

roc-parser

A manual for parsing text and data in Roc

THE ROC-PARSER CONTRIBUTORS

Version 2.0.0: Parser combinators, with CSV, YAML, XML, Markdown and HTTP/1.1 parsers

Table of Contents

START HERE	2
1. What roc-parser is	3
1.1. The problem it solves	3
1.2. What a first success looks like	4
1.3. Supported formats	4
1.4. How the pieces fit together	5
1.5. When to choose something else	6
1.6. Stability	6
2. Getting started	7
2.1. Before you start	7
2.2. Add the package	7
2.3. Read CSV into records	8
2.4. What the program does	9
2.5. Next steps	10
3. Choosing how to read your input	11
3.1. Decode into your own type	11
3.2. Read a document tree	12
3.3. Build a parser with combinators	13
3.4. Use no parser at all	14
3.5. Decision table	15
3.6. Decision flow	16
4. Your first parser	19
4.1. Step 1: read a key	19
4.2. Step 2: read a whole setting	20
4.3. Step 3: choose between kinds of value	21
4.4. Step 4: read every line, and read the errors	22
4.5. What you have learned	23
GUIDES	24
5. Read CSV data	25
5.1. Decode rows into records	25
5.2. Files without a header	28
5.3. Read every field as raw bytes	28
5.4. Hand-built record parsers	29
5.5. Report which row failed	30

5.6. Which CSV files are accepted	31
5.7. Limitations	31
5.8. Performance	32
6. Read YAML configuration and frontmatter	33
6.1. Decode a document into a record	33
6.2. Explore a document as a tree	35
6.3. Read Markdown frontmatter	36
6.4. How plain values are typed	37
6.5. Write multi-line text	38
6.6. Write lists	38
6.7. Quote strings	39
6.8. What is rejected	39
6.9. Show where an error is	40
6.10. Performance	41
7. Read XML documents	42
7.1. Parse a document and find elements	42
7.2. What the tree contains	43
7.3. Report malformed input	44
7.4. What is not supported	45
7.5. Performance	45
8. Markdown	46
8.1. Parse a document into blocks	46
8.2. Parse inline content	49
8.3. Read frontmatter	51
8.4. Walk the tree to produce output	52
8.5. Conformance and limitations	55
8.6. Performance	56
9. HTTP messages	57
9.1. Parse a request	57
9.2. Parse a response	59
9.3. Understand how the body is found	59
9.4. Read pipelined messages	60
9.5. Use the parsers inside a larger grammar	61
9.6. Rejected messages and request smuggling	61
9.7. Out of scope	63
9.8. Performance	63
10. Combinators by task	64
10.1. Running a parser	64
10.2. Reading text	64
10.3. Sequencing and transforming	65
10.4. Alternatives and optional parts	66
10.5. Repetition	68

10.6. Recursive structures	69
10.7. Bytes, not characters	69
10.8. Parsers over other input	70
11. Parse a format of your own	71
11.1. 1. Write down the result types	71
11.2. 2. Write one parser per part	71
11.3. 3. Combine the parts	72
11.4. 4. Test each parser	73
11.5. 5. Split the input into records yourself	73
11.6. 6. Use it	74
11.7. When the format is not line-based	75
12. Report parse errors	76
12.1. Summary	76
12.2. YAML and XML: show the location	77
12.3. CSV: report the record and field	78
12.4. HTTP: reject the message	78
12.5. Your own parsers: say where and what	79
REFERENCE	80
13. Modules	81
13.1. The modules at a glance	81
13.2. How the modules fit together	82
13.3. Parser	83
13.4. Utf8	84
13.5. CSV	84
13.6. HTTP	85
13.7. Markdown	85
13.8. Xml	85
13.9. Yaml	86
13.10. Choosing between modules	86
14. Conformance	88
14.1. How conformance is checked	88
14.2. Results	88
14.3. Reproducing the results	89
14.4. CSV	90
14.5. HTTP	90
14.6. Markdown	91
14.7. XML	92
14.8. YAML	92
14.9. Property testing	93
15. Compatibility	94
15.1. Roc compiler	94
15.2. Version numbers	94

15.3. Platforms	95
15.4. Dependencies	95
15.5. Changes in 2.0.0 at a glance	95
16. Performance	97
16.1. Summary	97
16.2. When roc-parser is fast enough and when it isn't	98
16.3. Complexity guarantees	99
16.4. Memory behaviour	100
16.5. What the comparison does and does not show	100
16.6. Why roc-parser is slower where it is	101
16.7. How roc-parser uses SIMD and slices	102
16.8. Tips for your own parsers	103
17. Glossary	105
CONTRIBUTING	109
18. Contributing to roc-parser	110
18.1. Report a problem or propose a change	110
18.2. Development setup	110
18.3. Repository map	111
18.4. Run the tests	111
18.5. Build the documentation	112
18.6. Before you open a pull request	113
18.7. License	114
19. Property tests and conformance reviews	115
19.1. Why property tests	115
19.2. Three kinds of target	115
19.3. Oracles	116
19.4. Run the property tests	117
19.5. Reproduce and minimise a failure	118
19.6. Add a regression	118
19.7. Fix the cause, do not gate it	119
19.8. Run a conformance review	119
19.9. Known-failures baselines	120
19.10. OpenSSF Scorecard	120
20. Add a format module	121
20.1. 1. Decide what you support	121
20.2. 2. Create the module	121
20.3. 3. Write the tests	122
20.4. 4. Add a probe and a review script	122
20.5. 5. Add property-test targets	123
20.6. 6. Register the targets	123
20.7. 7. Document the module	124
20.8. Checklist	124

21. Benchmarks	125
21.1. Run the benchmarks	125
21.2. How a measurement works	126
21.3. Corpora	127
21.4. Roc drivers	127
21.5. Comparison programs	128
21.6. Add a benchmark	129
21.7. Read a report	129
21.8. Continuous integration	130
21.9. Reproducible environment with roc-blueprint	130
21.10. Profile one workload	131
22. Compiler updates, releases and security reports	133
22.1. Roc nightly updates	133
22.2. Release a new version	134
22.3. Security reports	135
BACKGROUND	137
23. Where the design comes from	138
23.1. A parser is a function	138
23.2. Sequencing: monads and applicatives	139
23.3. Choice and backtracking	140
23.4. Repetition that always ends	141
23.5. Error reporting	141
23.6. Bytes, not characters	142
23.7. Recursion and laziness	142
23.8. Packrat parsing and performance	143
23.9. Combinators and parser generators	143
23.10. Testing against the specification	143
23.11. How roc-parser compares	144
23.12. Further reading	145
RELEASE NOTES	147
24. Release notes	148

Parse text and data in Roc. `roc-parser` gives you small parsers that combine into larger ones, and ready-made parsers for CSV, YAML, XML, Markdown and HTTP/1.1 built from the same pieces.

Choose a path

New to `roc-parser`

Read [What `roc-parser` is](#), then [Getting started](#), then [Your first parser](#).

Parsing a format `roc-parser` already supports

Go to its guide: [Read CSV data](#), [Read YAML configuration and frontmatter](#), [Read XML documents](#), [Markdown](#) or [HTTP messages](#).

Writing your own parser

Read [Combinators by task](#), [Parse a format of your own](#) and [Report parse errors](#).

Looking something up

[Modules](#), [Conformance](#), [Performance](#), [Compatibility](#) and [Glossary](#).

Not sure whether you need a parser

Read [Choosing how to read your input](#).

Contributing to `roc-parser`

Start with [Contributing to `roc-parser`](#).

Curious where the design comes from

Read [Where the design comes from](#).

Start here

Chapter 1. What roc-parser is

roc-parser turns text into typed `Roc` values. You give it a string, such as the contents of a CSV file, a YAML configuration, or an HTTP request, and it gives back either a Roc value you can use directly or an error that says what was wrong with the input.

It is written in pure Roc, so it works with any platform, and it comes in two layers:

- Ready-made parsers for common formats. You call one function and get a document tree, or, for CSV and YAML, your own record type: the compiler infers the type from your annotation and decodes straight into it.
- The small building blocks those parsers are made from, called *parser combinators*. You use them to read a format of your own, from a one-line `key=value` setting to a complete file format.

This manual is for Roc programmers. You do not need to know anything about parsing theory.

1.1. The problem it solves

Most programs start by reading text that someone else wrote. Splitting a line on commas, or matching it with a regular expression, works until the input contains a quoted comma, a missing field, or a value of the wrong kind. Then the code either crashes, silently produces wrong data, or grows a pile of special cases.

A parser describes the shape of valid input once, as a Roc value, and checks every input against it. With roc-parser:

- The result is typed. A CSV row becomes `{ name : Str, moons : U64 }`, not a list of strings you still have to convert.
- Invalid input is an `Err` value, never a crash. The ready-made parsers follow their published specifications, including the awkward corners, and report where the problem is.
- A parser is built from smaller parsers, so each piece can be tested on its own with `expect`.

Compared with a regular expression, a combinator parser is longer to write for a one-off match, but it can describe nested structure (a regular expression cannot match balanced brackets), it produces a value rather than a list of captured substrings, and it reads as ordinary Roc code.

1.2. What a first success looks like

This program decodes a CSV file with a header row into typed records and prints them. [Getting started](#) walks through it.

```
import cli.Stdout
import parser.CSV

Planet : { name : Str, moons : U64 }

input =
  \\name,moons
  \\Mercury,0
  \\Earth,1
  \\Mars,2

planets : Str -> Try(List(Planet), [InvalidCsv(CSV.Error), MissingRequiredField(Str)])
planets = |text| CSV.parse(text)

describe : Str -> Str
describe = |text| {
  match planets(text) {
    Ok(rows) => Str.join_with(rows.map(|p| "${p.name}: ${p.moons.to_str()} ${if p.moons
== 1 "moon" else "moons"}"), "\n")
    Err(InvalidCsv(error)) => "line ${error.line.to_str():} ${error.message}"
    Err(MissingRequiredField(name)) => "no ${name} column"
  }
}

main! = |_args| {
  Stdout.line!(describe(input))
}
```

Output:

```
Mercury: 0 moons
Earth: 1 moon
Mars: 2 moons
```

1.3. Supported formats

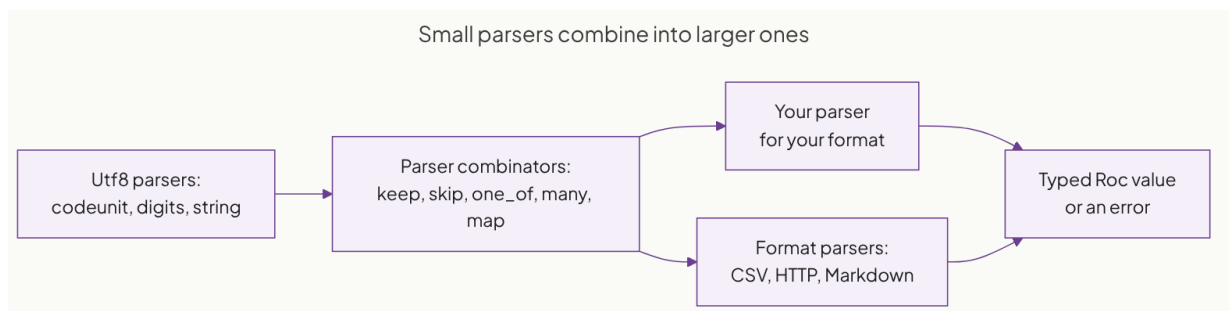
Each module follows a published specification. Where a module implements only part of one, its chapter says which part, and input outside that part is rejected with an error rather than misread.

Module	Reads	Specification
CSV	Comma-separated rows decoded into your own record type	RFC 4180 , with the usual relaxations (any line ending, rows of different lengths)
Yaml	Configuration files and Markdown frontmatter, as a tree or your own record type	A documented subset of YAML 1.2 : one document, no anchors, aliases or tags
Xml	A whole XML document as a tree	XML 1.0 (Fifth Edition) well-formedness, without document type declarations or namespaces
Markdown	A Markdown document as a syntax tree	CommonMark 0.31.2 with GitHub Flavored Markdown tables, task lists and strikethrough
HTTP	One HTTP/1.x request or response	RFC 9112 message syntax and RFC 9110 field syntax
Parser, Utf8	Anything you describe with combinators	Not applicable

The format chapters describe each module’s exact behaviour. The [API reference](#) lists every public function.

1.4. How the pieces fit together

A combinator is a function that takes parsers and returns a new parser. The `Utf8` module provides parsers for the smallest pieces of text, such as one byte, a digit, or an exact word. The `Parser` module combines them: one after another, one of several alternatives, repeated, or with the result transformed. Each format module also exposes its parser as a `Parser` value, so your own parsers and the library’s can be mixed.



Your first parser teaches the combinators by building a small parser one step at a time.

1.5. When to choose something else

roc-parser fits programs that read text formats of moderate size completely into memory and want typed results with clear errors. Consider something else when:

- You need full YAML, with anchors, aliases, tags or several documents in one stream, or XML with document type declarations, namespaces or schema validation. These modules reject such input on purpose.
- You need to parse a stream incrementally as it arrives, such as a very large log file or a network connection. Every parser here works on a complete input held in memory.
- You want to render Markdown to HTML or serialize values back to text. The modules read formats; they do not write them.
- You need JSON, TOML, or another format this package does not include. You can write it with the combinators, but a dedicated package may already exist.

1.6. Stability

roc-parser tracks recent nightly builds of the Roc compiler. Because Roc is still changing, the package's API may change between releases. Each release names the compiler build it was tested with, and the release notes describe any changes you need to make when upgrading.

Chapter 2. Getting started

In this chapter you add `roc-parser` to a Roc app and use it to read three lines of CSV into typed records. At the end the app prints one line per record. You need a terminal, an internet connection, and a little Roc: how to write a function, a record, and a `match`.

2.1. Before you start

You need:

- The Roc compiler, as a recent *nightly* build. Roc is still changing, so each `roc-parser` release is tested with one specific nightly, which its release notes name. Download that build from the [Roc nightlies page](#) and put the `roc` executable on your `PATH`.
- A platform for your app. The examples in this manual use `basic-cli` 0.23.0, which provides `main!` and printing to the terminal. `roc-parser` itself is pure Roc and works with any platform.

Check the compiler:

```
roc version
```

It prints the name of the nightly build, which should match the one in the release notes:

```
Roc compiler version nightly-<date>-<commit>
```

2.2. Add the package

A Roc app names each package it uses by URL in its header. The compiler downloads the package the first time it builds the app and checks it against the hash in the URL.

1. Open the [roc-parser releases page](#) and choose the newest release.
2. Under **Assets**, find the file ending in `.tar.zst`. Copy its link: it has the form `https://github.com/lukewilliamboswell/roc-parser/releases/download/<version>/<hash>.tar.zst`.
3. Put that link in your app's header under a short name. This manual uses `parser`.

Create a file named `main.roc` with this header, replacing the `parser` URL with the one you copied:

```
app [main!] {
  cli: platform "https://github.com/roc-lang/basic-
cli/releases/download/0.23.0/GNN5tt2gKdX4dhawg4915C4YB193woHFdcCkz31fhGxv.tar.zst",
  parser: "https://github.com/lukewilliamsoswell/roc-
parser/releases/download/<version>/<hash>.tar.zst",
}
```

The examples in this manual are tested against the package source in the repository, so their headers say `parser: "../../package/main.roc"` instead. Everything else in them is the same.

For complete programs to start from, download `roc-parser-examples-<version>.zip` from the same release. Its apps are already pinned to that release; unzip it and follow the `README.md` inside, which runs them with `roc <name>.roc` from its `examples/` directory.

2.3. Read CSV into records

Add the rest of the program below the header:

```
import cli.Stdout
import parser.CSV

Planet : { name : Str, moons : U64 }

input =
  \\name,moons
  \\Mercury,0
  \\Earth,1
  \\Mars,2

planets : Str -> Try(List(Planet), [InvalidCsv(CSV.Error), MissingRequiredField(Str)])
planets = |text| CSV.parse(text)

describe : Str -> Str
describe = |text| {
  match planets(text) {
    Ok(rows) => Str.join_with(rows.map(|p| "${p.name}: ${p.moons.to_str()} ${if p.moons
== 1 "moon" else "moons"}"), "\n")
    Err(InvalidCsv(error)) => "line ${error.line.to_str()}: ${error.message}"
    Err(MissingRequiredField(name)) => "no ${name} column"
  }
}

main! = |_args| {
  Stdout.line!(describe(input))
}
```

Run it:

```
roc main.roc
```

Output:

```
Mercury: 0 moons
Earth: 1 moon
Mars: 2 moons
```

The first run takes longer while the compiler downloads the platform and the package.

2.4. What the program does

`import parser.CSV` makes the CSV module from the package named `parser` available.

`Planet` is the record type each row becomes. The first line of the input is a header row, and each field of `Planet` reads the column with the same name.

`planets` calls `CSV.parse`, which splits the text into rows and decodes each row into a record. `CSV.parse` does not take the row type as an argument: the compiler infers it from the type annotation on `planets`, and finds the decoding code through the static dispatch method `parser_for`, which every record type has. Change the annotation and the same call decodes a different record.

The result is `Ok` with a list of `Planet` records, or `Err` with one of two tags:

- `InvalidCsv(error)` when the text is not valid CSV or a cell does not fit its field. `error` says which record and field failed, where it starts in the text, and why.
- `MissingRequiredField(name)` when the header has no column for a field. The compiler adds this tag to the error type of every type-directed decoder, so the annotation must name it.

`describe` handles all three cases with a `match`. Nothing is printed until every row has been read successfully.

To see the error path, change `Mars, 2` to `Mars, two` and run the program again. It prints a message that names line 4 and quotes `two`. Rename the `moons` column and it prints `no moons column`. [Report parse errors](#) shows how to report every kind of failure, and [Read CSV data](#) shows optional columns, empty cells and hand-built row parsers that match columns by position.

2.5. Next steps

- [Your first parser](#) teaches the building blocks by writing a parser for a small settings format.
- The format chapters show how to read CSV, YAML, XML, Markdown and HTTP.
- [Parse a format of your own](#) shows how to build and test a parser for a format of your own.

Chapter 3. Choosing how to read your input

Roc gives you four ways to turn text into values, and `roc-parser` provides two of them. This chapter helps you pick one before you write any code. In short:

1. **Decode into your own type** when the input is a standard format and you already know the shape you want. The record type is the specification, and there is almost no code to write.
2. **Read a document tree** with one of `roc-parser`'s format modules when you do not know the shape in advance, or you need what a record would throw away.
3. **Build a parser with combinators** when the format is your own, or you need control over exactly what is accepted and what the errors say.
4. **Use no parser at all** when a `Str` function does the job, or when the input is too large or too hot for any of these.

The examples come from `choosing-approaches.roc`; every `expect` in it passes.

3.1. Decode into your own type

Roc has type-directed decoding built in. A *format* (such as the builtin `Json`) knows how to read strings, numbers, lists and records. Your type knows which of those it is made of. The compiler combines the two, so the annotation alone decides what is accepted:

```
# (a) Type-directed decoding: the record type is the whole specification.
Service : { name : Str, port : U64 }

decode_service : Str -> Try(Service, [InvalidJson(Str), MissingRequiredField(Str)])
decode_service = |json| Json.parse(json)

expect decode_service("{\"name\": \"web\", \"port\": 8080}") == Ok({ name: "web", port: 8080 })
expect decode_service("{\"name\": \"web\"}") == Err(MissingRequiredField("port"))
```

The [parsers chapter of the language reference](#) describes the mechanism. Every type has a `parser_for` method, which the compiler derives for records, lists and tuples, and a format is a type module whose methods read one value at a time. Static dispatch joins the two. A field of type `Try(_, [Missing])` is optional, a missing required field is reported by name, and when the parser is a

top-level constant it is assembled at compile time.

Choose this when:

- the input is a format with a `parser_for` implementation, and
- the mapping from the format to your type is the obvious one: object keys are field names, arrays are lists, and so on.

roc-parser's `CSV` and `Yaml` modules are formats for this mechanism. `CSV.parse` reads a file with a header row into a list of records, matching columns to fields by name, and `Yaml.decode` reads a document into a record. Neither takes the type as an argument: the annotation chooses it. A required field with no column or key fails with `MissingRequiredField(name)`, a tag the compiler adds to the error type, so the annotation names it alongside the format's own error:

```
# roc-parser's CSV and Yaml modules are formats for the same mechanism.
Planet : { name : Str, moons : U64, rings : Try(Bool, [Missing]) }

planets : Str -> Try(List(Planet), [InvalidCsv(CSV.Error), MissingRequiredField(Str)])
planets = |text| CSV.parse(text)

expect planets("name,moons\nMars,2\n") == Ok([ { name: "Mars", moons: 2, rings: Err(Missing) } ])
expect planets("name\nMars\n") == Err(MissingRequiredField("moons"))

service : Str -> Try(Service, [InvalidYaml(Yaml.Error), MissingRequiredField(Str)])
service = |text| Yaml.decode(text)

expect service("name: web\nport: 8080\n") == Ok({ name: "web", port: 8080 })
expect service("name: web\n") == Err(MissingRequiredField("port"))
```

Its limits come from the same design. The format decides the representation, so you cannot ask it to accept `yes` as a boolean or to read a date in your own layout. Where a wrong value is depends on the format: `CSV.parse` reports the record, field, line and column, and `Yaml.decode` the line and column. And the type must be known when you compile.

3.2. Read a document tree

roc-parser's `Yaml`, `Xml` and `Markdown` modules read a whole document into a tree that mirrors the format, and `HTTP` reads a message into a record of its parts. You then walk the tree with pattern matching, or with the tree methods such as `get_path` and `as_i64` on a `Yaml` value and `attribute` and `children_named` on an `Xml.Node`:

```
# (b) A ready-made format parser: read the whole tree, then decide what to
# do with each key, whatever keys the file happens to contain.
top_level_keys : Str -> Try(List(Str), [NotAMapping, InvalidYaml(Yaml.Error)])
```

```

top_level_keys = |text| {
  match Yaml.parse_str(text)? {
    Mapping(entries) => Ok(entries.map(|entry| entry.key))
    _ => Err(NotAMapping)
  }
}

expect top_level_keys("name: web\nport: 8080\nextra: [1, 2]") == Ok(["name", "port",
"extra"])

# The tree methods follow a path without a match per level.
port_of : Str -> Try(I64, [Missing, WrongType, InvalidYaml(Yaml.Error)])
port_of = |text| Yaml.parse_str(text)?.get_path(["server", "port"])?..as_i64()

expect port_of("server:\n  port: 8080\n") == Ok(8080)
expect port_of("server: {}\n") == Err(Missing)

```

Choose this when:

- the shape is not known in advance: any key may appear, keys vary between files, or you are writing a tool that works on every document, such as a linter or a converter;
- you need what a record type cannot hold: the order of mapping keys, the mix of text and child elements in XML, or the block structure of a Markdown document;
- you want a precise error with a line and column from a parser that follows the format's specification, and you check the content yourself afterwards. The format modules drop comments, so a tool that must preserve them needs a parser of its own.

The cost is code: you write the walk, and a missing or mistyped key is something you check, not something the type system checks for you. The format chapters ([Read YAML configuration and frontmatter](#), [Read XML documents](#), [Markdown](#) and [HTTP messages](#)) show helper functions that keep the walk short.

3.3. Build a parser with combinators

When no format module reads your input, write a parser from the `Parser` and `Utf8` combinators. A parser is a value made of smaller parsers, so the code has the shape of the grammar:

```

# (c) A parser of your own: a legacy "host:port" line, where the port must
# be digits and the grammar is yours to define.
Endpoint : { host : Str, port : U64 }

endpoint : Parser(Utf8.Bytes, Endpoint)
endpoint =
  Parser.const(|host| |port| { host, port })
    .keep(Parser.chomp_while(|b| b != ':').map(Str.from_utf8_lossy))

```

```

        .skip(Utf8.codeunit(':'))
        .keep(Utf8.digits)

expect Utf8.parse_str(endpoint, "example.com:443") == Ok({ host: "example.com", port: 443 })
expect Utf8.parse_str(endpoint, "example.com:https").is_err()

```

Choose this when:

- the format is your own or a legacy one: a log line, a configuration dialect, a wire protocol, or a small language such as a query or a template;
- the grammar is defined by more than a record shape: alternatives, nesting, repetition with separators, or a part whose length is given earlier in the input;
- you want to reject invalid values while parsing, so that a port of `70000` or a date of `2026-02-30` fails at the right place with your own message;
- you want to read only the start of the input and keep the rest, such as one message from a buffer that holds several (`Utf8.parse_str_partial`).

Combinators are also how you extend a format module. `CSV.record` and `CSV.field` are combinators, every format module has a `parser` value to embed, and you can mix `Utf8` parsers into any parser of your own. A combinator parser's failure is the furthest failure, with a byte offset. [Your first parser](#) teaches the method and [Parse a format of your own](#) applies it to a real format.

3.4. Use no parser at all

Parser combinators are not always the right tool, and `roc-parser`'s own format modules show where the line falls.

The input has no structure worth a grammar

One separator, no quoting and no nesting is a job for `Str.split_on`, `Str.trim` and friends. A parser would be longer and no clearer:

```

# (d) No parser at all: one separator and no nesting is a job for `Str`.
fields : Str -> List(Str)
fields = |line| line.split_on("\t")

expect fields("a\tb\tc") == ["a", "b", "c"]

```

The input is very large or arrives as a stream

Every parser in this package reads a complete input held in memory, and the result holds copies of the text it read. For a multi-gigabyte log or a network connection, read and process one record at a time with a hand-written loop over the bytes, or use a streaming library.

The parser is on a hot path

Each combinator adds a function call and an intermediate `Try`. That is cheap for configuration files and messages, but [Add a format module](#) records that the CSV, XML, HTTP and Markdown modules moved their document-level parsing to byte scanners with an explicit stack, to stay linear and avoid deep recursion. Do the same when profiling shows the parser matters.

[Performance](#) compares the built-in parsers with Go, Rust and Python libraries.

The grammar is ambiguous or left-recursive

Combinators take the first alternative that succeeds and cannot handle a rule that starts with itself, such as `expr = expr "-" term`. A large grammar written for a parser generator, such as an existing yacc or ANTLR grammar, is often better served by a generator, which checks the grammar for conflicts before any input is read. [Where the design comes from](#) explains these limits.

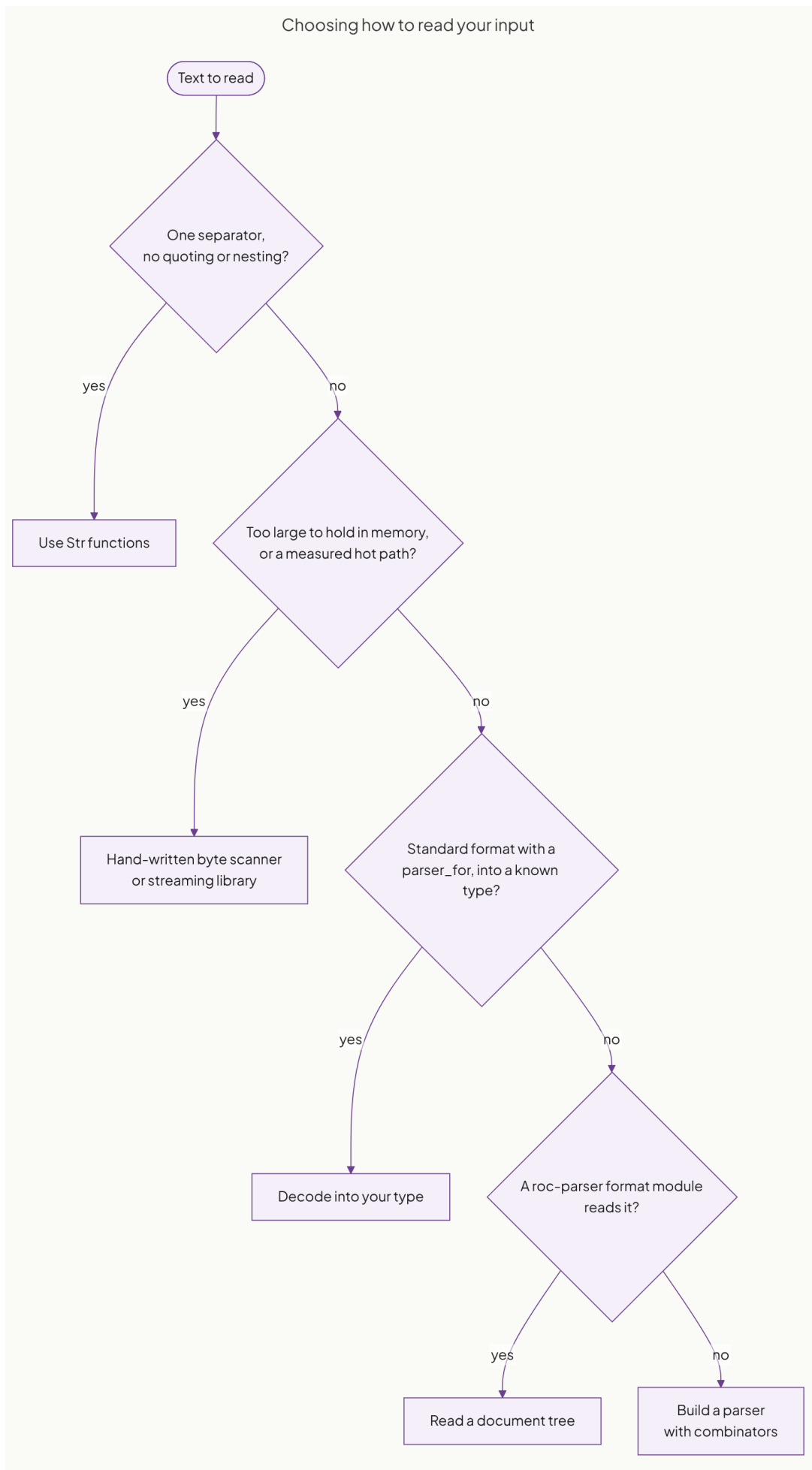
3.5. Decision table

You need	Decode into a type	Document tree	Combinators	No parser
A standard format (CSV, YAML) into known records, with little code	Best	Possible	Possible	No
Any keys or structure, discovered at run time	No	Best	Possible	No
Mapping key order or XML mixed content kept	No	Yes	Yes	No
Line and column in errors	Yes for <code>CSV</code> and <code>Yaml</code>	Yes	A byte offset	No
Your own or a legacy format, a protocol or a small language	No	No	Best	If trivial
Values validated while parsing, with your own messages	No	Afterwards	Best	By hand
Read one message and keep the rest	No	<code>HTTP.parse_request</code> returns the rest	Yes	By hand

You need	Decode into a type	Document tree	Combinators	No parser
Very large or streaming input, or a hot path	No	No	No	Best
Ambiguous or left-recursive grammar	No	No	No	Use a parser generator

3.6. Decision flow

The same choice as a sequence of questions:



The questions are ordered from least code to most. You can also combine answers: read a Markdown file as a tree, read its frontmatter as a YAML tree, and parse one custom field inside it with a combinator.

Chapter 4. Your first parser

In this tutorial you write a parser for a small settings format, one line per setting:

```
name="demo"
port=8080
debug=true
```

By the end you have a parser that turns that text into a list of typed records, and you know how to read its errors. Along the way you meet the combinators that every parser in this package is built from.

You need the setup from [Getting started](#). Each step is a complete program in the repository's `docs/examples` directory; the step's heading names the file. Run a step with `roc <file>` and its tests with `roc test <file>`.

4.1. Step 1: read a key

`combinator-primer-step1.roc`

A key is one or more lowercase letters or underscores. Start with a parser for one such byte, repeat it, and turn the bytes into a `Str`:

```
is_key_byte : U8 -> Bool
is_key_byte = |b| (b >= 'a' and b <= 'z') or b == '_'

key : Parser(Utf8.Bytes, Str)
key =
  Utf8.codeunit_satisfies(is_key_byte)
    .one_or_more()
    .map(Str.from_utf8_lossy)
```

Three ideas appear here:

- A `Parser(Utf8.Bytes, Str)` reads from UTF-8 bytes (`Utf8.Bytes` is `List(U8)`) and produces a `Str`. Every parser has an input type and a result type.
- `Utf8.codeunit_satisfies(is_key_byte)` reads one byte if `is_key_byte` accepts it, and fails otherwise.

- `.one_or_more()` repeats a parser until it fails and collects the results in a list. `.map(f)` transforms the result with an ordinary function; here `Str.from_utf8_lossy` turns the bytes into a `Str`.

`Utf8.parse_str` runs a parser on a whole string. These tests pass:

```
expect Utf8.parse_str(key, "name") == Ok("name")
expect Utf8.parse_str(key, "Name").is_err()
```

The second fails because `N` is not a lowercase letter, so not even one key byte can be read.

4.2. Step 2: read a whole setting

`combinator-primer-step2.roc`

A setting is a key, an `=`, and a value. For now, the value is everything up to the end of the line:

```
value : Parser(Utf8.Bytes, Str)
value =
  Parser.chomp_while(|b| b != '\n')
    .map(Str.from_utf8_lossy)

Entry : { key : Str, value : Str }

entry : Parser(Utf8.Bytes, Entry)
entry =
  Parser.const(|k| |v| { key: k, value: v })
    .keep(key)
    .skip(Utf8.codeunit('='))
    .keep(value)
```

This is the pattern for reading things in sequence:

1. `Parser.const(f)` is a parser that reads nothing and produces `f`, the function that builds the result. `f` takes its arguments one at a time (`|k| |v| ...`), because they arrive one at a time.
2. `.keep(p)` runs `p` next and passes its result to the function.
3. `.skip(p)` runs `p` next and throws its result away. The `=` must be there, but you do not need it in the record.

`Parser.chomp_while(pred)` reads bytes while `pred` accepts them. It always succeeds, even when it reads nothing, which is why `"empty="` gives an empty value:

```
expect Utf8.parse_str(entry, "name=roc") == Ok({ key: "name", value: "roc" })
expect Utf8.parse_str(entry, "empty=") == Ok({ key: "empty", value: "" })
```

Running the step prints:

```
Ok({ key: "name", value: "roc" })
```

4.3. Step 3: choose between kinds of value

`combinator-primer-step3.roc`

A value is a number, true or false, or text in double quotes. Write one parser for each kind, then try them in order with `Parser.one_of`:

```
Value : [Number(U64), Flag(Bool), Text(Str)]

number : Parser(Utf8.Bytes, Value)
number = Utf8.digits.map(|n| Number(n))

flag : Parser(Utf8.Bytes, Value)
flag =
  Parser.one_of([
    Parser.const(Flag(Bool.True)).skip(Utf8.string("true")),
    Parser.const(Flag(Bool.False)).skip(Utf8.string("false")),
  ])

text : Parser(Utf8.Bytes, Value)
text =
  Parser.chomp_while(|b| b != '"')
    .map(|bytes| Text(Str.from_utf8_lossy(bytes)))
    .between(Utf8.codeunit('"'), Utf8.codeunit('"'))

value : Parser(Utf8.Bytes, Value)
value = Parser.one_of([number, flag, text])
```

`Parser.one_of` tries each parser on the same input and returns the first success. If one fails part-way through, the next one starts again from the beginning, so `text` does not see input already examined by `number` or `flag`.

`Utf8.digits` reads a whole number as a `U64`, and `Utf8.string("true")` reads exactly that text. `.between(open, close)` reads `open`, the parser, and `close`, and keeps only the middle result.

```
expect Utf8.parse_str(value, "8080") == Ok(Number(8080))
expect Utf8.parse_str(value, "true") == Ok(Flag(Bool.True))
expect Utf8.parse_str(value, "\"hello world\"") == Ok(Text("hello world"))
expect Utf8.parse_str(value, "maybe").is_err()
```

The `entry` parser from step 2 now uses this `value`. Running the step prints:

```
Ok({ key: "port", value: Number(8080) })
```

4.4. Step 4: read every line, and read the errors

combinator-primer-step4.roc

A settings file is entries separated by line breaks. `sep_by` reads that shape and drops the separators:

```
config : Parser(Utf8.Bytes, List(Entry))
config = entry.sep_by(Utf8.codeunit('\n'))
```

```
expect
  Utf8.parse_str(config, "port=8080\ndebug=true")
    == Ok([ { key: "port", value: Number(8080) }, { key: "debug", value: Flag(Bool.True) } ])
})
```

`Utf8.parse_str` returns one of three results, and a real program should handle each:

```
report : Str -> Str
report = |source| {
  match Utf8.parse_str(config, source) {
    Ok(entries) => "parsed ${entries.len().to_str()} entries"
    Err(ParseError({ message, offset })) => "failed at byte ${offset.to_str()}: ${message}"
  }
}

report_entry : Str -> Str
report_entry = |source| {
  match Utf8.parse_str(entry, source) {
    Ok(_) => "parsed one entry"
    Err(ParseError({ message, offset })) => "failed at byte ${offset.to_str()}: ${message}"
  }
}

main! = |_args| {
  Stdout.line!(report("name=\"demo\"\nport=8080\ndebug=true"))?
  Stdout.line!(report("name=\"demo\"\nport=eighty"))?
  Stdout.line!(report("name=\"demo\"\n"))?
  Stdout.line!(report_entry("port=eighty"))
}
```

Output:

```
parsed 3 entries
failed at byte 17: expected char ``
failed at byte 12: expected a codeunit satisfying a condition, but input was empty.
```

```
failed at byte 5: expected char ``
```

Each line of output shows one outcome:

1. The whole input matched: `Ok`.
2. `port=eighty` is not a valid entry, so `sep_by` stopped after the first entry and did not reach the end of the input. `parse_str` reports the furthest failure instead: none of the value parsers accepted `eighty`, at byte 17.
3. The trailing line break is a separator with no entry after it, so the parse fails at the end of the input, where a key was expected. Real files often end with a line break; [Parse a format of your own](#) shows one way to accept it.
4. Parsing a single bad entry directly gives the same failure, at byte 5.

The second line is worth remembering. A repetition such as `sep_by` or `many` stops at the first element it cannot read, but that element's failure is kept, so a mistake in the middle of a file is reported where it is. [Report parse errors](#) explains how to get better messages.

4.5. What you have learned

- A parser has an input type and a result type, and `Utf8.parse_str` runs one on a whole string.
- `Parser.const(f).keep(a).skip(b).keep(c)` reads things in sequence.
- `.map(f)` transforms a result; `Parser.one_of` chooses between alternatives; `one_or_more`, `many` and `sep_by` repeat.
- A parse ends in `Ok` or a `ParseError` with a message and a byte `offset`.

[Combinators by task](#) covers the remaining combinators and the rules for alternatives and repetition in more depth.

Guides

Chapter 5. Read CSV data

This chapter shows how to turn comma-separated text into Roc values: as records chosen by the type you ask for, as tuples when the file has no header, as raw fields, and with hand-built record parsers when a column needs custom handling. It also shows how to tell the user which row was wrong. It is for Roc programmers who have added `parser` to their app's dependencies.

Every example on this page comes from `docs/examples/csv-guide.roc`, which CI runs, and imports `parser.CSV`, `parser.Parser` and `parser.UTF8`.

5.1. Decode rows into records

When the first row names the columns, `CSV.parse` decodes every other row into the record type you annotate. Each field reads the column whose header is exactly its name, so the column order does not matter:

```
Product : { name : Str, quantity : U64, price : Dec }

products : Str -> Try(List(Product), [InvalidCsv(CSV.Error), MissingRequiredField(Str)])
products = |text| CSV.parse(text)

print_products! : Str => Try({}, _)
print_products! = |text| {
  match products(text) {
    Ok(found) =>
      for p in found {
        Stdout.line!("${p.name}: ${p.quantity.to_str()} at ${p.price.to_str()}")?
      }

    Err(InvalidCsv({ line, column, message, record: _, field: _ })) =>
      Stdout.line!("line ${line.to_str()}, column ${column.to_str()}: ${message}")?

    Err(MissingRequiredField(name)) =>
      Stdout.line!("no column named ${name}")?
  }
  Ok({})
}
```

Running it on a good file, on a file without a `quantity` column, and on a file whose second data row is short:

```
print_products!("name,price,quantity\nbolt,0.15,250\nnut,.05,+40\n")?
print_products!("name,price\nbolt,0.15\n")?
print_products!("name,price,quantity\nbolt,0.15,250\nnut,0.05\n")?
```

Output

```
bolt: 250 at 0.15
nut: 40 at 0.05
no column named quantity
line 3, column 1: the record has 2 fields, fewer than the 3 columns
```

`MissingRequiredField(name)` is not declared by `CSV.parse`: the compiler adds it to the error type of any decoder whose record has required fields, so include it in your annotation. Columns that no field names are ignored, and when two columns share a name the last one wins.

A cell decodes according to its field's type:

Type	Accepted cell text
<code>Str</code>	Any text, kept exactly, including leading and trailing spaces.
<code>Bool</code>	<code>true</code> or <code>false</code> in any letter case.
<code>U8 ... U128, I8 ... I128</code>	ASCII digits with an optional leading <code>+</code> (or <code>-</code> for signed types) that fit in the type. Spaces, <code>_</code> and <code>0x</code> prefixes are rejected.
<code>Dec</code>	An optional sign, digits and an optional fraction: <code>1.50</code> , <code>-2</code> , <code>.5</code> .
<code>F32, F64</code>	An optional sign, then <code>inf</code> , <code>infinity</code> or <code>nan</code> in any letter case, or a decimal number with an optional fraction and exponent: <code>12</code> , <code>1.5</code> , <code>.5</code> , <code>5.</code> , <code>1e-3</code> , <code>2.5E+10</code> . Values too large become infinity and values too small become zero.
Tags without payloads	The tag name, such as <code>Active</code> .

These match Rust's `from_str` for each number type. A field that is a record, a tuple or a tag with a payload fails with an `InvalidCsv` error when it is decoded; a field that is a list or a dictionary is a compile-time error, because a CSV cell cannot hold one.

5.1.1. Optional columns, empty cells and header spelling

A field of type `Try(a, [Missing])` is `Err(Missing)` when the file has no such column. A field of type `Try(a, [Null])` is `Err(Null)` when its cell is empty. `CSV.parse_normalized` matches header names after trimming spaces, lowercasing ASCII letters and turning runs of spaces, - and into one, so `First Name`, `first-name` and `FIRST_NAME` all fill `first_name`:

```

Contact : {
  name : Str,
  email : Try(Str, [Missing]),
  age : Try(U64, [Null]),
  status : [Active, Inactive],
}

contacts : Str -> Try(List(Contact), [InvalidCsv(CSV.Error), MissingRequiredField(Str)])
contacts = |text| CSV.parse_normalized(text)

describe_contact : Contact -> Str
describe_contact = |{ name, email, age, status }| {
  email_text =
    match email {
      Ok(address) => address
      Err(Missing) => "no email column"
    }
  age_text =
    match age {
      Ok(years) => years.to_str()
      Err(Null) => "age not given"
    }
  status_text =
    match status {
      Active => "active"
      Inactive => "inactive"
    }
  "${name} (${email_text}, ${age_text}, ${status_text})"
}

```

For `"Name, Age, Status\nAda, 36, Active\nAlan, , Inactive\n"` this prints:

Output

```

Ada (no email column, 36, active)
Alan (no email column, age not given, inactive)

```

5.1.2. Row width and blank lines

Blank lines are skipped. Every other row must have exactly as many fields as the header: a shorter row and a longer row are both an `InvalidCsv` error naming the row, so that a misplaced comma cannot shift values into the wrong columns. A header with no data rows, and empty input, decode to an empty list.

5.2. Files without a header

`CSV.parse_headerless` decodes every row into a tuple, one element per column, with the same cell rules. Every row must have as many fields as the tuple has elements:

```
points : Str -> Try(List((Str, F64, F64)), [InvalidCsv(CSV.Error)])
points = |text| CSV.parse_headerless(text)
```

Output

```
Ok([("home", -33.87, 151.21), ("work", -33.86, 151.2)])
```

5.3. Read every field as raw bytes

Use `CSV.parse_records` when the columns vary or you want to inspect rows yourself. It returns one `CSV.Record`, a `List(List(U8))`, per row:

```
raw_records : Str -> Str
raw_records = |text| {
  match CSV.parse_records(text) {
    Ok(records) =>
      Str.join_with(records.map(|fields| Str.join_with(fields.map(Str.from_utf8_lossy),
" | ")), "\n")

    Err(InvalidCsv({ message, line, column, record: _, field: _ })) =>
      "not valid CSV at ${line.to_str()}:${column.to_str()}: ${message}"
  }
}
```

Given a header row, a quoted field containing `"` and `,`, and a row whose last field is empty:

```
Stdout.line!(raw_records("name,note\r\nwidget,\"says \\\"hi\\\"\"", twice"\nempty,"))?
```

Output

```
name | note
```

```
widget | says "hi", twice
empty |
```

`CSV.split_header` separates the first record as column names:

```
column_names : Str -> Str
column_names = |text| {
  match CSV.parse_records(text) {
    Ok(records) => {
      { header, rows } = CSV.split_header(records)
      "${Str.join_with(header, ", ")}; ${rows.len().to_str()} rows"
    }
    Err(_) => "not valid CSV"
  }
}
```

Output

```
sku, count; 2 rows
```

`CSV.parser` is the same reader as a `Parser(Utf8.Bytes, List(CSV.Record))`, for use inside larger parsers. It accepts any bytes.

5.4. Hand-built record parsers

When a column needs custom handling, such as a list stored in one cell, describe one row with `CSV.record`, giving it a function that takes one argument per column, then add one `.keep(CSV.field(...))` per column in order. A field parser is any `Parser(Utf8.Bytes, a); CSV.string`, `CSV.u64` and `CSV.f64` read cells with the rules above:

```
Movie : { title : Str, year : U64, cast : List(Str) }

cast : Parser(Utf8.Bytes, List(Str))
cast = CSV.string.map(|text| Str.split_on(text, ";"))

movie : Parser(CSV.Record, Movie)
movie =
  CSV.record(|title| |year| |actors| { title, year, cast: actors })
    .keep(CSV.field(CSV.string))
    .keep(CSV.field(CSV.u64))
    .keep(CSV.field(cast))
```

Run it on text with `CSV.parse_with`, or on records you already have with `CSV.decode`. Columns are matched by position, and the record parser must read every field in the row: a row with more fields

than `.keep` calls fails, and so does a row with fewer.

5.5. Report which row failed

Every function stops at the first problem and returns `InvalidCsv(error)`. The `CSV.Error` record says where the problem is:

Field	Meaning
<code>record, field</code>	The one-based record and field numbers. Records count every record of the input, including the header row and blank lines.
<code>line, column</code>	The one-based line and byte column where the field (or, when the field does not exist, the record) starts. They are 0 from <code>CSV.decode</code> , which does not see the text.
<code>message</code>	What is wrong. It quotes at most a short excerpt of the field, so it is safe to show for any input.

```
describe : Str -> Str
describe = |text| {
  match CSV.parse_with(movie, text) {
    Ok(movies) => "${movies.len().to_str()} movies"
    Err(InvalidCsv({ record, field, line, column, message })) =>
      "record ${record.to_str()}, field ${field.to_str()} (line ${line.to_str()},
column ${column.to_str()}: ${message}"
  }
}
```

```
Stdout.line!(describe("Airplane!,1980,Robert Hays;Julie Hagerty"))?
Stdout.line!(describe("Airplane!,1980,Robert Hays\nCaddyshack,soon,Bill Murray"))?
Stdout.line!(describe("Airplane!,1980,Robert Hays,extra"))?
```

Output

```
1 movies
record 2, field 2 (line 2, column 12): expected a U64, found `soon`
record 1, field 4 (line 1, column 28): the record has 4 fields, but the parser read only 3
```

5.6. Which CSV files are accepted

The dialect is RFC 4180 with the relaxations that Python's `csv` module (in strict mode) and Rust's `csv` crate also accept:

- Fields are separated by `,`. Records end with CRLF, LF or a lone CR, and the line break after the last record is optional.
- An unquoted field is any run of bytes other than `,`, CR and LF. A `"` inside an unquoted field is an ordinary character.
- A field that starts with `"` is quoted. It may contain `,`, CR and LF, and `""` stands for one `"`. The closing quote must be followed by `,`, a line break or the end of input; an unterminated quoted field is an error.
- Spaces and tabs are kept as part of the field.
- Empty input has no records. A blank line in the middle of the input is a record with one empty field.
- Rows may have different numbers of fields (the typed decoders then check the width).

```
Stdout.line!(raw_records("size,6\" pipe\rcount,"))?
Stdout.line!(raw_records("bolt,\"250\""))?
Stdout.line!("empty input: ${Str.inspect(CSV.parse_records(""))}")?
```

Output

```
size | 6" pipe
count |
not valid CSV at 1:6: unterminated quoted field
empty input: Ok([])
```

5.7. Limitations

- The separator is always `,`. Tab- or semicolon-separated files are not supported.
- A UTF-8 byte order mark is not removed; it becomes part of the first field.
- The whole input is parsed before decoding starts, so very large files are held in memory.

For every function and type in the module, see the [CSV API reference](#).

5.8. Performance

Reading every field as a string runs at 120–150 MB/s on an Apple M2, from a 1.6 KB export to a 1 MiB file: on par with Rust’s `csv` crate, about 2.3 times slower than Go’s `encoding/csv` and nearly twice as fast as Python’s C `csv` module. Delimiters and quotes are found 16 bytes at a time with SIMD, and fields without doubled quotes are slices of the input. The parser is a byte scanner and stays linear on very wide rows and long runs of escaped quotes. Decoding typed columns adds the cost of each conversion. The whole file and every field are held in memory: about 12 MiB of peak memory for a 1 MiB file. See [Performance](#) for the method and caveats.

Chapter 6. Read YAML configuration and frontmatter

This chapter shows how to read a YAML configuration file or the frontmatter at the top of a Markdown post into a Roc value, pull settings out of it, and report mistakes with a line and column. It is for Roc programmers who have added `parser` to their app's dependencies and know the YAML they want to read.

The parser handles the YAML that configuration files and frontmatter use in practice, and rejects the rest with an error instead of guessing. If your files use anchors, tags or several documents in one file, read [what is rejected](#) first.

Every example on this page comes from `docs/examples/yaml-guide.roc`, which CI runs, and imports `parser.Yaml`.

6.1. Decode a document into a record

`Yaml.decode` reads a document straight into your own Roc type. You do not pass the type: it is inferred from how the result is used, usually from a type annotation. This configuration, held in a Roc multi-line string (each `\\` starts one line of the file):

```
config_text =
  \\# deployment settings
  \\name: web
  \\replicas: 3
  \\debug: false
  \\ports: [80, 443]
  \\owners:
  \\  - name: Ada
  \\    email: ada@example.com
  \\  - name: Grace
```

is read by:

```
Owner : { name : Str, email : Try(Str, [Missing]) }

Config : {
```

```

    name : Str,
    replicas : U8,
    debug : Bool,
    ports : List(U16),
    owners : List(Owner),
  }

## The record type tells `Yaml.decode` what to expect.
load_config : Str -> Try(Config, [InvalidYaml(Yaml.Error), MissingRequiredField(Str)])
load_config = |text| Yaml.decode(text)

summarize : Str -> Try(Str, [InvalidYaml(Yaml.Error), MissingRequiredField(Str)])
summarize = |text| {
  config = load_config(text)?
  owner_names = config.owners.map(|owner| owner.name)

  Ok("${config.name}: ${config.replicas.to_str()} replicas, owners ${Str.join_with
(owner_names, " and ")}")
}

```

`summarize(config_text)` returns:

Output

```
web: 3 replicas, owners Ada and Grace
```

The rules that map YAML onto Roc types:

- A mapping decodes into a record, matching keys to field names, or into a `Dict` with `Str` or integer keys. Keys the record does not have are skipped.
- A sequence decodes into a `List`, or into a tuple when it has exactly as many items as the tuple.
- A plain (unquoted) scalar is resolved for the type that reads it, with the YAML 1.2 core schema (see [how plain values are typed](#)). A `Str` field takes any scalar except a null, so `version: 1.10` gives `"1.10"` rather than a float; a `U8` field takes `0x1F`; a `Bool` field takes `true` or `FALSE`, but not the string `"true"`.
- A tag union without payloads, such as `[Debug, Info]`, takes the tag's name.
- A null (`~`, `null` or an empty value) decodes as an empty list, dict or record, and as `Err(Null)` in a `Try(_, [Null])` field.
- A `Try(_, [Missing])` field is `Err(Missing)` when its key is absent, like `email` for Grace above.

A value of the wrong shape or out of range fails with `InvalidYaml` at that value's line and column, the same error invalid YAML gives. A required field whose key is absent fails with `MissingRequiredField(name)`. Your code does not declare that tag: the compiler adds it to the error type of every decoder of a record with required fields, so name it in the annotation:

```
Stdout.line!(Str.inspect(load_config("name: web\nreplicas: 300\n")))?
Stdout.line!(Str.inspect(load_config("name: web\nreplicas: 3\ndebug: no\n")))?
Stdout.line!(Str.inspect(load_config("name: web\nreplicas: 3\ndebug: false\nports: []\n")))?
```

Output

```
Err(InvalidYaml({ column: 11, line: 2, message: "integer `300` is outside the range of U8"
}))
Err(InvalidYaml({ column: 8, line: 3, message: "expected a boolean, found `no`" }))
Err(MissingRequiredField("owners"))
```

6.1.1. Other key spellings and unknown keys

`Yaml.decoder` builds a decoder with other conventions. `keys` says how YAML spells Roc's `snake_case` field names: `SnakeCase` (as written), `KebabCase` (`user-id`) or `CamelCase` (`userId`). `unknown_keys: Reject` turns a key the record does not have into an `InvalidYaml` error, which catches typos in hand-written configuration:

```
decode_strict = Yaml.decoder({ keys: KebabCase, unknown_keys: Reject })

load : Str -> Try({ user_id : U64 }, [InvalidYaml(Yaml.Error), MissingRequiredField(Str)])
load = |text| decode_strict(text)
```

Build the decoder once, as a top-level constant, and call it for each document. To write your own generic function over decodable types, require `a.Yaml.Parseable(errors)` in its `where` clause, the way `Yaml.decode` does.

6.2. Explore a document as a tree

When the shape is not known in advance, `Yaml.parse_str` returns a `Yaml` tree:

```
Yaml := [
  Null,
  Bool(Bool),
  Int(I64),
  Float(F64),
  Text(Str),
  Sequence(List(Yaml)),
  Mapping(List({ key : Str, value : Yaml })),
]
```

A mapping keeps its entries in source order. `get` looks up a key, `at` an index, and `get_path` follows several of them, using decimal strings such as `"0"` for sequence indexes. Each returns `Err(Missing)`

when there is nothing there. `as_str`, `as_i64`, `as_bool` and `as_list` unwrap one variant or return `Err(WrongType)`:

```
## Explore a document without a record type.
first_owner : Str -> Try(Str, [InvalidYaml(Yaml.Error), Missing, WrongType])
first_owner = |text| {
  config = Yaml.parse_str(text)?
  name = config.get_path(["owners", "0", "name"])?
  name.as_str()
}
```

For `config_text`, and then for `"owners: []"`:

Output

```
Ok("Ada")
Err(Missing)
```

Empty input, or a document holding only comments, parses as `Null`. To combine YAML with other parsers, `Yaml.parser` is a `Parser(Utf8.Bytes, Yaml)` that reads the rest of its input; its `ParseError` offset is the byte offset of the error's line and column.

6.3. Read Markdown frontmatter

Frontmatter is a YAML document between two `---` lines at the start of a Markdown file. Split the text at the closing `---` and parse the first part. The parser accepts the opening `---` (and a closing `...`) as document markers, so you do not need to strip them:

```
## Split a Markdown file into its YAML frontmatter and body.
frontmatter : Str -> Try({ meta : Yaml, body : Str }, [NoFrontmatter, InvalidYaml(Yaml.
Error)])
frontmatter = |markdown| {
  match Str.split_on(markdown, "\n---\n") {
    [header, .. as rest] if Str.starts_with(header, "---\n") =>
      Ok({ meta: Yaml.parse_str(header)?, body: Str.join_with(rest, "\n---\n") })
    _ => Err(NoFrontmatter)
  }
}
```

For `"---\ntitle: Hello\ntags: [roc, yaml]\n---\n# Hello\n"`, printing `Yaml.to_inspect(meta)` and then the body gives:

Output

```
Mapping([ { key: "title", value: Text("Hello") }, { key: "tags", value: Sequence([Text("roc"),
```

```
Text("yaml")) }]) }])
# Hello
```

`Yaml.decode` works on the same text when you know the frontmatter's fields.

6.4. How plain values are typed

In a `Yaml` tree, an unquoted value is resolved with the YAML 1.2 core schema. Anything that matches none of these forms is `Text`:

Written as	Becomes
<code>~</code> , <code>null</code> , <code>Null</code> , <code>NULL</code> , or nothing	<code>Null</code>
<code>true</code> , <code>True</code> , <code>TRUE</code> , <code>false</code> , <code>False</code> , <code>FALSE</code>	<code>Bool</code>
Decimal digits with an optional sign; unsigned <code>0x</code> hexadecimal or <code>0o</code> octal	<code>Int</code> (an <code>I64</code> ; a value outside its range is an error)
Decimal with a fraction or exponent; <code>.inf</code> , <code>-.inf</code> , <code>.nan</code> in the three letter cases	<code>Float</code>

```
show!("[~, null, true, FALSE, 42, -7, 0x1F, 0o17, 1.5, 6.02e23, .inf, -.Inf, .nan, 1.2.3,
1_000, 'true', 2026-10-01]")
```

Output

```
Sequence([Null, Null, Bool(True), Bool(False), Int(42), Int(-7), Int(31), Int(15),
Float(1.5), Float(6.02e23), Float(inf), Float(-inf), Float(nan), Text("1.2.3"),
Text("1_000"), Text("true"), Text("2026-10-01")])
```

The last four values show what stays text: a version number such as `1.2.3`, a number with `_` separators, anything in quotes, and dates (YAML 1.2 has no timestamp type). `true` and `0x1` are text too, because the core schema allows only the spellings in the table. In a tree, quote a value whenever it must stay text; `Yaml.decode` reads any non-null plain scalar into a `Str` field as written.

`Yaml.to_inspect` prints a tree in Roc notation, escaping line breaks, tabs and other control characters so each value stays on one line.

6.5. Write multi-line text

Use a block scalar for text that spans lines. `|` (literal) keeps line breaks; `>` (folded) joins lines with spaces and keeps a blank line as a paragraph break. A chomping indicator controls the final line break: none keeps one, `-` removes it, and `+` keeps every trailing blank line. A digit sets the content indentation, so the text can start with spaces:

```
show!(
  \\literal: |
  \\ line one
  \\ line two
  \\folded: >
  \\ joined
  \\ into one line
  \\
  \\ new paragraph
  \\stripped: |-
  \\ no final newline
  \\kept: |+
  \\ trailing blank lines kept
  \\
  \\indented: |2
  \\   starts with two spaces
  ,
)
```

Output

```
Mapping([ { key: "literal", value: Text("line one\nline two\n") }, { key: "folded", value:
Text("joined into one line\nnew paragraph\n") }, { key: "stripped", value: Text("no final
newline") }, { key: "kept", value: Text("trailing blank lines kept\n\n") }, { key:
"indented", value: Text("  starts with two spaces") }])
```

6.6. Write lists

A sequence item may start with a mapping on the same line as its `-` (compact), may hold another sequence the same way, and may sit at the same indentation as its parent key (indentless):

```
show!(
  \\steps:
  \\- run: build
  \\ name: compile
  \\- - nested
  \\ - compact
  ,
)
```

Output

```
Mapping([{"key": "steps", "value": Sequence([Mapping([{"key": "run", "value": Text("build") }, {"key": "name", "value": Text("compile") }])], Sequence([Text("nested"), Text("compact")]))})])
```

Short lists and mappings can also use flow style on one line, as in `ports: [80, 443]` or `labels: {tier: web }`.

6.7. Quote strings

A single-quoted string has no escapes except `'` for one quote. A double-quoted string supports every YAML 1.2 escape, including `\t`, `\n`, `\xXX`, `\uXXXX` and `\UXXXXXXXX`. A `#` inside quotes is text; outside quotes and after a space it starts a comment:

```
show!(
  \single: 'it's # not a comment'
  \plain: text # this is a comment
  ,
)
show!(
  "double: \"tab\\there, \\u00e9, \\x41\"",
)
```

Output

```
Mapping([{"key": "single", "value": Text("it's # not a comment") }, {"key": "plain", "value": Text("text") }])
Mapping([{"key": "double", "value": Text("tab\\there, é, A") }])
```

6.8. What is rejected

The following are outside the supported subset. Each one is an `InvalidYaml` error, never a silently different value:

- anchors (`&name`), aliases (`*name`) and tags (`!!str`);
- directives (`%YAML`) and complex keys (`? key`);
- more than one document in a file;
- a flow collection (`[...]` or `{...}`) that continues onto another line;
- a quoted or plain string that continues onto another line (use a block scalar instead);
- tabs used for indentation;

- duplicate keys in one mapping;
- nesting deeper than 100 levels, counting flow collections;
- in a `Yaml` tree, integers outside the `I64` range (decode them into a wider type such as `U64`, `I128` or `Str` instead);
- control characters other than tab and line breaks.

Mapping keys are kept as their source text, not resolved like values. `1` and `01` are different keys, and `true` is the string key `"true"`.

6.9. Show where an error is

`InvalidYaml` carries a `Yaml.Error` record: a one-based `line` and `column` and a message. Columns count bytes from the start of the line:

```
report : Str -> Str
report = |text| {
  match Yaml.parse_str(text) {
    Ok(value) => Yaml.to_inspect(value)
    Err(InvalidYaml({ line, column, message })) => "line ${line.to_str()}, column
${column.to_str()}: ${message}"
  }
}
```

```
Stdout.line!(report("server:\n\tport: 80"))?
Stdout.line!(report("base: &defaults\n port: 80"))?
Stdout.line!(report("one: 1\n--\ntwo: 2"))?
Stdout.line!(report("name: a\nname: b"))?
Stdout.line!(report("list: [1,\n 2]"))?
Stdout.line!(report("1: one\n01: zero-one"))?
```

Output

```
line 2, column 1: tabs may not be used for YAML indentation
line 1, column 7: anchors, aliases, and tags are not supported by this YAML subset
line 2, column 1: multiple YAML documents are not supported
line 2, column 1: duplicate mapping key `name`
line 1, column 7: unterminated flow collection
Mapping([ { key: "1", value: Text("one") }, { key: "01", value: Text("zero-one") } ])
```

For the full module interface, see the [Yaml API reference](#).

6.10. Performance

Parsing takes time linear in the size of the document, at about 51–55 MB/s on an Apple M2 from 4 KiB to 1 MiB: a 4 KiB configuration in under 0.1 ms and a 1 MiB one in about 20 ms. That is about as fast as `yaml-rust2` (Rust), 1.6 times faster than `serde_yaml`, 2.6 times faster than Go’s `yaml.v3` and seven times faster than PyYAML with `libyaml`. Nesting is limited to 100 levels. Decoding into a record builds the same document first, then walks it once. See [Performance](#) for the measurements and the comparison with other libraries.

Chapter 7. Read XML documents

This chapter shows how to parse an XML document into a tree, find elements, attributes and text in it, and report malformed input with a line and column. It is for Roc programmers who have added `parser` to their app's dependencies and need data out of XML such as feeds, SVG files or configuration.

The parser checks that a document is well-formed according to XML 1.0 and gives you the tree after the usual XML clean-up: references replaced by their text, line endings normalised, comments dropped. It does not validate against a schema or DTD, and it rejects documents that contain a `<!DOCTYPE>`; see [what is not supported](#) before using it on documents from other tools.

Every example on this page comes from `docs/examples/xml-guide.roc`, which CI runs, and imports `parser.Xml`.

7.1. Parse a document and find elements

`Xml.parse_str` takes the whole document and returns an `Xml` record with the document's `root` element and its `declaration`. Each node in the tree is one of:

```
Node := [
  Element({ name : Str, attributes : List(Attribute), children : List(Node) }),
  Text(Str),
]
```

An `Element` holds its name, its attributes in source order, and its children. There is no query language, but `Xml.Node` has four helpers for the common lookups, and you can pattern match on the tree for anything else:

- `node.name()` is `Ok(name)` for an element and `Err(NotAnElement)` for text.
- `node.attribute(name)` is the attribute's value, or `Err(Missing)`.
- `node.children_named(name)` lists the direct child elements with that name.
- `node.text()` joins all text inside the node, like the DOM's `textContent`.

```
describe_entry : Xml.Node -> Str
describe_entry = |entry| {
```

```

id = entry.attribute("id") ?? "?"
titles = Str.join_with(entry.children_named("title").map(|title| title.text()), "")

"entry ${id}: ${titles}"
}

```

Given this document:

```

feed_text =
  \\<?xml version="1.0" encoding="UTF-8"?>
  \\<feed lang="en">
  \\  <!-- newest first -->
  \\  <entry id="2"><title>Fish &amp; Chips</title></entry>
  \\  <entry id="1"><title><![CDATA[<Hello>]]> &#x1F600;</title></entry>
  \\</feed>

```

parse it and describe each entry element under the root:

```

match Xml.parse_str(text) {
  Ok(xml) =>
    for entry in xml.root.children_named("entry") {
      Stdout.line!(describe_entry(entry))?
    }

  Err(InvalidXml(problem)) => Stdout.line!("invalid XML: ${problem.message}")?
}

```

Output

```

entry 2: Fish & Chips
entry 1: <Hello> 🐼

```

The whitespace between elements is kept as `Text` nodes; `children_named` skips them because it only returns elements. `&`, the CDATA section and the character reference `😀` have already been turned into ordinary text.

7.2. What the tree contains

The tree is what an XML processor reports, not the literal source:

- The five predefined entities `<`, `>`, `&`, `'`, `"`, and decimal (`A`) or hexadecimal (`B`) character references are replaced by their characters, in text and in attribute values.
- CDATA sections become text, and adjacent text, including text on both sides of a comment, is merged into one `Text` node.

- Line endings (`\r\n` and lone `\r`) become `\n`.
- In attribute values, tabs and line breaks become spaces.
- Comments and processing instructions are checked and then dropped.
- The XML declaration is available as `declaration: Ok({ version, encoding })` or `Err(Missing)`. The version is a record such as `{ major: 1, minor: 0 }`, and the encoding is `Err(Missing)` when the declaration does not name one.

This document has every one of these in a single element:

```
print_tree!("<p class='a\tb'>x &lt; y &#65;&#x42;\r\n<![CDATA[<raw>]]><!-- gone  
-->z<br/></p>")?
```

Printing the root with `Str.inspect` gives:

Output

```
Element({ attributes: [{ name: "class", value: "a b" }], children: [Text("x < y AB  
<raw>z"), Element({ attributes: [], children: [], name: "br" })], name: "p" })
```

7.3. Report malformed input

When a document is not well-formed, `Xml.parse_str` returns `InvalidXml({ line, column, message })`. Lines and columns are one-based, and columns count UTF-8 bytes, so a column after non-ASCII text is larger than the number of characters a user sees.

```
check : Str -> Str
check = |text| {
  match Xml.parse_str(text) {
    Ok(_) => "well-formed"
    Err(InvalidXml({ line, column, message })) => "${line.to_str()}:${column.to_str()}:  
${message}"
  }
}
```

```
Stdout.line!(check("<a\n <b\n</a>"))?
Stdout.line!(check("<a&nbsp;</a>"))?
Stdout.line!(check("<a x='1' x='2'/>"))?
Stdout.line!(check("<a/><b/>"))?
Stdout.line!(check("<!DOCTYPE html><html/>"))?
```

Output

```
3:1: end tag </a> does not match start tag <b>
```

```

1:4: undeclared entity &nbsp;
1:10: duplicate attribute x
1:5: unexpected content after the root element
1:1: document type declarations are not supported

```

If you are combining XML with other parsers, `Xml.parser` is the same parser as a `Parser(Utf8.Bytes, Xml)`. It leaves any input after the document for the next parser. Its failures are `ParseError({ message, offset })`, with the same message as `Xml.parse_str` and the byte offset of the problem.

7.4. What is not supported

These limits are by design:

Document type declarations

A `<!DOCTYPE ...>` is rejected. As a result only the five predefined entities exist, and any other reference, such as ` `, is an error. Replace HTML entities with character references (` `) before parsing.

Namespaces

Names are kept exactly as written. `svg:path` is an element named `"svg:path"`, and `xmlns` declarations are ordinary attributes; prefixes are not resolved to URIs.

Declared encodings

The input is a `Roc Str`, which is already UTF-8. An `encoding="..."` in the declaration is reported in `declaration` but not used. Convert other encodings to UTF-8 before parsing.

Validation

Only well-formedness is checked. Required elements, attribute types and schemas are not.

For the full module interface, see the [Xml API reference](#).

7.5. Performance

`Xml.parse_str` builds the tree at about 120–135 MB/s on an Apple M2, and the rate stays flat from 4 KiB to 1 MiB documents: the parser is linear in the input size. On the generated documents that is about 2.2 times as fast as Go's `encoding/xml`, and 1.4–2 times slower than Rust's `quick-xml` and `roxmltree`. Character data and attribute values are found 16 bytes at a time with SIMD, the input is validated as UTF-8 once, and names, text and attribute values with nothing to unescape are slices of the input string, so they keep the input alive. Elements are parsed with an explicit stack, so 5,000 levels of nesting are safe, and duplicate-attribute checks are linear: an element with 5,000 attributes parses 19 times faster than with the Rust libraries. See [Performance](#) for the measurements and caveats.

Chapter 8. Markdown

This chapter shows how to turn Markdown text into a tree of Roc values and how to walk that tree to produce your own output, such as HTML or a table of contents. It is a set of how-to guides for readers who already know basic Roc and have added `roc-parser` to an app. By the end you can parse a document, read its blocks and inline content, pull out YAML frontmatter, and write a renderer.

The `Markdown` module follows CommonMark 0.31.2 plus the GitHub Flavored Markdown (GFM) extensions for tables, task list items, strikethrough and extended autolinks. It also reads a leading `---` frontmatter block.

WARNING

Raw HTML in the source passes through unchanged as `HtmlBlock` and `HtmlInline` values. The parser does not apply the GFM tagfilter and does not check link destinations, so a `javascript: URL` is kept as written. If you render Markdown you did not write, sanitise the HTML you produce or drop those nodes and unsafe URLs. See [Walk the tree to produce output](#).

8.1. Parse a document into blocks

Parse a whole document with `Markdown.parse_str`. It returns a `List(Markdown)`: one value per top-level block, in document order.

Markdown has no syntax errors. Every input is a valid document, so `Markdown.parse_str` returns the blocks directly rather than a `Try`; text that looks like broken syntax becomes ordinary paragraph text. To use the document parser inside your own combinators, `Markdown.parser` is the same parser as a `Parser(Utf8.Bytes, List(Markdown))`.

The following app parses a document that uses each kind of block and prints each block with `Str.inspect`, which shows what the parser produced.

```
source =
  \\# Shopping
  \\
  \\Buy these *today*:
  \\
```

```

\\- [x] apples
\\- [ ] pears
\\
\\> Quoted
\\
\\```roc
\\main = 1
\\```
\\
\\| Item | Qty |
\\| :--- | --: |
\\| pear | 2 |
\\
\\***
\\
\\<div>raw</div>

main! = |_args| {
  blocks = Markdown.parse_str(source)
  for block in blocks {
    Stdout.line!(Str.inspect(block))?
  }
  loose_list!()
}

```

A list in which no items are separated by blank lines is *tight*; otherwise it is *loose*. The `loose` field records this, which matters when you render HTML: a tight list normally shows its item paragraphs without `<p>` tags. Ordered lists keep their starting number:

```

loose_list! = || {
  blocks = Markdown.parse_str("3. first\n\n4. second\n")
  for block in blocks {
    Stdout.line!(Str.inspect(block))?
  }
  Ok({})
}

```

Output:

```

Heading({ level: One, content: [Text("Shopping")] })
Paragraph([Text("Buy these "), Emphasis([Text("today")]), Text(":")])
ListBlock({ kind: Unordered, loose: False, items: [{ blocks: [Paragraph([Text("apples")])],
task: Checked }, { blocks: [Paragraph([Text("pears")])], task: Unchecked } ] })
Blockquote([Paragraph([Text("Quoted")])])
Code({ info: "roc", pre: "main = 1
" })
Table({ header: [[Text("Item")], [Text("Qty")]], align: [Left, Right], rows:
[[[Text("pear")], [Text("2")]]] })
ThematicBreak

```

```
HtmlBlock("<div>raw</div>
")
ListBlock({ kind: Ordered({ start: 3 }), loose: True, items: [{ blocks:
[Paragraph([Text("first")])], task: NoTask }, { blocks: [Paragraph([Text("second")])], task:
NoTask }] })
```

The block variants are:

Variant	Contents
Heading({ level, content })	An ATX(#) or Setext (underlined) heading. level is One to Six; level.to_str() gives "1" to "6".
Paragraph(inlines)	A paragraph's inline content.
Blockquote(blocks)	The blocks inside a > quote.
ListBlock({ kind, loose, items })	kind is Unordered or Ordered({ start }). Each item has blocks and a task of NoTask, Unchecked ([]) or Checked ([x] or [X]). Item text is always wrapped in Paragraph, even in a tight list.
Code({ info, pre })	A fenced or indented code block. info is the info string after the opening fence (empty for indented code); pre is the literal content, ending in a newline.
ThematicBreak	A *, --- or _ line.
Table({ header, align, rows })	A GFM table. Each cell is a List(Inline). align holds one Default, Left, Center or Right per column.
HtmlBlock(raw)	Raw HTML, including its trailing newline, exactly as written.
Frontmatter(raw)	The raw text of a leading frontmatter block; only ever the first block. See Read frontmatter .

Block quotes and lists nest at most 1,000 levels deep. Deeper > and list markers are read as paragraph text. Emphasis, strikethrough, links and images also nest at most 1,000 levels deep (counted separately); the delimiters and brackets of deeper ones are read as text. Code that walks the tree recursively therefore cannot run out of stack on hostile input.

The syntax tree types support == and hashing, so blocks and inlines can be Dict keys and Set elements.

Link reference definitions ([label]: /url) do not appear in the tree. The parser resolves them into the links that use them.

8.2. Parse inline content

Inline content is the text inside a paragraph, heading or table cell. When you already have a single line or fragment, such as a title from a database, parse it directly with `Markdown.parse_inlines`, which returns a `List(Inline)`. `Markdown.inline_parser` is the same parser for use in combinators.

```
show_inlines! = |text| {
  inlines = Markdown.parse_inlines(text)
  for inline in inlines {
    Stdout.line!(Str.inspect(inline))?
  }
  Ok({})
}
```

```
show_inlines!("*em* **strong** ~~gone~~ `x + 1`")?
show_inlines!("[Roc](https://roc-lang.org \"Home\") and ![a cat](cat.png)")?
show_inlines!("<https://example.com> www.example.com me@example.com")?
show_inlines!("one\ntwo \nthree")?
```

Output:

```
Emphasis([Text("em")])
Text(" ")
Strong([Text("strong")])
Text(" ")
Strikethrough([Text("gone")])
Text(" ")
InlineCode("x + 1")
Link({ label: [Text("Roc")], target: { href: "https://roc-lang.org", title: Ok("Home") } })
Text(" and ")
Image({ alt: [Text("a cat")], target: { href: "cat.png", title: Err(Missing) } })
Link({ label: [Text("https://example.com")], target: { href: "https://example.com", title:
Err(Missing) } })
Text(" ")
Link({ label: [Text("www.example.com")], target: { href: "http://www.example.com", title:
Err(Missing) } })
Text(" ")
Link({ label: [Text("me@example.com")], target: { href: "mailto:me@example.com", title:
Err(Missing) } })
Text("one
two")
HardBreak
Text("three")
```

The output shows these rules:

- `*` and `_` produce `Strong`, and produce `Emphasis`, and `~` or `~~` produce `Strikethrough`.

- Autolinks in angle brackets and GFM extended autolinks (`www.`, `http://`, `https://` and email addresses written as plain text) become ordinary `Link` values. The parser adds `http://` to `www.` links and `mailto:` to email links.
- A *soft line break*, a plain newline inside a paragraph, stays as `"\n"` inside `Text`. A *hard line break*, two trailing spaces or a backslash before the newline, becomes `HardBreak`.
- Backslash escapes and entity references such as `&` are decoded, so `Text` holds the characters the reader sees, not the source.
- Inline HTML such as `<b>` becomes `HtmlInline(raw)`.
- A link's target is `{ href, title }`, where `title` is `Ok(text)` or `Err(Missing)`.

8.2.1. Reference links need the whole document

A reference link such as `[the guide][guide]` points to a definition elsewhere in the document.

`Markdown.parse_inlines` sees only the fragment you give it, so it cannot resolve the reference and leaves the brackets as text. Parse the whole document with `Markdown.parse_str` instead:

```
document =
  \\See [the guide][guide] and ![logo][].
  \\
  \\[guide]: https://example.com/guide "The guide"
  \\[logo]: /logo.png
```

```
blocks = Markdown.parse_str(document)
for block in blocks {
  Stdout.line!(Str.inspect(block))?
}
```

Output:

```
Paragraph([Text("See "), Link({ label: [Text("the guide")], target: { href:
"https://example.com/guide", title: Ok("The guide") } }), Text(" and "), Image({ alt:
[Text("logo")], target: { href: "/logo.png", title: Err(Missing) } }), Text(".")])
```

Reference labels match case-insensitively using Unicode case folding, so `[Straße]` matches a definition labelled `[STRASSE]`.

8.3. Read frontmatter

Many static-site tools put metadata at the top of a Markdown file between two `---` lines. When the document's first line is exactly `---` and a later line is exactly `---`, `Markdown.parse_str` returns the lines between them as the first block, `Frontmatter(raw)`, and parses the rest as Markdown.

`Markdown.frontmatter(blocks)` returns that text, or `Err(Missing)` when the document has none. The parser does not interpret the text. Pass it to the YAML parser (see [Read YAML configuration and frontmatter](#)) to read the fields:

```
post =
  \---
  \\title: Hello
  \\draft: false
  \---
  \\# Hello
  \\
  \\First post.

main! = |_args| show!(post)

show! = |text| {
  blocks = Markdown.parse_str(text)
  match Markdown.frontmatter(blocks) {
    Ok(raw) => {
      Stdout.line!("raw: ${Str.inspect(raw)}")?
      meta = Yaml.parse_str(raw)?
      Stdout.line!("meta: ${meta.to_inspect()}")?
      # The frontmatter is the first block.
      Stdout.line!("body blocks: ${(blocks.len() - 1).to_str()}")?
    }

    Err(Missing) => Stdout.line!("no frontmatter")?
  }
  Ok({})
}
```

Output:

```
raw: "title: Hello
draft: false
"
meta: Mapping([ { key: "title", value: Text("Hello") }, { key: "draft", value: Bool(False) } ])
body blocks: 2
```

Without a closing `---` line, the opening `---` is ordinary Markdown: a thematic break, or a Setext heading underline.

8.4. Walk the tree to produce output

To produce output, write two functions: one that matches each block variant and one that matches each inline variant. Each function calls the other for nested content. This section builds a small HTML renderer and a table-of-contents extractor for the following release notes:

```
blocks = Markdown.parse_str(source)
Stdout.write!(render_blocks(blocks))?
Stdout.line!("--- contents ---")?
for line in table_of_contents(blocks) {
  Stdout.line!(line)?
}
```

WARNING

HtmlBlock and HtmlInline hold raw HTML from the source. A renderer that copies them into a page, as this sketch does, lets the Markdown's author inject scripts. Drop or sanitise them when rendering untrusted input. Escape every Text, InlineCode and Code value, and every attribute value, because the parser has already decoded entities such as < into the characters they stand for.

Escape the characters that are significant in HTML:

```
escape : Str -> Str
escape = |text| {
  text
  .replace_each("&", "&amp;")
  .replace_each("<", "&lt;")
  .replace_each(">", "&gt;")
  .replace_each("\"", "&quot;")
}
```

Render inline content:

```
render_inlines : List(Markdown.Inline) -> Str
render_inlines = |inlines| {
  var $out = ""
  for inline in inlines {
    $out = $out.concat(render_inline(inline))
  }
  $out
}

render_inline : Markdown.Inline -> Str
render_inline = |inline| {
  match inline {
    Text(text) => escape(text)
    Strong(children) => "<strong>${render_inlines(children)}</strong>"
  }
}
```

```

    Emphasis(children) => "<em>${render_inlines(children)}</em>"
    Strikethrough(children) => "<del>${render_inlines(children)}</del>"
    InlineCode(code) => "<code>${escape(code)}</code>"
    Link({ label, target }) => "<a href=\"${escape(target.href)}\" ${title_attr(target
.title)}>${render_inlines(label)}</a>"
    Image({ alt, target }) => "<img src=\"${escape(target.href)}\" alt=\"${escape
(plain_text(alt))}\" ${title_attr(target.title)} />"
    HardBreak => "<br />\n"
    # Raw HTML passes through unchanged; see the warning above.
    HtmlInline(raw) => raw
  }
}

title_attr : Try(Str, [Missing]) -> Str
title_attr = |title| {
  match title {
    Ok(text) => " title=\"${escape(text)}\""
    Err(Missing) => ""
  }
}

```

Render blocks. A tight list item is rendered without its `<p>` wrapper:

```

render_blocks : List(Markdown) -> Str
render_blocks = |blocks| {
  var $out = ""
  for block in blocks {
    $out = $out.concat(render_block(block))
  }
  $out
}

render_block : Markdown -> Str
render_block = |block| {
  match block {
    Heading({ level, content }) => {
      n = level.to_str()
      "<h${n}>${render_inlines(content)}</h${n}>\n"
    }

    Paragraph(inlines) => "<p>${render_inlines(inlines)}</p>\n"
    Blockquote(children) => "<blockquote>\n${render_blocks(children)}</blockquote>\n"
    ListBlock({ kind, loose, items }) => {
      tag =
        match kind {
          Unordered => "ul"
          Ordered(_) => "ol"
        }
      var $body = ""
      for item in items {
        # A tight list shows its paragraphs without <p> tags.

```

```

        inner =
            match item.blocks {
                [Paragraph(inlines)] if !loose => render_inlines(inlines)
                _ => render_blocks(item.blocks).trim()
            }
        $body = $body.concat("<li>${inner}</li>\n")
    }
    "<${tag}>\n${$body}</${tag}>\n"
}

Code({ pre, .. }) => "<pre><code>${escape(pre)}</code></pre>\n"
ThematicBreak => "<hr />\n"
HtmlBlock(raw) => raw
# Tables, frontmatter and the remaining variants are left out of this sketch.
_ => ""
}
}

```

A table of contents only needs headings and their plain text:

```

plain_text : List(Markdown.Inline) -> Str
plain_text = |inlines| {
    var $out = ""
    for inline in inlines {
        piece =
            match inline {
                Text(text) => text
                InlineCode(code) => code
                Strong(children) | Emphasis(children) | Strikethrough(children) =>
plain_text(children)
                Link({ label, .. }) => plain_text(label)
                Image({ alt, .. }) => plain_text(alt)
                HardBreak => " "
                HtmlInline(_) => ""
            }
        $out = $out.concat(piece)
    }
    $out
}

table_of_contents : List(Markdown) -> List(Str)
table_of_contents = |blocks| {
    var $lines = []
    for block in blocks {
        match block {
            Heading({ level, content }) => {
                indent =
                    match level {
                        One => ""
                        Two => " "
                        _ => " "
                    }
            }
        }
    }
}

```

```

    }
    $lines = $lines.append("${indent}- ${plain_text(content)}")
  }

  _ => {}
}
}
$lines
}

```

Output:

```

<h1>Release notes</h1>
<p>Version <strong>2</strong> is out. Read the <a href="/changes" title="All
changes">changelog</a>.</p>
<h2>Fixed</h2>
<ul>
<li>Faster <code>parse</code></li>
<li>Safer <b>HTML</b> & entities</li>
</ul>
<h2>Known issues</h2>
<p>None.</p>
--- contents ---
- Release notes
  - Fixed
  - Known issues

```

A complete renderer would also handle `Table`, use `Code`'s info for a `language-` class, and pass `Ordered({ start })` through as the `start` attribute.

8.5. Conformance and limitations

- Block and inline parsing follow CommonMark 0.31.2, including tabs, container continuation, lazy continuation lines, all seven kinds of HTML block, entity and numeric character references, and the delimiter-run algorithm for emphasis.
- Left- and right-flanking delimiter runs use the Unicode definitions of whitespace and punctuation. Reference labels use Unicode case folding. Both come from the `roc-lang/unicode` package, which `roc-parser` depends on.
- GFM additions follow `cmarm-gfm`: tables, task list items, strikethrough and extended autolinks. The GFM *disallowed raw HTML* (tagfilter) extension is not applied; raw HTML is returned unchanged.
- Line endings may be LF, CRLF or a lone CR. A U+0000 character becomes U+FFFD, as CommonMark requires.

- The tree is a syntax tree, not HTML. CommonMark's conformance examples are written as HTML, so a renderer you write decides details such as tight-list paragraphs and attribute escaping.
- Frontmatter is a convention outside both specifications. Only the `---` / `---` form at the very start of the document is recognised.
- Block quotes and lists nest at most 1,000 levels deep, and so do emphasis, strikethrough, links and images; deeper markers, delimiters and brackets are text. CommonMark sets no limit, so a document nested more deeply than that parses differently from the specification.

The [API reference](#) lists every type and function in the module.

8.6. Performance

`Markdown.parse_str` parses about 30 MB/s on an Apple M2, building the block tree and every inline: a typical README in about 0.15 ms and a 1 MiB document in about 32 ms. That is about 2.7 times slower than goldmark (Go) and comrak (Rust), 7 times slower than pulldown-cmark, and 10 times faster than markdown-it-py. Parsing stays linear on runs of unmatched emphasis and brackets and on deeply nested block quotes and lists. See [Performance](#) for the measurements and caveats.

Chapter 9. HTTP messages

This chapter shows how to parse raw HTTP/1.x bytes into Roc records: a request or response line, its header fields, and its body. It is a set of how-to guides for readers who know basic Roc and are handling bytes from a socket, a test fixture or a capture file. By the end you can read requests and responses, decode chunked bodies, split pipelined messages, embed the parsers in a larger grammar, and know which inputs the parser refuses and why.

The `HTTP` module follows the message syntax of RFC 9112 and the field syntax of RFC 9110. It parses messages; it does not open connections or send anything.

WARNING

The parser has no size limits. It holds the whole message in memory and will read a Content-Length of any size the input contains. Before you call it on network input, bound the number of bytes you read.

9.1. Parse a request

Parse a request with `HTTP.parse_request`. It takes the raw bytes as a `List(U8)` (use `Str.to_utf8` for a `Str`) and returns the `request` and the `rest` of the bytes after it:

```
request_text = "POST /notes HTTP/1.1\r\nHost: example.com\r\nContent-Type: text/plain\r\nContent-Length: 5\r\n\r\nhello"

{ request, rest: _ } = HTTP.parse_request(request_text.to_utf8())?
Stdout.line!("method: ${method_name(request.method)}")?
Stdout.line!("target: ${request.target}")?
Stdout.line!("version: ${request.version.major.to_str()}.${request.version.minor.to_str()}")?
Stdout.line!("body: ${Str.from_utf8(request.body) ?? "<binary>"}")?
```

Output:

```
method: POST
target: /notes
version: 1.1
body: hello
```

A request is a record with these fields:

- `method`: a `Method`, one of `Get`, `Head`, `Post`, `Put`, `Delete`, `Connect`, `Options`, `Trace` or `Patch`, or `Extension(name)` for any other method token, such as `PURGE`. Method names are case-sensitive, so `get` is `Extension("get")`.
- `target`: the request target exactly as sent, such as `/notes?id=1`.
- `version`: a `Version`, { `major`, `minor` }.
- `headers`: a `List(Header)` in the order received, where each `Header` is { `name`, `value` }.
- `body`: the body bytes as a `List(U8)`.

Turning a `Method` back into its name covers every case with one `Extension` branch:

```
## A method as it appears on the request line.
method_name : HTTP.Method -> Str
method_name = |method| {
  match method {
    Get => "GET"
    Head => "HEAD"
    Post => "POST"
    Put => "PUT"
    Delete => "DELETE"
    Connect => "CONNECT"
    Options => "OPTIONS"
    Trace => "TRACE"
    Patch => "PATCH"
    Extension(name) => name
  }
}
```

```
purge = HTTP.parse_request("PURGE /cache/a HTTP/1.1\r\nHost: example.com\r\n\r\n".to_utf8())?
Stdout.line!("method: ${Str.inspect(purge.request.method)}")?
```

Output:

```
method: Extension("PURGE")
```

9.1.1. Read header fields

Look a field up with `HTTP.header`, which compares names case-insensitively (HTTP field names are) and returns the first match, or `Err(Missing)`:

```
content_type = HTTP.header(request.headers, "content-type") ?? "none"
accept = HTTP.header(request.headers, "Accept") ?? "none"
```

```
Stdout.line!("content type: ${content_type}, accept: ${accept}")?
```

Output:

```
content type: text/plain, accept: none
```

Header names keep the case they were sent in. Values have surrounding spaces and tabs removed. A field that appears more than once appears in `headers` once per occurrence, so walk `headers` yourself to see every value.

9.2. Parse a response

Parse a response with `HTTP.parse_response`. `status_code` is a `U16` and `reason` is the reason phrase, which may be empty:

```
response_text = "HTTP/1.1 404 Not Found\r\nContent-Length: 9\r\n\r\nNot here."

{ response, rest: _ } = HTTP.parse_response(response_text.to_utf8())?
Stdout.line!("status: ${response.status_code.to_str()} ${response.reason}")?
Stdout.line!("body: ${Str.from_utf8(response.body) ?? "<binary>"}")?
```

Output:

```
status: 404 Not Found
body: Not here.
```

9.3. Understand how the body is found

The headers decide where the body ends (RFC 9112 section 6.3). The parser applies these rules in order:

1. A response with a `1xx`, `204` or `304` status has no body.
2. `Transfer-Encoding: chunked` means the body is a series of chunks, which the parser decodes into one body.
3. `Content-Length: n` means the body is exactly `n` bytes. If the input has fewer, parsing fails.
4. With neither field, a request has no body, and a response's body runs to the end of the input, because the server marks the end by closing the connection.

WARNING

A response to a `HEAD` request, and a `2xx` response to a `CONNECT` request, also has no body, but the parser cannot see the request that the response answers. If you

parse such a response as-is, rule 3 or 4 reads the following bytes as its body. Handle those responses yourself: for example, stop after the blank line that ends the header fields.

9.3.1. Decode a chunked body

Chunk extensions (`;name=value` after a chunk size) and trailer fields (fields after the final `0` chunk) are checked for valid syntax and then discarded. Only the decoded content is kept.

```
chunked_text = "HTTP/1.1 200 OK\r\nTransfer-Encoding: chunked\r\n\r\n4\r\nWiki\r\n6;note=x\r\npedia \r\n0\r\nExpires: never\r\n\r\n"

chunked = HTTP.parse_response(chunked_text.to_utf8())?
Stdout.line!("decoded body: ${Str.from_utf8(chunked.response.body) ?? "<binary>"}")?
```

Output:

```
decoded body: Wikipedia
```

9.4. Read pipelined messages

A client may send several requests on one connection without waiting for responses.

`HTTP.parse_request` consumes exactly one message and returns the bytes that follow it as `rest`; parse `rest` again for the next message. To read a whole buffer of requests at once, use `HTTP.parse_requests`, which fails unless the buffer ends exactly at the end of a request:

```
pipelined = "GET /a HTTP/1.1\r\nHost: example.com\r\n\r\nGET /b HTTP/1.1\r\nHost: example.com\r\n\r\n".to_utf8()

first = HTTP.parse_request(pipelined)?
second = HTTP.parse_request(first.rest)?
Stdout.line!("first: ${first.request.target}, second: ${second.request.target}, left over: ${second.rest.len().to_str()} bytes")?

all = HTTP.parse_requests(pipelined)?
Stdout.line!("targets: ${Str.join_with(all.map(|r| r.target), ", ")}")?
```

Output:

```
first: /a, second: /b, left over: 0 bytes
targets: /a, /b
```

A response with no framing fields reads to the end of the input, so it is always the last message in a buffer.

9.5. Use the parsers inside a larger grammar

`HTTP.request` and `HTTP.response` are the same parsers as a `Parser(Utf8.Bytes, _)`, for when a message is one part of a larger format. They leave the bytes after the message for the next parser:

```
## A capture file: a one-line label, then the raw request.
capture : Parser(Utf8.Bytes, { label : Str, request : HTTP.Request })
capture =
  Parser.const(|label| |request| { label, request })
    .keep(Parser.chomp_while(|byte| byte != '\n').map(|bytes| Str.from_utf8_lossy(
bytes)))
    .skip(Utf8.codeunit('\n'))
    .keep(HTTP.request)
```

```
saved = Utf8.parse_str(capture, "health check\nGET /health HTTP/1.1\r\nHost:
example.com\r\n\r\n")?
Stdout.line!("${saved.label}: ${saved.request.target}")?
```

Output:

```
health check: /health
```

They fail with `ParseError({ message, offset })`. The message starts with `invalid HTTP request:` or `invalid HTTP response:`, and the offset is where the problem is, counted from the start of the whole input.

9.6. Rejected messages and request smuggling

Request smuggling happens when two programs, such as a proxy and your server, disagree about where one message ends and the next begins. An attacker can then hide a second request inside the first one's body. To avoid this, the parser rejects any message whose framing it would otherwise have to guess.

IMPORTANT

A rejected message is not safe to recover from. Do not retry it with a "cleaned-up" copy, and do not keep reading the same connection: close it. If a proxy in front of your server accepts a message this parser rejects, the two disagree about framing.

The parser rejects a message that has any of the following:

- Both `Transfer-Encoding` and `Content-Length`.
- More than one `Content-Length` value, unless every value is identical, or a value that is not plain decimal digits (for example `+4`), or a value with more than 19 significant digits.
- Any transfer coding other than exactly one `chunked`, such as `chunked`, `gzip`, `gzip`, or `chunked` given twice.
- `Transfer-Encoding` in an HTTP/1.0 message.
- Whitespace between a field name and its colon, as in `Transfer-Encoding :`.
- Obsolete line folding (a field line that starts with a space or tab).
- A line ending other than CRLF: a bare CR or a bare LF anywhere in the head or in chunk framing.
- Control characters in a field value or reason phrase, or a field value that is not valid UTF-8.
- In an HTTP/1.1 request, a missing `Host` field; in any request, more than one `Host` field.
- A malformed chunk size, such as ``5 `` with a trailing space, or one with more than 15 significant hexadecimal digits.

The status line has rules of its own: the status code is exactly three digits and at least `100`, and the space before the reason phrase may be left out when the phrase is empty.

`parse_request` and `parse_response` fail with `InvalidHttp(Error)`, where an `Error` is `{ offset, message }`: the message says which rule was broken and the offset is the byte where the problem is. A framing error points at the start of the field line responsible; a missing `Host` field is reported at offset 0, the start of the message.

```
smuggled = "POST / HTTP/1.1\r\nHost: example.com\r\nContent-Length: 3\r\nTransfer-Encoding:
chunked\r\n\r\n0\r\n\r\n"

match HTTP.parse_request(smuggled.to_utf8()) {
  Ok(_) => Stdout.line!("accepted")?
  Err(InvalidHttp({ offset, message })) => Stdout.line!("rejected at byte ${offset.to_str
  ()}: ${message}")?
}
```

Output:

```
rejected at byte 55: both Transfer-Encoding and Content-Length
```

The `fuzz/http-smuggling.roc` fuzz target generates messages built from these constructs and checks that each one is rejected; see [the fuzzing guide](#).

9.7. Out of scope

- Bodies of responses to `HEAD` and `CONNECT`, which need the request; see [Understand how the body is found](#).
- Chunk extensions and trailer fields, which are validated and dropped.
- Content codings such as `gzip`, and any `Transfer-Encoding` other than `chunked`, which is rejected rather than decoded.
- HTTP/2 and HTTP/3, which are binary protocols.
- Size limits and timeouts, which the caller must enforce.

The [API reference](#) lists the `HTTP` types and parsers.

9.8. Performance

A typical request or response head parses in 1.5–2 μ s on an Apple M2, as fast as Go's `net/http` and about 3 times slower than Rust's `httparse`, which checks less (see [What the comparison does and does not show](#)). Line ends, tokens and field values are checked 16 bytes at a time with SIMD. A `Content-Length` body is returned without copying, so body size barely affects the time; a chunked body is assembled once, in linear time even with thousands of one-byte chunks (20,000 of them in 0.7 ms). See [Performance](#) for the measurements.

Chapter 10. Combinators by task

This chapter is for readers who have finished [Your first parser](#) and are writing a parser of their own. It groups the combinators in the `Parser` and `Utf8` modules by the job they do, and explains the three rules that most often surprise people: how alternatives backtrack, when repetition stops, and why parsers read bytes rather than characters. The [API reference](#) has the exact signature of every function.

The examples come from `combinators-behaviour.roc`; every `expect` in them passes.

10.1. Running a parser

Function	Use it to
<code>Utf8.parse_str(p, str)</code>	Parse a whole <code>Str</code> . Returns <code>Ok(value)</code> or <code>Err(ParseError({ message, offset })),</code> where <code>offset</code> is the byte offset of the furthest failure. Input left over is a failure too.
<code>Utf8.parse_bytes(p, bytes)</code>	The same for a <code>List(U8)</code> .
<code>Utf8.parse_str_partial(p, str),</code> <code>Utf8.parse_bytes_partial(p, bytes)</code>	Parse the start of the input and return <code>{ value, rest },</code> where <code>rest</code> is what is left. Use these to read one message from a stream that holds several.
<code>Parser.parse(p, input), Parser.run(p, input)</code>	Run a parser whose input is not text, such as the <code>CSV.Record</code> that record parsers read. Call them as methods: <code>p.parse(input)</code> .

10.2. Reading text

All of these read from `Utf8.Bytes`, which is `List(U8)`.

Parser	Reads
<code>Utf8.codeunit(b)</code>	Exactly the byte <code>b</code> , such as <code>Utf8.codeunit('=')</code> .

Parser	Reads
<code>Utf8.codeunit_satisfies(pred)</code>	One byte that <code>pred</code> accepts.
<code>Utf8.any_codeunit</code>	Any one byte.
<code>Utf8.string(s)</code> , <code>Utf8.utf8(bytes)</code>	Exactly the given text, returned as a <code>Str</code> or as bytes.
<code>Utf8.digit</code> , <code>Utf8.digits</code>	One ASCII digit, or a run of them, as a <code>U64</code> . <code>digits</code> fails on a number too large for a <code>U64</code> .
<code>Parser.chomp_while(pred)</code>	Bytes while <code>pred</code> accepts them. Always succeeds, possibly reading nothing.
<code>Parser.chomp_until(b)</code>	Bytes up to, but not including, the byte <code>b</code> . Fails if <code>b</code> never appears.
<code>Utf8.rest_str</code> , <code>Utf8.rest</code>	All remaining input, as a <code>Str</code> (failing if it is not valid UTF-8) or as bytes.

10.3. Sequencing and transforming

Combinator	Does
<code>Parser.const(v)</code>	Reads nothing and produces <code>v</code> . Start a sequence with <code>`Parser.const(</code>
<code>a</code>	
<code>b</code>	<code>...)`</code> .
<code>p.keep(q)</code>	Runs <code>q</code> after <code>p</code> and passes its result to the function <code>p</code> produced.
<code>p.skip(q)</code>	Runs <code>q</code> after <code>p</code> and discards <code>q</code> 's result.
<code>p.map(f)</code> , <code>Parser.map2</code> , <code>Parser.map3</code>	Transforms one, two or three results with an ordinary function.
<code>p.between(open, close)</code>	Reads <code>open</code> , <code>p</code> , <code>close</code> and keeps only <code>p</code> 's result.
<code>p.ignore()</code>	Keeps the input <code>p</code> reads but replaces its result with <code>{}</code> .

Combinator	Does
<code>p.flatten()</code>	Turns a parser producing <code>Try(a, Str)</code> into one producing <code>a</code> , where <code>Err(msg)</code> becomes a parse failure with that message.

`flatten` is how you validate a value while parsing. Here a number above 65535 is rejected with a message of your choosing:

```
# Validate a value and turn a rejection into a parse failure.
port : Parser(Utf8.Bytes, U64)
port =
  Utf8.digits
    .map(
      |n| if n <= 65535 {
        Ok(n)
      } else {
        Err("port ${n.to_str()} is out of range")
      },
    )
    .flatten()

expect Utf8.parse_str(port, "8080") == Ok(8080)
expect Utf8.parse_str(port, "70000") == Err(ParseError({ message: "port 70000 is out of
range", offset: 0 })))
```

10.4. Alternatives and optional parts

Combinator	Does
<code>Parser.one_of(list)</code>	Tries each parser in order; the first success wins. On failure, reports the last parser's message.
<code>Parser.one_of(list), Parser.alt(p, q)</code>	The same for any input type. On failure, joins all the messages with "or".
<code>Parser.maybe(p)</code>	Produces <code>Ok(value)</code> if <code>p</code> succeeds, or <code>Err(Missing)</code> without reading anything if it fails.
<code>Parser.fail(message)</code>	Always fails with <code>message</code> . Useful as the last alternative.

```
# An optional leading sign, then digits.
signed : Parser(Utf8.Bytes, I64)
signed =
  Parser.const(
```

```

    |sign| |n| {
      match sign {
        Ok(_) => -(n.to_i64_wrap())
        Err(Missing) => n.to_i64_wrap()
      }
    },
  )
  .keep(Parser.maybe(Utf8.codeunit('-')))
  .keep(Utf8.digits)

expect Utf8.parse_str(signed, "-42") == Ok(-42)
expect Utf8.parse_str(signed, "42") == Ok(42)

```

10.4.1. Alternatives always backtrack

When an alternative fails, the next one starts from the same position the first one started from, however much input the failed one had examined. There is no "commit" point, so you never need a `try` wrapper as in some other parser libraries:

```

# Both alternatives start with "ab". When the first fails at "d", the
# second starts again from the beginning of the same input.
abc_or_abd : Parser(Utf8.Bytes, Str)
abc_or_abd = Parser.one_of([Utf8.string("abc"), Utf8.string("abd")])

expect Utf8.parse_str(abc_or_abd, "abd") == Ok("abd")

```

The cost is that the first success wins, even when a later alternative would have read more. Put longer or more specific alternatives first:

```

# The first alternative that succeeds wins, even if a later one would
# consume more input. Put the longer keyword first.
keyword_short_first : Parser(Utf8.Bytes, Str)
keyword_short_first = Parser.one_of([Utf8.string("in"), Utf8.string("int")])

keyword_long_first : Parser(Utf8.Bytes, Str)
keyword_long_first = Parser.one_of([Utf8.string("int"), Utf8.string("in")])

expect Utf8.parse_str_partial(keyword_short_first, "int").map_ok(|r| r.rest) == Ok("t")
expect Utf8.parse_str(keyword_long_first, "int") == Ok("int")

```

With the short keyword first, "int" reads as "in" and leaves `t` behind.

Backtracking also means a failure deep inside one alternative is replaced by the failures of the ones after it. If error messages matter, keep the alternatives distinct from their first byte, so that only one of them gets far.

10.5. Repetition

Combinator	Produces
<code>Parser.many(p)</code>	Zero or more results of <code>p</code> .
<code>Parser.one_or_more(p)</code>	One or more results; fails if the first <code>p</code> fails.
<code>p.sep_by(sep)</code>	Zero or more results of <code>p</code> separated by <code>sep</code> . The separators are dropped.
<code>p.sep_by_one_or_more(sep)</code>	The same, requiring at least one.

These combinators require the input type to support `==`, which every input type in this package does. They use it to enforce the next rule.

10.5.1. Repetition stops when an element makes no progress

Repetition ends at the first element that fails, or that succeeds without reading any input. Without the second condition, repeating a parser that can succeed on empty input, such as `chomp_while` or `maybe(p)`, would loop forever:

```
# chomp_while always succeeds, possibly consuming nothing. `many` stops at
# the first element that makes no progress instead of looping forever.
runs : Parser(Utf8.Bytes, List(List(U8)))
runs = Parser.many(Parser.chomp_while(|b| b == 'a'))

expect Utf8.parse_str_partial(runs, "aab").map_ok(|r| r.rest) == Ok("b")
```

10.5.2. A failed element ends the repetition, not the parse

When an element fails part-way through, repetition ends before that element and leaves its input unread. The repetition itself succeeds:

```
# A repeated element that fails part-way is not an error for `many`:
# repetition ends before that element, and its input is left unconsumed.
pair : Parser(Utf8.Bytes, U64)
pair = Utf8.digits.skip(Utf8.codeunit(';'))

expect Utf8.parse_str_partial(Parser.many(pair), "1;2;3x").map_ok(|r| r.rest) == Ok("3x")
```

The failure of the element that stopped the repetition is kept, though. So when you parse a whole input with `Utf8.parse_str`, a bad element in the middle of a list is reported by its own failure message and offset. [Report parse errors](#) shows how to turn that into a useful message.

10.6. Recursive structures

A parser for a nested format refers to itself. Wrap the self-reference in `Parser.lazy` so that it is built only when it is needed:

```
# A tree is a number or a bracketed, comma-separated list of trees: [1,[2,3]]
Tree := [Leaf(U64), Node(List(Tree))]

tree : Parser(Utf8.Bytes, Tree)
tree =
  Parser.one_of([
    Utf8.digits.map(|n| Leaf(n)),
    Parser.lazy(|_| tree)
      .sep_by(Utf8.codeunit(', '))
      .between(Utf8.codeunit('['), Utf8.codeunit(']'))
      .map(|children| Node(children)),
  ])

leaves : Tree -> U64
leaves = |t| {
  match t {
    Leaf(_) => 1
    Node(children) => children.fold(0, |sum, child| sum + leaves(child))
  }
}

expect Utf8.parse_str(tree, "[1,[2,3],[]]").map_ok(leaves) == Ok(3)
```

Two parsers that refer to each other at the top level currently need a lower-level workaround because of a compiler limitation, [roc-lang/roc#10098](https://github.com/roc-lang/roc/issues/10098). Keep the recursion inside one parser where you can, as here.

10.7. Bytes, not characters

The `Utf8` parsers read UTF-8 code units, which are bytes. An ASCII character is one byte, but other characters take two to four. `é`, for example, is the two bytes `0xC3 0xA9`:

```
# Parsers read UTF-8 code units (bytes), not characters. "é" is two bytes.
expect Utf8.parse_str_partial(Utf8.any_codeunit, "é").map_ok(|r| r.value) == Ok(0xC3)
expect Utf8.parse_str(Utf8.string("é"), "é") == Ok("é")
```

This has three consequences:

- `Utf8.string` and `Utf8.codeunit` work for any text, because they compare exact bytes.
- `codeunit_satisfies` and `chomp_while` see one byte at a time. A predicate such as "is a letter" written for ASCII never matches the bytes of `é`. A predicate such as `|b| b != ' '` is safe,

because no byte of a multi-byte character is below 128.

- `Str.from_utf8_lossy` replaces bytes that are not valid UTF-8 with U+FFFD. That cannot happen for a slice of a valid `Str` cut at ASCII bytes, as in this manual's examples. If your parser can stop in the middle of a character and you need to know, convert with `Str.from_utf8` and handle its error instead.

10.8. Parsers over other input

`Parser(input, a)` works for any input type, not only bytes. The `CSV` module's record parsers read a `CSV.Record`, a list of fields, and `CSV.field(p)` runs a byte parser `p` on one field. To write a primitive parser for your own input type, use `Parser.custom` with a function from input to `Ok({ value, rest })` or `Err(ParseError({ message, offset }))`.

Chapter 11. Parse a format of your own

This guide shows how to turn a line-based text format into typed Roc values, with tests for each part and error messages that name the line at fault. It assumes you know the combinators from [Your first parser](#).

The running example is an application log with one entry per line:

```
08:00 INFO started
12:30 WARN disk almost full
```

The finished program is [custom-format-log.roc](#).

11.1. 1. Write down the result types

Decide what a caller should get back before writing any parser. Use tags for fixed sets of words and records for groups of fields:

```
Level : [Info, Warn, Error]

Time : { hour : U64, minute : U64 }

Entry : { time : Time, level : Level, message : Str }
```

11.2. 2. Write one parser per part

Give each part of the line its own parser, named after what it reads. Small parsers are easier to test, and their names make the larger parser read like the format's description:

```
two_digits : Parser(Utf8.Bytes, U64)
two_digits =
  Parser.const(|tens| |ones| tens * 10 + ones)
    .keep(Utf8.digit)
    .keep(Utf8.digit)

time : Parser(Utf8.Bytes, Time)
time =
```

```

    Parser.const(|hour| |minute| { hour, minute })
      .keep(two_digits)
      .skip(Utf8.codeunit(':'))
      .keep(two_digits)
      .map(
        |t| if t.hour < 24 and t.minute < 60 {
          Ok(t)
        } else {
          Err("no such time")
        },
      )
      .flatten()

level : Parser(Utf8.Bytes, Level)
level =
  Parser.one_of([
    Parser.const(Info).skip(Utf8.string("INFO")),
    Parser.const(Warn).skip(Utf8.string("WARN")),
    Parser.const(Error).skip(Utf8.string("ERROR")),
  ])

message : Parser(Utf8.Bytes, Str)
message = Utf8.rest_str

```

Notes on the choices made here:

- `time` checks its value with `.map(...).flatten()`. A rule that the syntax cannot express, such as "hours are below 24", belongs here, so that bad values never reach the rest of your program.
- `level` lists whole words. If two words shared a prefix, the longer one would have to come first; see [Combinators by task](#).
- `message` takes the rest of the input. That is correct only because step 4 hands the parser one line at a time.

11.3. 3. Combine the parts

```

entry : Parser(Utf8.Bytes, Entry)
entry =
  Parser.const(|t| |l| |m| { time: t, level: l, message: m })
    .keep(time)
    .skip(Utf8.codeunit(' '))
    .keep(level)
    .skip(Utf8.codeunit(' '))
    .keep(message)

```

11.4. 4. Test each parser

Put `expect` tests next to the parsers, covering at least one accepted and one rejected input for each:

```
expect Utf8.parse_str(time, "09:05") == Ok({ hour: 9, minute: 5 })
expect Utf8.parse_str(time, "24:00") == Err(ParseError({ message: "no such time", offset: 0
}))
expect Utf8.parse_str(level, "WARN") == Ok(Warn)

expect
  Utf8.parse_str(entry, "12:30 WARN disk almost full")
  == Ok({ time: { hour: 12, minute: 30 }, level: Warn, message: "disk almost full" })

expect Utf8.parse_str(entry, "12:30 DEBUG hello").is_err()
```

Run them:

```
roc test custom-format-log.roc
```

The compiler reports how many tests passed. A failing test prints the expression and the values on each side of `==`.

11.5. 5. Split the input into records yourself

For a format with one record per line, split the text into lines and parse each line separately, rather than describing the whole file with `sep_by`. You get three benefits:

- You know the line number of every failure.
- The failure is reported against the line, not as a byte offset into the whole file.
- Blank lines and a final line break are easy to allow.

```
Problem : { line : U64, reason : Str }

parse_log : Str -> Try(List(Entry), Problem)
parse_log = |text| {
  var $entries = []
  var $number = 0
  for line in text.split_on("\n") {
    $number = $number + 1
    match Utf8.parse_str(entry, line) {
      _ if line.is_empty() => {}
      Ok(e) => {
        $entries = $entries.append(e)
      }
    }
  }
}
```

```

        Err(ParseError({ message: reason, offset: _ })) => return Err({ line: $number,
reason })
    }
}
Ok($entries)
}

```

This splits on `\n` only. If your files can come from Windows, also strip a trailing `\r` from each line.

11.6. 6. Use it

```

level_name : Level -> Str
level_name = |l| {
    match l {
        Info => "info"
        Warn => "warning"
        Error => "error"
    }
}

summary : Str -> Str
summary = |text| {
    match parse_log(text) {
        Ok(entries) =>
            Str.join_with(entries.map(|e| "${level_name(e.level)} at ${e.time.hour.to_str()
}h: ${e.message}"), "\n")
        Err({ line, reason }) => "line ${line.to_str()}: ${reason}"
    }
}

main! = |_args| {
    Stdout.line!(summary("08:00 INFO started\n12:30 WARN disk almost full\n"))?
    Stdout.line!(summary("08:00 INFO started\n\n25:00 ERROR late\n"))
}

```

Run the program:

```
roc custom-format-log.roc
```

Output:

```

info at 8h: started
warning at 12h: disk almost full
line 3: no such time

```

The second log has a blank second line, which is skipped, and a time that does not exist on the

third line, which is reported with that line number.

11.7. When the format is not line-based

If records can span lines, as in a nested or bracketed format, you cannot split the input first.

Describe the whole input with one parser instead:

- Use `sep_by` or `many` for lists, and `Parser.lazy` for nesting, as in [Combinators by task](#).
- Use `Utf8.parse_str_partial` to read one record at a time when you need to know where each record starts.
- Expect a bad record in the middle to be reported by its own failure, with a `byte offset` that [Report parse errors](#) turns into a line and column.

Chapter 12. Report parse errors

This guide shows what each module returns when its input is invalid, and how to turn that into a message a person can act on. The format parsers report invalid input as an `Err` value rather than crashing, and the repository's fuzz tests check that they do.

The examples come from `errors-report.roc`, whose complete output is checked in as `errors-report.expected`.

12.1. Summary

Every format type module reports invalid input with one tag named after the format, carrying a record with a `message` and the most precise location the format has. Combinator runners report the furthest failure.

Function	Error	What it tells you
<code>Yaml.parse_str</code> , <code>Yaml.decode</code>	<code>InvalidYaml({ line, column, message })</code>	Where the problem is and what it is. Columns count bytes.
<code>Xml.parse_str</code>	<code>InvalidXml({ line, column, message })</code>	Where the problem is and what it is. Columns count bytes.
<code>CSV.parse</code> , <code>CSV.parse_with</code> , <code>CSV.parse_records</code>	<code>InvalidCsv({ record, field, line, column, message })</code>	The record and field that failed, where they start in the text, and why: bad CSV syntax, a field your parser rejected, or a row of the wrong width.
<code>CSV.parse</code> , <code>Yaml.decode</code>	<code>MissingRequiredField(name)</code>	A record field that is not a <code>Try(_, [Missing])</code> had no column or key.
<code>HTTP.parse_request</code> , <code>HTTP.parse_response</code> , <code>HTTP.parse_requests</code>	<code>InvalidHttp({ offset, message })</code>	Why the message was rejected, and the byte offset of the problem.

Function	Error	What it tells you
<code>Markdown.parse_str</code>	None	Every input is a Markdown document, so parsing cannot fail.
Your own parsers, run with <code>Utf8.parse_str</code> or <code>Parser.parse</code>	<code>ParseError({ message, offset })</code>	The furthest failure, or <code>unexpected input</code> where input was left over, with its byte offset.

All of these are open tag unions, so `?` passes them through a function such as `main!` that returns other errors too, without `map_err`.

12.2. YAML and XML: show the location

Both report one-based line and column numbers. Print them in the `file:line:column: message` form that editors and terminals recognise, so a user can jump straight to the problem:

```

yaml_message : Str, Str -> Str
yaml_message = |file_name, source| {
  match Yaml.parse_str(source) {
    Ok(_) => "${file_name}: ok"
    Err(InvalidYaml({ line, column, message })) =>
      "${file_name}:${line.to_str()}:${column.to_str()}: ${message}"
  }
}

xml_message : Str, Str -> Str
xml_message = |file_name, source| {
  match Xml.parse_str(source) {
    Ok(_) => "${file_name}: ok"
    Err(InvalidXml({ line, column, message })) =>
      "${file_name}:${line.to_str()}:${column.to_str()}: ${message}"
  }
}

```

For the inputs `"name: demo\nport: [8080\n"` and `"<feed>\n <entry></feed>"` these print:

```

config.yaml:2:7: unterminated flow collection
feed.xml:2:10: end tag </feed> does not match start tag <entry>

```

YAML input outside the supported subset, such as an anchor or a tag, is reported the same way, with a message that names the feature. Tell users which subset you accept so that the message makes sense to them.

12.3. CSV: report the record and field

Every CSV failure is `InvalidCsv` with the same record, whether the text is not valid CSV, a field does not match your parser, or a row has the wrong number of fields. The `record` and `field` numbers are one-based and suit a spreadsheet user; `line` and `column` point into the text, where the field starts:

```
Row : { name : Str, count : U64 }

row : Parser(CSV.Record, Row)
row =
  CSV.record(|name| |count| { name, count })
    .keep(CSV.field(CSV.string))
    .keep(CSV.field(CSV.u64))

csv_message : Str -> Str
csv_message = |source| {
  match CSV.parse_with(row, source) {
    Ok(rows) => "${rows.len().to_str()} rows"
    Err(InvalidCsv({ line, column, record, field, message })) =>
      "line ${line.to_str()}, column ${column.to_str()} (record ${record.to_str()},
field ${field.to_str()}: ${message}"
  }
}
```

For an unterminated quote (`pears,"2`), a non-number in a number column (`pears,many`), and a row with an extra column (`apples,3,red`), these print:

```
line 2, column 7 (record 2, field 2): unterminated quoted field
line 2, column 7 (record 2, field 2): expected a U64, found `many`
line 1, column 10 (record 1, field 3): the record has 3 fields, but the parser read only 2
```

The messages quote a short excerpt of the field, so a huge or binary field cannot flood a log.

`CSV.parse` reports the same `InvalidCsv` errors, and adds `MissingRequiredField(name)` when the header has no column for a required record field (see [Read CSV data](#)).

12.4. HTTP: reject the message

The HTTP parsers reject anything ambiguous, because a server and a proxy that disagree about where a message ends can be exploited. Treat every `InvalidHttp` error as a `400 Bad Request` and close the connection; do not try to repair the message.

```
http_message : Str -> Str
http_message = |source| {
  match HTTP.parse_request(source.to_utf8()) {
    Ok({ request, rest: _ }) => "request for ${request.target}"
```

```
Err(InvalidHttp({ message, offset })) => "400 Bad Request (byte ${offset.to_str()}):
${message}"
    }
}
```

For an HTTP/1.1 request without a `Host` field this prints:

```
400 Bad Request (byte 0): an HTTP/1.1 request needs a Host field
```

The message is meant for your logs. It may quote the client's input, so think before sending it back in a response.

`HTTP.parse_request` returns the bytes after the message as `rest`, which is the next pipelined message, if any. `HTTP.parse_requests` reads a whole pipeline and fails if it does not end exactly at the end of a message.

12.5. Your own parsers: say where and what

`Utf8.parse_str` returns the furthest failure: the one that read furthest into the input, even when a repetition stopped at it or an alternative recovered from it (see [Combinators by task](#)). Its `offset` is a byte offset. Three habits give users better messages:

- Validate values with `.map(...).flatten()` and a message written for the user, such as "no such time", instead of relying on the generic failure from a low-level parser.
- For line-based formats, parse line by line and report the line number, as [Parse a format of your own](#) does.
- For other formats, turn the `offset` into a position: count the line breaks before it to get a line number, and show the input from there as context.

Keep the raw failure message for logs and tests. It describes the parser, not the user's input, so it is rarely the best thing to show a user on its own.

Reference

Chapter 13. Modules

This chapter is a map of the package for readers who know some Roc and want to find the right module and entry point for an input format. It says what each module is for, which types and functions you start from, and how the modules fit together. The [API reference](#) has the full signature and documentation of every type and function, generated from the source.

13.1. The modules at a glance

The package exposes seven modules. Two are general-purpose building blocks; the other five are ready-made parsers for one format each.

Module	Use it to	Start from
<code>Parser</code>	Combine small parsers into larger ones, for any input type	<code>Parser.const</code> , <code>keep</code> , <code>skip</code> , <code>map</code> , <code>one_of</code> , <code>many</code> , <code>sep_by</code>
<code>Utf8</code>	Parse text: match bytes, strings and digits, and run a parser on a <code>Str</code>	<code>Utf8.parse_str</code> , <code>string</code> , <code>codeunit</code> , <code>digits</code>
<code>CSV</code>	Read comma-separated values, raw or decoded into your own record type	<code>CSV.parse</code> , <code>CSV.parse_with</code> , <code>CSV.parse_records</code>
<code>HTTP</code>	Read one HTTP/1.1 request or response message, including its body	<code>HTTP.parse_request</code> , <code>HTTP.parse_response</code> , <code>HTTP.header</code>
<code>Markdown</code>	Turn CommonMark and GitHub Flavored Markdown into a syntax tree	<code>Markdown.parse_str</code> , <code>Markdown.parse_inlines</code>
<code>Xml</code>	Check that an XML document is well-formed and read it as a tree	<code>Xml.parse_str</code> , <code>Xml.Node.attribute</code> , <code>Xml.Node.children_named</code>
<code>Yaml</code>	Read a YAML configuration file or Markdown frontmatter	<code>Yaml.decode</code> , <code>Yaml.parse_str</code> , <code>Yaml.get_path</code>

13.2. How the modules fit together

A `Parser(input, a)` is a value that describes how to read an `a` from the start of an `input`. Running it returns the value and the input it did not consume, or a `ParseError` with a message and an offset. `Parser` provides the combinators that build parsers from parsers; `Utf8` provides the parsers that read text, with `Utf8.Bytes` (a `List(U8)`) as the input type.

Each format type module follows the same conventions:

- `parse_str` (or, for CSV, `parse`) reads a whole document and returns `Ok(value)` or an error tag named after the format: `InvalidCsv`, `InvalidHttp`, `InvalidXml` or `InvalidYaml`. The tag carries a record with a `message` and the most precise location the format has. Markdown has no syntax errors, so `Markdown.parse_str` returns the tree directly.
- `parser` (and, for HTTP, `request` and `response`) is the same parser as a `Parser(Utf8.Bytes, ...)` value, for embedding in a larger parser. Its failures are `ParseError({ message, offset })`.
- `CSV.parse` and `Yaml.decode` decode straight into your own record type through the static dispatch method `parser_for`; the compiler infers the type from how you use the result.
- Optional values are `Try(a, [Missing])`.

The following program uses one entry point from each module. It is `docs/examples/reference-modules.roc` in the repository, and CI runs it.

```
# Parser and Utf8: build a parser from combinators, run it on a Str.
pair : Parser(Utf8.Bytes, (U64, U64))
pair = Parser.const(|a| |b| (a, b)).keep(Utf8.digits).skip(Utf8.codeunit(',')).keep(Utf8.digits)

# CSV: decode each row into a record type the compiler infers from use.
people : Str -> Try(List({ name : Str, age : U64 }), [InvalidCsv(CSV.Error),
MissingRequiredField(Str)])
people = |text| CSV.parse(text)

# HTTP: parse one message; bytes after it are left for the next message.
http_target : Str -> Str
http_target = |text| {
  match HTTP.parse_request(text.to_utf8()) {
    Ok({ request, rest: _ }) => request.target
    Err(InvalidHttp(e)) => "invalid request at byte ${e.offset.to_str()}"
  }
}

# Xml and Yaml: whole-document functions that report line and column.
xml_root : Str -> Str
xml_root = |text| {
  match Xml.parse_str(text) {
    Ok(doc) => Str.inspect(doc.root)
    Err(InvalidXml(e)) => "${e.line.to_str()}:${e.column.to_str()}: ${e.message}"
  }
}
```

```

    }
  }

  yaml_value : Str -> Str
  yaml_value = |text| {
    match Yaml.parse_str(text) {
      Ok(value) => value.to_inspect()
      Err(InvalidYaml(e)) => "${e.line.to_str()}:${e.column.to_str()}: ${e.message}"
    }
  }
}

# Yaml can also decode straight into a record.
config : Str -> Try({ title : Str, draft : Bool }, [InvalidYaml(Yaml.Error),
MissingRequiredField(Str)])
config = |text| Yaml.decode(text)

# Markdown: parsing never fails, so `parse_str` returns the blocks directly.
markdown_blocks : Str -> Str
markdown_blocks = |text| Str.join_with(Markdown.parse_str(text).map(Str.inspect), "\n")

```

Running it prints one line per call (the Markdown result prints two lines, one per block):

```

Ok((3, 4))
Ok([ { age: 36, name: "Ada" }, { age: 41, name: "Alan" } ])
/index.html
Element({ attributes: [{ name: "lang", value: "en" }], children: [Text("hi")], name:
"greeting" })
1:7: end tag </a> does not match start tag <b>
Mapping([ { key: "title", value: Text("Notes") }, { key: "draft", value: Bool(False) } ])
Ok({ draft: False, title: "Notes" })
Heading({ level: One, content: [Text("Title")] })
Paragraph([Text("Some "), Emphasis([Text("text")]), Text(".")])

```

13.3. Parser

`Parser` is the combinator library. Use it when you are writing a parser for a format this package does not cover, or extending one of the format parsers.

- `Parser(input, a)` is an opaque nominal type, and `ParseResult(input, a), Try({ value, rest }, [ParseError({ message, offset } )])`, is what one step of parsing returns.
- Build values with `const` and feed it parsed arguments with `keep`; read and discard syntax with `skip`.
- Choose between alternatives with `alt` and `one_of`. Each alternative starts from the same input, so a failed alternative never consumes anything.
- Repeat with `many`, `one_or_more`, `sep_by` and `sep_by_one_or_more`. Repetition stops when an element

fails or consumes nothing.

- Convert results with `map`, `map2`, `map3` and `flatten`, and choose the next parser from a value with `and_then`. `maybe` makes a parser optional, returning `Try(a, [Missing])`.
- Run a parser with `parse` (whole input) or `run` (leaving the rest). For text, prefer the `Utf8` functions below.
- `custom` turns your own function into a parser when no combinator fits; `lazy` defers building a parser for recursive grammars.

13.4. Utf8

`Utf8` holds the text parsers and the functions that run a parser on text.

- `Utf8.parse_str` runs a parser on a whole `Str` and reports leftover text as a `ParseError`. `parse_str_partial`, `parse_bytes` and `parse_bytes_partial` cover partial input and byte input.
- `codeunit`, `codeunit_satisfies`, `any_codeunit`, `utf8` and `string` match bytes and literal text. `digit` and `digits` read unsigned decimal numbers.
- `rest` and `rest_str` consume the rest of the input.

The parsers work on bytes, not characters: `codeunit('é')` is not possible, because `é` is two bytes in UTF-8. Use `string("é")` instead.

13.5. CSV

`CSV` reads RFC 4180 comma-separated values; the module documentation lists the exact dialect.

- `CSV.parse` decodes a file with a header row into a list of records. Each record field reads the column with the same name, a `Try(a, [Missing])` field is optional, and a required field without a column fails with `MissingRequiredField(name)`. `CSV.parse_normalized` matches headers such as `First Name` to `first_name`, and `CSV.parse_headerless` decodes rows into tuples.
- `CSV.parse_with` matches columns by position instead, with a record parser you build from `CSV.record`, `CSV.field` and `.keep(...)`. Field parsers `CSV.string`, `CSV.u64` and `CSV.f64` are provided, and any `Parser(Utf8.Bytes, a)` works.
- `CSV.parse_records` returns the raw records, each a `CSV.Record` (a `List(List(U8))`), when rows differ in shape or you want to inspect them first. `CSV.split_header` separates the header row, and `CSV.decode` runs a record parser over them later.

Every failure is `InvalidCsv` with the record, field, line and column.

13.6. HTTP

HTTP reads one HTTP/1.1 (or HTTP/1.0) message: start line, header fields and body.

- `HTTP.parse_request` and `HTTP.parse_response` take bytes and return the message and the `rest`, which is the next pipelined message, if any. `HTTP.parse_requests` reads a whole pipeline. Failures are `InvalidHttp({ offset, message })`.
- `Request` and `Response` are records. `Header` is `{ name, value }`, and `HTTP.header(headers, name)` looks a field up case-insensitively. `Method` has a tag for each standard method and `Extension(name)` for any other token; `Version` is `{ major, minor }`.
- `HTTP.request` and `HTTP.response` are the same parsers for embedding.

The parser decodes chunked bodies and rejects any message whose length is ambiguous, which is what protects a proxy from request smuggling. It does not open connections, send messages, or know which request a response answers; see [Conformance](#) for what that means for HEAD and CONNECT.

13.7. Markdown

`Markdown` turns CommonMark 0.31.2 text, with the GitHub Flavored Markdown tables, task lists, strikethrough and extended autolinks, into a tree of `Markdown` blocks and `Markdown.Inline` nodes.

- `Markdown.parse_str` parses a whole document into `List(Markdown)`. `Markdown` has no syntax errors, so it accepts every input.
- `Markdown.parse_inlines` parses inline content only, such as a single line of text.
- A leading block between `---` lines becomes a `Frontmatter(text)` block; `Markdown.frontmatter(blocks)` returns its text, which you can pass to `Yaml.decode` or `Yaml.parse_str`.
- `Markdown.parser` and `Markdown.inline_parser` are the same parsers for embedding.
- `Str.inspect` prints a tree for tests and debugging, and every tree type has `is_eq` and `to_hash`.

The module does not render HTML; walk the tree to produce your own output. Raw HTML and link destinations pass through unfiltered, so sanitize them before you render untrusted input (see [Markdown](#)).

13.8. Xml

`Xml` checks XML 1.0 well-formedness and returns the document as a tree of `Xml.Node` values (`Element({ name, attributes, children })` and `Text`).

- `Xml.parse_str` returns an `Xml` record with the declaration : `Try(Declaration, [Missing])` and the root element, or `InvalidXml({ line, column, message })`.
- The node methods `name`, `attribute`, `children_named` and `text` read the tree: `root.attribute("href")` is `Try(Str, [Missing])`.
- `Xml.parser` is the same parser for use inside a larger parser.

Documents with a `<!DOCTYPE>` declaration are rejected, namespaces are not interpreted, and comments and processing instructions are checked and then dropped.

13.9. Yaml

`Yaml` reads a practical subset of YAML 1.2 aimed at configuration files and Markdown frontmatter.

- `Yaml.decode` reads a document straight into your own record, list, dict or tuple type. `Try(a, [Missing])` fields are optional, and a missing required key fails with `MissingRequiredField(name)`. `Yaml.decoder` takes options for kebab-case or camelCase keys and for rejecting unknown keys.
- `Yaml.parse_str` returns a `Yaml` tree (`Null`, `Bool`, `Int`, `Float`, `Text`, `Sequence` or `Mapping`). The methods `get`, `at`, `get_path`, `as_str`, `as_i64`, `as_bool` and `as_list` walk it.
- Both fail with `InvalidYaml({ line, column, message })`. `Yaml.parser` is the tree parser for embedding.

Aliases, aliases, tags, directives, complex keys, multi-line flow collections, multi-line plain or quoted scalars and multi-document streams are rejected rather than misread. Nesting is limited to 100 levels.

13.10. Choosing between modules

If your input is	Use
A small text format of your own, such as a command or a log line	<code>Parser</code> with <code>Utf8</code>
A spreadsheet export or other comma-separated data	<code>CSV</code>
Raw bytes of an HTTP/1.1 message from a socket or a capture	<code>HTTP</code>
Documentation, notes or a README	<code>Markdown</code>

If your input is	Use
A configuration file, or the frontmatter of a Markdown file	<code>Yaml</code>
An XML document without a document type declaration, such as SVG or a feed	<code>Xml</code>
JSON, TOML, or YAML that uses anchors or several documents	Another package: these modules do not support them

Conformance lists, for each format, the specification it follows, how that is checked, and the known gaps.

Chapter 14. Conformance

This chapter is for readers deciding whether a format parser is accurate enough for their input, and for contributors who need to reproduce the numbers. For each format it names the specification the parser follows, the reference implementation (the *oracle*) its results are compared with, the results last measured, and the known gaps.

14.1. How conformance is checked

Each format has a review script in `scripts/` that feeds a fixed set of inputs to a small Roc program (the *probe*) and to a pinned reference implementation, normalises both results and compares them. The inputs come from the format's official test suite where one exists, plus hand-written cases in `scripts/<format>/cases.json`.

Every case where the library and the oracle disagree on purpose is listed in a `known-failures.json` file with its reason. A review fails when a case disagrees and is not listed, or when a listed case starts to agree (the list is stale). The scripts never update the list themselves.

Where the oracle is itself wrong, the script records an *oracle disagreement* and uses the specification's expected result instead; these cases are reported but are not failures.

14.2. Results

The numbers below were measured on 1 October 2026 at commit `7afc741`, with Roc `nightly-2026-09-29-7f11a82` on macOS (Apple silicon) and Python 3.14. They will change as the parsers and test sets change; rerun the commands in [Reproducing the results](#) for current figures.

Format	Specification	Oracle	Result
CSV	RFC 4180	Python's <code>csv</code> module (standard library)	2035 of 2035 cases agree (35 fixed, 2000 random), no known failures

Format	Specification	Oracle	Result
HTTP	RFC 9112 (messages) and RFC 9110 (fields)	h11 0.16.0; llhttp's behaviour where the library is stricter than h11	134 cases: 112 agree, 22 are listed deliberate divergences
Markdown, inline	CommonMark 0.31.2 and GFM	The spec's own expected output; cmark-gfm 0.29.0.gfm.13 (through cmarkgfm 2025.10.22) for extra cases	389 of 389 applicable cases pass, no known failures
Markdown, blocks	CommonMark 0.31.2 and GFM	The spec's own expected output, compared as a tree; cmark-gfm 0.29.0.gfm.13 for extra cases	593 of 669 cases pass; 76 are listed known failures
XML	XML 1.0 (Fifth Edition), well-formedness only	expat 2.7.4 and the W3C XML Conformance Test Suite (2013-09-23)	324 of 326 cases pass; 2 are listed known failures
YAML	YAML 1.2 with the core schema	ruamel.yaml 0.18.16 (pure Python) and the yaml-test-suite (revision 6ad3d2c)	595 of 803 cases pass; 206 rejections and 2 key-typing differences, see YAML

14.3. Reproducing the results

The review scripts need Python 3, the pinned oracles, and a Roc compiler. Set `ROC` to the compiler you want to test with; CI uses the nightly named in `.roc-version` (see [Compatibility](#)). Install each oracle into its own virtual environment under `.roc-parser-tmp/`, which git ignores:

```
export ROC=/path/to/roc
python3 -m venv .roc-parser-tmp/venv-yaml
.roc-parser-tmp/venv-yaml/bin/pip install --no-deps -r scripts/yaml/requirements.txt
python3 -m venv .roc-parser-tmp/venv-http
.roc-parser-tmp/venv-http/bin/pip install --no-deps -r scripts/http/requirements.txt
python3 -m venv .roc-parser-tmp/venv-md
.roc-parser-tmp/venv-md/bin/pip install --no-deps -r scripts/markdown/requirements.txt
```

Then run one command per format. CSV and XML need only the standard library; the XML script downloads the W3C suite on first use and checks its SHA-256.

```
python3 scripts/review_csv.py
python3 scripts/review_xml.py check --baseline scripts/xml/known-failures.json
.roc-parser-tmp/venv-http/bin/python scripts/review_http.py
.roc-parser-tmp/venv-md/bin/python scripts/review_markdown.py check
.roc-parser-tmp/venv-md/bin/python scripts/review_markdown_blocks.py check --baseline
scripts/markdown/blocks/known-failures.json
.roc-parser-tmp/venv-yaml/bin/python scripts/review_yaml.py check --baseline
scripts/yaml/known-failures.json
```

Each script prints a summary, such as the following for HTTP:

```
134 cases, 22 known divergences, 0 failures, 0 crashes, 0 stale (h11 0.16.0)
```

The XML, YAML and Markdown block scripts also write a JSON report, with every case's expected and actual result, under `.roc-parser-tmp/`.

14.4. CSV

The parser follows RFC 4180 with the relaxations most CSV readers share: LF and lone CR line breaks as well as CRLF, an optional final line break, rows of different lengths, and any UTF-8 in unquoted fields. The [CSV module documentation](#) lists the dialect exactly.

`scripts/review_csv.py` compares the raw records with Python's `csv.reader` in strict mode, on the fixed cases and 2000 random inputs from a fixed seed.

Known differences, by design:

- A blank line is a record with one empty field, as RFC 4180's grammar says. Python returns an empty row; the script accounts for this.
- A byte order mark is not removed; it becomes part of the first field.
- There is no heading-row support, no other delimiter or quote character, and no whitespace trimming.

14.5. HTTP

The parser follows the message syntax of RFC 9112 and the field syntax of RFC 9110. It is a message parser: it reads bytes that are already in memory and does not manage connections.

`scripts/review_http.py` compares the verdict, start line, fields, decoded body and leftover bytes with h11. Most of the 22 listed divergences are places where RFC 9112 allows a parser to reject something and this one does, as `lhttp` does by default while `h11` accepts it: line folding, bare LF line endings, control characters in field values, `Transfer-Encoding` in an HTTP/1.0 message, and a

message with both `Transfer-Encoding` and `Content-Length`. Rejecting these is what prevents request smuggling. The others are by design:

- Methods are a closed tag union (`Method`), so methods outside it, and lowercase method names, are rejected.
- Field values are `Str`, so field values that are not valid UTF-8 are rejected.
- A `101` response the client did not ask for is returned like any other message; h1 rejects it because it tracks the connection.

Limitations:

- A response with neither `Content-Length` nor `Transfer-Encoding` takes the rest of the input as its body.
- The parser cannot see the request a response answers, so responses to `HEAD` and `2xx` responses to `CONNECT` must have their framing fields removed, or a status that implies no body, before parsing.

14.6. Markdown

The parser follows CommonMark 0.31.2 with the GitHub Flavored Markdown extensions for tables, task list items, strikethrough and extended autolinks. It also reads a frontmatter block between `---` lines at the start of a document.

Two scripts check it:

- `scripts/review_markdown.py` checks inline parsing against the spec examples (compared as HTML), the GFM extension examples, and hand-written cases compared with `cmark-gfm`'s syntax tree. Of its 399 entries it runs the 389 whose expected output consists only of paragraphs, so that block structure cannot affect the result. Its `corpus` mode compares fuzzing inputs with `cmark-gfm` and asks `markdown-it-py 4.0.0`, which implements CommonMark 0.31.2, to settle disagreements where `cmark-gfm` follows an older spec version.
- `scripts/review_markdown_blocks.py` checks the whole document, every CommonMark spec example plus extra block cases, as a tree.

All 76 known failures of the block check are differences of representation or of the oracle, not wrong structure:

- 69 cases contain a soft line break. The library keeps it as `\n` inside the `Text` node, while the comparison expects a space, which is how HTML renders it.
- 5 cases are bare URLs and email addresses that the library turns into links by the GFM extended autolink rules, which the oracle configuration does not enable.

- 2 cases (spec examples 28 and 354) are ones where the spec and cmark-gfm, which follows an older spec version, disagree.

The known-failures file does not yet give each entry its reason; the grouping above comes from comparing the expected and actual trees.

Markdown has no syntax errors, so every input produces a document.

14.7. XML

The parser checks well-formedness as defined by XML 1.0 (Fifth Edition) for documents without a document type declaration. It does not validate.

`scripts/review_xml.py` compares verdicts and trees with expat and selects the W3C conformance cases inside the supported subset: UTF-8 text, XML 1.0, no `<!DOCTYPE>`. The two known failures are:

- `decl/version-2`: expat accepts `version="2.0"`, but the specification requires `1.` followed by digits, so rejecting it is correct.
- `xmlconf/hst-lhs-007`: the input is already decoded text, so a declared encoding cannot contradict the byte order mark, and the parser does not report it.

The 11 oracle disagreements are cases where expat uses older name-character tables than the Fifth Edition; the suite's verdict is used.

Limitations: document type declarations and therefore all entities other than the five predefined ones are rejected, namespaces are not processed, the input must be a `Str` (already decoded), and comments and processing instructions are dropped from the tree.

14.8. YAML

The parser implements a subset of YAML 1.2 for configuration files and frontmatter, and resolves plain scalars with the core schema.

`scripts/review_yaml.py` runs every `yaml-test-suite` case and the hand-written cases, and compares the value with `ruamel.yaml`'s pure-Python parser. Of the 803 cases:

- 595 produce the same value or the same rejection as the oracle.
- 206 are valid YAML that the parser rejects because it uses a feature outside the subset. The largest groups are anchors, aliases and tags (47), multi-document streams (32), multi-line flow collections (29), multi-line quoted scalars (30) and complex mapping keys (19).
- 2 differ by design because mapping keys are their source text: `true`: one has the key `"true"`, not a Boolean, so `1` and `01` are different keys rather than duplicates.

Two of the 206 rejections, `limits/flow-depth-100` and `limits/flow-depth-512`, come from the 100-level nesting limit, which now also applies to flow collections; `scripts/yaml/known-failures.json` records them with that reason. The 58 oracle disagreements are cases where `ruamel.yaml` differs from the test suite's expected result; the suite wins.

Limitations: besides the features above, integers must fit in `I64`, tabs may not be used for indentation, and nesting deeper than 100 levels is rejected. The [Yaml module documentation](#) describes the subset.

14.9. Property testing

Conformance checks compare chosen inputs with an oracle. Alongside them, every parser runs under coverage-guided property tests built with `roc-fuzz`, which generate inputs, including malformed and pathological ones, and check properties such as: the parser never crashes or hangs, rendering a value and parsing it again gives the same value, and results agree with a simple model. The review scripts can also compare a fuzzing corpus with the oracles. CI runs a short campaign for every target on each pull request and a longer one every night. [Property testing](#) describes the targets and how to run them.

Chapter 15. Compatibility

This chapter is for readers choosing a release, upgrading, or matching a Roc compiler to the package. It states which compiler each release is built for, what a version number promises, where the package runs, and what changed in 2.0.0.

15.1. Roc compiler

Roc has no stable release yet, and its compiler changes the language and standard library often. Each roc-parser release is therefore built and tested with one Roc nightly:

- `.roc-version` at the repository root names the nightly for the package, its tests, the examples, the property tests, the conformance reviews and the benchmarks, for example `nightly-2026-09-29-7f11a82`.
- An automated workflow proposes an update to a newer nightly when one is published, and merges it only when the tests pass with it.

Use the nightly named by the release you depend on. A newer nightly often works, but a compiler change can break the build without any change to this package; the release notes say when a migration changed the package's API.

```
cat .roc-version
```

15.2. Version numbers

Releases follow [semantic versioning](#):

- A patch release (such as 1.0.2) fixes bugs without changing the API.
- A minor release (such as 1.2.0) adds API without removing or changing it.
- A major release (such as 2.0.0) may remove or change API, or change what an existing function accepts or returns.

For a parser, which inputs it accepts and the values it produces matter as much as its type signatures. Read the release notes for behaviour changes even when the API is unchanged.

Roc packages are fetched by URL, so a release is a bundle URL whose name is the hash of its content. Copy it from the [GitHub releases page](#); each release also publishes an SBOM and signed provenance.

15.3. Platforms

The package is plain Roc with no platform-specific code: it does no input or output, so it works with any Roc platform, on any system where that platform and the pinned compiler run. CI builds and tests it on Linux x86-64, and the conformance results in [Conformance](#) were measured on macOS with Apple silicon.

15.4. Dependencies

The package depends on one other package:

Package	Version	Used for
roc-lang/unicode	4.2.0	Unicode case folding, general categories and scalar handling in the Markdown module

`package/main.roc` names its release URL. Roc fetches it when it fetches `roc-parser`, so applications need not add it themselves.

15.5. Changes in 2.0.0 at a glance

Version 2.0.0 changes the API of every module and reworks every format parser to follow its specification, checked against a reference implementation (see [Conformance](#)). Code written for 1.x does not compile without changes, and many inputs now parse differently. The [2.0.0 release notes](#) start with an upgrade guide that maps each old name to its replacement. In summary:

Module	What changed
<code>Parser</code> , <code>Utf8</code>	The text parsers live in the <code>Utf8</code> module. Runners report the furthest failure as <code>ParseError({ message, offset })</code> , and partial runners return <code>{ value, rest }.maybe</code> returns <code>Try(a, [Missing])</code> . Several functions were renamed or removed, and <code>many</code> and <code>sep_by</code> stop when an element consumes no input instead of looping forever.

Module	What changed
CSV	<code>CSV.parse</code> decodes rows into records by header name. <code>CSV.Record</code> replaces the opaque record types, and every failure is <code>InvalidCsv({ record, field, line, column, message })</code> . Parses RFC 4180, with an optional final line break, lone CR line breaks and any UTF-8 in fields.
HTTP	<code>parse_request</code> , <code>parse_response</code> and <code>parse_requests</code> take bytes and fail with <code>InvalidHttp({ offset, message })</code> . Headers are <code>{ name, value }</code> records with a case-insensitive <code>HTTP.header</code> lookup, <code>Version</code> is now <code>{ major, minor }</code> , and <code>Method</code> gains <code>Extension(name)</code> . Parses HTTP/1.1 per RFC 9112 and rejects messages with ambiguous framing.
Markdown	<code>Markdown.parse_str</code> and <code>parse_inlines</code> return the tree directly, and the legacy single-construct parsers, the placeholder variant and the debug printers are gone. Rebuilt on the CommonMark 0.31.2 parsing strategy, with GFM tables, task lists, strikethrough and extended autolinks. Adds the roc-lang/unicode dependency.
Xml	<code>Xml.parse_str</code> reports <code>InvalidXml({ line, column, message })</code> , <code>Xml.parser</code> embeds the parser, elements are records, and optional parts of the declaration are <code>Try(_, [Missing])</code> . Checks XML 1.0 well-formedness and rejects documents with a document type declaration.
Yaml	<code>Yaml.decode</code> reads a document into a record. Text scalars are <code>Text</code> , errors are <code>InvalidYaml</code> , and helpers such as <code>get_path</code> and <code>as_str</code> walk the tree. Resolves plain scalars with the YAML 1.2 core schema and rejects unsupported features, such as anchors, instead of misreading them.

Chapter 16. Performance

This chapter is for readers deciding whether roc-parser is fast enough for their input. It compares each format parser with popular Go, Rust and Python libraries, says where roc-parser is and is not a good fit, states what is guaranteed about running time and memory, and gives tips for your own combinator parsers. [Benchmarks](#) explains how the numbers are produced and how to reproduce them.

16.1. Summary

The table shows how much faster each library parsed the same documents than roc-parser did. "6x" means the library took a sixth of the time; "0.3x" means roc-parser was about three times faster than it. Each figure is the geometric mean over the sample, small (4 KiB), medium (64 KiB) and large (1 MiB) documents of that format, with every program building an in-memory result. The HTTP row covers the three documents without a body (two sample heads and one with 64 header fields).

Measured on 1 October 2026 on an Apple M2 (8 cores, macOS 26.3), roc-parser at commit [2138e04](#) for CSV, XML and HTTP (after the SIMD and slicing work described in [How roc-parser uses SIMD and slices](#)) and [98ff21e](#) for YAML and Markdown (after the tuning described in [Why roc-parser is slower where it is](#)), built with Roc [nightly-2026-09-29-7f11a82](#) and `--opt=speed`, Rust 1.94, Go 1.26.2 and Python 3.14 (the Python ratios for CSV, XML and HTTP are scaled from the [9c7b2fd](#) run). Another process was using about three cores during the run, so read the ratios rather than the absolute rates, and treat both as rough: they move by tens of percent between machines and runs.

Format	roc-parser	Rust	Go	Python
CSV	145 MB/s	csv: 1.0x	encoding/csv: 2.3x	csv ©: 0.5x
YAML	60 MB/s	yaml-rust2: 0.9x; serde_yaml: 0.6x	yaml.v3: 0.4x	PyYAML with libyaml: 0.14x; pure Python: 0.02x
XML	120 MB/s	roxmltree: 2.0x; quick-xml: 1.4x	encoding/xml: 0.45x	ElementTree (expat): 0.3x

Format	roc-parser	Rust	Go	Python
Markdown	30 MB/s	pulldown-cmark: 7.1x; comrak: 2.7x	goldmark: 2.6x	markdown-it-py: 0.1x
HTTP/1.1 (message heads)	200–300 MB/s	httparse: 2.8x	net/http: 0.9x	h11: 0.05x

In short: roc-parser matches Rust's csv crate on the generated CSV documents and yaml-rust2 on the YAML documents, is within a factor of about two of the best Rust library for XML and three for HTTP heads, matches or beats Go's libraries for YAML, XML and HTTP, and is faster than every Python parser measured. It is about 2.7 times slower than comrak and goldmark for Markdown, and 7 times slower than pulldown-cmark. Bodies are not in the HTTP row because roc-parser returns a `Content-Length` body as a slice of the input without copying it, while the other programs copy it, so roc-parser's figures for large bodies are meaninglessly high.

The per-document tables, with peak memory and allocation counts, are in the report the benchmark workflow publishes (see [Benchmarks](#)).

16.2. When roc-parser is fast enough and when it isn't

roc-parser is a good fit when:

- **Documents are up to a few megabytes.** Configuration files, front matter, READMEs, SVG icons, CSV exports and API messages parse in microseconds to a few milliseconds, and a 1 MiB YAML, XML or Markdown document in 20, 9 or 32 ms.
- **You parse at start-up or per request.** An HTTP head parses in 1.5–2 μ s; a 4 KiB YAML configuration in under 0.1 ms.
- **You want a pure-Roc package,** typed errors with line and column, strict input checking, and no native code or FFI.
- **Your alternative is an interpreted parser:** roc-parser is 10–60 times faster than pure-Python YAML, Markdown and HTTP parsers.

Look elsewhere, or measure carefully first, when:

- **Parsing is your bottleneck** at high volume (log pipelines, Markdown rendering for a large site, a proxy parsing every request). The best Rust parsers are about 2 times faster for XML, 3 times for HTTP heads and 3–7 times for Markdown; for YAML, yaml-rust2 is about as fast.
- **Input arrives as a stream or does not fit in memory.** Every roc-parser parser needs the

complete input, and returns a tree; there is no streaming or event interface.

16.3. Complexity guarantees

The parsers are designed to run in time linear in the input size, with bounded nesting, and the fuzzing campaigns described in [Property tests and conformance reviews](#) run with a per-input time limit so that super-linear behaviour shows up as a failure. That work has found and fixed quadratic cases:

- Markdown inline parsing on runs of unmatched emphasis and brackets (commit [57137ae](#), with a dedicated `markdown-inline-pathological` fuzz target), Markdown block parsing ([64bc07a](#)) and deeply nested Markdown block quotes and lists;
- XML duplicate-attribute detection ([c8f84f2](#)), which is why `roc-parser` parses an element with 5,000 attributes in 1.2 ms while `quick-xml` and `roxmltree` take 23 ms;
- YAML block scalars, which used to re-scan the document's lines;
- YAML's nesting limit, extended to flow collections ([08cc39b](#)).

Throughput is now flat from 4 KiB to 1 MiB for every format: 51–55 MB/s for generated YAML, 120–135 MB/s for generated XML and 30–34 MB/s for Markdown. (The summary table's rates are geometric means that also include the small sample documents, so they differ slightly from these generated-document ranges.)

Limits that bound the work on hostile input:

- **YAML** rejects nesting deeper than 100 levels, block and flow combined. Each level of a flow collection re-scans its contents, so cost grows with the square of the flow depth, but the limit keeps it small: 99 nested flow sequences (200 bytes) take 39 μ s.
- **XML** parses elements with an explicit stack, so deep nesting (5,000 levels in the benchmarks) cannot overflow the call stack.
- **CSV, HTTP and Markdown** scan bytes with explicit state rather than recursion. HTTP rejects ambiguous framing; see [HTTP messages](#).
- **Markdown** finds the next non-space byte of a line by scanning, except for lines that start inside more than eight open block quotes and list items, which get a per-line index; so a long run of spaces is scanned at most eight times however deeply the containers nest.

16.4. Memory behaviour

- **The whole input is held in memory**, and the result is a tree of Roc values. Expect peak memory of roughly 15–35 times the document size for 1 MiB YAML, XML and Markdown documents (18–33 MiB measured), and about 12 times for CSV. Rust’s tree builders used 8–30 MiB (pulldown-cmark’s collected owned events, 270 MiB, are an artefact of the comparison program); Go’s and Python’s libraries were similar to roc-parser.
- **Text is shared with the input where it can be.** XML text and attribute values with nothing to unescape, YAML scalars and keys (quoted ones too, when they hold no escapes), and HTTP field values become `Str` values that point into the input’s memory rather than copies, so the input stays alive as long as any of them does. Values that need rewriting (escapes, entities, folded or chomped block scalars) are copied. A `Content-Length` body shares the input’s memory, and a chunked body is assembled once.
- **Allocation counts are roughly proportional to the number of values:** about one allocation per CSV field, 0.06 per input byte for YAML, 0.1 for XML and 0.33 for Markdown. `scripts/bench.py --allocations` reports them for every document.
- **Small programs stay small.** A Roc driver parsing an HTTP message peaks at 1.5 MiB of resident memory, against 17 MiB for the same Go program and 19 MiB for Python.

16.5. What the comparison does and does not show

Every program reads the document before starting its clock, parses it in a loop, and folds over the result so the work cannot be skipped. Read the ratios with these differences in mind:

- **Tree or stream.** roc-parser always builds a tree. roxmlltree, comrak, goldmark, yaml-rust2, serde_yaml, yaml.v3, ElementTree and PyYAML do too. quick-xml and encoding/xml are tokenizers, so their programs build an equivalent owned tree from the tokens; pulldown-cmark and markdown-it-py produce flat event and token lists, which are collected but not nested. A streaming consumer that does not keep a tree is faster still.
- **Strictness.** roc-parser’s HTTP parser enforces RFC 9112’s framing and smuggling rules (see [HTTP messages](#)), while httparse checks only the head’s syntax; roc-parser’s XML parser checks well-formedness and character validity like expat does. Several libraries accept YAML and Markdown that roc-parser rejects or vice versa; the comparison counts only documents that both accepted.
- **UTF-8.** Every program receives valid UTF-8 text. roc-parser validates UTF-8 when converting the input to `Str` (outside the timed loop in the drivers) and again for each value it returns.
- **What the result holds.** roc-parser’s YAML and Go’s yaml.v3 type scalars (integers, floats, booleans); Rust’s csv and Go’s encoding/csv produce strings like roc-parser’s raw CSV driver, which decodes every field with `CSV.string`. Typed CSV decoding with `CSV.u64` and friends costs

more.

- **Allocation.** Rust and Go reuse buffers within one parse; Roc and Python allocate each value. Peak memory includes each runtime (about 1.5 MiB for Roc and Rust, 12–20 MiB for Go and Python).
- **Python’s csv and ElementTree are C code**, and PyYAML’s libyaml loader parses in C but builds Python objects.

16.6. Why roc-parser is slower where it is

The October 2026 profiling (sampling `--opt=speed --debug` builds of the benchmark drivers, see [Benchmarks](#)) found two kinds of cost. The first were bugs in roc-parser and have been fixed: building text one byte at a time, rebuilding small lists for every keyword comparison, decoding every YAML value twice with `Str.from_utf8_lossy`, a closure and a list slice per byte in YAML’s mapping scanner, hashing every key into a set even for small mappings, and an HTTP field list that was copied and lowercased three times per message. Those fixes made YAML three times faster, XML half as fast again and HTTP heads 30–40% faster. A second YAML pass (matching core-schema words directly instead of through list literals, rejecting words before slicing them as floats, scanning for unprintable characters 16 bytes at a time, and keeping quoted scalars without escapes as slices) made it 2.5 times faster again, about as fast as yaml-rust2. What remains is mostly design and compiler cost:

- **Owned values and a tree.** roc-parser returns a tree of Roc values. pulldown-cmark, quick-xml and httparse hand out borrowed slices or events, and roxmltree keeps its whole tree in one arena. Every list, record and copied string is a separate allocation and later a separate free, and reference counting adds an increment and decrement whenever a value is shared. Allocation, freeing and reference counting are 20–40% of the time in every profile.
- **Markdown copies text more than once.** A paragraph’s lines are joined, turned into a placeholder `Str`, turned back into bytes, stripped of indentation and then cut into text nodes that are validated again, and the block phase updates a large state record for every line. The October 2026 tuning removed the byte-at-a-time copies (trimming trailing spaces by rebuilding the line, appending lists byte by byte, building cells and code spans byte by byte), `Str.from_utf8_lossy` for every text node, a per-line index of every byte’s column, `List.take_first` and `drop_first` (see below) and byte loops that `Utf8.find_any` now does 16 bytes at a time, which together made it 2.3 times faster.
- **Compiler and runtime.** The Roc compiler is young. Measured costs that a parser cannot avoid today: a string literal longer than 23 bytes is copied to the heap every time it is evaluated, so an error message built on the success path costs an allocation; `List.drop_last` followed by `append` can copy the whole list instead of reusing it, and `List.take_first`, `drop_first` and `drop_last` copy instead of slicing once they have several call sites (roc-lang/roc#11965); a list literal such as

`["null", "Null"].contains(text)` is rebuilt on every call; and `Str.from_utf8` validates text that came from a `Str` and is known to be valid (XML now avoids this, see [How roc-parser uses SIMD and slices](#)). The `basic-cli` platform used by the drivers also hashes every freed pointer, about 5–10% of the time in allocation-heavy parsers.

Getting closer to Rust would need either borrowed results (slices or events instead of an owned tree) or compiler work on those points; both are tracked as future work rather than promised.

16.7. How roc-parser uses SIMD and slices

Roc’s `U8x16` builtin lowers to SSE/AVX on x86-64, NEON on Arm and `simd128` on WebAssembly. `Utf8.ByteClass` builds on it: a class is any set of bytes, and `Utf8.find_any`, `Utf8.skip_class`, `Utf8.find_line_end` and `Utf8.span_class` load 16 bytes at a time, test them all at once and count the trailing zeros of the resulting bit mask. Sets of one to three bytes are compared directly; larger sets, such as RFC 9110 token characters or XML name characters, use the nibble-table technique from `simdjson` (two calls to `table_lookup`, which are `pshufb` and `tbl`). A byte loop handles the last few bytes. Measured on an Apple M2, scanning 1 MiB with no match runs at 38–41 GB/s against about 2 GB/s for a byte loop, but the gain falls with the distance between matches: 3.4 times for 77-byte lines, 1.3 times for markup every ten bytes, and none when every other byte matches.

The format parsers already track byte offsets into the input and hand out slices of it. Since Roc lists and strings can be seamless slices, a `List.sublist` of the input, or `Str.from_utf8` of such a slice, shares the input’s memory instead of copying it. `Str.from_utf8` still validates the bytes each time, so the XML parser validates the whole input once and cuts names, text and attribute values from that `Str` with `Str.drop_first_bytes` and `Str.drop_last_bytes`, which check only the two boundary bytes.

Measured effect, 1 MiB generated documents, Apple M2:

Format	Before	After	Allocations
XML	10.9 ms (96 MB/s)	8.6 ms (122 MB/s)	98,072 → 64,677
CSV	8.6 ms (121 MB/s)	7.2 ms (146 MB/s)	105,415 → 99,214
HTTP head (64 fields)	24.7 μs	11.9 μs	313 → 11
YAML	48.2 ms	47.7 ms	unchanged

The HTTP gain is mostly not SIMD: the old token check tested membership in a list literal that was rebuilt for every byte. YAML did not change because splitting lines is a small part of its time. A faster scan matters only where runs between interesting bytes are long; in every format the

remaining cost is building the result tree (allocation, freeing and reference counting).

Retained memory. A slice keeps its whole input alive. A program that parses a large document and keeps one short string from it keeps the whole document in memory; copy such values if that matters.

Using a position into a borrowed input instead of returning the rest of the input as a slice was also measured, with minimal combinator cores of both shapes on 1 MiB inputs: it was 1.4 times faster on a list of numbers, 1.15 times slower on alternatives and 1.6 times faster on byte-by-byte spans. That is not a consistent enough gain to change `Parser(input, a)`. The same measurement found that `Parser` itself was 5–20 times slower than the minimal core because `Utf8.codeunit` and its relatives built an error message on every failure; building it once, when the parser is made, closed that gap.

16.8. Tips for your own parsers

- **Build with `--opt=speed`** before measuring anything.
- **Scan runs with `Utf8.ByteClass`.** Build a class once, at the top level, and use `Utf8.span_class`, `Utf8.skip_class` or `Utf8.find_any` for runs longer than a few bytes; `Parser.span` returns what a parser consumed as a slice.
- **Prefer byte-level primitives.** `Parser.chomp_while` and `Parser.chomp_until` consume a run of bytes in one step; `Parser.many` over a single-byte parser makes one closure call and one list append per byte.
- **Keep backtracking shallow.** `Parser.one_of` and `Parser.alt` re-run later alternatives from the same position. Order alternatives so the common case comes first, and make each one fail on its first byte where possible, for example by dispatching on a leading keyword or punctuation with `Utf8.codeunit`.
- **Avoid failing on the hot path.** A failed alternative still costs a `Try` and a re-run of the input. Recognising the end of a list by failure once is fine; failing once per element is not.
- **Build error messages only when you fail.** A string literal longer than 23 bytes is copied to the heap each time it is evaluated, so do not store a default error in a variable that a loop overwrites with `Ok`; return the error where it happens.
- **Do not turn literals into lists in a loop.** `"<!--".to_utf8()` allocates a list each time. Compare bytes directly, or check one leading byte first.
- **Stay in `List(U8)`, and slice rather than copy.** `List.sublist` and `List.drop_first` share the input; `List.append` in a loop copies it byte by byte. Convert to `Str` once per value you keep with `Str.from_utf8`, which shares the bytes, rather than `Str.from_utf8_lossy`, which decodes them twice and copies them.

- **Parse the outer structure with a scanner.** Like the built-in formats, split the document into records, lines or tokens with a loop over the bytes, then use combinators for each small piece. [Choosing how to read your input](#) discusses when that is worth it.
- **Recursion depth.** Recursive combinators (via `Parser . lazy`) use the call stack; impose a nesting limit for untrusted input, as the YAML parser does.
- **Measure with real documents.** Copy a driver from `bench/roc/`, point it at your parser, and compare inputs of increasing size: time that grows faster than the input signals a quadratic step.

Chapter 17. Glossary

The words this manual uses with a specific meaning, in alphabetical order. Terms that belong to one format name the format in parentheses.

Backtracking

Returning to an earlier position in the input to try another alternative after one fails. `Parser.alt` and `one_of` always backtrack: each alternative starts from the input the first one was given, however much the failed one read.

Chomping (YAML)

How a block scalar (`|` or `>`) treats the line breaks at its end. *Clip* (the default) keeps one, *strip* (`-`) keeps none, and *keep* (`+`) keeps them all. Unrelated to `Parser.chomp_while` and `chomp_until`, which consume input while or until a condition holds.

Combinator

A function that builds a parser from other parsers, such as `keep`, `skip`, `many` or `one_of`. A parser for a whole format is built by combining small ones. See [Modules](#).

Consumption

The part of the input a parser has read when it succeeds. The rest is passed to the next parser. A parser that succeeds without consuming anything makes no *progress*; `many` and `sep_by` stop repeating when an element makes no progress, so they cannot loop forever.

Core schema (YAML)

The YAML 1.2 rules that decide what an unquoted (plain) scalar means: `null` and `~` are null, `true` and `false` are Booleans, and numbers such as `12`, `0o14`, `0xC`, `1.5` and `.inf` are integers or floats. Anything else is a string. Quoted scalars are always strings.

Flanking (CommonMark)

The rule that decides whether a run of `or` can open or close emphasis. A run is *_left-flanking* when it is not followed by whitespace and, if it is followed by punctuation, is preceded by whitespace or punctuation; *right-flanking* is the mirror image. So `*foo` is emphasis but `* foo *` is not.

Format

In type-directed decoding, a type whose methods (`parse_str`, `parse_record_field`,

`parse_list_start` and so on) read one value at a time from the input. `CSV.Format` and `Yaml.Format` are formats; you use them through `CSV.parse` and `Yaml.decode` and never name them. See `parser_for`.

Framing (HTTP)

How the receiver finds where a message body ends: by `Transfer-Encoding: chunked`, by `Content-Length`, or, for a response with neither, by the end of the connection. See [smuggling](#).

Furthest failure

Of all the failures seen while parsing, the one that read furthest into the input, including failures that an alternative or a repetition recovered from. The runners report it as `ParseError({ message, offset })`, so the error points at the bad element and not at the input left over after it.

Input

The data a parser reads, given by the first type parameter of `Parser(input, a)`. For text it is `Utf8.Bytes`, a list of UTF-8 bytes; for CSV record decoding it is a `CSV.Record`, a list of fields.

Known failure

A conformance case where the library and the oracle disagree on purpose, listed with its reason in a `known-failures.json` file. See [Conformance](#).

Method

A function that belongs to a type, defined in the `.{ }` block after the type declaration and called with a dot on a value: `parser.keep(field)` calls `Parser.keep(parser, field)`. This manual writes calls in that method-call style. The compiler finds the method from the value's type, which is [static dispatch](#).

Opaque nominal type

A nominal type declared with `::`, whose backing representation is hidden outside its module, so values can only be built and read through its methods. `Parser(input, a)` is one. A nominal type declared with `:=`, such as `Yaml` or `Markdown`, shows its tags, so you can match on them.

Oracle

A trusted reference implementation whose results a test compares with the library's, such as Python's `csv` module for CSV or `expat` for XML. When the oracle is itself wrong, the specification decides. See [Conformance](#).

Parseable

A where-clause alias, `row.Parseable(errs)`, that `CSV.parse` and the other decoding functions use to say "a type that this format can decode" without naming the format's internal state type. You rarely write it yourself.

Parser

A value of type `Parser(input, a)`: a description of how to read an `a` from the start of an input. Running it returns the value and the unconsumed input, or a `ParseError` with a message and an offset.

parser_for

The method the compiler derives for records, tuples, lists and the builtin scalar types, which reads a value of that type with a given `format`. It is how `CSV.parse` and `Yaml.decode` decode into the type your annotation names. A field of type `Try(a, [Missing])` is optional; a required field that is absent fails with `MissingRequiredField(name)`, which the compiler adds to the error type.

Parsing incomplete

The parser succeeded but did not consume the whole input. Functions such as `Utf8.parse_str`, which expect to read everything, report it as a `ParseError` at the furthest failure, or as `unexpected input` at the leftover.

Progress

See [consumption](#).

Property test

A test that generates many inputs and checks a rule that must hold for every one of them, such as "the parser never crashes" or "parsing a rendered value gives back the value". This package runs its property tests with coverage guidance from the `roc-fuzz` library. See [Property testing](#).

Request smuggling (HTTP)

An attack in which two programs that read the same bytes, such as a proxy and a server, disagree on where one message ends. The attacker hides a second request inside the first one's body. A parser prevents it by rejecting any message whose framing could be read two ways.

Soft line break (CommonMark)

A line break inside a paragraph that is not a hard break. HTML renders it as a space; this library keeps it as `\n` in the `Text` node.

Static dispatch

Choosing which function a method call runs from the type of the value, at compile time. It is how `value.to_inspect()` finds the `Yaml` version for a `Yaml` value, and how `CSV.parse` finds the `parser_for` of your row type.

Try

Roc's builtin type for a result that may fail: `Try(ok, err)` is `Ok(value)` or `Err(problem)`. Every fallible function in this package returns one. An optional value is `Try(a, [Missing])`.

Type module

A module named after the one type it defines, with that type's methods in the `.{ }` block of its declaration. `Parser`, `Yaml`, `Xml` and `Markdown` are type modules. `Utf8`, `CSV` and `HTTP` define their type as `[]`, an empty tag union, because they group functions and nested types rather than values of their own.

Well-formed (XML)

Following the syntax rules of XML 1.0: one root element, properly nested and matching tags, quoted and unique attributes, and only declared entities. Well-formedness says nothing about which elements are allowed; that is *validity*, which needs a schema or DTD and which this package does not check.

Where clause

The part of a type annotation that lists the methods a type variable must have, such as `where [input.len : input -> U64]` on `Parser.many`, which needs only the length of its input to detect progress.

Contributing

Chapter 18. Contributing to roc-parser

This chapter is for people who want to change roc-parser: fix a bug, improve a format parser, or improve this manual. It assumes you can read Roc and use Git and Python 3. When you finish it you will have a working checkout, know which checks CI runs, and know what a pull request needs before it can be merged.

Related chapters: [Property tests and conformance reviews](#) explains the fuzz targets and conformance reviews, [Add a format module](#) walks through adding a new format module, and [Compiler updates, releases and security reports](#) covers compiler updates and releases.

18.1. Report a problem or propose a change

Report bugs and propose changes in the [roc-parser issue tracker](#). For a parsing bug, include the roc-parser release or commit, the output of `roc version`, and the smallest input that shows the problem. For a change to a public API or to the syntax a format accepts, open an issue first so the shape can be agreed before you write the code.

Do not report a suspected vulnerability in a public issue. Follow the private process in [the security policy](#) instead.

18.2. Development setup

Install:

- the Roc nightly named in `.roc-version`, for example `nightly-2026-09-29-7f11a82`;
- Python 3, for the test, review and release scripts;
- Docker, to build this manual locally (see [Build the documentation](#)).

The scripts find the compiler through the `ROC` environment variable and fall back to `roc` on `PATH`. Point `ROC` at the nightly you installed:

```
export ROC=/path/to/roc_nightly/roc
$ROC version
```

Output:

```
Roc compiler version nightly-2026-09-29-7f11a82
```

With Nix, `blueprint shell` provides this exact nightly together with Go, Rust and Python, on Linux and on Apple silicon Macs; see [Reproducible environment with roc-blueprint](#).

`.roc-version` is the only compiler pin; `Blueprint.roc` mirrors it, and a test keeps the two in step. The package, its tests, its examples, its releases, the property tests, the conformance reviews and the benchmarks all use it, locally and in CI.

18.3. Repository map

Path	Purpose
<code>package/</code>	The library: <code>Parser</code> , <code>Utf8</code> , and one module per format
<code>examples/</code>	Small apps that use the package source (<code>../package/main.roc</code>); releases attach them pinned to the bundle URL
<code>fuzz/</code>	Property-test targets, seed inputs and dictionaries (Property tests and conformance reviews)
<code>scripts/</code>	Test, review, fuzz, documentation and release automation, all in Python
<code>scripts/<format>/</code>	Conformance probe, cases and known-failures baseline for each format
<code>scripts/tests/</code>	Unit tests for the scripts, including the repository policy
<code>docs/</code>	This manual, in AsciiDoc
<code>docs/releases/</code>	Release notes, one Markdown file per release

18.4. Run the tests

Each module carries its tests as `expect` blocks next to the code. Run them all through the package:

```
$ROC test package/main.roc
```

Output:

```
All (253) tests passed in 535.8 ms.
```

The number grows as tests are added.

Run the unit tests for the Python scripts. These include the repository policy described below:

```
python3 -m unittest discover -s scripts/tests -p "test_*.py"
```

Run the same validation as the `Tests` workflow. `scripts/all_tests.py` checks and tests every module, generates the API documentation, builds a bundle, and then packages the examples archive exactly as a release does, pinned to that bundle served on localhost, and checks, runs and builds every app in it. Its scratch files go under `.roc-parser-tmp/`:

```
python3 scripts/all_tests.py
```

A quick property-test pass over every fuzz target takes a few minutes (see [Property tests and conformance reviews](#)):

```
ROC=/path/to/quality_nightly/roc python3 scripts/run_fuzz.py smoke all
```

18.4.1. Repository policy

`scripts/tests/test_repository_policy.py` enforces three rules. A pull request that breaks one fails CI:

- Automation is Python under `scripts/`. No shell scripts (`.sh`) are allowed anywhere, and no `.py` file may live outside `scripts/`.
- Every `uses:` in a workflow names an action by its full 40-character commit SHA, not a tag or branch. Local actions (`./...`) are exempt.
- Workflows do not use the `pull_request_target` or `workflow_run` triggers, and do not contain multi-line `run: |` or `run: >` blocks. Put the logic in a script under `scripts/` and call it on one line.

`scripts/tests/test_run_fuzz.py` also requires every `fuzz/*.roc` target to be registered in `scripts/run_fuzz.py`, and the CI matrix in `fuzz.yml` to match that registry.

18.5. Build the documentation

The manual is AsciiDoc under `docs/`, rendered by Asciidoctor in Docker, so you need Docker running but no Ruby toolchain. Build the HTML manual:

```
python3 scripts/build_manual.py
```

Add `--pdf` to also build the PDF, and `--docs-version V` to stamp a version other than the

development one.

The API reference is generated from the doc comments in `package/` with `roc docs`. Build it, or check that it builds without writing anything:

```
python3 scripts/build_docs.py
python3 scripts/build_docs.py --check
```

Code examples in the manual are compiled and run, so a changed API breaks the build rather than silently leaving a stale example:

```
python3 scripts/check_doc_examples.py
```

The `Docs` workflow (`.github/workflows/docs.yml`) runs these on pull requests.

When you write a chapter, follow the [shared documentation guide](#): give each page one purpose (tutorial, how-to, reference or explanation), name the reader and the outcome at the top, label commands and their output separately, and put any warning before the step it applies to.

18.6. Before you open a pull request

- Format changed Roc files with `roc fmt path/to/file.roc`, using the compiler pinned in `.roc-version`.
- Add or update `expect` tests for the accepted input, the rejected input and the boundary cases of the behaviour you changed.
- When you change parser behaviour, run that format’s conformance review and, if the change touches a fuzzed path, a smoke run of its targets ([Property tests and conformance reviews](#)). If a fuzz target or review found the bug, add the input as a regression.
- Fix the cause of a failure. Do not add a gate, a skip or a known-failures entry to make a check pass unless the input is genuinely outside what the module supports, and say why in the entry.
- Update the doc comments, the manual and the release notes in `docs/releases/` when a public API or the accepted syntax changes.
- Keep the change focused: no unrelated formatting or refactoring, and no generated files or local build output (`.roc-parser-tmp/`, `.roc-fuzz/`).
- Sign your commits. The `main` branch requires verified signatures; see [GitHub’s guide to signing commits](#) to set up an SSH or GPG key.

Pull requests need passing CI: `Tests`, `Property quality tests` (smoke runs of every target and every format’s conformance review) and `Docs`. Human review is encouraged; answer review comments before asking for another review after a substantial change.

18.7. License

By contributing, you agree that your contribution is licensed under the [Universal Permissive License v1.0](#).

Chapter 19. Property tests and conformance reviews

This chapter is for contributors who change a parser and need to know whether the change is correct. The first half explains how roc-parser checks its parsers beyond the `expect` tests: property tests driven by a fuzzer, and reviews against a mature reference implementation. The second half is a set of how-to sections: run the targets, reproduce and minimise a failure, turn it into a regression, and update a conformance baseline.

It assumes the setup in [Contributing to roc-parser](#), including the Roc nightly pinned in `.roc-version`. Every command below uses that compiler:

```
export ROC=/path/to/quality_nightly/roc
```

19.1. Why property tests

An `expect` test checks one input that somebody thought of. Parsers fail on the inputs nobody thought of: a carriage return on its own, a quote in the middle of a field, 100,000 nested brackets. A property test states something that must hold for every input, and a fuzzer searches for an input where it does not.

The fuzzer is `roc-fuzz`, a Roc platform that embeds libFuzzer. It is coverage guided: it keeps inputs that reach new code and mutates them further, so it finds deep paths far faster than random input would. The targets are library quality tests, not a security programme, and they run on every pull request and nightly in CI.

A failing property is a bug in the parser, or occasionally in the property. Either way the outcome is a fix: see [Fix the cause, do not gate it](#).

19.2. Three kinds of target

Each file `fuzz/<target>.roc` is one roc-fuzz app. They come in three kinds, which differ in what the fuzzer's bytes mean and therefore in what the target can check.

Raw targets (`xml-raw`, `yaml-raw`, `csv-raw`, `http-raw`, `markdown-inline`, `markdown-document`, `string-`

`primitives`): The bytes are the input document, often invalid UTF-8. Nothing is known about the expected result, so the target checks properties that hold for any document: the parser does not crash or hang, an error points inside the input, LF/CRLF/CR line endings give the same result, appending a final line break changes nothing, and the result survives re-serialisation. These are called *metamorphic relations*: they relate two runs of the parser instead of comparing one run with a known answer. Many raw targets also switch to a pathological generator (deep nesting, thousands of attributes, long delimiter runs) when the input starts with a marker byte, so that the per-input `--timeout` catches quadratic behaviour.

Structured targets (`csv-roundtrip`, `csv-decode`, `xml-roundtrip`, `xml-malformed`, `yaml-roundtrip`, `yaml-block`, `http-roundtrip`, `http-smuggling`, `markdown-inline-ast`, `markdown-blocks`, `markdown-refdefs`, `parser-combinators`): The bytes are *choices*. A generator reads them to build a value (a table, an XML tree, an HTTP message, a Markdown syntax tree) and a way to write it down, and builds the expected parse alongside the text from the specification, never by calling the parser. The target then requires the parser to return exactly that value. Mutating generators such as `xml-malformed` and `http-smuggling` instead inject one known violation into a valid document and require the parser to reject it. `parser-combinators` builds a random combinator expression and compares the real parser with a small reference interpreter of the documented semantics.

Seeded targets (`csv`, `xml`, `yaml`, `http-request`, `http-response`)

The original targets read a string through roc-fuzz's `Fuzz.str` generator. They start from reviewable seeds in `fuzz/seeds/<name>.json`, which the runner encodes for `Fuzz.str` at run time, and use the token lists in `fuzz/dictionaries/` to mutate towards real syntax. Seed strings are limited to 255 UTF-8 bytes by that encoding.

Raw targets also take seeds and a dictionary, written as plain text. Structured targets start from an empty corpus, because their bytes are not text.

19.3. Oracles

A property test can only be as good as its expected answer. Structured targets build that answer from the specification, but a generator can share a misunderstanding with the parser. So each format is also compared with a mature, independent implementation, its *oracle*, pinned to an exact version:

Format	Oracle	Specification and suite
CSV	Python's <code>csv</code> module in strict mode	RFC 4180, plus the documented dialect
XML	expat (Python's <code>pyexpat</code>)	XML 1.0 5th edition, W3C XML Conformance Test Suite 2013-09-23

Format	Oracle	Specification and suite
YAML	ruamel.yaml	YAML 1.2, the yaml-test-suite
HTTP	h11	RFC 9112 and RFC 9110; cases pin where RFC 9112 is stricter than h11
Markdown	cmark-gfm 0.29.0.gfm.13 (via cmarkgfm), with markdown-it-py to adjudicate	CommonMark 0.31.2 spec examples and GFM extensions

Oracles have bugs too. When roc-parser and the oracle disagree, the review reports it, and a person decides which one the specification supports. Known oracle quirks are named in the review scripts and in the baselines, with the section of the specification that settles them.

19.4. Run the property tests

Run every target for a short, iteration-bounded smoke test. This is what CI runs on a pull request:

```
python3 scripts/run_fuzz.py smoke all
```

Run one or more targets by name, and lower the iteration count for a quick check while you work:

```
python3 scripts/run_fuzz.py smoke csv --runs 200
```

The last lines of the output are libFuzzer's statistics:

```
Done 200 runs in 0 second(s)
stat::number_of_executed_units: 200
stat::average_exec_per_sec:    0
stat::new_units_added:        39
stat::slowest_unit_time_sec:   0
stat::peak_rss_mb:            25
```

Run a longer, time-bounded campaign on the target closest to your change. CI runs a 600-second campaign for every target each night:

```
python3 scripts/run_fuzz.py campaign markdown-inline-ast --seconds 600
```

`--max-input-size`, `--timeout` and `--memory-limit` override a target's registered bounds, and `--no-build` reuses the binary from the last run.

The runner keeps everything under `.roc-parser-tmp/fuzz/`:

- `bin/<target>`: the built target;
- `corpus/<target>/`: the working corpus, which grows across runs;
- `runs/<target>/`: runner metadata, and the inputs of any failure.

19.5. Reproduce and minimise a failure

A failing run prints the path of its metadata and of the saved input. In CI, the `fuzz-failure-<target>` artifact holds the binary, the corpus and the run directory. Use the same target name for every step.

1. Show the input as the target decodes it. For a structured target this is the generated document and its expected value:

```
python3 scripts/run_fuzz.py show csv-roundtrip INPUT --no-build
```

2. Replay it to confirm that it still fails:

```
python3 scripts/run_fuzz.py replay csv-roundtrip INPUT --no-build
```

3. Minimise it into a smaller input that fails the same way:

```
python3 scripts/run_fuzz.py minimize csv-roundtrip INPUT OUTPUT --no-build
```

`INPUT` can be any file, including one from the corpus. Showing a corpus entry of the seeded `csv` target prints the string it decodes to:

```
python3 scripts/run_fuzz.py show csv .roc-parser-  
tmp/fuzz/corpus/csv/13d7ab0a19cc0856931196f0e8bac548e045234b --no-build
```

Output (the last line):

```
"a"
```

Replaying a saved input creates a `.roc-fuzz/` directory, which Git ignores.

19.6. Add a regression

Once you understand the minimised input:

1. Write the smallest `expect` in the module that fails for the same reason, and fix the parser until it

passes. The `expect` is the regression; it runs in every build.

2. For a seeded or raw target, add the readable input to `fuzz/seeds/<name>.json` so the fuzzer starts near the bug in future runs.
3. If the generator of a structured target avoided the input's shape, remove that restriction so the target covers it from now on.
4. Keep the raw minimised input and the target binary outside the repository when exact reproduction of a historical failure matters.

19.7. Fix the cause, do not gate it

When a property fails, change the parser, not the test. Do not add a condition to a target that skips the failing shape, a known-gaps flag, or a baseline entry, to make CI green. A gate hides the bug from every later run, and the fuzzer will not tell you when it comes back.

There are two legitimate exceptions, and both need a written reason:

- the input is outside the subset the module documents as supported (for example a DOCTYPE in XML, or multi-line flow collections in YAML); and
- the oracle is wrong and the specification says so.

Several gates that once existed were removed by fixing the parser, which is how the targets found the CSV final-record bug, the YAML block scalar chomping bugs and the quadratic Markdown inline cases.

19.8. Run a conformance review

Each format has a review script that runs a small probe app, `scripts/<format>/probe.roc`, which prints the parser's result as JSON, and compares it with the oracle. Install the pinned oracle into a virtual environment first:

```
python3 -m venv .venv
.venv/bin/python -m pip install --no-deps -r scripts/yaml/requirements.txt
```

`scripts/http/`, `scripts/markdown/` and `scripts/yaml/` each have a `requirements.txt` with every dependency pinned, so `--no-deps` installs exactly the reviewed versions; CSV and XML use the Python standard library. Then run the same checks as the `Conformance` jobs in CI:

```
.venv/bin/python scripts/review_yaml.py check --baseline scripts/yaml/known-failures.json
.venv/bin/python scripts/review_yaml.py campaign --smoke
python3 scripts/review_xml.py check --baseline scripts/xml/known-failures.json
python3 scripts/review_csv.py
```

```
.venv/bin/python scripts/review_http.py
```

The CSV review runs the hand-written cases, 2000 seeded random inputs and, with `--corpus DIR`, a fuzz corpus. Its last line summarises the run:

```
cases=2035 unexpected=0 known=0 fixed_known=0
```

The Markdown reviews compare against the CommonMark spec examples and `cmark-gfm`:

```
.venv/bin/python -m pip install --no-deps -r scripts/markdown/requirements.txt
.venv/bin/python scripts/review_markdown.py check
.venv/bin/python scripts/review_markdown_blocks.py check --baseline
scripts/markdown/blocks/known-failures.json
```

Several scripts can also check a structured target's generator against the oracle, which is how a generator's own mistakes are caught: `review_xml.py corpus <target> <corpus>`, `review_http.py --corpus DIR --fuzz-show BINARY`, `review_markdown.py generated BINARY CORPUS` and `review_markdown_blocks.py crosscheck BINARY CORPUS`.

19.9. Known-failures baselines

Each format keeps a baseline at `scripts/<format>/known-failures.json` (for Markdown blocks, `scripts/markdown/blocks/known-failures.json`): the cases where roc-parser and the oracle are allowed to disagree. Every entry carries a reason, of one of these kinds:

- the input is valid but outside the documented subset;
- the oracle is wrong, citing the specification (for example, expat does not check `VersionNum`); or
- a documented design decision (for example, YAML mapping keys are their source text).

A review fails when a case disagrees and is not in the baseline, and reports any baseline entry that now passes as stale. When your change makes a known failure pass, remove the entry. When it introduces a new disagreement, fix the parser; add an entry only for one of the reasons above. `review_yaml.py` and `review_markdown_blocks.py` can rewrite the baseline with `--record-known-failures PATH`; review the diff entry by entry before you commit it.

19.10. OpenSSF Scorecard

The OpenSSF Scorecard *Fuzzing* check detects OSS-Fuzz, ClusterFuzzLite and a few language-specific frameworks, but not Roc targets. A low Fuzzing score is a detection gap, not evidence that these campaigns did not run.

Chapter 20. Add a format module

This how-to is for contributors who want to add a ready-made parser for a new text format, alongside CSV, XML, YAML, Markdown and HTTP. It assumes [Contributing to roc-parser](#) and [Property tests and conformance reviews](#), and that you have opened an issue to agree that the format belongs in the package. When you finish, the new module has the same evidence behind it as the existing ones: tests, a review against a mature implementation, property tests in CI, and a chapter in this manual.

The steps use a hypothetical format called TOML as the example. Follow them in order: each step relies on the one before.

20.1. 1. Decide what you support

Write down, before any code, which specification and version the module follows, and which parts of it the module supports. Every existing module supports a documented subset and rejects the rest with an error, rather than misreading it. For example XML rejects a DOCTYPE, and YAML rejects multi-line flow collections. A rejected input is a clear limit; a misread one is a bug that nobody notices.

Pick the oracle now too: a widely used implementation that you can pin to an exact version and call from Python, such as `tomllib` for TOML. The oracles of the existing formats are listed in [Property tests and conformance reviews](#).

20.2. 2. Create the module

1. Add `package/TomL.roc`. Start it with a `##` doc comment that names the specification, lists what is supported and what is rejected, and describes the error type. This comment becomes the module's page in the API reference.
2. Give every exposed value a `##` doc comment with its purpose and, where it helps, a short `expect` example.
3. Expose the module in `package/main.roc`. `roc test package/main.roc`, which `scripts/all_tests.py` runs, then runs its tests with the rest.

4. Make it a type module and follow the conventions of the existing ones:

- ≡ `Toml.parse_str : Str -> Try(Toml, [InvalidToml(Toml.Error)])`, where `Toml.Error` is a record with a `message` and the most precise location the format has (`line` and `column` for a text format);
- ≡ `Toml.parser : Parser(Utf8.Bytes, Toml)` for callers who compose parsers, failing with `ParseError({ message, offset })`;
- ≡ `Try(a, [Missing])` for every optional value, and methods in the `.{ }` block, called in method-call style in examples;
- ≡ no public function that can crash, on any input.

5. If records are a natural target, also implement a format for type-directed decoding, as `CSV.Format` and `Yaml.Format` do, and expose `Toml.decode` with a `Parseable` where-clause alias. The parsers chapter of the Roc language reference lists the methods a format provides.

Prefer a scanner over the input bytes for document-level parsing, with an explicit stack for nesting. The combinator style is convenient for small grammars, but every rewritten module (`CSV`, `XML`, `HTTP`, `Markdown`) moved to scanners to stay linear in the input size and to avoid deep recursion.

20.3. 3. Write the tests

Put `expect` tests next to the code they cover. For every rule, test an input that is accepted, one that is rejected, and the boundary between them. Add a test for each limit you chose in step 1. Run them:

```
$ROC test package/Toml.roc
$ROC test package/main.roc
```

20.4. 4. Add a probe and a review script

A conformance review runs the same inputs through your module and the oracle and compares the results. It needs three pieces under `scripts/toml/`, modelled on `scripts/csv/`, the smallest of the existing reviews:

`probe.roc`

A basic-cli app that reads one document on standard input and prints one JSON line: the parsed value or an error. Use the public API only.

`cases.json`

Hand-written cases with a `name` and an `input`, one for each rule and each known edge case. Add

the specification’s own examples or test suite if there is one, pinned by URL and SHA-256 as `scripts/review_xml.py` does for the W3C suite.

`known-failures.json`

The baseline. It starts empty; every entry added later needs a reason (see [Property tests and conformance reviews](#)).

Then add `scripts/review_toml.py`. It builds the probe with `$ROC` (falling back to `roc` on `PATH`), runs every case through the probe and the oracle, converts both results to the same shape, and exits non-zero on any disagreement that is not in the baseline. Report baseline entries that now pass as stale. If the oracle is a third-party package, pin it, and all of its dependencies, in `scripts/toml/requirements.txt` so that `pip install --no-deps -r` installs exactly the reviewed versions.

Run it, fix the parser until it passes, and add a unit test under `scripts/tests/` for any logic in the script that is not a direct call to the oracle.

20.5. 5. Add property-test targets

Add at least two targets under `fuzz/`, as described in [Property tests and conformance reviews](#):

`toml-raw.roc`

Raw bytes. Check that the parser never crashes, that errors point inside the input, and the metamorphic relations that hold for the format, such as line-ending equivalence. Bound pathological shapes with the runner’s `--timeout`.

`toml-roundtrip.roc`

A structured generator. Build a value and a way of writing it from the bytes, build the expected result from the specification without calling the parser, and require the parser to return exactly that value.

Check the generator against the oracle on a corpus before you trust it: add a `corpus` mode to the review script, like `review_xml.py corpus`.

20.6. 6. Register the targets

1. Add each target to `TARGETS` in `scripts/run_fuzz.py`, with its seed encoding (`raw` for raw bytes, `none` for a structured target), seeds, dictionary, input size and timeout.
2. Add reviewable seeds in `fuzz/seeds/toml.json` and syntax tokens in `fuzz/dictionaries/toml.dict` for the raw target.
3. Add each target to the `target` matrix of the `fuzz` job in `.github/workflows/fuzz.yml`, and the review

to the conformance matrix with its `requirements` file and command.

`scripts/tests/test_run_fuzz.py` fails if a `fuzz/*.roc` file is not registered, or if the workflow matrix and the registry disagree. Check that and run the new targets:

```
python3 -m unittest discover -s scripts/tests -p "test_*.py"
python3 scripts/run_fuzz.py smoke toml-raw toml-roundtrip
```

20.7. 7. Document the module

1. Add a chapter `docs/toml.adoc` that shows how to parse a document, how to handle errors, and what is and is not supported. Include it from the manual's index with `leveloffset=+1`, like the other format chapters.
2. Make every code example in the chapter a complete program that `scripts/check_doc_examples.py` compiles and runs, so it cannot drift from the API.
3. Add the module to the release notes for the next version in `docs/releases/`.

Build and check the documentation:

```
python3 scripts/build_docs.py --check
python3 scripts/check_doc_examples.py
python3 scripts/build_manual.py
```

20.8. Checklist

- ≡ Supported subset and oracle written down in the module doc comment
- ≡ Module exposed in `package/main.roc` and listed in `scripts/all_tests.py`
- ≡ `expect` tests for accepted, rejected and boundary inputs
- ≡ Probe, cases, review script and empty baseline under `scripts/`
- ≡ Raw and structured fuzz targets, registered in `run_fuzz.py` and `fuzz.yml`
- ≡ Chapter with tested examples, and a release notes entry

Chapter 21. Benchmarks

This chapter is for contributors who want to measure a change, add a benchmark or reproduce the figures in [Performance](#). It covers the harness, the corpora, the comparison programs, the environments the benchmarks run in, and how to read a report.

21.1. Run the benchmarks

You need the Roc nightly named in `.roc-version`. Go, Rust (cargo) and Python 3 are optional: a missing toolchain is skipped and named in the report.

```
$ export ROC=~/.roc_nightly-macos_apple_silicon-2026-09-29-7f11a82/roc
$ python3 scripts/bench.py --quick           # about two minutes
$ python3 scripts/bench.py --allocations     # the full comparison
```

`--quick` measures each format's samples, its small generated document and one pathological document, with three short batches. Use it to check that nothing is broken, not to compare numbers. The full run takes about half an hour on a laptop.

Narrow a run with these options:

`--formats csv,xml`

Only these formats.

`--langs roc,go`

Only these implementations (`roc`, `rust`, `go`, `python`).

`--only large,sample-svg`

Only documents with these names or kinds (`sample`, `small`, `medium`, `large`, `pathological`).

`--label before`

Write the report to `.roc-parser-tmp/bench/before/`, so two runs can be compared side by side.

`--repetitions`, `--target-ms`, `--timeout`

Batch count, approximate batch duration and the time limit for one batch.

--corpus-only

Write the corpus and its manifest, then stop.

To compare two revisions of the parser, check each one out in turn and run the harness with a different `--label`; the report records the commit and whether the tree was dirty.

21.2. How a measurement works

Every implementation, in Roc or in another language, is a small program that follows one protocol. It reads a document from standard input, starts a monotonic clock (Roc's is the UTC clock, so batches must be long), parses the document a given number of times, stops the clock, and prints one JSON line:

```
{"iterations":120,"elapsed_ns":20414000,"successes":120,"checksum":43811}
```

Process start-up, reading the input and printing are outside the timed region. Each parse builds the implementation's in-memory result and then folds over it into the checksum, so the work cannot be optimised away and every implementation pays for the structure it returns.

For each document and implementation, `scripts/bench.py`:

1. runs one parse to estimate its cost, and picks the number of parses per batch so a batch takes about `--target-ms` (200 ms, or 20 ms with `--quick`);
2. runs one warmup batch and discards it;
3. runs `--repetitions` batches (7, or 3 with `--quick`), each in a fresh process;
4. records the median time per parse, the minimum, maximum, standard deviation and spread ($(\text{max} - \text{min}) / \text{median}$), throughput in MB/s (10^6 bytes per second, from the median), and the peak resident set size of the batch processes;
5. with `--allocations`, replays the document once through `bench/roc/alloc.roc`, a diagnostic built with `roc build --fuzz`, and records how many times the Roc parser called `roc_alloc` or `roc_realloc`. The program crashes on purpose to report the count, so its exit code 77 is expected.

Peak RSS is the whole process, so it includes the language runtime: about 1.5 MiB for Roc and Rust, 12–20 MiB for Go and Python, before any document is read. Compare it within a language or across document sizes, not as an absolute cost of a parser.

21.3. Corpora

The corpus is generated by `build_corpus` in `scripts/bench.py` every run. Each generated document uses a random number generator seeded from its own name, so the bytes, and the SHA-256 recorded for each one in the report, are identical on every machine and run. Changing a generator changes the hashes, which marks results as not comparable with earlier ones.

Each format has five kinds of document:

sample

Small real-world-shaped files in `bench/corpus/samples/`: a CSV order export, this repository's fuzz workflow (YAML), a design-tool-style SVG, this repository's README, and an HTTP browser request and JSON API response (kept in `HTTP_SAMPLES` so their CRLFs survive). They are frozen copies; do not update them.

small, medium, large

Generated documents of about 4 KiB, 64 KiB and 1 MiB in the shape of typical data: CSV rows with quoted fields and embedded newlines, YAML service configuration with flow collections and block scalars, an XML catalogue with attributes and entities, a Markdown document with headings, lists, code, quotes and tables, and HTTP POST requests with a JSON body and chunked HTML responses.

pathological

Inputs that stress one mechanism: very wide CSV rows and long runs of escaped quotes, YAML nesting at the 100-level limit and long flow sequences, deeply nested XML and thousands of attributes, unclosed Markdown emphasis and nested brackets, and HTTP messages with many headers or thousands of one-byte chunks. Several of these were super-linear before the fuzzing work described in [Complexity guarantees](#).

The corpus and a `manifest.json` with every document's size and hash are written to `.roc-parser-tmp/bench/<label>/corpus/`, so you can feed a document to a driver by hand:

```
$ .roc-parser-tmp/bench/full/bin/roc-yaml 100 < .roc-parser-
tmp/bench/full/corpus/yaml/generated-medium.yaml
```

21.4. Roc drivers

`bench/roc/` holds one driver per format, each a few lines calling the format's public entry point: `CSV.parse_with` with `Parser.many(CSV.field(CSV.string))`, `Yaml.parse_str`, `Xml.parse_str`, `Markdown.parse_str`, and `HTTP.parse_request` or `HTTP.parse_response`. The timing loop is shared in `bench/roc/Bench.roc`. The harness builds each driver with `roc build --opt=speed`; never benchmark `roc dev` or an unoptimised build, which is many times slower.

When an entry point is renamed, only the call in that format's driver (and the matching line in `alloc.roc`) changes.

21.5. Comparison programs

Language	Where	Libraries, pinned by
Rust	<code>bench/compare/rust/</code>	<code>csv</code> , <code>yaml-rust2</code> , <code>serde_yaml</code> , <code>quick-xml</code> , <code>roxmltree</code> , <code>pulldown-cmark</code> , <code>comrak</code> and <code>httparse</code> , pinned exactly in <code>Cargo.toml</code> and <code>Cargo.lock</code> ; built with <code>cargo build --release --locked</code> , thin LTO and one codegen unit.
Go	<code>bench/compare/go/</code>	<code>encoding/csv</code> , <code>gopkg.in/yaml.v3</code> , <code>encoding/xml</code> , <code>goldmark</code> and <code>net/http</code> , pinned by <code>go.mod</code> and <code>go.sum</code> .
Python	<code>scripts/bench_python.py</code> and <code>bench/compare/python/requirements.txt</code>	<code>csv</code> , <code>PyYAML</code> (pure Python and <code>libyaml</code>), <code>xml.etree</code> , <code>markdown-it-py</code> and <code>h11</code> . The program is under <code>scripts/</code> because repository policy keeps every Python file there. The harness installs the pins into a virtual environment in <code>~/.cache/roc-parser/bench-venv</code> (outside the checkout, for the same reason); pass <code>--python</code> to use another interpreter.

Each program builds the same kind of in-memory structure as `roc-parser`:

- **CSV:** every field copied into an owned string, rows into lists.
- **YAML:** the library's generic value tree, walked once.
- **XML:** a tree of elements, attributes and text. `roxmltree` and `ElementTree` build one; for `quick-xml` and `encoding/xml`, which are streaming tokenizers, the program builds an owned tree from the events.
- **Markdown:** `comrak` and `goldmark` build an AST. `pulldown-cmark` and `markdown-it-py` produce a flat event or token list, which is collected and, for `pulldown-cmark`, copied into owned events.
- **HTTP:** the request line or status line, headers and the body, decoded from `Content-Length` or chunked framing and copied. `httparse` only parses the head, so the program frames the body itself with `httparse`'s chunk-size parser.

These choices, and where they still differ, are listed in [What the comparison does and does not](#)

`show`. Read them before quoting a ratio.

To add a comparator, add a function to the language's program, register its name in `COMPARATORS` in `scripts/bench.py` with a one-line description of what it measures, pin the library, and run `--quick --langs <lang>` to check that it accepts the samples.

21.6. Add a benchmark

- **A document:** add a generator branch or a pathological entry in `scripts/bench.py`, or a frozen file to `bench/corpus/samples/` with a licence that allows it, noted in that folder's README. Keep generated documents deterministic: draw randomness only from the `rng` passed in.
- **A format:** add `bench/roc/<format>.roc` calling the public entry point through `Bench.run!`, a branch in `alloc.roc`, a generator and samples, and at least one comparator per language.

`scripts/tests/test_bench.py` checks that the corpus is deterministic, that every format has every kind of document, and that each format has a driver and comparators; run it with `python3 -m unittest discover -s scripts/tests`.

21.7. Read a report

Each run writes three things to `.roc-parser-tmp/bench/<label>/`:

`results.json`

Metadata (date, machine and CPU, OS, Roc, Go, Rust and Python versions, every crate, module and package version, the commit and whether the tree was dirty, the corpus manifest with hashes, and what each implementation measures), one record per document and implementation, and the comparison table.

`report.md`

The same as Markdown tables.

`logs/`

Build output and allocation diagnostics.

The **comparison** table gives, per format and library, the geometric mean over the non-pathological documents that both implementations accepted of the ratio of median times. "roc-parser is 3x slower" means the library parsed those documents three times as fast. Pathological documents are left out of the mean because they measure worst cases, which differ by design between parsers; read them individually.

In the per-document table, check these before trusting a number:

- **Result:** `rejected` means the implementation refused the document (for example `ElementTree`’s nesting limit, or a strictness difference). A rejection is often much faster than a parse, so the times are not comparable.
- **Spread:** above about 10%, the machine was busy or the batch too short; rerun on an idle machine or raise `--target-ms`.
- **Corpus hashes and versions:** two reports can be compared only when their corpus hashes match and only the variable under test changed.

21.8. Continuous integration

`.github/workflows/bench.yml` runs the full comparison with `--allocations`, through `roc-blueprint` ([Reproducible environment with roc-blueprint](#)), on `ubuntu-latest` every Monday and on demand (**Actions → Benchmarks → Run workflow**), then publishes `report.md` as the job summary and uploads the reports as the `bench-<run id>` artifact for 90 days. GitHub’s shared runners vary by a few tens of percent between runs, so compare ratios within one run rather than absolute times across runs.

Pull requests that change `package/`, `bench/` or the harness run `--quick --langs roc --fail-on-error`. It fails only when a Roc driver does not build, crashes, or exceeds its 60-second batch limit, which catches a parser that has turned quadratic without gating on noisy timings.

21.9. Reproducible environment with roc-blueprint

`Blueprint.roc` at the repository root describes the benchmark environment for `roc-blueprint` 0.3.0: Roc, Go, Rust, Python and Git from Nix, pinned by the committed `Blueprint.lock`. It declares `x86_64-linux` (CI) and `aarch64-darwin` (Apple silicon Macs), and needs only Nix with flakes enabled. Run the CLI straight from its flake, pinned to the commit of the 0.3.0 tag:

```
$ nix run github:lukewilliamboswell/roc-blueprint/db71ab518d04dc24c5fa1fe1a2423fffdc87aad5 --
run bench-quick
```

or put `blueprint` on `PATH` with `nix shell github:lukewilliamboswell/roc-blueprint/db71ab518d04dc24c5fa1fe1a2423fffdc87aad5` and run:

```
$ blueprint run bench-quick      # small corpus
$ blueprint run bench            # full comparison, with --allocations
$ blueprint run bench -- --only csv # extra arguments go to scripts/bench.py
$ blueprint run test             # scripts/all_tests.py
$ blueprint run fuzz-smoke       # scripts/run_fuzz.py smoke
$ blueprint shell                # every toolchain on PATH
```

```
$ blueprint update
```

```
# refresh Blueprint.lock
```

The first run downloads the toolchains; later runs start in a few seconds. Generated files go to `.blueprint/`, which is not committed.

Pinning. Roc is pinned to exactly the nightly in `.roc-version`: roc-overlay publishes every nightly as its own package, and `Blueprint.roc` names it as the tool `rocpkgs.<nightly tag>`.

`scripts/update_roc_nightly.py` rewrites that tag along with `.roc-version`, and a test fails if the two disagree. After a nightly update, run `blueprint update` so that `Blueprint.lock` records a roc-overlay revision that has the new nightly; until then a blueprint run fails with a missing `rocpkgs` attribute instead of using another compiler. The roc-overlay input is currently pinned to the commit of its pending pull request that adds `nightly-2026-09-29-7f11a82`; switch it back to `github:roc-lang/roc-overlay` once that is merged. The shell sets `ROC=roc`, so the scripts use the pinned compiler rather than the older one the blueprint CLI runs on.

Go, Rust and Python follow the nixpkgs revision in `Blueprint.lock`. Library versions are pinned by `Cargo.lock`, `go.sum` and `requirements.txt` as above, so a blueprint run and a plain run measure the same library code. The Python packages go into one virtual environment per interpreter, under `~/.cache/roc-parser/`.

Provenance. The blueprint shell sets `ROC_PARSER_BLUEPRINT=1`. The report's Environment line, and `metadata.environment` in `results.json`, then record the SHA-256 of `Blueprint.lock` and the nixpkgs and roc-overlay revisions it pins; outside blueprint they say `outside blueprint`. Both record the path of each toolchain, so you can check that it came from `/nix/store`.

CI. The benchmark workflow described above installs Nix with `cachix/install-nix-action`, caches the store with `magic-nix-cache-action`, and runs `blueprint run bench-quick` for pull requests and `blueprint run bench` weekly on `ubuntu-latest`, so CI and a Mac use the same pins.

Without Nix, install the toolchains yourself and run `scripts/bench.py` as shown in [Benchmarks](#); the report records every version either way.

21.10. Profile one workload

For a closer look at one parser, build its driver with debug information and sample it. On macOS, `sample` works on the optimised binaries:

```
$ "$ROC" build --opt=speed --debug bench/roc/yaml.roc --output=.roc-parser-tmp/yaml-prof
$ sample record .roc-parser-tmp/yaml-prof 1000 < .roc-parser-
tmp/bench/full/corpus/yaml/generated-medium.yaml
```

Some Roc functions keep hashed linker names even with `--debug`; `nm -n` maps recorded offsets back to symbols.

`scripts/bench_yaml.py` is an older, YAML-only harness kept for comparing two revisions of the YAML parser with `--replace-dep (--parser-root <checkout>/package)` on the YAML microbenchmark fixtures, and for its allocation counter, `fuzz/yaml-alloc.roc`. Its reports go to `.roc-parser-tmp/yaml-bench/<label>/.`

Chapter 22. Compiler updates, releases and security reports

This chapter is for maintainers with write access to the repository. It covers how the Roc compiler pin is kept current, how a release is cut and what it publishes, and how security reports are handled. It assumes [Contributing to roc-parser](#).

22.1. Roc nightly updates

Roc has no stable release yet, so roc-parser pins one nightly build in `.roc-version` and moves it forward as the compiler changes. This is automated.

The `Update Roc nightly` workflow (`update-roc-nightly.yml`) runs once a day at 13:16 UTC, about four hours after the upstream nightly build. It calls the shared controller in `roc-automation`, which:

1. writes the newest nightly tag to `.roc-version` on the reserved branch `automation/roc-nightly` and opens a pull request;
2. dispatches the validation workflows listed in `.github/roc-nightly.json` (`Tests`, `Published example compatibility`, `Release`, `Property quality tests` and `CodeQL`) on the candidate, with the `nightly_validation` input set, so that the release path runs without publishing anything; and
3. merges the pull request automatically once every check passes and the controller has rechecked the signed commit and branch protection.

A failed candidate stays open for investigation. Diagnose it: a new compiler error, a changed standard library function, or a real regression. Put the compatibility fix on a separate branch, not on `automation/roc-nightly`, which is reserved for the bot's pin-only commits.

WARNING

Do not weaken a test, gate a fuzz target or re-record a conformance baseline to make a compiler candidate pass. A changed result after a compiler update is a bug in the compiler or in the package, and needs a fix or an upstream issue.

`.github/ROC_NIGHTLY.md` records the integration details: required checks, token permissions and the pinned controller commit.

22.1.1. One compiler pin

Every workflow, including the property tests, conformance reviews, docs and benchmarks, reads `.roc-version` through `scripts/workflow_helpers.py roc-version`. A nightly candidate is therefore checked by the fuzz targets and conformance reviews as well as the tests. Do not add a second pin to a workflow; if one part of the repository needs a newer compiler, move `.roc-version` forward for everything.

22.2. Release a new version

Releases follow semantic versioning: a release that changes the result of a public function on any input, or removes or changes a public signature, is a major release.

22.2.1. Before you start

1. Confirm that `main` is green: `Tests`, `Property quality tests` and `Docs`.
2. Write the release notes as `docs/releases/<version>.md`, for example `docs/releases/2.0.0.md`, and link them from `docs/releases/index.adoc`. Put the upgrade impact first: every change a user's code or data might notice. Merge them through a normal pull request.
3. Optionally run the `ReRelease` workflow from the Actions tab with `nightly_validation` checked. It runs the whole release path, including the documentation build, without publishing.

22.2.2. Run the release

WARNING

A published release is permanent: the bundle URL is content-addressed and users pin it in their app headers. Check the version number before you start the workflow; a mistake needs a new release, not an edit.

Run the `ReRelease` workflow (`release.yml`) from the Actions tab on `main`, with `release_version` set to the new version, for example `2.0.0`. It runs these jobs:

Build release bundle

Validates the version and that the run is on the default branch, runs the Python unit tests and `scripts/all_tests.py`, checks that the version increases on the previous release, and bundles the package with `scripts/bundle.py`. It also packages the examples archive from the bundle's metadata, as a dry run of the asset the release attaches.

Test bundles

Packages the examples archive pinned to the new bundle, served locally, and checks, runs and builds every app in it.

Publish GitHub release

Creates the tag and the GitHub release, with the release notes from `docs/releases/<version>.md`, the bundle, an SPDX SBOM (`scripts/release_security_assets.py sbom`), and build-provenance and SBOM attestations, combined into one offline attestation bundle. It also attaches `roc-parser-examples-<version>.zip` (`scripts/release_helpers.py package-examples`): the examples, with their `parser` dependency pinned to the release bundle's URL, and a README saying how to run them.

Docs

Builds the manual and the API reference for the version with `scripts/build_manual.py --docs -version <version>` and `scripts/build_docs.py`, packages them with `scripts/release_helpers.py package-docs`, assembles the GitHub Pages site with `scripts/release_helpers.py assemble-pages`, and deploys it.

Nothing in the repository changes after a release. The examples in `examples/` always use the package source (`parser: "../package/main.roc"`); only the archive attached to the release points them at its URL.

22.2.3. After the release

1. Check the release page: the notes, the `.tar.zst` bundle, the examples archive, the SBOM and the attestation bundle.
2. Verify the provenance of the bundle you downloaded:

```
gh attestation verify <bundle>.tar.zst --repo lukewilliamsboswell/roc-parser
```

3. Open the published manual and API reference and check that they show the new version.

22.3. Security reports

The policy is in `SECURITY.md`: security fixes are made for `main` and the latest release, and reports arrive privately through GitHub private vulnerability reporting, never in public issues.

When a report arrives:

1. Acknowledge it within 7 calendar days and give an initial assessment within 14.
2. Reproduce it with the reporter's input. If it is a crash, hang or excessive resource use, add the input to the matching fuzz target's seeds or the module's tests in the private fix branch, as described in [Property tests and conformance reviews](#).
3. Prepare the fix in the private security advisory's temporary fork, and agree a disclosure date with the reporter. The target is coordinated disclosure within 90 days.

4. Release the fix as a new version, then publish the advisory, crediting the reporter if they agree.

Background

Chapter 23. Where the design comes from

roc-parser is not a new idea. Parser combinators have been studied and used for fifty years, and almost every choice in the `Parser` and `Utf8` modules repeats, adapts or deliberately declines something from that history. This chapter traces those ideas to their sources, so you can see why the library behaves as it does and where it differs from the libraries you may already know. It assumes you have read [Your first parser](#); you do not need it to use the package.

The examples come from `background-lineage.roc`; every `expect` in it passes.

23.1. A parser is a function

The founding idea is that a parser is an ordinary function: it takes the input, reads something from the front of it, and returns what it read together with the input that is left. Because parsers are values, functions that take parsers and return new parsers (the *combinators*) can express sequencing, choice and repetition directly, so the parser has the shape of the grammar it reads.

William Burge described such a set of combinators in his 1975 book *Recursive Programming Techniques*. Philip Wadler's 1985 paper "How to replace failure by a list of successes" showed how a lazy functional language can represent a parser's possible outcomes as a list: an empty list is failure, and several elements are the several ways the input could be read. Graham Hutton's 1992 paper "Higher-order functions for parsing" turned these ideas into a complete, teachable library and became the standard reference.

roc-parser keeps the function and drops the list. A `Parser (input, a)` wraps a function from `input` to a `Try` holding either the value and the rest of the input, `{ value, rest }`, or a `ParseError` with a message and an offset. `Parser.custom` takes such a function directly, so you can always step outside the combinators, and `Parser.run` runs a parser and returns the same shape:

```
# A parser is a function from input to a result and the rest of the input.
# `custom` wraps such a function directly.
take : U64 -> Parser(Utf8.Bytes, Utf8.Bytes)
take = |n|
  Parser.custom(
    |input|
      if input.len() >= n {
        Ok({ value: input.take_first(n), rest: input.drop_first(n) })
      } else {
```

```

        Err(ParseError({ message: "expected ${n.to_str()} more bytes", offset: 0 })))
    },
)

# Monadic style: the next parser depends on a value already read. Here a
# length prefix says how many bytes follow.
length_prefix : Parser(Utf8.Bytes, U64)
length_prefix = Utf8.digits.skip(Utf8.codeunit(':'))

counted : Parser(Utf8.Bytes, Str)
counted = length_prefix.and_then(take).map(Str.from_utf8_lossy)

expect Utf8.parse_str(counted, "3:abc") == Ok("abc")
expect Utf8.parse_str(counted, "3:ab").is_err()

```

A parser returns one result, not a list. Wadler's list of successes allows ambiguous grammars, where one input has several readings, but the formats this package reads are designed to have exactly one reading. A single result is cheaper, and it gives one clear place to put an error.

23.2. Sequencing: monads and applicatives

Running one parser and then another needs a way to combine their results. Hutton and Erik Meijer's "Monadic parser combinators" (1996) and the shorter "Monadic parsing in Haskell" (1998) observed that parsers form a *monad*: the sequencing operation, usually called `bind` or `and_then`, passes the value from the first parser to a function that chooses the second. That is what `counted` above does with `and_then`: the length it reads decides how many bytes the next parser takes.

Most of a grammar does not need that power. In "Applicative programming with effects" (2008), Conor McBride and Ross Paterson named the weaker *applicative* interface: a pure value lifted into the parser, and a way to apply a parser of functions to a parser of arguments. Their paper points to parsing as a case where monads are more than you need, citing S. Doaitse Swierstra and Luc Duponcheel's 1996 combinators, whose structure is known before any input is read. Applicative parsers cannot make the grammar depend on earlier values, and in return they read as a description of the input.

`Parser.const` with `keep` and `skip` is exactly that applicative interface, with the names Evan Czaplicki's `elm/parser` gave its pipeline operators: `|=` keeps a result and `|.` ignores one. You list the pieces of the input in order, and the curried constructor receives the ones you keep:

```

# Applicative style: a constructor function, then one 'keep' per field and
# one 'skip' per piece of punctuation. No step depends on an earlier value.
Point : { x : U64, y : U64 }

point : Parser(Utf8.Bytes, Point)
point =

```

```

Parser.const(|x| |y| { x, y })
  .skip(Utf8.codeunit('('))
  .keep(Utf8.digits)
  .skip(Utf8.codeunit(','))
  .keep(Utf8.digits)
  .skip(Utf8.codeunit(''))

expect Utf8.parse_str(point, "(3,4)") == Ok({ x: 3, y: 4 })

```

roc-parser builds `map` and `keep` on this same application step, and also offers the monadic `and_then` method for the cases that need it, such as `counted`. Prefer `keep` and `skip` when the grammar allows: a parser that uses only them can be read top to bottom as a description of its input.

23.3. Choice and backtracking

A choice between alternatives must decide what happens when the first one fails after reading part of the input. The libraries in this lineage give three answers.

Full backtracking

The next alternative starts again from the original input. Wadler's lists and Hutton's combinators work this way, and so does Bryan O'Sullivan's `attoparsec`, which keeps the whole input so that it can backtrack arbitrarily.

Committed choice

Daan Leijen and Erik Meijer's `Parsec` (2001) tries the second alternative only if the first failed *without consuming input*. Their paper gives two reasons: naive backtracking holds on to the input and leaks space, and when the first alternative has already read part of the input, its failure is usually the error worth reporting. When you do want to backtrack, you wrap the first alternative in `try`. Elm's parser takes the same default for the same reason, with `backtrackable` in place of `try`; its README uses the input `[ 1, 23zm5, 3 ]`, where the useful error is at the `z`, not at the `[`.

Ordered choice

Bryan Ford's *parsing expression grammars* (PEGs, 2004) define choice as prioritized: `A / B` tries `A`, and tries `B` from the same position only if `A` fails. Once `A` succeeds, `B` is never considered, even if the rest of the parse then fails.

roc-parser's `Parser.alt` and `Parser.one_of` are PEG ordered choice. A failed alternative is always retried from the original input, so there is no `try`, and the first success wins:

```

# Ordered choice, as in a PEG: the first alternative that succeeds wins, and
# a failed alternative is retried from the same input.
sign : Parser(Utf8.Bytes, [Plus, Minus, PlusPlus])
sign =

```

```

Parser.one_of([
  Parser.const(PlusPlus).skip(Utf8.string("++")),
  Parser.const(Plus).skip(Utf8.string("+")),
  Parser.const(Minus).skip(Utf8.string("-")),
])

expect Utf8.parse_str(sign, "++") == Ok(PlusPlus)
expect Utf8.parse_str(sign, "-") == Ok(Minus)

```

The trade-offs follow from the history. Without a commit point, you never have to decide where to put `try`, and a grammar reads the same as its PEG. The costs are the ones Parsec was designed to avoid: a grammar whose alternatives share long prefixes re-reads them, and without help a failure deep inside an early alternative would be replaced by the failures of the later ones. The furthest-failure rule described below provides that help, and [Combinators by task](#) shows how to order alternatives so that re-reading rarely matters.

23.4. Repetition that always ends

PEG repetition is greedy: `e*` matches as many `e` as it can and never gives any back. `Parser.many`, `one_or_more` and `sep_by` behave the same way, which is why a failed element ends the repetition rather than the parse.

Repeating a parser that can succeed without reading anything, such as `chomp_while` or `maybe(p)`, would loop forever. Parsec refuses at run time, with the error "combinator 'many' is applied to a parser that accepts an empty string". `roc-parser` instead stops the repetition at the first element that makes no progress, and returns what it has collected. The check compares input lengths, so it costs $O(1)$ per element and needs only the `len` method from the input type, which the `where` clause on `many` states. A Roc program should not crash on its input, so a well-defined result is better than an error that only appears for some inputs.

23.5. Error reporting

The early papers say little about errors. Parsec made them a design goal: on failure it reports the position and the set of things that would have been accepted there. Mark Karpov's [megaparsec](#), a fork of Parsec, keeps that and, when it merges the errors of two branches, prefers the one that got further into the input. Elm's parser adds context: a parser can say "I am reading a list", so the error can name what was being read and not just the row and column, which the Elm compiler uses for its error messages.

`roc-parser` adopts `megaparsec`'s rule. Parsers track the *furthest failure*, the failure that read furthest into the input, and every runner reports it as `ParseError({ message, offset })`, where `offset` is a byte offset. A repetition that stops at a bad element remembers that element's failure, so if

the parse then fails, the error points at the bad element rather than at the input left over after the repetition:

```
# Furthest failure: 'many' stops at the bad second point, but the error the
# runner reports is that point's own failure, at the byte where it happened.
points : Parser(Utf8.Bytes, List(Point))
points = Parser.many(point)

expect Utf8.parse_str(points, "(1,2)(3,x)") == Err(ParseError({ message: "Not a digit",
offset: 8 })))
```

The alternative that got furthest is almost always the one the author of the input meant, so this recovers most of the precision of committed choice while keeping ordered choice. `ParseError` is an open tag union, so `?` passes it through a function that returns other errors too.

The format modules go further on their own: `Yaml` and `Xml` report line and column, `CSV` the record and field, and `HTTP` the byte offset, each in an error tag named after the format. [Report parse errors](#) shows how to present them.

23.6. Bytes, not characters

Haskell's early combinator libraries read lists of characters. The libraries built for speed read bytes: `attoparsec` has a `ByteString` interface, and Geoffroy Couprie's `nom` for Rust describes itself as "byte-oriented" and "zero-copy". The Roc language reference gives the same advice for Roc: in [the strings chapter of the language reference](#), it recommends working in UTF-8 bytes when implementing a parser.

So the input type of the text parsers, `Utf8.Bytes`, is `List(U8)`, and `'a'` is a byte. Every format in this package is defined over ASCII delimiters, which in UTF-8 are single bytes that never appear inside a multi-byte character, so reading bytes is both correct and fast. Text is turned back into `Str` only once a whole field has been read.

23.7. Recursion and laziness

A grammar for nested data refers to itself: a list contains values, and a value can be a list. In Haskell this costs nothing, because laziness delays building a parser until it is used. Roc evaluates eagerly, so a parser that refers to itself would be built forever. `Parser.lazy` takes a function that builds the parser and calls it only when the parser runs, exactly as `elm/parser`'s `lazy : (() -> Parser a) -> Parser a` does for eager Elm. [Combinators by task](#) shows it in use.

Left recursion, where a rule starts with itself (`expr = expr "-" term`), is a different problem. A top-down parser calls the rule again before reading anything and never stops. Ford's PEG paper notes

that left recursion is not available in PEGs. Research has since shown how to support it in combinators, for example Richard Frost, Rahmatullah Hafiz and Paul Callaghan's 2008 "Parser combinators for ambiguous left-recursive grammars", and Joshua Barretto's [chumsky](#) for Rust offers left recursion and memoization as opt-in features. `roc-parser` supports neither. Write the repetition explicitly instead: a `term` followed by many of `" "` `term`.

23.8. Packrat parsing and performance

Backtracking can re-run the same parser at the same position many times, and in the worst case the time grows exponentially with the input. Ford's *packrat parsing* (2002) fixes this by memoizing every rule's result at every position, which guarantees linear time at the cost of memory proportional to the input times the number of rules.

`roc-parser` does not memoize. For the formats it targets, careful ordering of alternatives keeps backtracking shallow, and memoization would cost memory on every input to protect against grammars that are rare in practice. Where speed matters most, the package goes further: [Add a format module](#) explains that the CSV, XML, HTTP and Markdown modules moved their document-level parsing from combinators to byte scanners with an explicit stack, to stay linear in the input size and avoid deep recursion. Combinators remain the interface for composing them and for writing your own parsers.

23.9. Combinators and parser generators

The other tradition is the parser *generator*. Stephen C. Johnson's `yacc` (1975) reads a grammar file and generates an LALR(1) parser; Terence Parr's [ANTLR](#) (described by Parr and Russell Quong in 1995) generates LL(k) parsers, with its later versions using more powerful strategies. A generator analyses the whole grammar before any input is read, so it can report ambiguities and conflicts at build time, it accepts left-recursive rules, and the parsers it generates are fast.

Combinators trade that analysis for being ordinary code. The grammar is written in the host language, can be tested piece by piece with `expect`, uses the host's types, and needs no build step. You can add a combinator of your own whenever the grammar needs one, such as a parser whose next step depends on a value already read. [Choosing how to read your input](#) covers when that trade is worth making.

23.10. Testing against the specification

The newest layer in this lineage is testing. Koen Claessen and John Hughes's QuickCheck (2000) introduced property-based testing: state a property that should hold for every input, and let the tool generate inputs to try to break it. `roc-parser`'s quality process follows that approach. Fuzz

targets check properties such as "parsing never crashes", and review scripts compare every format with a mature independent implementation, its *oracle*, on generated and conformance-suite inputs. [Property tests and conformance reviews](#) describes the process.

23.11. How roc-parser compares

Idea	Source	roc-parser
Parser as a function returning the rest of the input	Burge 1975, Wadler 1985, Hutton 1992	Yes, with one result instead of a list of successes
Applicative sequencing	Swierstra and Duponcheel 1996, McBride and Paterson 2008, elm/parser	<code>const</code> , <code>keep</code> and <code>skip</code>
Monadic sequencing	Hutton and Meijer 1996 and 1998	<code>and_then</code> , and <code>custom</code> for anything else
Choice	Parsec 2001 (committed, with <code>try</code>); Ford 2004 (ordered)	Ordered choice that always backtracks; no <code>try</code>
Repetition of empty-matching parsers	Parsec raises an error	Stops at the first element that makes no progress
Error reporting	Parsec, megaparsec (longest match), elm/parser (context)	Furthest failure with a byte offset; line and column in the formats
Input type	attoparsec, nom (bytes)	UTF-8 bytes, as the Roc language reference recommends
Memoization, left recursion	Ford 2002; Frost, Hafiz and Callaghan 2008; chumsky	Not supported; write repetition explicitly
Quality	Claessen and Hughes 2000	Fuzzing, property tests and oracle comparisons

23.12. Further reading

Burge, W. H. (1975). *Recursive Programming Techniques*. Addison–Wesley. [Open Library record](#)

The first published set of parsing combinators, in a book about programming with recursion and higher-order functions. Out of print.

Wadler, P. (1985). How to replace failure by a list of successes. *Functional Programming Languages and Computer Architecture*, LNCS 201, 113–128. doi:10.1007/3-540-15975-4_33

Represents failure and backtracking with lazy lists of results; the technique behind the first functional parser libraries.

Hutton, G. (1992). Higher-order functions for parsing. *Journal of Functional Programming*, 2(3), 323–343. doi:10.1017/S0956796800000411

The classic introduction to combinator parsing, building sequencing, choice and repetition from a handful of primitives.

Hutton, G., and Meijer, E. (1996). *Monadic parser combinators*. Technical report NOTTCS-TR-96-4, University of Nottingham. [PDF on Graham Hutton’s site](#)

A long tutorial that recasts combinator parsing in monadic style; its introduction surveys the earlier work.

Hutton, G., and Meijer, E. (1998). Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 437–444. doi:10.1017/S0956796898003050

The short, widely taught version of the same ideas.

Swierstra, S. D., and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. *Advanced Functional Programming*, LNCS 1129, 184–207. doi:10.1007/3-540-61628-4_7

Non-monadic combinators whose fixed structure allows analysis and error correction; the early example of applicative parsing.

Leijen, D., and Meijer, E. (2001). *Parsec: Direct style monadic parser combinators for the real world*. Technical report UU-CS-2001-27, Utrecht University. [Microsoft Research publication page](#)

Committed choice, `try`, and error messages that name the position and the expected input. The library is on Hackage as [parsec](#).

McBride, C., and Paterson, R. (2008). Applicative programming with effects. *Journal of Functional Programming*, 18(1), 1–13. doi:10.1017/S0956796807006326

Defines applicative functors and names parsing as a case that needs less than a monad; the theory behind `keep` and `skip`.

Ford, B. (2002). Packrat parsing: simple, powerful, lazy, linear time. *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 36–47. doi:10.1145/581478.581483

Memoizes a backtracking parser to guarantee linear time.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 111–122. doi:10.1145/964001.964011

Prioritized choice and greedy repetition as a grammar formalism; the closest formal model of roc-parser's `one_of` and `many`.

Frost, R. A., Hafiz, R., and Callaghan, P. (2008). Parser combinators for ambiguous left-recursive grammars. *Practical Aspects of Declarative Languages (PADL)*, LNCS 4902, 167–181. doi:10.1007/978-3-540-77442-6_12

One way to support left recursion in combinators, the feature roc-parser leaves out.

Claessen, K., and Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 268–279. doi:10.1145/351240.351266

Property-based testing, the basis of this package's fuzz and property tests.

Johnson, S. C. (1975). *Yacc: Yet Another Compiler-Compiler*. Bell Laboratories. [Online copy](#)

The parser generator that set the pattern for grammar files compiled to LALR(1) tables.

Parr, T. J., and Quong, R. W. (1995). ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7), 789–810. doi:10.1002/spe.4380250705

The first journal description of ANTLR, a widely used LL parser generator.

Libraries

- [elm/parser](#) by Evan Czaplicki: parser pipelines (`|=` and `|.`), no backtracking by default, and context in error messages. Roc is, in the words of its [FAQ](#), "a direct descendant of Elm".
- [megaparsec](#): Parsec's successor, with typed errors and longest-match error merging.
- [attoparsec](#): fast parsing of bytes and text with arbitrary backtracking and incremental input.
- [nom](#), [winnow](#) (a fork of nom) and [chumsky](#): Rust combinator libraries. nom is byte-oriented and zero-copy, winnow is a fork of nom that keeps the fundamentals in one crate, and chumsky focuses on error recovery.

Release notes

Chapter 24. Release notes

Each release has a notes file in `docs/releases/`, written in Markdown so the release workflow can publish it unchanged as the GitHub release description. The notes start with the upgrade impact: every change your code or data might notice.

- [roc-parser 2.0.0 release notes](#): a redesigned API with an upgrade guide, type-directed decoding of CSV and YAML into records, CommonMark and GFM Markdown, strict RFC 9112 HTTP, XML 1.0 well-formedness, the YAML 1.2 core schema, RFC 4180 CSV, and property tests and conformance reviews for every format.

Earlier releases are described on the [roc-parser GitHub releases page](#).