```
    {match: /"""[^]*?"""/,
    lineBreaks: true, value: x =>
    x.slice(3, -3)},
    {match: /"(?:\\["\\rn]|[^"\
    \])*?"/, lineBreaks: true,
    value: x => x.slice(1, -1)},
    {match: /'(?:\\['\\rn]|[^'\
    \])*?'/, lineBreaks: true,
    value: x => x.slice(1, -1)},
  ],
  // ...
})
```

Contributing

Do check the FAQ.

Before submitting an issue, remember…

Moo!

Moo is a highly-optimised tokenizer/lexer generator. Use it to tokenize your strings, before parsing 'em with a parser like nearley or whatever else you're into.

- Fast
- Convenient
- uses Regular Expressions
- tracks Line Numbers
- handles Keywords
- supports States
- custom Errors
- is even Iterable
- has no dependencies
- 4KB minified + gzipped
- Moo!

Is it fast?

Yup! Flying-cows-and-singed-steak fast.

Moo is the fastest JS tokenizer around. It's **~2–10x** faster than most other tokenizers; it's a **couple orders of magnitude** faster than some of the slower ones.

Iteration

Iterators: we got 'em.

```
for (let here of lexer) {
  // here = { type: 'number', value:
     '123', ... }
}
```

Create an array of tokens.

```
let tokens = Array.from(lexer);
```

Use itt's iteration tools with Moo.

```
for (let [here, next] of
     itt(lexer).lookahead()) { //
     pass a number if you need more
     tokens
  // enjoy!
}
```

Transform

Moo doesn't allow capturing groups, but you can supply a transform function, `value()`, which will be called on the value before storing it in the Token object.

```
moo.compile({
  STRING: [
    moo.reset('invalid')
    moo.next() // -> { type: 'myError',
        value: 'invalid', text:
        'invalid', offset: 0,
        lineBreaks: 0, line: 1, col: 1 }
    moo.next() // -> undefined
```

You can have a token type that both matches tokens *and* contains error values.

```
    moo.compile({
      // ...
      myError: {match: /[\$?`]/, error:
        true},
    })
```

Formatting errors

If you want to throw an error from your parser, you might find `formatError` helpful. Call it with the offending token:

```
throw new
        Error(lexer.formatError(token,
        "invalid syntax"))
```

It returns a string with a pretty error message.

```
Error: invalid syntax at line 2 col 15:

  totally valid `syntax`
                ^
```

Define your tokens **using regular expressions**. Moo will compile 'em down to a **single RegExp for performance**. It uses the new ES6 **sticky flag** where possible to make things faster; otherwise it falls back to an almost-as-efficient workaround. (For more than you ever wanted to know about this, read adventures in the land of substrings and RegExps.)

You *might* be able to go faster still by writing your lexer by hand rather than using RegExps, but that's icky.

Oh, and it avoids parsing RegExps by itself. Because that would be horrible.

Usage

First, you need to do the needful: `$ npm install moo`, or whatever will ship this code to your computer. Alternatively, grab the `moo.js` file by itself and slap it into your web page via a `<script>` tag; moo is completely standalone.

Then you can start roasting your very own lexer/tokenizer:

```
const moo = require('moo')

let lexer = moo.compile({
  WS:      /[ \t]+/,
  comment: /\/\/.*?$/,
  number:  /0|[1-9][0-9]*/,
  string:  /"(?:\\["\\]|[^\n"\
    \])*"/,
  lparen:  '(',
  rparen:  ')',
  keyword: ['while', 'if', 'else',
    'moo', 'cows'],
  NL:      { match: /\n/,
    lineBreaks: true },
})
```

And now throw some text at it:

```
lexer.reset('while (10) cows\nmoo')
lexer.next() // -> { type:
    'keyword', value: 'while' }
lexer.next() // -> { type: 'WS',
    value: ' ' }

    lexer.next() // -> { type:
    'lparen', value: '(' }

    lexer.next() // -> { type:
    'number', value: '10' }
// ...
```

When you reach the end of Moo’s internal buffer, next() will return `undefined`. You can always `reset()` it and feed it more data when that happens.

```
  },
  lit: {
    interp:   {match: '${', push:
    'main'},
    escape:   /\\./,
    strend:   {match: '`', pop: 1},
    const:    {match: /(?:[^$`]|\$
    (?!\{))+/, lineBreaks: true},
  },
})
// <= `a${{c: d}}e`
// => strstart const interp lbrace
    ident colon space ident rbrace
    rbrace const strend
```

The `rbrace` rule is annotated with `pop`, so it moves from the `main` state into either `lit` or `main`, depending on the stack.

Errors

If none of your rules match, Moo will throw an Error; since it doesn’t know what else to do.

If you prefer, you can have moo return an error token instead of throwing an exception. The error token will contain the whole of the rest of the buffer.

```
moo.compile({
  // ...
  myError: moo.error,
})
```

States

Moo allows you to define multiple lexer **states**. Each state defines its own separate set of token rules. Your lexer will start off in the first state given to `moo.states({})`.

Rules can be annotated with `next`, `push`, and `pop`, to change the current state after that token is matched. A “stack” of past states is kept, which is used by `push` and `pop`.

- `next: 'bar'` moves to the state named `bar`. (The stack is not changed.)
- `push: 'bar'` moves to the state named `bar`, and pushes the old state onto the stack.
- `pop: 1` removes one state from the top of the stack, and moves to that state. (Only `1` is supported.)

Only rules from the current state can be matched. You need to copy your rule into all the states you want it to be matched in.

For example, to tokenize JS-style string interpolation such as `a${{c: d}}e`, you might use:

```
let lexer = moo.states({
  main: {
    strstart: {match: '`', push:
    'lit'},
    ident:    /\w+/,
    lbrace:   {match: '{', push:
    'main'},
    rbrace:   {match: '}', pop: 1},
    colon:    ':',
    space:    {match: /\s+/,
    lineBreaks: true},
```

On Regular Expressions

RegExps are nifty for making tokenizers, but they can be a bit of a pain. Here are some things to be aware of:

- You often want to use **non-greedy quantifiers**: e.g. `*?` instead of `*`. Otherwise your tokens will be longer than you expect:

```
  let lexer = moo.compile({
  string: /".*"/,   // greedy quantifier *
  // ...
})

lexer.reset('"foo" "bar"')
lexer.next() // -> { type: 'string', value:
        'foo" "bar' }
```

Better:

```
  let lexer = moo.compile({
  string: /".*?"/,   // non-greedy
        quantifier *?
  // ...
})

lexer.reset('"foo" "bar"')
lexer.next() // -> { type: 'string', value:
        'foo' }
```

```
lexer.next() // -> { type: 'space', value: '
        ' }
lexer.next() // -> { type: 'string', value:
        'bar' }
```

- The **order of your rules** matters. Earlier ones will take precedence.

```
  moo.compile({
    identifier:  /[a-z0-9]+/,
    number:  /[0-9]+/,
}).reset('42').next() // -> { type:
        'identifier', value: '42' }
```

```
moo.compile({
    number:  /[0-9]+/,
    identifier:  /[a-z0-9]+/,
}).reset('42').next() // -> { type:
        'number', value: '42' }
```

- Moo uses **multiline RegExps**. This has a few quirks: for example, the **dot `/./` doesn't include newlines**. Use `[^]` instead if you want to match newlines too.
- Since an excluding character ranges like `/[^ ]/` (which matches anything but a space) *will* include newlines, you have to be careful not to include them by accident! In particular, the whitespace metacharacter `\s` includes newlines.

Keyword Types

Keywords can also have **individual types**.

```
let lexer = moo.compile({
  name: {match: /[a-zA-Z]+/, type:
    moo.keywords({
    'kw-class': 'class',
    'kw-def': 'def',
    'kw-if': 'if',
  })},
  // ...
})
lexer.reset('def foo')

    lexer.next() // -> { type: 'kw-
    def', value: 'def' }
lexer.next() // space
lexer.next() // -> { type: 'name',
    value: 'foo' }
```

You can use `Object.fromEntries` to easily construct keyword objects:

```
Object.fromEntries(['class', 'def',
        'if'].map(k => ['kw-' + k, k]))
```

```
moo.compile({
  IDEN: {match: /[a-zA-Z]+/, type:
    moo.keywords({
    KW: ['while', 'if', 'else',
    'moo', 'cows'],
  })},
  SPACE: {match: /\s+/, lineBreaks:
    true},
})
```

Why?

You need to do this to ensure the **longest match** principle applies, even in edge cases.

Imagine trying to parse the input `className` with the following rules:

```
keyword: ['class'],
identifier: /[a-zA-Z]+/,
```

You'll get *two* tokens — `['class', 'Name']` – which is *not* what you want! If you swap the order of the rules, you'll fix this example; but now you'll lex `class` wrong (as an `identifier`).

The keywords helper checks matches against the list of keywords; if any of them match, it uses the type `'keyword'` instead of `'identifier'` (for this example).

Line Numbers

Moo tracks detailed information about the input for you.

It will track line numbers, as long as you **apply the** `lineBreaks: true` **option to any rules which might contain newlines**. Moo will try to warn you if you forget to do this.

Note that this is `false` by default, for performance reasons: counting the number of lines in a matched token has a small cost. For optimal performance, only match newlines inside a dedicated token:

```
newline: {match: '\n', lineBreaks:
    true},
```

Token Info

Token objects (returned from `next()`) have the following attributes:

- `type`: the name of the group, as passed to compile.
- `text`: the string that was matched.
- `value`: the string that was matched, transformed by your `value` function (if any).
- `offset`: the number of bytes from the start of the buffer where the match starts.
- `lineBreaks`: the number of line breaks found in the match. (Always zero if this rule has `lineBreaks: false`.)

- `line`: the line number of the beginning of the match, starting from 1.
- `col`: the column where the match begins, starting from 1.

Value vs. Text

The `value` is the same as the `text`, unless you provide a value transform.

```
const moo = require('moo')

const lexer = moo.compile({
  ws: /[ \t]+/,
  string: {match: /"(?:\\["\\]|[^\n"\
        \])*"/, value: s => s.slice(1,
        -1)},
})

lexer.reset('"test"')
lexer.next() /* { value: 'test', text:
        '"test"', ... } */
```

Reset

Calling `reset()` on your lexer will empty its internal buffer, and set the line, column, and offset counts back to their initial value.

If you don't want this, you can `save()` the state, and later pass it as the second argument to `reset()` to explicitly control the internal state of the lexer.

```
lexer.reset('some line\n')
let info = lexer.save() // ->
    { line: 10 }
lexer.next() // -> { line: 10 }
lexer.next() // -> { line: 11 }
// ...
lexer.reset('a different line\n',
    info)
lexer.next() // -> { line: 10 }
```

Keywords

Moo makes it convenient to define literals.

```
moo.compile({
  lparen:  '(',
  rparen:  ')',
  keyword: ['while', 'if', 'else',
    'moo', 'cows'],
})
```

It'll automatically compile them into regular expressions, escaping them where necessary.

Keywords should be written using the `keywords` transform.