



# Regular Expressions

André Restivo

# Index

Introduction	Literal Characters	Character Classes	Zero-Length Matches
Alternation	Quantifiers	Grouping and Capturing	Backreferences
Backtracking	Lookaround		NFA
In HTML	In PHP	In Javascript	

# Introduction

# Regular Expressions

A sequence of characters that forms a search pattern.

Used in:

- Data validation.
- Search and Replace.
- Parsing.

# Variants

- There are **several** different regular expression **processors**.
- All of them translate regular expressions into a Nondeterministic Finite Automaton (NFA).
- But can have slightly **different** **syntaxes**.

# Matching

- Matching is the process of **applying** a regular expression **pattern** to a text string and finding strings that are represented by that pattern.
- When **validating**, we normally expect the **whole** string to match the pattern.
- When **searching**, we expect a **substring** of that string to match the pattern.

# References

- [Online Regular Expression Tester](#)
- [Regular Expressions Tutorial](#)
- [Regex Golf](#)
- [Mail RFC822 Regexp](#)

# Literal Characters



# Literal Characters

A literal character matches the first occurrence of that character in the string.

```
a
```

```
fl I ate an apple.
```

You can tell the processor to match all occurrences if needed.

```
fl I ate an apple.
```

# Literal Characters

A series of literal characters, matches those same characters in the same order.

```
at
```

```
fl I ate an apple.
```

# Special Characters

There are twelve characters that have special meanings in regular expressions:

```
\ ^ $ . | ? * + ( ) [ {
```

To match any of these symbols, you need to escape them with a **backslash**.

```
\+
```

```
fl 1+1=2
```

All other characters should not be escaped as the backslash also has special meaning.

# Non-Printable Characters

- `\t` - tab
- `\r` - carriage return
- `\n` - line feed

# Character Classes

# Character Classes

A character class, or set, matches **only one** out of several characters.

```
gr[ae]y
```

```
fl gray or grey
```

# Ranges

You can use an hyphen to specify ranges in a character class.

```
[0-9a-zA-Z]
```

This matches all digits from 0 to 9 and all letters (both in lowercase and uppercase).

```
fl The cat is inside the box.
```

# Negated

A caret(^) after the opening square bracket negates the character class.

```
[^A-Za-f]
```

This matches all letters except those in uppercase and from *a* to *f*.

```
fl The cat is inside the box.
```



# Special Characters

Inside a character class, the only special characters are:

```
] \ ^ -
```

All others do not need to be escaped.

# Shorthand Character Classes

- `\d` digit - the same as `[0-9]`
- `\w` word character - the same as `[A-Za-z0-9_]`
- `\s` whitespace character - the same as `[\t\r\n\f]`
- `\D` not a digit - the same as `^[^0-9]` or `[^\d]`
- `\W` not a word character - the same as `^[^A-Za-z0-9_]` or `[^\w]`
- `\S` not a whitespace character - the same as `^[^\t\r\n\f]` or `[^\s]`

# Dot

The dot(.) matches any character except line breaks.

```
c.t
```

```
fl The cat is inside the box.
```

# Zero Length Matches

# Anchors

Anchors can be used to specify the position of the matched string.

- The **caret**(`^`) matches the position before the first character in the string.
- The **dollar sign**(`$`) matches right after the last character in the string.
- We can use both anchors to validate a string.

```
boys$
```

Matching all occurrences:

```
fl Everyone knows boys will be boys
```

# Word Boundaries

- The metacharacter `\b` is an anchor.
- It matches at a position that is called a *word boundary*.
- It always produces a zero-length match.
- This allows you to do whole word searches.

```
\bis\b
```

```
fl This island is beautiful.
```

# Alternation

# Alternation

The vertical bar(|) allows you to match a single regular expression out of several possible regular expressions.

```
cat|dog
```

Matching all occurrences:

```
fl I like both cats and dogs.
```



# Quantifiers

# Optional Items

The question mark(?) makes the preceding token in the regular expression optional.

```
colou?r
```

Matching all occurrences:

```
fl Do you write color our colours?
```

# Repetition quantifiers

Repetition quantifiers allow the preceding token to repeat:

- The star(\*) allows the token to repeat 0 or more times.
- The plus(+) allows the token to repeat 1 or more times.

```
[0-9]+
```

```
fl My phone number is 12345.
```

The question mark(?) is also a repetition quantifier that allows the token to repeat 0 or 1 times.

# Custom Repetitions

Using curly brackets({}) we can specify the maximum and minimum number of repetitions:

Repeat exactly 9 times:

```
[0-9]{9}
```

Repeat between 1 and 3 times:

```
[0-9]{1,3}
```

Repeat at least twice:

```
[0-9]{2,}
```

Repeat at most three times:

```
[0-9]{,3}
```

# Repetitions are Greedy

By default, regular expression processors try to match as many characters as possible when handling repetitions.

```
<.+>
```

```
fl This tea is <strong>good</strong>.
```

This might cause unexpected effects.

# Lazy Repetitions

To make repetitions lazy, we add a **question mark(?)** after the repetition operator.

```
<.+?>
```

Matching all occurrences:

```
fl This tea is <strong>good</strong>.
```

# Being lazy is hard work!

The reason why repetitions are greedy by default, is because being lazy forces the processor to backtrack more often.

An alternative would be using negated classes:

```
<[^>]+>
```

Matching all occurrences:

```
fl This tea is <strong>good</strong>.
```

# Grouping and Capturing



# Grouping

Putting part of a pattern inside **parentheses** creates a group.

Groups can be used to apply **quantifiers** and **alternation** to specific parts of the pattern.

```
((https?|ftp)://)?www\.example\.com
```

Matching all occurrences:

```
fl ftp://www.example.com or just www.example.com
```

# Capturing

Groups are automatically captured and numbered.

This allow you to **extract** different parts of the matched expression.

```
(cats|dogs) are (lazy|smart)
```

```
fl i think cats are lazy
```

- Group #0: cats are lazy
- Group #1: cats
- Group #2: lazy

The complete match is always group #0.

# Capturing

## Other Example

```
((https?|ftp)://)?www\.example\.com
```

fl <http://www.example.com>

- Group #0: <http://www.example.com>
- Group #1: http://
- Group #2: http

# Non Capturing

Sometimes we want to create a group without capturing it. To do that we start the group with a question mark(?) and a colon(:):

```
(?:(:https?|ftp)://)?www\.example\.com
```

fl <http://www.example.com>

- Group #0: <http://www.example.com>

# Backreferences

# Backreferences

Backreferences can be used to match the same text twice.

Some regular expression processor use `\n` to reference captured groups while other use `$n`.

Number with at least 3 digits and where the first number is the same as the last.

```
([0-9])[0-9]+\1
```

fl 1231

# Backtracking

# Backtracking

Although regular expression processors are greedy, they can backtrack if they fail to find a match.

```
([0-9])[0-9]+\1
```

```
fl 41231
```

Here, the processor starts by matching the 4 but when it fails to find another 4 in the text it backtracks and tries to start with the 1.

```
fl 41231
```



# Lookaround

# Lookahead and lookbehind

Lookahead and lookbehind are zero-length assertions (just like the start and end of line and word boundaries)

- These are also called **lookaround** assertions.
- They match characters but then **give up the match** without consuming the characters.
- They only **assert** whether a match is possible or not.

# Positive lookahead

Matches something followed by something else.

```
(cat|dog)(?=s)
```

Matches *cat* or *dog* if followed by an *s*:

```
fl My dog is not like other dogs.
```

# Negative lookahead

Matches something not followed by something else.

```
(cat|dog)(?!s)
```

Matches *cat* or *dog* if not followed by an *s*:

```
fl All the cats are smarter than my cat.
```

# Positive lookahead

Tells the processor to temporarily **step backwards** in the string and check if the text inside the lookahead can be **matched** there.

```
(?<=is)land
```

Matches *land* if preceded by *is*:

```
fl England is part of an island.
```

# Negative lookahead

Tells the processor to temporarily **step backwards** in the string and check if the text inside the lookahead **cannot** be matched there.

```
(?<!some)thing
```

Matches *thing* if it is not preceded by *some*:

```
fl There is something about this thing.
```

# Nondeterministic Finite Automaton

# Regular Expressions are NFAs

Deterministic Finite Automaton (DFA) are finite state machines where:

- each of its transitions is **uniquely** determined by its **source** state and **input** symbol, and
- reading an input symbol is **required** for each state transition.

Nondeterministic Finite Automaton don't need to obey these restrictions.

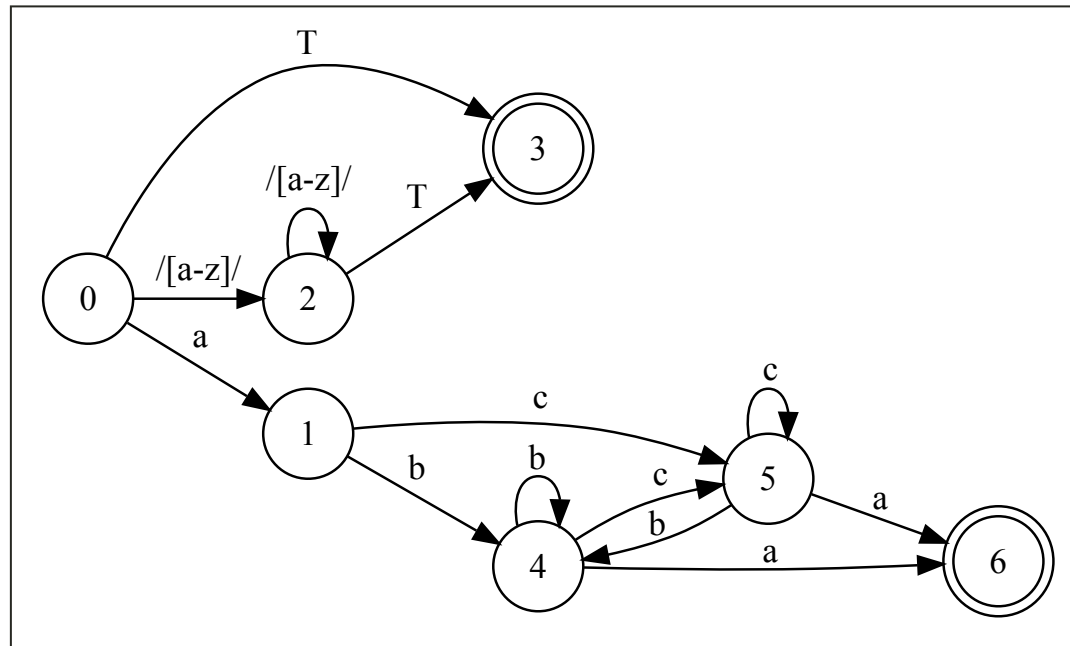
Regular expressions can **easily** be transformed into NFAs. And NFA can **easily** be transformed into DFAs.

<http://hackingoff.com/compilers/regular-expression-to-nfa-dfa>



# Example

$(a(b|c)+a)|([a-z])^*T$



In HTML

In HTML, input elements have a pattern attribute that can contain a regular expression pattern specifying the allowed values of the field.

```
<input type="text" pattern="\d{9}|\d{3}-\d{3}-\d{3}">
```

In PHP

# Patterns

- PHP uses Perl-Compatible Regular Expressions (PCRE)
- In PHP, patterns must be delimited by either forwardslashes (/), hash signs (#) or tildes (~).

```
/ab|c/
```

- This means that the chosen delimiter must be **escaped** inside the pattern.
- You may add **pattern modifiers** after the ending delimiter.

```
/ab|c/i
```

For example, the **i** pattern modifier makes the pattern case **insensitive**.

# preg\_match

```
int preg_match ( string $pattern , string $subject [, array &$matches ] )
```

The **preg\_match**, searches *subject* for a match to the regular expression given in *pattern*.

- If matches is provided, then it is filled with the results of the search.
- Returns 1 if the pattern matches given subject, 0 if it does not and false if an error occurred.

```
<?php
preg_match('/(\d{4})(?:-(\d{3}))?/', '4100-122', $matches);
print_r($matches);
?>
```

```
Array
(
    [0] => 4100-122
    [1] => 4100
    [2] => 122
)
```

# preg\_match\_all

```
int preg_match_all ( string $pattern , string $subject [, array &$matches ])
```

The **preg\_match\_all**, searches *subject* for **all** matches to the regular expression given in *pattern*.

- If matches is provided, then it is filled with all the results of the search in a multi-dimensional array.
- Returns the number of full pattern matches and false if an error occurred.

```
<?php
preg_match_all('/(\d{4})(?:-(\d{3}))?/', '4100-122 4200', $matches);
print_r($matches);
?>
```

```
Array
(
    [0] => Array ([0] => 4100-122 [1] => 4200)
    [1] => Array ([0] => 4100 [1] => 4200)
    [2] => Array ([0] => 122 [1] => )
)
```

# preg\_replace

```
mixed preg_replace ( mixed $pattern , mixed $replacement , mixed $subject )
```

The **preg\_replace** function, searches *subject* for matches to *pattern* and replaces them with *replacement*.

- The replacement can contain backreferences in the form **n** or **{n}**.

```
<?php
echo preg_replace('/(cat|dog)/', 'my $1s', 'dog are dog');
?>
```

```
my dogs are my dogs
```



# Validation

Using the `preg_match` function, we can easily validate data using regular expressions:

```
function is_phone_number($element) {  
    return !preg_match ("/^\d{9}|\d{3}-\d{3}-\d{3}$/", $element);  
}
```

Don't forget the beginning and end of string anchors.

# Cleaning

You can also use the `preg_replace` function to clean up input data before storing it in the database.

```
$text = preg_replace('/^[^w\d\s\.\!,\?]/', '', $_GET['text']);
```

**In Javascript**

# Patterns

- In javascript, patterns must be delimited by forwardslashes (/).
- This means that the forwardslashes must be **escaped** inside the pattern.
- You may add modifiers after the ending delimiter:

The **g** modifier is used to perform a global match (find all matches).

The **i** modifier is used to perform a case insensitive match.

# test

```
regexObj.test(str)
```

The **test** function, tests for a match in a string. It returns true or false.

```
console.log(/(\d{4})(?:-(\d{3}))?/.test('4100-122'));
```

```
true
```

# match

```
str.match(regex)
```

The **match** function, executes a search for a regular expression in a string.

```
console.log('4100-122 4200'.match(/(\d{4})(?:-(\d{3}))?/));  
console.log('4100-122 4200'.match(/(\d{4})(?:-(\d{3}))?/g));
```

```
["4100-122", "4100", "122", index: 0, input: "4100-122 4200"]  
["4100-122", "4200"]
```

# search

```
str.search([regex])
```

If successful, **search** returns the index of the first match of the regular expression inside the string.

```
console.log('My zip code is 4100-122'.search(/(\d{4})(?:-(\d{3}))?/));
```

```
15
```

# replace

```
str.replace(regex, replacement)
```

- The replacement can contain backreferences in the form \$n.

```
console.log('dog are dog'.replace(/(cat|dog)/, 'my $1s'));  
console.log('dog are dog'.replace(/(cat|dog)/g, 'my $1s'));
```

```
my dogs are dog  
my dogs are my dogs
```



# Validation

Using the test function, we can easily validate data using regular expressions:

```
function is_phone_number(element) {  
  return /^\\d{9}|\\d{3}-\\d{3}-\\d{3}$/.test(element);  
}
```

Don't forget the beginning and end of string anchors.

