validation finite-state: literals, character classes `[...]` (with ranges and negation), `.`, anchors `^` `$`, grouping `(...)`, alternation `|`, quantifiers `*` `+` `?` `{m}` `{m,n}`, and the escapes `\d` `\.` `\/` `\\` and `\xHH`. Backreferences, lookaround, and lazy quantifiers are excluded — they are incompatible with the constant-memory DFA execution model (§B.3). An escaped ASCII letter or digit other than `\d` and `\x` (`\1`, `\w`, `\s`, `\b`, ...) is **outside the dialect and rejected** — treating it as a literal would silently change the pattern's meaning relative to the dialects these shorthands come from. The conformance suite exercises exactly this subset; a pattern outside it is an `INVALID_CONSTRAINT_ERROR`. A pattern body additionally may not contain a literal `'` (`p-char`, above), which keeps the metadata/namepath delimiter the first `'` on the line.

The `\xHH` escape — exactly two hexadecimal digits, either case, naming the ASCII character `%x00`–`%x7F` — is the one letter escape beyond `\d`, admitted because it means the same character in every dialect that has it. It is valid both as an atom and as a character-class member or range endpoint (`[\x20-\x7e]`). A truncated form, a non-hex digit, or a value past `7F` is outside the dialect. `\x27` is how a pattern matches the one character its body cannot contain literally — the `'` delimiter — which RFC-faithful types need: `'` sits in RFC 3986's `sub-delims` and in the e-mail address grammar.

10.5 Metadata Annotation Lines (Rule 6 — Authored Files Only)

```
metadata-line   = type-annotation-line / prov-annotation-line
                / named-annotation-line / constraint-line
                / faiv-def-line          ; .faiv only (§ Unit Definition
   Files)

type-annotation-line  = "!" ( union-type / map-type / type-ref *inline-
   constraint
                        [ ":" unit-expr ] ) [ 1*ws re-literal ] eol
                      / "!:" unit-expr [ "?" provenance-list ] eol
                        ; elided-type unit annotation (§ Elided-Type Unit
```

```
                        ; Annotation): inherits the governing head, else
                        ; float; no inline constraints, no re-literal
prov-annotation-line  = "?" provenance-list eol
named-annotation-line = "&" type-name *( 1*ws constraint-item ) eol

; constraint lines — space-separated items above a .taiv &name=
; definition, or above a .saiv field definition as an anonymous
; refinement of the implicit str (value items only — no "!"/"&"
; items on the .saiv form):
constraint-line = constraint-item *( 1*ws constraint-item ) eol
constraint-item = pattern / span / range / enum / length
                / "!" type-ref *inline-constraint [ ":" unit-expr ]
                / "&" type-name
                ; a re-literal item stands alone between whitespace
                ; boundaries — never glued to a neighboring item
```

10.6 Content Lines (Rule 5)

Canonical data lines (`.daiv` / `.raiv`):

```
canonical-line   = metadata-prefix "'" canonical-namepath "=" value eol
metadata-prefix  = "!" type-ref *inline-constraint [ ":" unit-expr ]
                   [ "?" provenance-list ]
                 / "!:" unit-expr [ "?" provenance-list ]
                   ; the elided-type form is .raiv-only — resolved by
                   ; the Denormalizer, never present in .daiv
                   ; (§ Elided-Type Unit Annotation)

canonical-namepath = canonical-steps "::" terminal    ; /a/b::f  |  /@a::0
                   / "::" name                        ; root field: ::f
canonical-steps  = 1*( "/" step )
step             = name / "@" name / index
terminal         = name / index
```

Structural constraints the regular grammar does not capture (enforced by the Validator against the schema, accepted by the Lexer): an `index` step or terminal is meaningful only directly after an `@`-step; in `.raiv`, `$`-field references may still appear in value position (§2.5.10), and the elided-type prefix `!:unit` may still appear — resolved by the Denormalizer, never present in `.daiv` (§2.7.4).

Compiled schema lines (`.csaiv`):

```
csaiv-line       = csaiv-field-line / collection-line / map-coll-line
                 / deleg-line
                 / declaration-line / comment-line / doc-line / blank-line
csaiv-field-line = csaiv-constraint "'" csaiv-namepath [ "?" ] "=" [ value
   ] eol
                   ; the right side is the compile-time-resolved
                   ; applicable default (§ Default Values); the
                   ; Validator ignores it
map-coll-line    = ns-path 1*( 1*ws map-clause ) eol
map-clause       = "[key::" pattern "]"
                 / "[" ( "min" / "max" ) "=" 1*DIGIT "]"
```

```
                  ; key-pattern and entry-count bounds are each
                  ; independently optional — at least one clause
                  ; (§ Maps in the Compiled Schema); ns-path has no "@"
deleg-line       = ns-path 1*ws deleg-clause eol
deleg-clause     = "[schema::" schema-alt *( "|" schema-alt ) "]"
                  ; delegated namespace (§ Delegated Namespaces in the
                  ; Compiled Schema): the namespace's entire compiled
                  ; presence — it contributes no field lines of its own
csaiv-constraint = csaiv-item *( 1*ws csaiv-item )
csaiv-item       = pattern / span / range / enum / length
                 / "!" ( csaiv-union
                       / type-ref [ ":" unit-expr ] )
                  ; the retained head type item (§ The Schema Compiler):
   the
                  ; authored head name ("!int", "!text",
                  ; "!std/time/datetime", "!acme/net/port"), optionally
                  ; unit-carrying ("!type:unit" — the Validator
                  ; byte-compares the unit), or a union (the data-line
                  ; discriminant). The identity head "str" is
                  ; retained iff authored — the identity declaration;
                  ; an unannotated field carries no type item, and its
                  ; lex-saver, when one is needed, is the vacuous
                  ; pattern "/^/" — a constraint item, not a head.
                  ; Never "&".
                  ; canonical item order: head type, pattern, span,
                  ; range/enum, length
csaiv-namepath   = canonical-namepath
                 / canonical-steps "/" "::" name   ; element-level: /
   @servers/::host
                 / canonical-steps "::"            ; empty terminal: @ in
   steps ->
                                                   ; scalar-array element
   (/@ports::);
                                                   ; no @ -> map entry
                                                   ; (/config/settings::)
```

The elided index position (`/@servers/::host`, `/@ports::`) marks an element-level schema line — the constraint applies to every indexed element (§7.4).

Mapping lines (`.maiv`, §8):

```
maiv-map-line   = maiv-namepath "=" maiv-rhs eol
maiv-rhs        = "$" maiv-namepath [ "|" ( override-value / "!null" ) ]
                / value              ; constant injection / default; a
                                     ; constant starting with "$" is
                                     ; written with "$$" doubling
override-value  = <value not containing "|">
                  ; a literal "|" in an override constant is a
                  ; DelimiterCollisionError (§ Errors)
maiv-namepath   = [ maiv-steps ] "::" ( name / index )
maiv-steps      = "/" maiv-step *( "/" maiv-step )
maiv-step       = step / "*"
                  ; "*" (every element) is valid only directly after
                  ; an "@"-step, in the same position as an index
```

Authored data lines (`.kaiv`) and schema field definitions (`.saiv`):

```
kaiv-content     = kv-line / append-line / extend-line / struct-line
                 / append-struct-line / map-assign-line / var-def-line
                 / var-splat-line
kv-line          = key "=" value eol
key              = name / authored-namepath
authored-namepath = [ "/" ] step *( "/" step ) "::" ( name / index )
                    ; leading "/" conventional; bare segments tolerated in
   sugar
append-line      = array-path "+=" value eol
extend-line      = array-path ";=" ext-value *( ";" ext-value ) eol
ext-value        = <value not containing ";">
                   ; ";" inside data → use "+=" (§ Arrays)
struct-line      = ns-path ":=" ( pair *( "|" pair ) / ns-var-ref ) eol
append-struct-line = array-path "+:=" ( pair *( "|" pair ) / ns-var-ref )
   eol
pair             = ( name "=" value-np ) / ( "@" name ( "+=" / ";=" ) value
   -np )
value-np         = <value not containing "|">      ; "|" in data → separate
    lines
array-path       = [ "/" ] *( step "/" ) "@" name
ns-path          = "/" step *( "/" step )

map-assign-line  = ns-path "=" map-value eol
                   ; namespace map assignment — requires a !map
                   ; annotation above (§ Map Type). With a bare-name
                   ; key (root-level map) the line is an ordinary
                   ; kv-line; the !map annotation selects the
                   ; interpretation.
map-value        = "{}" / map-entry *( ";" map-entry )
map-entry        = map-key ":" map-val
map-key          = <text not containing ":" or ";">
map-val          = <text not containing ";">
                   ; ":" / ";" cannot appear literally in inline
                   ; entries — author such entries as namespaced
                   ; field lines instead (§ Map Type)

var-def-line     = "." bare-name "=" value eol                        ;
   scalar
                 / "@." bare-name ( "+=" / ";=" ) value eol           ;
   array
                 / "/." bare-name ":=" ( pair *( "|" pair )
                                       / ns-var-ref ) eol            ;
   namespace

ns-var-ref       = "$/." bare-name
var-splat-line   = ns-var-ref eol
                   ; namespace-variable splat as a standalone line:
                   ; valid only inside an open section or namespace
                   ; block; expands the variable's pairs at that
                   ; point (§ Namespace-Variable Splat)
```

```
; reference tokens embedded in values (§ Field References, token
; boundaries): the token is the longest run of reference characters;
; a trailing separator is not part of it; the first character outside
; the set ends the reference and the remainder is literal text.
; In a document declaring .!verbatim (§ Verbatim Documents) these
; tokens do not exist — every "$" in a value is literal.
hidden-ref       = "$." 1*ref-char
field-ref        = "$" ( ref-name / ref-path "::" ref-name )
ref-path         = ref-step *( "/" ref-step )
ref-step         = [ "@" ] ref-name
ref-name         = 1*ref-char                ; bare names and indices
ref-char         = ALPHA / DIGIT / "_"

saiv-field-line  = key [ "?" ] "=" value eol
                   ; "?=" optional, "=" required; the right side is the
                   ; field's DEFAULT — a kaiv value is never absent, only
                   ; empty, so an empty right side is the empty-string
                   ; default; the type annotation is the metadata line
                   ; above (§ Named Types in Schemas)
saiv-vector-line = array-path ";=" eol
                   ; scalar-array declaration: the annotation above
                   ; constrains every element; compiles to the
                   ; element-level `items'/@path::=` line

taiv-def-line    = "&" type-name "=" value eol
                   ; .taiv type definition: the right side is the
                   ; type's default value (empty for none); the
                   ; constraint lines are the metadata lines above
                   ; (§ Named Types)
```

10.7 Authored Structure Lines (Blocks — `.kaiv` / `.saiv`)

```
section-open   = "[" array-path [ 1*ws table-header ] "]" eol   ; [/
   @servers]
section-close  = "[]" eol
ns-block-open  = "(" ns-path [ 1*ws "schema:" schema-alt
                 *( "|" schema-alt ) ] ")" eol
                 ; a data block names ONE schema (the selection); a
                 ; parent-schema block may name the SET — the
                 ; discriminated schema set (§ Namespace-Scoped Schemas)
schema-alt     = library-path / url
ns-block-close = "()" eol

; Level 2 table headers and .csaiv collection-constraint lines:
table-header   = table-clause *( 1*ws table-clause )
table-clause   = clause-group / card-clause
clause-group   = clause-spec *( "|" clause-spec )    ; id=!|customer=/
   @customers/*::id
clause-spec    = unique-spec / fk-clause
unique-spec    = name "=!" *( "," name "=!" )        ; [/@servers host=!,
   port=!]
```

```
fk-clause      = name "=" fk-path                    ; department=/
   @departments/*::name
card-clause    = ( "min" / "max" ) "=" 1*DIGIT
fk-path        = "/" *( step "/" ) "@" name "/*::" name

collection-line = array-path 1*ws coll-clause-group *( 1*ws coll-clause-
   group ) eol
coll-clause-group = coll-clause *( "|" coll-clause )
coll-clause    = "[unique::" name *( "," name ) "]"
               / "[ref::" name "=" fk-path "]"
               / "[" ( "min" / "max" ) "=" 1*DIGIT "]"

; .saiv map schema blocks (ostensive key constraints,
; see Maps in the Compiled Schema):
map-block-open = "(" ns-path *( 1*ws card-clause ) ")" eol
                                       ; (/name min=1 max=40)
map-key-line   = pattern "=" eol       ; /^[a-z]{2,3}$/= — recognized
                                       ; whole-line ahead of the rule-5
                                       ; split (§ The Six Rules): the
                                       ; pattern body may contain "="
```

In table headers, `min` and `max` are **reserved words**: a clause of the form `min=N` or `max=N` (with `N` all digits) is always a cardinality clause. An array element field literally named `min` or `max` MUST use a quoted name (`"min"=!`) to appear in a table header. Together with the pattern-literal introducer `re` (reserved in leading name position of `.saiv`/`.taiv` content lines — §2.6.11), this is the complete reserved-word set.

10.8 Values

There is no value grammar beyond `value = *any-char`: values are verbatim byte sequences (§9.5). Typed interpretation of a value (integer syntax, base64 alphabet, ISO 8601 shape) is a *constraint check* performed by the Validator against the compiled pattern — never a lexical concern. The two-layer split is exact: the Lexer knows lines, the Validator knows constraints, and nothing in between parses values.

11 Errors

This section catalogs the error conditions a conformant implementation detects. Errors are grouped into two categories:

- **Lexer errors** are raised at parsing time, when a line violates the kaiv syntax (the regular grammar). The Lexer detects them before the Compiler/Denormalizer/Validator stages run.
- **Application errors** are raised at validation time, when the data does not conform to its schema. Detected by the Validator (or an integrity check) against a `.csaiv`, or by the application consuming the token stream.

In case of multiple errors on the same line, Lexers MUST report the highest-priority error (in the order listed below), and MAY report further errors from the same line.

11.1 Lexer Errors

Error	Condition	Notes
`BOM_ERROR`	Text begins with the UTF-8 BOM `EF BB BF`.	Detected before line parsing begins; reported without a line number.
`INVALID_UTF8_ERROR`	Input contains invalid UTF-8 sequences.	
`INVALID_CHARACTER_ERROR`	Standalone CR (U+000D) not part of CRLF, or NUL (U+0000).	
`MISSING_FINAL_EOL_ERROR`	The final non-empty line lacks an EOL terminator.	MAY be reported without a line number.
`INVALID_VERSION_ERROR`	The version following a format declaration keyword does not match `^[0-9]+(\.[0-9]+){0,2}$`. The version is optional for every kind (the bare form means version 1). On the identity-carrying kinds (`.!saiv` / `.!csaiv` / `.!taiv` / `.!faiv` / `.!msaiv`) a digit-first first token is read as the version slot (an identity is alpha-first by grammar), so a malformed digit-first token — `1.2.3.4`, `1x` — is this error, not an identity error.	See §2.1.2 — omitted components are zero, so `1` ≡ `1.0` ≡ `1.0.0`.
`UNSUPPORTED_VERSION_ERROR`	The version is well-formed but not one supported by the implementation.	A 1.x implementation MUST NOT raise this for any earlier 1.y where y < x.
`FORMAT_KIND_ERROR`	A stream consumed as a kind whose declaration is not optional — the canonical kinds (`.raiv`, `.daiv`, `.csaiv`) and the REQUIRED-header authored kinds (`.saiv`, `.taiv`, `.faiv`, `.maiv`) — does not open with the matching format declaration: the declaration is absent, or names a different kind (e.g. a `.!raiv` stream fed to a `.daiv` consumer). Only authored `.kaiv` may omit the declaration (§2.1.2).	Raised by the consuming stage's Lexer, which is told the expected kind. This is what keeps the family self-describing: kind checks need no filename or out-of-band context.

Error	Condition	Notes
`EMPTY_KEY_ERROR`	A data line starts with `=` (optionally preceded by whitespace) — the key is empty or pure whitespace.	
`MISSING_OPERATOR_ERROR`	A line that is neither blank, comment, declaration, structure line, variable-splat line, `.csaiv` collection-constraint line, nor metadata-annotation does not contain `=`.	

Error	Condition	Notes
`INVALID_KEY_ERROR`	A bare key — or **any unquoted segment** of an authored namepath — does not match `bare-name = ( ALPHA / "_" ) *( ALPHA / DIGIT / "_" )`: e.g. a leading digit (`9port`), a hyphen (`retry-count`), or a dot (`a.b`). Or a quoted key is empty or contains a literal `"` other than as the `""` doubling.	The Lexer MUST validate every bare namepath segment and raise this on violation. An **index segment** (`index`: all digits, no leading zero) is bare by definition and exempt from the bare-name shape. Names with `-`, a leading digit followed by non-digits, or other out-of-set characters MUST be quoted (§2.3). `path-seg` (which permits `-` and a leading digit) applies only to library paths, schema IDs, and provenance identifiers — never to data names. A **reserved word** used unquoted where it is reserved also raises this error: `re` in leading name position of a `.saiv`/`.taiv` content line (§2.6.11), and `min`/`max` as unquoted field names in a Level 2 table header (§10.7). A `.saiv` map-key line is exempt: its key is a whole pattern, not a name (§1.3.1).

Error	Condition	Notes
`INVALID_DIRECTIVE_ERROR`	A line beginning with `.!` (after stripping leading whitespace) does not begin with a keyword from the §2.1.1 table: `.!kaiv`, `.!raiv`, `.!daiv`, `.!saiv`, `.!csaiv`, `.!taiv`, `.!faiv`, `.!maiv`, `.!msaiv`, `.!schema`, `.!source`, `.!target`, `.!via`, `.!drop`, `.!types`, `.!units`, `.!verbatim`, `.!registry`, `.!provenance`, `.!ref`, `.!compose`, `.!bind`, `.!unique`, `.!fk`.	The inventory table in §2.1.1 is authoritative; this list mirrors it.
`INVALID_CONSTRAINT_ERROR`	A constraint clause does not match the constraint productions (range `[min,max]`, enum `{a,b,c}`, length `…#[]`, etc.), or a unit annotation names a unit that resolves against neither the built-in set nor an imported `.faiv` library (§2.7.10).	Malformed clauses are Lexer-detected wherever constraints appear — schemas, `.taiv` constraint lines, and metadata annotations in authored `.kaiv`. Unknown-unit membership is Lexer-detected when no `.!units` import leaves the namespace open (closed built-in set), and Compiler-detected otherwise (resolution required).

Lexers SHOULD reference line numbers in error reports.

11.2 Application Errors

Schema-related application errors come in two naming conventions:

— `Schema*Error` — the schema definition itself is malformed.
— `*SchemaError` — a data text violates a schema constraint.

Compiler-stage errors (resolution and expansion failures in authored text) carry plain descriptive names.

Error	Condition	Detected by
`MetadataWithoutTargetError`	A metadata annotation in authored `.kaiv` is not followed by a data line before the next blank line, comment, or EOF. A **stack** of at most one type-designating annotation (`!…type` or `&name`) plus at most one provenance annotation above a single data line is permitted (§1.3.4); a second annotation of the same kind also raises this error.	Compiler
`UndefinedReferenceError`	In a document not declaring `.! verbatim` (§2.5.6): a value references a hidden variable (`$.x`, `$@.x`, `$/.x`) or a data field (`$field`, `$path::field`) that is not defined on an earlier line (§2.5.4); or a value contains a `$` that begins neither a well-formed reference nor the `$$` doubling. Not possible in a verbatim document — every `$` there is literal.	Compiler
`VariableContextError`	In a document not declaring `.! verbatim`: a container-variable reference appears where its expansion cannot be placed: `$/.name` outside the two splat positions (§2.5.2), or `$@.name` in a scalar position. Also raised for a field reference (`$field`, `$path::field`) inside a hidden-variable definition's value — variables resolve before the referenced field is guaranteed stable, so they cannot capture field values. In a verbatim document the machinery itself is banned — see `VerbatimContextError`.	Compiler

Error	Condition	Detected by
`VerbatimContextError`	A document declaring `.! verbatim` (§2.5.6) contains reference machinery in a positional form: a variable-definition line (`.name=`, `@. name+=/;=`, `/.name:=`), a standalone splat line (`$/.name`), or a compound-form right side that is entirely a splice (a `:=/+:=` right side `$/.name`; a `+=/;=` value exactly `$@.name`). A `$` inside value text is never this error — it is a literal character. Also raised when `.!verbatim` itself is misused: carries arguments, repeats, follows a content line (declarations precede content, §2.1), or appears in a kind other than `.kaiv`/`.raiv` (there, by the consuming stage).	Compiler; consuming stage (out-of-place declaration)
`DelimiterCollisionError`	A compound-form value collides with that form's delimiter: a `:=/+:=` pair value containing `|` (the segment after the split is not a well-formed pair), or an inline map entry without `:` after a `;` split. The colliding character is representable only in the single-value `key=value` form (§9.5). (A `;` in `;=` data is not a detectable collision — it simply delimits further elements, which is why such values are authored with `+=` instead.) Also raised where a retype would reinterpret delimiter bytes: the str→text coercion on a value carrying a literal `|:|` (§2.6.4), and a `.maiv` override constant containing `|` (§8.3).	Compiler; Denormalizer / Validator (text coercion); mapper, at publish (override)

Error	Condition	Detected by
`UnitConversionError`	An authored value in a same-dimension unit cannot be converted exactly into the field's declared unit: the exact decimal result is non-terminating, or violates the field's type — e.g. a non-integer result on an int-derived field (§2.7.3).	Denormalizer
`SchemaDuplicateKeyError`	A `.saiv` schema text contains duplicate field definitions.	Schema compiler
`SchemaResolutionError`	A `.!schema`, `.!types`, `.!source`, `.!target`, or `.!via` reference cannot be retrieved from the specified URL or registry.	Compiler / mapper
`RegistryStrictError`	Strict resolution mode is enabled and a document-level `.!registry` declaration would determine the base of a resolved artifact (§2.1.4). Raised before any retrieval is attempted.	Resolver, strict mode only
`SchemaInheritanceCycleError`	A `.!schema` inheritance chain among `.saiv` files — or a delegation-member chain (§3.5.3) — revisits a schema already in the chain (§3.5.6).	Schema compiler
`SchemaDelegationError`	A parent schema both delegates a namespace and declares fields under it, or declares an empty or duplicate-member schema set (§3.5.3).	Schema compiler
`DelegationSchemaError`	A document whose schema delegates a namespace carries no scoped `.!schema` declaration for it, or the declaration names a schema outside the declared set (§7.6).	Denormalizer / Validator
`SchemaOptionalWithoutDefaultError`	An optional field (`?=`) whose resolved default is inapplicable (fails the field's own constraints, §2.6.13) and whose type does not admit `!null` — the Denormalizer would have nothing to materialize for an absent instance.	Schema compiler; Denormalizer, on a stale `.csaiv` predating the compile-time check

Error	Condition	Detected by
`RequiredFieldSchemaError`	At build time: the authored data omits a field declared required (`=`) — the Denormalizer materializes optional fields only (§2.6.17). At validation time: a schema-declared field has no corresponding `.daiv` line — materialization guarantees every declared field a line, so a missing line of any kind fails the parallel scan.	Denormalizer / Validator (Pass 1)
`DuplicateKeySchemaError`	A data text contains two or more entries for a single field defined in the schema — a name the schema declares exactly, or a group field repeated within one namespace-array element (§7.1). Detection checks the scanned line against the *schema*, so memory stays schema-bounded. (Data-introduced names — fields not defined in a relaxed schema, and collection entry keys — are unaffected; their duplicate handling is application-level, §9.8.)	Validator
`UndefinedFieldStrictSchemaError`	A data text contains an entry for a field not defined in a strict schema.	Validator

Error	Condition	Detected by
`TypeMismatchError`	The data line's type annotation fails the field's nominal requirement (§7.1): no union alternative matches the discriminant; a `!null` or `std/enc`-typed line meets a non-`null` / non-embed head (or vice versa); a non-`text` annotation meets a `!text` head (or `!text` data meets any other head); a non-`str` annotation meets an explicit `!str` identity head; or the canonical unit is not byte-identical to the declared one. A structurally-checked head raises this only for `!null`-, `!text`-, or `std/enc`-typed data lines; any other annotation is accepted, with the head's constraint group governing the value — a wrong-shaped *value* fails as `ConstraintViolationError` instead. Also raised by the Compiler when a union-annotated authored value satisfies no alternative (active-variant resolution, §2.6.17).	Validator; Compiler
`ConstraintViolationError`	A data value fails a pattern, range, enumeration, or length constraint.	Validator
`IncompleteMappingError`	A `.maiv` leaves a *required* target-schema field unproduced: no mapping line, no constant line, and no applicable target-schema default (§8.5). Checked statically at publish time.	Mapper / registry publish

Error	Condition	Detected by
`ProvenanceSchemaError`	A data line violates the schema's `.!provenance` requirement level (§2.4.3): missing source/timestamp under `required`/`source`, or any provenance under `none`. Also raised by the schema compiler when a `required`/`source` level is combined with an optional field — materialized lines carry no provenance, so the combination is statically unsatisfiable.	Validator; Schema compiler
`UniquenessViolationError`	A field declared `[unique::field]` (Level 2) has duplicate values across array elements.	Pass 2 (application layer, §4.7)
`ReferentialIntegrityError`	A field declared `[ref::field=/@path]` (Level 2) has a value not present in the referenced field set.	Pass 2 (application layer)
`CardinalityViolationError`	An array's element count violates `min=N` or `max=M`.	Validator (Pass 1)
`CollationUnsupportedError`	A Level 0–2 runtime encountered `..lex[locale]` and was configured to reject (rather than fall back to bare `..lex`); **or** a Level 3 validator encountered a `..lex[locale]` constraint it cannot evaluate exactly per §5.3 — an unsupported locale or tailoring, a well-formed tag whose primary language subtag has no CLDR 48 tailoring, or an unrecognized `-u-` extension key — and rejected it under the honest-partial rule rather than order differently.	Validator (Levels 0–3)

Strict-vs-relaxed schemas: schemas are *relaxed* by default — data texts MAY contain fields not defined in the schema, without any `UndefinedFieldStrictSchemaError` being raised for them. A schema marked strict requires that every field in the data text is declared. The strict modifier is the literal keyword `strict` as the final space-separated token of the `.!saiv` declaration:

```
.!saiv acme/server-config strict
```

The schema compiler carries the declaration — identity, any modifier, and any explicit non-1 version — into the `.csaiv` header, rewriting the keyword to `.!csaiv` (§2.1.2); that header is where the Validator reads the modifier. A `.csaiv` whose header carries no `strict` modifier is relaxed.

12 File Representation

12.1 File Extensions

Files SHOULD use the corresponding extension from the family:

Extension	Role
`.kaiv`	authored data
`.raiv`	relational canonical
`.daiv`	denormalized canonical
`.saiv`	authored schema
`.csaiv`	compiled schema
`.taiv`	type library
`.qaiv`	query
`.msaiv`	metaschema (reserved — corpus specification, experimental; §6)
`.faiv`	unit definitions (factors)
`.maiv`	mapping

12.2 File Encoding

kaiv files MUST be encoded as UTF-8 without a Byte Order Mark.

12.3 Canonical Emission

Canonical artifacts are deterministic at the byte level. The pipeline stages that emit canonical kinds — the Compiler (`.raiv`), the Denormalizer (`.daiv`), and the schema compiler (`.csaiv`) — MUST:

- terminate every line, the final line included, with **LF** (`U+000A`) — never CRLF;
- emit no blank lines;
- place a carried `//` doc string (§2.2.2) immediately above the line it documents, in source order;
- never emit `#` comments (§2.2.1).

Consequently, two conforming builds of the same inputs MUST produce byte-identical canonical artifacts — the file-level extension of the per-line determinism the format already guarantees (single index spellings, deterministic quoting, canonical unit forms, exact-decimal unit conversion). This is what makes canonical artifacts content-addressable: a hash of the `.daiv` bytes identifies the build's output with no normalization step. Input tolerance is unaffected — consumers of canonical files still accept CRLF

and blank lines (§9.3, §9.4); emission discipline binds producers, not readers, and DFA processing is unchanged.

12.4 Media Types

Two media types are defined for kaiv content, following the XML precedent (RFC 7303). The choice depends on the intended use of the content.

Context	Recommended type	Rationale
API payloads, configuration delivery, machine-to-machine, streaming pipelines	`application/kaiv`	Safe default; preserves exact byte-for-byte content; no risk of EOL normalization or stripping.
Display in browsers, email, documentation systems, debugging output	`text/kaiv`	Signals human-readability; may enable inline display or syntax highlighting.

When in doubt, use `application/kaiv` — it is the safer choice for preserving data integrity across transport layers.

12.4.1 Provisional Types

Until IANA registration (RFC 6838) is obtained, implementations SHOULD use the provisional forms:

- `application/vnd.kaiv`
- `text/vnd.kaiv`

12.4.2 Parameters

Both media types support the following optional parameters:

- **`version`** — OPTIONAL. If present, the value MUST be a string containing the kaiv version number (e.g. `1.0`).
- **`charset`** — for `application/kaiv` it is OPTIONAL, and if present MUST be `utf-8`. For `text/kaiv` it is REQUIRED and MUST be `utf-8` (the `text/` top-level type defaults to US-ASCII per RFC 2046; omitting `charset=utf-8` may cause receivers to misinterpret non-ASCII content).
- **`kind`** — OPTIONAL. Names the family kind of the content: an extension name without the dot (`kaiv`, `raiv`, `daiv`, `saiv`, `csaiv`, `taiv`, `faiv`, `maiv`, `qaiv`, `msaiv` — §12.1). One line grammar, one media type: the kind is a parameter, mirroring the per-kind interpretation of the `=`-split (§1.3). In content negotiation, `Accept: application/kaiv;kind=saiv` requests the authored schema where a registry's default for the path is the compiled one (§2.1.3). Absent the parameter, the kind is read from the content's format declaration (§2.1.2) or the server's documented default.

12.4.3 Transport Considerations for `text/kaiv`

The `text/` top-level type carries inherent risks that senders MUST consider:

- **Line-ending normalization** — some transport systems may convert line endings (e.g. CRLF → LF or vice versa).
- **Final-EOL stripping** — some systems may remove the final EOL, causing a Lexer to raise `MISSING_FINAL_EOL_ERROR`.
- **Charset re-encoding** — without an explicit `charset=utf-8`, legacy systems may apply ASCII-targeted conversions.

Senders SHOULD use `application/kaiv` when exact byte-for-byte preservation is required, or when the transport path is not fully trusted to preserve `text/` semantics.

12.5 Shebang Lines

A *shebang line* is an optional first line in a kaiv file specifying how to execute the file as a script. It typically takes the form:

```
#!/path/to/interpreter [optional argument]
```

Rules:

- A shebang line MUST start with the bytes `#!` and is recognized **only on the physical first line** of the file.
- Shebang recognition happens **before** line classification (`document = [ shebang-line ] *line` in §10). The six-rule classifier never sees the shebang line; in particular, rule 2 (`#` comment) does not apply to it. Shebang lines are file-level directives, not part of the kaiv text — they MUST NOT be treated as comments.
- When a shebang line is present, the format declaration occupies the first line *after* it (§2.1.2).
- When a kaiv file begins with a shebang line, Lexers MAY emit a *shebang token* for it. If tokens are emitted out of order (parallel parsing), the shebang token MUST be emitted.
- Shebang tokens MUST preserve the entire shebang line verbatim except the EOL terminator, including the `#!` prefix.
- Shebang tokens MUST NOT be emitted for any line beyond the first.

A Terminology

> This appendix is the **single source of truth** for kaiv terminology, consolidated from the specification repository's former `TERMINOLOGY.md` (now a pointer here). It restates some body material in glossary form for quick reference; on any conflict the body text and the formal grammar (§10) win. A few entries — the ecosystem services and the time-series file types (`.aaiv`, `.iaiv`) — describe surface outside this specification's conformance boundary; they are included so the family's terminology has one home.

A.1 The Filesystem Analogy

The core structural concepts of kaiv map directly onto the Unix filesystem:

kaiv concept	Filesystem analog	Key property
namespace	directory	A container; you can descend into it
namespace path	directory path	An address pointing to a container — the *steps* of a namepath
field	file	A leaf value; terminal, nothing below it
namepath (= field address)	file path	The full address of a value: the steps plus the `::` projection

The `/` operator works exactly like the Unix path separator — it means "descend from root" and "separate path segments." The `::` operator has no filesystem analog; it marks the transition from the tree (containers) to a leaf value.

A namespace path (the *steps*) always points to a namespace (container). A namepath always points to a value (leaf): per the formal grammar (`canonical-namepath`, §10), the `::` field projection is the final, terminal step *of* the namepath — the boundary between container addressing and leaf access. **Field address** is a synonym for namepath, used when the leaf-addressing role needs emphasis.

A.2 Core Structural Concepts

Term	Definition
namespace	A data structure — a container that holds fields and/or other namespaces. The directory in the filesystem analogy. Namespaces use the `/` prefix. Examples: `/server`, `/app/db`, `/@servers/0` (a namespace array element). Arrays are a special form of namespace that uses integer (index) fields.
namespace path	The container-addressing portion of a namepath — a path of namespace names (the *steps*; `canonical-steps` in the formal grammar). Always points to a container, never to a leaf value. The directory path in the filesystem analogy. Examples: `/server`, `/app/db`, `/@servers/0`.
namepath	The full address of a leaf value within the tree: the namespace steps followed by the `::` field projection and a terminal name or index. Always points to a value. The file path in the filesystem analogy. Examples: `/server::host`, `/@servers/0::port`, `::host` (root field — the steps are empty). Synonym: **field address**.
name	A single identifier at one level of the tree; the scope-local label for a node. Each `/`-separated segment of the steps, or the terminal after `::`.

Term	Definition
field	A named leaf value inside a namespace. The file in the filesystem analogy. Accessed via the `::` field projection operator. A field is always terminal — you cannot descend further. Examples: `::host`, `::port`, `::timeout_ms`.
field address	Synonym for **namepath** — the steps plus the field projection, the full address of a leaf value. Always points to a value, never to a container.
field projection	The operation of selecting a field from a namespace, extracting a scalar value from a tree node. Uses the `::` operator. Always the last step in a field address — after `::`, the path is terminal.
literal	A leaf value in the data model — either a scalar field or a scalar array element. Literals are the terminal nodes of the tree; they are always reached via `::` field projection.
scalar array	A named, indexed collection of scalar values; each element is a leaf reached via `::index` field projection, where the numeric index is the field name. Syntax: `@name::index`.
namespace array	A named, indexed collection of namespaces (interior nodes); each element is an interior node descended into via `/index`, with its own fields reached by further `::` projection. Syntax: `@name/index`.
mixed array	A named, indexed array whose elements may be either scalar (`@name::index=value`) or namespace (`@name/index::field=value`). The operator before the index — `::` for scalar, `/` for namespace — discriminates element kind per element.

A.3 Operators and Annotations

Symbol	Formal name	Role	Example	Survives
`/`	tree descent operator	Root descent and path separator between namespace segments. Marks that what follows is a container (namespace or array) addressed from root. Present in both authored and canonical form.	`/server`, `/@servers/0`	[x]
`::`	field projection operator	Accesses a named leaf field inside a namespace. Always terminal in a field address — nothing can follow `::name`.	`::host`, `/server::host`	[x]
`@`	array annotation	Part of the identifier — marks that the named container is iterable by index. `@tags` is the name; the `@` is permanent. A root-level array uses `/@arr`.	`@tags`, `/@ports`	[x]
`.`	hidden name prefix	Part of the name for variable definitions — marks a hidden variable, elided from canonical output. Unix hidden-file convention.	`.host=localhost`	[ ]

Symbol	Formal name	Role	Example	Survives
`$`	dereference operator	Two roles. `$.name` (variable dereference): substitutes a hidden variable — resolved by the Compiler, does not survive to `.raiv`. `$field` / `$path::field` (field reference): substitutes another field's value — survives into `.raiv`, expanded by the Denormalizer into `.daiv`. Absent entirely in a verbatim document (§2.5.6).	`$.host`, `$server::host`	[ ]
`&`	named type reference	Short-form type annotation in authored files. Resolved to `!library/path/typename` by the Compiler. Never appears in canonical form.	`&int`, `&port`	[ ]
`'`	apostrophe delimiter	The type–namepath boundary in canonical form: on every canonical data line, `'` separates the metadata prefix (type plus optional provenance list) from the namepath. Split always on the first bare `'`.	`!str'::host=x`	[x] (canonical only)
`?`	provenance annotation	Per-value source attribution. Authored `.kaiv`: a standalone metadata line. Canonical `.daiv`/`.raiv`: an inline provenance list before `'`. References source IDs declared via `.?id uri`.	`?sensor23`	[x]

A.4 Sigil Categories

Category	Sigils	Survives	Why
Structural operators	`/`, `::`	[x]	Carry addressing information needed by the DFA and queries
Structural annotations	`@`	[x]	Part of the identifier — marks iterability
Resolution sigils	`.`, `$`	[ ]	Authoring mechanisms resolved by the Compiler (`.` and `$.var`) or by the Denormalizer (`$field`)
Authoring-only type ref	`&`	[ ]	Resolved to `!library/path/typename` by the Compiler

The DFA dispatch after `'` in canonical form follows from these categories; the dispatch table is in §1.4.1.

A.5 File Types

Ext	Full name	Role
`.kaiv`	kaiv — authored data file	Human-written data. May contain variables, namespace blocks, inline syntax, and other authoring conveniences. Source code.
`.daiv`	Denormalized Attributed Information Values — canonical data file	Machine-generated deployment artifact. One `!type 'namepath=value` line per field, fully qualified, in schema-declared order. No variables, no blocks, no authoring sugar, no `$field` references. Output of the Denormalizer.
`.raiv`	Relational Attributed Information Values — relational canonical form	Like `.daiv` but field references (`$path::field`) are preserved rather than resolved. Output of the Compiler. For design tools, diffing, auditing, and converters.
`.saiv`	authored schema file	Human-written schema. Field definitions are `key =default` content lines (`?=` for optional fields; root fields are bare keys), optionally preceded by a type-annotation metadata line.
`.csaiv`	compiled schema file	Machine-generated compiled schema. Single-line `constraint'namepath=` format (`?=` marks optional fields; the right side carries the field's resolved default). The artifact consumed by the Validator.
`.taiv`	type library file	Named type definitions. `&name=` pairs defining types in terms of `str` plus constraints.
`.qaiv`	query file	Queries over kaiv data using the qaiv query language. qaiv is derived from quarb (`quarb.org`) — shared surface syntax, separate streaming-DFA engine, plus kaiv-specific predicate sugars. See `kaiv/QUERY.md` §4 (in the spec repository).
`.faiv`	unit-definition file	**Factor** Attributed Information Values — custom unit libraries (§2.7.9): each definition is a dimension (by unit reference) plus a conversion factor to the dimension's base unit; currencies ($ dimension) carry a rate-source URL template instead. Served from `kfaiv.com`; imported with `.!units`.
`.maiv`	mapping file	Structural correspondence declarations between two schemas (§8). A mapping has no name of its own: its identity is the `.!source`/`.!target` endpoint pair, and its registry address derives from it (§8.2). Each mapping line assigns a `$`-reference into the source schema (or a literal constant) to a target namepath — a pure namepath rewrite with no value transformation beyond an optional override constant.

Ext	Full name	Role
`.msaiv`	metaschema file (reserved)	Corpus-level contract (Level 4 — reserved here, §6; specified separately in the experimental corpus specification, `corpus/DRAFT.md` in the spec repository). Declares schema bindings, corpus-wide uniqueness, and cross-document foreign keys over a corpus of `.daiv` files.
`.aaiv`	aggregate file (time-series; ecosystem)	Immutable, time-partitioned file holding many instances of one schema, each data point identified by its `#dpid`. Used by the `chraiv.com` services; specified in the sibling `chraiv-spec` repository. Outside this specification's conformance boundary.
`.iaiv`	predicate index file (time-series; ecosystem)	Hash-bucketed sorted predicate index for structured time-series queries; binary-searchable. Used by the `chraiv.com` services; specified in the sibling `chraiv-spec` repository. Outside this specification's conformance boundary.

A.6 Pipeline Stages

Term	Definition
authored form	The human-written representation (`.kaiv`, `.saiv`, `.taiv`). May contain variables, namespace blocks, inline syntax, and other authoring conveniences. Input to the Compiler.
canonical form	The fully denormalized, machine-generated representation (`.daiv`). No variables, no blocks, no authoring sugar, no `$field` references. One line per field, fully qualified. Output of the Denormalizer.
`$.var` (variable dereference)	A variable reference in authored `.kaiv`. The `.var` was defined with `.var=value`. Resolved by the **Compiler** during compilation; does not survive to `.raiv`. Pure authoring sugar.
`$field` (field reference)	A reference to the value of another field in the same document — the data is normalized because it refers to a value by name instead of repeating it. Survives compilation into `.raiv`; the **Denormalizer** expands it, replacing the reference with the actual value. That is what "denormalization" means in the database sense.
verbatim document	A document declaring `.!verbatim` (§2.5.6): values are fully verbatim ($ literal, $$ two dollars, no references) and the reference machinery is banned. The declaration is carried into `.raiv` and discharged by the Denormalizer — the `.daiv` output is byte-identical to the escaped-authored equivalent's.

Term	Definition
Compiler	The first stage (build-time). Reads `.kaiv`, resolves variables, expands blocks and inline syntax, resolves `&name` annotations, prepends `::` to root-level fields, and produces `.raiv` (field references preserved). Does not require the schema. Never runs on the certified target.
Denormalizer	The second stage (build-time). Reads `.raiv` and expands `$field` references. Schema-aware: when the data declares a schema, it also materializes absent optional fields (the resolved default or `!null`) in schema-declared order, and lifts untyped lines to each field's retained head type (§7.3) — for `!text` heads under the guarded str→text coercion — so `.daiv` carries the schema's types on its own lines. An absent required field is a build-time `RequiredFieldSchemaError`. Produces `.daiv`.
Validator	The third stage (build-time). Reads `.daiv` and the compiled schema (`.csaiv`). Performs schema constraint checking, field existence and order verification, obligation enforcement, and provenance enforcement. Produces pass/fail. Never runs on the certified target.
integrity check	Optionally re-running the Validator's validation logic on an existing `.daiv` at deployment or runtime. Same parallel scan against `.csaiv`, same constant-memory properties — different point in time. Not a separate pipeline stage.
parallel scan	The Validator's mechanism: reading `.daiv` and `.csaiv` line-by-line in lockstep, checking each canonical data line against the corresponding compiled constraint. A strict lockstep walk — materialization guarantees every schema-declared field a `.daiv` line — with only empty-collection element lines skippable. O(1) memory per field. Also used by the integrity check.

A.7 Type System

Term	Definition
`str`	The single primitive type — every value between `=` and end-of-line is a string of characters. All other types are named types defined in terms of `str`. Type annotation: `!str`.
named type	A reusable type defined in a type library (`.taiv`) in terms of `str` plus optional constraints. All types other than `str` are named types — including `int`, `float`, `bool`, `null`, `b64`.
type annotation	The `!type` prefix on a canonical data line or metadata annotation line. `!int`, `!float`, `!str`, `!bool`, `!null`, `!b64`, `!text` are canonical `std/core` shorthands; non-core types use `!library /path/typename`.

Term	Definition
constraint triple	`(pattern, span, range)` — the universal type constructor (§2.6.11). Every type is `str` narrowed by zero or more of: a pattern (`/regex/`), a span ordering, and a range. An enumeration (`{a,b,c}`) composes as a further narrowing constraint; a length constraint (`#[min,max]`) restricts character, element, or decoded byte count.
`std/core`	The standard type library — always implicitly imported, frozen after publication, shipped embedded. Defines `int`, `float`, `bool`, `null`, `b64`, `text`. Canonical shorthands do not expand further.
`std/enc`	The encoding library — named types derived from `b64` that type the decoded payload (`json`, `yaml`, `toml`, `xml`, `html`, `md`, `csv`, `bin`, `plain`). Shipped embedded, imported explicitly; canonical form is the full path.
`std/time`	The time library — RFC 3339 shapes with `..time` chronological ordering (`datetime`, `localdatetime`, `date`, `time`). Shipped embedded, imported explicitly.
`std/num`	The numeric-markers library — IEEE non-finite values as enum types (`inf`, `nan`). kaiv floats are deliberately finite; an extended-real field is the union idiom `!float|std/num/inf`. Shipped embedded, imported explicitly.
span ordering	Declares how range constraints are evaluated: `..num` numeric, `..lex` byte-lexicographic, `..lex[locale]` locale-aware (Level 3), `..time` chronological, `..ver` semantic version.
`!null`	The canonical representation of null: `!null'::field=` — the value after `=` is always empty. Distinct from the empty string (`!str'::field=`). Nullability is explicit and opt-in via `!null|T`.
type resolution	The build-time process of locating the `.taiv` file that defines a named type. Distinct from type identity: a type's canonical path is fixed regardless of which registry serves its file.
unit	An annotation attached to a numeric (`..num`) type via `:`, declaring the physical or economic quantity (§2.7). Compound units use `*` `/` `^` with a strict no-whitespace grammar; canonical form is ASCII-sorted. The Validator byte-compares units and never converts; the schema-aware build converts authored same-dimension units into the field's declared unit (exact decimal arithmetic only). The elided-type annotation `!::unit` inherits the governing head, else `float`.
dimension	The classification of a unit by physical character. Seven SI base dimensions plus the currency dimension `$`; compound units derive their dimension from their factors. The dimensionless unit `1` has the empty dimension product.
base unit	The reference unit within a dimension (conversion factor 1). Conversion factors live in unit definitions and are not consulted by the Validator.

Term	Definition
currency unit	A unit variant for monetary value: `~` plus an ISO 4217-shaped code (`~EUR`). All currency units share dimension `$`; definitions deliberately omit conversion factors — exchange rates are external and time-varying.

A.8 Schema and Validation

Term	Definition
`caiv`	The shared line grammar underlying every file in the kaiv family (§1.3): six rules — blank, `#` comment, `//` doc comment, `.!`/`.?` declaration, `=`-split content, metadata annotation. Every file type is `caiv` plus an interpretation of the two sides of `=`. Not a file extension. Pronounced "cave."
table definition	A schema-level declaration for an array of namespaces with collection-level constraints — uniqueness, referential integrity, cardinality (§4). Written `[/@name constraints…][]` in `.saiv`.
unique constraint	`field=!` declares a single-field unique key; `f1=!,f2=!` a compound key — distinct across all elements of the array.
foreign key reference	`field=/@path/*::field` — the fixed `fk-path` shape pinned in the formal grammar as the Level 2 subset of qaiv path syntax.
cardinality constraint	`min=N` / `max=M` bounds on an array's element count.
identity declaration	An authored `!str` annotation on a schema field: retained in the `.csaiv` as a nominal head, so only `str`-typed data lines satisfy the field — the opt-in "plain string, reject asserted types" contract. Distinct from an unannotated or anonymously refined field, which carries no head and accepts structural assertions; such a field's lex-saver, when one is needed, is the vacuous pattern `/^/`.
discriminated schema set	The one-of-N contract a parent schema declares for a delegated namespace (§3.5.3): the parent's ostensive block names the member schemas, the data's block annotation selects one, the canonical document carries the selection as a scoped `.! schema` declaration (the discriminant), and the selected member's compiled contract governs inside the namespace. Compiled form: the `[schema…::]` delegation line.

A.9 Provenance

Term	Definition
provenance	Source attribution metadata on data lines (§2.4): the `?sourceID@timestamp#dpid` form in the canonical metadata prefix — where a value came from, optionally when it was observed, and optionally a stable data-point identifier.
provenance source	A named external source — URI, sensor ID, URN, or any opaque string. Declared via `.?id uri` at file top; referenced via `?id` on value lines.
temporal attribution	The optional `@instant` qualifier recording when a value was observed — dashed extended ISO 8601 UTC, `YYYY-MM-DDTHH:MM:SSZ`, 20 characters. The compact 16-character basic form is accepted but deprecated, and is removed at 1.0.
data point identifier	The optional `#dpid` qualifier — any valid identifier, not restricted to UUIDs. The time-series platform auto-assigns UUID values at ingest.
`.?` declaration	A document-level declaration mapping a short provenance ID to a full URI (`.?id uri`). Placed at file top; preserved unchanged in `.daiv` and `.raiv`.

A.10 Higher-Level Concepts

Term	Definition
mapping (`.maiv`)	A structural correspondence declaration between two schemas (§8) — a pure namepath rewrite reusing `=` assignment and the `$` sigil. Enables conversion between any two schemas connected through the mapping graph.
hub schema	A `hub/` namespace schema on `ksaiv.com` serving as a structural archetype for the mapping graph — common data shapes (server endpoint, credentials, postal address) that many domain schemas map to. Curated.
`ktaiv.com`	The immutable production type-library registry (`.taiv`). Published content is never modified. Default base for type resolution; overridable per prefix via `.!registry`.
`ksaiv.com`	The immutable production schema registry (`.saiv`, `.csaiv`, `.maiv` — including the `hub/` namespace).
`kdaiv.com`	The immutable production document store (`.kaiv`, `.raiv`, `.daiv`). See `ECOSYSTEM.md` (in the spec repository).
`kfaiv.com`	The unit-definition registry (`.faiv`), paralleling `ktaiv.com` for types. Built-in units ship embedded and need no lookup.
mapping graph	The graph of registered schemas connected by `.maiv` declarations. Enables transitive conversion: if A and B each map to hub H, data converts from A to B through H.

A.11 Level System

Levels are **conformance profiles** — certification targets for environments with different safety and resource constraints.

Level	Name	Key features
Level 0	Scalars	`key=value` scalars, declarations, variables, type annotations, provenance. Minimal DFA, constant memory.
Level 1	Trees	`@` arrays, `/` namespace paths, `::` field projection, block syntax, the `+=`/`;=`/`:=`/`+:=` operators. Still constant memory.
Level 2	Tables	Table definitions: unique, foreign-key, and cardinality constraints. Post-scan validation pass; O(N) memory for uniqueness sets.
Level 3	Collation	`..lex[locale]` — BCP 47 locale tag on lexicographic ordering. Requires a collation library (ICU/CLDR). Platform-dependent.
Level 4	Corpus-Dependent	Reserved here (§6); specified separately in the experimental corpus specification. Operations over a corpus of `.daiv` files; executed by `kaiv db`-class tooling, never by the certified runtime.

Levels 3 and 4 are independent extensions of Level 2 — neither subsumes the other. An implementation may support either, both, or neither on top of Levels 0–2.

A.12 Implementation Architecture

Term	Definition
DFA	Deterministic Finite Automaton — the Lexer architecture. kaiv's regular grammar means the Lexer needs no stack, recursion, or backtracking. Each line classifies in O(1); content lines split on the first bare `'` (located via SIMD `memchr`).
Lexer	Layer 1 of the three-layer architecture. Reads raw characters; classifies each line per the six rules. The component that receives safety certification.
Parser	Layer 2. Consumes Lexer tokens; comprises the Compiler (`.kaiv` → `.raiv`), the Denormalizer (`.raiv` → `.daiv`, with materialization and the head-type lift), and the Validator (`.daiv` + `.csaiv` → pass/fail).
Application	Layer 3. Consumes the validated kaiv document — only fully-validated, typed, namepath-addressed values.

Term	Definition
kaiv document	The validated abstract tree produced by the pipeline; the root namespace with typed fields, resolved namepaths, and checked constraints. Serializable as `.daiv` (all references resolved) or `.raiv` (field references preserved).

B Implementation Notes: Certification and Performance

> This appendix is **non-normative**. It records the implementation architecture the format was designed around — where the pipeline stages run, what gets safety-certified, and why the text format carries no performance penalty — absorbed from the specification repository's design-era `ARCHITECTURE.md`, which is now frozen as a historical record. Nothing here adds conformance requirements; the normative surface is the body of this specification and the conformance suite.

B.1 The Build/Runtime Split

The pipeline stages themselves are specified normatively in the body (§1.2, §7); what this section records is *where they run*. The pipeline is deliberately split in two:

- **Build side** (developer workstation / CI): the schema compiler (`.saiv` → `.csaiv`), the Compiler (`.kaiv` → `.raiv`), the Denormalizer (`.raiv` → `.daiv`; schema-aware — `$field` expansion, materialization, the head-type lift), and the Validator (`.daiv` + `.csaiv` → pass/fail). These stages use a block stack, a variable table, a field table, and prefix expansion — mechanisms appropriate for a build tool, not for a safety-certified runtime. They never execute on the target.
- **Runtime side** (target / production — certified): the Lexer and the integrity check — a re-run of the Validator's parallel scan logic (§7.2) over an existing `.daiv` against its `.csaiv`. Line-by-line, no stack, no dynamic allocation, constant memory. The integrity check is not a separate pipeline stage: the build pipeline ends at `.daiv`.

Everything in this pipeline is either a DFA or something simpler than one. The Lexer is a DFA over bytes; the schema compiler is a canonicalizer that emits canonical text; the parallel scan is two file pointers advancing in lockstep — simpler than a DFA walk. The `.daiv` file and the compiled schema (`.csaiv`) are the only two artifacts the certified runtime needs.

B.2 The Certification Boundary

The `.daiv` format defines the certification boundary: everything to its left in the pipeline is a build tool, everything to its right is the certified runtime.

Component	Runs where	Certified?	Why
Compiler	Workstation / CI	No — validated through testing	Stack, variable table, field table, prefix expansion — context-free mechanisms unsuited to safety certification
Denormalizer	Workstation / CI	No — validated through testing	`$field` expansion; schema-aware materialization and head-type lift
Validator	Workstation / CI	No — validated through testing	Full schema constraint checking at build time
`.raiv` file	Build artifact	—	Relational intermediate for design tools, diffing, auditing, converters; never consumed by the certified runtime
`.daiv` file	Transmitted / stored	—	The interface contract between build and runtime; the certification boundary itself
Lexer	Target / production	Yes	DFA over bytes; regular grammar; constant memory
Integrity check	Target / production	Yes	The parallel text scan against `.csaiv`; no stack, no dynamic allocation, constant memory
`.csaiv` file	Loaded at startup	Build artifact	Canonical schema text, consumed read-only by the integrity check

This is a standard pattern in safety-critical systems: automotive ECUs routinely run certified firmware that consumes configuration generated by uncertified build tools. The build tool is validated; the runtime is certified; the interface format is specified. A certified deployment omits the Compiler, Denormalizer, and Validator entirely — the target never sees authoring sugar, variable interpolation, or namespace blocks, just flat, typed, fully-qualified lines it can validate in constant memory.

Levels describe **runtime capability**, not authoring syntax. A Level 0 certified runtime processes canonical Level 0 lines; a Level 1 runtime processes canonical Level 1 lines. The authoring syntax that produced those lines is irrelevant to the runtime.

B.3 Performance Model

kaiv is, to our knowledge, the only schema-driven data format whose entire runtime validation path — raw bytes to typed, validated values — is implementable with no heap allocation and static memory bounds. This follows from three design choices: a **regular lexical grammar** (the Lexer is a DFA — no stack, lookahead, or backtracking), **ordered keys** (a single-pass parallel scan against the compiled schema), and a **flat lexical grammar** (nesting lives in the namepath, bounded by the schema — no syntactic recursion). Remove any one of the three and the property is lost.

The same choices close the conventional binary-vs-text performance gap. That gap has three sources:

Gap source	JSON / YAML / XML	kaiv (`.daiv`)
Parsing complexity (stack, lookahead, backtracking)	Large — recursive descent with high constant factors	**Zero** — the DFA is O(N) with minimal constant factors, the same work per byte as binary tag dispatch
Value conversion (`atoi`, `atof`)	Medium — every value converts	**Bounded** — per-value, not per-byte; SIMD-parallelizable; dominated by the constraint validation that must run anyway
Compactness (bytes on wire / disk)	Medium — delimiter overhead	**Accepted** — the namepath is the cost of self-containment; small for string-heavy data; compression-mitigable

Three properties eliminate the first — dominant — gap entirely:

- **DFA scanning is branch-free in the hot path.** One table lookup per byte (`next_state = table[state][byte]`) — the same operation as reading a binary tag byte and dispatching on it. The transition table is tiny and lives in L1 cache; a multi-byte varint decode costs more ALU work per field than the DFA costs per byte.
- **The `'` and `=` splits are `memchr` calls.** The two core parsing operations on a canonical line are each a single SIMD-accelerated `memchr` — 16–32 bytes per cycle on x86-64. A typical line parses completely in two SIMD comparisons.
- **The parallel scan has no schema-lookup cost.** Binary formats save per-value parsing but pay a per-field descriptor lookup (hash map or binary search). The `.daiv` + `.csaiv` scan has none: the "schema lookup" is reading the next line — two sequential streams advancing in lockstep, the most prefetcher-friendly access pattern possible.

The remaining two gaps are real but bounded. Value conversion is per-value, SIMD-parallelizable (the simdjson techniques apply), and in certified deployments dominated by the pattern-constraint validation that runs regardless. Compactness favors binary for small numerics (a small-integer line is several times larger than its varint), but the namepath overhead is precisely what makes every line independently intelligible to `cat`, `grep`, and `diff`; for string-heavy payloads the ratio approaches parity, namepath repetition compresses at 5–10× under `gzip`/`zstd`, and the primary deployment model is block-aligned file I/O where small size ratios do not matter.

kaiv's text format is not human-readable at the cost of performance; it is human-readable at virtually no performance cost, because the regular grammar removes the parsing overhead that makes other text formats slow.

References

The standards cited in this document:

— **RFC 2046** — Freed, N., and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, November 1996.
— **RFC 2119** — Bradner, S., *Key words for use in RFCs to Indicate Requirement Levels*, BCP 14, March 1997.
— **RFC 3339** — Klyne, G., and C. Newman, *Date and Time on the Internet: Timestamps*, July 2002.
— **RFC 3986** — Berners-Lee, T., Fielding, R., and L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, STD 66, January 2005.
— **RFC 4648** — Josefsson, S., *The Base16, Base32, and Base64 Data Encodings*, October 2006.
— **RFC 5234** — Crocker, D., and P. Overell, *Augmented BNF for Syntax Specifications: ABNF*, STD 68, January 2008.
— **RFC 5321** — Klensin, J., *Simple Mail Transfer Protocol*, October 2008.
— **RFC 6838** — Freed, N., Klensin, J., and T. Hansen, *Media Type Specifications and Registration Procedures*, BCP 13, January 2013.
— **RFC 7303** — Thompson, H., and C. Lilley, *XML Media Types*, July 2014.
— **RFC 8174** — Leiba, B., *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*, BCP 14, May 2017.
— **BCP 47** — Phillips, A., and M. Davis (eds.), *Tags for Identifying Languages*, RFC 5646, September 2009.
— **UTS #10** — Whistler, K., and M. Davis, *Unicode Collation Algorithm*, Unicode Technical Standard #10.
— **CLDR** — *Unicode Common Locale Data Repository*, Unicode Consortium (reference version: CLDR 48, §5.3).
— **ISO 4217** — *Codes for the representation of currencies*, International Organization for Standardization.
— **ISO 8601** — *Date and time — Representations for information interchange*, International Organization for Standardization.

Index