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

Specification for the YINI Format

Version: 1.0.0-RC.6 **Date:** 2026-05-30

> **Status:** This is a release-candidate draft. Minor clarifications and corrections may still be made before YINI 1.0.0.

⇨ Table of Contents

Preface

YINI (by the YINI-lang project) was designed with a simple idea in mind: configuration files should be clear to read, straightforward to edit, and predictable to interpret.
Its design emphasizes explicit structure and human-readable syntax, while remaining formal enough to support consistent parsing and reliable tooling.
The name *YINI* originates from "Yet another INI", reflecting its inspiration from the traditional INI format.

YINI grew from a concrete question: Could a configuration format be more structured than INI, more readable than JSON, and less fragile than indentation-sensitive formats like YAML? From that starting point, YINI gradually developed around a small set of core design goals:

- Clarity over cleverness.
- Readability without sacrificing structure.
- Simplicity with serious usability.
- Predictability over magic.
- Explicitness over hidden behavior.
- Structure without visual clutter.
- Human-friendly editing with deterministic parsing.

YINI is therefore intended to be simple, but not simplistic: explicit where structure matters, readable at a glance, and defined clearly enough to support robust implementations.

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

While inspired by well-known formats such as INI, JSON, Python, Markdown, and YAML, YINI defines its own approach through explicit structure, clear typing rules, and predictable interpretation.
A central part of this approach is its handling of nested sections, which makes hierarchy visible without requiring indentation to define structure.

This specification defines YINI with the goal of serving both human authors and implementers by describing the format in a clear, precise, and consistent way.

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.

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 supports scalar values, lists, inline objects, members, and nested sections. These constructs allow documents to represent both flat and hierarchical configuration data.

1.2. Purpose and Design Goals

The YINI format was designed with the following philosophy and key goals in mind:

1. **Clarity over cleverness**
   YINI is designed to favor clear and understandable syntax over compact, implicit, or surprising constructs. The format aims to reduce ambiguity and make both documents and rules easier to interpret correctly.

2. **Readability without sacrificing structure**
   YINI is intended to remain easy for humans to read and follow, while still providing real structure for nested and non-trivial configuration data. Readability is treated as a primary requirement, not as an afterthought.

3. **Simplicity with serious usability**
   YINI aims to keep the core syntax simple, consistent, and approachable, while still supporting practical configuration needs such as lists, objects, comments, and nested sections. The goal is to remain simple, but not simplistic.

4. **Predictability over hidden behavior**
   YINI is designed so that the meaning of a document follows explicit and stable rules. It avoids relying on invisible or context-sensitive behavior whenever possible, so that users can more easily understand how a document will be interpreted.

5. **Explicit structure without visual clutter**
   YINI is designed to represent hierarchy and grouping explicitly, without depending on indentation to define structure. The goal is to make structure visible and reliable while keeping the overall notation compact and readable.

6. **Human-friendly editing**
   YINI is intended to be comfortable for humans to write, review, and maintain. Its syntax is designed to support direct editing of configuration files without requiring excessive punctuation, ceremony, or visual noise.

7. **Deterministic parsing**
   YINI is designed to support deterministic (predictable and consistent) parsing by implementations, especially in strict mode. This helps reduce ambiguity and improves robustness for validation, tooling, and long-term maintainability.

1.3. Background and Intent

YINI was created as a configuration format for cases where human readability, explicit structure, and predictable interpretation are more important than compactness or maximum syntactic flexibility.

Its design aligns with several established software-design principles, including **KISS** ("*Keep It Simple, Stupid*"), the **principle of least surprise / least astonishment** (POLA), **explicit is better than implicit**, and the contract-oriented idea behind the **Liskov Substitution Principle (LSP)**. These principles are reflected in YINI's visible section structure, quoted strings, and deterministic parsing rules.

YINI also makes deliberate trade-offs. It does not try to infer structure from indentation, minimize every character of syntax, or behave like a general expression language. Instead, it favors explicit structure and stable interpretation rules, because configuration files are often read, reviewed, edited, and validated over time.

For a deeper discussion of the motivation and design philosophy, see Why YINI?.

1.4. Key Features

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

needed.

YINI prioritizes **clarity, structure, and predictability**. The features below show how those properties take shape in the format.

- **Explicit Section Structure:** Sections are defined by explicit header markers, not indentation. The primary marker is `^`, with `§`, `>`, and `<` accepted as alternatives. This keeps document structure visible and unambiguous.
- **Indentation-insensitive Syntax:** YINI does not use indentation to determine structure. Indentation may be used for readability, but it does not affect parsing.
- **Typed Values Without Type Annotations:** YINI supports strings, numbers, booleans, nulls, lists, and inline objects. The type of a value is inferred from how it is written, so no separate type annotation is needed.
- **Inline Collections:** Lists use `[]`, and inline objects use `{}` with `:` as the canonical member separator. Inline collections are not indentation-sensitive.
- **Commenting and Documentation:** Users can comment/annotate values and sections with comments. Comments are ignored by the parser and have no effect on the parsed output.
- **Nested Sections:** YINI supports hierarchically nested sections, allowing complex configurations to be expressed with explicit structure.
- **Multiple String Forms:** YINI supports several string forms for different needs, including single-quoted strings, double-quoted strings, raw strings, classic escaped strings, and triple-quoted multi-line strings.
- **Multi-line Strings:** YINI supports multi-line string literals for longer text values, without requiring unrelated document structure to be affected.
- **Dual Parsing Modes:** YINI defines both lenient and strict parsing modes. Lenient mode is more permissive and is the default. Strict mode applies stronger validation rules for environments that require more predictable and constrained input.
- **`@yini` Document Marker:** A YINI document supports a `@yini` marker near the top of the document. It identifies the document as YINI and MAY optionally declare the expected parsing mode.
- **Document Terminator:** YINI supports an optional document terminator (`/END`); see Section 3.5 for mode-specific requirements. When present, it provides an unambiguous signal of document completion, which is especially helpful for streaming parsers and documents embedded within larger payloads.

1.5. Terminology

These terms are used throughout this specification to describe YINI's grammar, structure, and semantics. The list also includes a few general English terms used with specific meaning in this document.

Term	Definition
Array	In JSON, and in some markup and many programming languages a list is called an *array*. In YINI, the preferred term is *list*. For more details, see: List.
Backticked identifier	An identifier enclosed in backticks, such as `` `display name` ``, allowing names that cannot be written as simple identifiers.
Boolean literal	A case-insensitive literal representing `true` or `false`, such as `true`, `false`, `yes`, `no`, `on`, or `off`.
Canonical form	The preferred syntax form for a construct, used by documentation and recommended for formatters.
Classic string	A string literal prefixed with `C` or `c`, where supported escape sequences are interpreted.
Comment	Text ignored by the parser and not represented in the parsed data model.
Compound value	A value that contains one or more nested values. In YINI, lists and inline objects are compound values.
Configuration	A structured set of sections and members representing settings or data.
Delimiter	A character (or sequence of characters) used to show where something starts, ends, or is separated from something else.
Diagnostic	A warning or error reported by a parser or implementation.
Disabled line	A line whose first non-whitespace characters are `--`; the entire line is ignored by the parser.
Document terminator	The `/END` marker that explicitly marks the end of a YINI document. It is optional in lenient mode and required in strict mode.
Explicit	Written or stated directly in the source text, code, or specification.
Identifier	A name used for a key, object member key, or section name.
Implementation	A parser, validator, formatter, syntax highlighter, tool, library, or host integration that supports YINI.
Implicit	Inferred from surrounding syntax, context, or defined rules rather than written directly.
Inline object	A compound value enclosed in `{` and `}`, containing zero or more object members.

Term	Definition
Key	The identifier on the left side of a member assignment. The identifier can either be *simple* or *backticked*.
Lenient mode	The default parsing mode. It is more permissive than strict mode, while still preserving predictable parsing behavior.
List	An ordered compound value enclosed in `[` and `]`, containing zero or more comma-separated values. In JSON and many programming languages, this structure is commonly called an *array*.
Logical line	A line of YINI source after recognizing supported line endings (`LF`, `CRLF`, or `CR`). Some constructs require tokens to appear on the same logical line.
Member	A **key-value** (pair) entry at root or section level, written as `key = <value>`. For example: `num = 100` or `str = "Hello"`.
Mode declaration	An optional declaration in the YINI marker, such as `@yini strict` or `@yini lenient`, that states the document's expected parsing mode.
Nested section	A section contained under another section, represented by a greater section depth.
Null literal	The case-insensitive literal `null`, representing the null value.
Number literal	A numeric value written using decimal notation or one of the supported base-prefixed forms.
Object member	A key-value entry **inside an inline object**, normally written as `key: <value>`. For example: `num: 100` or `str: "Hello"`. In lenient mode only, `key = <value>` may also be accepted.
Orphan member	A root-level member not contained in an explicit section.
Parser	Software that reads a YINI document and produces a parsed data representation, or reports diagnostics.
Raw string	A string literal that does not interpret escape sequences. Raw string behavior is the default.
Root-level member	A member written outside any explicit section. Root-level members are allowed only in lenient mode.
Scalar value	A non-compound value: string, number, boolean, or null.
Section	A named grouping of members and/or nested sections.
Section depth	The nesting level of a section. A top-level section has depth 1.

Term	Definition
Section header	A line that introduces a section, using a section marker such as `^`, `§`, `>`, or `<`.
Section marker	A character used to introduce a section header. The primary section marker is `^`.
Simple identifier	An identifier written without backticks, using only letters, digits, and underscores, and not beginning with a digit.
Strict mode	An optional parsing mode that enforces stricter structural validation, including exactly one explicit top-level section, no top-level orphan members, and a required document terminator (`/END`). It is not the default mode.
String literal	A quoted text value. YINI strings MUST be quoted.
Top-level section	A section at depth 1. Strict mode requires exactly one explicit top-level section.
Triple-Quoted String	A string literal enclosed in `""" ... """` that may span multiple lines.
Value	The data assigned to a key. The value can be a string, number, boolean, null, list, or inline object.
YINI document	A complete, self-contained YINI configuration input. The term "file" is used informally when a YINI document is stored on disk.
YINI file	Informal term for a YINI document. See: YINI document.
YINI marker	The optional `@yini` marker near the top of a document, used to identify the document as YINI and optionally declare the expected parsing mode.

2. File Structure

A YINI file consists of an explicit document structure based on sections, members, and values. It may also contain comments, but indentation does not determine structure. 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 documents MUST be encoded in UTF-8.

- **BOM Policy:** A UTF-8 byte order mark (BOM) SHOULD NOT be used. However, implementations MAY accept and ignore an initial UTF-8 BOM for compatibility.

- **Character Set:** YINI documents use Unicode. Control characters and other non-printable characters SHOULD be avoided except where explicitly allowed, such as tabs, spaces, and line endings, or when represented through valid string literal forms. Other special characters may be included using escape sequences, or by placing them inside Raw or Triple-Quoted Strings.

For details about whitespace and control characters, see Section 3.2, "Whitespace and Indentation".

2.2. File Extension

YINI files SHOULD use the `.yini` file extension. This extension makes it easier to identify the file type and ensures proper handling by tools and parsers designed for the YINI format.

2.2.1. Strict-Mode Filename Suffix

YINI files meant to be parsed and validated in strict mode MAY use the filename suffix `.strict.yini`.

This suffix is a naming convention only. It does not automatically switch the parser into strict mode. The parser, tool, API, command-line option, or host application configuration remains responsible for selecting the active parsing mode.

Implementations MAY use the `.strict.yini` suffix to provide helpful diagnostics. For example, if a file named `config.strict.yini` is parsed in lenient mode, an implementation SHOULD report a warning or mode-mismatch diagnostic.

The `.strict.yini` suffix does not replace `@yini strict`, and it does not replace strict-mode validation. A file using this suffix MUST still satisfy all strict-mode requirements when parsed in strict mode.

2.3. Optional Shebang (`#!`)

A shebang line MAY appear only as the first line of a YINI document.

If a shebang is present, the shebang marker MUST be the first two non-BOM characters of the document: `#!`. An optional UTF-8 BOM MAY appear before it if the implementation supports BOM handling.

The shebang line is ignored by the YINI parser.

A `#!` sequence that appears after leading whitespace, after a blank line, or anywhere other than the first line MUST NOT be treated as a shebang.

If such a misplaced shebang-like sequence appears where `#` would otherwise begin a comment, it MUST be treated as a comment.

- In lenient mode, a misplaced shebang-like sequence SHOULD be reported as a warning.
- In strict mode, a misplaced shebang-like sequence SHOULD be reported as an error.

- To avoid repeated diagnostics and unnecessarily complex implementation logic, an implementation SHOULD report only the first misplaced shebang-like sequence per document.

Example:

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

^ Config
key = "value"
```

This is not a valid shebang: `#!/usr/bin/env yini`, because it has leading whitespace. It is treated as a comment in lenient mode, but SHOULD produce a warning in lenient mode, and SHOULD produce an error in strict mode.

2.4. YINI Marker (`@yini`)

The optional YINI marker (`@yini`) MAY be used and is RECOMMENDED for clarity and identification. If present, the YINI marker MUST appear before any meaningful YINI content. It MAY be preceded only by a shebang line, comments, or whitespace. (If both a shebang and a YINI marker are present, the shebang MUST appear first.)

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"
```

2.4.1. Optional Mode Declaration

The YINI marker MAY include an optional mode declaration.

Supported mode declarations are:

```
@yini strict
@yini lenient
```

The marker and mode declaration are case-insensitive. Therefore, `@yini strict`, `@YINI STRICT`, and `@Yini Strict` are equivalent.

A mode declaration states which parsing mode the document expects. It forms part of the document contract, but it does not automatically switch the parser into that mode. This is intentional: the parser, tool, API, command-line option, or host application configuration remains responsible for selecting the active parsing mode.

Mode declarations are not treated as ordinary syntax compatibility. They are a guard against accidentally parsing a document under weaker or stronger rules than the author intended.

If the declared mode does not match the active parser mode, the parser MUST report a **mode mismatch diagnostic**. A document declaring `@yini strict` is invalid when parsed in lenient mode. A document declaring `@yini lenient` MAY still be parsed in strict mode, but the parser MUST emit a warning diagnostic.

Declaration	Parsed in lenient mode	Parsed in strict mode
`@yini strict`	❌ Error	✅ Valid mode match
`@yini lenient`	✅ Valid mode match	⚠️ Valid + mode mismatch warning
`@yini`	✅ Allowed	✅ Allowed
No marker	✅ Allowed	✅ Allowed

The parser MUST already be operating in the selected active mode. It MUST NOT silently switch modes because of the declaration.

A mode declaration does not by itself make a document valid. A document declaring `@yini strict` MUST still satisfy all strict-mode requirements.

Important: The parsing mode is always determined by external configuration — the parser, tool, API, CLI option, or host application — regardless of whether a mode declaration is present.

3. Syntax Overview

The syntax of YINI is designed to emphasize clarity, readability, predictability, and explicit structure in configuration files. This section provides a high-level overview of the principal syntax rules defined by the format.

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. In section headers, whitespace between the marker and the section name is optional in repeated/basic form, but required in numeric shorthand form (to make it clear where the marker ends and where the name starts). Outside strings and backticked identifiers, spaces and tabs are generally insignificant except where this specification explicitly requires or forbids whitespace, such as in numeric shorthand section headers or between tokens. Tabs and multiple spaces MAY be used for visual alignment and readability where whitespace is otherwise insignificant.

For definitions of whitespace and control characters, see Section 3.2, "Whitespace and Indentation".

Sections

YINI files support sections, which group related members. A section begins with a section header, marked by one of the allowed section markers (commonly `^`) or by a numeric shorthand form. In the repeated/basic form, whitespace between the marker and the section name is optional. In the numeric shorthand form, at least one space or tab is required after the number.

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"
```

Inside inline objects (`{ ... }`), object members use a colon (`:`) as the canonical separator. This distinction from `=` exists because the inline object as a whole, including its contents, is treated as the **value** of the surrounding member; the entries inside the inline object are therefore member definitions within that value.

This follows a familiar convention from languages and formats such as JavaScript, TypeScript, JSON, Python, YAML, Ruby, and Swift, where `:` is commonly used for inline object, map, dictionary, or hash members.

```
config = { enabled: true, retries: 3 }
```

In **lenient mode only**, `=` MAY also be accepted inside inline objects, but tools and formatters SHOULD normalize inline object members to `:`.

Comments

YINI primarily 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, at least one space or tab is required between a numeric shorthand section marker (such as `^7`) and the section name, so it becomes e.g. `^7 Section`.

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 generally insignificant outside string literals and backticked identifiers, except where this specification explicitly requires or forbids them, such as in numeric shorthand section headers and between distinct lexical tokens.

- Indentation using whitespace is allowed purely for visual clarity — it has no effect on parsing or structure.

3.3. Comments

YINI supports three categories of comments, **using four comment syntaxes:**

Syntax	Type	Valid position
`//`	Line comment	**Anywhere whitespace is allowed**; ignores the rest of the line
`#`	Line comment	**Anywhere whitespace is allowed**; ignores the rest of the line
`/* ... */`	Block comment	**Anywhere whitespace is allowed**; may span multiple lines
`;`	Full-line comment	Only at the beginning of a logical line, after optional spaces or tabs

These rules apply:

- Comments are ignored by the parser and have no effect on the parsed data model.
- Comment markers are recognized only outside string literals and outside backticked identifiers. A comment marker appearing inside a quoted string, triple-quoted string, Classic string, or backticked identifier is treated as ordinary content.
- Line comments begin at `//` or `#` and continue until the next line ending.
- Block comments begin at `/*` and end at the next `*/`. Nested block comments are not supported.
- Comments may appear only where whitespace is allowed. A comment MUST NOT split an identifier, keyword, number literal, string prefix, string literal delimiter, section marker, or other token.
- A line comment may begin immediately after a complete token; no whitespace is required before `//` or `#`.

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.

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 — outside string literals, everything from `//` to the end of the line is treated as a comment:

```
key = "value" // This is an inline comment.
key = true//This is also an inline comment.
```

- **Hash `#` comments** are supported too — outside string literals, everything from `#` to the end of the line is treated as a comment:

```
key = "value" # This is an inline comment.
key = true#This is also an inline comment.
```

Outside string literals, the `#` character always begins a comment. Everything from `#` to the end of the line is ignored by the parser. No whitespace is required before or after `#`.

✅ **Valid `#` comments:**

- `# This is a comment`
- `# Also valid`
- `#\tTabbed too`
- `#Comment too` — Valid with or without horizontal whitespace.
- `#FF9900` — Interpreted as a comment.
- `##` — Also a comment.

Because `#` always begins a comment outside string literals, it cannot appear inside identifiers, prefixes, or other tokens.

```
^ MySection # comment      // Valid: comment after a section header.
0xFF#comment               // Valid: comment after a complete value.

na#commentme = "Kim"       // Invalid: `#` begins a comment after `na`, leaving an incomplet
hex:#FF                    // Invalid: `#` begins a comment before any hexadecimal digits.
C#comment"hello"           // Invalid: `#` begins a comment after `C`, splitting the string
```

3.3.2. Multi-line Block Comments

YINI supports multi-line block comments.

Block comments begin with `/*` and end with `*/`. They may span multiple lines.

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

3.3.3. Full-line Comments

A semicolon (`;`) begins a full-line comment only when it is the first non-whitespace character on a line. Spaces and tabs may appear before it. A semicolon after meaningful content does not begin an inline comment.

```
; Valid full-line comment
   ; Valid full-line comment with leading whitespace

key = "value" ; INVALID: Semicolon is not an inline comment marker.
```

If `;` appears after meaningful content outside a string literal, it is not treated as a comment marker and MUST result in an error unless otherwise permitted by a future extension.

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 member assignments) and for section names (in section headers).

An ***identifier*** can be written in one of the following forms:

- **Form 1: Simple identifier**
    - A simple identifier MUST be non-empty.
    - A simple identifier is case-sensitive (`Title` and `title` are different).
    - A simple identifier can contain only letters (`a-z` , `A-Z`), digits (`0-9`), and underscores (`_`).
    - A simple identifier MUST begin with a letter or an underscore (`_`).
    - A simple identifier MUST NOT contain hyphens (`-`) or periods (`.`). To use such characters, the identifier MUST be written as a backtick-quoted identifier (form 2).

  Example:

  ```
  name
  ```

- **Form 2: Backticked Identifier:**
    - The identifier MUST start and end with a backtick <code>`</code>.
    - A backticked identifier MUST be written on a single logical line. It MUST NOT contain raw tabs, raw newlines, or control characters. Special control characters MUST NOT appear in backticked identifiers.
    - Ordinary spaces are allowed.

- Special control characters (U+0000–U+001F) MUST be escaped and MUST NOT appear as raw characters.
- A backtick-quoted identifier MAY be empty (<code>``</code>). This permits representing an identifier equivalent to the JSON empty key <code>""</code>. Although permitted, empty identifiers are discouraged because they reduce readability.

Example:

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

3.5. Document Terminator

Mode requirement: The document terminator is optional in lenient mode and required in strict mode.

Mode	`/END` requirement
Lenient mode	Optional
Strict mode	Required

A YINI document MAY end with a terminator line in lenient mode.
A YINI document MUST end with a terminator line in strict mode.

The document terminator explicitly marks the end of the configuration content and reduces ambiguity about whether the document was fully read. In strict mode, this makes document completion explicit rather than relying on end-of-file alone, **which enables implementations to detect truncated, partially copied, or prematurely cut-off documents.**

The default and recommended terminator is:

```
/END
```

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

The terminator MAY be preceded by spaces or tabs, but the canonical form has no leading whitespace.

After the document terminator, only whitespace and comments are permitted.
Any other content appearing after the terminator MUST result in an error.

3.6. Disable Line

A disabled line is a line whose first non-whitespace characters are `--`. The entire line, including any otherwise valid YINI syntax, MUST be **ignored** by the parser.

Disabled lines are non-meaningful content. They do not contribute to empty-document detection.

Non-normative note: Disabled lines are intended for temporarily excluding configuration lines without deleting them. Syntax highlighters MAY style disabled lines differently from comments.

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
```

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 the `=` 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

Currently, YINI value types map 1-to-1 to native JSON types. Constructs such as date-time values **SHOULD currently be expressed as strings**. See more in 11.1.3, "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: At root and section level, values are assigned using `=`. The colon (`:`) is not a general assignment operator and MUST NOT be used to define root-level or section-level members or lists.

Inside inline objects (`{ ... }`), `:` is the canonical member separator. In lenient mode only, `=` MAY also be accepted inside inline objects as a compatibility convenience. In strict mode, inline object members MUST use `:`.

For compound values written after `=`, the opening `[` or `{` **MUST appear on the same logical line** as the `=`. A newline immediately after `=` means the member has no explicit value.

Strings

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

Unquoted text MUST NOT be interpreted as a string literal, even in lenient mode.

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).
- YINI also supports explicit base-prefixed number formats such as binary, octal, duodecimal, and hexadecimal literals. See Section 7.4, "Number Formats".

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, separated by commas, the entire value is treated as a List. List items MAY be any supported YINI value type: string, number, boolean, null, list, or inline object.

Null

If the value is the keyword `null` (case-insensitive), or if a root-level or section-level member has no value after `=` in lenient mode, it is treated as **Null**.

Missing values inside lists or objects are not treated as `Null` . A trailing comma inside a list or object is permitted only in lenient mode and is ignored; in strict mode it is an error.

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 is a line that:

- Starts with one or more section marker characters (`^` , `§` , `>` , or `<`), or with a numeric shorthand such as `^7` .
- Is followed by a section name, which may be either a simple identifier or a backticked identifier.
- Ends at the newline.

Spacing rules:

- In the repeated/basic form, no space is required between the marker and the section name.
- In the numeric shorthand form, at least one horizontal space is required after the number before the section name.

Examples:

```
^ SectionName
^SectionName2
^^ Database
^^Database2

^7 DeepSection
§100 `Very Deep Section`
<12 Title
```

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

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. It is within the 7-bit ASCII range and is the most portable option.

- `§` — Supported Unicode aesthetic option for authors who prefer it and know their toolchain handles it correctly. It may be less portable because it lies outside the 7-bit ASCII range.
- `>` — (Not recommended) Quote-like ASCII fallback marker. It is portable, but authors should **be aware** that some email clients, forum renderers, and Markdown-like environments may treat it as a quote prefix.
- `<` — (Not recommended) Explicit ASCII fallback marker. It provides a quote-safe counterpart to `>` for plain-text environments where `§` is unsuitable and `>` may be treated as a quote prefix.

Recommended preference order is: **`^` first, then `§` when Unicode aesthetics are desired** and supported, then `>` or `<` only when an ASCII fallback is needed.

When using a repeated marker to indicate nesting, a maximum of nine (`9`) repeated markers is allowed.

That is, the following denote nesting levels 1–9 when using the primary marker:

```
^         ← level 1
^^        ← level 2
^^^       ← level 3
^^^^      ← level 4
^^^^^     ← level 5
^^^^^^    ← level 6
^^^^^^^   ← level 7
^^^^^^^^  ← level 8
^^^^^^^^^ ← level 9
```

The same repeated-marker depth rule applies to all supported section markers.

Using ten or more section marker characters in a repeated marker header is invalid.
For example, `^^^^^^^^^^ Section` is invalid and MUST be written using numeric shorthand instead, such as `^10 Section`.

For nesting levels deeper than 9, the **numeric shorthand section header** syntax MUST be used (see Section 5.3.2).

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 level, the document may return directly to any previously established shallower level.

- Repeated section marker (`^`, `§`, `>`, or `<`) headers may only be repeated up to nine times — to level 9 (maximum).
- Beyond level 9, the numeric shorthand section MUST be used (see section 5.3.2).

- **Going deeper (increase nesting):** Must increment exactly one level at a time. E.g.: `^^` → `^^^` but not `^^` → `^^^^`.
  This rule applies equally to **repeated marker** form and **numeric shorthand** form. Numeric shorthand does not permit skipping intermediate levels.
- **Going shallower (decrease nesting):** May drop directly to any previous level. E.g.: `^9` → `^^` or `^9` → `^`.
- Numeric shorthand MAY be used for any level. For levels 10 and deeper, numeric shorthand MUST be used.
- The maximum supported section depth is 255 levels.

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

Optionally, indentation may be omitted:

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

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

Although indentation has no semantic meaning, authors SHOULD visually indent nested section headers for 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. Section Marker Separators

For readability, underscores (_) MAY be placed between repeated section markers. These underscores are visual separators only and do not change the section depth. Section marker

separators are supported only in repeated section marker form. They MUST NOT be used in numeric shorthand section headers.

Rules:

- An underscore MAY appear only between two repeated occurrences of the same section marker character.
- An underscore MUST NOT appear at the beginning or end of the section marker sequence.
- An underscore MUST NOT appear adjacent to another `_`.
- Section marker separators are supported only in repeated section marker form, not in numeric shorthand form.
- Section marker separators do not count toward the section depth. Only actual section marker characters are counted.

For example, the following are equivalent:

```
^^^^^ Section    // depth 5
^^_^^_^ Section  // depth 5

^^^^^^ Section   // depth 6
^^^_^^^ Section  // depth 6

^^^^^^^^^ Section   // depth 9
^^^_^^^_^^^ Section // depth 9
```

These following examples are all invalid:

```
^^_ Section      // ❌ Invalid trailing underscore.
^__^ Section     // ❌ Invalid adjacent underscores.

_^ Section       // ❌ Invalid leading underscore.
^_ Section       // ❌ Invalid trailing underscore.

^_< Section      // ❌ Invalid: section marker characters must not be mixed.
^1_0 Section      // ❌ Invalid: separators are not allowed in numeric shorthand section hea
```

5.3.2. Shorthand Section Heading

Shorthand Section Headings: Numeric shorthand is required for nesting levels greater than 9. The syntax is `<marker><n>`, where `<marker>` is one of the allowed section marker characters (`^`, `§`, `>`, `<`) and `<n>` is an integer ≥ 1 indicating the nesting level. For levels 1–9, repeated markers such as `^`, `^^`, `^^^` are RECOMMENDED. (Using the shorthand is optionally valid for levels 1–9 as well, though repeated markers are RECOMMENDED but not required).

For example:

- To go from depth 9 to depth 10: write `^10 SectionName`.
- To go from depth 10 to depth 11: write `^11 SectionName`.
- To go from depth 11 to depth 12: write `^12 SectionName`.
- And so on...

This prevents arbitrarily long runs of the same marker beyond level 9.

✅ **Valid examples:**

Repeated marker form for levels 1–9:

```
^         Level1      // depth 1
^^        Level2      // depth 2
^^^       Level3      // depth 3
^^^^      Level4      // depth 4
^^^^^     Level5      // depth 5
^^^^^^    Level6      // depth 6
^^^^^^^   Level7      // depth 7
^^^^^^^^  Level8      // depth 8
^^^^^^^^^ Level9      // depth 9
```

For deeper repeated marker headers, `_` separators may make the section depth easier to read:

For example:

```
^^^_^       Level4      // depth 4
^^^_^^      Level5      // depth 5
^^^_^^^     Level6      // depth 6
^^^_^^^_^   Level7      // depth 7
^^^_^^^_^^  Level8      // depth 8
^^^_^^^_^^^ Level9      // depth 9
```

Numeric shorthand form for levels deeper than 9:

```
^         Level1      // depth 1
^^        Level2      // depth 2
^^^       Level3      // depth 3
^^^^      Level4      // depth 4
^^^^^     Level5      // depth 5
^^^^^^    Level6      // depth 6
^^^^^^^   Level7      // depth 7
^^^^^^^^  Level8      // depth 8
^^^^^^^^^ Level9      // depth 9
^10       Level10     // depth 10
^11       Level11     // depth 11
```

Numeric shorthand may also be used for levels 1–9:

```
^1      Level1      // depth 1
^2      Level2      // depth 2
^3      Level3      // depth 3
^9      Level9      // depth 9
```

Going back to a shallower level is allowed:

```
^       Level1
^^      Level2
^^^     Level3
^       BackTo1     // allowed: returns to depth 1
^^      BackTo2     // allowed: descends from depth 1 to depth 2
```

❌ **Invalid examples:**

Skipping an intermediate level is invalid:

```
^       Level1
^^^     Level3      // ❌ Invalid: depth 2 was not established
```

The same rule applies to numeric shorthand:

```
^1      Level1
^2      Level2
^9      Level9      // ❌ Invalid: depths 3 through 8 were not established
```

A numeric shorthand section header requires whitespace after the number:

```
^7Level7            // ❌ Invalid: shorthand requires at least one space or tab after the num
```

Using ten or more repeated section markers is invalid:

```
^^^^^^^^^^ Level10 // ❌ Invalid: use numeric shorthand instead
```

Numeric shorthand does not permit skipping intermediate nesting levels. It is only an alternative notation for expressing a section depth and a notation for going deeper than level 9. Therefore, a shorthand section at level `n` is valid ONLY if level `n - 1` has already been explicitly established in the current nesting chain.

6. String Literals

Prefix Glossary Table:

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

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)
C	Classic String	Supports escape sequences like `\n`, `\t`

Prefix R is optional and has no functional effect — raw strings are the default.

YINI defines two string behaviors and two delimiter forms:

- Behavior:
  - Raw behavior: escape sequences are not interpreted.
  - Classic behavior: escape sequences are interpreted.
- Delimiter form:
  - Single-line quoted form: `'...'` or `"..."`.
  - Triple-quoted form: `"""..."""`.

Combining these gives Raw Strings, Classic Strings, Raw Triple-Quoted Strings, and C-Triple-Quoted Strings.

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

Quote character note: In this specification, the quote characters used to delimit strings are the plain ASCII quote characters:

- Single quote: `'` U+0027 APOSTROPHE
- Double quote: `"` U+0022 QUOTATION MARK

Typographic or curly quotation marks such as ‘, ’, “, and ” are ordinary Unicode text characters. They do not begin or end YINI string literals.

Note: Text MUST be enclosed in quotation marks to be parsed as a string literal.

YINI supports a number of string literal forms: single-line quoted strings, Classic strings, and Triple-Quoted strings. String behavior is determined by the optional prefix before the opening quote sequence.

The `R` prefix is optional and has no semantic effect because Raw strings are the default. The `C` prefix enables escape-sequence interpretation.

Triple-Quoted Strings may also use the `R` or `C` prefix. Without a prefix, a Triple-Quoted String is Raw by default. YINI supports multi-line strings via Triple-Quoted Strings.

String behavior is controlled by an optional prefix before the opening quote sequence:

- No prefix: Raw string behavior.
- `R` or `r`: Raw string behavior; allowed for explicitness only.
- `C` or `c`: Classic string behavior; escape sequences are interpreted.

This applies to both single-line quoted strings and Triple-Quoted Strings. Triple-Quoted Strings without a prefix are Raw by default.

Rules and Behavior for Strings:

- All string literals **MUST start and finish on the same line**, except for **Triple-Quoted Strings** (see section 6.3.), which can span multiple lines.
- Multiple string literals can be **concatenated** to create longer strings (see section 6.5, "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
Triple-Quoted (Raw)	`"""..."""`	✅ Yes	❌ No	❌ No	Multi-line raw text	Python raw triple-quote
C-Triple-Quoted	`c"""..."""` or `c"""..."""`	✅ Yes	✅ Yes	❌ No	Multi-line with escapes	Python triple-quote

6.1. Raw Strings (R-Strings)

A raw string is enclosed in either single quotes (`'...'`) or double quotes (`"..."`).

Raw strings do not interpret escape sequences, backslashes (`\`) are treated as ordinary characters.

To include a single quote, use double quotes as the delimiter. To include a double quote, use single quotes as the delimiter.

Raw strings MUST NOT contain line breaks. For multi-line raw text, use a Triple-Quoted String.

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

Examples:

```
single_quote = "Amanda's file"
double_quote = 'He said "hello"'

path_raw1 = "C:\Users\Kim\Documents"  // Raw string
path_raw2 = "D:\Users\John Smith\"    // The final backslash is part of the string content.
path_raw3 = '/home/Leila Häkkinen'
path_raw4 = '/Users/kim-lee'
```

The value of `path_raw2` is:

```
D:\Users\John Smith\
```

Important: In Raw strings, a backslash before the closing quote does not escape the quote.

Curly quotation marks may appear inside quoted strings as ordinary text:

```
meaning = "It comes from Greek akmḗ (ἀκμή), meaning “the highest point” or “best”."
```

In the above example, the outer `"` characters are plain ASCII double quotes and delimit the string. The inner `“` and `”` characters are typographic quotation marks and are part of the string value.

Raw String Prefix

Any string enclosed in single quotes (`'`) or double quotes (`"`) MAY be prefixed with `R` or `r` to explicitly mark it as a Raw String. This prefix is optional because strings are raw by default.

6.2. Classic Strings (C-Strings)

A Classic String is a string literal prefixed with `C` or `c`. Classic Strings interpret escape sequences before the final string value is produced. The parsed value is a normal string; the `C` prefix is not preserved in the resulting data model.

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 logical line.

Examples:

```
text1 = C"hello\nworld"

text2 = c"This is a newline \n and this is a tab \t character."
```

Both `text1` and `text2` parse to a string containing an actual newline, not the two characters `\` and `n` or `\t`.

6.2.1. Escape Sequences

Escape sequences are supported only in Classic Strings (C-Strings) and C-Triple-Quoted Strings.

Unicode escape sequences MUST represent valid Unicode scalar values.
Surrogate code points in the range U+D800 to U+DFFF are invalid.

Full list of escape sequences

The following escape sequences are case-sensitive:

- `\\` — Backslash.
- `\'` — Single quotation mark.
- `\"` — Double quotation mark.
- `\/` — Normal/forward slash (yields a plain `/`).
- `\0` — Null byte.
- `\?` — Literal question mark, provided 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, using exactly 2 hexadecimal digits.
- `\uhhhh` — Unicode code point in the range U+0000 to U+FFFF, using exactly 4 hexadecimal digits.
- `\Uhhhhhhhh` — Unicode code point in the range U+00000000 to U+10FFFF, using exactly 8 hexadecimal digits.
- `\oOOO` — Octal value (up to 3 digits, valid range `\o0` to `\o377`):
  - Equivalent in value to `\OOO` in C/C++.
  - `\o0` has the 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, such as `\z` or `\o378`, MUST result in a parse error in standard YINI parsing.

Additional recommendation: Implementations SHOULD provide helpful diagnostics for unsupported escape-sequence forms. For example, C/C++-style octal escape sequences such as `\1` through `\377` are not valid YINI escape sequences. In strict mode, such sequences SHOULD be reported as errors. In lenient mode, implementations MAY recover by interpreting them as the corresponding YINI octal escape form, but any such recovery MUST be reported through diagnostics.

6.3. Triple-Quoted Strings

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

- **Begins** with `"""` and **ends** at the next `"""` sequence that is recognized as the closing delimiter.
- **May span multiple lines**, including embedded newline characters.
- In a Raw Triple-Quoted String, the sequence `"""` cannot appear as content because it terminates the string. In a C-Triple-Quoted String, quote characters may be escaped using `\"`.
- Without a prefix, a Triple-Quoted String is Raw: escape sequences are not interpreted.
- With a `C` or `c` prefix, escape sequences are interpreted using the same rules as Classic Strings.
- With an `R` or `r` prefix, behavior is identical to the unprefixed Raw form. The `R` prefix is allowed only for explicitness.

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 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'"""
```

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.

6.4. String Types Summary

YINI string behavior is determined by two independent choices:

1. The optional string prefix:
    - No prefix, `R`, or `r` means Raw string behavior.
    - `C` or `c` means Classic string behavior, where escape sequences are interpreted.
2. The delimiter form:
    - Single-line quoted strings use `'...'` or `"..."`.
    - Triple-Quoted Strings use `"""..."""` and may span multiple lines.

Raw strings are the default. The `R` prefix is allowed for explicitness only and has no semantic effect. The `C` prefix enables escape-sequence interpretation.

For the full table of supported string forms, see the overview table at the beginning of Section 6.

6.5. String Concatenation

YINI supports explicit string **concatenation** using the plus sign (`+`).

Concatenation joins two or more operands into a single string value. The result of concatenation is always a string.

Example:

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

The result is equivalent to:

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

Concatenation Operands

The `+` operator in YINI is exclusively a string concatenation operator. YINI does not define numeric addition. For readability, whitespace around `+` is recommended, but not required.

In strict mode, all concatenation operands MUST be string literals. This includes Raw/R-prefixed, Classic/C-prefixed, or any Triple-Quoted strings.

In lenient mode, a concatenation expression MUST begin with a string literal. Additional operands MAY be string literals, number literals, boolean literals, or null literals. Numeric addition is not performed. Non-string scalar operands MUST first be converted to their parsed scalar canonical string representation before concatenation.

Lists and inline objects MUST NOT be operands in concatenation expressions.

Each string literal is interpreted according to its own string type before concatenation. For example, Classic Strings interpret escape sequences before being joined, while Raw Strings preserve backslashes literally.

Valid in both lenient and strict mode:

```
escaped = C"Line 1\n" + "Line 2"

longText = "This is a long string that is split " +
           "across multiple source lines."
```

Valid in lenient mode only:

```
str1 = "Port: " + 8080
// "Port: 8080"

str2 = "1" + 2 + 3
// "123"

str3 = "a" + +42
// "a42"
```

Invalid in both modes:

```
invalid1 = 1 + 2 + 3          // Invalid: YINI does not define numeric addition.
invalid2 = 8080 + " is port"  // Invalid: concatenation must begin with a string literal.
invalid3 = 1 + 2 + "3"        // Invalid: concatenation must begin with a string literal.
```

Rationale: Requiring concatenation to begin with a string literal avoids ambiguity between numeric-looking expressions and string concatenation. For example, `1 + 2 + "3"` could otherwise be misread as either `"123"` or `"33"`. YINI rejects such expressions instead, because `+` is not a numeric addition operator.

Multi-line Concatenation

A line break MAY occur after the `+` operator. This allows long string values to be split across multiple source lines without using a Triple-Quoted String.

A line break MUST NOT occur before the `+` operator. The `+` operator MUST appear on the same logical line as the preceding operand.

Valid:

```
longText = "This is a long string that is split " +
           "across multiple source lines, but " +
           "the resulting value is still one string."
```

Also valid:

```
title = "YINI: " + "A human-friendly configuration format"
```

Invalid:

```
message = "hello "
        + "world"  // ❌ Invalid: newline before +
```

For longer human-authored text where exact line breaks should be preserved, authors SHOULD use Triple-Quoted Strings instead. Triple-Quoted String literals MAY still be used as operands in a concatenation expression.

6.6. Scalar Conversion in Lenient Concatenation

Scalar conversion to strings is allowed only in lenient-mode concatenation expressions.

Conversion is based on the **parsed scalar value**, not the original lexical spelling in the source document. **This keeps parsing predictable, avoids hidden rules, and produces stable output.**

In lenient mode, scalar operands are **converted to strings before concatenation** as follows:

- Strings use their interpreted string value.
- Numbers are converted to their canonical textual numeric representation (from their converted parsed numeric value). For example:
  - `42` becoming `42`.

    - `0xFF` becoming `255`. Non-decimal numeric notation is normalized to decimal form.
    - `-19` becoming `-19`. The minus sign **is preserved** in negative values.
    - `+100` becoming `100`. The plus sign **is NOT preserved** (consumed) in positive values.
- Booleans are converted to `true` or `false`.
- Null is converted to `null`.
- Base prefixes such as `0x`, `0b`, `%`, `0o`, `0z`, and `hex:` are not preserved during scalar-to-string conversion.
- Lists and inline objects MUST NOT be converted to strings and MUST NOT be used as operands in concatenation expressions.

Example:

```
a = "a: " + 9999
b = "n: " + 1_000_000
c = "hex: " + 0xFF

d = "bool: " + YES
e = "null: " + NULL
```

This produces:

```
a: 9999
n: 1000000
hex: 255

bool: true
null: null
```

Multi-line concatenation does not change scalar conversion rules:

```
label = "port-" +
        5432
```

In lenient mode, this produces:

```
port-5432
```

In strict mode, this is invalid because `5432` is not a string literal (it's a number literal).

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. Digit Separators

For readability, an underscore (`_`) MAY be used as a digit separator inside number literals. It MAY appear either immediately after a supported base prefix or between successive digits. Digit separators do not change the numeric value.

Underscores are permitted:

- Between two digits.
- Immediately after a supported base prefix: `0b` , `0o` , `0x` , `0z` , `hex:` , or `%` .
- Between digit groups in base-prefixed number literals, such as `0xFF_AA` , `0b1010_1100` , or `0o755_644` .
- Underscores MAY appear between digits in the integer part, fractional part, or exponent part of a decimal number.

Underscores are not permitted:

- At the beginning or end of a number literal.
- Adjacent to another underscore.
- Inside or between the characters of a base prefix.
- Underscores MUST NOT appear adjacent to a sign character (`+` or `-`), decimal point (`.`), exponent marker (`e` or `E`), or at the beginning or end of the literal.

Examples:

```
decimal = 2_468
binary = 0b_1101
octal = 0o_640
hex1 = 0x_A3
hex2 = 0xCafe_Babe
hex3 = 0x_ab_cd_12_34_ef
hex4 = hex:_FF_AA_00
```

```
binary_grouped = 0b1111_0001
binary_alt = %_1010
octal_grouped = 0o6_4_0
duodecimal = 0z_2EX9
```

Invalid examples:

```
bad1 = 73_        // ❌ INVALID: Trailing underscore.
bad2 = 5__9       // ❌ INVALID: Adjacent underscores.

bad3 = 0_b1101    // ❌ INVALID: Underscore breaks the base prefix.
bad4 = _73        // ❌ INVALID: Leading underscore.

bad_hex1 = 0x_    // ❌ INVALID: Separator without any digits.
bad_hex2 = hex:_  // ❌ INVALID: Separator without any digits.
```

7.3. Exponent Format

Exponent notation uses the format:

```
<significand>e<sign><exponent>
```

Where:

- `<significand>` is an integer or decimal number.
- `e` may be written as `e` or `E`.
- `<sign>` may be `+`, `-`, or omitted.
- `<exponent>` is a sequence of decimal digits, optionally using permitted digit separators.

Example:

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

```
valid_float = 1_000.5_25
valid_exp = 1.5e1_0

bad_float1 = 1_.5     // ❌ Invalid: underscore before decimal point.
bad_float2 = 1._5     // ❌ Invalid: underscore after decimal point.
bad_exp1 = 1_e10      // ❌ Invalid: underscore before exponent marker.
bad_exp2 = 1e_10      // ❌ Invalid: underscore after exponent marker.
```

7.4. Number Formats

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

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`
`0b1010`	`%1010`	Binary number prefix	2-base	Digits: `0` and `1` only
`0o7477`	-	Octal number	8-base prefix	Digits: `0 – 7`
`0z2EX9`	`0z2BA9`	Duodecimal (AKA dozenal) prefix	12-base	`X = A = 10`, `E = B = 11` (case-insensitive)
`0xF390`	`hex:F390`, `HEX:f390`	Hexadecimal number prefix	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`, and `0Xf390` are all valid hexadecimal literals.

```
color1 = hex:fa90
color2 = Hex:Fa90
color3 = HEX:FA90
```

The `#` character is not a hexadecimal prefix in YINI. Outside string literals, `#` always begins a comment.

In the `hex:` form, the value after `hex:` MUST contain hexadecimal digits and permitted digit separators only. The `0x` prefix is not used inside the `hex:` form.

```
color1 = hex:FFAA00     // ✅ valid
color2 = 0xFFAA00       // ✅ valid
color3 = hex:_FF_AA_00  // ✅ valid
color4 = hex:0xFFAA00   // ❌ invalid
color5 = hex: FFAA00    // ❌ invalid
```

8. Boolean and Null Literals

8.1. Booleans

Boolean literals in a YINI document **are case-insensitive** and may be written as follows:

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

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

8.2. Null Literal

The null literal (value) is `null`, case-insensitive.

- In lenient mode, a root-level or section-level member with no value after `=` is interpreted as `null`.
- In strict mode, such a member is invalid, null values MUST be written explicitly as `null`.

```
key = null
```

Missing values inside lists or inline objects are never interpreted as `null`.

Invalid in both lenient and strict mode:

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

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

Trailing commas inside lists and objects do not create `null` values. In lenient mode, a trailing comma is ignored. In strict mode, a trailing comma is invalid and MUST emit an error.

Examples:

```
^ Section1
// In lenient mode:
key1 =                  // ✅ Valid, same as: key1 = null
key2 = [1, 2, ]         // ✅ Valid, same as: [1, 2]
key3 = { a: 1, b: 2, }  // ✅ Valid, same as: {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 compound value type. An inline object contains zero or more object members enclosed in braces `{` and `}`.

An inline object behaves like a map or dictionary. Each object member has a key and a value, and members are separated by commas. Objects may be nested; an object value may itself contain another inline object.

An empty object `{ }` is valid in both lenient and strict mode.

An object has the following canonical form:

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

```
settings = { enabled: true, retries: 3 }
empty = { }
```

Syntax rules

There MUST be **no newline** between the `=` and the opening brace `{`. If a newline appears after `=`, the member is treated as having no explicit value: in lenient mode this is interpreted as `null`; in strict mode it is a missing-value error. The `{ ... }` block on the following line is not treated as the value of that member.

Inside an **inline object**, the value of each object member MUST begin on the same logical line as the object member separator.

✅ Valid:

```
object1 = { a: 1, b: 2 }

object2 = {
    a: 1,
```

```
    b: 2
}
```

❌ Invalid:

```
object1 =
{ a: 1, b: 2 }  // Invalid: `{` does not appear on the same line as `=`.

// Not parsed as an object value for object2
object2 =
    {
        a: 1,
        b: 2
    }
```

Inside inline objects, the value MUST begin on the same logical line as the object member separator.

This is invalid (value does not start on the same line as the **object member separator** `:`):

```
obj = {
    a:
        1  // Invalid: Value and object-member separator `:` not on the same line.
}
```

The **key** in each object member follows the same identifier rules as any YINI key.

The **value** in each object member may be any supported YINI value type:

- String
- Number
- Boolean
- Null
- List
- Inline object

The following structural rules apply to inline objects:

- An inline object begins with `{` and ends with `}`.
- Between { and }, zero or more object members may appear.
- Object members are separated by commas.
- Leading commas are not allowed.
- Empty member slots are not allowed.
- In lenient mode only, a trailing comma after the last member is permitted and ignored.

- In strict mode, trailing commas are invalid.
- Whitespace is ignored except inside quoted strings.
- Line comments may begin after a complete token, even without preceding whitespace. However, a comment marker MUST NOT split a token.
- Object member keys MUST be unique within the same inline object.
  - In lenient mode, the first object member definition wins: the first member MUST be kept, later duplicate object members MUST be ignored, and the implementation MUST report the duplicate as a warning diagnostic.
  - In strict mode, any duplicate object member key MUST result in an error.
  - Implementations MUST NOT silently overwrite an earlier object member with a later one.

The **canonical object member separator is `:`**. See Section 9.2, "Object Member Separators", for the exact separator rules in lenient and strict mode.

Examples:

```
// Valid in both lenient and strict mode.
obj1_a = { a: 1, b: 2 }
obj1_b = {
    a: 1,
    b: 2
}

// ✅ Valid in both lenient and strict mode.
obj2 = { }

// ✅ Lenient mode only: trailing comma is ignored.
obj3_a = { a: 1, b: 2, }
obj3_b = {
    a: 1,
    b: 2,  // Trailing comma here.
}

// ❌ Invalid in both modes: leading comma.
obj4_a = { , a: 1 }
obj4_b = {
    ,
    a: 1
}

// ❌ Invalid in both modes: empty member slot.
obj5_a = { a: 1,, b: 2 }
obj5_b = {
    a: 1,,
    b: 2
}
obj5_c = {
    a: 1,
    ,
```

```
    b: 2
}
```

A complete nested object example:

```
^ System

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" }
            ]
        }
    }
}
```

9.2. Object Member Separators

Inside inline objects, `:` is the canonical member separator. A newline MUST NOT appear between the object member separator and the value.

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

At root and section level, YINI uses `=` for ordinary members:

```
name = "Kim"
```

Inside inline objects, `:` is used since the object itself is already the value of a surrounding member. The colon helps distinguish object member definitions from ordinary root-level or section-level assignments.

In lenient mode only, implementations MAY also accept `=` as an object member separator:

```
obj = { a = 1, b = 2 }  // Lenient mode only
```

This exists as a compatibility and user-friendliness feature, especially for users who are accustomed to writing `key = "value"` throughout a configuration file.

The following rules apply:

- In lenient mode, inline object members MAY use either `:` or `=`.
- In strict mode, inline object members MUST use `:`.
- In strict mode, using `=` inside an inline object is invalid, whether mixed with `:` or used consistently.
- Within a single inline object, mixing `:` and `=` is discouraged in lenient mode.
- Tools and formatters SHOULD normalize inline object members to `:`.

```
// Canonical form, valid in lenient and strict mode.
obj1 = { a: 1, b: 2 }

// Lenient mode only.
obj2 = { a = 1, b = 2 }

// Discouraged in lenient mode; invalid in strict mode.
obj3 = { a: 1, b = 2 }
```

Conceptual grammar:

```
COLON =  ':';
EQUALS = '=';

// Lenient mode:
objectMember
  : KEY hspace* (COLON | EQUALS) hspace* value
  ;

// Strict mode:
objectMember
  : KEY hspace* COLON hspace* value
  ;
```

Where `hspace` means spaces/tabs only, no newlines.

10. List Literals

Lists in YINI correspond to *arrays* in JSON and 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 (`[...]`)

A list is assigned to a key using the equals sign `=`, followed by square brackets `[ ]` containing zero or more comma-separated values. The opening `[` MUST appear on the same logical line as `=`.

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 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]



// ❌ Invalid list literal.
items1 =
[
    "a",
    "b"
]

// ✅ Valid
items2 = [
    "a",
    "b"
]
```

❌ Invalid:

```
list =
["item1", "item2"]  // Invalid: Due to the newline in previous line.
```

✅ 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

The syntax space for anchors and aliases is reserved for possible future use. These constructs would allow users to define reusable fragments within a configuration.

The exact anchor and alias syntax is not defined in this version.

Implementations of this specification MUST NOT assign normative meaning to these reserved constructs unless operating under an explicitly documented extension.

11.1.2. Includes

The `@include` directive name is reserved for possible future file inclusion support.

These features are not implemented in this version. They are reserved for future use and MUST NOT be used for user-defined syntax.

11.1.3. Date-time Type

Currently, the YINI **value 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 backticked identifiers but are otherwise restricted:

Character	Usage Context	Description
`=`	Assignment / lenient object separator	Assigns values to keys at root/section level. In lenient mode only, MAY also separate keys and values inside inline objects.
`^`	Section header	Used to denote section start
`:`	Object member separator	Canonical separator between keys and values inside inline objects
`,`	Item separator	Used in lists
`§`	Section header (Unicode alternative)	Used to denote section start
`>`	Section header (quote-like ASCII fallback)	Used to denote section start
`<`	Section header (ASCII fallback)	Used to denote section start
`/* */`	Block comment	Marks multi-line comment block
`//`	Inline comment	
`#`	Comment	Begins a comment outside string literals; everything from `#` to the end of the line is ignored
`;`	Full-line comment	

Character	Usage Context	Description
`[ ]`	List literal	
`{ }`	Object literal	
`--`	Line disabling	Experimental use (see Section 3.6)
`%`	Binary prefix	Begins binary number
`hex:`	Hexadecimal number prefix	Begins an explicit hexadecimal number literal; case-insensitive
`@`	Directive prefix	Reserved for future syntax

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.

12.2.1. Structural Requirements

Note: In lenient mode, top-level members outside any section may be accepted. In strict mode, the document structure is further restricted; see Section 12.3 and Section 13.1.

- In lenient (default) mode, a document may consist of **zero or more root-level members, section-level members, and sections**.
- In strict mode, a document MUST contain **exactly one explicit top-level section**. All members MUST appear inside that section or one of its subsections.
- Section headers MUST begin with a valid marker (`^` , `§` , `>` , or `<`).
- In repeated/basic section headers, whitespace between the marker and the section name is optional.
- In numeric shorthand section headers, at least one space or tab is required after the number before the section name.
- Keys are case-sensitive. Spaces, quotes, and other special characters are allowed only in backticked identifiers.

- **Duplicate keys** within the **same section and nesting level** are not allowed.
    i. In lenient mode, the first definition wins: the first key definition MUST be kept, later duplicate key definitions MUST be ignored, and the implementation MUST report the duplicate as a warning diagnostic.
    ii. In strict mode, any duplicate key MUST result in an error.
    iii. In neither mode may an implementation silently overwrite an earlier key with a later one. This rule reflects an intentionally conservative approach. It avoids surprising behavior in large files and prevents accidental overrides later in the document.
- **Duplicate sections** with the same name under the **same parent section** are not allowed.
    i. In lenient mode, the first section definition wins: the first section MUST be kept, later duplicate section definitions at the same level MUST be ignored, and the implementation MUST report the duplicate as a warning diagnostic.
    ii. In strict mode, any duplicate section at the same level MUST result in an error.
    iii. Implementations MUST NOT silently merge, overwrite, or extend an earlier section with a later section of the same name. If a repeated section definition is rejected, all assignments belonging to that duplicate section block MUST also be rejected or ignored; they MUST NOT be reinterpreted as belonging to another section or context.
- Lines that do not match any syntactic role (member, comment, section, terminator) are considered malformed.
- The document terminator (`/END`) is **optional in lenient (default) mode** and **required in strict mode**.
- **In lenient (default) mode, an empty document is permitted.** A document that contains only whitespace, comments, and/or disabled lines (`--`) is considered empty. In such cases, the parser MUST NOT fail; it SHOULD instead report a warning diagnostic indicating that the document appears empty or contains no meaningful content.
- **In strict mode, an empty document is invalid.** A document that contains only whitespace, comments, and/or disabled lines (`--`) MUST result in an error.

12.2.2. Character Encoding

YINI documents (files) **MUST** be encoded as **UTF-8**.

A UTF-8 byte order mark (BOM) SHOULD NOT be used. Implementations MAY accept and ignore an initial UTF-8 BOM for compatibility.

12.2.3. Line Endings

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

12.2.4. 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`.
- Inline object members MUST use `:` in strict mode. In lenient mode, implementations MAY also accept `=` inside inline objects, but `:` remains canonical.

12.2.5. Document Terminator

Note: The authoritative syntax and placement rules for the document terminator are defined in Section 3.5, "Document Terminator". This section restates the validation requirements by mode.

- In lenient mode, the terminator is optional.
- In strict mode, the terminator is required.
- See Section 3.5, "Document Terminator", for terminator syntax.
- A valid YINI file in **strict mode** MUST end with the terminator marker (`/END`).
- A valid YINI file in **lenient mode** MAY end with the terminator marker (`/END`).
- Only one terminator is permitted per file.
- Missing terminators:
  - **In lenient mode:** No error is required.
  - **In strict mode:** MUST be treated as an error.
- After the terminator, only whitespace and comments are allowed.
- Any other content appearing after the terminator MUST result in an error.

Table: Terminator Requirement by Mode

Mode	Terminator Requirement
Lenient	Optional
Strict	Required

12.2.6. Escaping and String Literals

- Escape sequences are **ONLY allowed** 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).

12.2.7. Shortest Valid YINI Documents in Strict Mode

- **Valid short documents in strict mode:**
  - The shortest valid strict-mode document is:

```
^T
/END
```

Note: This document contains a single explicit top-level section named T and the required document terminator.

- **Invalid short documents:**
  - ❌ Invalid: no title section present in strict mode:

    ```
    /END
    ```

12.3. Lenient vs. Strict Modes

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

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

Lenient Mode

- **Lenient mode** is the default mode of operation.
  - Useful for hand-edited configuration files.
  - Permissive with minor issues (for example, trailing commas after the last value/member are ignored, and mixed line endings are tolerated).
  - The document terminator (`/END`) is optional in lenient mode and MAY be omitted entirely. If present, it marks the end of the YINI document. Any non-comment, non-whitespace content appearing after it MUST result in an error.
  - 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 for members at section top level:
    - A missing/empty value (ONLY outside lists and objects) is treated as `Null` .
    - A trailing comma after the last value/member inside a list or object is ignored ONLY in lenient mode. In strict mode, it is a parse error.
  - Inside inline objects, `=` MAY be accepted as an alternative to `:` for object member separation. This is a lenient mode compatibility feature only. The canonical form remains `key: "value"` .
  - In lenient (default) mode, an empty document is permitted. For this purpose, a document containing only whitespace, comments, and/or disabled lines (`--`) is considered empty. An implementation MUST NOT treat such a document as a parse failure solely because it is empty; however, it SHOULD report a warning diagnostic indicating that the document appears empty or contains no meaningful content.

Strict Mode

- **Strict mode** is an optional mode with stricter rules.
  - Intended for production and tool-chain use.
  - Enforces full well-formedness.
  - Empty values are not allowed; they MUST always be written explicitly as `null`, `Null`, or `NULL`.
  - Stray trailing commas after the last value/member inside lists and objects are not permitted.
  - Strict mode requires exactly one explicit top-level section. Any additional sections MUST appear only as subsections nested within that section. Top-level orphan members are not allowed in strict mode. Otherwise, an error MUST be reported.
  - If a mode declaration is present, it is checked against the active parser mode. If `@yini strict` is parsed in lenient mode, the parser MUST emit a mode-mismatch error. If `@yini lenient` is parsed in strict mode, the parser MUST emit a mode-mismatch warning. Mode declarations do not replace strict-mode validation, and strict mode MAY still be used when the declaration is absent.
  - The document terminator (`/END`) MUST be present in a YINI document in strict mode. It marks the end of the document. Any non-comment, non-whitespace content appearing after it MUST result in an error. **Note:** Because strict mode also requires the document terminator `/END`, the end of the document is made explicit rather than being inferred from EOF alone. This improves deterministic parsing, reduces ambiguity about incomplete input, and makes **truncated, partially copied, or prematurely cut-off documents** easier to detect. Together with the requirement for exactly one explicit top-level section, this provides 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 (for example, `^ Title`).
  - Empty sections (with no members) are still allowed.
  - Inside inline objects, object members MUST use `:`. Using `=` inside an inline object MUST result in an error.
  - Files intended for strict-mode parsing MAY use the `.strict.yini` filename suffix as described in Section 2.2.1. This suffix is only a naming convention and does not select strict mode or replace strict-mode validation. An explicit `@yini strict` declaration MAY still be used when the document should declare its expected mode.
  - In strict mode, an empty document is invalid. For this purpose, a document containing only whitespace, comments, and/or disabled lines (`--`) is considered empty. Parsing such a document MUST result in an error.

Implementations SHOULD clearly document the validation mode in use and describe 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)
object2 = { a = 1, b = 2 }      // ✅ Lenient only; formatter should normalize to { a: 1, b:
object3 = { a: 1, b = 2 }       // ⚠️ Discouraged mixing; formatter should normalize to `:`


; Strict mode examples:
list_bracketed2 = [1, 2, ]      // ❌ Error: stray trailing comma
object4 = { a: 1, b: 2, }       // ❌ Error: stray trailing comma
object5 = { a = 1, b = 2 }      // ❌ Error: `=` not allowed inside inline objects in strict

list_bracketed3 = [1, 2]        // ✅ OK
object6 = { 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	⚠️ First wins	❌ Error	Later duplicate keys are ignored in lenient mode and MUST produce a warning diagnostic.
Duplicate sections at same level	⚠️ First wins	❌ Error	Later duplicate sections are ignored in lenient mode. Implementations MUST NOT merge or overwrite sections.
Exactly one explicit top-level section required	❌	✅	In strict mode, all other sections MUST be nested within it.
Top-level orphan members allowed	✅	❌	In lenient mode they may be mounted at root or under implicit base.
`/END` required at end of document	❌	✅	
`=` inside inline objects	✅	❌	Lenient mode MAY accept `key = "value"` inside `{ ... }`; strict mode requires

Feature	Lenient Mode (Default)	Strict Mode	Notes
			canonical `key: "value"`.
Mixed `:` and `=` inside same inline object	⚠️ Discouraged	❌	Lenient parsers MAY accept mixed separators, but formatters SHOULD normalize all inline object members to `:`.
Trailing commas after value (inside lists/objects)	✅	❌	In lenient mode the comma is ignored, error in strict mode
Missing (empty) value (only in section/root-level)	✅	❌	Will result in a `Null` value in lenient mode
Empty value before comma	❌	❌	Applies to cases such as `key = ,`, `[1, , 2]`, or `{ a: , b: 2 }`. Lenient implementations SHOULD report a warning or error; strict implementations MUST report an error.
Invalid escape sequences	❌	❌	Invalid escape sequences MUST result in an error unless an implementation explicitly documents a non-standard extension.
String concatenation with scalar operands	✅	❌	In lenient mode, a concatenation expression MUST begin with a string literal. Additional operands MAY be string literals, number literals, boolean literals, or null literals.
Multi-line concatenation after `+`	✅	✅	A line break MAY occur after `+`, but MUST NOT occur before `+`.

⚠️ 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 here 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 12.2, "Well-Formedness Requirements", for formal validation criteria.

13.1. Top-Level Sections and Implicit Root

A YINI document may contain both explicit top-level sections and, in lenient mode only, top-level members that are not placed inside any explicit section. Such members are referred to here as **orphan members**.

13.1.1. Lenient Mode: Orphan Members

In lenient mode, orphan members MAY be accepted.
NOTE: In strict mode, orphan members are forbidden.

In lenient mode only, an implementation MUST expose accepted orphan members in a well-defined way. The preferred behavior is to mount orphan members directly onto the parsed result, alongside explicitly defined top-level sections. If the underlying platform, host language, or target representation does not allow this cleanly, orphan members MUST instead be placed under an implicit section named `base`.

Accepted orphan members SHOULD be exposed as direct members of the parsed root object, alongside explicitly defined top-level sections.

For example:

```
name = "App"

^ Server
host = "localhost"
```

SHOULD be represented conceptually as:

```
{
  "name": "App",
  "Server": {
    "host": "localhost"
```

```
    }
  }
```

It SHOULD NOT be represented under an explicit `root` or `base` object unless the implementation cannot directly mix orphan members and top-level sections in its target representation.

When the implicit `base` section strategy is used, `base` becomes a reserved top-level section name for that parsed document. Therefore, if orphan members are present, an explicitly defined top-level section named `base` MUST result in an error.

If orphan members are not present, an explicitly defined top-level section named `base` is treated as an ordinary section name.

The following collisions MUST result in an error:

- An orphan member name colliding with an explicitly defined top-level section name.
- The implicit `base` section colliding with an explicitly defined top-level section named `base` .

The implementation SHOULD document clearly which orphan-member strategy it uses.

13.1.2. Top-Level Section Mounting

Explicitly defined top-level sections are mounted directly onto the parsed result. Their hierarchy MUST still be respected: descending into deeper nesting may not skip intermediate levels.

This applies:

- A level-3 section MUST follow a level-2 section.
- Skipping levels when descending (for example, level 1 directly to level 3) is invalid.

13.1.3. Strict Mode

In strict mode, there MUST be exactly one explicit top-level section.

Any additional sections MUST appear only as subsections nested within that section. Top-level orphan members are not allowed in strict mode. Otherwise, an error MUST be reported.

13.2. Line Handling and Whitespace

- Newline normalization is required:
  - Support all three forms: LF (`0x0A`), CRLF (`0x0D 0x0A`), and CR (`0x0D`).
- Leading/trailing whitespace (tabs or spaces):
  - Trim from section headers and keys.
- 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
  ```

  it MUST be interpreted as having a value of `null`.
- Key uniqueness:
  - Keys MUST be unique within the same section and nesting level.
  - In **lenient mode**, later duplicate keys MUST be ignored, and the implementation MUST report the duplicate as a warning diagnostic.
  - In **strict mode**, any duplicate key MUST result in an error.
  - In neither mode may an implementation silently overwrite an earlier key with a later one.

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`
- Boolean literals MUST NOT be written as `1` or `0`. Host applications MAY perform their own conversions after parsing, but such conversions are outside the YINI data model and this specification.

13.5. Objects

Inline object members use `:` as the canonical separator:

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

In lenient mode only, parsers MAY accept `=` as an alternative object member separator:

```
obj = { a = 1, b = 2 }
```

If accepted, implementations SHOULD treat this as equivalent to the canonical `:` form internally. Formatters SHOULD normalize object members to `:`.

Mixing `:` and `=` within the same inline object is discouraged in lenient mode because it reduces visual consistency and can make the object harder to read:

```
obj = { a: 1, b = 2 }  // Discouraged in lenient mode; invalid in strict mode.
```

In strict mode, `=` MUST NOT be accepted inside inline objects.

Any empty slot inside `{ ... }`, for example:

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

is an error in strict mode and SHOULD be reported as at least a warning or error in lenient mode.

13.6. Lists

The syntax for a list is: **(a) Bracketed form (preferred):**

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

- A bracketed list line MUST not begin with a comma.
- In lenient mode, a trailing comma in `[ ]` is ignored.
- In strict mode, a trailing comma in `[ ]` MUST result in an error.

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. String Concatenation

Implementations SHOULD treat string concatenation as a value-level expression formed by two or more operands separated by the plus sign (`+`).

The `+` token is not a general arithmetic operator in YINI. Implementations MUST NOT evaluate numeric operands using numeric addition. When a `+` expression is accepted as concatenation, all operands are evaluated as concatenation operands and the final result is a string.

A line break MAY occur after the `+` operator, allowing long string values to be split across multiple source lines. A line break MUST NOT occur before the `+` operator.

Lists and inline objects MUST NOT be used as concatenation operands.

In strict mode, validation is simple: every operand in a concatenation expression MUST be a string literal. If any operand is a number, boolean, null, list, or inline object, the implementation MUST reject the expression.

In lenient mode, a concatenation expression MUST begin with a string literal. Additional operands MAY be string literals, number literals, boolean literals, or null literals. The whole `+` expression

MUST be treated as string concatenation.

```
value = 1 + 2 + "3"  // Invalid!
```

This expression is evaluated as neither `"123"` nor `"33"`. It MUST result in an error because the first operand is not a string literal.

The correct way to do this:

```
value1 = "1" + 2 + 3     // Valid, result: "123"
value2 = "1" + 2 + "3"   // Valid, result: "123"
value3 = "" + 1 + 2 + 3  // Valid, result: "123"
```

The result of concatenation is always a single string value.

It MUST NOT be evaluated as numeric addition followed by string conversion.

If a lenient-mode `+` expression does not begin with a string literal, the expression MUST be rejected. For example:

```
invalid1 = 100 + 99      // Invalid: YINI does not define numeric addition.
invalid2 = 1 + 2 + "3"   // Invalid: concatenation must begin with a string literal.
```

The first expression is invalid because YINI does not define numeric addition. The second expression is invalid because a concatenation expression must begin with a string literal.

Implementations SHOULD report invalid concatenation with diagnostics that identify the reason, such as:

- Non-string operand in strict mode.
- Concatenation expression does not begin with a string literal.
- List or inline object used as a concatenation operand.
- Line break before the + operator.

13.8. String Literal Types

Note: If text is not quoted, it MUST NOT be interpreted as a string literal.

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

Type	Prefix	Example	Behavior
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

13.9. Comments

Implementations MUST recognize comments according to Section 3.3, "Comments".

In summary:

- `//` begins a line comment outside string literals and backticked identifiers.
- `#` begins a line comment outside string literals and backticked identifiers.
- `/* ... */` begins a block comment outside string literals and backticked identifiers.
- `;` begins a full-line comment only when it is the first non-whitespace character on a line.
- Nested block comments are not supported.
- Comments may appear only where whitespace is allowed and MUST NOT split tokens.

Implementations SHOULD remove or ignore comments during parsing before semantic interpretation, while preserving source-position information when useful for diagnostics.

13.10. Additional Implementation Guidance

Implementations are encouraged to provide the following features to improve parser robustness, diagnostics, and developer experience:

- Support both **lenient** and **strict** parsing modes.
- Attach **position metadata**, such as line and column numbers, to tokens, parsed values, and diagnostics.
- Normalize scalar values to platform-agnostic canonical representations before exposing parsed output. For example:
    - **Booleans** SHOULD be represented as the logical values `true` and `false`.
    - **Null** SHOULD be represented as the logical null value.

    The concrete host-language representation may differ by implementation, but the exposed semantic value SHOULD remain equivalent.

    For example, the same YINI logical values may be represented by different host-language values:

YINI logical value	JavaScript / TypeScript	Python
Boolean true	`true`	`True`
Boolean false	`false`	`False`
Null	`null`	`None`

- Support configurable **diagnostic severity thresholds** for parser recovery and abort behavior.

  Recommended severity thresholds:

  - **Level 0 — Recover when possible:** Report errors and warnings, but attempt to continue parsing where recovery is possible.
  - **Level 1 — Abort on errors:** Continue after warnings, but abort when an error is encountered.
  - **Level 2 — Abort on warnings and errors:** Abort when either a warning or an error is encountered.

When recovering from invalid syntax, implementations MAY use internal placeholder names or recovery nodes. Such recovered constructs MUST be reported through diagnostics and MUST NOT be silently treated as valid source constructs.

14. Compatibility and Versioning

This section covers YINI's compatibility and interoperability principles.

14.1. Fallback Rules

14.1.1. Invalid Sections or Keys

- In lenient mode, implementations MAY retain invalid key names or section headers as-is where possible, or MAY remap illegal characters, provided a warning diagnostic is reported. In strict mode, invalid key names or section headers MUST result in an error.

14.1.2. Graceful Degradation

In lenient mode, implementations MAY report warnings for unsupported reserved constructs when recovery is possible. In strict mode, unsupported or unrecognized syntax MUST result in an error unless explicitly allowed by this specification.

14.2. Versioning Strategy

The YINI Specification uses Semantic Versioning-style version numbers to signal format evolution.

Version identifiers use the form:

```
MAJOR.MINOR.PATCH[-STAGE.N]
```

Examples:

```
1.0.0-RC.6
1.0.0
1.1.0
```

- **MAJOR:** Incompatible changes.
- **MINOR:** Backward-compatible additions.
- **PATCH:** Backward-compatible clarifications, corrections, and non-breaking fixes.
- **STAGE:** Optional pre-release stage, such as Beta or RC.

Before the final `1.0.0` release, staged labels such as Beta and RC indicate that the format is intended to be close to stable, but may still receive clarifications or breaking adjustments before final release.

14.3. Encoding Notes

14.3.1 Required Encoding

YINI documents MUST be encoded in UTF-8.

- The canonical and recommended encoding is UTF-8 without BOM.
- Use of other encodings (such as UTF-16 or Latin-1) is not supported by this specification and is not guaranteed to be portable.

14.3.2 Byte Order Mark (BOM)

- A UTF-8 BOM (`0xEF 0xBB 0xBF`) SHOULD NOT be used.
- If present, for compatibility implementations MAY accept and ignore an initial UTF-8 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 as a shebang line when it appears as the first line of the document.

14.4. JSON Compatibility

A well-formed YINI document can be converted to JSON while preserving its parsed data structure, value types, and identifier names. Many JSON objects can also be represented as YINI documents, provided they are mapped according to YINI's section, member, identifier, and value rules.

Although YINI and JSON are distinct formats with different goals, their core data models are closely aligned.

Comments, disabled lines, the `@yini` marker, and the document terminator are metadata or syntax-only constructs. They are not represented as JSON values.

Not every JSON document maps directly or cleanly to YINI:

- A top-level JSON array cannot directly become a complete YINI document unless it is wrapped in a member or section.
- JSON object keys may be any string, including empty strings or strings containing characters that require backticked identifiers in YINI.
- Some JSON structures may require a chosen mapping strategy when converted to YINI, especially when deciding whether nested JSON objects should become inline objects or nested sections.

Therefore, JSON-to-YINI conversion can preserve the same data model when a suitable mapping strategy is used, but it is not always a purely mechanical one-to-one syntax conversion.

Converting YINI to JSON

- **Keys / Identifiers:** YINI identifiers become JSON object member names. When serialized as JSON, all keys are written as JSON strings.
- **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	✅ Root/section: `key = "value"` Object: `key: "value"` canonical	✅ Keys always quoted, `:` for objects	YINI uses `=` for root/section members and canonical `:` for object members. Lenient mode may accept `=` inside inline objects, but formatters should normalize to `:`.
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	✅ Supported in YINI (`/END`)	🚫 Not supported in JSON	YINI's document terminator has no JSON equivalent and is ignored when converting to JSON.

See also:

See Sections 15.3 and 15.4 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

This is a lenient-mode example. It has multiple top-level sections and no `/END`.

```
@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

This is a lenient-mode example. It has multiple top-level sections and no `/END`.

```
@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' }
            ]
        }
```

15.7. Large-Scale Real-World Configuration Example C: Industrial Monitoring & Automation Platform

This example is valid when parsed in strict mode.

```
@yini strict

// Example C: Industrial Monitoring & Automation Platform.
/*
  Covers:
  - Strict-mode-compatible single top-level section.
  - Deep section nesting.
  - Realistic industrial / factory domain modeling.
  - Inline objects and nested inline objects.
  - Arrays of scalars and arrays of objects.
  - Strings, numbers, booleans, null.
  - Scheduling, alerts, telemetry, maintenance, and safety rules.
  - File paths, endpoints, and policy-style configuration.
*/

^ PlantOps
systemName = 'Orion Manufacturing Grid'
description = 'Central control and monitoring platform for production lines, machine telemet
siteCode = 'SE-GOT-PLANT-07'
environment = 'production'
debug = false
timezone = 'Europe/Stockholm'
contacts = ['ops@orion-industries.io', 'maintenance@orion-industries.io', 'safety@orion-indu

    ^^ Identity
    tenant = 'orion-industries'
    region = 'eu-north-1'
    cluster = 'plantops-main'
    build = {
        version: '3.2.1',
        releaseChannel: 'stable',
        commit: 'f7d23aa',
        signed: true
    }

    ^^ Telemetry
    enabled = true
    ingestionMode = 'streaming'
    retentionDays = 120
    sampleIntervalsSeconds = [1, 5, 15, 60]
    normalizeUnits = true
    deadband = {
        temperature: 0.2,
        pressure: 0.05,
        vibration: 0.01,
        power: 0.5
    }
```

```
    ^^^ Buffers
    memoryQueueSize = 50000
    diskSpool = {
        enabled: true,
        path: '/srv/plantops/spool',
        maxSizeGB: 120,
        compression: 'zstd'
    }
    flushPolicy = {
        maxBatchSize: 1000,
        maxWaitMs: 750,
        retry: {
            maxRetries: 12,
            backoffMs: [100, 250, 500, 1000, 2000]
        }
    }

    ^^^ Sensors
    sources = [
        { id: 'temp-line-a', type: 'temperature', unit: 'C', enabled: true },
        { id: 'press-line-a', type: 'pressure', unit: 'bar', enabled: true },
        { id: 'vib-motor-12', type: 'vibration', unit: 'mm/s', enabled: true },
        { id: 'power-feed-1', type: 'power', unit: 'kW', enabled: true },
        { id: 'humidity-zone-3', type: 'humidity', unit: '%', enabled: false }
    ]
    qualityRules = {
        rejectNegative: ['pressure', 'power', 'humidity'],
        clamp: [
            { metric: 'temperature', min: -40, max: 180 },
            { metric: 'pressure', min: 0, max: 25 },
            { metric: 'humidity', min: 0, max: 100 }
        ]
    }

^^ Production
shifts = [
    { name: 'morning', start: '06:00', end: '14:00' },
    { name: 'evening', start: '14:00', end: '22:00' },
    { name: 'night', start: '22:00', end: '06:00' }
]
lines = [
    { code: 'LINE-A', enabled: true, cells: 8 },
    { code: 'LINE-B', enabled: true, cells: 6 },
    { code: 'LINE-C', enabled: false, cells: 4 }
]
scheduling = {
    autoDispatch: true,
    planningWindowHours: 48,
    constraints: {
        maxConcurrentChangeovers: 2,
        requireQualifiedOperator: true,
        pauseDuringMaintenance: true
```

```
    }
}

    ^^^ Recipes
    defaultRecipe = 'steel-frame-v4'
    available = [
        { id: 'steel-frame-v4', revision: 12, active: true },
        { id: 'aluminium-housing-v2', revision: 7, active: true },
        { id: 'test-batch-calibration', revision: 3, active: false }
    ]
    validation = {
        requireSignedRecipe: true,
        checksumAlgorithm: 'sha256',
        allowDowngrade: false
    }

    ^^^ Traceability
    enabled = true
    batchIdFormat = 'ORD-{date}-{line}-{seq}'
    retainBatchHistoryDays = 3650
    tags = {
        useQr: true,
        useRfid: true,
        fallbackManualEntry: false
    }

^^ Maintenance
enabled = true
workOrderSystem = 'cmms'
defaultPriority = 'normal'
planners = ['planner-a', 'planner-b']
rules = {
    createOnThresholdBreach: true,
    requireApprovalForShutdown: true,
    autoAssignByArea: true
}

    ^^^ Preventive
    windows = [
        { assetGroup: 'motors', intervalHours: 500, durationMinutes: 90 },
        { assetGroup: 'compressors', intervalHours: 1000, durationMinutes: 180 },
        { assetGroup: 'conveyors', intervalHours: 750, durationMinutes: 120 }
    ]
    reminders = {
        notifyBeforeHours: [72, 24, 4],
        channels: ['email', 'dashboard']
    }

    ^^^ Predictive
    enabled = true
    models = [
        { name: 'bearing-wear', minConfidence: 0.86, action: 'inspect' },
        { name: 'seal-leakage', minConfidence: 0.81, action: 'schedule-service' },
```

```
        { name: 'motor-overload', minConfidence: 0.90, action: 'slowdown-line' }
    ]
    suppression = {
        cooldownMinutes: 180,
        ignoreDuringManualService: true
    }

^^ Alerts
enabled = true
defaultSeverity = 'warning'
channels = ['dashboard', 'email', 'sms']
deduplicationWindowSeconds = 600
templates = {
    subjectPrefix: '[PlantOps]',
    includeAssetContext: true,
    includeRunbookLink: true
}

    ^^^ Routing
    rules = [
        { event: 'temperature-high', severity: 'warning', target: 'ops-team' },
        { event: 'pressure-drop', severity: 'critical', target: 'maintenance-team' },
        { event: 'guard-door-open', severity: 'critical', target: 'safety-team' },
        { event: 'network-loss', severity: 'critical', target: 'infra-team' }
    ]
    escalation = {
        enabled: true,
        afterMinutes: [5, 15, 30],
        steps: [
            { level: 1, notify: ['shift-lead'] },
            { level: 2, notify: ['plant-manager'] },
            { level: 3, notify: ['regional-director'] }
        ]
    }

    ^^^ QuietHours
    enabled = true
    start = '23:00'
    end = '05:30'
    suppressSeverities = ['info']
    exceptions = ['critical', 'safety']

^^ Safety
enabled = true
lockoutTagoutRequired = true
incidentWebhook = 'https://safety.orion-industries.io/hooks/incidents'
restrictedZones = ['robot-cell-4', 'robot-cell-5', 'laser-area-2']

    ^^^ AccessControl
    badgeRequired = true
    biometricRequired = false
    visitorsAllowed = false
    roles = [
```

```
        { role: 'operator', canAcknowledge: true, canOverride: false },
        { role: 'supervisor', canAcknowledge: true, canOverride: true },
        { role: 'auditor', canAcknowledge: false, canOverride: false }
    ]

    ^^^ Interlocks
    active = true
    checks = [
        { name: 'guard-door', failMode: 'stop-line' },
        { name: 'e-stop-circuit', failMode: 'shutdown-cell' },
        { name: 'light-curtain', failMode: 'halt-motion' }
    ]
    overridePolicy = {
        allowed: false,
        emergencyContact: null
    }

^^ Integrations
erp = {
    enabled: true,
    endpoint: 'https://erp.orion-industries.io/api',
    syncIntervalMinutes: 10
}
mes = {
    enabled: true,
    endpoint: 'https://mes.orion-industries.io/api',
    timeoutMs: 5000
}
historian = {
    enabled: true,
    endpoint: 'opc.tcp://historian.orion.internal:4840',
    namespace: 'urn:orion:plantops'
}

^^ Logging
level = 'info'
format = 'json'
outputs = ['stdout', 'file']
file = {
    path: '/srv/plantops/log/plantops.log',
    rotateDaily: true,
    keepDays: 21
}

^^ Security
requireTls = true
minimumTlsVersion = '1.3'
apiKeys = [
    { name: 'telemetry-writer', active: true, scope: 'ingest' },
    { name: 'maintenance-sync', active: true, scope: 'workorders' },
    { name: 'legacy-debug-client', active: false, scope: 'readonly' }
]
audit = {
```

```
        enabled: true,
        retainDays: 730,
        recordConfigChanges: true,
        recordOperatorActions: true
    }

/END
```

16. Appendices and Reserved Areas

16.1. License

Apache License, Version 2.0, January 2004, http://www.apache.org/licenses/ Copyright 2024-2026 Gothenburg, Marko K. Seppänen, (Sweden via Finland).

16.2. Acknowledgments

YINI would not exist in its current form without the many insights, challenges, and constructive feedback contributed by 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 6, 2026-05-30

- **Changed:** The `#` character now always begins a comment outside string literals. No whitespace is required before or after `#`.
- **Changed:** Increased the maximum repeated section marker depth from 6 to 9 because section marker separators (`_`) may now be used to make headers easier to read. Repeated marker headers may now express levels 1–9 directly, while numeric shorthand remains required for levels 10 and deeper.
- **Added:** Added the explicit hexadecimal notation `hex:` as an alternative to `0x...`. The `hex:` prefix is case-insensitive and MUST be followed immediately by hexadecimal digits or an allowed digit separator.
- **Removed:** Support for `#` as a hexadecimal number prefix was removed. Hexadecimal numbers MUST instead be written using `0x...` or the explicit `hex:` form.
- **Removed:** Hyper Strings (H-Strings) were removed. While useful for readable long-form text, they served a narrow use case and overlapped with existing string forms. Their removal keeps the core language smaller, clearer, and more predictable.
- **Added:** In lenient mode, inline object members MAY use `=` as an alternative to `:`. The canonical form remains `key: "value"`.
- **Clarified:** In strict mode, inline object members MUST use `:`. Using `=` inside inline objects is invalid, whether mixed with `:` or used consistently.
- **Clarified:** Tools and formatters SHOULD normalize inline object members to `:`.
- **Changed:** Updated string concatenation rules. In both modes, a concatenation expression MUST begin with a string literal. In strict mode, all concatenation operands MUST be string literals. In lenient mode, additional operands MAY be string literals, number literals, boolean literals, or null literals. Numeric addition is not defined. Non-string scalar operands are converted to their parsed canonical string representation before concatenation. A concatenation expression MAY span multiple source lines only when the line break occurs after the `+` operator; a line break before `+` is invalid. Lists and inline objects MUST NOT be used as concatenation operands.
- **Clarified:** Missing values vs. trailing commas:
  - If the value is the keyword `null` (case-insensitive), or if a root-level or section-level member has no value after `=` in lenient mode, it is treated as **Null**.
  - Missing values inside lists or objects are not treated as `Null`. A trailing comma inside a list or object is permitted only in lenient mode and is ignored; in strict mode it is an error.
- **Added:** For readability, an underscore character `_` may appear after a base prefix or between successive digits.
- **Added:** For readability, added support for section marker separators `_`.

- **Added:** Added optional mode declarations to the YINI marker using `@yini strict` and `@yini lenient`. These declarations state the document's expected parser mode but MUST NOT automatically switch the active parser mode. If `@yini strict` is parsed in lenient mode, the parser MUST emit a mode-mismatch error. If `@yini lenient` is parsed in strict mode, the parser MUST emit a mode-mismatch warning.
- **Clarified:** Defined empty-document handling by mode. In lenient mode, a document containing only whitespace, comments, and/or disabled lines is permitted but SHOULD produce a warning. In strict mode, such a document is invalid and MUST result in an error.
- **Changed:** Re-added `>` as a supported ASCII section marker based on feedback. It is now documented as a quote-like ASCII fallback marker, with a portability caveat because some email clients, forum renderers, and Markdown-like environments may treat it as a quote prefix.

v1.0.0 RC 5, 2026-04-09

- **Changed:** The document terminator (`/END`) is now required in strict mode and remains optional in lenient mode.
- **Clarified:** Updated the specification text, validation rules, and strict/lenient mode table to reflect that strict mode requires `/END` at the end of the document.
- **Clarified:** Clarified whitespace rules for section headers: repeated/basic section headers do not require a space before the section name, while numeric shorthand headers (such as `^7`) do.
- **Clarified:** Defined top-level structure rules for lenient and strict mode. In lenient mode, orphan members may be exposed at the root or under an implicit base section; in strict mode, exactly one explicit top-level section is allowed, all additional sections MUST be nested beneath it, and top-level orphan members are forbidden.
- **Updated:** Added a third large real-world configuration example (C) for parsing in strict mode.
  - See Section **15.7**.
  - The full YINI and JSON versions of these examples are also included under Large-Scale Real-World Configuration Examples.

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.

16.5. Reserved: Grammar (Formal)

Reserved for future inclusion of the formal grammar.

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 such as YAML, JSON, or classic INI. This table clarifies syntax edge cases and helps avoid subtle bugs.

Trailing commas in list and objects

Trailing commas after values or members inside lists or objects never produce `Null` values. Strict mode disallows a comma with no element after it altogether.

✅ YINI Syntax Cheatsheet – Common Confusions

Element	Correct Syntax	Common Mistake	Clarification
Root/section member / List	`name = "John" / items = ["a", "b", "c"]`	`name: "John" / items:`	`:` is not valid assignment syntax for root-level or section-level members; use `=` for ordinary members 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`, `#Comment`, or `// Comment`	Assuming `#Comment` is not a comment	Outside string literals, `#` always begins a comment. No whitespace is required.
Hex values	`color = 0xFF0033` or	`color = #FF0033`	`#` is not a hex prefix in YINI. It begins a comment outside string

Element	Correct Syntax	Common Mistake	Clarification
	`color = hex:FF0033`		literals.
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.
Inline object member separator	`obj = { a: 1, b: 2 }`	`obj = { a = 1, b = 2 }`	`:` is the canonical separator inside inline objects. In lenient mode, `=` MAY be accepted, but formatters should normalize to `:`. In strict mode, `=` inside inline objects is invalid.
Mixed object separators	`obj = { a: 1, b: 2 }`	`obj = { a: 1, b = 2 }`	Mixing `:` and `=` inside the same inline object is discouraged in lenient mode and invalid in strict mode.
Quote characters	`text = "hello"` or `text = 'hello'`	Using curly quotes as delimiters (start/end characters): `text = “hello”`	String delimiters MUST be plain ASCII quotes: `'` U+0027 or `"` U+0022. Curly quotes may appear inside strings as ordinary text, but they do not delimit strings.
String concatenation start	`text = "value: " + 123`	`text = 123 + " value"`	A concatenation expression MUST begin with a string literal. In lenient mode, later scalar operands may be numbers, booleans, or null.
Multi-line concatenation	`text = "hello " +   "world"`	Placing `+` at the start of the next line	A line break is allowed after `+`, but not before `+`.

16.7. 📃 Unicode Whitespace Characters

Informational / reserved for future

Below is a categorized list of Unicode whitespace characters recognized as within <Unicode-WS>.

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
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

Code Point	Character Name	Abbreviation	Unicode Category	Notes
`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 clear, structured, and human-friendly configuration format.

yini-lang.org · YINI-lang on GitHub