YINI: A lightweight configuration file format — clean, readable, structured.

Specification for the YINI Format

Version: 1.0.0-RC.4 **Date:** 2026-03

> **Note:** This specification of the YINI format may introduce changes that are not backward-compatible (see Section 14.2, "Versioning Strategy").

⇨ Table of Contents

Preface

YINI was designed with a simple idea in mind: configuration files should be easy for humans to write, read, and understand — without sacrificing structure or future flexibility.
It aims to remain minimal while still being expressive enough to support a wide range of configuration needs.
The name *YINI* originates from "Yet another INI", reflecting its inspiration from the traditional INI format.

That said, there are already many excellent configuration formats out there, and they are great at what they do. **YINI is not intended to replace existing formats** — it was created to fill a specific niche and, in part, out of personal curiosity and exploration.

The idea started as a personal project — a search for something a bit more readable than JSON, a bit more structured than INI, and a bit less surprising than YAML. It gradually evolved into a format that aims to be:

- Minimal, but expressive.
- Structured, but not rigid.
- Easy to hand-edit, and just formal enough to support tooling and validation.

YINI aims to embrace simplicity as a strength, offering just enough rules to stay consistent while remaining forgiving enough for practical use.

See A.1. Why was YINI created? for background.

While inspired by established formats such as INI, JSON, Python, Markdown, and YAML, YINI introduces its own principles: consistent grammar, clear typing, and readable structure — with

human readability and developer experience in mind.
One of YINI's key strengths is its intuitive approach to **nesting sections**, allowing structured configuration without relying on indentation rules or verbose syntax, while remaining visually clear and easy to scan — without the strict indentation rules or syntax overhead found in some other formats.

This specification defines YINI with care and clarity, aiming to serve both casual users and implementers looking for a clean, predictable format.

Some parts of the YINI specification have benefited from valuable feedback and insights shared by users in the broader community.

For more feedback details, see section D.2, *"Acknowledgments & Special Thanks"*, in the Rationale document.

Above all, YINI remains true to its founding goal: make configuration effortless — and maybe even enjoyable.

Table of Contents

Part II – Grammar and Literals

Part III – Validation, Implementation & Compatibility

Part IV – Examples and Appendices

1. Introduction

1.1. What is YINI?

YINI is a human-readable text format for representing structured information.
It is designed to be clear, predictable, and easy for humans to read and write, and is defined by a formal grammar that provides simplicity, flexibility, and a clear separation of concerns for configuration data.

Its syntax is inspired by widely used configuration formats such as INI and YAML, as well as general-purpose languages and data notations including JSON, C, and Python.
YINI aims to offer a consistent and intuitive structure that is easy to parse and edit by both humans and machines.

YINI is targeted at users who require a straightforward format for storing and organizing data—such as configuration files, application settings, and other general data storage—where human readability and ease of use are of primary importance.

YINI is flexible enough to support a wide range of use cases, from simple key-value pairs to deeply nested structured data, making it suitable for everything from basic application preferences to complex system configuration.

1.2. Purpose and Design Goals

1.2.1. The # Marker as a Comment Symbol

In earlier drafts of YINI (up to Beta 5), the `#` character was temporarily used as a section header marker, inspired by Markdown-style headers. However, based on feedback and concerns about clarity, expectations from other formats, and common usage across tools and communities, this decision was revised.

YINI now treats `#` as a comment symbol (instead of being used as a section marker), aligning with conventions found in formats like classic INI, Bash, YAML, and various scripting environments. This change improves predictability for users familiar with other configuration file styles. See A.4. Design Philosophy for background.

- The `#` starts a comment **only** when followed by a space or tab.
- **Note:** `##` is invalid, `# #` is valid as a comment.

1.2.2. Key Design Goals

The YINI format was created with the following key design goals in mind:

- **Simplicity:** YINI is designed to be as simple and intuitive as possible. The syntax is minimalistic yet expressive, with clear conventions for defining sections, keys, and values. **Prioritize clarity over cleverness.**
- **Human Readability:** A core principle of YINI is that it SHOULD feel natural and intuitive for humans to work with. Its structure is designed to be clear and predictable, minimizing unnecessary complexity so that configuration files are easy to read, understand, write, and maintain—even for non-programmers.
- **Flexibility:** While simple, YINI is designed to accommodate a variety of data structures, including primitive values (strings, numbers, booleans, nulls) and more complex ones like lists and nested sections.
- **Compatibility:** YINI is meant to be compatible with a variety of tools and libraries, ensuring that it can be easily integrated into different programming languages and ecosystems.

> It also allows for optional extensions, enabling future enhancements without breaking backward compatibility.

- **Extensibility:** The format is designed to be extendable, allowing for future features and syntax to be incorporated as needed, such as support for anchors, includes, or custom validation rules.
- **Deterministic parsing in strict mode:** Strict mode requires a single root section and prohibits ambiguous document endings.
-

1.3. Background and Intent

YINI was created to serve as a clean, minimal, and predictable configuration format that balances readability with structure. For a deeper look into the motivation and design philosophy, see Why YINI?.

1.4. Key Features

Note: Unless explicitly stated otherwise, YINI parsers are expected to operate in lenient (non-strict) mode by default. Strict mode is optional and intended for validation-intensive environments.

YINI aims to prioritize **human readability, clarity, and clean syntax**.

- **Clear Sectioning:** Sections are clearly delineated, allowing for organized groupings of related configuration data. Section headers use `^` as the primary marker, with `<` and `§` supported as alternatives.

- **Flexible Data Types:** YINI supports a variety of data types, including strings, numbers, booleans, nulls, and lists. This flexibility makes it suitable for both simple and complex configuration needs.
    - **Inline objects & lists** — expressive nested data using `{}` and `[]` without indentation sensitivity.
- **Type Inference:** There is no need to declare types explicitly — the parser determines the value type by how it is written (e.g., quotes, brackets, keywords).
- **Commenting and Documentation:** YINI allows for inline comments, enabling users to document their configuration files directly. This enhances the human-readable nature of the format and makes it easier for teams to collaborate on configuration management.
- **Multi-line and Nested Data:** The format supports multi-line strings and nested sections, providing the ability to express more complex configurations while maintaining readability.
- **Clear End of Document:** YINI supports (optional in both lenient/strict-mode) a clear document terminator marker (`/END`).

1.5. Terminology

The following key terms are used consistently throughout this specification. Understanding these terms will help interpret YINI's grammar, structure, and semantics.

Term	Definition
Classic String (C-String)	A string prefixed with `C` that supports escape sequences like `\n`, `\t`, etc.
Configuration	A structured set of members and sections that defines settings or data in a YINI document or file.
Document Terminator	A special line (`/END`) that explicitly marks the end of a YINI document (optional in both lenient/strict-mode).
Hyper String (H-String)	A multi-line string prefixed with `H` that normalizes whitespace and trims edges.
Identifier	The name of a key or section. Can be a simple word (e.g., `title`) or a **backticked identifier** (wrapped in backticks).
Key	An identifier on the left side of an assignment (`=`). Keys MUST be unique within their section (and depth/level).
Lenient Mode	This is the default parsing mode in YINI.

Term	Definition
List	Lists, also known as Arrays, are a compound value type consisting of zero or more comma-separated items enclosed in square brackets and assigned using `=`.
Member	A key-value pair entry, such as `key = value`, representing a single entry within a section or root.
Raw String (R-String)	A string literal that does not interpret escape sequences **(default type)**.
Section	A logical grouping of members, introduced by a header using a section marker such as `^`, `<`, or `§`.
Section Marker	A special character used to denote a new section header. Supported section markers are `^`, `<`, and `§`; `^` is the primary and recommended marker.
Strict Mode	An optional parsing mode where all structural and validation rules (incl. the document terminator) are strictly enforced. Not the default.
Triple-Quoted String	A string enclosed in `""" ... """` that may span multiple lines and, by default, preserves all content (including whitespace and line breaks) exactly; when prefixed with `c`, it recognizes standard escape sequences.
Value	The data assigned to a key. Can be of type string, number, boolean, null, or list.
YINI document	A complete YINI configuration. In this specification, "document" and "file" mean the same thing.
YINI file	A complete YINI configuration. In this specification, "file" and "document" mean the same thing.
YINI	YINI is a human-readable configuration format blending INI-style sections with modern typing and structure.

2. File Structure

The structure of a YINI file is designed to be simple, clear, and highly readable. The file structure determines how data is organized, encoded, and presented. Below are the key elements of the file structure.

2.1. File Encoding

YINI files MUST be encoded in **UTF-8**. This encoding ensures compatibility with most systems and applications, providing a consistent method for interpreting characters.

- **Mandatory Encoding:** All YINI files SHOULD be encoded using UTF-8 without a Byte Order Mark (BOM). This guarantees that the file content is universally readable across different platforms.
- **Character Set:** YINI files MUST use Unicode. Control characters and other non-printable characters SHOULD be avoided, except for spaces, tabs, and newlines. Other special characters may be included using escape sequences, or by placing them inside Raw or Triple-quoted strings.

Exactly which control characters see more in section 3.2, "Whitespace and Indentation".

2.2. File Extension

YINI files SHOULD use the `.yini` file extension. This extension helps clearly identify the file type and ensures proper handling by tools and parsers designed for the YINI format.

2.3. Optional Shebang (`#!`)

For Unix-based systems, a shebang (#!) is commonly used in script files to specify the interpreter. This feature is supported in YINI files, making it possible to use YINI documents as configuration files for scripts or command-line applications.

How to Use the Shebang:

- The **very first line** of the document may optionally begin with a Unix-style **shebang** (`#!`), which specifies the interpreter for the script.
- If present, the shebang line will be ignored by the YINI parser.

Here's an example of a YINI document with a shebang that could be used in a Unix-based scripting environment:

```
#!/usr/bin/env yini

^ Config
key = value
```

2.4. YINI Marker (`@yini`)

The optional YINI marker (`@yini`) MAY be used and is RECOMMENDED for clarity and identification. If present, it MUST be added at the very top of a YINI file (if present, it MUST appear after any shebang line, or comments).

The marker is case-insensitive: `@yini` , `@YINI` , and `@Yini` are all valid.

Its main purpose is to clearly indicate — both to humans and to programs — what format the file is in.

Although YINI files typically have the `.yini` extension, the filename is not always visible (for example, when files are embedded, or copied as snippets, etc.). The marker line provides immediate identification regardless of context.

- It also helps clarify the file format when files are opened in editors or included in bug reports.

Example:

```
@YINI

^ Config
key = value
```

Example (with shebang):

```
#!/usr/bin/env yini

@yini

^ Config
key = value
```

3. Syntax Overview

The syntax of YINI is designed to be minimalistic and human-readable while offering enough flexibility for structured data representation. This section provides an overview of the key syntax rules for YINI files.

3.1. General Syntax Rules

YINI files consist of a series of **sections, members** (key-value pairs), and optional **comments**. The following rules define the basic structure of a valid YINI file:

Whitespace: Whitespace (spaces and newlines) is used to separate elements in the file. Tabs do not contribute to the logical structure, except in section headers, where spacing (tabs or spaces) is required between the section marker and the section name. Other than this tabs are totally ignored, though tabs or multiple spaces may be used to make it clearer for humans to read.

Exactly which control characters see more in section 3.2, "Whitespace and Indentation".

Sections: YINI files support sections, which groups related members. A section begins with a section header, marked by one of the allowed characters (commonly `^`), and then at least one space or tab, followed by the section name. Before a section header there may exist indentation and spacing for human readability.

Example of a section:

```
^ SectionName
key = value
```

Keys and Values (Members): The basic unit of YINI is a key-value pair, called a Member. A key and its associated value are separated by an equal sign (=). Any number of spaces or tabs may appear before or after the `=`.

Example:

```
key = value
```

Comments: YINI primarly follows C-style commenting rules using `//` and `/* ... */`. Alternative inline `#` comments, and full-line `;` comments are supported too. These are ignored by parsers and exist solely for human readability.

Example:

```
; Full-line comment.
key1 = "Banana"

key2 = "Mango"  # This is an inline comment.
key3 = "Peach"  // This is also an inline comment.

/*
  This is a block comment.
  It spans several lines.
*/
```

3.2. Whitespace and Indentation

While YINI is not indentation-sensitive, the following whitespace `<WS>` behaviors are defined:

- Newlines (`<NL>`) may be either Unix/Linux-style (`LF` , U+000A), Windows-style (`CRLF` , U+000D U+000A), or (`CR` , U+000D).
- Tabs (`<TAB>` , U+0009) and spaces (`<SPACE>` , U+0020) are ignored outside of strings and section headers.
- Indentation using whitespace is allowed purely for visual clarity — it has no effect on parsing or structure.
- Note: In the context of strings, the term `<Unicode-WS>` is used to refer to a broader range of Unicode whitespace characters, beyond just tab and space. For a complete table, see more in section 16.7, "📃 Unicode Whitespace Characters".

3.3. Comments

YINI supports **three types of comments**:

Comment Type	Prefix	Position
Inline comment	`//` or `#`	At end of line
Block comment	`/* ... */`	Anywhere (multi-line)
Full-line comment	`;`	Start of line ONLY

While both `//` and `#` are valid for inline comments, it is recommended to use **only one style per file** to maintain clarity and consistency for human readers.

Rule Summary — `#` Interpretation

- `#` followed by a space or tab → **comment**.
- `#` followed by anything else → **hex literal**.

See also Section 3.6, "Disable Line", for a related mechanism used to deactivate valid lines of configuration.

3.3.1. Inline Comments

YINI supports two syntaxes for inline comments.

- **Double slash** `//` comments are the default:

  ```
  // This is a single-line comment.
  ```

- **Hash `#` comments** are supported too — MUST be followed by **at least one space or tab** to be recognized:

  ```
  # This is also a single-line comment.
  ```

This rule is a deliberate compromise to avoid ambiguity with hex-like values (e.g., `#FF0033`) commonly used in domains such as styling or color settings.

✅ **Valid `#` comments:**

- `# This is a comment`
- `# Also valid`
- `#\tTabbed too`

❌ **Invalid `#` comments (not treated as comments):**

- `#FF9900` — Interpreted as a hex value.

- `#Invalid comment` — No space or tab after `#`.
- `##` — Not recognized as a valid comment, no space/tab follows the `#`.

3.3.2. Multi-line Block Comments

Multi-line (block) Comments.

Begin with `/*` and end with `*/`. These comments may span multiple lines.

```
/*
  This is a multi-line comment.
  It can span multiple lines.
*/
```

3.3.3. Full-line Comments

A full-line comment starts with a semicolon `;` and occupies the entire line.

```
; This is a full-line comment.
```

Note: Block comments may appear between any two members, or on their own lines. They cannot appear inside a quoted string or within an identifier. Comments are ignored by parsers and exist solely for human readability.

3.4. Identifiers

Identifiers are names used for keys (in members) and sections (section headers).

An ***identifier*** can be one of two forms below:

- **Form 1: Identifier of Simple Form:**
  - Keys MUST be non-empty.
  - Keys are case-sensitive (`Title` and `title` are different).
  - Can only contain letters (a-z or A-Z), digits (0-9) and underscores `_`.
  - Must begin with a letter or an underscore `_`.
  - Note: Cannot contain hyphens (`-`) or periods (`.`).

  Example:

  ```
  name
  ```

- **Form 2: Backticked Identifier:**

- The identifier MUST start and end with a backtick <code>`</code>.
- It MUST be on a single line, and MUST not contain tabs or newlines unless using escape codes (`\n` , `\r` , `\t` , etc.).
- Ordinary spaces are allowed.
- Special control characters (U+0000–U+001F) MUST be escaped and MUST NOT appear as raw characters.
- **Note:** A backticked identifier **may be empty** <code>``</code> (to conform with the JSON empty key <code>""</code>), though permitted, its use is discouraged.

Example:

```
`Description of Project`
`Amanda's Project`
```

3.5 Document Terminator

Note: The document terminator is only optional in lenient (non-strict) mode. Lenient mode is the default parser mode.

A YINI document **MAY end with a terminator line**.

The document terminator explicitly marks the end of the configuration content and prevents ambiguity about whether the document was fully read.

In other words, it acts as a clear, unambiguous signal that the document is complete — without relying on end-of-file (EOF) and leaving parsers **to "guess" whether the final section or member was fully parsed**.

The **default and recommended** terminator is:

```
/END
```

This line is **not case-sensitive** (`/end` , `/End` , etc. are also valid). Only **whitespaces or comments** may appear after the terminator.

It is recommended that there are no leading spaces or tabs, on the same line as the terminator. If there are comments after the marker, these SHOULD be ignored.

3.6. Ignore / Disable Line

A line that begins with a **double dash** (`--`) is treated as a **disabled line** and will be completely ignored by the YINI parser. Everything after `--` until the end of the line (`<NL>`) is disregarded — including any comments or syntactically valid members.

This mechanism is similar to a comment, but serves a **distinct purpose**: disabling or temporarily excluding valid configuration lines without deleting them.

Example 1:

```
// The next line is ignored — even though it's valid syntax.
--key = "Apples"
```

Example 2:

```
^ Server
host = 'localhost'
port = 8080

--^ Features
--login = true
--notifications = false
```

Purpose and Usage:

- Disabled lines are intended for **temporary deactivation** of configuration content.
- Unlike comments, which are typically used for documentation or explanatory notes, **disabled lines are structurally valid syntax that is intentionally skipped**.
- This allows developers or users to **temporarily toggle** parts of the configuration without removing them.

Visual Distinction (Editor Highlighting): A recommendation is that syntax highlighters use a **distinct color** or style for disabled lines — different from comments — to improve visual clarity. This helps distinguish between:

- **Comments** (notes for humans), and
- **Disabled lines** (intended-to-be-ignored syntax).

Technical Note: Disabled lines are **functionally identical to comments** from the parser's point of view: they are ignored. However, their **semantic intent** is different — they represent *temporarily inactive* configuration logic rather than annotations.

4. Keys and Values

YINI represents configuration and structured data through a series of ***members***, each of which is a key-value pair. This section defines the syntax and rules for keys and values, including allowed characters, data types, quoting, and related behaviors.

4.1. Key Naming Rules

A **key** is an identifier used to reference a specific value in a member (a `key = value` pair) within a YINI file.

- Keys MUST be valid identifiers — either a **simple form** or a **backticked identifier** (a backticked string). (See Section 3.4: Identifiers.)
- Keys MUST be **unique** within the same section and nesting level.
- Keys are assigned values using `=` operator.

Examples:

```
username = "admin"
user_id = 12345
```

4.2. Value Types (Simple, Compound, Special)

> YINI infers the type of each value automatically based on its syntax. There is no need to declare types explicitly — the parser determines the value type by how it is written (e.g., quotes, brackets, keywords).

A YINI ***value*** can be of one of the following three groups of native/built-in types:

- **Simple/scalar types:**
  - String
  - Number
  - Boolean
- **Compound types:**
  - Object (similar to a map) — a sequence of key-value pairs (members), separated by commas
  - List (also known as Arrays) — a sequence of values, separated by commas
- **Special type:**
  - NULL

Note: Currently, YINI types map 1-to-1 to native JSON types. Constructs and objects like date-time, **SHOULD currently be expressed as strings**. See more in 10.1.3, "Date-time Type".

4.3. Type Rules

This section describes how values (on the right-hand side of `=`) are interpreted based on their syntax.

Note: In YINI, values are assigned using `=`. The colon (`:`) is not an assignment operator and MUST not be used to define members or lists.

Strings

If the value is meant to be a string, it MUST be quoted — either with single quotes (`'`), double quotes (`"`), or triple quotes (`"""`).

ONLY when quoted, (even in non-strict mode) the value is considered to be of type **String**.

Numbers

- A sequence of digits **without a period** (`.`) is treated as a **Number** (integer).
- A sequence of digits **with a period** (`.`) is treated as a **Number** (floating-point / float).

Booleans

If the value matches any of the following keywords `true`, `false`, `on`, `off`, `yes`, or `no` (case-insensitive) — it is interpreted as a **Boolean**.

Lists

If the value is a **bracketed sequence** (`[ ... ]`) of values (any of the supported YINI types: Numbers, Strings, Booleans, Lists, Nulls), separated by commas — the entire value is treated as a **List**.

Null

If the value is the keyword `null` (case-insensitive), or if the value is missing (e.g., blank after `=`, or a blank item within a bracketed sequence) — it is treated as **Null**.

Summary:

Value form examples	Meaning
`'something'`, `"something"`, or `"""something"""`	**String**
`123`	**Number** (integer)
`3.1415`	**Number** (float)
`true`, `FALSE`, `On`, `off`, `YES`, `No` *(any casing)*	**Boolean**
`null` *(any casing)*, *(blank)*	**Null**
(Any value such as unquoted words that are not booleans, numbers, or null)	**ERROR**

Example:

```
name = 'Sarosh'                // String
BTW = "BTW means By The Way."  // String
age = 42                       // Number
e = 2.718                      // Number
isActive = True                // Boolean
nightMode = OFF                // Boolean
nothing = Null                 // Null
alsoNothing =                  // Null (blank, not recommended)
scores = [1, 2, 3]             // List
mixed = ["Arial", 12, true]    // List (mixed types)
```

5. Section Headers

Sections in YINI are used to organize related members (key-value pairs) into logical groups. This allows for improved readability, structure, and modularity within configuration files.

5.1. Syntax

A ***section header*** starts a new logical grouping of members. Section headers MUST ALWAYS appear on their own line.

```
// A section header with a simple identifier.
^ SectionName

// A section header with a backticked identifier.
^ `Section name`

// Backticked identifiers can include other special symbols too.
^ `Section-name`
```

- A section header begins with a **section marker**, it is recommended (but not required) that it is followed by **one or more whitespace (space or tab) characters**, then the section name.
- The section name MUST be a **valid identifier**, either a **simple identifier** or a **backticked identifier** (enclosed in backticks).
- **Backticks (backticked identifier) are only required** when the identifier contains spaces, punctuation, or other special characters.
- The section header ends at the newline. There may follow a comment on the same line, but this will get ignored by the parser.

```
^ UserSettings
username = "alice"
theme = "dark"
```

5.2. Section Markers (`^` , `<` , `§`)

YINI allows a limited set of ***section markers*** to identify section headers. These markers help visually and semantically distinguish section starts from key-value members or comments.

Supported markers:

- `^` (Primary and recommended section marker, within the 7-bit ASCII range for maximum compatibility.)
- `<` (Alternative section marker (if `^` causes issues somehow), within the 7-bit ASCII range for maximum compatibility.)
- `§` (Alternative section marker; supported, but less portable than ASCII-only markers.)

Note: The `€` character is no longer supported as a valid section marker in YINI.

When using a repeated marker to indicate nesting, a maximum of six (`6`) repeated markers is allowed.

That is, the following denote nesting levels 1–6:

```
^       ← level 1
^^      ← level 2
^^^     ← level 3
^^^^    ← level 4
^^^^^   ← level 5
^^^^^^  ← level 6
```

Using seven or more of the same marker in succession (e.g. `^^^^^^^`) is invalid. To represent nesting deeper than level 6, switch to the **numeric shorthand section header** syntax (see Section 5.3.1).

5.3. Nested Sections

To place a section under another (i.e., to nest sections), repeat the section marker character (this technique with repeating characters is inspired by Markdown) without skipping any intermediate levels. Each additional repetition indicates one more nesting level. However, when moving to a less-nested (closer to section header) level, you may drop directly to any smaller level.

- Section heading markers (`^` , `<` , or `§`) may only be repeated up to six times — to level 6 (maximum).
- Beyond level 6, the numeric shorthand section MUST be used (see section 5.3.1).
- **Going deeper (increase nesting):** Must increment exactly one level at a time. E.g.: `^^` → `^^^` but not `^^` → `^^^^` .
- **Going shallower (decrease nesting):** May drop directly to any previous level. E.g.: `^9` → `^^` or `^9` → `^` .

- Optionally the short-hand section notation may be used for any levels and that notation is the only way to define levels 7 or beyond.

```
^ Prefs
    ^^ Section
        ^^^ SubSection
```

Optionally, indentation may be omitted:

```
^ Prefs
^^ Section
^^^ SubSection
```

```
^ Level1
^^^ Level3  // ❌ Invalid: cannot skip Level2
```

💡 While YINI does not require indentation before section headers, it is recommended to visually indent section headers according to their nesting level — this make for for better human readability.

✅ **Example of valid section nesting:**

```
^ Section 1         // Main section 1 (depth 1)
^^ Section 1.1      // Sub-section of 1 (depth 2)
^^^ Section 1.1.1   // Sub-section of 1.1 (depth 3)

^^ Section 1.2      // Sub-section of 1 (depth 2)

^ Section 2         // Main section 2 (depth 1)
^^ Section 2.1      // Sub-section of 2 (depth 2)
^^^ Section 2.1.1   // Sub-section of 2.1 (depth 3)

^ Section 3         // Main section 3 (depth 1)
```

5.3.1. Short-hand Section Heading

Short-hand Section Headings: Numeric shorthand is required for nesting levels greater than 6. The syntax is `<marker><n>`, where `<marker>` is one of the allowed section marker characters (`^`, `<`, etc.) and `<n>` is an integer ≥ 1 indicating the nesting level. Levels 1 - 6 is recomended to use repeated markers `^`, `^^`, `^^^`, etc. (Using the shorthand is optionally valid for levels 1–6 as well, though repeated markers are RECOMMENDED but not required).

For example:

- To go from depth 6 to depth 7: write `^7`.

- To go from depth 7 to depth 8: write `^8`.
- To go from depth 8 to depth 9: write `^9`.
- And so on...

This prevents arbitrarily long runs of the same marker. When ascending (moving to a shallower level), you may skip multiple levels at once (e.g., from `^9` back to `^^`).

Examples:

```
^      Level1      # One caret
^^     Level2      # Two carets   → depth 2
^^^    Level3      # Three carets → depth 3
^^^^   Level4      # Four carets  → depth 4
^^^^^  Level5      # Five carets  → depth 5
^^^^^^ Level6      # Six carets   → depth 6
^7     Level7      # Shorthand    → depth 7
^8     Level8      # Shorthand    → depth 8
^9     Level9      # Shorthand    → depth 9
^10    Level10     # Shorthand    → depth 10

^     BackTo1      # Going back to level 1 (allowed)
^^    BackTo2      # Going back to level 2 (allowed)
```

6. String Literals

Prefix Glossary Table:

Prefix letters are **case-insensitive**, e.g. lowercase `h` behaves identically to `H`.

Prefix	Type Name	Behavior Summary
none	Raw String	Raw (default) if no prefix is used
R *(optional and has no functional effect)*	Raw String	No escapes, preserves text exactly (default)
H	Hyper String	Multi-line, trims & normalizes whitespace
C	Classic String	Supports escape sequences like `\n`, `\t`
Note: Prefix R is optional and has no functional effect — raw strings are the default.		

YINI has four types of string literals — Raw, Classic, Hyper, and Triple-quoted — each designed to help express text clearly and appropriately in different situations, whether for escape handling, whitespace normalization, or multi-line content.

String literals in YINI **MUST be enclosed** in either single quotes `'` or double quotes `"`, or optionally in triple double quotes `"""`. You MAY use whichever is preferred or most appropriate for the context.

Note: If a string is not quoted, it's not a string — period.

YINI supports four types of string literals, distinguished by an optional prefix character before the opening quote `'` or `"` (**except for triple-quoted strings** `"""`, which do not support any prefix). YINI supports multi-line strings via Hyper Strings or triple-quoted strings.

If no prefix is used, the string is treated as a **Raw string literal** by default.

Note: Triple-quoted strings (`"""`) only support the `R` and `C` prefixes. If no prefix is given, a Triple-quoted string is treated as raw.

Rules and Behavior for Strings:

- All string literals **MUST start and finish on the same line**, except for **H-Strings** (see section 6.4.) and **Triple-Quoted Strings** (see section 6.3.), which can span multiple lines.
- Multiple string literals can be **concatenated** to create longer strings (see section 6.6, "String Concatenation").

String Overview

Type	Quotes Used	Multi-Line	Escapes	Trims Whitespace	Description	Similar To
Raw String	`'...'` or `"..."`	❌ No	❌ No	❌ No	Simple 1-line literal	Raw literal strings
Classic String (C)	`C'...'` or `c"..."`	❌ No	✅ Yes	❌ No	1-line with escapes	C / JSON strings
Hyper String (H)	`H'...'` or `h"..."`	✅ Yes	❌ No	✅ Yes	Clean multi-line text, trimmed	HTML text flow
Triple-Quoted (Raw)	`"""..."""`	✅ Yes	❌ No	❌ No	Multi-line raw text	Python raw triple-quote

Type	Quotes Used	Multi-Line	Escapes	Trims Whitespace	Description	Similar To
C-Triple-Quoted	`c"""..."""` or `c"""..."""`	✅ Yes	✅ Yes	❌ No	Multi-line with escapes	Python triple-quote

6.1. Raw Strings (R-Strings)

In (Raw) strings, the backslash (`\`) is treated as a literal character — **it is "just a backslash"**. This means escape sequences are not interpreted, and most special characters can be included directly.

However, Raw strings **cannot contain newlines**, as they MUST appear on a single line.

For multi-line Raw strings, see Triple-quoted string literals.

Raw strings are particularly suitable for representing file paths and other literal text.

> myPath = "C:\Users\John Smith" // Raw string or myPath = '/home/Leila Häkkinen' or myPath = '/Users/kim-lee'

Raw String Prefix

Any string enclosed in quotes (single `'` or double `"`) can be prefixed with either `R` or `r` explicitly to denote it as a Raw-String, but Prefixing Raw strings is not required as strings are Raw as standard.

6.2. Classic Strings (C-Strings)

YINI also supports standard string literals, referred to as **Classic Strings**, or **C-Strings** for short. These strings are prefixed with either `C` or `c` .

C-Strings support all common escape sequences, including those for newlines, tabs, form feeds, and more. Thus, all special control characters (U+0000–U+001F), except for space and tab, MUST be written using escape sequences — they cannot appear directly in Classic Strings.

Classic strings MUST begin and end on the same line.

> myText = c"This is a newline \n and this is a tab \t character."

6.2.1. Escape Characters

Escape sequences are supported only in Classic Strings (C-Strings), which MUST be enclosed in quotes and prefixed with `C` (or `c`).

Full List: Escape Sequences: (case-sensitive, only valid in C-Strings and C-Triple-Quoted Strings)

- `\\` — Backslash

- `\'` — Single Quote
- `\"` — Double Quote
- `\/` — Normal Slash (yields a plain `/` as JSON)
- `\0` — Null Byte
- `\?` — Literal question mark (for C/C++ compatibility)
- `\a` — Alert (bell)- ASCII 7
- `\b` — Backspace - ASCII 8
- `\f` — Form Feed - ASCII 12
- `\n` — Newline - ASCII 10
- `\r` — Carriage Return - ASCII 13
- `\t` — Tab - ASCII 9
- `\v` — Vertical tab - ASCII 11
- `\xhh` — Hex byte (2-digit)
- `\uhhhh` — Unicode character (4-digit hex) (UTF-16)
- `\Uhhhhhhhh` — Unicode character (8-digit hex) (UTF-32)
- `\oOOO` — Octal value (up to 3 digits, valid range `\o0 - \o377`)
  - Equivalent to `\OOO` in C/C++
  - `\o0` has same effect as `\0`

Where:

- `h` = *Hexadecimal digit* `0-9`, `a-f`, or `A-F`.
- `O` = *Octal digit* `0-7`.

Invalid Escapes

Invalid escape sequences (e.g. `\z` or `\o378`) MUST result in a parse error unless explicitly allowed by a custom extension or parser configuration.

6.3. Hyper Strings (H-Strings)

YINI supports a special kind of string literal called a **Hyper String**, or **H-String** for short. These strings are prefixed with either `H` or `h`.

Like raw strings, Hyper Strings treat backslashes as literal characters — escape sequences are not interpreted.

Hyper Strings are designed to be **multi-line friendly** and **visually readable**, especially for long text blocks:

- `<NL>` refers to newline/linebreak of in combination of `CR`, `LF`, or `CRLF`. `<Unicode-WS>` includes all relevant Unicode whitespace characters: the categories `Zs` (space separators), `Zl` (line separators), `Zp` (paragraph separators), and selected `Cc` (control characters), primarily

from the C0 range (`U+0000–U+001F`). For a complete table, see more in section 16.7, " Unicode Whitespace Characters".

- Hyper Strings **can span multiple lines**, and indentation using `<Unicode-WS>` is allowed to improve human readability.
- Multiple consecutive newlines (`<NL>`) and/or whitespace (`<Unicode-WS>`) are normalized into a **single space** (`U+0020`).
- Leading and trailing `<NL>` and/or `<Unicode-WS>` are trimmed.
- For the complete set of characters in <Unicode-WS>, refer to section 16.7 — Unicode Whitespace Characters.

Hyper Strings behave similarly to how text is rendered in HTML: extra spacing and line breaks are reduced to clean, flowing text.

The following:

```
H"My name is
  John Doe,
  and this is a test string."
```

Will result in:

```
My name is John Doe, and this is a test string.
```

6.4. Triple-Quoted Strings

A **Triple-Quoted String** is a string literal that:

- **Begins and ends** with three double-quote characters: `"""` .
- **May span multiple lines**, including embedded newline characters.
- **May contain any characters**, including quotes (`"`) and double quotes (`""`), **except** an unescaped sequence of three double quotes (`"""`), which ends the string.
- **Preserves all content exactly as written**, including whitespace and line breaks (new lines) — unless escape sequences are enabled by prefixing the string with `C` or `c` (see below).
- **Ends at the first unescaped** sequence of three double quotes (`"""`).
- **Does not support `H` prefixes**.

By default, Triple-quoted strings are treated as **Raw** — escape sequences are not interpreted. To explicitly indicate that a Triple-quoted string is raw, a prefix `R` (or `r`) may optionally be used. This prefix is purely **syntactic sugar** and does not affect its default behavior.

If **prefixed with `C` or `c`** , the string supports escape sequences, just like Classic Strings (C-Strings). This includes support for: `\n` , `\t` , `\\` , `\"` , `\xhh` , `\u1234` , `\o123` , etc.

Examples

Raw (default) triple-quoted strings:

```
"""This is a multiline
string that spans
three lines."""

"""He said, "hello" and left."""

"""You can use double quotes (") inside."""
```

C-Triple-quoted strings (with escapes enabled):

```
C"""This spans multiple lines with a tab\tand newline\n"""

C"""Quotes inside: "double" and 'single'"""
```

Note: Triple-quoted strings always preserve their contents exactly — including all whitespace and line breaks (new lines) — unless prefixed with `C` (or `c`), in which case escape sequences are interpreted.

6.5. String Types Summary

Summary

String Type	Enclosed In	Multi-Line	Escape Sequences	Trims Whitespace	Notes	Behavior Hint
Raw Strings (default)	`' '` or `" "`	❌ No	❌ No	❌ No	Ideal for file paths and literal text	Simple raw, 1-line, no escapes
Classic Strings (C-Strings)	`C' '` or `c" "`	❌ No	✅ Yes	❌ No	Standard escaped strings	Behaves like C/JSON strings
Hyper Strings (H-Strings)	`H' '` or `h" "`	✅ Yes	❌ No	✅ Yes	Readable multi-line text, whitespace is normalized and trimmed	Behaves like HTML text flow

String Type	Enclosed In	Multi-Line	Escape Sequences	Trims Whitespace	Notes	Behavior Hint
Triple-Quoted Strings	`"""` `"""`	✅ Yes	❌ No	❌ No	Multi-line literal string, Raw by default	Raw multiline, no escapes
C-Triple-Quoted Strings	`c"""` `"""` or `c"""` `"""`	✅ Yes	✅ Yes	❌ No	Multi-line with escapes, like Classic but multiline	Like Python triple strings

6.6. String Concatenation

Strings in YINI can be **concatenated** using the plus sign `+`. This operator joins two or more string literals into a single combined string. Any number of strings can be chained together using this method.

Example:

```
greeting = "Hi, " + "hello " + "there"
```

The result of the above will be equivalent to:

```
greeting = "Hi, hello there"
```

Concatenation is supported between all string types, but mixing different types (e.g., Raw + Classic) is discouraged unless necessary for special use cases.

6.7. String Type Mixing

Concatenation of string literals of different types (e.g., Raw + Classic, Classic + Hyper) is **permitted**, but generally **discouraged**. This flexibility exists to support **rare or advanced use cases** where such combinations may be helpful or necessary.

Engines SHOULD handle mixed-type concatenations correctly, but authors are encouraged to use consistent string types within concatenations to ensure clarity and predictable behavior.

7. Number Literals

7.1. Numbers

YINI supports both integer and floating-point literals. Numbers may be signed and written in decimal notation.

- **Integers:** A sequence of digits, optionally prefixed with + or -.
- **Floats:** Must include a decimal point (`.`) and optional exponent (`e` or `E`).

Examples:

```
age = 42
pi = 3.14159
negative = -12
scientific = 1.23e4
```

7.2. Exponent Format

Exponent notation uses the format:

```
<base>e<sign><exponent>
```

Where:

- `<base>` is any integer number.
- `<sign>` can be `+`, `-`, or blank (positive).
- `<exponent>` is any non-negative number.

Example:

```
3e4 // Is same as 3 × 10⁴ = 30000
```

7.3. Number Formats

In addition to standard decimal numbers (base-10), YINI supports other number base literals as well.

Note, binary and hexadecimal values also allow **alternative notations** for convenience and readability.

Number Format (case-insensitive)	Alternative Format	Description	Base	Notes
`3e4`	-	Exponent notation number	10-base	Result: `3 × 10⁴`

Number Format (case-insensitive)	Alternative Format	Description	Base	Notes
`0b1010`	`%1010`	Binary number	2-base	Digits: `0` and `1` only
`0o7477`	-	Octal number	8-base	Digits: `0` – `7`
`0z2EX9`	`0z2BA9`	Duodecimal (dozenal)	12-base	`X` = `A` = 10, `E` = `B` = 11 (case-insensitive)
`0xF390`	`#F390`	Hexadecimal number	16-base	`0` – `9`, `a` – `f` (or `A` – `F`) = 10–15

Note: All prefix-based number formats in YINI are case-insensitive. For example, `0xF390`, `0XF390`, `0xf390`, `0Xf390`, and `#f390` are all valid hexadecimal literals.

8. Boolean and Null Literals

8.1. Booleans

Booleans in a `YINI` document can be following literals (NON CASE-SENSITIVE):

- Treated as **TRUE** (by the engine):
  - `true`
  - `yes`
  - `on`
- Treated as **FALSE** (by the engine):
  - `false`
  - `no`
  - `off`

The engine SHOULD convert the literal value to the corresponding Boolean value in the host language.

8.2. Null Literal

Value/literal `NULL` (NON CASE-SENSITIVE).

- Empty or missing value in section-top-level key-value pair (member outside any list or object), is treated as NULL in lenient-mode, error in strict-mode. If written `key =` with nothing after `=`, that member's value is `null` (lenient only; strict mode requires explicitly `key = null`).

- Note: At top level (outside any `[ ]` or `{ }`), `key =` with nothing after `=` → `key = null` in lenient mode; in strict mode that is a syntax error unless you write `key = null` explicitly.

  Invalid Examples (both strict and lenient):

  ```
  ^ Section1
  key = { a = }    # ❌ Missing value within object.

  ^ Section2
  key = { a = , }  # ❌ Comma without preceding value.
  ```

  👆 An isolated comma or missing value is never interpreted as `Null` inside `{ }`.

Examples:

```
^ Section1
# In lenient-mode:
key1 =                    # ✅ Lenient: key1 → null
key2 = [1, 2, ]           # ✅ Lenient: [1, 2]
key3 = { a = 1, b = 2, }  # ✅ Lenient: {a: 1, b: 2}

^ Section2
# In Strict-mode:
key1 =                    # ❌ Error: Missing value
key2 = [1, 2, ]           # ❌ Error: Stray trailing comma
key3 = { a = 1, b = 2, }  # ❌ Error: Stray trailing comma
```

9. Object Literals

9.1. Objects (using { and })

YINI supports inline objects as a value type, allowing zero or more key–value pairs to be nested inside braces `{` and `}`. An inline object behaves like a map or dictionary: each entry inside `{ ... }` is a YINI `key: value` definition, separated by commas. Objects may nest arbitrarily (an object value can itself contain another `{ ... }`).

An empty object `{ }` is allowed as well (both in lenient and strict-mode).

An object in YINI have the following form:

```
<identifier> = { <key1>: <value1>, <key2>: <value2>, ... }
```

- The **key** in each member follows the same rules as any YINI identifier. The **value** in each member may be:
  - A string (quoted with `'...'`).

- A number (integer or float).
- A boolean (true/false/on/off).
- A list (`[ ... ]`).
- Another inline object (`{ ... }`).
- An empty object `{ }` is allowed as well (both in lenient/strict-mode).

Note: The equal sign `=` is never used between keys and values inside an object literal, always use `:` in objects.

The following rules apply to objects in YINI:

- Begins with `{` and ends with `}`.
- Between `{` and `}`, write zero or more members (including no members at all is allowed both in lenient and strict mode) of the form `key: value` (definition), separated by commas.
- Optionally and only in lenient-mode, after a value allow a trailing comma before the closing `}`
- Whitespace (spaces, tabs, newlines) is ignored except inside quoted strings.
- Comments (e.g. `// ...` or `# ...`) may appear anywhere whitespace is allowed.

💡 **In lenient-mode only**, a trailing comma after the last member is permitted and ignored (no `Null` member is added). **In strict mode**, trailing commas result in a parse error.

Grammar rule:

```
objectMemberList
  : objectMember ( COMMA NL* objectMember )* ( COMMA )?
  | empty_object NL*
  ;

objectMember
  : KEY WS? COLON NL* value
  ;
```

Example:

```
// ✅ Valid.
^ section
object = { member1: "value1", member2: "value2" }

// ❌ Invalid: Never use = inside object
obj = { a = 1, b = 2 }
```

More Examples:

```
# In lenient mode:
obj1 = { a: 1, b: 2, }     # → {a: 1, b: 2} ✅
```

```
obj2 = { a: 1,, b: 2 }     # → ❌ Parse error or warning (double comma)
obj3 = { , a: 1 }          # → ❌ Parse error (leading comma not allowed)
obj4 = { }                 # → {} ✅

# In strict mode:
obj1 = { a: 1, b: 2, }     # → ❌ Error: trailing comma not allowed in strict mode
obj4 = { }                 # → {} ✅
```

A Complete YINI Example:

```
// Section at level-1 starts here
^ System

// Withing the section "System", inline object "config" starts here
config = {
    name: "production",
      services: {
          web: {
              ports: [80, 443],
              routes: [
                  { path: "/", secure: true },
                  { path: "/api", secure: false }
              ]
          },
          database: {
              replicas: [
                  { host: "db1", role: "primary" },
                  { host: "db2", role: "secondary" }
              ]
        }
    }
}
```

10. List Literals

Lists in YINI correspond to what are called *Arrays* in JSON and serve the same purpose as arrays in many programming languages (e.g., in JavaScript).

YINI defines lists using bracketed list notation with `=` and square brackets `[ ]`, similar to JSON.

10.1. Bracketed Lists (using `=`)

A list is assigned to a key using the equals sign `=`, followed by square brackets `[ ]` containing zero or more comma-separated values.

Whitespace (spaces, tabs, and newlines) is allowed within the brackets.

```
list1 = ["value1", "value2", "value3"]
list2 = [100, 200, 300]
list3 = []  // An empty list.
```

For convenience, a trailing comma (`,`) may be optionally be included (only in lenient-mode).

```
// A list with THREE items.
list1 = [
  "a",
  "b",
  "c",  # Trailing comma here is ignored (parse error in strict-mode).
]

// A list with THREE items.
list2 = ["a", "b", "c", ]  # Trailing comma here is ignored.

// A list with FOUR items.
list3 = ["a", "b", "c", NULL]
```

> **Note:** A parser may optionally support strict and lenient modes, where trailing commas are either disallowed or accepted.

Syntax Rule

There MUST be **no newline** between the `=` and the opening bracket `[`, otherwise the value will be interpreted as `null`.

❌ Invalid:

```
list =
["item1", "item2"]  // Not a valid list!
```

✅ Valid:

```
list = ["item1", "item2"]
```

Nested Lists

Lists may contain other lists:

```
linkItems = [
        ["stylesheet", "css/general.css"],
        ["stylesheet", "css/themes.css"]
]
```

11. Advanced Constructs

11.1. Future / Reserved Features *(For Future Use)*

The following features are reserved for potential support in future versions of the YINI specification. They are **not currently active** in this version, but their syntax and keywords **are reserved**.

11.1.1. Anchors and Aliases

YINI may introduce a mechanism similar to YAML's anchors and aliases. These constructs would allow users to define reusable fragments within a configuration.

- **Anchors (`@`):** Or some other token, to assign a name to a key, section, or structure.
- **Aliases (`use`):** Reference a previously defined anchor using the use keyword.

11.1.2. Includes

YINI may support modular configuration files via external file inclusion.

- **Directive:** `@include "<path/to/file>"`
- **Description:** Imports the contents of another YINI file into the current one.
- **Usage:** Proposed as a preprocessor-style directive.

These features, while not implemented in this version, are reserved and MUST not be repurposed by user-defined syntax.

11.1.3. Date-time Type

Currently, the YINI types in the specification map 1-to-1 to native JSON types. With that said YINI does not currently support native date, time, or date-time types. All date and time values **SHOULD currently** be represented as strings.

Support for standardized date-time literals may be considered in a future version, once the core specification is stable.

12. Validation Rules

YINI enforces a set of validation rules to ensure the structure and content of files are consistent, unambiguous, and semantically correct. These rules fall into two main categories: **reserved syntax protections** and **well-formedness**. Validation ensures compatibility across implementations and minimizes user errors.

12.1. Reserved Syntax

Certain characters and keywords are **reserved** by the YINI specification for internal syntax or future use. Using them incorrectly may lead to a parse error or undefined behavior.

12.1.1. Reserved Characters

The following characters are reserved by the YINI syntax and MUST not be used improperly outside of their defined contexts. They may appear inside quoted strings or phrase identifiers but are otherwise restricted:

Character	Usage Context	Description
`=`	Assignment	Assign to key at root/section level
`^`	Section header	Used to denote section start
`:`	Define a property inside an object	Defines a value for a key inside an inline object
`,`	Item separator	Used in lists
`<`	Section header (alternative)	Used to denote section start
`§`	Section header (alternative)	
`%`	Binary prefix	Begins binary number
`;`	Full-line comment	
`#`	Hexadecimal prefix	Begins hexadecimal number
`#`	Inline comment (alternative)	Comment, if starts with `#` followed by at least one space or tab, due to `#` without space is reserved for hex literals (e.g. #FF00FF)
`//`	Inine comment	
`/* */`	Block comment	Marks multi-line comment block
`@`	Directive prefix	Reserved for future syntax
`[ ]`	List literal	
`{ }`	Object literal	
`--`	Line disabling	Experimental use (see Section 3.6)

12.1.2. Reserved Keywords

The following keywords are restricted and SHOULD not be used as bare identifiers (e.g., for keys, values, or section names) unless enclosed in quotes or backticks:

- `/END` *(case-insensitive)*
- `@ver` , `@version`
- `@include` , `@anchor` , `@alias`

Use of these keywords outside their defined roles may result in a parse error.

12.2. Well-Formedness Requirements

A YINI file is considered **well-formed** if it adheres to the core syntactic and structural rules defined in this specification.

11.2.1. Structural Requirements

- A file may consist of zero or more **sections**.
- A file may consist of zero or more valid key-value pairs (members).
- Section headers MUST begin with a valid marker (`^` , `<`).
- At least one space or tab is required between a section marker and the section name.
- Duplicate keys **within the same section and depth level** are not allowed.
  - Keys are case-sensitive, and no spaces nor quotes allowed unless enclosed in backticks (phrase identifiers).
- Lines that do not match any syntactic role (member, comment, section, terminator) are considered malformed.
- The document terminator (`/END`) is **optional in both lenient/strict-mode**.

11.2.2. Character Encoding

Files **MUST** be encoded as **UTF-8 without BOM**.

11.2.3. Line Endings

- Acceptable: Unix-style `<LF>` or Windows-style `<CR><LF>` line endings.
- Mixed line endings are discouraged but tolerated in lenient mode.

11.2.4. Valid Value Types

- Values MUST be one of the supported data types: **String**, **Number**, **Boolean**, **Null**, **List**, or **Object**.
- Boolean values are **case-insensitive**: `True` , `On` , `Yes` , etc.
- Null values: `null` , `NULL` , `Null` are all interpreted as `null` .

11.2.5. Document Terminator

- The terminator is optional in both lenient/strict-mode.
- See Section 3.5, "Document Terminator" for terminator syntax.
- A valid YINI file, MAY end with the terminator marker (`/END`).
- Only one terminator is permitted per file.
- Missing terminators:
  - **In lenient mode:** No error or warning.
  - **In strict mode:** Treated as an error.
- After the terminator, only whitespaces and comments are allowed.

Table: Terminator Requirement by Mode

Mode	Terminator Requirement
Lenient	Optional (*)
Strict	Optional (*)

(*) In systems that value deterministic parsing, MAY favor explicit termination.

11.2.6. Escaping and String Literals

- Escape sequences are **ONLY allowed** in in C-Triple-quoted and Classic strings (quoted with `'` or `"`, **and prefixed** with `C` or `c`).
- Triple-quoted strings MUST use `"""` for both opening and closing (`'''` is not supported).

11.2.7. Shortest Valid YINI Documents in Strict Mode

- **Valid short documents in strict mode:**
  - ✅ In strict mode, the following is the shortest valid YINI document **with a member**:

    ```
    ^T
    ```

    Note: A section heading named `T`, without any members.
- **Invalid short documents:**
  - ❌ Invalid: no title section present:

    ```
    /END
    ```

12.3. Lenient vs. Strict Modes *(Optional Feature)*

Non-strict mode (lenient mode) is the default mode of operation. Parsers SHOULD operate in this mode by default unless explicitly configured otherwise to operate in fully strict mode.

Some YINI parsers may support multiple **validation modes**:

- **Lenient Mode:**
  - Permissive with minor errors (e.g., trailing commas after last value/member (are ignored), mixed line endings).
  - The document terminator (`/END`) is **optional** in both lenient and strict modes and MAY be omitted entirely.
  - All typing rules still apply — for example, string literals MUST be quoted: if a value is not quoted, it is not a string — no exceptions.
  - Empty values are allowed ONLY in members in section-top-levels:
    - Missing/empty value (ONLY outside lists and objects) are treated as `Null` .
    - Trailing comma (after last value/member) inside lists and object - comma is ignored - ONLY in lenient-mode (parse error in strict-mode).
  - Useful for hand-edited configuration files.
- **Strict Mode:**
  - Enforces full well-formedness.
  - No empty values are allowed, MUST always be explicitly with `Null` .
  - No (stray) trailing commas (after last value/member) inside lists and object permitted.
  - Strict mode requires a **title section header** (a level 1 header).

    If also using the OPTIONAL document terminator `/END` then these constraints will provide increased robustness: if a YINI document is split into two halves, **both halves will be invalid**.
    - The **first half** is invalid because it is missing the required `/END` marker.
    - The **second half** is invalid because it lacks the required single level 1 section header (e.g., `^ Title`).
  - Disallows trailing commas.
    - Empty values are disallowed — they MUST be explicitly typed as `null` , `Null` , or `NULL` (case-insensitive), although empty sections (with no members) are allowed.
  - For production and tool-chain use.

Note: Implementations SHOULD clearly document the validation mode in use and detail which rules are fully enforced under strict parsing.

Example:

```
; Lenient mode examples:
list_bracketed1 = [1, 2, ]   # ✅ → [1, 2]
```

```
object1 = { a: 1, b: 2, }    # ✅ → {a: 1, b: 2} (trailing comma dropped)

; Strict mode examples:
list_bracketed2 = [1, 2, ]   # ❌ Error: Stray trailing comma
object2 = { a: 1, b: 2, }    # ❌ Error: Stray trailing comma

list_bracketed3 = [1, 2]     # ✅ OK
object3 = { a: 1, b: 2 }     # ✅ OK
```

12.3.1. Table: Lenient vs. Strict Mode

Feature	Lenient Mode (Default)	Strict Mode	Notes
Explicit string quoting	✅	✅	All strings MUST be enclosed with `"` or `'` — no ambiguity over strings.
Empty sections allowed	✅	✅	Sections may contain no members (e.g., `^ Config`).
Duplicate keys or sections	❌ (may warn)	❌	And never overwrite existing keys/sections.
Required one single level-1 section header	❌	✅	AKA *Title Section*.
`/END` is optional	✅	✅	
Trailing commas after value (inside lists/objects)	✅	❌	In lenient-mode the comma is ignored, error in strict-mode
Missing (empty) value (only in section-top-level)	✅	❌	Will result in a `Null` value in lenient-mode
Missing (empty) value before comma	-	❌	In lenient-mode SHOULD warn or make error
Invalid escape sequences	✅ (may warn)	❌ (error)	

⚠️ Strict mode enforces a stricter contract suitable for automated validation and reproducible builds. Lenient mode favors user-friendliness and flexibility for human editing.

Example:

```
# Lenient:
key1 =          # → ✅ key1 = null
```

```
key2 = ,        # → ❌ error: unexpected comma (may warn)

# Strict:
key1 =          # → ❌ error: missing value
key2 = ,        # → ❌ error: unexpected comma
```

13. Implementation Notes

The following guidance is intended to assist developers implementing YINI parsers, engines, or tools. These notes aim to promote consistent interpretation of YINI syntax across different platforms and environments.

See also [Section 11.2: Well-Formedness Requirements] for formal validation criteria.

13.1. Top-Level Sections and Implicit Root

- If a document contains **multiple top-level sections** (i.e., multiple level-1 sections), they SHOULD be considered **children of an implicit root**.
- The implicit root section:
  - **Has no name**, or may be labeled " `base` ", " `root` " or similar (implementation-defined).
- Section hierarchy MUST be respected:
  - A level-3 section **MUST follow** a level-2 section.
  - Skipping levels (e.g., directly from level-1 to level-3) is invalid.

13.2. Line Handling and Whitespace

- Newline normalization is required:
  - Support all three forms: LF (`0x0A`), CRLF (`0x0D 0x0A`), and CR (`0x0D` .
- Leading/trailing whitespaces (tabs or spaces):
  - Trim from section headers and keys.
- Hyper Strings (H-Strings):
  - Leading/trailing whitespaces (tabs or spaces) and newlines are trimmed.
  - Inside a H-string, whitespaces (tabs or spaces) and newlines are normalized to one single space character.
- Full-line and inline comments may follow key-value members or appear on separate lines.
- Whitespace is permitted within lists, including across lines.

13.3. Value and NULL Handling

- If a key is assigned without a value (only in lenient-mode):

```
key =           // NULL, same as: key = Null
key:            // NULL, same as: key = [Null]
```

it MUST be interpreted as having a value of `null`.

- Key uniqueness:
  - Keys MUST be **unique within the same section and section level**.
  - Duplicates in the same section are a **parse warning**.

13.4. Boolean Canonicalization

- Boolean literals are **case-insensitive**.
- The following values MUST be interpreted as Booleans:
  - `true`, `yes`, `on` → `true`
  - `false`, `no`, `off` → `false`
- DOES NOT allow Boolean values like `1` or `0` unless explicitly cast by the host software.

13.5. Objects

Any empty slot inside `{ ... }` e.g.:

```
object = { a: 1, , b: 2 }
```

is error in strict-mode or at least a warning (in lenient-mode).

13.6. Lists

- Two syntaxes are valid for lists:
  - **(a) Bracketed form (preferred):**

    ```
    items = ["a", "b", "c"]
    ```

    - A bracketed list line MUST not begin with a comma.
    - A trailing comma in `[ ]` is ignored.
  - **(b) Unbracketed multiline:**

    ```
    items1: "a", "b", "c"

    items2:
    "a",
    "b",
    "c"
    ```

    - Trailing commas are allowed in unbracketed lists (form b).

Bracketed lists MUST not have a newline between `=` and the opening bracket `[` (otherwise, the value is interpreted as `null`, not a list):

```
invalidList = // NULL!
[1, 2, 3]     // Not parsed as a list!
```

13.7. Strings Concatenation

- Strings can be concatenated using the `+` operator:

  ```
  name = "Hello, " + "world"
  ```

- Concatenation SHOULD occur **on a single line**.
- Concatenating different types of strings (e.g., r"..." + c'...') is **permitted** (for use in some special or advanced cases), but generally **discouraged**.
- Only Classic strings (C-Strings) interpret escape sequences (`\n` , `\t` , etc).

13.8. String Literal Types

Note: If a string is not quoted, it's not a string — period.

Type	Prefix	Example	Behavior
(Raw) String	*None*, `R` , or `r`	"Some text"	Text is as-is (raw), no escaping, backslash is literal, preserves all whitespace
Classic String	`C` or `c`	`C"..."`	Escape sequences are interpreted
Hyper String	`H` or `h`	`H"..."`	(*) Multi-line, whitespace-collapsing, trimmed
Triple-quoted String	*None*, `R` , or `r`	`"""..."""`	Can be multi-line, text is as-is (raw), preserves all whitespace, including line breaks
C-Triple-quoted String	`C` , or `c`	`C"""..."""`	Can be multi-line, escapes interpreted, preserves all whitespace, including line breaks

(*) Hyper string behavior:

- Allow multi-line strings.
- Collapse sequences of whitespace and newlines into a single space.
- Trim leading/trailing whitespace.

13.9. Comments

- YINI supports:
  - `//` for inline comments (rest of the line is ignored).
  - `#` followed by at least one whitespace character (`SPACE` or `TAB`), for alternative inline comments (rest of the line is ignored).

- `/* ... */` for block (multi-line) comments (may span lines).
- `;` at start of line is treated as full-line comments (there may appear only spaces or tabs before `;`).
- **Nested block comments are not supported.**

13.10. Error Handling Recommendations

If the parser encounters:

- A **missing intermediate section level** (e.g., level 3 without level 2),
- A **duplicate key in the same section and section level**,
- A **malformed list or string**

It SHOULD:

- **Fail gracefully** and report an error, OR
- **Use host-defined fallback logic**, if robustness is preferred.

13.11 Bonus Tips for Implementation

Developers are encouraged to implement the following features to improve parser robustness and developer experience:

- Attach **position metadata** (line/column) to tokens for better diagnostics.
- Normalize internal representations of:
  - `true / false`
  - `null`
- Support both `strict` as well as `lenient` parsing modes.
- Support different **Bail/Abort Sensitivity Levels** while parsing a YINI document: (AKA severity threshold)
  - Level 0 = Ignore errors and try parse anyway (may remap falty key/section names to something more valid so parsing can continue).
  - Level 1 = Abort on errors only.
  - Level 2 = Abort even on warnings.
- **Extra Bonus:** In strings, if C/C++-style octal escape codes like `\1` to `\377` are used (which are not valid in YINI), parsers SHOULD treat this as an error in strict mode (and suggest the correct YINI syntax). In lenient mode, parsers may optionally emit a warning and suggest the correct YINI syntax: `\o1` to `\o377`, and interpret the octal code as intended.

14. Compatibility and Versioning

This section covers YINI's compatibility and interoperability principles.

14.1. Fallback Rules

14.1.1. Invalid Sections or Keys

- Invalid key names or section headers **SHOULD be retained as-is** if possible and/or illegal characters remapped (both in lenient and strict mode, and optionally also support "Abort Sensitivity Levels"), though at least a warning SHOULD be issued depending on it's severity.

14.1.2. Graceful Degradation

- Parsers SHOULD issue warnings instead of errors when encountering unrecognized features (e.g., unknown directives, anchors, or section markers).
- Implementations SHOULD strive to process known-valid content even if advanced features are not supported.

14.2. Versioning Strategy

Version Format

- In the future, the YINI Specification will adopt *Semantic Versioning* (`MAJOR.MINOR.PATCH STAGE`) to signal format evolution.
- **`STAGE`** : For the time being, the specification version remains `v1.0.0` until all primary features are implemented, tested, and the specification exits in the `Beta` stage. The next stage after `Beta` will be denoted `RC` (Release Candidate). Each stage may be appended with an incremental number, such as `Beta 2` , `Beta 3` , `Beta 4` , etc.

Semantic Versioning:

- **MAJOR:** Incompatible changes.
- **MINOR:** Backward-compatible additions.
- **PATCH:** Backward-compatible fixes.

14.3. Encoding Notes

14.3.1 Required Encoding

- **UTF-8 without BOM** is the required and default encoding for YINI files.
- Parsers SHOULD support UTF-8 fully.
- Use of other encodings (e.g., UTF-16, Latin-1) is discouraged and not guaranteed to be portable.

14.3.2 Byte Order Mark (BOM)

- A UTF-8 BOM (`0xEF 0xBB 0xBF`) is **not required** and **SHOULD be avoided**.
- If present, parsers MUST detect and ignore the BOM without failing.

14.3.3 Shebang Line (Optional)

A shebang line may be used at the very top of the file:

```
#!/usr/bin/env yini
```

This line is ignored by YINI parsers but may affect script execution in Unix environments. It MUST be ignored by the YINI parser as a comment or metadata line.

14.4. JSON Compatibility

A valid YINI file can be converted into a valid JSON file (and vice versa) while preserving **data structure, types, and identifier names exactly**.

Although YINI and JSON are distinct formats with different goals, their core data models are closely aligned. Strings, numbers, booleans, nulls, and lists in YINI map naturally to their JSON equivalents, and key names (identifiers) maintain structural correspondence when properly quoted.

You can convert a YINI file into JSON (and vice versa) and preserve all real data — structure, types, and names — correctly,
but you SHOULD ignore or remove YINI-only features like section markers, comments, and terminators when expressing it as JSON.

Converting YINI to JSON

- **Keys (Identifiers):** As long as keys are quoted properly in JSON, and identifiers follow JSON string rules.
- **Value types (String, Number, Boolean, Null, List):** Direct mapping is possible between YINI and JSON native types.
- **Lists (Arrays):** Lists `[]` map 1-to-1.
- **Sections:** YINI sections correspond to nested objects `{}` in JSON.
- **String types:** ⚠️ All strings become plain JSON strings once parsed, escaping is normalized when serialized.

Metadata Limitations

- 🚫 Metadata (such as comments and the document terminator) will not be carried over to JSON.

Summary

When converted carefully, a well-formed YINI document can be faithfully represented as a valid JSON object, preserving all structure, data types, and identifier names.
Conversely, a valid JSON object can be mapped into a YINI document, provided that JSON-specific

constructs (such as deeply nested objects) are expressed as sections and that YINI syntax rules are respected.

Table: Correspondence Between YINI and JSON

Entity / Feature	To YINI	To JSON	Notes
Structure Mapping	✅ Yes (sections become objects)	✅ Yes (objects, nested)	Each YINI section is mapped to a JSON object.
Types Mapping	✅ Yes (direct type mapping)	✅ Yes (direct type mapping)	All core types (string, number, bool, null, object, list) are preserved.
Key/Value Syntax	✅ Top-level: `key = value` Object: `key: value`	✅ Keys always quoted, `:` for objects	YINI uses `=` for top-level, `:` for object members.
Identifiers (Keys)	✅ Unquoted, or backticked if needed	✅ Must always be quoted	Backticks in YINI for special chars; JSON always quotes keys.
Comments	✅ Supported (full-line, inline, or multi-line)	🚫 Not supported (discarded)	Comments are dropped when converting to JSON.
Terminator	✅ Optional (for doc-ending)	🚫 Not supported (ignored)	YINI document terminator (`/END`) is ignored in JSON.

See also:

See Sections 14.3 and 14.4 for examples of YINI ⇆ JSON mappings.

15. Examples

15.1. Minimal Example

```
^ Prefs

name = "Kim"
entries = 10
enabled = true
```

Explanation:

- Begins with a single section `^ Prefs`.
- Contains three keys (`name`, `entries`, `enabled`) with string, number, and boolean values, respectively.

15.2. Realistic Config Use Cases

15.2.1. User Preferences Configuration

```
^ Preferences

theme = "dark"
language = "en"
notifications = true
volume = 85
recent_files = [
    "report.yini",
    "draft_0423.yini",
    "budget2025.yini"
]
```

15.2.2. Application Settings with Sections

```
^ Database
host = "localhost"
port = 5432
username = "appuser"
password = "s3cret"

^ Logging
level = "debug"
file = "/var/log/myapp.log"
rotate = true

^ Features
enable_experimental = false
api_version = "v2.1"
```

Notes:

- The `^` marker cleanly divides logical domains into sections.

15.2.3. Script Metadata

```
^ Metadata
name = "Data Fetch Script"
version = "1.3.0"
author = "Jane Doe"
schedule = "daily"
active = true
```

15.2.4. Feature Flags

```
^ FeatureFlags
debug = true
experimental_ui = false
use_cache = true
cache_expiry = 86400            // In seconds
last_purge_date = "2025-05-25"  // YYYY-MM-DD
```

15.2.5. Feature Toggles with Alternative Syntax

```
/*
  Feature Toggles with Alternative Syntax
*/

< `Feature Toggles`
`Debug` = ON
`Experimental UI` = OFF
`Night Mode` = OFF
`Use Cache` = ON

    << `Cache Config`
    `Cache Expiry` = 86400              // In seconds
    `Last Purge Date (YYYY-MM-DD)` = "2025-05-25"
```

Notes:

- Uses the alternative section marker `<`.
- Demonstrates alternative boolean literals: `ON` and `OFF`.
- `Cache Config` is a nested subsection of `Feature Toggles`.
- All keys and section header identifiers are enclosed in backticks, allowing the use of spaces and special characters.

15.3. Examples of YINI → JSON Mapping

15.3.1. Simple Flat Structure to JSON

YINI:

```
^ User
name = "Alice"
age = 28
active = true
```

JSON Equivalent:

```
{
  "User": {
    "name": "Alice",
    "age": 28,
    "active": true
  }
}
```

15.3.2. Nested Sections (multiple levels) to JSON

YINI:

```
^ Settings
theme = "dark"
language = "en"

    ^^ Display
    resolution = "1920x1080"
    fullscreen = true
```

JSON Equivalent:

```
{
  "Settings": {
    "theme": "dark",
    "language": "en",
    "Display": {
      "resolution": "1920x1080",
      "fullscreen": true
    }
  }
}
```

15.3.3. Lists (Arrays) to JSON

YINI:

```
^ Server
```

```
hosts = ['server1.example.com', 'server2.example.com']
```

JSON Equivalent:

```
{
  "Server": {
    "hosts": [
      "server1.example.com",
      "server2.example.com"
    ]
  }
}
```

15.3.4. Nulls and Booleans to JSON

Note: Null and boolean literals (e.g., `Null`, `True`, `On`, `Yes`) in YINI are **case-insensitive**, while keys are **case-sensitive**.

YINI:

```
^ Flags
enabled = On
archived = Off
description = Null
```

JSON Equivalent:

```
{
  "Flags": {
    "enabled": true,
    "archived": false,
    "description": null
  }
}
```

15.4. Examples of JSON → YINI Mapping

15.4.1. Simple JSON Object to YINI

JSON:

```
{
  "Profile": {
    "username": "bob",
    "age": 35,
    "verified": true
```

```
  }
}
```

YINI Equivalent:

```
^ Profile
username = "bob"
age = 35
verified = true
```

15.4.2. Nested JSON Objects to YINI

JSON:

```
{
  "App": {
    "version": "2.5",
    "settings": {
      "theme": "light",
      "notifications": true
    }
  }
}
```

YINI Equivalent:

```
^ App
version = "2.5"

    ^^ settings
    theme = "light"
    notifications = true
```

15.4.3. JSON Array to YINI

JSON:

```
{
  "Servers": {
    "hosts": ["alpha.local", "beta.local", "gamma.local"]
  }
}
```

YINI Equivalent:

```
^ Servers
hosts = ["alpha.local", "beta.local", "gamma.local"]
```

15.5. Large-Scale Real-World Configuration Example A: Corporate SaaS Platform

```
@YINI

// Example A: Corporate SaaS Platform.
/*
  Covers:
  - Sections & deep nesting
  - Real-world domain structure
  - Objects in arrays
  - Arrays of objects
  - Scalars of every type
  - Complex policy logic
  - Auth & security modeling
  - Unicode in strings.
  - Strings in double quotes.
  - Large but readable
 */

^ App
name = "Acme Platform"      // Example Platform
description = "The word “Acme” has been used for over 100 years in technical and business ex
meaning = "It comes from Greek akmḗ (ἀκμή), meaning “the highest point” or “best”."
version = "2.3.1"
debug = OFF
environment = "production"
maintainers = ["ops@acme.com", "dev@acme.com"]

    ^^ Features
    enableSearch = true
    enablePayments = true
    enableAnalytics = false
    experimental = ["new-ui", "streaming-api"]

    ^^ Limits
    maxUsers = 50000
    requestTimeoutMs = 3500
    retryPolicy = { maxRetries: 5, backoff: "exponential" }

    ^^ Database
    engine = "postgres"
    host = "db.internal.acme.com"
    port = 5432
    ssl = true
    pool = { min: 5, max: 50 }
```

```
        ^^^ Credentials
        username = "app_user"
        password = "****"
        rotateEveryDays = 90

    ^^ API
    baseUrl = "https://api.acme.com"
    publicEndpoints = ["/health", "/status"]
    internalEndpoints = ["/admin", "/metrics"]

        ^^^ Auth
        provider = "oauth2"
        tokenTTLSeconds = 3600
        scopes = ["read", "write", "admin"]

                ^^^^ Clients
                web = { clientId: "web-123", redirectUri: "https://acme.com/callback" }
                mobile = { clientId: "mob-456", redirectUri: "acme://auth" }

^ Logging
level = "info"
format = "json"
outputs = ["stdout", "file"]

    ^^ File
    path = "/var/log/acme/app.log"
    maxSizeMB = 100
    rotate = true
    keepFiles = 10

    ^^ Metrics
    enabled = true
    endpoint = "/metrics"
    sampleRate = 0.25

^ Services
enabled = true

    ^^ Email
    provider = "smtp"
    host = "smtp.acme.com"
    port = 587
    secure = false
    from = "no-reply@acme.com"

        ^^^ Credentials
        user = "mailer"
        pass = "mailer-secret"

    ^^ Cache
    type = "redis"
    host = "cache.internal.acme.com"
    port = 6379
```

```
    ttlSeconds = 600

        ^^^ Cluster
        nodes = [
          { host: "cache-1.internal", port: 6379 },
          { host: "cache-2.internal", port: 6379 },
          { host: "cache-3.internal", port: 6379 }
        ]

^ Observability
tracing = true
tracingProvider = "opentelemetry"
traceSampleRate = 0.1

    ^^ Exporters
    jaeger = { enabled: true, endpoint: "http://jaeger:14268/api/traces" }
    prometheus = { enabled: true, endpoint: "http://prom:9090" }

^ Security
allowedIPs = ["10.0.0.0/8", "192.168.0.0/16"]
blockedCountries = ["KP", "SD"]

    ^^ Policies
    passwordMinLength = 14
    require2FA = true
    sessionTTLMinutes = 120

        ^^^ Lockout
        maxAttempts = 5
        lockoutMinutes = 30
```

15.6. Large-Scale Real-World Configuration Example B: High-Security Distributed Control System

```
@YINI

// Example B: High-Security Distributed Control System.
/*
  Covers:
  - Nested arrays inside inline objects.
  - Sections & deep nesting.
  - Real-world domain structure.
  - Objects in arrays.
  - Arrays of objects.
  - Scalars of every type.
  - Complex policy logic.
  - Auth & security modeling.
  - Unicode in strings.
  - Strings in single quotes.
  - Large but readable.
 */
```

```
^ App
name = 'Nebula Control Suite'
description = 'A distributed operations platform for autonomous systems and edge analytics.'
meaning = 'Nebula comes from Latin "nebula" meaning mist or cloud.'
version = '5.0.0-rc.4'
debug = ON
environment = 'staging'
maintainers = ['infra@nebula.io', 'platform@nebula.io', 'secops@nebula.io']

    ^^ Features
    enableSearch = false
    enablePayments = false
    enableAnalytics = true
    experimental = ['vector-engine', 'adaptive-ui', 'ai-routing']

    ^^ Limits
    maxUsers = 120000
    requestTimeoutMs = 7200
    retryPolicy = {
        maxRetries: 9,
        backoff: 'fibonacci',
        retryOn: ['timeout', '5xx', 'throttle'],
        schedule: [
            { attempt: 1, delayMs: 80 },
            { attempt: 2, delayMs: 160 },
            { attempt: 3, delayMs: 320 },
            { attempt: 4, delayMs: 640 },
            { attempt: 5, delayMs: 1280 }
        ]
    }

    ^^ Database
    engine = 'cockroachdb'
    host = 'cluster.db.nebula.io'
    port = 26257
    ssl = true
    pool = {
        min: 12,
        max: 120,
        warmup: {
            enabled: true,
            strategy: 'aggressive',
            steps: [10, 25, 50, 75, 100],
            healthChecks: [
                { name: 'connectivity', timeoutMs: 300 },
                { name: 'replication', maxLagMs: 200 },
                { name: 'quorum', minNodes: 3 }
            ]
        }
    }

        ^^^ Credentials
        username = 'nebula_app'
```

```
        password = '****'
        rotateEveryDays = 45
        history = [
            { rotatedAt: '2025-05-10', reason: 'scheduled' },
            { rotatedAt: '2025-03-02', reason: 'key-compromise' },
            { rotatedAt: '2024-12-15', reason: 'policy-change' }
        ]

    ^^ API
    baseUrl = 'https://api.nebula.io'
    publicEndpoints = ['/health', '/status', '/version']
    internalEndpoints = ['/admin', '/metrics', '/orchestrator', '/scheduler']

        ^^^ Auth
        provider = 'oidc'
        tokenTTLSeconds = 5400
        scopes = ['read', 'write', 'deploy', 'audit']

            ^^^^ Clients
            web = {
                clientId: 'nebula-web-prod',
                redirectUri: 'https://nebula.io/auth/callback',
                allowedOrigins: ['https://nebula.io', 'https://console.nebula.io'],
                secrets: [
                    { id: 'alpha', value: 'QX7faP9', active: true },
                    { id: 'beta', value: 'LM8KdW2', active: true },
                    { id: 'legacy', value: 'OLD-KEY-DO-NOT-USE', active: false }
                ]
            }

            mobile = {
                clientId: 'nebula-mobile',
                redirectUri: 'nebula://auth',
                platforms: [
                    { name: 'ios', minVersion: '15.2', enabled: true },
                    { name: 'android', minVersion: '11', enabled: true },
                    { name: 'harmonyos', minVersion: '4', enabled: false }
                ],
                refreshPolicy: {
                    enabled: true,
                    limits: { perHour: 60, perDay: 600 },
                    audit: [
                        { event: 'refresh', severity: 'info' },
                        { event: 'suspicious-location', severity: 'warning' },
                        { event: 'credential-stuffing', severity: 'critical' }
                    ]
                }
            }

^ Logging
level = 'debug'
format = 'ndjson'
outputs = ['stdout', 'file', 'syslog']
```

```
    ^^ File
    path = '/srv/log/nebula/nebula.log'
    maxSizeMB = 250
    rotate = true
    keepFiles = 30

    ^^ Metrics
    enabled = true
    endpoint = '/internal/metrics'
    sampleRate = 0.75

^ Services
enabled = true

    ^^ Email
    provider = 'ses'
    host = 'email.nebula.io'
    port = 465
    secure = true
    from = 'system@nebula.io'

        ^^^ Credentials
        user = 'mailer-nebula'
        pass = 'MAIL-SEC-9921'

    ^^ Cache
    type = 'keydb'
    host = 'cache.nebula.internal'
    port = 6380
    ttlSeconds = 1800

        ^^^ Cluster
        nodes = [
            { host: 'cache-a.nebula', port: 6380, role: 'primary', zones: ['eu-north-1a'] },
            { host: 'cache-b.nebula', port: 6380, role: 'replica', zones: ['eu-north-1b'] },
            { host: 'cache-c.nebula', port: 6380, role: 'replica', zones: ['eu-north-1c'] },
            { host: 'cache-d.nebula', port: 6380, role: 'observer', zones: ['eu-north-1a'] }
        ]

        ^^^ Failover
        strategy = {
            mode: 'predictive',
            thresholds: { errorRate: 0.08, latencyMs: 180 },
            actions: [
                { step: 'drain-traffic', timeoutMs: 1500 },
                { step: 'promote-replica', timeoutMs: 2000 },
                { step: 'resync', propagate: true },
                { step: 'notify', channels: ['pagerduty', 'slack', 'email'] }
            ]
        }

^ Observability
```

```
tracing = true
tracingProvider = 'tempo'
traceSampleRate = 0.35

    ^^ Exporters
    jaeger = {
        enabled: false,
        endpoint: 'http://jaeger.internal/api/traces',
        tags: {
            region: 'eu-north',
            environment: 'staging',
            build: { version: '5.0.0-rc.4', commit: 'c8f91d2', dirty: true }
        }
    }

    prometheus = {
        enabled: true,
        endpoint: 'http://prometheus.nebula:9090',
        scrapeIntervals: [2, 5, 10, 30],
        retention: { days: 90, maxSeries: 3500000 }
    }

^ Security
allowedIPs = ['172.16.0.0/12', '100.64.0.0/10']
blockedCountries = ['KP', 'NG', 'BY']

    ^^ Policies
    passwordMinLength = 18
    require2FA = true
    sessionTTLMinutes = 45

        ^^^ Lockout
        maxAttempts = 4
        lockoutMinutes = 60
        escalation = {
            enabled: true,
            notify: ['security@nebula.io', 'ciso@nebula.io'],
            rules: [
                { attempts: 3, action: 'captcha' },
                { attempts: 4, action: 'temporary-block', minutes: 120 },
                { attempts: 6, action: 'account-freeze' },
                { attempts: 9, action: 'permanent-block' }
            ]
        }
```

16. Appendices and Reserved Areas

16.1. License

16.2. Acknowledgments

_YINI would not exist in this form and shape what it is today, without the many insights, challenges, and thoughtful, constructive feedback from the community.

For more details, see section D.2, *"Acknowledgments & Special Thanks"*, in the Rationale document.

16.3. Author(s)

This specification is created and maintained by Marko K. Seppänen.

Creator

First authored in **2024 in Gothenburg, Sweden,** by **Marko K. Seppänen** (Sweden via Finland).

Mr. Seppänen has been programming since the **mid-1980s**, starting with platforms and languages such as **BASIC and various BASIC dialects, C, Java, and Assembler**, and later programming across both mainstream and more niche languages — from **C#** to **Haskell** and **Erlang**.

He studied **Computer Science & Technology** and **Master's in Software Development** with a focus on **Programming Languages**, at **Chalmers University of Technology** (Gothenburg, Sweden).

Professionally, he has **decades of experience in software development and engineering**, particularly in **TypeScript, JavaScript, Python, PHP,** and **full-stack web development**, as well as **tooling and both user- and developer-focused systems**.

16.4. Spec Changes

A running log of changes and updates **to this YINI specification**.

Notes:

- More details of the feedback, see section D.2, *"Acknowledgments & Special Thanks"*, in the Rationale document.
- All dates in international format, YYYY-MM-DD.

v1.0.0 RC 4, 2026-03-29

- **Removed:** Support for colon-based list syntax (`key: value1, value2` and multi-line `key:` list form).
- **Clarified:** Lists in YINI are defined only with `=` and square brackets `[ ... ]`.
- **Rationale:** The colon-list syntax added convenience but did not add core expressive power, and its removal improves clarity, predictability, and grammar simplicity.

- **Changed:** Removed support for the additional alternative section marker `€`, due to no clear practical benefit compared to the existing markers.
- **Clarified:** The supported section markers are now explicitly `^` (primary), `<`, and `§`.
- **Fixed:** Corrected an error in Example 15.1.
- **Updated:** Added two large real-world configuration examples (A and B), featuring nested inline objects, lists, and complex structures.
  - See Sections **15.5** and **15.6**.
  - The full YINI and JSON versions of these examples are also included under Large-Scale Real-World Configuration Examples.
- **Clarified:** Added clarifying bullets to Sections 1.2 and 1.4.
- **Fixed:** Fixed a few typos and made various minor wording and consistency improvements.

v1.0.0 RC 3, 2025-09-01

- The specification has been revised to clarify that the document terminator `/END` is no longer a mandatory requirement in strict mode. The terminator is now defined as optional in both lenient and strict parsing modes. Implementing parsers MAY optionally provide an option to require this in both lenient and strict mode.

v1.0.0 RC 2, 2025-08-11

- Added case-insensitive support for digits `A` (10) and `B` (11) as alternative syntax in duodecimal (base-12) notation.

v1.0.0 RC 1, 2025-07-26

- Added support for YINI marker `@yini`, section 2.4, "YINI Marker (`@yini`)".
- Discontinued alternative marker character `~` (visually ambiguous) in favor of `<`.
- Promoted the section markers `§` and `€` to "Experimental" from only being "Reserved".
- Dropped the use of `=` in object literals, objects now use `:` (similar as to JSON, etc).

v1.0.0 Beta 7, 2025-06-12

- Clarified where special/control characters (U+0000–U+001F) are allowed in Backticked Identifiers, Raw Strings, Classic Strings, and Triple-Quoted Strings. These characters are now disallowed in Backticked Identifiers and Classic Strings unless they are escaped, with exceptions for TAB and SPACE in the latter.
- Updated Hyper Strings to support <Unicode-WS> for indentation and whitespace normalization.
- Added table of all "Unicode Whitespace Characters" in <Unicode-WS>.
- Added support for Triple-quoted strings with the prefix `C`, which interprets escape codes. Additionally, they can optionally be prefixed with `R` but this is not required since they are Raw by default.

- Added " `base` " as an alternative name for the implicit root section, in addition to the previously suggested " `root` ".
- Changed policy in 14.1, "Fallback Rules" to keep invalid key names or section headers as-is.
- Added note about optional "Abort Sensitivity Levels" in parsing.
- Added a couple of sections in future:
  - 10.1.1, "Short-hand Section Marker"
  - 10.1.2, "Inline Objects"
- Updated section heading rule:
  - Section heading markers (`^` , `~` , or `§`) may be repeated up to six times to indicate levels 1–6.
  - Beyond level 6, numeric shorthand MUST be used (see Section 5.3.1).
- Added support for objects (`{ ... }`).
- Changed handling of trailing comma to:
  - A missing value (empty value) for a **section-top-level** key-value assignment and that is not inside `[ ]` or `{ }`) is equivalent to `Null` .
  - A missing **last element/member** inside `[ ... ]` or `{ ... }` is always considered a trailing comma and does not produce a null value/element (the comma is ignored).

v1.0.0 Beta 6, 2025-05-20

- Reworked the use of `#` **based on feedback**: it is no longer a section marker and is now used exclusively as a comment symbol (more in line with formats like classic INI, Bash, etc).
  - **(An important caveat):** comments starting with `#` MUST be followed by a space or tab.
  - This requirement prevents clashes with hex-like values. Using `#` for hex numbers (e.g., `#FF0033`) is a deliberate design choice and compromise to align with conventions found in CSS (for color) and similar contexts. For example: `#FF0033` is a hex value, whereas `# FF0033` is treated as a comment.
- Due to the change where `#` is no longer used as a section marker, the tilde (`~`) was initially considered as the new default. However, multiple tildes on a line tend to visually blend together. In the end, the caret (`^`) was chosen instead for its clarity, visual distinctiveness, and compatibility with the 7-bit ASCII range.
- Added support for full line comment using `;` and disable line using `--` .
- Added section 16.6, "Appendix C – Common Mistakes and Pitfalls".

v1.0.0 Beta 5, 2025-05-13

- Added new section 16.2, "Acknowledgments".
- Changed the default mode (after feedback of not requiring the /END) to non-stict (lenient) from Strict-mode:
  - Thus the "Document Terminator" is now only optional.
  - Renamed section name to 12.3, "Lenient vs. Strict Modes".
- Added tab as illegal character in backticked identifiers.

- Deprecated `>` for use as section marker, due to its tendency to be confused with quoting syntax in forums, emails, and messaging platforms, etc.
- Added missing escape codes in strings (matching those from C/C++), with one exception: YINI uses `\OOO` instead of `\oOOO` for octal values, as the `o` clearly indicates that an octal sequence follows, whereas the C-style form does not.
- Reserved `{ }` for future syntax (inline objects).
- Renamed the term "Phrased identifiers" to "Backticked identifiers", it's simpler. – Removed support (after feedback by user JoshYx at Reddit) for the alternative document terminator `###`. Although it was intended as a shorter, a one character shorter alternative to `/END`, it contradicted YINI's core principle of simplicity. Its presence risked confusing users unfamiliar with YINI's syntax and ultimately undermined clarity.

v1.0.0 Beta 4

- Fixed an issue with very short YINI files in the grammar: both members and sections are now explicitly optional.

v1.0.0 Beta 3, 2025-04-25

v1.0.0 Beta 2, 2025-04-23

- Added (new) support for triple-quoted strings (`"""`).
- Added support for alternative hexadecimal literals using `#`.
- Added support for binary literals using `%`.
- Reintroduced support for the alternative terminator marker `###`.

16.6. Appendix C – Common Mistakes and Pitfalls

Below are some common mistakes and misunderstandings when writing YINI files, especially for users familiar with other formats like YAML, JSON, or classic INI. This table aims to clarify syntax edge cases and help avoid subtle bugs.

Trailing commas in list and objects

Note: Trailing commas (after any value/member) inside list or objects, does never result in any `Null` values. Strict mode disallows a comma with no element after it altogether.

✅ YINI Syntax Cheatsheet – Common Confusions

Element	Correct Syntax	Common Mistake	Clarification
Key–Value pair / List	`name = "John"` / `items = ["a",`	`name: "John"` / `items:`	`:` is not valid assignment syntax in YINI; use `=` for both single

Element	Correct Syntax	Common Mistake	Clarification
	`"b", "c"]`		values and lists.
Inline List	`items = ["a", "b", "c"]`	`items =` followed by newline and `[` on next line	Line break after `=` causes the value of `items` to be parsed as null.
Trailing comma (inline)	`list = ["a", "b", "c",]`	Empty value assumed to be null	The comma is ignored, and does NOT add any `null` item at the end of the list. The result is same as: `list = ["a", "b", "c"]`
Comments	`# Comment` or `// Comment`	`#Comment`	`#` MUST be followed by **space or tab** to be recognized as a comment.
Hex values	`color = #FF0033`	Assumed to be a comment	Without space after `#`, this is a valid hex value.
Disable line	`--key = "something"`	Treated like a comment	Entire line is ignored, including valid config syntax.
List nesting	`list = [[1, 2], [3, 4]]`	Using inner lists without brackets	All nested lists MUST be bracketed explicitly.
Section skipping	`^^ Section`, `^^^ Subsection`	Jumping directly to `^^^`	❌ Invalid — cannot skip intermediate nesting levels.

16.7. 📃 Unicode Whitespace Characters

Below is a categorized list of Unicode whitespace characters recognized as within <Unicode-WS>, these are normalized or trimmed in Hyper Strings.

Code Point	Character Name	Abbreviation	Unicode Category	Notes
`U+0009`	-	TAB	Cc	Common ASCII tab
`U+000A`	LINE FEED	LF	Cc	Newline / Unix line ending
`U+000D`	CARRIAGE RETURN	CR	Cc	Used in Windows line endings

Code Point	Character Name	Abbreviation	Unicode Category	Notes
U+0020	-	SPACE	Zs	Standard space
U+00A0	NO-BREAK SPACE	NBSP	Zs	Common in HTML
U+1680	OGHAM SPACE MARK		Zs	Rare, historic
U+2000	EN QUAD		Zs	Typographic space
U+2001	EM QUAD		Zs	Typographic space
U+2002	EN SPACE		Zs	Narrower than EM space
U+2003	EM SPACE		Zs	Wider than EN space
U+2004	THREE-PER-EM SPACE		Zs	Typographic space
U+2005	FOUR-PER-EM SPACE		Zs	Typographic space
U+2006	SIX-PER-EM SPACE		Zs	Typographic space
U+2007	FIGURE SPACE		Zs	Aligns with numeric digits
U+2008	PUNCTUATION SPACE		Zs	Same width as a period
U+2009	THIN SPACE		Zs	Narrow space
U+200A	HAIR SPACE		Zs	Very narrow space
U+202F	NARROW NO-BREAK SPACE		Zs	Used in Mongolian, etc.
U+205F	MEDIUM MATHEMATICAL SPACE		Zs	Used in math layout
U+3000	IDEOGRAPHIC SPACE		Zs	Full-width space in CJK
U+2028	LINE SEPARATOR		Zl	Treated as newline in some contexts
U+2029	PARAGRAPH SEPARATOR		Zp	Paragraph break

^ YINI Specification ≡

> A simple, structured, and human-friendly configuration format.

yini-lang.org · YINI on GitHub