kaiv — Specification

kaiv.io

version 1.0.0-alpha.5

Contents

1 Foundations **6**
- 1.1 Scope . . . 6
- 1.2 The Data Model . . . 7
- 1.3 The Universal Line Grammar (`caiv`) . . . 7
  - 1.3.1 The Six Rules . . . 8
  - 1.3.2 Classifier Pseudocode . . . 9
  - 1.3.3 How Each File Type Interprets the Left Side and Right Side . . . 10
  - 1.3.4 Metadata Annotations (Rule 6) . . . 10
- 1.4 The Sigil System . . . 12
  - 1.4.1 Sigil Survival Rule . . . 13
  - 1.4.2 The Two Path Operators . . . 14
  - 1.4.3 Authoring vs. Canonical Sigil Resolution . . . 14
  - 1.4.4 $ Is Additive (Prepend, Not Replace) . . . 15
- 1.5 Ordered Keys . . . 16

2 Level 0: Scalars **16**
- 2.1 Declarations . . . 17
  - 2.1.1 Declaration Inventory . . . 17
  - 2.1.2 Format Declaration . . . 18
  - 2.1.3 Schema Declaration . . . 18
  - 2.1.4 Type Registry Resolution . . . 19
- 2.2 Comments . . . 22
  - 2.2.1 General Comments (#) . . . 22
  - 2.2.2 Documentation Comments (//) . . . 22
- 2.3 Quoted Names . . . 23
  - 2.3.1 What Gets Quoted . . . 23
  - 2.3.2 When to Quote . . . 24
  - 2.3.3 Quoted Name Rules . . . 24
  - 2.3.4 Examples in Canonical Form . . . 24
  - 2.3.5 DFA Impact . . . 24
- 2.4 Provenance . . . 24
  - 2.4.1 Provenance Syntax . . . 25
  - 2.4.2 Disambiguation from Length Constraints . . . 25
  - 2.4.3 Requiring Provenance in Schemas . . . 25

2.5 Variables and Field References . . . 26
2.5.1 The Variable Name System . . . 26
2.5.2 Namespace-Variable Splat . . . 27
2.5.3 Field References . . . 27
2.5.4 Resolution Rules . . . 28
2.5.5 The “Almost Verbatim” Principle . . . 28
2.5.6 Worked Example — Variables . . . 29
2.5.7 Worked Example — Field References . . . 30
2.5.8 Architectural Impact . . . 31
2.5.9 Why `.raiv` Exists . . . 31
2.6 Type System . . . 32
2.6.1 The Single Primitive: `str` . . . 32
2.6.2 Unannotated Scalars Canonicalize to `!str` . . . 32
2.6.3 The `std/core` Standard Library . . . 33
2.6.4 The `std/enc` Encoding Library . . . 34
2.6.5 The `std/time` Time Library . . . 34
2.6.6 The `std/num` Numeric Markers Library . . . 35
2.6.7 The Constraint Triple . . . 35
2.6.8 Span Orderings . . . 36
2.6.9 Named Types . . . 37
2.6.10 Tagged Unions (`oneOf`) . . . 43
2.6.11 Schema Composition (`allOf`) . . . 43
2.6.12 `anyOf` . . . 43
2.6.13 Null Semantics . . . 44
2.6.14 Map Type . . . 46
2.7 Units . . . 48
2.7.1 Metadata Prefix Order . . . 48
2.7.2 Validation: Units Do Not Convert . . . 48
2.7.3 Compound Units . . . 49
2.7.4 Canonical Form: ASCII-Sorted Factors . . . 49
2.7.5 Built-in Units . . . 50
2.7.6 Custom units (`kfaiv.com`) . . . 53
2.7.7 Unit Definition Files (`.faiv`) . . . 53
2.7.8 Referencing Custom Units: `.!units` . . . 54
2.7.9 Units on Named Types . . . 55
2.7.10 Currencies . . . 55
2.8 Constraints . . . 56
2.8.1 Inline Constraints on Type Annotations . . . 56
2.8.2 Length Constraints . . . 56

3 Level 1: Trees **58**
3.1 Arrays . . . 58
3.1.1 Arrays Are Namespaces with Integer Fields . . . 58
3.1.2 Section Block Semantics . . . 59
3.1.3 Mixed Arrays . . . 60
3.1.4 Nesting Depth . . . 61
3.2 Structs . . . 61

3.3 Fields and Literals . . . 62
3.4 Namespaces . . . 62
3.4.1 Namespaced Structs . . . 62
3.4.2 Namespaced Field Keys . . . 63
3.5 Namespace Blocks . . . 63
3.5.1 Syntax . . . 63
3.5.2 Basic Example . . . 64
3.5.3 Namespace-Scoped Schemas . . . 64
3.5.4 Nested Namespace Blocks . . . 65
3.5.5 Canonical Form . . . 66
3.5.6 Encapsulated Hub Schema Extension (`.!schema:/ns hub/x`) . . . 66
3.5.7 Array-Element Hub Schema Extension (`.!schema:/@arr hub/x`) 67
3.5.8 Schema Composition for Namespace Blocks . . . 68

4 Level 2: Tables **68**
4.1 The Gap: No Collection-Level Constraints . . . 68
4.2 Table Declaration Syntax . . . 69
4.3 Unique Constraints . . . 70
4.4 Foreign Key References . . . 71
4.5 Cardinality Constraints . . . 71
4.6 Compiled Form . . . 72
4.7 Validation . . . 73
4.8 Why This Is Level 2 . . . 74
4.9 Architectural Impact . . . 75
4.10 SQLite Comparison . . . 76

5 Level 3: Collation **77**
5.1 The Problem: `..lex` Is Byte Order . . . 77
5.2 Syntax: `..lex[locale]` . . . 78
5.3 Reference Collation: CLDR Version and Strength . . . 78
5.4 In Type Definitions (`.taiv`) . . . 79
5.5 In Compiled Schema (`.csaiv`) . . . 79
5.6 Certification Impact . . . 79
5.7 Query Impact . . . 80
5.8 Architectural Impact . . . 80

6 Level 4: Corpus-Dependent Features **81**
6.1 What “Corpus-Dependent” Means . . . 81
6.1.1 Updated Level System Summary . . . 82
6.2 Path Identity and the Corpus Tree . . . 82
6.3 The Metaschema . . . 83
6.3.1 The `.msaiv` Surface . . . 84
6.4 Schema Composition: `.!compose` . . . 86
6.4.1 Syntax . . . 86
6.4.2 Semantics . . . 86
6.4.3 Composed Schemas . . . 87
6.5 Cross-Schema Foreign Keys: `.!ref` and `$alias.field` . . . 87

6.5.1 The `.!ref` Declaration . . . 87
6.5.2 The `$alias.field` Value Expression . . . 88
6.5.3 Complete Example . . . 88
6.6 Level 2 vs. Level 4 — Intra-Document vs. Cross-Document FK . . . 88
6.7 Validator + Cross-Schema FK Check . . . 89
6.8 RESTRICT-Only Deletion . . . 90
6.9 Architectural Impact . . . 90
6.10 Compose vs. Cross-Schema FK — When to Use Which . . . 92
6.11 Open Design Questions . . . 92

7 Compiled Schema (`.csaiv`) 93
7.1 Parallel Scan Validation . . . 93
7.2 Validator Pseudocode . . . 95
7.3 The Schema Compiler . . . 96
7.4 Table Declarations in the Compiled Schema . . . 97
7.5 Maps in the Compiled Schema . . . 99

8 Mappings (`.maiv`) 100
8.1 Header Declarations . . . 100
8.2 Mapping Lines . . . 100
8.3 Execution Model . . . 101
8.4 Publish-Time Validation . . . 102
8.5 Composition . . . 102
8.6 Auto-Derived Mappings . . . 103

9 Parsing Requirements 103
9.1 Line Numbers . . . 103
9.2 UTF-8 Processing . . . 103
9.2.1 BOM Handling . . . 103
9.2.2 Forbidden Characters . . . 103
9.3 EOL . . . 104
9.4 Whitespace Handling . . . 104
9.5 Value Preservation . . . 104
9.5.1 Empty Values . . . 105
9.6 Empty Documents . . . 105
9.7 Parsing Models . . . 105
9.8 Duplicate Keys . . . 105
9.9 Implementation Limits . . . 105

10 Formal Grammar (Levels 0–1) 106
10.1 Common Productions . . . 106
10.2 Line Classification . . . 107
10.3 Declaration Lines (Rule 4) . . . 107
10.4 Type References, Constraints, Units, Provenance . . . 108
10.5 Metadata Annotation Lines (Rule 6 — Authored Files Only) . . . 111
10.6 Content Lines (Rule 5) . . . 111
10.7 Authored Structure Lines (Blocks — `.kaiv` / `.saiv`) . . . 114

10.8 Values 114

11 Errors **115**
11.1 Lexer Errors 115
11.2 Application Errors 117

12 File Representation **120**
12.1 File Extensions 120
12.2 File Encoding 121
12.3 Media Types 121
12.3.1 Provisional Types 121
12.3.2 Parameters 121
12.3.3 Transport Considerations for `text/kaiv` 122
12.4 Shebang Lines 122

Terminology reference. All kaiv terms used in this document (namespace, namepath, field address, canonical form, Compiler, Denormalizer, Validator, integrity check, etc.) are defined in `TERMINOLOGY.md` (in the spec repository), which is 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 `TERMINOLOGY.md` and `ARCHITECTURE.md` §6 (in the spec repository). 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 (§2.7.5), 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 (§2.6.13). Implementation-side detail (Lexer token table, Compiler/Denormalizer/Validator steps, performance model) lives in `ARCHITECTURE.md`. Level 4 drafts the corpus model — path identity and the metaschema, with the authored `.msaiv` surface drafted (§6.3.1) and its compiled form and frozen grammar still open (§6.11). The unit-definition file format (`.faiv`, §2.7.7) and the `.!units` import are populated; constrained-union lowering is specified (§2.6.10).

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) — 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.

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, and materializes absent optional fields (§2.6.9) 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` (§6.3.1) — 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
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 — 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 path followed by uniqueness, cardinality, and foreign-key clauses: /@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.

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 '('   → structure line   # authored .kaiv/.saiv
   only:
                                                    # block open/close,
   incl. table
                                                    # header; BEFORE the
   '=' test
    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`), 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 (§2.6.9)
`.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
`?`	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, §2.6.9). 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 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	—	—	—	—	—
`$`	dereference operator. **Variables** (dot-prefixed): `$.name`, `$@.name`, `$/.name`. **Field references** (no `.` after `$`): `$field`, `$path::field`	—	—	—	—	—

Sigil	kaiv Construct	JSON	TOML	ProtoBuf	GraphQL	ASN.1
`!`	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
`$`	Dereference	[x] present	[ ] elided	`$.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
`?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`	`.kaiv`, `.raiv`, `.daiv`	§2.1.2
`.!kaivschema`	`.!kaivschema VERSION ID-OR-URL [strict]`	`.saiv`, `.csaiv`	§2.6.9; strict modifier in §11
`.!kaivtype`	`.!kaivtype VERSION LIBRARY-ID`	`.taiv`	§2.6.9
`.!kaivunit`	`.!kaivunit VERSION LIBRARY-ID`	`.faiv`	§2.7.7
`.!kaivmap`	`.!kaivmap VERSION MAP-ID`	`.maiv`	§8.1
`.!source`	`.!source ID-OR-URL`	`.maiv`	§8.1
`.!target`	`.!target ID-OR-URL`	`.maiv`	§8.1
`.!via`	`.!via MAP-ID`	`.maiv`	§8.5
`.!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`	§2.6.9
`.!units`	`.!units LIBRARY-ID`	`.kaiv`, `.saiv`, `.taiv`; survives into canonical output (§2.7.8)	§2.7.8
`.!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`	`.saiv` (Level 4)	§6.5
`.!compose`	`.!compose:NAMEPATH SCHEMA JOIN`	`.saiv` (Level 4)	§6.4

Keyword	Syntax	File types	Defined in
`.!kaivmetaschema`	`.!kaivmetaschema VERSION CORPUS-ID`	`.msaiv` (Level 4, draft)	§6.3.1
`.!bind`	`.!bind:PATTERN SCHEMA`	`.msaiv` (Level 4, draft)	§6.3.1
`.!unique`	`.!unique:PATTERN NAMEPATH`	`.msaiv` (Level 4, draft)	§6.3.1
`.!fk`	`.!fk:PATTERN NAMEPATH TARGET:: NAMEPATH`	`.msaiv` (Level 4, draft)	§6.3.1
`.?<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** uses the `.!kaiv` command followed by the kaiv text version.

The format declaration MUST be placed on the first line of the kaiv text — or, when an optional shebang line is present (§12.4), on the first line after it.

The version 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. The conventional authored form is the shortest, with trailing zero components omitted — which is why every example in this document writes `.!kaiv 1`. The same version syntax applies to the `.!kaivschema`, `.!kaivtype`, `.!kaivunit`, `.!kaivmap`, and `.!kaivmetaschema` format 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: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, but this can be customized using MIME type request headers.

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 `.!kaiv`, before content lines.

```
.!kaiv 1
.!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`.
- 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.

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 1

/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.

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.

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`:

```
# Both of these resolve to the same type identity:
.!registry acme=https://types.acme.com       # Layer 1 (fetches from
    acme.com)
# [no .!registry declaration]                  # Layer 4 (fetches from
    ktaiv.com)

# Both produce identical canonical output:
!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.

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

`#` 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. They never appear in `.daiv` or `.csaiv`.

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

2.2.2 Documentation Comments (`//`)

`//` doc comments are allowed **only in the definition-bearing file types** — `.saiv` schema files, `.taiv` type library files, `.faiv` unit definition files, `.maiv` mapping files, and (draft) `.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.

A `.kaiv` Lexer encountering `//` SHOULD emit an error (or MAY treat it as a `COMMENT`).

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. Authored text MAY quote a bare-able name — necessary when the bare spelling is positionally reserved, e.g. a schema field named `re` (§2.6.7) — 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 ('@' timestamp)? ('#' dpid)?
id              = identifier
timestamp       = YYYYMMDD 'T' HHmmSS 'Z'   (16 chars fixed)
dpid            = identifier
```

A single provenance entry carries three optional components:

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. `@timestamp` — the temporal qualifier (compact ISO 8601 UTC, `YYYYMMDDTHHmmSSZ`, always 16 characters) recording *when* the value was observed.
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.

All three components are optional and independent. 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:

```
.!kaivschema 1 hub/log-entry
.!provenance:required
```

Four 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:timestamp`	`@timestamp` required; `?sourceID` 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. 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 `$` dereference operator in values. They are resolved during Compiler canonicalization and completely elided from the canonical output. In addition to hidden variable references, kaiv supports **field references** that reference previously-defined data fields — distinguishable from variable references by the absence of `.` after `$`.

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

Behavior in Each Canonical Form

Construct	In `.raiv`	In `.daiv`
Variable definitions (`.name=`)	Elided	Elided
Variable references (`$.name`)	Resolved	Resolved

Construct	In `.raiv`	In `.daiv`
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.

2.5.6 Worked Example — Variables

Authored kaiv

```
.!kaiv 1
.!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`)

```
!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.

2.5.7 Worked Example — Field References

Authored kaiv

```
.!kaiv 1
.!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

```
!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

```
!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.

2.5.8 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 `$path::field` references from the field table; when the data declares a schema, it also reads the `.csaiv` and materializes absent optional fields (§2.6.9). Produces `.daiv` (field references resolved, every schema-declared field present).
- **Validator:** Reads `.daiv` (no field references remain) and `.csaiv`. Performs schema constraint checking. Produces pass/fail.
- **Compiled schema (`.csaiv`):** Unchanged. Variables and field references are invisible to the schema — they are resolved before validation.
- **Denormalized form (`.daiv`):** No dot-prefixed definitions, `$.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.

2.5.9 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` (§2.6.9); 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`

Property	Detail
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 on a canonical (`.daiv`/`.raiv`) line 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 canonicalize 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.

Schema-declared types are **not** stamped onto data lines either: a `.daiv` line keeps its authored type, and the schema's type/constraint for that field lives in the `.csaiv` and is checked against the data line's type and value by the Validator (§7). A field whose value must be an integer is enforced by the schema constraint at validation time, not by rewriting the data line's `!str` to `!int` during canonicalization.

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.

```
.!kaivtype 1 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 §5, unpadded)
/^[A-Za-z0-9_-]*$/
&b64=
```

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 §2.6.13.
`!b64`	`!b64`	`std/core` shorthand — stays as `!b64`
`&port`	`!std/net/port`	Resolved from `std/net.taiv` — library path `std/net`, type `port`

Written (authoring)	Canonical form	Note
`&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 `std/enc` Encoding Library

kaiv values are single-line, so multi-line or binary content 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:

```
.!kaivtype 1 std/enc

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

Members: `bin` (generic binary), `txt` (multi-line UTF-8 text), `json`, `yaml`, `toml`, `xml`, `html`, `md`, `csv`. Usage:

```
.!kaiv 1
.!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).

2.6.5 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:

```
.!kaivtype 1 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`). Because `..time` values in these shapes compare correctly as strings within a fixed offset, range constraints (`[2026-01-01,2026-12-31]`) work at every Level; cross-offset chronological comparison is consumer tooling's concern.

2.6.6 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:

```
.!kaivtype 1 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`).

2.6.7 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.

2.6.8 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`	ISO 8601 chronological ordering.
`..ver`	Semantic version ordering.

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 46, tertiary strength; §5.3)
`..time`	ISO 8601 chronological	Date parse + comparison
`..ver`	Semantic version	Semver parse + comparison

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.

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.

2.6.9 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 `.!kaivtype`, paralleling `.!kaivschema` for `.saiv` files. The version number (`1`) and the library identifier (`std/net`) follow the same convention as other format declarations.

Example — `std/net.taiv`

```
.!kaivtype 1 std/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)

Position	File type	Syntax	Meaning
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` = canonical `std/core` forms (not expanded further). `!library/path/name` = fully-qualified for all other types.
`&`	`.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.

```
.!kaivschema 1 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`, §2.6.13) 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`:

```
.!kaivschema 1 https://example.org/server-config.saiv
.!types std/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 (`std/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 1
.!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 schema's declared type for consistency.

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:

```
.!kaiv 1
.!schema:acme/server-config

!std/time/datetime'::created_at=2025-01-15T09:30:00Z
!std/net/ipv4'::host=192.168.1.1
!std/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.

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.

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.

2.6.10 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 §2.6.13 for the complete null model.

Maps to:

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

2.6.11 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`

2.6.12 `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.

2.6.13 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 (§2.6.10).

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

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

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

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`). The data line in `.daiv` carries only the **active variant** — the concrete type of the actual value:

```
# 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 (§2.6.9):

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 (§2.6.9)	`!type'::field=default_value`
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`, §2.6.9.)

Two consequences of working from the *compiled* schema: the `!type` token of a materialized line is the `.csaiv` field's retained type item when one exists (`!str`, `!type:unit`, or the matching union alternative) and `!str` otherwise — for a field lowered to a bare value constraint the type name is no longer in the artifact, and the parallel scan checks such fields by constraint, not by name (§7.1). 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 (§2.6.9), 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 (§2.6.9): `!str'::field=`
`!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`

2.6.14 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.

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@20250115T093000Z#p-17'/trip::length=42.5
!str/[a-zA-Z0-9.-]+/#[1,253]?dns1@20250115T093000Z#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 to the `'` delimiter as a single unit token, with no context-sensitive parsing.

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` (§2.7.8).
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).

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 is the responsibility of consumer tooling — queries, post-processing, or format converters — typically operating outside 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 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`).

2.7.4 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.

2.7.5 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 compile time (a unit name that is neither built-in nor resolvable on `kfaiv.com` 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`
`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`
`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)

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 consumer tooling that elects to convert between same-dimension values.

2.7.6 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.

2.7.7 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 `.!kaivunit VERSION LIBRARY-ID` header, then a **definition line** above each `&name=`:

```
.!kaivunit 1 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, §2.7.10).

— **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}` (a value's `?@timestamp` provenance, `YYYYMMDDTHHmmSSZ`), 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=`.

```
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
kaivunit-decl  = ".!kaivunit" 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.

2.7.8 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 1
.!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.

2.7.9 Units on Named Types

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

```
.!kaivtype 1 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
```

Unit narrowing rules mirror the rest of the constraint-narrowing model:

- A named-type unit may be narrowed to a different unit of the **same dimension** (e.g. `:km` narrowed to `:m`). The canonical form records the narrowed unit, and the value is interpreted in the narrowed unit's scale.
- Narrowing across dimensions is a compile-time error (e.g. `:km` cannot be narrowed to `:s`).
- A named type without a unit can be annotated with one in a schema or data line that uses it; the converse — stripping a unit at use site — is not permitted.

2.7.10 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 value's `?@timestamp` 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}`, §2.7.7), so that consumer tooling can pull the exchange rate contemporaneous with a value's `?@timestamp` 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):

```
!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**.

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 1
.!schema:acme/server-config

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

Canonical Output

```
.!kaiv 1
.!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 schema's compiled DFA in addition to the parent document's schema. 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 1
.!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 set of schemas — analogous to how a tagged union declares a set of type alternatives. The block's `schema:` annotation selects which sub-DFA validates its contents. Ordered keys ensure that `algorithm` is resolved before the `(/parameters …)` block is entered, making the discriminant available at parse time.

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 **consumed during validation and does not appear in the canonical output**. The schema reference lives in the parent schema's compiled DFA, not in the data stream.

For tooling that must preserve schema-selection provenance through the canonical form, the annotation may optionally be emitted as a scoped declaration line:

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

This is a scoped `.!schema` declaration — "schema `crypto/rsa-params` applies to namespace `parameters`" — using the standard namespace-qualified form of the `.!schema` declaration (`schema-registry-ref` in §10). Whether to emit this line is a tooling decision; the runtime validator does not require it.

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, not a local validation delegation.

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 (`.!kaivschema`, `.!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:

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

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 schema line corresponding to the namespace block field.
2. It suspends the parent schema scan and delegates to the sub-schema (`.csaiv`) identified by the `schema:` annotation.
3. The sub-schema scan validates the block's contents.
4. When `()` is encountered, control returns to the parent schema scan at the next line after the namespace field.

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 `ARCHITECTURE.md` §7.4 (in the spec repository) 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=
/^-?[0-9]+$/ ..num [1,65535]'/@servers/::port=
/^-?[0-9]+$/ ..num [1,3600]'/@servers/::timeout?=
```

The collection constraint line `/@servers [unique::id]|[unique::host,port] [min=1] [max=50]` is recognized by its structure: an `@`-prefixed name followed by bracketed constraint clauses, with no `'` delimiter and no `::` field projection. 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 (implementation detail for the pipeline stages is in `ARCHITECTURE.md` §7.2, in the spec repository):

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. An element that omits an optional constrained field participates in Pass 2 with its empty value — a kaiv value is never absent, only empty (§2.6.9). Two elements that both omit a unique-constrained field therefore collide, and an omitted foreign-key field references the empty 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 — recognized by `@`-prefixed name + bracket clauses with no `'` delimiter and no `::`. 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 46** root collation and locale tailorings (the Unicode Collation Algorithm, UTS #10). An implementation SHOULD use CLDR 46; 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 46. 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 46.

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.

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:

```
.!kaivtype 1 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:
/^.{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 44 may sort differently under CLDR 46. This spec pins **CLDR 46 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

> **Status: design draft.** Level 4 collects operations that read or query a *corpus* of `.daiv` files rather than a single document. The corpus model is now drafted: document identity is *path identity* (§6.2), and the contract over a corpus is a *metaschema* (§6.3) — the same specification move that takes a type to a schema, taken once more. The two original features — schema composition (`.!compose`) and cross-schema foreign keys (`.!ref` / `$alias.field`) — relocate into the metaschema; their subsections below retain the surface syntax. The metaschema's own surface syntax is not yet frozen; implementations should treat this Level as a draft. See Open Design Questions (§6.11) at the end of this chapter.

Levels 0–3 share a self-containment property: validating or compiling a `.daiv` requires only that `.daiv` plus its `.csaiv`. Level 4 features break self-containment by definition — they read, enumerate, or query other `.daiv` files. The certified runtime never executes Level 4 operations; they live in `kaiv db`-class tooling, distinct from the parallel-scan Validator.

Two features currently inhabit this Level:

— **Schema composition** (`.!compose`) — at compose time, gather records from a related schema and embed them as an intra-document array. Materializes a JOIN at publish time; the output `.daiv` is self-contained and validates under Level 2.

— **Cross-schema foreign keys** (`.!ref` / `$alias.field`) — declare that a field's value must exist as a value in another schema's dataset. Validation requires querying that dataset; checked by `kaiv db`-class tooling, not by the certified Validator.

Both share the same hardness profile: executing them requires access to a corpus of `.daiv` files (a directory, a `kdaiv.com` namespace, a `dbaiv.com` snapshot, etc.) that the kaiv format does not currently abstract over.

See `dbaiv/DESIGN.md` §12 (in the spec repository) for the canonical implementation in `kaiv db`: the compose pipeline, the `ref_values` table, `kaiv db validate`'s cross-schema FK check, and deletion behavior.

6.1 What "Corpus-Dependent" Means

A *corpus* is an enumerable collection of `.daiv` files. The kaiv format defines named entities — a single `.daiv`, a single `.saiv`, a single `.taiv` — and registries to fetch them by name (`ktaiv.com`, `ksaiv.com`, `kdaiv.com`). It does **not** define a queryable collection abstraction: there is no format-level mechanism to ask "give me all `.daiv` files conforming to schema X." A corpus is exactly that abstraction.

A corpus-dependent feature is one whose semantics require enumerating or querying such a collection. Concrete examples:

— Compose enumerates all records of a related schema and filters by a join key.

— Cross-schema FK validation looks up whether a given value exists as a field value in any record of the target schema.

The corpus abstraction is pinned by two ideas: *path identity* — every document in a corpus is addressed by a path, under the same `/` navigation semantics the namepath grammar

already defines — and the *metaschema*, the declared contract over the corpus (both below). Backends differ only in what serves the paths:

— `kaiv db` (see `dbaiv/DESIGN.md` §12, in the spec repository) maintains a local SQLite index (`.kaiv.db`) over the staged `.daiv` files; that index is the corpus.
— A simpler tool might walk a directory of `.daiv` files and read each one's `.!schema` declaration to determine membership.
— A future tool might query a `kdaiv.com` namespace prefix or a `dbaiv.com` snapshot URL.

6.1.1 Updated Level System Summary

Level	Name	Key Features	Memory	Platform
Level 0	Scalars	`key=value` scalars, declarations, variables, type annotations, provenance	Constant	Any
Level 1	Trees	`@` arrays, `/` namespace paths, `::` field projection, block syntax, operators	Constant	Any
Level 2	Tables	UNIQUE, intra-document FOREIGN KEY, cardinality constraints	O(N)	Any
Level 3	Collation	`..lex[locale]` — BCP 47 locale-aware ordering, ICU/CLDR dependency	Constant	Any (with collation library)
Level 4	Corpus-dependent	path identity, metaschema (schema bindings, corpus collections), `.!compose`, `.!ref/$alias.field`	External I/O over a corpus of `.daiv` files	`kaiv db`-class tooling

Level 4 is not a superset of Level 3. Levels 3 and 4 are independent extensions of Level 2. A Level 4 implementation need not support collation; a Level 3 implementation need not support corpus-dependent features. Both extend Level 2 separately.

6.2 Path Identity and the Corpus Tree

Within a corpus, a document's identity **is its path**. The corpus is a tree navigated with the same `/` descent semantics as a document's namespaces: corpus root, path segments, document, namespaces, terminal `::field` projection — one continuous backbone under one grammar. A reference such as

```
/configs/prod/server.daiv/network::host
```

descends through the corpus into a document and onward to a leaf without switching navigation models. The document boundary is *transparent to navigation* but *load-bearing for specification*: schemas govern what is inside one document, the metaschema (next section) governs the corpus around it.

What serves the paths is a backend choice — a filesystem directory, an object-store bucket, an HTTP host. The **canonical corpus is the immutable remote store** (`kdaiv.com` -class): only a store that never rewrites a published path can *guarantee* the three properties corpus-level semantics lean on — immutability, persistence, and referential integrity ("the referenced document exists" as an invariant, not a hope). A local directory is the development approximation of that model, with the same fragility relative to it that a local type directory has relative to a registry. Reproducibility follows the same line: a Level 4 run is reproducible exactly when the corpus handle is immutable (a `kdaiv.com` namespace, a `dbaiv.com` snapshot); over a mutable directory, reproducibility is explicitly not promised.

6.3 The Metaschema

> **Status: concept pinned, authored surface drafted.** This section fixes what a metaschema *is* and what it declares, and §6.3.1 drafts the authored `.msaiv` surface. Its compiled form and the frozen grammar remain future work (§6.11).

A schema cannot be stretched across the file boundary without tearing — a document schema that declares cross-document edges is reaching outside its own realm. The resolution is not to stretch it but to climb: kaiv's specification ladder repeats one move, and the corpus needs the next rung.

- A **type** constrains one *value*.
- A **schema** constrains one *document*: fields at namepaths, typed; tables with uniqueness, foreign keys, and cardinality over *elements*.
- A **metaschema** constrains one *corpus*: documents at paths, schema-bound; collections with uniqueness, foreign keys, and cardinality over *documents*.

The correspondence is mechanical:

Document realm (schema)	Corpus realm (metaschema)
namepath	path
field	document
`!type` annotation on a field	schema binding on a document
table declaration over elements	collection declaration over documents
Validator parallel scan	corpus scan

The realm split. Each realm is closed: a schema validates a document knowing nothing of the corpus; a metaschema validates a corpus by *delegating* every within-document question to the bound schemas. Document validation therefore stays at Levels 0–3 with its certification profile untouched; everything the metaschema adds is Level 4, in the application layer.

A metaschema declares four kinds of clause:

1. **Schema bindings** — path pattern → schema ID: every document matching the pattern must validate against the schema (e.g. "everything under `/customers/` conforms to `acme/customer`").
2. **Corpus collections** — uniqueness, foreign-key, and cardinality clauses over the documents matching a pattern: Level 2's table machinery lifted one rung (hash sets built over document fields instead of element fields).
3. **Reference declarations** — the `.!ref` / `$alias.field` cross-document edges (§6.5), relocated here from the document schema.
4. **Compose recipes** — the `.!compose` declarations (§6.4), likewise relocated.

Relocation. Earlier drafts of this Level placed `.!ref` and `.!compose` in the `.saiv` header; that placement is the circle this section squares, and it is now deprecated. The subsections below keep the original syntax — unchanged in shape — but the declarations' home is the metaschema. Registry-wise, metaschemas live beside schemas (`ksaiv.com`-class service).

The corpus as an arbor. A corpus under a metaschema is a tree-spanned graph: the path/namespace backbone plus the declared reference edges as crosslinks. The reference declarations are deliberately conservative — key equality only, mirroring Level 2's `fk-path` — because kaiv's job ends at declaring and validating the graph; *querying* it belongs to a query language over arbors (the direction explored in the companion qaiv design, converging with the Quarb language). To that end, kaiv pins a compatibility rule: **every path-pattern surface this Level adds must remain a strict subset of Quarb path syntax**, exactly as `fk-path` is already pinned as a frozen subset — so the declared corpus is queryable by an identity-map adapter, with canonical namepaths valid as query paths verbatim.

6.3.1 The `.msaiv` Surface

> The authored surface below is a *draft proposal*. It fixes the clause spellings for the two metaschema declarations that lack one — schema bindings and corpus collections — and reuses the `.!ref` and `.!compose` surfaces (§6.5, §6.4) unchanged. The compiled `.msaiv` form and the frozen grammar remain open (§6.11).

A metaschema is authored as an `.msaiv` document — a member of the `*aiv` family, sharing the one line grammar — and so opens with a version header naming the corpus it constrains:

```
.!kaivmetaschema 1 acme/shop
```

The corpus id (`acme/shop`) is the metaschema's registry identity, resolved beside schemas on the `ksaiv.com`-class service. Every path pattern in the clauses below is a strict subset of Quarb path syntax (the compatibility rule above), so a canonical namepath is a valid query path verbatim.

Schema bindings. A binding attaches a schema to every document matching a path pattern — the corpus-realm analogue of a `!type` annotation on a field:

```
.!bind:PATTERN SCHEMA
```

— `PATTERN` — a path pattern over the corpus (e.g. `/customers/*`); every matching document MUST validate against `SCHEMA`.
— `SCHEMA` — the registry path of the bound schema (e.g. `acme/customer`).

A document may match several bindings and MUST validate against each; a document matching none is unconstrained (the corpus does not require total coverage).

Corpus collections. Level 2's table machinery lifted from elements to documents: a constraint over the *set* of documents matching a pattern. The uniqueness and foreign-key forms mirror the Level 2 table header and the `fk-path` production, introducing no new value grammar:

```
.!unique:PATTERN NAMEPATH
.!fk:PATTERN NAMEPATH TARGET::NAMEPATH
```

— `.!unique` — the field at `NAMEPATH` is unique across all documents matching `PATTERN` (a hash set built over document fields, exactly as Level 2 builds one over element fields).
— `.!fk` — referential integrity between document sets: the value at `PATTERN::NAMEPATH` MUST occur as `TARGET::NAMEPATH` in some document matching `TARGET`. An optional `#[min,max]` suffix pins the edge cardinality, mirroring the Level 2 length constraint.

Both are checked by a *corpus scan* — the metaschema-level counterpart of the Validator's parallel scan, the one pass a corpus-blind document validation cannot perform.

References and compose. The remaining two clause kinds keep the surfaces they already have: reference declarations (`.!ref` / `$alias.field`, §6.5) and compose recipes (`.!compose`, §6.4), now authored in the `.msaiv` document rather than the `.saiv` header.

A complete metaschema. The four clause kinds together:

```
.!kaivmetaschema 1 acme/shop

.!bind:/customers/* acme/customer
.!bind:/orders/* acme/order

.!unique:/customers/* ::email
.!fk:/orders/* ::customer_id /customers/*::id

.!ref:customer /customers/*
.!compose:/@orders acme/order customer_id=$::id
```

This binds two document sets, pins email uniqueness and order-to-customer referential integrity across the corpus, imports `acme/customer` as a reference target, and records how an `acme/customer-with-orders` view materializes. How `kaiv db` *executes* queries and

composition over such a corpus — including the split between an edge scan near the store and a client continuation — is specified in `dbaiv/DESIGN.md` (in the spec repository).

6.4 Schema Composition: `.!compose`

`.!compose` declares that records of a related schema should be embedded into composed output documents at compose time. The declaration is a metaschema clause (§6.3); the composed `.daiv` is self-contained — all cross-file references are resolved into intra-document arrays — and validates under Level 2 (intra-document FK and uniqueness). Compose is not a query: it is *denormalization one rung up* — the Denormalizer resolves intra-document `$field` references to produce a self-contained `.daiv`; compose resolves inter-document references to produce a self-contained `.daiv` from a corpus. Same operation, next level of the ladder.

6.4.1 Syntax

```
.!compose:NAMEPATH SCHEMA JOIN_CONDITION
```

- `NAMEPATH` — the namepath array in the output document where related records are embedded
- `SCHEMA` — the source schema to pull records from (a registry path, e.g. `acme/order`)
- `JOIN_CONDITION` — the equality predicate determining membership, of the form `targetfield=$::ownfield`

Example:

```
.!compose:/@orders acme/order customer_id=$::id
```

This instructs a Level 4 processor to read every `.daiv` of schema `acme/order` from the corpus, group them by `::customer_id`, and for each customer record being composed, embed the matching orders as an array under `/@orders`.

6.4.2 Semantics

A processor with Level 4 support inserts a **Compose** stage between the format's Compiler and Denormalizer:

```
.kaiv → Compiler → .raiv → Compose → .raiv → Denormalizer → .daiv
                          ($field preserved)    (cross-file refs resolved)   (literal values)
```

The Compose stage:

1. Reads each `.!compose` declaration and identifies the source schema and join condition.
2. Enumerates the corpus (implementation-defined source) for `.daiv` files of the source schema.

3. Groups source records by the join key.
4. For each composing record, embeds the matching source records as an intra-document array at `NAMEPATH`.

The Denormalizer then expands any remaining `$field` references in the composed document and emits the final `.daiv`.

6.4.3 Composed Schemas

A composed schema has its own identity in the registry, distinct from its source schemas. For example, `acme/customer-with-orders` is a composed schema: its `.saiv` declares the embedded structure as ordinary Level 1/Level 2 shape, the metaschema's compose recipe (`.!compose:/@orders acme/order …`) says how to materialize it, and its `.daiv` artifacts are independently fetchable, independently validated, and content-addressed in their own right.

Cardinalities supported via composition:

Relationship	Mechanism
1:1	Direct schema composition — embed related fields directly. No `.!compose` needed.
1:many	`.!compose` — embed related records as an array in the parent. One composed file per parent entity.
Many:many	Chained `.!compose` — compose from a join schema, then denormalize fields from the related entity into each array element. Or produce primary-centric and secondary-centric views as separate composed schemas.

See `dbaiv/DESIGN.md` §12 (in the spec repository) for the canonical `kaiv db` implementation, including extended examples and the publish-time integration.

6.5 Cross-Schema Foreign Keys: `.!ref` and `$alias.field`

6.5.1 The `.!ref` Declaration

```
.!ref:alias schemapath
```

A metaschema clause (§6.3; earlier drafts placed it in the `.saiv` header — deprecated, see *Relocation* there). Imports a schema as a **reference target** — not extension, not inheritance. The `alias` is a short local name used in value position. `schemapath` is the registry path of the target schema (e.g. `acme/customer`).

Multiple `.!ref` declarations are permitted, one per target schema:

```
.!ref:customers acme/customer
.!ref:products  acme/product
```

6.5.2 The `$alias.field` Value Expression

```
!str'::customer_id=$customers.id
```

`$alias.field` in value position is a **cross-schema FK declaration**: this field's value must exist as a value of `field` in a dataset governed by the schema registered under `alias`. The value position uses `$` (the dereference sigil) followed by the alias, a `.`, and the target field name.

This extends the existing `$` reference system:

Form	Where	Meaning	Available at
`$field`	Value position in `.kaiv`/`.raiv`	Intra-document field reference (data level)	Level 0 (§2.5.3)
`$.name`	Value position	Hidden variable dereference	Level 0
`$alias.field`	Value position in `.saiv`	Cross-schema FK declaration (schema level)	**Level 4**

The dot in `$alias.field` distinguishes it from `$field` (plain field reference) and `$.name` (hidden variable). An alias never starts with `.` (that would collide with the variable namespace).

6.5.3 Complete Example

```
// acme/order.saiv
.!kaivschema 1 acme/order
.!ref:customers acme/customer
.!ref:products  acme/product

!str
id=
!str
customer_id=$customers.id    # FK: value must exist in acme/customer::id
!str
product_id=$products.sku     # FK: value must exist in acme/product::sku
!int
quantity=
!int
amount=
```

6.6 Level 2 vs. Level 4 — Intra-Document vs. Cross-Document FK

Level 2 intra-document FKs (`[/@orders customer_id=/@customers/*::id]`) are self-contained: they reference another array in the same file, validated in a post-scan pass, O(N) memory, no external I/O. They are certifiable and can run anywhere.

Level 4 cross-schema FKs reference a dataset governed by a different, separately-stored schema. Validation requires at minimum one index query per FK field per validation pass (a SQLite `SELECT` against the `ref_values` table in `.kaiv.db`, or equivalent in another Level 4 implementation). This is **external I/O over a corpus** — it cannot be performed by the certified DFA runtime, which is a constant-memory, self-contained computation with no network access.

Level	FK scope	Validation model	Where it runs
Level 2	Intra-document	Post-scan pass, O(N) memory, self-contained	Anywhere
Level 4	Cross-document	Corpus query per FK field, external I/O required	`kaiv db`-class tooling

Cross-schema FKs only make sense where a coordinated index of referenced values exists — for example, `kaiv db validate` with its `ref_values` table. On immutable wings (`kdaiv.com`, `ksaiv.com`, `chraiv.com`) there is no mutation coordination and no `ref_values` index; `.!ref` declarations are ignored there.

6.7 Validator + Cross-Schema FK Check

Level 4 cleanly separates the validation work between the certified Validator (which is unchanged) and an additional cross-schema FK check that only runs in `kaiv db` tooling. This separation preserves the certifiability of the existing runtime — the Validator never performs external I/O.

The Validator (Existing Certified Runtime — Unchanged)

- Reads `.daiv` and `.csaiv` in lockstep via the parallel scan
- Constant memory, self-contained, no external I/O
- Certifiable to ASIL-D / IEC 61508
- Validates types, constraints, intra-document FKs (Level 2), and locale-aware ordering (Level 3)
- Completely unaware of cross-schema FK declarations

Cross-Schema FK Check (`kaiv db validate` — New)

- Runs during `kaiv db validate` (explicit) or `kaiv db add --validate-fks`
- Performs one SQLite query per FK field: `SELECT 1 FROM ref_values WHERE schema…= AND field…= AND value…=`
- Reports a FK violation if any FK value does not exist
- Not part of the certified runtime — `kaiv db` tooling concern only

The boundary: the Validator produces the pass/fail result. The cross-schema FK check layers referential-integrity diagnostics on top. The Validator is always run; the cross-schema FK check runs only when the schema has `.!ref` declarations and the tooling is `kaiv db`.

The certified runtime is unchanged. The ~2–5K LOC certified DFA + parallel scan sees no new grammar, no new states, no new memory requirements. FK declarations embedded in the `.csaiv` are invisible to the Validator (the `[fk::]` lines use a bracket-only format without `'`, which the DFA skips by the same rule as other non-DFA annotation lines).

Compilation model change. A schema with `.!ref` declarations compiles as "one file plus its transitive `.!ref` closure in, one file out." The schema compiler resolves the referenced schemas from `ksaiv.com` (same machinery used for `.!schema` extension) and performs type compatibility checks at compile time. The `ref_values` SQLite table is populated and queried only by `kaiv db add` / `kaiv db validate`.

6.8 RESTRICT-Only Deletion

When a namespace is an FK target (another schema has `.!ref` pointing to it), deletes are subject to referential integrity:

— **Before any delete:** `kaiv db validate --before-delete` queries `SELECT 1 FROM ref_values WHERE schema…= AND field…= AND value…=`
— **If any row exists:** `kaiv db` reports the conflict and blocks the delete
— **If no rows exist:** the delete proceeds; `ref_values` rows for the deleted file are cleaned up on next `kaiv db add`

RESTRICT is the only deletion behavior supported. No CASCADE, no SET NULL. CASCADE requires rewriting multiple files triggered by a single delete; SET NULL requires knowing every referencing file and updating each one — both require a level of coordination that is explicitly out of scope. RESTRICT is a single SQLite query: simple, fast, and architecturally consistent with the rest of `kaiv db`.

Future exploration may add CASCADE/SET NULL deletion behaviors, but that is explicitly deferred.

6.9 Architectural Impact

Component	Impact
Lexer	Recognizes `.!compose:NAMEPATH SCHEMA JOIN_CONDITION` and `.!ref:alias schemapath` as `DECLARATION` tokens (same class as `.!schema`, `.!types`, `.!registry`). Recognizes `$alias.field` in value position — the `.` after the alias name distinguishes it from `$field` (plain field ref) and `$.name` (variable ref). Regular-grammar extension — no DFA structural change.

Component	Impact
Schema compiler	Reads `.!compose` and `.!ref` declarations. For `.!ref`: fetches target schemas from `ksaiv.com` (same resolution path as `.!schema`), performs cross-schema type compatibility check at compile time, and emits `[fk::field=namespace::targetfield]` lines in the `.csaiv`. For `.!compose`: records the composition declaration in the `.csaiv` for the Level 4 processor to consume. Compilation is now "one file + `.!ref`/`.!compose` closure in, one file out."
Compiled schema (.csaiv)	Gains `[fk::]` constraint lines (FK) and compose-directive lines. These are lines read by Level 4 tooling, not DFA lines — bracket-only format without `'`, invisible to the Validator.
Compose stage (new — Level 4 only)	A Level 4 processor inserts a Compose stage between the Compiler and the Denormalizer (see §6.4 above for semantics). The Compose stage reads the corpus, joins, and emits self-contained composed `.raiv` files. Not part of the certified runtime — the Compose stage requires corpus access.
Validator	**Unchanged.** Skips `[fk::]` lines and compose-directive lines (same skip rule as other non-DFA annotation lines). The DFA never evaluates Level 4 declarations.
Cross-schema FK check (new — Level 4 only)	`kaiv db validate` (or any Level 4 tool) reads `[fk::]` lines from `.csaiv`. For each FK field in the indexed `.daiv`, performs an existence query against the corpus (e.g. `SELECT 1 FROM ref_values WHERE schema…= AND field…= AND value…=`). Reports violations found.
`.kaiv.db` `ref_values` table	New table in `kaiv db`'s SQLite index (schema in `dbaiv/DESIGN.md` §12.6, in the spec repository). Updated on every `kaiv db add` for FK-target schemas. Queried by `kaiv db validate` for FK-referencing schemas.
Index path	`kaiv db add` gains `ref_values` update for FK-target schemas.
Delete path	Gains a RESTRICT check against `ref_values` before any delete.
Canonical form (.daiv)	Unchanged. `$alias.field` is a schema-level declaration, not a data-level expression — it does not appear in `.daiv` files. Composed `.daiv` files are regular `.daiv` documents with embedded arrays; no Level 4-specific syntax appears in them.
Immutable wings	**None.** `.!compose` and `.!ref` declarations are ignored on `kdaiv.com`, `ksaiv.com`, `chraiv.com`. No corpus index exists on those platforms.
Certification	**None.** The certified runtime (the Validator) is entirely unchanged. Level 4 is a `kaiv db`-class tooling concern, not a certified-runtime concern.

6.10 Compose vs. Cross-Schema FK — When to Use Which

Compose and cross-schema FK are complementary mechanisms within Level 4, not substitutes. Compose is **eager materialization** — pay the join cost at compose time, ship self-contained pre-joined files. FK is a **lazy constraint** — store the reference, validate existence on demand. Both can coexist in the same database, on different relationships.

Use compose when…	Use cross-schema FK when…
Relationship is parent-centric (1:many — customer with orders)	Reference fan-out is high (orders → customer; embedding the customer in every order would duplicate data ~1000×)
Embedded entity adds value at the consumer (CDN-served, self-contained fetch — no second request to resolve a relationship)	Each record has multiple cross-references (order → customer + shipping_address + billing_address + product — embedding all four inflates the file and duplicates data across many records)
Both schemas are owned by the same publisher	Reference target is externally owned (you can validate against it without taking on the right or ergonomic fit to compose against it)
Eager materialization is acceptable (low-write workloads — re-emission on update is cheap)	Constraint is the goal, not the materialization (audit logs, event streams, fact tables in analytics)
Relationship semantics fit as embedding	RESTRICT-on-delete referential integrity is required across files

The choice is per-relationship, not per-schema: a `customer-with-orders` view composed at publish time can coexist with a separate `audit_log` schema using FK references to point at customers without embedding them.

6.11 Open Design Questions

The path-identity and metaschema drafts (§6.2, §6.3) resolve the questions earlier drafts left open — data-source standardization (a corpus is a path-addressed tree; a metaschema binds to its root), schema membership (declared by path-pattern bindings, not discovered by directory walks or naming conventions), versioning (reproducible exactly over an immutable corpus handle), the relationship to `.!registry` (registries serve *named entities*; the corpus is the path-addressed *document tree*; the metaschema is that tree's contract), and cross-file UNIQUE (a corpus-collection clause). What remains open:

- **Metaschema surface syntax.** The authored file format (working extension `.msaiv`), its compiled form, the clause grammar for schema bindings and corpus collections, and how a metaschema names its corpus root. The path-pattern language is bounded in advance by the Quarb compatibility rule (§6.3).

— **Binding negotiation.** How a processor is told *which* metaschema governs a corpus (a well-known path in the corpus root, a registry lookup, tool configuration), and precedence when several claim overlapping patterns.
— **Auth, caching, and failure modes.** Out of scope for the format; left to implementations. The spec should at least describe what a Level 4 processor must do when a corpus is unreachable (fail vs warn vs skip) — currently unspecified.
— **Other corpus-dependent features.** Cross-file aggregate views and federated queries remain future work — now bounded by the corpus model rather than by data-source ambiguity.

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 (§2.6.9). 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.

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`
Type check	*Only where the compiled field carries a type item* (`!str` or a union — see §7.3): the data line's type annotation MUST equal it or be one of the union alternatives, else `TypeMismatchError`. A field lowered to a bare value constraint (`!int`, `!float`, `!bool`, `!null`, `!b64`, any named type) carries **no** type item; its conformance is enforced by the constraint check below, not a type-name comparison, so an unannotated `!str` data line whose value satisfies the constraint validates.
Namepath match	Compare namepath from schema line against data line

Step	Action
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 (§2.6.13) guarantees every declared field a line, so the scan never branches on the optional marker
Constraint validation	If schema line has constraints, validate data value against them
Array loop	If schema namepath contains `@` and the next data line has the same namepath prefix with an incremented index, stay on the same schema line and continue consuming data lines

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 (`/^-?[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, 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. A `!null` alternative therefore enforces its `/^$/` empty-payload constraint (§2.6.13). 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 '
// (since quoted names use "" doubling, not '', there is no ambiguity;
//  the first bare ' in the line is always the metadata-data delimiter;
//  any provenance list ?sourceID or ?id1,id2 with optional per-entry
   @timestamp
//  and optional #dpid in the metadata prefix is before this ' and is
   skipped
//  or validated against the .? header table without affecting the split)
schema_line = next(csaiv);
for each data_line in daiv:
    // split each line on "'" to extract type and namepath
    (data_type, data_namepath_value) = split_on_apostrophe(data_line);
    if !matches_any_schema_line(data_namepath_value):  // undefined field
        if strict: error("undefined field", data_namepath_value);
        continue;  // relaxed: skip the data line, keep the schema pointer
    (schema_constraint, schema_namepath_req) = split_on_apostrophe(
   schema_line);
    while schema_line && is_element_line(schema_namepath_req)
                      && !namepath_matches(schema_namepath_req,
   data_namepath_value):
        // empty collection: contributes no data lines; element counts
        // are enforced by the Pass-1 cardinality check, not here
        schema_line = next(csaiv);
        if schema_line: (schema_constraint, schema_namepath_req) =
   split_on_apostrophe(schema_line);
    if !schema_line || !namepath_matches(schema_namepath_req,
   data_namepath_value):
        // materialization guarantees presence and order (§ Null Semantics)
   :
        // a mismatch is always a missing or out-of-order declared field
        error("missing required field", schema_namepath_req);
    if !type_matches(schema_constraint, data_type):
        error("type mismatch", schema_namepath_req);
    validate_constraints(schema_constraint, value_from(data_namepath_value)
   );
    if is_array_field(schema_namepath_req) && has_next_index(
   data_namepath_value):
        continue;  // stay on same schema line for array index-loop
    schema_line = next(csaiv);
// remaining schema lines: empty collections are fine, anything else is
   missing
while schema_line:
    if !is_element_line(split_on_apostrophe(schema_line).namepath):
        error("missing required field", split_on_apostrophe(schema_line).
   namepath);
    schema_line = next(csaiv);
```

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 anywhere in the document (§11, strict vs. relaxed), and the defined fields that follow still have to find their schema lines in order. `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.

Approximately 12 lines of C. Constant memory. Linear time. No heap allocation. 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]+$/` + `..num`. The `.csaiv` drops every `&`-reference and lowers the type *name* of any type that reduces to a value constraint (`!int`, `!float`, `!bool`, `!null`, `!b64`, and all named `.taiv` types → `/pattern/ ..span [range/enum]`). **Three cases retain a type token**, because their semantics are not captured by a value constraint: **`!str`** — the identity type, "any string" — is emitted as `!str`; a **unit-carrying type** is emitted as `!type:unit` (first item) so the Validator can byte-compare the unit; and a **union** is emitted as its type item — each alternative carrying its lowered constraint group in parentheses (`!null(/^$/)|int(/^-?[0-9]+$/..num)`) — so the Validator can discriminate the active variant from the data line's annotation and apply that

variant's constraints (§2.6.10). No other type names, and no `&`-references, 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, all types lowered to `/pattern/ ..span [range]` or `{enum}` forms (or a retained type item — `!str` for an unconstrained string field, `!alt1|alt2` for a union), 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 the retained `!str` type item first: `!str #[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=
/^-?[0-9]+$/ ..num [1,65535]'/@servers/::port=
/^-?[0-9]+$/ ..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 `/^-?[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 (§2.6.14) 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>
settings=
```

Compiled `.csaiv`

```
/^-?[0-9]+$/ ..num'/config/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_]*$/] [max=100]
/^-?[0-9]+$/ ..num'/config/settings::=
```

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

> **Implementation status.** The reference pipeline implements both sides: map data lowering (the entry lines, §2.6.14) and `!map<…>` schema-field lowering with the map validation scan (conformance group `schema/005-map`).

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

```
.!kaivmap 1 acme/config-to-hub
.!source acme/server-config
.!target hub/server-endpoint
```

- `.!kaivmap VERSION MAP-ID` is the format declaration, parallel to `.!kaivschema` / `.!kaivtype` / `.!kaivunit`; `MAP-ID` is the mapping's own registry path on `ksaiv.com`. The version follows the same syntax and rules as every other format declaration (§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. Both are required, each exactly once.
- `.!via MAP-ID` (optional, repeatable) records the provenance trail of a *composed* mapping: one declaration per hop, in application order (§8.5).
- `.!drop SOURCE-NAMEPATH` (optional, repeatable) documents a deliberately unmapped source field. Unmapped source fields are skipped either way (§8.3); the declaration makes the omission machine-readable so publish-time tooling can distinguish intent from oversight.

8.2 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.

8.3 Execution Model

The **mapper** is a single-pass streaming 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 (§2.6.13), ready for the target's Validator with no further resolution. The mapper needs schema-sized memory (the mapping table plus both compiled schemas) and streams the data — the same memory class as the Level 1 pipeline; it performs no corpus I/O.

8.4 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 (§2.6.9). 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.

8.5 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 compose too, because an override produces a valid target value that enters the next mapping as ordinary input.

A composed, published mapping records its derivation with `.!via` declarations, one per hop in application order:

```
.!kaivmap 1 acme/config-to-helm
.!source acme/server-config
.!target helm/values-v1
.!via acme/server-to-k8s
.!via k8s/k8s-to-helm
```

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.

8.6 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 and the corresponding ARCHITECTURE.md sections take over from there.

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.

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 `//` 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.

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; Level 4 productions are exploratory and excluded (§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         = 1*DIGIT

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.
library-path  = path-seg *( "/" 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
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 / kaivschema-decl / kaivtype-decl / kaivunit
   -decl
                 / kaivmap-decl / schema-decl / types-decl / units-decl
                 / registry-decl / provenance-req-decl / ref-decl / compose
   -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" 1*ws version eol
kaivschema-decl  = ".!kaivschema" 1*ws version 1*ws ( library-path / url )
                   [ 1*ws "strict" ] eol
```

```
kaivtype-decl    = ".!kaivtype" 1*ws version 1*ws library-path eol
                   ; kaivunit-decl (.faiv header) in § Unit Definition
   Files
kaivmap-decl     = ".!kaivmap" 1*ws version 1*ws library-path eol

; .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
registry-decl    = ".!registry" 1*ws path-seg "=" url eol
provenance-req-decl = ".!provenance:" ( "required" / "source" / "timestamp"
    / "none" ) eol
source-id-decl   = ".?" prov-ident 1*ws uri eol

ref-decl         = ".!ref:" bare-name 1*ws library-path eol            ;
   Level 4 (WIP)
compose-decl     = ".!compose:" *any-char eol                          ;
   Level 4 (WIP)

url              = <an absolute http(s) URI per RFC 3986>
uri              = <a URI per RFC 3986, or an opaque non-whitespace
   identifier>
```

The Level 4 productions are recognized by the Lexer (keyword membership in §2.1.1) but their interiors are exploratory; `compose-decl` is deliberately left opaque here pending the corpus abstraction (§6.11).

10.4 Type References, Constraints, Units, Provenance

```
core-type     = "int" / "float" / "bool" / "null" / "b64" / "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    ; a constraint narrows the
                                               ; alternative it follows
union-type    = union-alt *( "|" union-alt )   ; schema annotation position
   only
csaiv-alt     = type-ref [ "(" *lowered-item ")" ]
                ; compiled form: each alternative's lowered definition +
                ; narrowing, concatenated whitespace-free (items are
                ; self-delimiting); bare when the group is empty
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
   therefore
                ; cannot be pattern-constrained in this version; match it
   with
                ; "." or a broader class.

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 ]
timestamp     = 8DIGIT "T" 6DIGIT "Z"         ; YYYYMMDDTHHmmSSZ, 16 chars
```

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 \. \/ \\`. Backreferences, lookaround, and lazy quantifiers are excluded — they are incompatible with the constant-memory DFA execution model (ARCHITECTURE.md §11). An escaped ASCII letter or digit other than `\d` (`\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.

10.5 Metadata Annotation Lines (Rule 6 — Authored Files Only)

```
metadata-line   = type-annotation-line / prov-annotation-line
                / named-annotation-line / constraint-line

type-annotation-line  = "!" ( union-type / map-type / type-ref *inline-
   constraint
                        [ ":" unit-expr ] ) *( 1*ws re-literal ) eol
prov-annotation-line  = "?" provenance-list eol
named-annotation-line = "&" type-name *( 1*ws constraint-item ) eol

; .taiv constraint lines — space-separated items above a &name= definition:
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 ]

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 (§2.5.9).

Compiled schema lines (`.csaiv`):

```
csaiv-line       = csaiv-field-line / collection-line / map-coll-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*ws "[key::" pattern "]"
                   *( 1*ws "[" ( "min" / "max" ) "=" 1*DIGIT "]" ) eol
                   ; optional key-pattern / entry-count bounds for a map
                   ; (§ Maps in the Compiled Schema); ns-path has no "@"
csaiv-constraint = csaiv-item *( 1*ws csaiv-item )
```

```
csaiv-item       = pattern / span / range / enum / length
                 / "!" ( csaiv-union [ ":" unit-expr ]
                       / "str"
                       / type-ref ":" unit-expr )
                   ; three retained type items (§ The Schema Compiler): "!
   str"
                   ; (identity), a unit-carrying "!type:unit" (the
   Validator
                   ; byte-compares the unit), and a union (the data-line
                   ; discriminant); every other single type is lowered to a
                   ; value constraint and its name dropped. Never "&". A
                   ; non-core name survives only as a union alternative or
   a
                   ; unit carrier.
                   ; canonical item order: 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 ) 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)

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:" ( library-path / url ) ] ")"
   eol
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 "]"
```

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 — §2.6.7), 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 `.!kaiv` (or `.!kaivschema` / `.!kaivtype` / `.!kaivunit` / `.!kaivmap` / `.!kaivmetaschema`) does not match `^[0-9]+(\.[0-9]+){0,2}$`.	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.
`EMPTY_KEY_ERROR`	A data line starts with `=` (optionally preceded by whitespace) — the key is empty or pure whitespace.	

Error	Condition	Notes
`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 `=`.	
`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. Names with `-`, a leading digit, 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 (§2.6.7), and `min`/`max` as unquoted field names in a Level 2 table header (§10.7).
`INVALID_DIRECTIVE_ERROR`	A line beginning with `.!` (after stripping leading whitespace) does not begin with a keyword from the §2.1.1 table: `.!kaiv`, `.!kaivschema`, `.!kaivtype`, `.!kaivunit`, `.!kaivmap`, `.!kaivmetaschema`, `.!schema`, `.!source`, `.!target`, `.!via`, `.!drop`, `.!types`, `.!units`, `.!registry`, `.!provenance`, `.!ref`, `.!compose`, `.!bind`, `.!unique`, `.!fk`.	The inventory table in §2.1.1 is authoritative; this list mirrors it.

Error	Condition	Notes
`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 (§2.7.8).	Malformed clauses are Lexer-detected wherever constraints appear — schemas, `.taiv` constraint lines, and metadata annotations in authored `.kaiv`. Unknown-unit membership requires resolution and is detected by the Compiler.

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`	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.	Compiler

Error	Condition	Detected by
`VariableContextError`	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.	Compiler
`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.)	Compiler
`SchemaDuplicateKeyError`	A `.saiv` schema text contains duplicate field definitions.	Schema compiler
`SchemaResolutionError`	A `.!schema`, `.!types`, `.!ref`, `.!source`, `.!target`, or `.!via` reference cannot be retrieved from the specified URL or registry.	Compiler / mapper / `kaiv db validate`
`SchemaInheritanceCycleError`	A `.!schema` inheritance chain among `.saiv` files revisits a schema already in the chain (§3.5.6).	Schema compiler
`SchemaOptionalWithoutDefaultError`	An optional field (`?=`) whose resolved default is inapplicable (fails the field's own constraints, §2.6.9) 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 (§2.6.13). 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. (Fields not defined in a relaxed schema are unaffected.)	Validator
`UndefinedFieldStrictSchemaError`	A data text contains an entry for a field not defined in a strict schema.	Validator
`TypeMismatchError`	For a field whose compiled form carries a type item (`!str`, a unit-carrying `!type:unit`, or a union `!a|b`): the data line's type annotation is neither that type nor among the union alternatives, or its canonical unit is not byte-identical to the declared one. A field lowered to a bare value constraint carries no type item and never raises this — a wrong-shaped value fails the constraint as `ConstraintViolationError` instead.	Validator
`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 (§8.4). 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/timestamp`, or any provenance under `none`.	Validator
`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) or `$alias.field` (Level 4) has a value not present in the referenced field set.	Pass 2 (application layer) / `kaiv db validate`
`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`).	Validator (Level 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 `.!kaivschema` declaration:

```
.!kaivschema 1 acme/server-config strict
```

The schema compiler propagates the `.!kaivschema` declaration — including the modifier — into the `.csaiv` header verbatim, which is where the Validator reads it. A `.csaiv` without a `strict` modifier (including one with no `.!kaivschema` header at all) 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

Extension	Role
`.csaiv`	compiled schema
`.taiv`	type library
`.qaiv`	query
`.msaiv`	metaschema (Level 4, draft — §6.3.1)
`.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 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.

12.3.1 Provisional Types

Until IANA registration (RFC 6838) is obtained, implementations SHOULD use the provisional forms:

- `application/vnd.kaiv`
- `text/vnd.kaiv`

12.3.2 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).

12.3.3 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.

12.4 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.

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 46, §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

array, **54**
array appending operator, **54**
array extending operator, **54**

block stack, **55**

collection constraint line, **93**
collection constraint lines, **68**
collection-level constraints, **65**
Compiler, **3**
constraint triple, **31**

data point identifier, **21**
Declarations, **13**
default value, **36**
definition line, **49**
denormalized kaiv text, **3**
Denormalizer, **3**
DFA composition, **60**
dimensionless unit, **45**

element counter, **55**
elements, **54**
encapsulated schema reference, **15**

field keys, **59**
field projection operator, **3**
field references, **22**
field table, **24**
fields, **58**
format declaration, **14**

kaiv document, **3**
kaiv text, **3**

layered resolution model, **15**
length constraint, **52**
literals, **3**, **58**

map-entry line, **95**
mapper, **97**
mapping, **96**
metaschema, **79**
mixed array, **3**, **56**

name, **3**
named type, **28**
namepath, **3**
namespace, **3**, **58**

Provenance, **20**

relational kaiv text, **3**

schema declaration, **14**
splat, **23**
struct assignment operator, **57**
Structs, **57**
structure line, **4**

table definitions, **64**
tree, **3**

unit, **44**

Validator, **3**
vector assignment operator, **54**