Chapter NLP:I

I. Natural Language Processing Basics

- String Processing
- Grammars
- Regular Expressions

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

String Processing

Overview

String processing forms the basis of many tasks, algorithms, and evaluation methods in NLP.

String problems:

- Sorting
- Manipulation
- Exact / inexact matching
  Search and indexing, similarity and distance
- Alignment
  Longest common subsequence
- Parsing
  Splitting
- Compression

Data structures:

- Buffer
- Inverted index
- Trie, suffix tree, suffix array

Computational models:

- Finite-state machines
- Dynamic programming

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- String processing combines methods from formal language theory, compilers, and bioinformatics.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Terminology

- **Alphabet Σ.**

  An alphabet Σ is a finite non-empty set of letters or symbols.

- **Word w.**

  A word w is a finite sequence of symbols from Σ. The length of a word $|w|$ is the number of its symbols.

  ε denotes the empty word; it is the only word with length 0.

  Σ^* denotes the set of all words over Σ.

- **Language L.**

  A language L is a set of words over an alphabet Σ.

- **Grammar G.**

  A grammar G is a calculus for defining a language—that is, a set of rules that can be used to derive words. The language belonging to G consists of all derivable, terminal words.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Terminology

- **Alphabet Σ.**
  An alphabet Σ is a finite non-empty set of letters or symbols.

- **Word w.**
  A word w is a finite sequence of symbols from Σ. The length of a word $|w|$ is the number of its symbols.
  ε denotes the empty word; it is the only word with length 0.
  Σ^* denotes the set of all words over Σ.

- **Language L.**
  A language L is a set of words over an alphabet Σ.

- **Grammar G.**
  A grammar G is a calculus for defining a language—that is, a set of rules that can be used to derive words. The language belonging to G consists of all derivable, terminal words.

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- When defining language characteristics, a distinction is made between different levels of abstraction. Level 1 deals with the notation of symbols, while Level 2 deals with the syntactic structure of the language.
  [Exkurs: Programmiersprachen]

- To distinguish between the different levels of grammar usage, the following terms can be used:
  - Level 1: Alphabet, symbol, word, language
  - Level 2: Vocabulary, token, sentence, language

- The words {alphabet, vocabulary}, {symbol, token} or {word, sentence} are the respective equivalents of the elementary symbolic level and the syntactic level.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Definition 1 (Grammar)

A grammar is a quadruple $G = (N, \Sigma, P, S)$ with

N = finite set of non-terminal symbols

Σ = finite set of terminal symbols, $N \cap \Sigma = \emptyset$

P = finite set of productions or rules

$$P \subset (N \cup \Sigma)^* \, N \, (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

S = start symbol, $S \in N$

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Definition 1 (Grammar)

A grammar is a quadruple $G = (N, \Sigma, P, S)$ with

N = finite set of non-terminal symbols

Σ = finite set of terminal symbols, $N \cap \Sigma = \emptyset$

P = finite set of productions or rules

$$P \subset \underbrace{(N \cup \Sigma)^* \, N \, (N \cup \Sigma)^*}_{A} \times \underbrace{(N \cup \Sigma)^*}_{Ab}$$

$$A \rightarrow Ab$$

S = start symbol, $S \in N$

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- A rule consists of a left side (premise) and a right side (conclusion), each of which is a word consisting of terminals and non-terminals. The left side must contain at least one non-terminal, and unlike the left side, the right side can also be the empty word. [Wikipedia]
- A rule can be applied to a word consisting of terminals and non-terminals, whereby any occurrence of the left side of the rule in the word is replaced by the right side of the rule: $w \rightarrow w'$.
- Given the rule $w \rightarrow w'$, then w, w' are in the so-called *transitive relation* $\rightarrow_G$. A sequence of rule applications is called a *derivation*.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Definition 2 (Generated Language)

The language $L(G)$ generated by a grammar $G = (N, \Sigma, P, S)$ contains exactly those words that consist only of terminal symbols and can be derived from the start symbol with a finite number of steps:

$$L(G) := \{w \in \Sigma^* \mid S \to_G^* w\} \ .$$

$\to_G^*$ denotes the arbitrary application of the productions in G, i.e., the reflexive-transitive hull of the transition relation $\to_G$.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Definition 2 (Generated Language)

The language $L(G)$ generated by a grammar $G = (N, \Sigma, P, S)$ contains exactly those words that consist only of terminal symbols and can be derived from the start symbol with a finite number of steps:

$$L(G) := \{w \in \Sigma^* \mid S \rightarrow_G^* w\} \ .$$

$\rightarrow_G^*$ denotes the arbitrary application of the productions in G, i.e., the reflexive-transitive hull of the transition relation $\rightarrow_G$.

Example:

$G = (N, \Sigma, P, S)$ with $N = \{S, A, B\}$, $\Sigma = \{a, b\}$ and the production rules:

$$\begin{array}{lllllllll} S & \rightarrow & ABS & BA & \rightarrow & AB & Ab & \rightarrow & ab \\ S & \rightarrow & \varepsilon & BS & \rightarrow & b & Aa & \rightarrow & aa \\ & & & Bb & \rightarrow & bb & & & \end{array}$$

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- It is conventional to use uppercase letters to denote non-terminal symbols and lowercase letters to denote terminal symbols.
- Another grammar that generates the same language as in the example is:
  $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, $P = \{S \rightarrow aSb, S \rightarrow \varepsilon\}$

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Chomsky Hiearchy

Grammars are divided into four classes based on the complexity of the languages they generate.

- Type 0 $\sim$ recursively enumerable.
- Type 1 $\sim$ context-sensitive.
- Type 2 $\sim$ context-free.
- Type 3 $\sim$ regular.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Chomsky Hiearchy

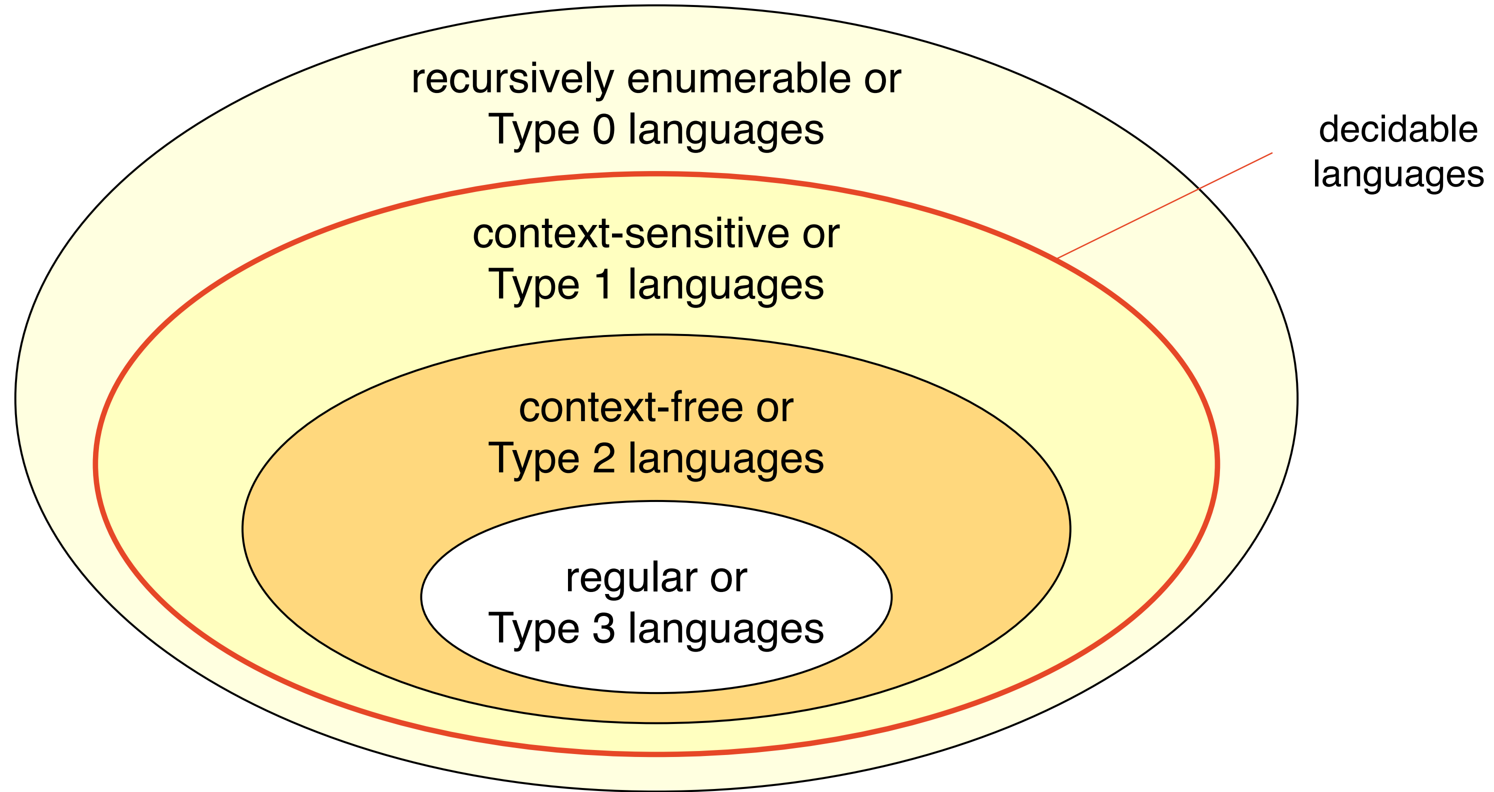
Grammars are divided into four classes based on the complexity of the languages they generate.

- Type 0 $\sim$ recursively enumerable.
  There are no restrictions for the rules in P.
- Type 1 $\sim$ context-sensitive.
  For all rules $w \to w' \in P$ holds: $|w| \leq |w'|$
- Type 2 $\sim$ context-free.
  For all rules $w \to w' \in P$ holds: w is a single variable; i.e., $w \in N$.
- Type 3 $\sim$ regular.
  The grammar is of Type 2, and for all rules, $w \to w'$ additionally holds: $w' \in (\Sigma \cup \Sigma N)$, i.e., the right-hand sides of the rules consist either of a terminal symbol or of a terminal symbol followed by a non-terminal.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Chomsky Hierarchy



Definition 3 (Language of Type)

A language $L \subseteq \Sigma^*$ is called a Type 0 (Type 1, Type 2, Type 3) language if there exists a Type 0 (Type 1, Type 2, Type 3) grammar G such that $L(G) = L$.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- The Chomsky hierarchy represents a hierarchy with genuine subset relationships:
  Type 3 $\subset$ Type 2 $\subset$ Type 1 $\subset$ Type 0
- All languages of types 1, 2, or 3 are decidable:
  - $\rightarrow$ The decision problem if a word belongs to a language is decidable for all languages of Type 1, 2, or 3.
  - $\rightarrow$ There is an algorithm that, given a grammar G and a word w, decides in finite time if $w \in L(G)$ or not.
- The set of Type 0 languages is identical to the set of recursively enumerable or semi-decidable languages. Therefore, there are Type 0 languages that are not decidable.

  There are countably infinite grammars for generating a recursively enumerable language (Type 0)
- In compiler theory and natural language processing, Type 3 languages and grammars (lexical analysis, tokenization) and Type 2 languages and grammars (syntactic structure analysis) play a central role.

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Calculi for Regular Languages

Different calculi for generating words in a regular language:

(a) finite acceptor or automaton

(b) regular expressions

(c) Type 3 grammar

(d) Specifying finitely many equivalence classes (of the Nerode relation)

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Calculi for Regular Languages

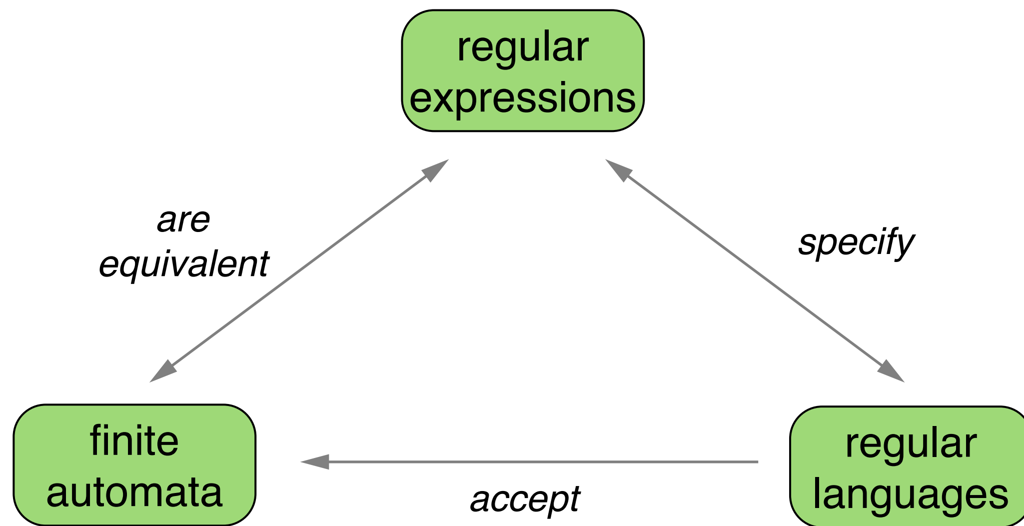
Different calculi for generating words in a regular language:

(a) finite acceptor or automaton

(b) regular expressions

(c) Type 3 grammar

(d) Specifying finitely many equivalence classes (of the Nerode relation)



[Haenelt 2005, Jurafski/Martin 2000]

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars
Summary

	Language generation calculi
Type 0	Type 0 grammar Turing machine
Type 1	context-senstive grammar linear bounded Turing machine [Wikipedia]
Type 2	context-free grammar pushdown automaton
Type 3	regular grammar (Type 3 grammar) deterministic/non-deterministic finite automaton regular expression

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Grammars

Summary

	Language generation calculi
Type 0	Type 0 grammar
	Turing machine
Type 1	context-senstive grammar
	linear bounded Turing machine [Wikipedia]
Type 2	context-free grammar
	pushdown automaton
Type 3	regular grammar (Type 3 grammar)
	deterministic/non-deterministic finite automaton
	regular expression

	Complexity of the word problem [Wikipedia]
Type 0	undecidable
Type 1	exponential complexity, NP-hard
Type 2	$O(n^3)$
Type 3	linear complexity

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Chapter NLP:I

I. Natural Language Processing Basics

- String Processing
- Grammars
- Regular Expressions

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions [Kastens]

Syntax

A regular expression R can be composed recursively as follows. F and G denote regular expressions.

	R	**Semantics**
1.	a	A letter a
2.	FG	Concatenation
3.	$F \mid G$	Alternation
4.	(F)	Grouping
5.	F^+	Non-empty sequence of words from $L(F)$
6.	F^*	Arbitrary sequence of words from $L(F)$
7.	F^n	Sequence of n words from $L(F)$
8.	ε	The empty word

 © WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions [Kastens]

Syntax

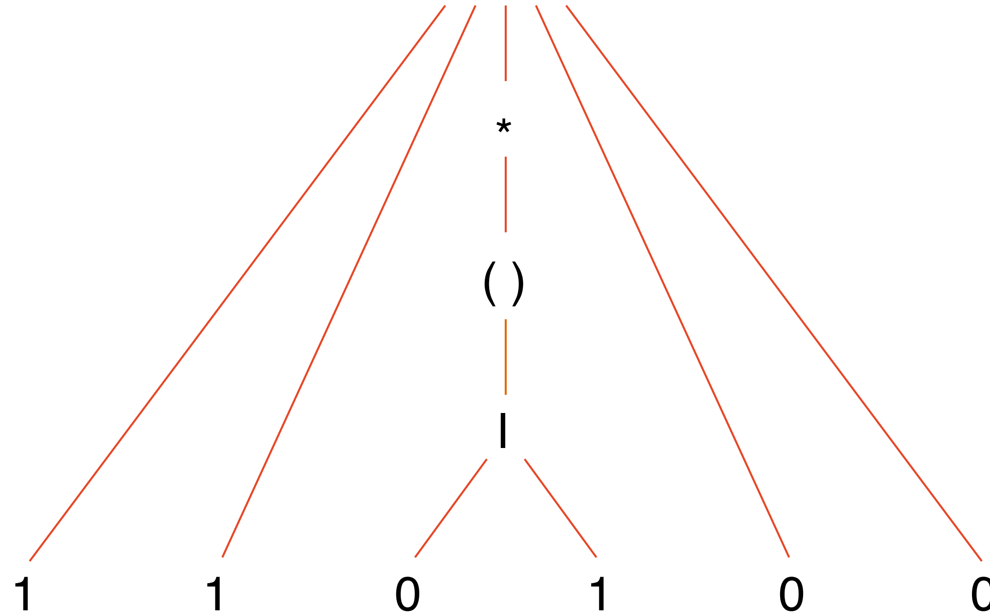
A regular expression R can be composed recursively as follows. F and G denote regular expressions.

	R	Language $L(R)$	Semantics
1.	a	$\{a\}$	A letter a
2.	FG	$\{fg \mid f \in L(F),\ g \in L(G)\}$	Concatenation
3.	$F \mid G$	$\{f \mid f \in L(F)\} \ \cup \ \{g \mid g \in L(G)\}$	Alternation
4.	(F)	$(L(F))$	Grouping
5.	F^+	$\{f_1 f_2 \ldots f_n \mid f_i \in L(F),\ n \geq 1,\ i = 1, \ldots, n\}$	Non-empty sequence of words from $L(F)$
6.	F^*	$\{\varepsilon\} \cup L(F^+)$	Arbitrary sequence of words from $L(F)$
7.	F^n	$\{f_1 f_2 \ldots f_n \mid f_i \in L(F),\ i = 1, \ldots, n\}$	Sequence of n words from $L(F)$
8.	ε	$\{\varepsilon\}$	The empty word

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions [Kastens]

Examples



Concatenation (5 times)

Sequence

Bracketing

Alternative

Symbol

Every word in the language of this regular expression consists of two ones, followed by any number of zeros or ones, followed by two zeros.

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions [Kastens]

Examples (continued)

R	**Name of language $L(R)$**	**Example words from $L(R)$**
(a | b) (c | d | ε)	*Abc*	ac, bc, ad, bd, a, b
Dear(ε | est) (Sir | Madam)	*Salutation*	Dear Sir

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions [Kastens]

Examples (continued)

R	Name of language $L(R)$	Example words from $L(R)$
(a | b) (c | d | ε)	*Abc*	ac, bc, ad, bd, a, b
Dear(ε | est) (Sir | Madam)	*Salutation*	Dear Sir
0 | 1 | . . . | 9	*Digit*	7
a | b | . . . | z	*sLetter*	x
A | B | . . . | Z	*cLetter*	B
sLetter | *cLetter*	*Letter*	m, N

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions [Kastens]

Examples (continued)

R	Name of language $L(R)$	Example words from $L(R)$
(a | b) (c | d | ε)	*Abc*	ac, bc, ad, bd, a, b
Dear(ε | est) (Sir | Madam)	*Salutation*	Dear Sir
0 | 1 | . . . | 9	*Digit*	7
a | b | . . . | z	*sLetter*	x
A | B | . . . | Z	*cLetter*	B
sLetter | *cLetter*	*Letter*	m, N
Letter (*Letter* | *Digit*)*	*Label*	Maximum, min7, a
$\mathit{Digit}^{+}.\mathit{Digit}^{2}$	*MoneyAmount*	23.95, 0.50
$(\mathit{cLetter} \mid \mathit{cLetter}^{2} \mid \mathit{cLetter}^{3})$– $(\mathit{cLetter} \mid \mathit{cLetter}^{2})$– $(\mathit{Digit} \mid \mathit{Digit}^{2} \mid \mathit{Digit}^{3} \mid \mathit{Digit}^{4})$	*LicensePlatesDE*	PB–AS–0815
1^{3} (1 | 0)* 0^{3}	*Dual*	1111000, 1111101010000

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

Examples (continued)

An important use of regular expressions in languages used for natural language processing is the specification of text patterns.

Example: Display of all file names of the form

$$\text{“winter-term}(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^2\text{.html”}$$

- Unix-Shell.

```
ls winter-term[0-9][0-9].html
```

- PHP.

```
$d = "[0-9]";
preg_match("/winter-term$d$d\.html/", $files)
```


©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- If names of regular expressions are used in other regular expressions, they must be identified as part of the meta language.

  Here: Use of *italics*.

- Each scripting language for text processing uses a different syntax to specify regular expressions; the construction principles and power are comparable.

- The specification of regular expressions in PHP is taken from the Perl scripting language.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

Definition 4 (Exact Matching Problem [Gusfield 1997])

Given a string R called the pattern and a longer string T called the text, the exact matching problem is to find all occurrences, if any, of pattern R in text T.

Example:

- Given $R = aba$ and $T = bbabaxababay$, then R occurs in T starting at locations 3, 7, and 9.
- Two occurrences of R overlap.

Importance of the exact matching problem:

- Exact matching is solved for searching local files and data.
- Main bottleneck is the speed with which the data can be read into memory.
- Speed rests on “preprocessing” pattern R (or text T) to enable skipping during search.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

Perl Compatible Regular Expressions (PCRE) [pcre.org, PHP, Python, JavaScript, Java, .NET]

Many programming languages use a Perl-like syntax for regular expressions:

"*Delimiter Regular_Expression Delimiter* [*Flags*]"

Pattern

- The delimiter must be a non-alphanumeric character.
- Optional flags influence the matching strategy. [pcre.org, PHP, Python, JavaScript, Java, .NET]
- Search and replace functions with regular expressions. [pcre.org, PHP, Python, JavaScript, Java, .NET]

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

Perl Compatible Regular Expressions (PCRE) [pcre.org, PHP, Python, JavaScript, Java, .NET]

Many programming languages use a Perl-like syntax for regular expressions:

"*Delimiter Regular_Expression Delimiter* [*Flags*]"

Pattern

- The delimiter must be a non-alphanumeric character.
- Optional flags influence the matching strategy. [pcre.org, PHP, Python, JavaScript, Java, .NET]
- Search and replace functions with regular expressions. [pcre.org, PHP, Python, JavaScript, Java, .NET]

Examples: (PHP)

Pattern *Text*

```
echo preg_match("/def/" , "defabcdef" );   ⤳ 1

echo preg_match("=def=", "defabcdef");     ⤳ 1
```


© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

Perl Compatible Regular Expressions (continued) [Regular Expressions (Syntax)]

A regular expression R can be composed recursively as follows. Let F and G denote regular expressions and $L(F)$, $L(G)$ their languages over the alphabet Σ.

	R [pcre.org]	Semantics
1.	a	The letter a
	.	Any symbol from Σ (except line break)
	$[\Sigma']$	Character class: Any symbol from $\Sigma' \subseteq \Sigma$
	$[\hat{}\Sigma']$	No symbol from character class $\Sigma' \subseteq \Sigma$
	$\backslash\Sigma'$	Escaped letter from a controlled $\Sigma' \subset \Sigma$, denoting a character class
2.	FG	Concatenation
3.	$F \mid G$	Alternation
4.	(F)	Grouping or capturing
	$F?$	F is optional (same semantics as $F \mid \varepsilon$)
5.	$F+$	Non-empty sequence of words from $L(F)$
6.	$F*$	Arbitrary sequence of words from $L(F)$
7.	$F\{n\}$	Sequence of n words from $L(F)$
	$F\{m,n\}$	Sequence with at least m and at most n words from $L(F)$
	^ or $\$$	Assertion: Beginning or end of the text string (nothing may precede or follow it)

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Character Classes [pcre.org]

Character classes match a single token from a list of characters.

R	Semantics	Example
[...]	Positive character class matches listed character	`[aeiou]` matches vowels
[^...]	Negative character class matches only unlisted characters	`[^aeiou]` matches anything except for vowels
$[x-y]$	Range: matches any letter between x and y inclusive	`[a-zA-Z]` matches alphabetic characters
`\d`	Matches any decimal digits. Short for `[0-9]`	
`\D`	Matches all except decimal digits. Short for `[^0-9]`	
`\s`	Matches white space characters. Short for `[\t\n\r\f\v ]`	
`\S`	Matches all except white spaces. Short for `[^\t\n\r\f\v ]`	
`\w`	Matches alphanumeric characters. Short for `[a-zA-Z0-9]`	
`\W`	Matches non-alphanumeric characters. Short for `[^a-zA-Z0-9]`	

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- ❑ `]` needs to be escaped inside character classes to be matched: `[^]]` does not match `a` but, e.g., `]]`.
- ❑ The range in a character class denotes a series of codepoints.
  As such, `[--2]` is valid and identical to `[-./012]`. `[A-z]` matches not only alphabetic characters.
- ❑ Character classes cannot be nested but `\d`, `\D`, `\s`, … can be used inside character classes:
  - `[^[a-z][A-Z]]` matches the regex `[^[a-z]`, `[A-Z]`, and `]` in sequence.
  - `[^\s\w]` matches any character that is neither a space, nor a word-character.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Groups and Backreferencing [pcre.org]

Groups give precedence to subformulas and can store (capture) matches for later reference.

R	Semantics	Example
(F)	Capture Group: Groups F and stores the matched value	(ab)+ matches ab, abab, ...
(?:F)	Non-capturing group: Groups F	(?:ab)+ matches ab, abab, ...
(?<*name*>F)	Named capture group: Groups F and stores it under a *name*	
\\n	Backreference the n-th capture group	(\w+)\1 matches fully reduplicated words
\k{*name*}	Backreference capture group named "*name*"	(?<word>\w+)\k{word} is the same as above

- Capture groups store the last match.
  The regex ([ab])+\1 matches abb but not aba.

- Named capture groups can extract information from different positions in the regex.
  The regex \d\d.(?<month>\d\d).\d\d|(?<month>\d\d)/\d\d/\d\d extracts the month from 16.01.70 and 01/16/70.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- The `\n` syntax is highly ambiguous:
  - If n is an octal number (`\1`, `\2`, ..., `\7`, `\10`, ...), it matches the character point with value n.
  - If n is a single non-octal digit (`\8`, `\9`), it matches the respective character.
  - If n is a valid index for a capture group, the above is irrelevant and the capture group is backreferenced.

  `\8` matches the character `8` but `()()()()()()()()\8` only matches the empty word.
  `\49` gives an error because there is no 49th capture group but `\50` matches `(`.

➜ PCRE introduces `\gn` or `\g{n}` to reference capture groups and `\on` for octal codepoints. These are less common in other flavors of regex.

- Regular expressions compatible with PCRE2 are more powerful than regular grammars.
  The language $L = \{ww \mid w \in \{0,1\}^*\}$ is not context free but can be decided by the regex `([01])*\1`.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Quantifiers [pcre.org]

Quantifiers allow for repeated matching of a regular expression F.

R	Semantics	Example
`F{m}`	Sequence of m words from $L(F)$	`\d{2}` matches exactly two digits
`F{m,}`	Sequence of at least m words from $L(F)$	`\d{2,}` matches two or more digits
`F{m,n}`	Sequence of at least m and at most n words from $L(F)$	`\d{2,5}` matches two to five digits
`F{,n}`	Sequence of at most n words from $L(F)$ *(since PCRE2 10.43)*	`\d{,10}` matches up to ten digits
`F?`	Equivalent to $F\{0,1\}$	`\w?` matches at most one word-character
`F*`	Equivalent to $F\{0,\}$	`\w*` matches any number of word-characters
`F+`	Equivalent to $F\{1,\}$	`\w+` matches one or more word-characters

- Default. Greedy matching: as much as possible and backtracking until a match is found.
  `\d*\d\d` tries to match `1234`

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Quantifiers [pcre.org]

Quantifiers allow for repeated matching of a regular expression F.

R	Semantics	Example
`F{m}`	Sequence of m words from $L(F)$	`\d{2}` matches exactly two digits
`F{m,}`	Sequence of at least m words from $L(F)$	`\d{2,}` matches two or more digits
`F{m,n}`	Sequence of at least m and at most n words from $L(F)$	`\d{2,5}` matches two to five digits
`F{,n}`	Sequence of at most n words from $L(F)$ *(since PCRE2 10.43)*	`\d{,10}` matches up to ten digits
`F?`	Equivalent to $F\{0,1\}$	`\w?` matches at most one word-character
`F*`	Equivalent to $F\{0,\}$	`\w*` matches any number of word-characters
`F+`	Equivalent to $F\{1,\}$	`\w+` matches one or more word-characters

- Default. Greedy matching: as much as possible and backtracking until a match is found.
  `\d*\d\d` tries to match `1234` but `\d\d` cannot be matched

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Quantifiers [pcre.org]

Quantifiers allow for repeated matching of a regular expression F.

R	**Semantics**	**Example**
`F{m}`	Sequence of m words from $L(F)$	`\d{2}` matches exactly two digits
`F{m,}`	Sequence of at least m words from $L(F)$	`\d{2,}` matches two or more digits
`F{m,n}`	Sequence of at least m and at most n words from $L(F)$	`\d{2,5}` matches two to five digits
`F{,n}`	Sequence of at most n words from $L(F)$ *(since PCRE2 10.43)*	`\d{,10}` matches up to ten digits
`F?`	Equivalent to $F\{0,1\}$	`\w?` matches at most one word-character
`F*`	Equivalent to $F\{0,\}$	`\w*` matches any number of word-characters
`F+`	Equivalent to $F\{1,\}$	`\w+` matches one or more word-characters

- Default. Greedy matching: as much as possible and backtracking until a match is found.
  `\d*\d\d` tries to match `1234` but `\d` cannot be matched

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Quantifiers [pcre.org]

Quantifiers allow for repeated matching of a regular expression F.

R	**Semantics**	**Example**
`F{m}`	Sequence of m words from $L(F)$	`\d{2}` matches exactly two digits
`F{m,}`	Sequence of at least m words from $L(F)$	`\d{2,}` matches two or more digits
`F{m,n}`	Sequence of at least m and at most n words from $L(F)$	`\d{2,5}` matches two to five digits
`F{,n}`	Sequence of at most n words from $L(F)$ *(since PCRE2 10.43)*	`\d{,10}` matches up to ten digits
`F?`	Equivalent to $F\{0,1\}$	`\w?` matches at most one word-character
`F*`	Equivalent to $F\{0,\}$	`\w*` matches any number of word-characters
`F+`	Equivalent to $F\{1,\}$	`\w+` matches one or more word-characters

- Default. Greedy matching: as much as possible and backtracking until a match is found.
  `\d*\d\d` tries to match `1234` and but `\d\d` matches as well → success!

©WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Quantifiers [pcre.org]

Quantifiers allow for repeated matching of a regular expression F.

R	Semantics	Example
F{m}	Sequence of m words from $L(F)$	`\d{2}` matches exactly two digits
F{m,}	Sequence of at least m words from $L(F)$	`\d{2,}` matches two or more digits
F{m,n}	Sequence of at least m and at most n words from $L(F)$	`\d{2,5}` matches two to five digits
F{,n}	Sequence of at most n words from $L(F)$ *(since PCRE2 10.43)*	`\d{,10}` matches up to ten digits
F?	Equivalent to $F\{0,1\}$	`\w?` matches at most one word-character
F*	Equivalent to $F\{0,\}$	`\w*` matches any number of word-characters
F+	Equivalent to $F\{1,\}$	`\w+` matches one or more word-characters

- Modify the matching behavior by adding `?` (lazy) or `+` (possessive) behind the quantifier.
  `\d+?` matches as few digits as possible, but at least one. `\d++` matches greedily without backtracking

Strings	R			
	`".*"`	`".*?"`	`".*+"`	`"[^"]*+"`
`"hi"`	`{"hi"}`	`{"hi"}`	$\emptyset$	`{"hi"}`
`"foo" "bar"`	`{"foo" "bar"}`	`{"foo","bar"}`	$\emptyset$	`{"foo","bar"}`

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Remarks:

- Possessive matching is more efficient than greedy or lazy matching because no backtracking is performed.
- Possessive matching is a special case of an atomic group (written `(?>`F`)`). Atomic groups cannot be backtracked.

  The regex `(?>a|aa)a` matches `aa` but not `aaa` because for the latter it would initially match `a` for the capture group (the first matching option in the alternative) and then have to backtrack when the end of the string is reached.

  What strings would `(?>aa|a)a` match?

- F`?+`, F`*+`, and F`++` are short for `(?>`F`?)+`, `(?>`F`*)+`, and `(?>`F`+)+` respectively.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025

Regular Expressions

PCRE: Assertions [pcre.org]

Assertions test if a regex would match without matching it.

R	Semantics
`(?=`F`)`	(Positive) lookahead: Asserts that F matches at the current location.
`(?!`F`)`	Negative lookahead: Asserts that F does not match at the current location.
`(?<=`F`)`	(Positive) lookbehind: Asserts that F matches right before the current location.
`(?<!`F`)`	Negative lookbehind: Asserts that F does not match right before the current location.
	`(?<=\s)(?<!$)\d+` matches any number in the text that is not preceded by a $.
`^`	Begin of string anchor: Asserts the start of the input sequence.
`$`	End of string anchor: Asserts the end of the input sequence.
	`^[01]+\$` checks that the input entirely represents a binary number
`\b`	Asserts a word boundary. Short for `(?<=\W)(?=\w)|(?<=\w)(?=\W)`
`\B`	Asserts the opposite of `\b`. Short for `(?<=\W)(?=\W)|(?<=\w)(?=\w)`
	`\b[Tt]he\b` matches the word “the” but not if it occurs inside another word as in “atheist”.
	`\Bing\b` matches the suffix -ing. For example in “climbing” or “sing” but not in “singer”.

- `^`, `$`, `\b`, `\B` are simple assertions and `(?=`F`)`, `(?!`F`)`, `(?<=`F`)`, `(?<!`F`)` are assertion groups.
- Simple assertions cannot be followed by quantifiers.
  For example, `\b+` is illegal but `(\b)+` is allowed.

© WIEGMANN/WOLSKA/HAGEN/POTTHAST/STEIN 2025