KLEENE*

User guide

v0.1.0 2026-02-19 MIT

PEG-based parser combinator

Neven Villani
neven@crans.org

KLEENE is a parser combinator based on Parsing Expression Grammars (PEG), that places a particular focus on **accessibility**, **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 or a non-native parser (Section VII.1).

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

Part I

Quick start

I.1 Skeleton

At minimum, using KLEENE takes the following form:

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

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

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

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

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

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

```
#let grammar = 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(grammar, <main>, "foo")
```

```
#assert(ok)
#ans

// In case of failure, the result is a pair (false, error-message)
#let (ok, ans) = kleene.parse(grammar, <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 `#prelude.yy` and `#prelude.nn`, as well as `#prelude.rw` to transform the parsed result post-matching.

```
#let grammar = kleene.grammar(
  digit: {
    pat(`[0-9]`)
  },
  digits: {
    // Using the <label> notation you can recursively refer to other rules
    pat(iter(<digit>))
    // `rw`, standing for "rewrite" applies a post-parsing transformation
    // to the data. By default, `iter` produces an array.
    rw(ds => ds.join())
  },
  int: {
    pat(<digits>)
    rw(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`		**Parsing error:** The parser did not consume the entire input.
		`1 | 42b`
		`~~|^` Surplus characters
		`| Valid <int>`
		Hint: halted due to the following:
		`1 | 42b`
		`^` Regex does not match
		While trying to parse: `<int>` → `<digits>` → `<digit>`.

`⌀`		**Parsing error:** The input does not match the expected format. `1 | ⌀` `^` Regex does not match While trying to parse: `<int>` → `<digits>` → `<digit>`.

I.2.2 Integers

KLEENE can backtrack, and the backtracking points are indicated by the operator `#prelude.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 grammar = kleene.grammar(
  decdigit: pat(`[0-9]`),
  hexdigit: {
    // `fork` introduces a backtracking point
    pat(fork(`[a-f]`, `[A-F]`))
    // multiple `pat` invocations also implicitly induce a `fork`
    pat(<decdigit>)
  },
  decint: {
    pat(iter(<decdigit>))
    rw(ds => 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>))
    rw(ds => ds.flatten().join())
  },
  value: {
```

```
    pat(<hexint>)
    rw(i => (hex: i)) // a `rw` applies to all the `pat` that come before
    pat(<decint>)
    rw(i => (int: i))
    yy(`42`, `0xfae`)
  },
  value-bad: {
    // Note that the order of rules can be significant.
    // Here, a <hexint> has as prefix a <decint> (0),
    // so putting them in the wrong order will cause problems because
    // the parser will think it has a successful <decint> match and then
    // stop before the end.
    pat(fork(<decint>, <hexint>))
    nn(`0xfae`)
  },
)
```

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

value-bad		**counterexamples**

`0xfae`		**Parsing error:** The parser did not consume the entire input.
		`1 | 0xfae`
		`~|^^^^` Surplus characters
		`|` Valid `<value-bad>`
		Hint: halted due to the following:
		`1 | 0xfae`
		`^` Regex does not match
		While trying to parse: `<value-bad>` → `<decint>` → `<decdigit>`.

Part II

Operators

II.1 Constants

II.1.1 EOF

`eof()` matches the end of the stream and returns `none`.

```
#let grammar = kleene.grammar(
  main: {
    pat(eof())
    yy(``)
    nn(`x`)
  },
)
```

main		examples
∅		

main		counterexamples
x		**Parsing error:** The input does not match the expected format. 1 | x ^ Expected end of stream While trying to parse: <main>.

II.1.2 Str

`str("..")` matches the literal string `".."` and returns the match. Any `str` is implicitly cast to `str`.

```
#let grammar = kleene.grammar(
  main: {
    pat("foo")
    yy(`foo`)
    nn(`bar`)
  },
)
```

main		examples
foo		foo

main		counterexamples
`bar`		**Parsing error:** The input does not match the expected format. 1 | bar ^ Can't match string "foo" While trying to parse: <main>.

II.1.3 Regex

`regex("..")`, with `".."` the same string you would pass to the standard function `std.regex`, matches a regular expression. Any raw text (i.e., wrapped in `` `..` ``) is implicitly cast to `regex`.

```
#let grammar = kleene.grammar(
  main: {
    pat(`[a-zA-Z]+`)
    yy(`Word`)
    nn(`w0rd`)
  }
)
```

main		examples
`Word`		Word

main		counterexamples
`w0rd`		**Parsing error:** The parser did not consume the entire input. 1 | w0rd ~|^^^ Surplus characters | Valid <main> Hint: halted due to the following: 1 | w0rd ^ No longer part of the regex match While trying to parse: <main>.

II.1.4 Label

`label("lab")` recursively matches the rule named `lab`, which must be defined in the same grammar. Any `label` is implicitly cast to `label`.

```
#let grammar = kleene.grammar(
  sub: pat("foo"),
  main: {
    pat(<sub>)
    yy(`foo`)
```

```
  },
)
```

main		examples
`foo`		foo

II.2 Sequencing and branching

II.2.1 Seq

`seq(pattern1, pattern2, ..)` matches the successive patterns `pattern1`, `pattern2`, … one after the other. Any `array` is implicitly cast to a `seq`. Furthermore most other operators (`iter`, `star`, `maybe`, `drop`, `rewrite`, …) are variadic and accept multiple arguments that are understood as a `seq`.

> An **explicit** call to `seq` will always return an array, even if it has only one element. Implicit calls however, such as those induced by variadic `pat`, `iter`, … may for convenience return a single element rather than a singleton array. In both cases the array returned by `seq` is pre-filtered to exclude elements that are `none` (either `eof()` matches, or those that were dropped as explained in Section II.4).

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => int(ds))
  },
  pair: {
    pat("(", <int>, ",", <int>, ")")
    yy(`(42,64)`)
    nn(`(42,`)
  },
)
```

pair		examples
`(42,64)`		`("(", 42, ",", 64, ")")`

pair		counterexamples

`(42,`		**Parsing error:** The input does not match the expected format.
		`1 | (42,`
		`^ Regex does not match`
		While trying to parse: `<pair> → <int>`.

II.2.2 Fork

`fork(pattern1, pattern2, ..)` is the standard dual of `seq`, returning the first of `pattern`, `pattern2`, … that has a successful match.

> `fork` always returns the **first match** in the order provided, rather than the longest match. As such `fork` does not commute.

> `fork` is the only operator that may return objects of different types. It is recommended that you insert the appropriate `rw` calls to be able to identify the provenance of the result.

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    // The rewrites will help us identify the provenance of the match
    rw(ds => (int: int(ds)))
  },
  float: {
    pat(`[0-9]+`, ".", `[0-9]+`)
    rw(ds => (float: float(ds.flatten().join())))
  },
  number: {
    pat(fork(<float>, <int>))
    yy(`42`, `12.1`)
  },
  // Beware: when a rule is a valid prefix of another,
  // swapping them in the `fork` may lead to failures.
  number-bad: {
    pat(fork(<int>, <float>))
    nn(`12.1`)
  }
)
```

number		examples
`42`		`(int: 42)`
`12.1`		`(float: 12.1)`

number-bad		counterexamples

`12.1`		**Parsing error:** The parser did not consume the entire input.
		`1 | 12.1`
		`~~|^^ Surplus characters`
		`| Valid <number-bad>`
		Hint: halted due to the following:
		`1 | 12.1`
		`^ No longer part of the regex match`
		While trying to parse: `<number-bad>` → `<int>`.

II.3 Repetitions

II.3.1 Maybe

`maybe(pattern)` returns either a match of `pattern` or a unit `()`, i.e. 0 or 1 matches of another pattern. If `maybe` is given multiple parameters, they will be wrapped in a `seq`. It is equivalent to `?` in regular expressions.

> In the case of a match, the result will be wrapped in a singleton array. This follows a design convention of KLEENE that a given operator should always return an object of the same type. It is further motivated by the observed usage of `maybe`, which is commonly to match one more optional element of an already repeating sequence: it is much easier to append to an array an optional element `elt` when it is presented as `(elt,)` or `()` than if it were as `elt` or `()`.

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => int(ds))
  },
  optional: {
    pat(maybe(<int>))
    yy(`42`, ``)
  }
)
```

optional		examples
`42`		`(42,)`
`∅`		`()`

II.3.2 Iter

`iter(pattern)` matches one or more repetitions of `pattern`. If `iter` is given multiple parameters, they will be wrapped in a `seq`. It is equivalent to `+` in regular expressions.

The result of an `iter` is always an array, even if there is only one match.

```
#let grammar = kleene.grammar(
  digit: pat(`[0-9]`),
  int: {
    pat(iter(<digit>))
    rw(ds => int(ds.join()))
    yy(`420`)
    nn(``)
  }
)
```

int		examples
`420`		420

int		counterexamples

`ø`		**Parsing error:** The input does not match the expected format.
		`1 | ø`
		`^ Regex does not match`
		While trying to parse: `<int>` → `<digit>`.

`iter` is sophisticated enough to not get trapped in an infinite loop if you give a pattern that risks consuming no input. In that case it will stop after one iteration.

```
#let grammar = kleene.grammar(
  emp: {
    pat("[", star(""), "]")
    yy(`[]`)
  },
)
```

emp		examples
`[]`		`("[", (""), "]")`

II.3.3 Star

`star(pattern)` matches arbitrarily many repetitions of `pattern`, including 0. If `star` is given multiple parameters, they will be wrapped in a `seq`. It is equivalent to `*` in regular expressions.

The result of a `star` is always an array, even if there is only one match.

```
#let grammar = kleene.grammar(
  whitespace: {
    pat(iter(fork(" ", "\n", "\t")))
    rw(none)
  },
  comment: {
    pat("//", star(`[^\n]`), "\n")
    pat("/*", star(`[^*]`), "*/")
    rw(none)
  },
  irrelevant: {
    pat(star(fork(<whitespace>, <comment>)))
    rw(none)
    yy(```

      // a comment

      //
        /* another */



    ```)
  }
)
```

| irrelevant | | examples |
|---|---|---|
| ␣␣␣¤<br>␣␣//␣a␣comment¤<br>¤<br>␣␣//¤<br>␣␣␣␣/*␣another␣*/␣␣¤<br>¤<br>␣¤<br>␣␣␣ | | |

# II.4 Ghost operators

These operators do not consume text or produce outputs, but they can manipulate how the parser behaves, transform data, and sometimes improve error messages.

## II.4.1 Drop

`drop(pattern)` matches `pattern` but does not include its outcome in the result. It can be useful to have less tuple unpacking in `rw`. If `drop` receives multiple patterns it will implicitly wrap them in a `seq`.
```

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => int(ds))
  },
  tuple: {
    pat(drop("("), star(<int>, drop(",")), maybe(<int>), drop(")"))
    rw(l => l.flatten())
    yy(`()`, `(1,)`, `(1,2,3)`, `(1,2,3,)`)
  },
)
```

tuple		examples
`()`		`()`
`(1,)`		`(1,)`
`(1,2,3)`		`(1, 2, 3)`
`(1,2,3,)`		`(1, 2, 3)`

II.4.2 Rewrite

`rewrite(fun)(pattern)` matches `pattern` then applies the transformation `fun`. This can also be achieved with a `rw`, but `rewrite` is inlined. If multiple patterns are provided, they are implicitly wrapped in a `seq`.

```
#let grammar = kleene.grammar(
  int: {
    pat(rewrite(ds => (int: int(ds)))(`[0-9]+`))
    yy(`42`)
  },
)
```

int		examples
`42`		`(int: 42)`

II.4.3 Commit

`commit()`, also written `$$`, blocks backtracking. This can be very useful to improve both the performance of the parser and the quality of error messages.

Without `commit`, a grammar could look like this:

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => (int: int(ds)))
  },
  hex: {
    pat(drop("0x"), `[0-9A-Fa-f]+`)
    rw(ds => (hex: ds))
  },
  val: {
    pat(fork(<hex>, <int>))
    yy(`42`, `0xFF`)
    nn(`0xZ`)
  },
)
```

val		examples
`42`		`(int: 42)`
`0xFF`		`(hex: "FF")`

val		counterexamples

`0xZ`		**Parsing error:** The parser did not consume the entire input.
		`1 | 0xZ`
		`~|^^ Surplus characters`
		`  | Valid <val>`
		Hint: halted due to the following:
		`1 | 0xZ`
		`  ^ No longer part of the regex match`
		While trying to parse: `<val>` → `<int>`.

First we try to parse a `<hex>`, but this fails due to `"z"` so the `fork` goes to the second branch where we try to pars an `<int>`. This also fails, but for a different reason. The error message of `fork` takes the last failure, i.e. the one where we fail to parse an `<int>`.

Consider instead an alternative that uses a commit point to block the backtracking once we've read the hexadecimal prefix `"0x"`:

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => (int: int(ds)))
  },
  hex: {
    // Once we see "0x", we know it's supposed to be a hexadecimal number.
    pat(drop("0x"), $$, `[0-9A-Fa-f]+`)
```

```
    rw(ds => (hex: ds))
  },
  val: {
    pat(fork(<hex>, <int>))
    yy(`42`, `0xFF`)
    nn(`0xZ`)
  },
)
```

val		examples
`42`		`(int: 42)`
`0xFF`		`(hex: "FF")`

val		counterexamples

`0xZ`		**Parsing error:** The input does not match the expected format.
		`1 | 0xZ`
		`^ Regex does not match`
		While trying to parse: `<val>` → `<hex>`.

The new error message much more accurately pinpoints the issue.

II.4.4 Try

`try(pattern)` is the inverse of `commit()`, restoring the ability to backtrack.

II.5 Lookaheads

Lookaheads perform a match, but do not actually consume the input stream. This is only useful when the lookahead is followed by a pattern that actually does consume input, and the lookahead only serves as a guard.

> As usual, lookaheads come with advantages and drawbacks depending on if you use them correctly. They might improve your error messages and reduce the amount of backtracking required,… or they might lead to worst-case exponential-time parsing.

II.5.1 Peek

`peek(pat)` introduces a positive lookahead: it will try to match `pat`, but even if it succeeds it will not consume any input. If multiple patterns are provided, they are implicitly cast to a `#prelude.seq`. For example here is an example where a lookahead determines if we parse a string as a key or as a value:

```
#let grammar = kleene.grammar(
  string: pat(drop("\""), `[^"]*`, drop("\"")),
```

```
  ident: pat(`[a-zA-Z][a-zA-Z0-9]*`),
  named-arg: {
    pat(fork(<string>, <ident>), drop(` *: *`), <string>)
    rw(((k,v),) => (key: k, val: v))
    yy(`foo: "foo"`)
    yy(`"bar": "bar"`)
  },
  pos-arg: {
    pat(<string>)
    rw(v => (val: v))
    yy(`"baz"`)
  },
  arg: {
    // Read as: once we know that there is a ":" ahead,
    // we commit to parsing a <named-arg> without possibility
    // of backtracking anymore.
    pat(peek(`[^:,]*:`), $$, <named-arg>)
    pat(<pos-arg>)
    yy(`foo: "foo"`, `"bar": "bar"`, `"baz"`)
    nn(`"foo":`, `foo:`)
  },
  arg-worse: {
    // Contrast the quality of the error message above
    // compared to if we omit the lookahead:
    pat(<named-arg>)
    pat(<pos-arg>)
    nn(`"foo":`, `foo:`)
  },
)
```

named-arg		examples
`foo: "foo"`		(key: "foo", val: "foo")
`"bar": "bar"`		(key: "bar", val: "bar")

pos-arg		examples
`"baz"`		(val: "baz")

arg		examples
`foo: "foo"`		(key: "foo", val: "foo")
`"bar": "bar"`		(key: "bar", val: "bar")
`"baz"`		(val: "baz")

arg		counterexamples
`"foo":`		**Parsing error:** The input does not match the expected format. 1 | "foo": ^ Can't match string "“" While trying to parse: <arg> → <named-arg> → <string>.
`foo:`		**Parsing error:** The input does not match the expected format. 1 | foo: ^ Can't match string "“" While trying to parse: <arg> → <named-arg> → <string>.

arg-worse		counterexamples
`"foo":`		**Parsing error:** The parser did not consume the entire input. 1 | "foo": ~~~~~|^ Surplus characters | Valid <arg-worse> Hint: halted due to the following: 1 | "foo": ^ No longer part of the regex match While trying to parse: <arg-worse> → <pos-arg> → <string>.
`foo:`		**Parsing error:** The input does not match the expected format. 1 | foo: ^ Can't match string "“" While trying to parse: <arg-worse> → <pos-arg> → <string>.

II.5.2 Neg

Symmetrically `neg(pat)` will perform a negative lookahead, succeeding without consuming input when the inner pattern fails. If multiple patterns are provided, they are implicitly cast to a `#prelude.seq`.

```
#let grammar = kleene.grammar(
  int: {
    pat(neg("0x"), `[0-9]`, $$, `[0-9]*`)
    rw(ds => (int: int(ds.flatten().join())))
  },
  hex: {
    pat(drop("0x"), `[0-9a-fA-F]+`)
    rw(hex => (hex: hex))
  },
  val: {
    pat(fork(<int>, <hex>))
    yy(`42`, `0`, `0x31`)
    nn(`0b1`)
  }
)
```

val		examples
`42`		`(int: 42)`
`0`		`(int: 0)`
`0x31`		`(hex: "31")`

val		counterexamples

`0b1`		**Parsing error:** The parser did not consume the entire input.
		`1 | 0b1`
		`~|^^ Surplus characters`
		`| Valid <val>`
		Hint: halted due to the following:
		`1 | 0b1`
		`^ No longer part of the regex match`
		While trying to parse: `<val>` → `<int>`.

Part III

Unit tests

Many have already been included in previous examples, but to reiterate:

- `yy(input1, input2, ..)` declares as many **positive** unit tests as there are inputs, i.e. they should succeed.
- `nn(input1, input2, ..)` declares as many **negative** unit tests as there are inputs, i.e. they should fail.

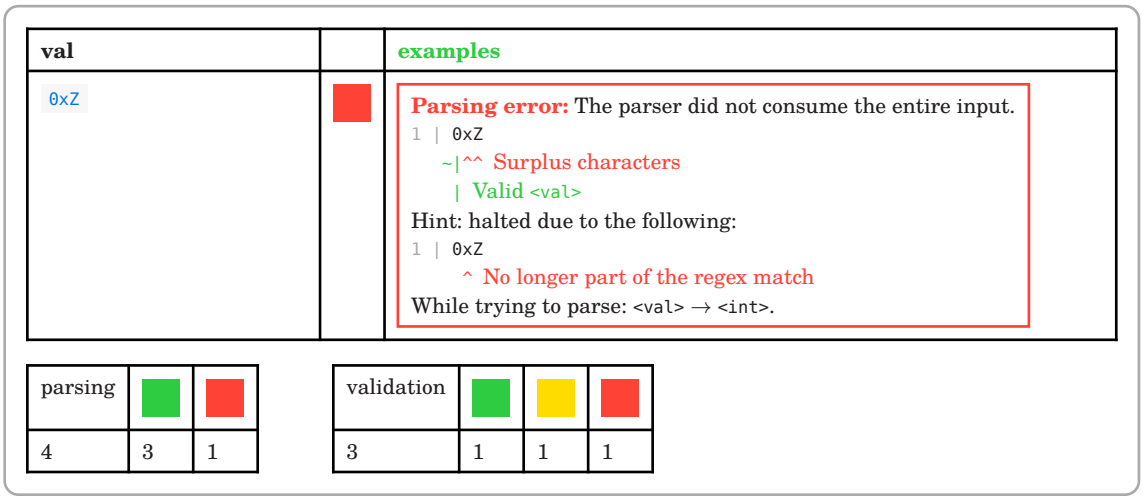
III.1 Post-parsing validation

In addition to checking that positive tests parse and negative tests do not, you may specify an additional `validate` parameter in the form of a function that will receive the input of the test and the parsed output. This function should return `none` if the result is valid, and anything else will cause the test to fail.

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => (int: int(ds)))
  },
  hex: {
    pat(drop("0x"), `[0-9A-Fa-f]+`)
    rw(ds => (hex: ds))
  },
  val: {
    pat(fork(<hex>, <int>))
    yy(`0`)
    yy(`42`, `0xFF`, `0xZ`,
      validate: (_,ans) => if "int" not in ans [Integer expected])
  },
)

#kleene.test(grammar)
```

val		examples
`0`		`(int: 0)`
`42`		`(int: 42)`
`0xFF`		`(hex: "FF")`
		Validation failed: Integer expected



A tally of all test outcomes is added to the bottom, showing from left to right

- the total number of tests
- positive tests that parse and negative tests that fail,
- positive tests that fail and negative tests that parse,
- number of tests that have a `validate` parameter, and of those:
- how many passed validation,
- how many were skipped due to an incorrect result in the parsing stage,
- and how many failed validation,

Part IV

Modularity

The first way in which KLEENE is modular is that you can access any intermediate rules.

```
#let grammar = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => (int: int(ds)))
  },
  hex: {
    pat(drop("0x"), `[0-9A-Fa-f]+`)
    rw(ds => (hex: ds))
  },
  val: {
    pat(fork(<hex>, <int>))
  },
)

#kleene.parse(grammar, <int>, "42")

#kleene.parse(grammar, <hex>, "0xDEAD")
```

```
(true, (int: 42))
(true, (hex: "DEAD"))
```

This alone is not the focus of this section, however.

IV.1 Extend

`kleene.extend(grammar1, grammar2)` produces a new grammar that has the combined rules of `grammar1` and `grammar2`, in the following manner:

- rules that exist in `grammar1` but not `grammar2` are left unchanged,
- so are rules that exist in `grammar2` but not `grammar1`,
- rules that exist in both are concatenated in a way that gives `grammar1` precedence,
- unit test sets are merged.

```
#let grammar1 = kleene.grammar(
  hex: {
    pat(drop("0x"), $$, `[0-9A-Fa-f]+`)
    rw(ds => (hex: ds))
    yy(`0x42`)
  },
  val: {
```

```
    pat(<hex>)
    yy(`0x42`)
  }
)

#let grammar2 = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => (int: int(ds)))
    yy(`42`)
  },
  val: {
    pat(<int>)
    yy(`42`)
  }
)

#let grammar = kleene.extend(grammar1, grammar2)
```

Extending a grammar in this way can cause negative tests to fail. You can use `kleene.strip(grammar, yy: false)` to remove only negative unit tests from the extended grammar, or `kleene.strip(grammar)` to remove all unit tests.

IV.2 Patch

This operation is similar to `kleene.extend`, but rather than adding to the existing rules, `kleene.patch(grammar1, grammar2)` will keep only `grammar2`'s version of rules that exist in both. This includes removing all unit tests from the rules of `grammar1` that are also declared in `grammar2`.

The example below shows an application, where we have a grammar that parses lists of integers, and we patch it into a grammar that parses lists of booleans.

```
#let grammar1 = kleene.grammar(
  int: {
    pat(`[0-9]+`)
    rw(ds => int(ds))
    yy(`42`)
  },
  comma: pat(drop(` *, *`)),
  elem: pat(<int>),
  list: {
    pat(drop("["), star(<elem>, <comma>), maybe(<elem>), drop("]"))
    rw(elts => elts.flatten())
    yy(`[]`, `[1]`, `[1,]`, `[1, 2, 3, 4]`)
  }
)
```

```
#let grammar2 = kleene.grammar(
  // <bool> is a new rule (be careful of accidental collisions)
  bool: {
    pat(fork("true", "false"))
    rw(e => ("true": true, "false": false).at(e))
    yy(`true`, `false`)
  },
  // <elem> will replace the original one
  elem: pat(<bool>),
  // We re-specify <list> but without a `pat`,
  // just to replace its unit tests that wouldn't work anymore.
  list: {
    yy(`[true, false, false]`)
  }
)

#let grammar = kleene.patch(grammar1, grammar2)
```

int		**examples**
`42`		42

bool		**examples**
`true`		true
`false`		false

list		**examples**
`[true, false, false]`		(true, false, false)

Part V

Advanced techniques

V.1 Operator precedence

As is standard in PEGs, there is no notion of operator precedence. This can make it more challenging to parse e.g. arithmetic expressions, but there are standard tricks to escape this limitation. The usual way of having a form of precedence is simply to have multiple levels of expressions, as below:

```
#let grammar = kleene.grammar(
  whitespace: {
    pat(maybe(iter(" ")))
    rw(none)
  },
  int: {
    pat(iter(`[0-9]`))
    rw(ds => int(ds.join()))
  },
  atom: {
    pat(<int>)
    rw(i => i)
    pat(drop("("), <expr>, drop(")"))
    rw((e,) => e)
    yy(`11`)
  },
  mul-expr: {
    pat(<atom>, iter(<whitespace>, fork("*", "/"), <whitespace>, <atom>))
    rw(elts => (mul: elts.flatten()))
    pat(<atom>)
    rw(auto)
    yy(`42`, `11 * 12 * 4`)
  },
  add-expr: {
    pat(<mul-expr>, iter(<whitespace>, fork("+", "-"), <whitespace>, <mul-
expr>))
    rw(elts => (add: elts.flatten()))
    pat(<mul-expr>)
    rw(auto)
    yy(`1 + 1 * 2 - 1`)
  },
  expr: {
    pat(<add-expr>)
    yy(`1 + (1 + 1) * 2`)
  },
)
```

atom		**examples**
`11`		11

mul-expr		**examples**
`42`		42
`11␣*␣12␣*␣4`		`(mul: (11, "*", 12, "*", 4))`

add-expr		**examples**
`1␣+␣1␣*␣2␣-␣1`		`(add: (1, "+", (mul: (1, "*", 2)), "-", 1))`

expr		**examples**
`1␣+␣(1␣+␣1)␣*␣2`		`(add: (1, "+", (mul: ((add: (1, "+", 1)), "*", 2))))`

As a related limitation, PEG grammars are not suited to left-recursive rules (when the rule appears immediately at the left of itself without consuming any input). Prioritize using iteration instead.

Part VI
Public API

VI.1 Toplevel definitions

Unless explicitly marked `private`, the functions in this module are available directly as `kleene.function(..)`.

VI.1.1 Parsing

`#kleene.auto-cast` `#kleene.parse`

Private

`#kleene.auto-cast((pat)) → pattern`

Automatically reinterprets builtin types/values to operators.

Argument

`⟨pat⟩` `any`

Pattern to interpret.

- `function`: native
- `str`: cast through `#prelude.str`
- `label`: cast through `#prelude.label`
- `array`: cast through `#prelude.seq`
- `$$`: shorthand for `#prelude.commit`
- `#raw`: cast through `#prelude.regex`

`#kleene.parse((rules), (pat), (input)) → (bool, result)`

Initiate parsing. Returns a boolean and a result. The boolean indicates if the parsing is successful. It also determines the type of the result:

- whatever type is returned by the last rewriting function in the case of a success,
- content that can be directly displayed for an error message in the case of a failure.

Argument

`⟨rules⟩` `grammar`

Typically constructed by `#kleene.grammar`.

Argument

`⟨pat⟩` `label`

Indicates the entry point for the parsing.

Argument

`⟨input⟩` `str`

Input data to parse.

VI.1.2 Grammar builders

`#kleene.extend((g1), (g2))` → `grammar`

Edits a grammar by adding new rules and new cases to existing rules.

See: Section IV.1.

Argument

`(g1)` `grammar`

Base grammar, takes precedence on rules that are defined by both.

Argument

`(g2)` `grammar`

New rules and cases.

`#kleene.grammar(..(rules))` → `grammar`

Constructs a new grammar from its rules.

Argument

`..(rules)` `rule`

List of named rules, as constructed by `#prelude.pat` and its related functions.

`#kleene.patch((g1), (g2))` → `grammar`

Edits a grammar by substituting old rules for new ones.

See: Section IV.2.

Argument

`(g1)` `grammar`

Base grammar.

Argument

`(g2)` `grammar`

Patch, replaces rules already defined by `g1`.

`#kleene.strip((g1), (yy): true, (nn): true)` → `grammar`

Removes unit tests from a grammar. Particularly useful when patching a grammar to remove those that would no longer pass.

Argument

`(g1)` `grammar`

Base grammar.

Argument

⟨yy⟩: true `bool`

Whether to remove positive tests.

Argument

⟨nn⟩: true `bool`

Whether to remove negative tests.

VI.1.3 Running tests

`#kleene.show-invisible` `#kleene.test`

#kleene.show-invisible(⟨line⟩, ⟨preserve-linebreaks⟩: false, ⟨dim-color⟩: black.lighten(70%), ⟨mode⟩: "special") → content

Private

Transforms a string to show invisible characters (spaces, linebreaks, tabs). Returns the resulting text as a #raw block.

Argument

⟨line⟩ `str`

Input text to transform.

Argument

⟨preserve-linebreaks⟩: false `bool`

By default, linebreaks ("\n") are replaced by a substitute character. If this option is enabled, the formatted text will still have the original linebreaks in addition to the marker.

Argument

⟨dim-color⟩: black.lighten(70%) `color`

Which color the special characters should be shown as.

Argument

⟨mode⟩: "special" `str`

Determines the substitution dictionary to use.

- "special": \n and \t.
- "unicode": ¤, », ␣, ⌀ denote respectively linebreak, tab, space, eof.

#kleene.test(⟨grammar⟩, ⟨select⟩: auto, ⟨total⟩: true) → content

Runs the unit tests attached to a grammar See #prelude.yy and #prelude.nn for details. Runs all positive and negative tests, and formats them in a table.

Argument

⟨grammar⟩ `grammar`

The grammar to test, as constructed by `#kleene.grammar`.

Argument

`(select): auto` `auto` | `array` | `function`

Pass an array or a function to filter a subset of the tests.

Argument

`(total): true` `bool`

Whether to display the final tally of passed/failed tests.

VI.2 Pattern combinators

`#prelude.commit`	`#prelude.maybe`	`#prelude.seq-aux`
`#prelude.drop`	`#prelude.neg`	`#prelude.star`
`#prelude.eof`	`#prelude.peek`	`#prelude.str`
`#prelude.fork`	`#prelude.regex`	`#prelude.try`
`#prelude.iter`	`#prelude.rewrite`	
`#prelude.label`	`#prelude.seq`	

`#prelude.commit` → `pattern`

Inserts a non-backtracking point. Returns `#none`.

See: Section II.4.3.

`#prelude.drop(..(pats))` → `pattern`

Matches an arbitrary pattern, but returns `#none`.

See: Section II.4.1.

Argument

`..(pats)` `pattern`

Implicitly cast to a `#prelude.seq`.

`#prelude.eof` → `pattern`

Matches the end of the stream. Returns `#none`.

See: Section II.1.1.

`#prelude.fork(..(pats))` → `pattern`

Branching choice between multiple possible subpatterns. Returns the first match.

See: Section II.2.2.

Argument

`..(pats)` `pattern`

List of ordered patterns between which a choice is made.

`#prelude.iter(..⟨pats⟩) → pattern`

Matches 1 or more instances of the inner pattern. Returns an `array`.

See: Section II.3.2.

Argument

`..⟨pats⟩` `pattern`

List of patterns, implicitly cast to a `#prelude.seq`.

`#prelude.label(⟨lab⟩) → pattern`

Recursively invokes another rule.

See: Section II.1.4.

Argument

`⟨lab⟩` `str`

Label of a rule defined elsewhere in the grammar.

`#prelude.maybe(..⟨pats⟩) → pattern`

Matches 0 or 1 instances of the inner pattern. Returns an `array`, either empty or singleton.

See: Section II.3.1.

Argument

`..⟨pats⟩` `pattern`

Implicitly cast to a `#prelude.seq`.

`#prelude.neg(..⟨pats⟩) → pattern`

Negative lookahead: checks that a pattern does not match. Returns `#none` and does not consume the input.

See: Section II.5.2

Argument

`..⟨pats⟩` `pattern`

Inner patterns that should not match. Auto-cast to a `#prelude.seq`.

`#prelude.peek(..⟨pats⟩) → pattern`

Positive lookahead: matches a pattern without consuming the input. Returns `#none`.

See: Section II.5.1.

Argument

`..⟨pats⟩` `pattern`

Inner patterns that should match. Auto-cast to a `#prelude.seq`.

`#prelude.regex(⟨re⟩)`

Uses a regular expression to more efficiently match a string. Returns `str`.

See: Section II.1.3.

Argument

`⟨re⟩` `str`

String representing a regular expression using the syntax of the standard library

`#prelude.rewrite(⟨fun⟩)` → `function(..pattern) => pattern`

Currified so that common rewriting patterns can be conveniently written as standalone functions. The type of the return value is that of the function. If multiple patterns are given, they are implicitly cast to a `#prelude.seq`.

See: Section II.4.2.

Argument

`⟨fun⟩` `none` | `auto` | `function`

Pattern to rewrite with.

- `#none`: the value is dropped,
- `#auto`: identity transformation,
- `function`: applies the given function.

`#prelude.seq(..⟨pats⟩, ⟨array⟩: true)` → `pattern`

Matches a sequence of patterns in order. Returns an `array`, except for possible optimizations where an implicit invocation matches only one element.

See: Section II.2.1.

Argument

`..⟨pats⟩` `pattern`

Sub-patterns.

Argument

`⟨array⟩: true` `bool`

If true, forces the result to be an array even if it is of size 1.

`#prelude.seq-aux(⟨pats⟩, ⟨array⟩: false)` → `pattern`

Helper to build sequences.

Argument

`⟨pats⟩` `array(pattern)`

Patterns that need to be matched in order.

Argument

`⟨array⟩: false` `bool`

If true, forces the result to be an array even if it is of size 1.

`#prelude.star(..(pats)) → pattern`

Matches 0 or more instances of the inner pattern. Returns an `array`.

See: Section II.3.3.

Argument

`..(pats)` `pattern`

List of patterns, implicitly cast to a `#prelude.seq`.

`#prelude.str((string)) → pattern`

Matches a string literal. Returns a `str`.

See: Section II.1.2.

Argument

`(string)` `str`

Any string to match as-is.

`#prelude.try(..(pats))`

Cancels out a `#prelude.commit` to re-enable backtracking. Returns the inner match.

See: Section II.4.4

Argument

`..(pats)` `pattern`

Implicitly cast to a `#prelude.seq`.

VI.3 Rule definition

```
#prelude.nn                 #prelude.rw
#prelude.pat                #prelude.yy
```

`#prelude.nn(..(tests), (validate): auto) → rule`

Declares one or more negative unit tests. They must produce a parsing error and pass validation. See Section III for details.

Argument

`..(tests)` `raw`

Inputs passed to the parser.

Argument

`(validate): auto` `function` | `auto`

Additional checks to run on the output. If a function, it will be given two arguments: the input and the error message.

`#prelude.pat(..(pats)) → rule`

Adds one case to the rule. Multiple invocations are globally taken as a `#prelude.fork`.

Argument

`..(pats)` `pattern`

Pattern that the rule matches. If there are multiple, they are cast to a `#prelude.seq`.

`#prelude.rw((fun)) → rule`

Adds a rewrite to all preceding `#prelude.pat` that do not already have one. See Section II.4.2 for more information.

Argument

`(fun)` `function` | `auto` | `none`

Rewriting a function. Uses the same convention as Section II.4.2, where `#auto` is the identity (also the default), and `#none` removes the matched value entirely.

`#prelude.yy(..(tests), (validate): auto) → rule`

Declares one or more positive unit tests. They must parse correctly and pass validation. See Section III for details.

Argument

`..(tests)` `raw`

Inputs passed to the parser.

Argument

`(validate): auto` `function` | `auto`

Additional checks to run on the output. If a function, it will be given two arguments: the input and the parsed output.

Part VII
About

VII.1 Alternatives

- JIEXI was recently released, employing a completely different approach compared to KLEENE in both implementation and philosophy: JIEXI is based on shift-reduce automata and uses an external tool via WASM. This will result in a completely different tradeoff in terms of performance, quality of error messages, customizability, etc.
- KLEENE is general-purpose, but if the language you want to read already has a dedicated parser you will likely get much better results with a specialized tool, of which many already exist.
- Writing a plugin is not strictly speaking a native solution, but if you're willing to write the parser in Rust you will be able to import it into Typst.

VII.2 Dependencies

KLEENE itself has no dependencies.

This manual is built using MANTYS and TIDY.