Iota Syntaxis Specification

iotasyntaxis.org

Version 1.0 RC5 (2026-07-11)

Abstract

Iota Syntaxis is a lightweight annotation syntax orthogonal to most text formats (Markdown, HTML, LaTeX, plain text). A text is any file or string carrying Iota Syntaxis constructs.

This document is normative.

The renderer layer (how parsed Iota Syntaxis constructs are transformed into host-format markup) is specified separately in `iotasyntaxis.org/renderer/latest`. Parser conformance (this document) and renderer conformance are independent.

The key words MUST, SHOULD, and MAY are used as defined in Bradner [1997].

Contents

Part I: Syntax & Semantics

1 Design Principles

1. **Format-agnostic.** Every construct works identically in `.md`, `.txt`, `.tex`, and `.html` files.
2. **Doubled sigils.** Every core construct uses a doubled ASCII punctuation character.
3. **Unicode identifiers.** Sigils use ASCII characters available on most keyboards. Identifier character sets follow Unicode `XID_Start` / `XID_Continue` (plus underscore as an honorary `XID_Start`); see the ASCII-only Jot Syntax variant for the portability-first counterpart.
4. **Inline and unobtrusive.** Constructs appear within prose without disrupting human readability.
5. **Orthogonal to tooling.** This specification defines syntax and parser output; renderer behavior is specified separately in `iotasyntaxis.org/renderer/latest`. Storage, indexing, querying, and rendering beyond the renderer output rules are out of scope.
6. **No escaping.** Implementations MAY offer parsing-control flags (§9) to disable individual construct families.

2 The Doubled-Symbol Family

Iota Syntaxis comprises eight core constructs:

Sigil	Name	Example
`..`	Tag	`..project`
`::`	Category	`::work::projects`
`--`	ID Reference	`--meeting_notes`, `--work--42`
`//`	File Reference	`//notes.md`, `///work/notes.md`
`;;`	Resource	`;;report.pdf`
`??`	Todo	`??review proposal`
`==`	URL	`==https://example.com`
`!!`	Bookmark	`!!introduction`

Each sigil is immediately followed by its payload. No whitespace is permitted between the sigil and the payload.

In addition, three suffix forms attach to references and resources:

Suffix	Name	Example
`:N`, `:N-M`, `:N+L`	Line fragment specifier	`--42:10-20`
`#name`	Bookmark specifier	`--42#intro`
`##a,b`	Bookmark range specifier	`--42##intro,summary`

3 Construct Boundaries

Every construct is delimited by a left and a right boundary. These rules apply uniformly unless a per-construct exception is noted.

3.1 Left Boundary

A construct's sigil MUST be preceded by either start-of-line or whitespace (space or tab). Any other preceding character — including punctuation — is not a valid left boundary.

Invalid	Corrected
`(--42)`	`(see --42)` or `( --42)`
`"--42"`	`"see --42"` or `" --42"`
`--42/--43`	`--42 / --43`
`foo..bar`	`foo ..bar`

3.2 Right Boundary

A construct's payload ends at the first of:

1. whitespace (space or tab),
2. end of line, or
3. a trailing punctuation character from the set `. , ; : " \' ! ? ) ] }`.

Trailing punctuation is **not** consumed as part of the payload.

Input	Payload	Trailing
`See --42.`	`42`	`.`
`Check --42, --43, and --44.`	`42`, `43`, `44`	`,`, `,`, `.`
`(see --42)`	`42`	`)`
`Read ..meeting, then continue.`	`meeting`	`,`

Per-construct exceptions.

— **Resource references (`;;`) and file references (`//`)** apply the `<filename>` / `<filepath>` grammar (§7.6) greedily before trailing-punctuation trimming. Internal `.` characters within a filename are consumed by the grammar. After the greedy match, trailing characters from the trim set `. , ; : " ' ! ? ) ] }` are stripped from the end of the filename and treated as following punctuation outside the construct, exactly as for URLs below. Internal occurrences of trim-set characters (e.g., the parentheses in `;;report(1).pdf`) are not affected. A filename that genuinely ends in a trim-set character (e.g., `draft(old)`) consequently cannot be written flush against following punctuation.

— **URLs (`==`)** apply the `<http-url>` grammar (§7.8) greedily, matching all non-whitespace characters. After the greedy match, trailing characters from the trim set `. , ; : " \' ! ? ) ] }` are stripped from the end of the URL and treated as following punctuation outside the construct. Stripping is applied repeatedly until the trailing character is outside the trim set or the URL is empty. Internal occurrences of trim-set characters (e.g., `?` introducing a query string, `;` in a matrix parameter, `.` in a hostname) are not affected.

— **Labeled URLs** (`==(LABEL)URL`, `==[LABEL]URL` — §4.7) MAY contain whitespace inside the label. The whitespace rule applies only after the label's closing delimiter, i.e., to the URL portion.

— **Todo items (`??`)** are terminated by end of line, not by whitespace or trailing punctuation (§4.6). The rules above apply instead to the individual constructs post-scanned from the todo description (§5.2).

Suffix introducers before whitespace. A suffix or specifier introducer appended to a construct is treated as construct syntax only when payload content follows. When an introducer composed of trim-set characters (`:`, `::`, `..`, `;;`) is immediately followed by whitespace or end of line, it is **not** a suffix attempt: the construct ends before the introducer, and the introducer's characters are ordinary trailing punctuation. `See --42: the discussion` emits the bare reference `--42` with `:` as trailing punctuation; `--42::` at end of line likewise emits `--42`. The bookmark-specifier introducers `#` and `##` are not trim-set characters; followed by whitespace or end of line they remain parse errors (§13). This rule concerns introducers *appended to* a construct; a free-standing sigil at a construct boundary with no payload is an empty payload parse error (§11).

Trim-then-validate. The right boundary is determined in two steps: the construct's grammar consumes greedily, then a maximal run of trim-set characters extending to whitespace or end of line is trimmed as trailing punctuation. Any remaining character the grammar cannot consume — an internal trim-set character (`..meet!ing`) or a character outside the trim set (`--42$ref`) — invalidates the whole candidate span: the parser MUST report the full span as a parse error (`Jots/Parsing/InvalidPayloadChars`, or a more specific canonical code where one applies — §11) and MUST NOT silently emit a shorter valid prefix. `..meeting!` is therefore a clean tag (the `!` run reaches whitespace and is trimmed), while `..meet!ing` and `--42$ref` are parse errors.
URL right-boundary examples:

Input	URL payload	Trailing
`==https://example.com.`	`https://example.com`	`.`
`(see ==https://example.com)`	`https://example.com`	`)`
`==https://example.com/p?q=1.`	`https://example.com/p?q=1`	`.`
`==https://example.com/a.b)`	`https://example.com/a.b`	`)`

4 Constructs

4.1 Tags (`..`)

Syntax. `..NAME` or `..NAME..NAME2..NAME3...`

A tag is two periods immediately followed by one or more identifier characters (Unicode `XID_Start` + `XID_Continue`, plus underscore). Adjacent tags MAY be chained without intervening whitespace; each `..NAME` segment in the chain is a separate tag.

Semantics.

- Tags are case-insensitive. The canonical form is lowercase, and is used in serialization, indexing, and rendered output.
- Tags carry no hierarchy.
- A text MAY contain any number of tagstrings, each carrying one or more chained tags; tagstrings are separated from one another by whitespace or other non-tag content.
- Extracted tags are deduplicated after case-folding.
- Tags appearing as suffixes inside other constructs (annotated references, §6; annotated todos, §6.5) are governed by those constructs' grammar, not by the free-standing tag production.

Optional extension. Implementations MAY accept `#word` as an alternate tag form when the `parse_hashtags_as_tags` flag is enabled. The hashtag payload MUST begin with a

letter (§7.9): `#2026` and `#_draft` are not matched, although `..2026` and `.._draft` are. This extension is disabled by default.

Example. A tagged meeting note:

```
Met with the design team today ..meeting ..urgent
```

4.2 Categories (`::`)

Syntax. `::NAME` or `::NAME::SUB::SUBSUB...`

A category is two colons immediately followed by one or more `::`-separated word segments (each segment is one or more identifier characters — Unicode `XID_Start` + `XID_Continue`, plus underscore).

Semantics.

- Categories are hierarchical: each `::` separator introduces a deeper level.
- Categories are case-insensitive. The canonical form is lowercase.
- A filter on `::work` matches `::work` and any deeper prefix (`::work::projects`, `::work::projects::alpha`, ...).
- Multiple categories MAY appear anywhere in a text.
- Extracted categories are deduplicated after case-folding.
- Categories appearing as suffixes inside other constructs are governed by those constructs' grammar, not by the free-standing category production.

Example. A categorized project note:

```
Project kickoff notes ::work::projects::alpha
```

4.3 ID References (`--`)

ID references link between texts. An ID is one or more identifier characters (`XID_Start` + `XID_Continue`, plus underscore); its meaning is determined by the host tool.

Forms.

Form	Syntax	Example
Same-space	`--ID`	`--42`, `--meeting_notes`
Cross-space	`--SPACE--ID`	`--work--42`

Semantics.

- ID and space identifiers are case-sensitive.
- References are directional: the file containing the reference is the *source*; the referenced text is the *target*. Bidirectional indexing (forward references and backlinks) is implementation-defined.
- An ID reference MAY carry annotation suffixes (§6) and a single optional specifier (line fragment, bookmark, or bookmark range — §4.9–§4.11).

Examples. ID references in a sentence:

```
See --42 for the previous discussion.
Related to --work--42 in the work journal.
```

4.4 File References (//)

File references identify another text by file path rather than ID.

Forms.

Form	Syntax	Example
Same-space	`//FILEPATH`	`//notes.md`
Cross-space	`///SPACE/FILEPATH`	`///work/notes.md`

`FILEPATH` is a `/`-separated sequence of filename segments, where each segment follows the `<filename>` grammar (§7.6).

Semantics.

- Filepaths and space identifiers are case-sensitive.
- A file reference MAY carry annotation suffixes (§6) and a single optional specifier (§4.9–§4.11).

Examples. File references in a sentence:

```
See //notes.md and //subfolder/architecture.md.
Diagram in ///design/assets/layout.svg.
```

4.5 Resource References (;;)

Syntax. `;;FILENAME`

A resource reference identifies an auxiliary file associated with a text (e.g., an attachment, an embedded image, a bibliography file).

Filename grammar. A valid filename is one or more identifier characters (Unicode `XID_Start` / `XID_Continue`, plus underscore), period (`.`), hyphen (`-`), tilde (`~`), comma (`,`), at-sign (`@`), exclamation mark (`!`), and parentheses (`(`, `)`), subject to:

- It MUST NOT begin with `-` or `.`.
- Consecutive periods (`..`) are not permitted.

The exclusions of whitespace, shell metacharacters (`* ? [ ] { } | \ ``), URI-sensitive characters (`% + # & =`), `:`, and `;` apply uniformly. The full grammar is in §7.6.

Portability. File and resource references intentionally support a portable subset of path syntax (identifier characters plus a small punctuation set). Implementations targeting hosts with arbitrary filesystem path conventions (Windows backslash paths, paths containing spaces or non-ASCII characters, NFC/NFD normalization, etc.) SHOULD provide a host-specific mapping layer that translates the portable syntax to and from native paths, rather than extending the grammar.

Semantics.

— Filenames are case-sensitive.
— A resource reference MAY carry annotation suffixes (§6.6) and a single optional specifier (§4.9–§4.11).
— `;;filename` denotes a resource on the current text. For resources on other texts, see inter-jot resources (§5).

Examples. Resource references in a sentence:

```
See the attached ;;report.pdf for details.
Diagram in ;;architecture.png.
```

4.6 Todo Items (??)

Syntax. A todo item is composed of:

```
??[.][<importance>][<blockers>]<action>[<due>][<cat>][<tag>]...[ <description>]
```

A todo item begins with `??` and is terminated by end of line. Throughout this specification, `<cat>` denotes a full sigil-inclusive category suffix (`::path`) and `<tag>` denotes a full sigil-inclusive tag suffix (`..word`); the leading sigil is part of the placeholder.

Components.

Component	Syntax	Req.	Example
Done marker	`.` immediately after `??`	No	`??.review`, `??.!!review`, `??.(draft)implement`
Importance	`!` (1–3 times)	No	`??!`, `??!!`, `??!!!`
Blockers	`(<entry>(,<entry>)*)` — see *Blockers* below	No	`??(draft)implement`, `??(draft::spec..john)review`
Action	`<word>`	Yes	`??buy`, `??review`
Due date	`@YYYY-MM-DD` or `@YYYY-MM-DDTHH:MM[TZ]`	No	`??send@2026-03-01`
Annotation	`[<cat>][<tag>]...`	No	`??review::work..important`
Description	`: free text` or `free text`	No	`??buy: milk and eggs`

Semantics.

— A todo MAY be marked complete by inserting a single literal period (`.`) immediately after the `??` sigil. The period MUST appear in that exact position — before any importance prefix, before the blocker slot, and never embedded in the description. `??.review`, `??.!!review`, and `??.(draft)implement` are well-formed done todos; `??!.review`, `??(draft).implement`, and `??review.` are **not** the done form (the period is not in marker position and is either part of normal parsing or a parse error per the action grammar). Done and active todos share an identical object schema; they are emitted on separate top-level lists in parser output — active todos on `todos`, done todos on `done_todos` (§8). The `parse_todos` flag (§9) governs both lists uniformly.
— A done todo MAY carry an optional **completion timestamp** of the form `.@<iso-date>` anywhere in its description. The leading `.` distinguishes it from a stray due-date `@`; the date grammar is the same `<iso-date>` production as `@` (§7.7) and admits both forms — a calendar date (`.@2026-03-15`) or a full datetime with timezone offset

(`.@2026-03-15T14:30-05:00`). The `.@` MUST be preceded by whitespace (the standard construct-boundary rule reduces to this in a description, where start-of-line is structurally unreachable). Right-boundary trimming (§3.2) applies to any trailing punctuation after the date. The parser emits the matched date on the `completed` field of the done todo (§8) and applies the same timezone-completion behavior as for `due` (datetimes without an offset SHOULD receive the local offset at creation time; date-only values are not modified). At most one `.@<iso-date>` MAY appear per done todo; a second occurrence is a validation error (§13) and the first occurrence wins. The completion timestamp is recognized **only** inside a done todo's description span; the same character sequence in an active todo's description, or anywhere outside a todo span, is inert text and is neither extracted nor diagnosed. As with other description-post-scanned constructs (§5.2), the `.@<iso-date>` text remains textually present in `description`.

— Importance levels: 0 (none), 1 (`!`), 2 (`!!`), 3 (`!!!`).

— Due dates use ISO 8601. The `@` prefix is required: without it the string is parsed as part of the description.

— The `@` MAY be directly concatenated with the action (`??send@2026-03-01`) or whitespace-separated (`??send @2026-03-01`).

— When both a due date and annotation suffixes are present, the due date MUST precede the suffixes. `??review::work@2026-03-15` is not valid.

— An optional colon before the description is purely cosmetic. The colon MAY be directly attached to the action (`??buy:milk`), whitespace-separated from the action (`??buy : milk`), or attached to the description (`??buy: milk`); all three are equivalent. The colon is not stored in the parsed `description` field; only the free-text portion that follows is retained, with leading and trailing whitespace trimmed.

— When a datetime omits a timezone offset, implementations SHOULD append the local timezone offset at creation time. Date-only values are not modified.

— The annotation slot accepts an optional category and zero or more tags per §6.5; the category MUST precede the tags. Additional categories, tags, and other constructs may appear in the description and are merged into the todo per §5.2.

— Todos are emitted in document order, one object per matched occurrence. Two textually identical todos on different lines are preserved as two distinct entries; deduplication is the consumer's responsibility.

Examples. Active todos, done todos, and todos with completion timestamps:

```
??buy milk
??!send@2026-03-01 quarterly report
??!!review@2026-03-15T09:00-05:00 architecture doc
??review::work..important
??.review proposal
??.!!(draft::spec)merge::work..urgent
??.send quarterly report .@2026-03-02
??.review architecture doc .@2026-03-15T17:45-05:00
```

Description-form classifier. The forms below are all valid and parse to action `buy` with description `milk`:

Input	Action	Description
`??buy milk`	`buy`	`milk`
`??buy: milk`	`buy`	`milk`
`??buy : milk`	`buy`	`milk`
`??buy:milk`	`buy`	`milk`

When a due date or annotations are present, the optional colon may attach to whichever of them is last:

```
??send@2026-03-01: quarterly report
??review::work..important: architecture doc
```

Todo span and nested constructs. A todo claims everything from `??` to end of line. Tags, categories, references, file references, resource references, inter-jot resources, and URLs that appear inside that span — whether in the annotation slot or post-scanned from the description — are emitted into the enclosing todo, not the text's top-level lists. See §5.2. For tags and categories this means the annotation-slot form (`??buy..groceries milk`) and the description form (`??buy milk ..groceries`) are equivalent: both yield a todo whose `tags` includes `groceries`.

Bookmark declarations (`!!name`) are the one exception: a `!!name` inside a todo span is **not** extracted as a bookmark and is **not** attached to the todo. The parser MUST drop the declaration from output and SHOULD emit a diagnostic (§14). To anchor the line that carries a todo, place the bookmark **before** the `??`:

```
!!followup ??review proposal
```

Because bookmarks are line-level (§4.8), this anchors the entire line — todo included — without entangling the bookmark with the todo's span.

Blockers. A todo MAY declare one or more *blockers* — patterns identifying other todos whose completion must precede this one. The blocker slot appears in parentheses between the optional importance prefix and the action. Each entry is structured as an action plus optional `<cat>` and `<tag>` constraints:

```
??(draft)implement
??(draft::spec)review
??(draft..john)review
??(draft::spec..john)review
??(review,typeset)publish
??!(review)merge
```

Entry grammar. Each entry takes one of two shapes:

- `<action>[<cat>][<tag>]...` — a literal action word (`<word>`), optionally constrained by a single `<cat>` and zero or more `<tag>`s.
- `*<cat>[<tag>]...` — the wildcard `*` matches any action. A `<cat>` is REQUIRED in this form to provide scope; zero or more `<tag>`s MAY follow.

Within an entry, `<cat>` MUST precede the `<tag>`s, mirroring the annotation-slot ordering rule. At most one `<cat>` per entry, with zero or more `<tag>`s. No importance, due date, or description fragments are permitted inside a blocker entry.

Multiple entries are comma-separated and combine **conjunctively** (all must complete):

```
??(draft::spec,review..urgent)merge
```

Parse errors.

- Empty parens: `??()action`.
- Empty entries: `??(,a)action`, `??(a,)action`, `??(a,,b)action`.
- Bare wildcard with no `<cat>`: `??(*)action`.
- Wildcard with tags only: `??(*..tag)action`.
- Whitespace inside the parens, between the closing paren and the action, or between the importance prefix and the opening paren.

Canonical diagnostic codes for these conditions are listed in §11.

Resolution is application-defined. The parser MUST NOT attempt to resolve a blocker entry to any todo. Each entry is emitted on the todo's `blockers` field as a structured object (§8); matching it to a real todo — within the same text, across texts, against an external index, or not at all — is the responsibility of the consuming application.

4.7 URLs (==)

Syntax. `==URL`, `==(LABEL)URL`, or `==[LABEL]URL`

Semantics.

- The URL MUST begin with `http://` or `https://`.
- Two label syntaxes are supported. The parenthesized form MUST NOT contain literal `)`; the bracketed form MUST NOT contain literal `]`. Both produce identical renderer output.
- URLs are emitted in document order, one object per matched occurrence. Deduplication is the consumer's responsibility.

Examples. Bare and labeled URLs in prose:

```
Check out ==https://example.com for more info.
See ==(project homepage)https://github.com/example/project for source.
See ==[docs (internal)]https://example.com/docs for details.
```

4.8 Bookmarks (!!)

Syntax. `!!NAME`
A bookmark declares a named anchor at the line containing the declaration.

Semantics.

- A bookmark is line-level: a `!!name` declaration anchors the entire line on which it appears, regardless of where in the line it occurs. Bookmarks have no inline extent and do not subdivide a line.
- Bookmark names are case-sensitive.
- Bookmark names MUST be unique within a text (see §14). On duplicate declarations, the first declaration in document order wins: subsequent duplicates are dropped from the `bookmarks` list (§8) and each raises `Jots/Validation/DuplicateBookmark`.
- Bookmark declarations are not permitted inside a todo span (§4.6). A `!!name` between `??` and end of line is dropped from output and produces a diagnostic (§14); to anchor the line, place the bookmark before the `??`.
- The renderer emits a bookmark as an invisible anchor in supported formats (`iotasyntaxis.org/renderer/latest` §2.6).

Example. A bookmark declaring an anchor for the line it sits on:

```
!!introduction
This is the introduction section.
```

4.9 Bookmark Specifiers (#name)

A `#name` suffix appended to an ID reference, file reference, or resource reference links to the named anchor in the target.

Form	Example
Same-space ID reference	`--42#introduction`
Cross-space ID reference	`--work--42#summary`
Same-space file reference	`//notes.md#conclusion`
Cross-space file reference	`///work/notes.md#overview`
Standalone resource	`;;report.pdf#chapter1`
Inter-jot resource	`--42;;report.pdf#chapter1`

A bookmark specifier is mutually exclusive with a line fragment specifier (§4.11) and with a bookmark range specifier (§4.10): a reference MAY carry at most one specifier.

When a bookmark specifier is used on a resource whose type does not support named anchors (e.g., a plain CSV file), implementations SHOULD emit a resolution warning (§14).

4.10 Bookmark Range Specifiers (##)

A `##` suffix selects the content between two bookmarks.

Form	Syntax	Meaning
Start to bookmark	`##,name`	From the beginning of the text/resource up to (but excluding) `!!name`
Bookmark to end	`##name,`	From immediately after `!!name` to the end of the text/resource
Between bookmarks	`##name1,name2`	From immediately after `!!name1` up to (but excluding) `!!name2`

Semantics.

- Selection is **line-level and strictly between** the two bookmark declaration lines. Both endpoint lines are excluded from the selected range; only lines strictly between them are included. `##,name` selects from the first line of the text up to (but excluding) the line on which `!!name` is declared. `##name,` selects from the line immediately after `!!name`'s declaration line through the end of the text.
- The `,` separates the two bookmark names; bookmark names contain only word characters and so the comma is unambiguous as a separator.
- At least one bookmark name MUST be provided. `##,` alone is not valid.
- Mutually exclusive with line fragment and bookmark specifiers; a reference MAY carry at most one specifier.
- The canonical order with annotation suffixes is: base reference → specifier → category → tags.

Where it applies. ID references, file references, resource references, and inter-jot resources (on the resource filename, not on the locator prefix).

Examples. Bookmark range specifiers on each construct family that accepts them:

```
--42##intro,summary
--work--42##,conclusion
//notes.md##intro,summary
;;data.csv##header,footer
--42;;report.pdf##intro,conclusion
```

Host format support. No widely deployed host format natively supports range selection between named anchors. Implementations SHOULD pass `##` syntax through into rendered link targets unchanged.

Parser representation. Parsers expose a `bookmark_range` object with `start_bookmark` and `end_bookmark` (each `string | null`). `start_bookmark` is `null` for the `##,name` form; `end_bookmark` is `null` for the `##name,` form. Resolution to byte offsets occurs at render time, not at parse time.

4.11 Line Fragment Specifiers

A line fragment specifier narrows a reference or resource to a specific line or line range, by numeric position.

Form	Syntax	Meaning	Example
Single line	`:LINE`	One line; shorthand for `:LINE-LINE`	`//notes.md:42`
Range	`:START-END`	Lines `START` through `END` inclusive	`--42:10-20`
Offset+length	`:START+LENGTH`	Line `START` plus `LENGTH` additional lines	`--42:10+5`

Where it applies. ID references, file references, resource references, and inter-jot resources (on the resource filename, not on the locator prefix). Line fragment specifiers also appear in inline include constructs defined in `iotasyntaxis.org/preprocessor/latest`.

Semantics.

— All three forms normalize to a `(start, end)` pair:

- `:LINE` → `(LINE, LINE)`
- `:START-END` → `(START, END)`
- `:START+LENGTH` → `(START, START + LENGTH)`

— Mutually exclusive with bookmark and bookmark range specifiers; a reference MAY carry at most one specifier.

— Line numbers are 1-based. `0` is not a valid line number.

— The canonical order with annotation suffixes is: base reference → specifier → category → tags.

— Implementations MAY impose a maximum line number (e.g., $2^{31} - 1$) and SHOULD treat arithmetic overflow during normalization of the `:START+LENGTH` form as a validation error.

Examples. Line fragments combined with annotations:

```
--42:10-20::work..draft
//notes.md:100-120..review
;;data.csv:1-50::import..urgent
```

5 Inter-Jot Resources

Reference and resource constructs compose to denote resources owned by other texts:

Syntax	Meaning
`;;file.pdf`	Resource on the current text
`--42;;file.pdf`	Resource on text `42` (same space)
`--work--42;;file.pdf`	Resource on text `42` in space `work`
`//notes.md;;file.pdf`	Resource on file-text `notes.md`
`///work/notes.md;;file.pdf`	Resource on file-text `notes.md` in space `work`
`--42;;data.csv:1-50`	Lines 1–50 of `data.csv` on text `42`
`--42;;report.pdf##intro,conclusion`	Bookmark range on `report.pdf` on text `42`

Disambiguation rule. When multiple construct patterns could match overlapping spans of input, the longest match takes priority. Among matches of equal length, the more specific construct wins: inter-jot resource references take priority over standalone references or standalone resources; cross-space forms take priority over same-space forms; and annotated references take priority over bare references.

Locator vs. resource. In an inter-jot reference, the prefix (`--42`, `--work--42`, `//notes.md`, `///work/notes.md`) serves only as a locator. Annotation suffixes and specifiers MUST NOT appear between the locator and the `;;` separator (§12). They MAY appear after the filename:

```
--42;;report.pdf..draft
--42;;report.pdf::work
--42;;data.csv:1-50
--42;;report.pdf##intro,conclusion
```

These suffixes annotate the resource edge, not the current text's own tag or category lists.

5.1 Malformed Tokens

When the input begins a candidate token with a recognized sigil and the following characters introduce suffix, specifier, or inter-jot syntax, the parser MUST validate the entire candidate token against the grammar before deciding what to emit. If the token violates an ordering or mutual-exclusion constraint defined in this specification, the parser MUST emit a single diagnostic covering the full candidate span and MUST NOT silently emit a shorter valid prefix.

For example, given the input `--42..tag;;file.pdf`, a parser MUST NOT emit `--42..tag` as a tagged ID reference and treat the remainder as inert text. The presence of `..tag;;file.pdf` makes the whole token a candidate inter-jot resource reference; the annotation suffix between the locator and `;;` violates §12. The parser MUST report the entire `--42..tag;;file.pdf` span as a parse error.

The same rule applies to specifier composition: given `--42:10;;file.pdf`, the parser MUST report the entire span as a parse error per §12 rather than emitting `--42:10` as a same-space reference with a line fragment specifier.

5.2 No Construct Nesting

Constructs do not nest. Once the parser consumes a span as part of a matched construct, no other construct is emitted from within that span. Specifically:

- **URL labels are opaque.** Sigils, sigil-like sequences, and other payload-shaped text inside a URL label do not produce constructs. `==(see ..tag)https://example.com` yields one URL with label `see ..tag`; no tag is extracted.
- **Filenames are opaque.** Internal periods and other characters in a filename matched by `<filename>` (§7.6) do not produce tags, categories, or other constructs.
- **A todo owns every construct inside its span.** A todo's span runs from `??` to end of line and covers two regions: the annotation slot (between the action or due date and the description) and the description text. Tags and categories in the annotation slot, and any tags, categories, ID references, cross-references, file references, cross-file references, inter-jot resources, resource references, and URLs found by post-scanning the description, are emitted into the enclosing todo's lists (`categories`, `tags`, `references`, `cross_references`, `file_references`, `cross_file_references`, `inter_jot_resources`, `resources`, `urls`). They do **not** contribute to the text's top-level lists. On done todos (`??.`, §4.6) the post-scan also recognizes the completion-timestamp construct (`.@<iso-date>`) and emits the parsed date on the todo's `completed` field; the same syntax is **not** recognized in an active todo's description or outside any todo span. Matched constructs remain textually present in the description; the `description` string is not modified. Bookmark declarations (`!!name`) are excluded from the post-scan: the todo span owns the line, so a positional anchor inside it has no coherent meaning. See the next bullet.
- **Annotation slot is positional but not exclusive.** The annotation slot grammar (`[<cat>][<tag>]...`) still applies between the action (or due date) and the description, with the category-before-tags ordering constraint of §6.5. Tags and categories outside that slot are picked up by description post-scan instead, and feed the same todo lists; the two sources are merged and deduplicated.
- **Bookmark declarations are zero-width.** A `!!name` declaration consumes only its own span; surrounding text is parsed normally for other constructs.
- **Bookmark declarations are forbidden inside todo spans.** A `!!name` between `??` and end of line is dropped: the parser MUST NOT emit it as a bookmark and MUST NOT attach it to the enclosing todo, and SHOULD emit a diagnostic (§14). The text remains in the todo's `description` unchanged. Authors who want to anchor the line that carries a todo MUST place `!!name` before the `??` (§4.6); a bookmark is line-level, so a pre-`??` declaration anchors the entire todo line.

6 Annotated References

ID references and file references MAY carry tag and category suffixes that annotate the *relationship* (edge) from source to target, not the current text itself.

6.1 Tagged References

A reference MAY be followed by one or more tag suffixes (`..word`):

```
--42..next
--42..important..draft
--work--42..next
//notes.md..draft
///work/notes.md..urgent
```

Tags on references follow the same case-folding rules as free-standing tags: case-insensitive, stored in canonical lowercase.

6.2 Categorized References

A reference MAY be followed by a single category suffix (`::path`):

```
--42::work
--42::work::projects
//notes.md::review
```

Only one category MAY appear per reference. Categories on references follow the same case-folding rules as free-standing categories.

6.3 Combined Suffixes

When both a category and tags are present, the category MUST precede the tags:

```
--42::work..important
--work--42::projects::alpha..next..urgent
//notes.md::review..draft
```

`--42..important::work` is **not valid** and MUST NOT be matched as an annotated reference (§12).

6.4 Semantics

— Suffixes annotate the edge from source text to target, not the texts themselves.
— A reference's suffixes do not add to the text's own tag or category lists. `--42..next` does not tag the current text with `..next`.
— Tags on a single reference are deduplicated after case-folding.
— At most one category MAY appear per annotated reference.

6.5 Annotated Todos

Todos accept the same suffix system as references. A todo's suffixes classify the todo item itself, not the current text.

```
??review..accounts
??review..accounts..overdue
??review::work
??review::work::projects..urgent..draft
??!send::finance..quarterly
```

The category-before-tags ordering constraint is mandatory (§12). The annotation slot accepts at most one category and zero or more tags. Description post-scan (§5.2) MAY contribute additional categories and tags; the slot and post-scan sources are merged and deduplicated into the todo's `categories` and `tags` lists. None of these contribute to the text's own tag or category lists.

6.6 Annotated Resource References

Resource references accept the same suffix system as ID and file references. Suffixes annotate the resource edge from the current text to the resource.

```
;;report.pdf..draft
;;report.pdf::work
;;report.pdf::work..draft..urgent
```

```
--42;;report.pdf..draft
--work--42;;report.pdf::work..draft..urgent
///work/notes.md;;file.pdf..tag
```

The category-before-tags ordering constraint is mandatory (§12). On inter-jot resource references, suffixes MUST appear *after* the filename, not between the locator and `;;` (§12). Suffixes do not add to the text's own tag or category lists.

7 Formal Grammar

The following grammar defines Iota Syntaxis in BNF. All productions are applied as a global, multiline scan of the input text.

7.1 Common Productions

Listing 1: Common productions

```
<word-char>          ::= <letter> | <digit> | "_"
<word>               ::= <word-char>+
<letter>             ::= "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
<digit>              ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<uint>               ::= <digit>+
<ws>                 ::= " " | "\t"
<line-terminator>    ::= "\n" | "\r\n" | "\r"
<line-start>         ::= (* zero-width assertion: start of input, or
                            position immediately after a
                            <line-terminator> *)
<line-end>           ::= (* zero-width assertion: end of input, or
                            position immediately before a
                            <line-terminator> *)
<construct-boundary> ::= (* zero-width assertion: position preceded by
                            beginning of line or whitespace; equivalent
                            to the regex lookbehind (?<=^|[ \t]) with
                            multiline ^ *)
```

> *In the Iota Syntaxis variant, `<word-char>` is widened so that `<letter>` ranges over Unicode `XID_Start` and `<word-char>` additionally admits `XID_Continue` code points. The enumeration above is written for the ASCII Jot Syntax variant; both variants share the rest of the grammar verbatim.*

`<construct-boundary>`, like `<line-start>` and `<line-end>`, is a zero-width assertion: it constrains position but does not consume input.

A line is the maximal run of input characters between two `<line-terminator>` matches (or the bounds of the input). LF (U+000A), CRLF (U+000D U+000A), and CR (U+000D) are all recognized as terminators; CRLF is consumed as a single two-byte unit, never as two adjacent CR + LF terminators. Implementations MUST handle mixed-terminator input without splitting CRLF, and SHOULD preserve the original terminators verbatim when emitting `raw` substrings.

7.2 Tags

Listing 2: Tags

```
<tagstring> ::= <construct-boundary> <tag-tail>
<tag-tail>  ::= ".." <word> ( ".." <word> )*
```

A `<tagstring>` is one or more `..word` segments chained without intervening whitespace; each segment is emitted as a separate tag. The `<tag-tail>` production is also reused by annotated reference and annotated todo suffixes (§7.11, §7.12).

Matched tags are case-folded to lowercase and deduplicated.

7.3 Categories

Listing 3: Categories

```
<category>      ::= <construct-boundary> "::" <category-path>
<category-path> ::= <word> ( "::" <word> )*
```

Matched categories are case-folded to lowercase and deduplicated.

7.4 ID References

Listing 4: ID references

```
<id-locator>       ::= <construct-boundary> "--" <word>
<cross-id-locator> ::= <construct-boundary> "--" <word> "--" <word>

<same-ref>  ::= <id-locator>       ( <line-fragment-specifier> | <bookmark-specifier>
    | <bookmark-range-specifier> )?
<cross-ref> ::= <cross-id-locator> ( <line-fragment-specifier> | <bookmark-specifier>
    | <bookmark-range-specifier> )?
```

`<line-fragment-specifier>` is defined in §7.13. `<bookmark-specifier>` and `<bookmark-range-specifier>` are defined in §7.14.

ID identifiers and space identifiers are case-sensitive. ID references are emitted in document order, one parser-output object per matched occurrence. Deduplication, indexing, and cross-text reconciliation are the responsibility of the consumer, not the parser.

7.5 File References

Listing 5: File references

```
<file-ref-locator>       ::= <construct-boundary> "//" <filepath>
<cross-file-ref-locator> ::= <construct-boundary> "///" <word> "/" <filepath>
<filepath>               ::= <filepath-segment> ( "/" <filepath-segment> )*
<filepath-segment>       ::= <filename>

<same-file-ref>  ::= <file-ref-locator>       ( <line-fragment-specifier> | <bookmark-
    specifier> | <bookmark-range-specifier> )?
<cross-file-ref> ::= <cross-file-ref-locator> ( <line-fragment-specifier> | <bookmark-
    specifier> | <bookmark-range-specifier> )?
```

Filepaths and space identifiers are case-sensitive. File references are emitted in document order, one parser-output object per matched occurrence. Deduplication is the consumer's responsibility.

7.6 Resource References

Listing 6: Resource references

```
<resource-ref>     ::= <construct-boundary> ";;" <filename> ( <line-fragment-
    specifier> | <bookmark-specifier> | <bookmark-range-specifier> )? <annotations>?
<filename>         ::= <fname-segment> ( "." <fname-segment> )*
<fname-segment>    ::= <fname-start-char> <fname-char>*
<fname-start-char> ::= <word-char> | "~" | "," | "@" | "!" | "(" | ")"
<fname-char>       ::= <fname-start-char> | "-"
```

<annotations> is defined in §7.11.

A <filename> MUST NOT begin with - or .. Consecutive periods (..) within a filename are not producible under this grammar.

After the greedy <filename> / <filepath> match, trailing trim-set characters are stripped repeatedly and treated as following punctuation, per the right-boundary exception in §3.2.

7.7 Todo Items

Listing 7: Todo items

```
<todo>             ::= <construct-boundary> "??" <todo-body>
<todo-body>        ::= <todo-content> <todo-terminator>
<todo-terminator>  ::= <line-end>
<todo-content>     ::= <done-marker>? <importance>? <blockers>? <action> <due-date>?
    <annotations>? <description>?

<done-marker>      ::= "."
<completion-time>  ::= <construct-boundary> ".@" <iso-date>
<blockers>         ::= "(" <blocker-entry> ( "," <blocker-entry> )* ")"
<blocker-entry>    ::= <word> <category-suffix>? <tag-tail>?
                     | "*" <category-suffix> <tag-tail>?

<category-suffix> ::= "::" <category-path>
<importance>       ::= "!" | "!!" | "!!!"
<action>           ::= <word>
<due-date>         ::= <ws>* "@" <iso-date>
<description>      ::= ":" <free-text>
                     | <ws>+ ":"? <free-text>

<iso-date>         ::= <date-only> | <datetime>
<date-only>        ::= <digit> <digit> <digit> <digit> "-" <digit> <digit> "-" <digit>
    <digit>
<datetime>         ::= <date-only> "T" <time> <timezone>?
<time>             ::= <digit> <digit> ":" <digit> <digit> ( ":" <digit> <digit> )?
<timezone>         ::= "Z" | ( "+" | "-" ) <digit> <digit> ":" <digit> <digit>

<free-text>        ::= (* any characters except newline *)
```

<annotations> is defined in §7.12; the category-before-tags ordering constraint is mandatory. The <due-date>, when present, MUST precede the annotations.

A <done-marker> is positional: a literal . MUST appear at the single position immediately after ??, and only there, to mark a done todo. The parser MUST consume <done-marker> before attempting any of the productions that follow; a . in any other position within <todo-content> does not satisfy <done-marker>. Done todos are extracted by the same `parse_todos` flag and emitted with the same object schema as active todos, but routed to a separate `done_todos` list (§8).

<completion-time> is recognized **only** during description post-scan (§5.2) of a done todo (one whose <todo-content> began with <done-marker>). In an active todo's description, in non-description regions of a todo, or in input outside any todo span, the .@ sequence is not a recognized sigil. When matched, the parsed date is emitted on the done todo's `completed` field (§8) and the matched span remains textually present in `description`. At most one <completion-time> MAY contribute to a single done todo's `completed` field; the first match in document order wins, and each subsequent match raises `Jots/Validation/DuplicateCompletionTime` (§13).

Extracted todos are trimmed of leading/trailing whitespace. Todos are emitted in document order, one object per matched occurrence, within their respective list.

The <bookmark> production (§7.14) is suppressed inside <todo-content>: a !!name between ?? and the todo terminator is not matched as a bookmark and produces a diagnostic per §14. All other constructs in the description body are post-scanned per §5.2.

7.8 URLs

Listing 8: URLs

```
<url-construct>       ::= <construct-boundary> "==" <url-payload>
<url-payload>         ::= <labeled-url> | <bare-url>
<labeled-url>         ::= <paren-labeled-url> | <bracket-labeled-url>
<paren-labeled-url>   ::= "(" <paren-label-text> ")" <http-url>
<bracket-labeled-url> ::= "[" <bracket-label-text> "]" <http-url>
<bare-url>            ::= <http-url>
<paren-label-text>    ::= (* any characters except ")" *)
<bracket-label-text>  ::= (* any characters except "]" *)
<http-url>            ::= ( "http://" | "https://" ) <non-ws>+
```

URLs are emitted in document order, one object per matched occurrence.

7.9 Hashtag Tags (Optional Extension)

Listing 9: Hashtag tags

```
<hashtag> ::= <construct-boundary> "#" <letter> <word-char>*
```

Requires `parse_hashtags_as_tags` to be enabled. The payload MUST begin with a `<letter>`: hashtag payloads are a strict subset of `..` tag payloads (`#2026` is not matched). Matched hashtags are case-folded to lowercase, merged with `..` tags, and deduplicated.

Extension freeze. Hashtag tags are the only sanctioned extension to Iota Syntaxis 1.0. A conforming implementation MAY support this extension and MUST NOT define additional construct families, sigils, suffix forms, or specifier syntaxes beyond those in this specification. Implementations are free to add tooling, output formats, indexing strategies, or storage models around the parser output; the syntax surface itself is closed for 1.0.

7.10 Inter-Jot Resources

Listing 10: Inter-jot resources

```
<inter-jot-ref> ::= <cross-id-locator>       ";;" <filename> ( <line-fragment-
    specifier> | <bookmark-specifier> | <bookmark-range-specifier> )? <annotations>?
                  | <id-locator>             ";;" <filename> ( <line-fragment-
                      specifier> | <bookmark-specifier> | <bookmark-range-specifier> )
                      ? <annotations>?
                  | <cross-file-ref-locator> ";;" <filename> ( <line-fragment-
                      specifier> | <bookmark-specifier> | <bookmark-range-specifier> )
                      ? <annotations>?
                  | <file-ref-locator>       ";;" <filename> ( <line-fragment-
                      specifier> | <bookmark-specifier> | <bookmark-range-specifier> )
                      ? <annotations>?
```

Annotation suffixes and specifiers MUST NOT appear on the locator prefix. They appear only on the resource side, after `<filename>`.

7.11 Annotated Reference Suffixes

Listing 11: Annotated reference suffixes

```
<annotated-ref>   ::= <ref-base> ( <line-fragment-specifier> | <bookmark-specifier> |
    <bookmark-range-specifier> )? <annotations>
```

```
<ref-base>        ::= <id-locator> | <cross-id-locator> | <file-ref-locator> | <cross-
    file-ref-locator>
<annotations>     ::= <category-suffix> <tag-tail> | <category-suffix> | <tag-tail>
```

`<tag-tail>` is as defined in §7.2; `<category-suffix>` is as defined in §7.7. The category-before-tags ordering constraint is mandatory.

7.12 Annotated Todo Suffixes

Listing 12: Annotated todo suffixes

```
<annotated-todo> ::= <construct-boundary> "??" <done-marker>? <importance>?
    <blockers>? <action> <due-date>? <annotations> <description>? <todo-terminator>
<annotations>    ::= <category-suffix> <tag-tail> | <category-suffix> | <tag-tail>
```

`<tag-tail>` is as defined in §7.2. The category-before-tags ordering constraint is mandatory; the `<due-date>`, when present, MUST precede the annotations.

7.13 Line Fragment Specifiers

Listing 13: Line fragment specifiers

```
<line-fragment-specifier> ::= ":" <uint>
                            | ":" <uint> "-" <uint>
                            | ":" <uint> "+" <uint>
```

The three forms normalize to a `(start, end)` pair as described in §4.11. In a reference or resource context, a single `:` (i.e., not `::`, which always introduces a category suffix) that is immediately followed by at least one non-whitespace character introduces a line fragment specifier; the characters that follow are validated against the productions above. If those characters do not form a valid `<uint>`, range, or offset, the malformed-token rule (§5.1) applies and the entire reference or resource span is reported as a parse error per §13. The parser MUST NOT silently emit the bare reference and treat the `:` and what follows as inert text. A `:` immediately followed by whitespace or end of line is **not** a specifier attempt: the bare reference is emitted and the `:` is ordinary trailing punctuation (§3.2) — `See --42: the discussion` emits the bare reference `--42`.

The grammar admits zero as a line number (`--42:0`, `--42:0-5`), but zero is not a valid line number: such fragments are matched and emitted by the parser, then flagged as validation errors per §13. Permissive matching ensures the diagnostic carries the offending construct's `loc` and `raw` for accurate reporting. A zero offset (`:START+0`) is well-formed and equivalent to `:START` — it normalizes to `(START, START)`.

7.14 Bookmarks

Listing 14: Bookmarks

```
<bookmark>                ::= <construct-boundary> "!!" <word>
<bookmark-specifier>      ::= "#" <word>
<bookmark-range-specifier> ::= "##" <word>? "," <word>?
```

A `<bookmark-range-specifier>` MUST have at least one of the two `<word>?` slots non-empty; `##,` alone is not valid.

Bookmark names in declarations and specifiers are case-sensitive. Bookmark names within a single text MUST be unique. Bookmark declarations are extracted in document order.

The `<bookmark>` production is suppressed inside `<todo-content>` (§7.7); see §14.

`<line-fragment-specifier>`, `<bookmark-specifier>`, and `<bookmark-range-specifier>` are mutually exclusive on a given reference or resource: at most one of the three may be matched.

Part II: Parser

8 Parser Output

The **parser** accepts source text and optional configuration flags and returns structured metadata. It does not modify the input text.

Output fields.

Field	Type	Description
`tags`	list of strings	Canonical (lowercase) free-standing tag names outside any todo span
`categories`	list of strings	Canonical (lowercase) free-standing category paths outside any todo span
`references`	list of objects	Same-space ID references outside any todo span
`cross_references`	list of objects	Cross-space ID references outside any todo span
`file_references`	list of objects	Same-space file references outside any todo span
`cross_file_references`	list of objects	Cross-space file references outside any todo span
`inter_jot_resources`	list of objects	Inter-jot reference–resource pairs outside any todo span
`resources`	list of objects	Standalone resource references outside any todo span
`todos`	list of objects	Parsed active todo items (no done marker)
`done_todos`	list of objects	Parsed done todo items (`??.` form); same `todo` object schema as `todos`
`urls`	list of objects	Parsed URL constructs outside any todo span
`bookmarks`	list of objects	Bookmark declarations in this text (case-sensitive, document order; on duplicate declarations the first wins, §4.8)
`diagnostics`	list of objects	Diagnostics raised during parsing; per-object shape per §10.1

Object structures. Every construct object listed below additionally carries `loc` (an object locating it in the source) and `raw` (the exact matched source string). These two fields are omitted from the per-object field lists for compactness; they are normative and MUST be present on every object emitted in `references`, `cross_references`, `file_references`, `cross_file_references`, `inter_jot_resources`, `resources`, `todos`, `done_todos`, `urls`, and `bookmarks`, unless suppressed by the `emit_loc` flag (§9). The nested `fragment` and `bookmark_range` sub-objects do not carry their own `loc` / `raw`.

Object	Fields
`reference`	`id`, `fragment`, `bookmark`, `bookmark_range`, `category`, `tags`
`cross_reference`	`space`, `id`, `fragment`, `bookmark`, `bookmark_range`, `category`, `tags`
`file_reference`	`filepath`, `fragment`, `bookmark`, `bookmark_range`, `category`, `tags`
`cross_file_reference`	`space`, `filepath`, `fragment`, `bookmark`, `bookmark_range`, `category`, `tags`

Object	Fields
inter_jot_resource	space, id, filepath, filename, resource_fragment, resource_bookmark, resource_bookmark_range, category, tags
resource	filename, fragment, bookmark, bookmark_range, category, tags
fragment	start (integer), end (integer), raw (string)
bookmark_range	start_bookmark (string | null), end_bookmark (string | null)
blocker	action (string), cat (string | null), tags (list of strings)
todo	importance (integer), blockers (list of blocker objects), action, categories, tags, due (string | null), completed (string | null), description, references, cross_references, file_references, cross_file_references, inter_jot_resources, resources, urls
url	url, label (string | null)
bookmark	name (string)

Field rules.

- category (singular) and tags on references, cross-references, file references, cross-file references, inter-jot resources, and resources carry annotation suffixes per §6. They are edge-level annotations and do not appear in the text's top-level tags or categories lists. Defaults are null and [] respectively.
- categories (plural) and tags on a todo are populated from two sources: the annotation slot (§6.5) and description post-scan (§5.2). The two sources are merged and deduplicated. Both default to [].
- references, cross_references, file_references, cross_file_references, inter_jot_resources, resources, and urls on a todo are populated by post-scanning the description text. They follow the same per-construct schema as their text-top-level counterparts, default to [], and are emitted in document order. Constructs found inside a todo description appear *only* on the enclosing todo; they are excluded from the text's top-level lists of the same kind.
- blockers on a todo is an ordered list of blocker objects as written in the blocker slot (§4.6), preserving declaration order. Each entry carries action (string; the literal action word, or "*" for a wildcard entry), cat (string | null; the ::path constraint with the leading :: stripped, case-folded to lowercase), and tags (list of strings; each ..word constraint with the leading .. stripped, case-folded to lowercase, deduplicated within the entry). The field defaults to []. The parser MUST NOT attempt to resolve a blocker entry to any todo; matching is application-defined.
- completed on a todo is null by default. On a done todo (§4.6), if the description contains a .@<iso-date> construct, completed is the matched date string, normalized identically to due (datetimes without an offset SHOULD receive the local offset at creation time; date-only values are not modified). On an active todo, completed is always null — the .@<iso-date> syntax is not recognized in active todo descriptions. At most one contributing match per done todo; subsequent matches in the same description raise Jots/Validation/DuplicateCompletionTime (§13) and are dropped without overwriting completed.
- id, space, filepath, filename, and bookmark names are case-preserved. category and tags values are case-folded to lowercase.
- fragment is null when no line fragment specifier is present; otherwise an object with three fields: start and end (integers) represent the normalized 1-based inclusive line range (§4.11), and raw is the original fragment syntax as it appeared in source (e.g., ":42", ":10-20", ":10+5"), including the leading :. The raw form is used for round-tripping by {fragment} substitution in iotasyntaxis.org/renderer/latest §1.2.
- bookmark is null when no bookmark specifier is present; otherwise the case-sensitive

bookmark name from the `#name` specifier.

- `bookmark_range` is `null` when no bookmark range specifier is present; otherwise an object with `start_bookmark` and `end_bookmark` (each `string | null`).
- `fragment`, `bookmark`, and `bookmark_range` are mutually exclusive: at most one MAY be non-null on a given construct.
- On `inter_jot_resource`, the resource-side specifiers and annotations are exposed as `resource_fragment`, `resource_bookmark`, `resource_bookmark_range`, `category`, and `tags`. The same mutual-exclusion rule applies to the `resource_*` triple.
- On `inter_jot_resource`, exactly one of `id` or `filepath` MUST be non-null. `space` is `null` for same-space references.
- Each entry in the `bookmarks` list is a `bookmark` object whose `name` field is the case-sensitive declared name (without the `!!` sigil), carrying `loc` / `raw` like every other construct object. The `loc` / `raw` span the whole `!!name` declaration, giving `iotasyntaxis.org/renderer/latest` the source location it needs to render the anchor.
- `loc` is an object with fields `unit`, `start`, and `end` locating the matched construct in the input.
  - `unit` is a string identifying the offset unit. Defined values are `"byte"` (UTF-8 byte offsets) and `"char"` (Unicode code-point offsets). Implementations MUST emit one of the defined values and MUST be consistent within a single parser invocation. Future revisions MAY define additional `unit` values (e.g., UTF-16 code units, grapheme clusters); consumers MUST treat unknown values as an error rather than silently assuming a unit. (The shared `texts-tests` conformance suite uses an additional fixture-only sentinel `"ascii"` in expected outputs to allow either defined value when the input is pure ASCII; this is a test-harness convention only, and implementations MUST NOT emit it.)
  - `start` and `end` are integer offsets in the chosen unit, zero-based and half-open: `start` is the offset of the first element of the construct, `end` is the offset immediately past the last element (so `end - start` equals the match length).
  - The location covers the construct itself after right-boundary trimming (§3.2) and excludes any trimmed trailing punctuation.
- `raw` is the exact source substring covered by `loc` — i.e., the matched construct text including its sigil, payload, specifier, and annotation suffixes, and excluding trimmed trailing punctuation.
- `diagnostics` is an array of diagnostic objects per §10.1. It is always present, defaulting to `[]` when no diagnostics fired. Canonical-code entries are deterministic in `code`, `severity`, `loc`, and order; see §10.1 for the determinism guarantee and the canonical ordering rule.

Example. Given the input:

```
Met with design team ..meeting ..urgent ::work::projects
See --42..next for context. Related to --work--42::review..urgent..draft
Check ;;report.pdf and --42;;slides.pdf and --work--99;;deck.pdf
See //notes.md..draft and ///work/design.md::review
??!review@2026-03-15::work..important architecture doc
??.draft initial outline .@2026-03-10
==(docs)https://docs.example.com
!!introduction
```

the parser returns the structure shown below. For brevity the `loc` and `raw` fields are elided from each emitted construct object; they are nonetheless required on every entry of `references`, `cross_references`, `file_references`, `cross_file_references`, `inter_jot_resources`, `resources`, `todos`, `done_todos`, and `urls`.

```
{
  "tags": ["meeting", "urgent"],
  "categories": ["work::projects"],
  "references": [
    { "id": "42", "fragment": null, "bookmark": null, "bookmark_range": null, "category": null, "tags": ["next"] }
  ],
  "cross_references": [
    { "space": "work", "id": "42", "fragment": null, "bookmark": null, "bookmark_range": null, "category": "review", "tags": ["urgent", "draft"] }
  ],
  "file_references": [
    { "filepath": "notes.md", "fragment": null, "bookmark": null, "bookmark_range": null, "category": null, "tags": ["draft"] }
  ],
  "cross_file_references": [
    { "space": "work", "filepath": "design.md", "fragment": null, "bookmark": null, "bookmark_range": null, "category": "review", "tags": [] }
  ],
  "inter_jot_resources": [
    { "space": null, "id": "42", "filepath": null, "filename": "slides.pdf", "resource_fragment": null, "resource_bookmark": null, "resource_bookmark_range": null, "category": null, "tags": [] },
    { "space": "work", "id": "99", "filepath": null, "filename": "deck.pdf", "resource_fragment": null, "resource_bookmark": null, "resource_bookmark_range": null, "category": null, "tags": [] }
  ],
  "resources": [
    { "filename": "report.pdf", "fragment": null, "bookmark": null, "bookmark_range": null, "category": null, "tags": [] }
  ],
  "todos": [
    {
      "importance": 1, "blockers": [], "action": "review", "categories": ["work"], "tags": ["important"],
      "due": "2026-03-15", "completed": null, "description": "architecture doc",
      "references": [], "cross_references": [],
      "file_references": [], "cross_file_references": [],
      "inter_jot_resources": [], "resources": [], "urls": []
    }
  ],
  "done_todos": [
    {
      "importance": 0, "blockers": [], "action": "draft", "categories": [], "tags": [],
      "due": null, "completed": "2026-03-10", "description": "initial outline .@2026-03-10",
      "references": [], "cross_references": [],
      "file_references": [], "cross_file_references": [],
      "inter_jot_resources": [], "resources": [], "urls": []
    }
  ],
  "urls": [
    { "url": "https://docs.example.com", "label": "docs" }
  ],
  "bookmarks": [{ "name": "introduction" }],
  "diagnostics": []
}
```


Description post-scan example. Given the input:

```
??review proposal ..draft ;;contract.pdf before signing
```

the todo emits:


```
{
  "importance": 0, "blockers": [], "action": "review", "categories": [], "tags": ["draft"],
```

```
  "due": null, "completed": null, "description": "proposal ..draft ;;contract.pdf
      before signing",
  "references": [], "cross_references": [],
  "file_references": [], "cross_file_references": [],
  "inter_jot_resources": [],
  "resources": [
    { "filename": "contract.pdf", "fragment": null, "bookmark": null, "bookmark_range":
        null, "category": null, "tags": [] }
  ],
  "urls": []
}
```

Note that `..draft` and `;;contract.pdf` remain textually in `description`, but are *also* extracted into the todo's `tags` and `resources`. Neither appears in the text's top-level `tags` or `resources` lists.

The specifier sub-objects, by example:

- For a reference with a line fragment, e.g. `--42:10-20::work..draft`, `fragment` is `{ "start": 10, "end": 20, "raw": ":10-20" }`. For `--42:10+5`, `fragment` is `{ "start": 10, "end": 15, "raw": ":10+5" }`.
- For a bookmark specifier, e.g. `--42#introduction`, `bookmark` is `"introduction"`.
- For a bookmark range, e.g. `--42##intro,summary`, `bookmark_range` is `{ "start_bookmark": "intro", "end_bookmark": "summary" }`. The `##name,` form yields `{ "start_bookmark": "name", "end_bookmark": null }`; the `##,name` form yields the converse.

With `loc` and `raw` included, the first entry of `references` in the example above (from a parser using UTF-8 byte offsets) is:

```
{
  "id": "42",
  "fragment": null,
  "bookmark": null,
  "bookmark_range": null,
  "category": null,
  "tags": ["next"],
  "loc": { "unit": "byte", "start": 61, "end": 71 },
  "raw": "--42..next"
}
```

and the `introduction` entry of `bookmarks` is:

```
{
  "name": "introduction",
  "loc": { "unit": "byte", "start": 373, "end": 387 },
  "raw": "!!introduction"
}
```

9 Parsing Control Flags

Configuration sources, in decreasing priority:

1. **Environment variables** — `JOTS_*` prefixed variables.
2. **Local project file** — `.jot.toml` in the current working directory or any parent directory (searched upward).
3. **User-level file** — `$HOME/.jot.toml`.

A higher-priority source overrides a lower-priority source for the same key. Keys not specified in any source use the defaults below. Implementations that expose CLI flags for these keys (e.g., `--no-loc` for `emit_loc`) SHOULD give them precedence over all three sources.

The `JOTS_*` prefix, the `.jot.toml` file name, the output field names (§8), and the `Jots/...` diagnostic layer (§10.1) are shared verbatim by the Jot Syntax and Iota Syntaxis variants; none of them vary with the variant.

Key	Env var	Default	Controls
`parse_tags`	`JOTS_PARSE_TAGS`	`true`	`..tag` extraction
`parse_categories`	`JOTS_PARSE_CATEGORIES`	`true`	`::category` extraction
`parse_references`	`JOTS_PARSE_REFERENCES`	`true`	`--id` and `--space--id` extraction
`parse_file_references`	`JOTS_PARSE_FILE_REFERENCES`	`true`	`//filepath` and `///space/filepath` extraction
`parse_resources`	`JOTS_PARSE_RESOURCES`	`true`	`;;file` extraction
`parse_todos`	`JOTS_PARSE_TODOS`	`true`	`??todo` extraction
`parse_urls`	`JOTS_PARSE_URLS`	`true`	`==url` extraction
`parse_bookmarks`	`JOTS_PARSE_BOOKMARKS`	`true`	`!!name` bookmark extraction
`parse_hashtags_as_tags`	`JOTS_PARSE_HASHTAGS_AS_TAGS`	`false`	`#word` → `..word` synonym
`emit_loc`	`JOTS_EMIT_LOC`	`true`	Emission of `loc` and `raw` on construct objects (§8)

When parsing is disabled for a construct family, matching text remains inert: it is not extracted, indexed, or transformed by the renderer.

When `emit_loc` is `false`, parser output omits the `loc` and `raw` fields from every construct object. All other fields are unchanged. Implementations SHOULD expose this as a `--no-loc` CLI flag (or equivalent) for compact output when only structured metadata is wanted.

9.1 Flag Interactions

Several flags govern more than one production. The interactions below are normative.

- **`parse_tags=false`** disables both free-standing tags (§4.1) and tag annotation suffixes on references, resources, and todos (§6). A token that would otherwise be a tag annotation is not consumed; the surrounding reference is parsed as if no tag suffix were present.
- **`parse_categories=false`** symmetrically disables both free-standing categories (§4.2) and category annotation suffixes.
- **`parse_bookmarks=false`** disables `!!name` bookmark *declarations* (§4.8) only. The `#name` bookmark specifier (§4.9) and `##` bookmark range specifier (§4.10) on references and resources are governed by the relevant reference flag (`parse_references`, `parse_file_references`, or `parse_resources`). A reference with a bookmark specifier still parses normally when `parse_bookmarks=false`.
- **Inter-jot resources** (§5) require `parse_resources=true` *and* the relevant locator family enabled: same-space ID-based locators require `parse_references=true`; file-based locators require `parse_file_references=true`. If any required flag is disabled, the candidate inter-jot token is not emitted, and its constituent text is not reparsed as a shorter construct (the malformed-token rule in §5.1 applies).
- **`parse_hashtags_as_tags`** has effect only when `parse_tags=true`. When tags are disabled, hashtags are also inert regardless of this flag.

Part III: Error Conditions

This part enumerates error conditions that may arise when parsing or rendering Iota Syntaxis.

10 Severity Levels

Severity	Meaning	Recommended behavior
Parse error	Input violates the grammar and cannot be matched as a construct.	Text is left as inert content. No entry is emitted in parser output.
Validation error	Input is grammatically valid but violates a semantic constraint.	Parser SHOULD report via diagnostics. Strict implementations MAY reject; lenient MAY accept with warning.
Resolution warning	Input is syntactically and semantically valid, but a target cannot be resolved.	Parser emits the construct normally. Renderer SHOULD warn and render the construct as-is (no link target).

Scope of "parse error". A parse error covers two distinct situations:

1. *No construct ever matched.* The input contains no candidate sigil, or a sigil whose payload is rejected by the grammar before any prefix could be considered a complete construct. The text is inert.
2. *Candidate construct rejected.* A construct's sigil is present and begins consuming characters that would form a payload, suffix, specifier, or inter-jot syntax, but the resulting token violates the grammar's payload, ordering, or composition rules (including trim-then-validate, §3.2). The parser MUST diagnose the rejected token and MUST NOT silently emit a shorter valid prefix.

For example, `==ftp://example.com` is a parse error in the second sense: the `==` sigil starts a URL construct, but the payload fails the `<http-url>` rule that requires `http://` or `https://`. No shorter URL match is emitted; the entire `==ftp://example.com` token is reported as a single parse error.

10.1 Diagnostic Shape

A conforming implementation MUST expose error conditions through a diagnostics interface. Each diagnostic MUST carry at least the following fields:

Field	Type	Description
`code`	string	Stable identifier; see naming convention below.
`severity`	`"parse_error" | "validation_error" | "resolution_warning"`	Maps to a row in §10.
`message`	string	Human-readable explanation.
`loc`	`{ unit, start, end } | null`	Source location covered by the diagnostic, using the same shape and unit semantics as parser-output `loc` (§8). `null` for diagnostics that have no source construct (configuration errors, §15).

Implementations MAY add fields (e.g., `hint`, `related_locs`, `source_label`).

Location in parser output. Implementations conforming to §8 MUST surface diagnostics in the top-level `diagnostics` array of the parser output. CLI implementations MAY additionally render a human-readable summary on stderr; the structured array is the contract.

Emitting component. Parse and validation diagnostics are emitted by the parser from the input text and configuration alone. Resolution-tier diagnostics (`Jots/Resolution/…`) additionally require *resolution context* — knowledge of which texts, files, resources, and bookmarks exist. They are emitted by whichever component performs resolution (typically the host tool or renderer) and appear in the parser-output `diagnostics` array only when the parser is invoked with such context. A parser operating on input text alone emits no resolution warnings.

Code naming convention. Diagnostic codes are slash-separated PascalCase identifiers shaped `<Layer>/<Tier>/<Name>`. The layer segment identifies the producing component:

- `Jots/…` — parser (codes in this section).
- `Jotpp/…` — preprocessor (codes in iotasyntaxis.org/preprocessor/latest §9.1).
- `Jotr/…` — renderer (no normative codes in 1.0).

The tier segment identifies severity:

- `…/Parsing/…` for parse errors (§11, §12, the parse-error rows of §13 and §14).
- `…/Validation/…` for validation errors (the validation rows of §13–§15).
- `…/Resolution/…` for resolution warnings (§14, §16).

Codes in this section are therefore all `Jots/<Tier>/<Name>`. The shape is deliberately distinct from the upper-snake-case `JOTS_PARSE_TAGS` env-var family (§9) and the lower-snake-case `parse_tags` config-flag family.

Canonical codes. Each row of §11–§16 carries a **Code** column giving the canonical code for that condition. Conforming implementations MUST emit the canonical code when reporting the listed condition, and MUST NOT re-purpose a canonical code for a different condition. The canonical-code list is append-only within a major spec version: new conditions add new codes; removed or renumbered conditions leave their codes retired (codes are never reused).

When more than one canonical condition describes the same rejected span, the most specific condition's code is emitted. `Jots/Parsing/InvalidPayloadChars` is the generic fallback: it fires only when no more specific row matches.

Diagnostic location. For each canonical-code condition in §11–§16 other than the configuration errors (§15, whose diagnostics carry `loc null`), `loc` covers the full matched construct — from the sigil through the construct's right boundary per §3.2, including any specifier and annotation suffixes. This matches the `loc` extent the construct would carry on its parser-output object had it parsed cleanly. Extension-code diagnostics MAY use any `loc` extent appropriate to the condition they report.

Determinism. For a given (input, configuration) pair, a conforming parser MUST emit a deterministic sequence of canonical-code `Jots/Parsing/…` and `Jots/Validation/…` diagnostics: every conforming parser produces the same canonical diagnostics in the same order, agreeing on each diagnostic's `code`, `severity`, and `loc`. Resolution-tier diagnostics are excluded from this guarantee — they depend on resolution context (see **Emitting component** above), not on the input alone. The `message` field is human-readable and MAY vary across implementations; consumers comparing parser outputs across implementations MUST ignore `message` and MUST filter out extension-code entries (`<Layer>/<Tier>/X/<Name>`, defined below) before comparison. Within the canonical subset, diagnostic order is ascending `loc.start`, then ascending `loc.end`, then `code` lexicographically. Diagnostics with `loc null` (configuration errors) precede all located diagnostics and are ordered by `code` lexicographically.

Extension codes. Implementations MAY emit additional diagnostics for conditions not listed in §11–§16 using the reserved extension namespace `<Layer>/<Tier>/X/<Name>` — i.e., a fixed `X` segment between the tier and a vendor-defined PascalCase name (for example, `Jots/Parsing/X/VendorFoo`). Extension codes are implementation-defined and need not be portable across parsers; consumers route them by `severity`.

Unknown-code handling. Consumers MUST tolerate unrecognized codes in the standard `Jots/Parsing/…` / `Jots/Validation/…` / `Jots/Resolution/…` namespace — these may originate in a future spec revision — by routing on `severity` and falling back to the human-readable `message`. Codes in the `<Layer>/<Tier>/X/<Name>` extension namespace MAY be ignored by consumers that don't recognize the emitting implementation.

Retired codes. Codes assigned to conditions that have since been removed from the spec are listed below for historical reference. Conforming implementations MUST NOT emit retired codes, and the codes are never reassigned to a different condition (the canonical-code list is append-only — see **Canonical codes** above).

Code	Originally	Retired in	Reason
`Jots/Parsing/InvalidLeftBoundary`	§11 (parse error; pre-1.0 row 3.2.1)	2026-04-30	Left-boundary application per §3.1 is silent skipping by design, not a reportable error condition.

One further condition was removed before canonical codes were assigned and therefore has no code to retire: a zero-length offset validation error (`:START+0`; pre-1.0 row 3.4.3). The form is now well-formed and equivalent to `:START` (§7.13).

Condition numbering. Condition rows in §11–§16 are numbered *section.row* and renumber with the document; the stable identifier for a condition is its canonical code, never its row number. (Documents predating 1.0 numbered these rows 3.2.1–3.7.7.)

11 Structural and Grammar Violations

One row originally listed here (pre-1.0 row 3.2.1) was retired; see **Retired codes** in §10.1.

#	Error condition	Severity	Reference	Example	Code
11.1	**Empty payload** — sigil with no content following	Parse error	§7.2–§7.8	`.. ` , `:: ` , `-- ` , `??`	`Jots/Parsing/EmptyPayload`
11.2	**Invalid payload characters** — candidate span contains characters the production cannot consume that are not trailing punctuation (trim-then-validate, §3.2)	Parse error	§3.2, §7.1–§7.8	`..meet!ing`, `--42$ref`	`Jots/Parsing/InvalidPayloadChars`
11.3	**Empty category segment** — consecutive `::` separators with no word between them	Parse error	§7.3	`::work::::projects`	`Jots/Parsing/EmptyCategorySegment`
11.4	**Filename starts with `-` or `.`** — forbidden by `<fname-start-char>`	Parse error	§7.6	`;;-report.pdf`, `;;.hidden`	`Jots/Parsing/InvalidFilenameStart`
11.5	**Consecutive dots in filename** — not producible under `<filename>`	Parse error	§7.6	`;;report..pdf`	`Jots/Parsing/ConsecutiveDotsInFilename`

#	Error condition	Severity	Reference	Example	Code
11.6	**Invalid filename characters** — characters outside the allowed set	Parse error	§4.5, §7.6	`;;file[1].pdf`, `;;report#2.pdf`	`Jots/Parsing/InvalidFilenameChars`
11.7	**URL missing scheme** — `==` followed by a URL without `http://` or `https://`	Parse error	§7.8	`==ftp://example.com`, `==example.com`	`Jots/Parsing/UrlMissingScheme`
11.8	**Unmatched or stray label delimiter** — label opened with `(` / `[` but not closed before end of line, or content between the closing delimiter and the URL scheme	Parse error	§4.7, §7.8	`==(label`, `==(label)text)https://...`	`Jots/Parsing/UnmatchedLabelDelimiter`
11.9	**Todo missing action** — `??` (optionally followed by `.`) followed immediately by whitespace, due date, or end of line	Parse error	§7.7	`?? @2026-03-01`, `??`, `??.`	`Jots/Parsing/TodoMissingAction`
11.10	**Malformed due date** — `@` introduces a due date whose characters do not form a valid `<iso-date>`	Parse error	§7.7	`??send@2026-3-1`	`Jots/Parsing/MalformedDueDate`
11.11	**Empty blocker list** — `()` with no entries	Parse error	§4.6, §7.7	`??()action`	`Jots/Parsing/EmptyBlockerList`
11.12	**Empty blocker entry** — leading, trailing, or doubled comma in the blocker list	Parse error	§4.6, §7.7	`??(,a)action`, `??(a,)action`, `??(a,,b)action`	`Jots/Parsing/EmptyBlockerEntry`
11.13	**Invalid wildcard blocker entry** — wildcard `*` without the REQUIRED `<cat>` constraint	Parse error	§4.6, §7.7	`??(*)action`, `??(*..tag)action`	`Jots/Parsing/InvalidWildcardBlocker`
11.14	**Whitespace in blocker slot** — whitespace inside the parens, between `)` and the action, or between the importance prefix and `(`	Parse error	§4.6, §7.7	`??(draft, review)merge`, `??(draft) merge`	`Jots/Parsing/WhitespaceInBlockers`

12 Ordering Constraint Violations

#	Error condition	Severity	Reference	Example	Code
12.1	**Tags before category on annotated reference**	Parse error	§6.3, §7.11	`--42..next::work`	`Jots/Parsing/TagsBeforeCategoryRef`
12.2	**Tags before category on annotated todo**	Parse error	§6.5, §7.12	`??review..important::work`	`Jots/Parsing/TagsBeforeCategoryTodo`
12.3	**Tags before category on annotated resource reference**	Parse error	§6.6	`;;report.pdf..draft::work`	`Jots/Parsing/TagsBeforeCategoryResource`
12.4	**Annotation suffix between locator and `;;` in inter-jot resource reference**	Parse error	§5, §7.10	`--42..tag;;file.pdf`, `--42::work;;file.pdf`	`Jots/Parsing/AnnotationBeforeInterJotSep`
12.5	**Specifier on locator prefix of inter-jot resource reference**	Parse error	§5, §7.10	`--42:10;;file.pdf`, `--42#intro;;file.pdf`	`Jots/Parsing/SpecifierOnInterJotLocator`

13 Specifier Errors

#	Error condition	Severity	Reference	Example	Code
13.1	**Start greater than end in range** — `:START-END` where `START > END`	Validation error	§4.11	`--42:20-10`	`Jots/Validation/RangeStartAfterEnd`
13.2	**Zero line number** — line numbers are 1-based	Validation error	§4.11	`--42:0`, `//notes.md:0-5`	`Jots/Validation/ZeroLineNumber`
13.3	**Non-numeric fragment content** — `:` followed by non-whitespace content that does not form a valid fragment	Parse error	§7.13	`--42:abc`	`Jots/Parsing/NonNumericFragment`
13.4	**Multiple mutually exclusive specifiers on same reference**	Parse error	§4.9, §4.10, §7.14	`--42:10-20#intro`, `--42#intro##a,b`	`Jots/Parsing/MutuallyExclusiveSpecifiers`
13.5	**Integer overflow in line fragment specifier**	Validation error	§4.11	`--42:99999999999999`	`Jots/Validation/FragmentIntegerOverflow`
13.6	**Empty bookmark range** — `##` with no bookmark names or no comma	Parse error	§4.10, §7.14	`--42##,`, `--42##`	`Jots/Parsing/EmptyBookmarkRange`
13.7	**Empty bookmark specifier name** — `#` on a reference or resource followed by whitespace or end of line	Parse error	§4.9, §7.14	`--42#`, `//notes.md#`	`Jots/Parsing/EmptyBookmarkSpecifier`
13.8	**Same bookmark in range** — `##name,name` where both names are identical	Validation error	§4.10	`--42##intro,intro`	`Jots/Validation/SameBookmarkInRange`
13.9	**Todo with semantically invalid ISO 8601 date** — grammar matches but date is invalid	Validation error	§4.6, §7.7	`??send@2026-02-30`	`Jots/Validation/InvalidDate`
13.10	**Done todo with semantically invalid completion date** — `.@<date>` matches grammar but date is invalid	Validation error	§4.6, §7.7	`??.send report .@2026-02-30`	`Jots/Validation/InvalidCompletionDate`
13.11	**Multiple completion timestamps on a single done todo** — more than one `.@<iso-date>` in description	Validation error	§4.6, §7.7	`??.send report .@2026-03-01 .@2026-03-02`	`Jots/Validation/DuplicateCompletionTime`

14 Bookmark Errors

#	Error condition	Severity	Reference	Example	Code
14.1	**Duplicate bookmark names within a text**	Validation error	§4.8	Two `!!introduction` declarations	`Jots/Validation/DuplicateBookmark`
14.2	**Bookmark specifier referencing a nonexistent bookmark**	Resolution warning	§4.9	`--42#missing` where text 42 has none	`Jots/Resolution/BookmarkNotFound`
14.3	**Empty bookmark name** — `!!` followed by whitespace or end of line	Parse error	§7.14	`!!`	`Jots/Parsing/EmptyBookmarkName`
14.4	**Bookmark declaration inside a todo span** — `!!name` between `??` and end of line	Validation error	§4.6, §5.2	`??buy milk !!followup`	`Jots/Validation/BookmarkInTodoSpan`

15 Configuration Errors

#	Error condition	Severity	Reference	Example	Code
15.1	**Invalid boolean value for parsing control flag** — value is not `true`/`false`	Validation error	§9	`JOTS_PARSE_TAGS=maybe`	`Jots/Validation/InvalidFlagValue`
15.2	**Invalid template** — unrecognized placeholder name or unbalanced `{`/`}`	Validation error	renderer §3, §4	`ref_target = "jot://{unknown}"`	`Jots/Validation/InvalidTemplate`

16 Resolution Warnings

#	Error condition	Severity	Reference	Example	Code
16.1	**ID reference to nonexistent text**	Resolution warning	§4.3	`--99999`	`Jots/Resolution/JotNotFound`
16.2	**File reference to nonexistent file**	Resolution warning	§4.4	`//deleted.md`	`Jots/Resolution/FileNotFound`
16.3	**Cross-space reference to nonexistent space**	Resolution warning	§4.3	`--nonexistent--42`	`Jots/Resolution/SpaceNotFound`
16.4	**Resource reference to nonexistent resource**	Resolution warning	§4.5	`;;missing.pdf`	`Jots/Resolution/ResourceNotFound`
16.5	**Inter-jot resource where the text exists but the resource does not**	Resolution warning	§5	`--42;;deleted.pdf`	`Jots/Resolution/InterJotResourceNotFound`
16.6	**Bookmark range referencing nonexistent bookmark(s)**	Resolution warning	§4.10	`--42##missing,also_missing`	`Jots/Resolution/BookmarkRangeNotFound`
16.7	**Bookmark range with start after end** — `!!a` appears after `!!b` in the target's document order	Resolution warning	§4.10	`--42##summary,intro`	`Jots/Resolution/BookmarkRangeStartAfterEnd`

17 Severity Classification Summary

Severity	Conditions
Parse error	11.1–11.14, 12.1–12.5, 13.3, 13.4, 13.6–13.7, 14.3
Validation error	13.1–13.2, 13.5, 13.8–13.11, 14.1, 14.4, 15.1–15.2
Resolution warning	14.2, 16.1–16.7

Conformance

A tool or library **conforms to the Iota Syntaxis parser specification** if it:

1. Locates all eight core constructs from input text using the grammar in §7.
2. Provides a **parser** interface that returns structured metadata per §8.
3. Applies the disambiguation rule in §5: longest match wins; among equal-length matches, inter-jot resources take priority over standalone references; cross-space takes priority over same-space; annotated forms take priority over bare.
4. Case-folds tags and categories to canonical lowercase, and preserves case for IDs, spaces, filepaths, filenames, and bookmark names.

5. Deduplicates extracted free-standing tags and categories after case-folding, and enforces bookmark-name uniqueness within a text (§4.8). References, resources, inter-jot resources, todos (active and done), and URLs are emitted in document order with one parser-output object per matched occurrence; deduplication of these constructs is the consumer's responsibility.
6. Enforces left-boundary rules (§3.1) for free-standing constructs, and applies trim-then-validate right boundaries (§3.2): trailing trim-set runs are trimmed, and any other unconsumable character rejects the full candidate span.
7. Parses annotation suffixes on references, resource references, and todos (§6), enforces the category-before-tags ordering, and keeps edge annotations separate from the text's own tag and category lists.
8. Recognizes the todo done marker (`??.` form, §4.6) only when the `.` immediately follows the `??` sigil, and routes done todos to `done_todos` rather than `todos` in parser output (§8). The two lists share an identical object schema. On done todos only, post-scans the description for an optional completion timestamp (`.@<iso-date>`) and emits the parsed date on the todo's `completed` field, applying the same timezone-completion behavior as for `due`. The `.@<iso-date>` syntax MUST NOT be recognized in active todo descriptions or outside any todo span.
9. Parses line fragment, bookmark, and bookmark range specifiers (§4.9–§4.11), enforces their mutual exclusivity, normalizes line fragments to `(start, end)` pairs, and exposes them in parser output per §8. Specifiers MUST NOT appear on the locator prefix of an inter-jot resource reference.
10. Parses bookmark declarations (§4.8), enforces bookmark-name uniqueness within a text, and preserves bookmark name case.
11. Parses inter-jot resource references (§5), including annotation suffixes and specifiers on the resource portion.
12. Classifies error conditions per Part III and reports parse errors, validation errors, and resolution warnings through a diagnostics interface. Each diagnostic MUST expose at minimum the fields defined in §10.1 (`code`, `severity`, `message`, `loc`). Implementations MAY offer a strict mode that promotes validation errors and resolution warnings to fatal errors.

A tool MAY implement the optional hashtag extension (§7.9) without affecting conformance.

Renderer-side conformance — default output templates, host-format patterns, escaping, bookmark and inter-jot resource rendering — is defined separately in the Conformance section of `iotasyntaxis.org/renderer/latest`. The two surfaces are independent: a tool MAY conform to either, both, or different host-format subsets of either.

References

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, BCP 14, March 1997. URL https://datatracker.ietf.org/doc/html/rfc2119.