

# KLEENE

## User guide

v0.1.0      2026-02-16      MIT

PEG-based parser combinator

NEVEN VILLANI

✉ [neven@crans.org](mailto:neven@crans.org)

**KLEENE** is a parser combinator based on Parsing Expression Grammars (PEG), that places a particular focus on **modularity**, **quality of error messages**, and **unit testing**. Parsers generated by **KLEENE** can be modified on the fly or extended with new rules, have built-in well-formatted error reporting, and are easy to test.

**KLEENE** is **not** focused on performance above all else: PEGs interpreted by recursive descent are known to exhibit exponential-time worst-case parsing. If the parsing work you need is very time-sensitive, consider using a more specialized library.

### Contributions

If you have ideas for improvements, or if you encounter a bug, you are encouraged to contribute to **KLEENE** by submitting a [bug report](#), [feature request](#), or [pull request](#). This includes submitting test cases.

### Versions

- [dev](#)
- [0.1.0 \(latest\)](#)
- [...](#)

# Table of Contents

<b>Quick start .....</b>	<b>3</b>
I.1 Skeleton .....	3
I.2 Simple examples .....	4
I.2.1 Decimal integers .....	4
I.2.2 Integers .....	6
I.2.3 Arithmetic expressions .....	6
<b>Operators .....</b>	<b>7</b>
<b>Unit tests .....</b>	<b>8</b>
<b>Modularity .....</b>	<b>9</b>

Highlighted chapters denote breaking changes<sup>(↳)</sup>, major updates<sup>(↑)</sup>, minor updates<sup>(↑)</sup>, and new additions<sup>(+)</sup>, in the latest version 0.1.0

# Part I

## Quick start

### I.1 Skeleton

At minimum, using `KLEENE` takes the following form:

```
#import "@preview/kleene:0.1.0"

#let rules = {

    // kleene.prelude contains all the operators, which we usually
    // don't want to have polluting the global namespace.
    import kleene.prelude: *

    grammar(
        // define the rules here
        main: {
            ...
        },
        ...
    )
}

// If the grammar includes any unit tests, this will evaluate them.
#kleene.test(rules)

// This is how you invoke the parser, with <main> as the entry point.
#kleene.parse(rules, <main>, "...")
```

For example, here is a very simple grammar that will simply look for the string "main":

```

#let rules = kleene.grammar(
    // This defines a rule <main>
    main: {
        // Each `pat` gives one possible form of the match, here the
        // literal string "main"
        pat("foo")
    }
)

// In case of success, the result is a pair (true, value)
#let (ok, ans) = kleene.parse(rules, <main>, "foo")
#assert(ok)
#ans

// In case of failure, the result is a pair (false, error-message)
#let (ok, ans) = kleene.parse(rules, <main>, "bar")
#assert(not ok)
#ans

```

foo

**Parsing error:** The input does not match the expected format.

1 | bar

  ^ Can't match string "foo"

While trying to parse: <main>.

## I.2 Simple examples

### I.2.1 Decimal integers

To start with an easy example, let's write a parser of integers in base 10. We take the opportunity to demonstrate the use of `kleene.test` to run the inline unit tests declared by `yy` and `nn`, as well as `tr` to transform the parsed result post-matching.

```
#let rules = kleene.grammar(
    digit: {
        pat(range("0", "9"))
    },
    digits: {
        // Using the <label> notation you can recursively refer to other rules
        pat(iter(<digit>))
        // `tr` applies a post-parsing transformation to the data.
        // By default, `repeat` produces an array.
        tr(ds => ds.join())
    },
    int: {
        pat(<digits>)
        tr(ds => int(ds))
        // The command `yy` adds positive examples
        yy(`42`, `4096`)
        // The command `nn` adds negative examples
        nn(`42b`, ``)
    },
)
```

int		examples
42		42
4096		4096

int		counterexamples
42b		<p><b>Parsing error:</b> The parser did not consume the entire input.</p> <pre>1   42b   ~ ^ Surplus characters     Valid &lt;int&gt;</pre> <p>Hint: halted due to the following:</p> <pre>1   42b       ^ Character is out of range</pre> <p>While trying to parse: &lt;int&gt; → &lt;digits&gt; → &lt;digit&gt;.</p>
		<p><b>Parsing error:</b> The input does not match the expected format.</p> <pre>1   ∅       ^ End of input stream</pre> <p>While trying to parse: &lt;int&gt; → &lt;digits&gt; → &lt;digit&gt;.</p>

## I.2.2 Integers

`KLEENE` can backtrack, and the backtracking points are indicated by the operator `fork`. Several possible sub-patterns can be given, and they will be explored sequentially until a match. This lets you define a rule as the union of other rules. To illustrate this, we add hexadecimal numbers to our parser.

```
#let rules = kleene.grammar(
    decdigit: pat(range("0", "9")),
    hexdigit: {
        // `fork` introduces a backtracking point
        pat(fork(range("a", "f"), range("A", "F")))
        // multiple `pat` invocations also implicitly induce a `fork`
        pat(<decdigit>)
    },
    decint: {
        pat(iter(<decdigit>))
        tr(ds => (int: int(ds.join())))
    },
    hexint: {
        // If some part of the input is matched but not used,
        // such as here the prefix "0x", we indicate this with a `drop`
        // and it will be removed before the transformation is applied.
        pat(drop("0x"), iter(<hexdigit>))
        tr(ds => (hex: ds.flatten().join()))
    },
    value: {
        pat(fork(<hexint>, <decint>))
        yy(`42`, `0xfae`)
    }
)
```

value		examples
42		(int: 42)
0xfae		(hex: "fae")

## I.2.3 Arithmetic expressions

# **Part II**

## **Operators**

# **Part III**

## **Unit tests**

# **Part IV**

## **Modularity**