Wyrażenia regularne: BRE, ERE, PCRE

Poziom: srednio **Dotyczy: wszystkie dystrybucje** **Czas: ~16 min**

Wyrażenia regularne opisują kształt tekstu, nie konkretny tekst. Problem w tym, że w Linuksie istnieją trzy niekompatybilne dialekty tego samego pomysłu — i to jest najczęstsze źródło „ten regex mi nie działa".

Skróty użyte na tej stronie (11)

Skrót	Co znaczy
ASCII	American Standard Code for Information Interchange: podstawowy zestaw znaków (litery, cyfry) używany w komputerach
BRE	Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami
BSD	Berkeley Software Distribution: rodzina Uniksów z uczelni Berkeley, inna gałąź niż Linux, stąd czasem inna składnia tych samych poleceń
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
ERE	Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
IPv4	Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77
PCRE	Perl Compatible Regular Expressions: odmiana wyrażeń regularnych z językiem Perl, najbogatsza w możliwości spośród trzech
PHP	PHP: Hypertext Preprocessor, język programowania, w którym napisany jest frontend Zabbixa i wiele innych aplikacji webowych
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach
RFC	Request for Comments: oficjalny dokument opisujący standard internetowy, na przykład sposób działania danego protokołu

Fiszka

◫◫ TYLKO ODCZYT

Ten sam wzorzec w trzech dialektach

```
grep 'a\+b' plik.txt                                    # BRE – + trzeba escapować
grep -E 'a+b' plik.txt                                   # ERE – + działa wprost, bez backslasha
grep -P '\d{3}-\d{2}-\d{4}' plik.txt                     # PCRE – klasy skrótowe \d, \w, \s
find . -regextype posix-extended -regex '.*\.(log|txt)$'  # find: własny dialekt domyślnie, ERE po zmianie
```

Ten sam pomysł zapisany w trzech niekompatybilnych składniach. Różnice między BRE, ERE i PCRE są tematem dalszej części strony — tu tylko widać, że ta sama para nawiasów i ten sam `+` znaczą co innego w każdym wierszu.

◫◫ **BRE** = Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami · **ERE** = Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem · **PCRE** = Perl Compatible Regular Expressions: odmiana wyrażeń regularnych z językiem Perl, najbogatsza w możliwości spośród trzech · **grep** = global regular expression print: szukaj wzorca w tekście · **find** = przeszukaj drzewo katalogów według nazwy, typu, daty albo rozmiaru

◫◫ TYLKO ODCZYT

Kotwice — początek, koniec i granica słowa

```
grep '^Error' plik.log
```

Znakiem `^` przypinamy wzorzec do **początku** linii, nie zużywając przy tym żadnego znaku.

Przykład:

```
Error: connection refused
```

```
grep 'Error$' plik.log
```

Znakiem `$` robimy to samo dla **końca** linii.

```
grep -P '\bError\b' plik.log      # PCRE — granica słowa
grep '\<Error\>' plik.log         # GNU BRE/ERE — ten sam efekt
```

Znakiem `\b` (PCRE) albo parą `\<` / `\>` (rozszerzenie GNU w BRE/ERE) dopasowujemy granicę słowa — złapiemy samodzielne „Error", nie złapiemy go w środku „ErrorCode".

PCRE = Perl Compatible Regular Expressions: odmiana wyrażeń regularnych z językiem Perl, najbogatsza w możliwości spośród trzech · **BRE** = Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami · **grep** = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Klasy znaków

```
grep '[aeiou]' plik.txt
```

Dopasowujemy dowolny **jeden** znak z podanego zestawu w nawiasach kwadratowych.

Przykład:

```
kot
```

```
grep '[^0-9]' plik.txt
```

Znak `^` zaraz po otwierającym `[` neguje zestaw — dopasowujemy wtedy dowolny znak **oprócz** cyfry.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Klasy POSIX

```
grep '[[:digit:]]' plik.txt
grep '[[:alpha:]]' plik.txt
grep '[[:space:]]' plik.txt
```

Odpowiedniki `[0-9]`, `[a-zA-Z]` i „biały znak", ale zdefiniowane semantycznie, nie przez zakres w tabeli kodowej — bezpieczne niezależnie od ustawionego locale.

□□ **POSIX** = Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach · **grep** = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Kwantyfikatory

```
grep -E 'ab*' plik.txt        # a, potem zero lub więcej b
grep -E 'ab+' plik.txt        # a, potem jeden lub więcej b
grep -E 'ab?' plik.txt        # a, potem zero lub jeden b
grep -E 'ab{2,4}' plik.txt    # a, potem od 2 do 4 razy b
```

Określają, ile razy ma się powtórzyć poprzedzający element wzorca. Działają wprost w ERE i PCRE; w BRE wymagają backslasha (`\+`, `\?`) albo w ogóle nie istnieją (`\{n,m\}` jako rozszerzenie GNU).

□□ **ERE** = Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem · **PCRE** = Perl Compatible Regular Expressions: odmiana wyrażeń regularnych z językiem Perl, najbogatsza w możliwości spośród trzech · **BRE** = Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami · **grep** = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Grupy

```
grep -E '(kot|pies)em' plik.txt
```

Nawiasy `()` (w BRE: `\(` `\)`) grupują fragment wzorca, dzięki czemu kwantyfikator albo alternatywa działa na całej grupie, a nie na pojedynczym znaku — tu dopasujemy „kotem" albo „psem".

□□ **BRE** = Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami · **grep** = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Backreferencje

```
echo "prosimy o o kontakt" | grep -E '\b(\w+)\s+\1\b'
```

`\1` odwołuje się do tego, co faktycznie dopasowała pierwsza grupa — nie do jej wzorca, tylko do konkretnego złapanego tekstu. Pozwala to wykryć zdublowane słowo.

Przykład:

```
prosimy o o kontakt
```

TYLKO ODCZYT

Alternatywa

```
grep -E 'blad|error|fail' plik.txt
```

`|` (ERE/PCRE) albo `\|` (BRE, rozszerzenie GNU) oznacza „albo to, albo tamto" — działa na całych gałęziach wzorca, więc `kot|pies` to „kot" ALBO „pies", nie „k, albo o, albo t".

ERE = Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem · **BRE** = Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami ·**grep** = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Escapowanie

```
grep -E '10\.20\.10\.[0-9]{1,3}' plik.txt
```

`.` w regexie znaczy „dowolny znak", więc dosłowną kropkę trzeba zescapować jako `\.` — inaczej `10.20.10.1` dopasuje też `10a20a10a1`. Ukośnik `/` nie jest specjalny w samym regexie, ale w `sed s/wzorzec/zamiana/` pełni rolę separatora, więc dosłowny `/` w treści trzeba escapować albo zmienić separator (`sed 's#/etc/passwd#/etc/shadow#'`).

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Porównanie dialektów

Cecha	BRE	ERE	PCRE

`+` `?` `\`	`(` `)` `{` `}`	wymagają `\`	działają wprost
`\d` `\w` `\s` (klasy skrótowe)	brak	brak (GNU: `\w` / `\s` jako rozszerzenie)	tak
lookahead/lookbehind `(?=)` `(?!)`	brak	brak	tak
grupy nieprzechwytujące `(?:...)`	brak	brak	tak
backreferencje `\1`	tak	tak	tak
kto używa	`grep`, `sed` (domyślnie)	`grep -E`, `sed -E`, `awk`, `egrep`	`grep -P`, Perl, PHP, część `pcregrep`

PHP = PHP: Hypertext Preprocessor, język programowania, w którym napisany jest frontend Zabbixa i wiele innych aplikacji webowych

TYLKO ODCZYT

Który dialekt ma które narzędzie

Narzędzie	Dialekt domyślny	Jak przełączyć
`grep`	BRE	`-E` (ERE), `-P` (PCRE, nie wszędzie)
`sed`	BRE	`-E` lub `-r` (ERE, GNU)
`awk`	ERE	brak przełącznika — zawsze ERE
`find -regex`	emacs (własny dialekt)	`-regextype posix-extended` itd.
`less`, `vim` (`/wzorzec`)	własne, zbliżone do BRE z modyfikacjami	zależnie od narzędzia

Uważaj

- `[a-z]` w locale innym niż `C`/`POSIX` bywa nieprzewidywalne — w niektórych locale sortuje litery inaczej niż alfabetycznie (np. wielkie i małe litery przeplecione) i może pomijać albo łapać znaki spoza oczekiwanego zakresu. `[[:lower:]]` jest bezpieczne niezależnie od locale.
- `grep -E '\d+'` bez `-P` nie zwróci błędu, tylko po cichu nie zadziała tak, jak myślisz — `\d` w ERE nie jest zdefiniowane jako klasa cyfr.

ERE = Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem · **PCRE** = Perl Compatible Regular Expressions: odmiana wyrażeń regularnych z językiem Perl, najbogatsza w możliwości spośród trzech · **BRE** = Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami

Z życia wzięte

Z ŻYCIA WZIĘTE

Czwartek, 11:20. Tomek na `mon01` pisze `grep -E '\d+' zabbix.log`, żeby wyłapać komunikaty z numerami zdarzeń, i dostaje pustą listę mimo że numery na pewno tam są. Ola tłumaczy, że `\d` to składnia PCRE, a nie ERE, więc `-E` po cichu traktuje `\d` inaczej, niż Tomek zakładał. Po zmianie na `grep -P '\d+'` komenda w końcu łapie to, czego szukał.

Po co to istnieje

Pojęcie: wyrażenie regularne. Zapis opisujący kształt tekstu, a nie jego dosłowną treść — sekwencja znaków zwykłych (dopasowują same siebie) i metaznaków (niosą znaczenie w rodzaju „dowolny znak", „powtórz poprzedni element", „albo to, albo tamto"). Silnik wyrażeń regularnych bierze taki zapis i sprawdza, czy fragment tekstu do niego pasuje. Problem w tym, że nie istnieje jeden uniwersalny zapis — różne narzędzia rozumieją te same symbole inaczej, i właśnie to jest temat tej strony.

Chcesz znaleźć w logu wszystkie linie wyglądające jak adres IP, niezależnie od konkretnych liczb. Albo zwalidować, że każda linia CSV zaczyna się datą `RRRR-MM-DD`. Zwykłe dopasowanie tekstu tego nie załatwi — potrzebujesz opisać **kształt** tekstu: „cztery grupy cyfr oddzielone kropkami", „dokładnie dziesięć cyfr", „słowo powtórzone dwa razy pod rząd".

Problem praktyczny, który tę stronę uzasadnia, jest inny: ten sam zapis wyrażenia regularnego **nie znaczy tego samego** w różnych narzędziach. Wzorzec, który działa w `grep -E`, może dać zupełnie inny wynik w zwykłym `grep`. To nie jest kwestia stylu — to inny język, tylko podobnie wyglądający.

Analogia

Pomyśl o **szablonie do wycinania** — takim z metalu albo twardego plastiku, z wyciętym otworem w konkretnym kształcie, którego używa się do przycinania materiału albo kontroli jakości części.

Wyrażenie regularne jest tym szablonem: kształt otworu określa, co ma przejść, a co nie. Przesuwasz tekst pod spodem i sprawdzasz, gdzie fragment pasuje do otworu. Jeśli otwór ma kształt „cztery cyfry, myślnik, dwie cyfry, myślnik, dwie cyfry", to przepuści `2026-09-17`, ale nie przepuści `17 września 2026`.

Teraz ograniczenie, które ta analogia niesie i które tłumaczy realny problem: **różne fabryki produkują szablony według różnych norm**, wyglądających niemal identycznie, ale z nieco innymi wymiarami otworów. Szablon BRE i szablon ERE oba mają otwór opisywany symbolem `+`, ale w normie BRE ten symbol jest tylko dekoracją (zwykły znak), a w normie ERE jest częścią kształtu otworu (kwantyfikator „jeden lub więcej"). Przyłożysz szablon jednej normy myśląc, że działa jak drugi — i przepuścisz zupełnie inny zestaw elementów. Śruby prawie pasują, ale nie do końca — i to „prawie" jest źródłem błędów trudnych do zauważenia, bo wzorzec **nie zwraca błędu składni**, tylko cicho dopasowuje coś innego.

Jak to działa

Rozwiń: jak to działa pod spodem

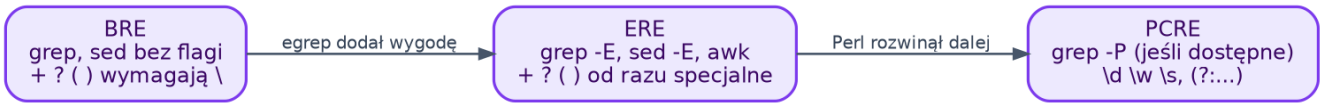
> **Pojęcie: dialekt wyrażeń regularnych.** Konkretny zestaw reguł mówiący, które symbole są metaznakami i co dokładnie znaczą. Te same znaki — `+`, `?`, `|`, `(`, `)` — w jednym dialekcie są zwykłymi, literalnymi znakami, a w innym kwantyfikatorami czy operatorami grupowania. W Linuksie spotykasz trzy główne dialekty wyrażeń regularnych: BRE, ERE i PCRE, opisane niżej — każdy z innym zestawem reguł pod tymi samymi symbolami.

Dlaczego w ogóle są trzy dialekty. Wczesny Unix (`ed`, potem `grep`) miał prosty silnik regexów, który później sformalizowano jako POSIX BRE (Basic Regular Expressions). W BRE tylko garstka znaków ma specjalne znaczenie bez ukośnika: `.` `*` `^` `$` `[` `]`. To, co „rozszerzało" składnię — grupowanie `()`, alternatywa `|`, kwantyfikatory `+` `?` `{}` — wymagało

backslasha, żeby przestać być znakiem literalnym.

Okazało się to niewygodne, więc powstał `egrep` z drugim dialektem — POSIX ERE (Extended Regular Expressions), gdzie te same symbole są od razu specjalne, bez backslasha. `egrep` jest dziś przestarzały (POSIX go nie zaleca) — zastąpiło go `grep -E`.

Trzeci dialekt, PCRE (Perl Compatible Regular Expressions), pochodzi z Perla, gdzie regexy rozwinęły się dużo dalej: klasy skrótowe `\d` (cyfra), `\w` (znak słowa), `\s` (biała spacja), asercje wyprzedzające `(?=...)`, grupy nieprzechwytujące `(?:...)`. Biblioteka PCRE trafiła też do `grep -P`, PHP i innych narzędzi — ale nie jest gwarantowana wszędzie: część buildów `grep` na BSD nie ma `-P` w ogóle.



Trzy dialekty regexów w kolejności powstania: BRE, ERE i PCRE, z narzędziami które ich używają (kliknięcie otwiera pełny rozmiar)

Które narzędzie, który dialekt — to jest tabela w Fiszce, ale zapamiętaj regułę praktyczną: `grep` i `sed` bez dodatkowej flagi to BRE, `awk` zawsze ERE (nie ma trybu BRE), `-E` w `grep`/`sed` przełącza na ERE, `-P` w `grep` na PCRE tam, gdzie jest dostępne.

Kotwice. `^` oznacza początek linii, `$` koniec linii — działają tak samo we wszystkich trzech dialektach. `\b` (granica słowa) jest częścią PCRE i rozszerzeń GNU; POSIX BRE/ERE formalnie tego nie definiuje, ale GNU grep/sed obsługują `\<` i `\>` jako swój odpowiednik („początek słowa" i „koniec słowa").

Klasy znaków. `[abc]` oznacza „jeden z tych znaków", `[^abc]` — negację (uwaga: `^` tu znaczy co innego niż na początku wzorca). Klasy POSIX, jak `[[:digit:]]`, `[[:alpha:]]`, `[[:space:]]`, `[[:upper:]]`, są bezpieczniejsze niż ręczne zakresy typu `[a-z]`, bo uwzględniają bieżące locale — w locale innym niż `C` zakres `[a-z]` może nie obejmować dokładnie tego, czego oczekujesz, a `[[:lower:]]` zawsze oznacza „małe litery" danego locale.

Kwantyfikatory. `*` — zero lub więcej, `+` — jeden lub więcej (ERE/PCRE; w BRE `\+` jako rozszerzenie GNU), `?` — zero lub jeden (ERE/PCRE; w BRE `\?`), `{n,m}` — od n do m powtórzeń.

> **Pojęcie: zachłanność.** Domyślne zachowanie kwantyfikatorów (`*`, `+`, `{n,m}`) polegające na dopasowaniu jak najdłuższego możliwego fragmentu tekstu, jaki wzorzec dopuszcza — nie najkrótszego. Silnik oddaje dopasowane znaki z powrotem tylko wtedy,

> gdy to konieczne, żeby reszta wzorca w ogóle mogła się dopasować. Przy tekście z wieloma powtórzeniami tego samego fragmentu zachłanność często łapie znacznie więcej, niż zamierzałeś — przykład jest w *Częste błędy*.

Grupy i backreferencje. Nawiasy `()` (w BRE: `\(` `\)`) grupują fragment wzorca, a `\1`, `\2` odwołują się do tego, co dana grupa faktycznie dopasowała — nie do wzorca grupy, tylko do konkretnego złapanego tekstu. To pozwala wykryć duplikat: `\b(\w+)\s+\1\b` znajdzie „the the", bo `\1` musi być identyczne z tym, co złapała pierwsza grupa.

Alternatywa. `|` (ERE/PCRE) lub `\|` (BRE, rozszerzenie GNU) oznacza „albo to, albo tamto" — działa na całych gałęziach wzorca, nie tylko na pojedynczych znakach, więc `kot|pies` to „kot" ALBO „pies", nie „k, albo o, albo t".

Escapowanie kropki i ukośnika. `.` w regexie oznacza „dowolny znak", więc w adresie IP trzeba go escapować jako `\.`, żeby dopasować dosłowną kropkę — inaczej `10.20.10.1` dopasuje też `10a20a10a1`. Ukośnik `/` nie jest specjalny w samym regexie, ale w `sed s/wzorzec/zamiana/` pełni rolę separatora, więc dosłowny `/` w treści trzeba escapować (`\/`) albo zmienić separator: `sed 's#/etc/passwd#/etc/shadow#'`.

Krok po kroku

1. Sprawdź, czy linia wygląda jak adres IPv4.

```
echo "app01 10.20.10.41 aktywny" | grep -E '([0-9]{1,3}\.){3}[0-9]{1,3}'
```

```
app01 10.20.10.41 aktywny
```

Zastrzeżenie: wzorzec nie sprawdza, czy liczby mieszczą się w zakresie 0–255 — `999.999.999.999` też przejdzie. Pełna walidacja zakresu wymaga dużo bardziej rozbudowanego wzorca albo parsowania liczbowego w skrypcie.

2. Znajdź linie z datą w formacie ISO.

```
grep -E '^[0-9]{4}-[0-9]{2}-[0-9]{2}' dziennik.txt
```

3. Wyciągnij linie komentarza z pliku konfiguracyjnego.

```
grep -E '^\s*#' /etc/nginx/nginx.conf
```

Kotwica `^` plus `\s*` obsługuje też komentarze wcięte spacjami czy tabulatorami, nie tylko te od `#` w pierwszej kolumnie.

4. Znajdź puste linie (do usunięcia albo policzenia).

```
grep -c -E '^\s*$' plik.txt
```

5. Wykryj zdublowane słowo — częsty błąd w tekście pisanym.

```
echo "prosimy o o kontakt" | grep -E '\b(\w+)\s+\1\b'
```

```
prosimy o o kontakt
```

6. Ten sam wzorzec w BRE i ERE — porównaj składnię.

```
sed 's/\(kot\)\1/PIES/' <<< "kotkot śpi"
sed -E 's/(kot)\1/PIES/' <<< "kotkot śpi"
```

Obie komendy dają ten sam wynik `PIES śpi` — różnica jest wyłącznie w tym, czy nawiasy grupujące trzeba escapować.

Częste błędy

`grep 'a+b' plik` bez `-E`, oczekiwanie kwantyfikatora.

```
grep 'error+' app.log
```

W BRE `+` jest znakiem literalnym, więc wzorzec szuka dosłownego ciągu `error+`, nie „error z powtórzoną ostatnią literą". Wynik będzie pusty, mimo że intencją było złapać dowolną liczbę powtórzeń. Rozwiązanie: `grep -E 'erro+r' app.log` albo escapowanie w BRE: `grep 'erro\+r' app.log`.

`grep -E '\d+'` — cichy brak dopasowania.

```
grep -E '\d{3}-\d{4}' kontakty.txt
```

`\d` nie istnieje w POSIX ERE — GNU grep często potraktuje `\d` jako dosłowne `d`, więc komenda się wykona, ale nie znajdzie tego, czego szukasz, bez żadnego ostrzeżenia. Klasy skrótowe `\d`, `\w`, `\s` należą do PCRE:

```
grep -P '\d{3}-\d{4}' kontakty.txt      # jeśli grep -P jest dostępne
grep -E '[0-9]{3}-[0-9]{4}' kontakty.txt # przenośna wersja bez -P
```

Zapomniana kropka w adresie IP dopasowuje za dużo.

```
grep -E '10.20.10.1' log.txt
```

Niesescapowana `.` dopasowuje dowolny znak, więc wzorzec złapie też `10a20b10c1` czy podciąg większej liczby. W logu z dużą ilością cyfr to realnie generuje fałszywe trafienia. Popraw:

```
grep -E '10\.20\.10\.1' log.txt
```

Zachłanność `.*` łapie za dużo.

```
echo '<b>pogrubione</b> i <i>kursywa</i>' | grep -oE '<.*>'
```

```
<b>pogrubione</b> i <i>kursywa</i>
```

Zamiast pojedynczego znacznika `<b>` dostajesz cały ciąg od pierwszego `<` do ostatniego `>`, bo `.*` jest zachłanne. Rozwiązanie zależy od narzędzia: w PCRE dopisuje się `?` dla trybu leniwego (`<.*?>`), w ERE/BRE bez trybu leniwego lepiej ograniczyć klasę znaków: `<[^>]*>` — „dowolny znak oprócz `>`", co zatrzyma się na pierwszym zamknięciu.

Próba pełnej walidacji e-maila jednym wyrażeniem regularnym.

Da się napisać wzorzec łapiący 95% poprawnych adresów, ale specyfikacja e-maila (RFC 5322) jest na tyle skomplikowana (komentarze w nawiasach, cudzysłowy, znaki specjalne), że w pełni poprawny regex liczy setki znaków i i tak ma wyjątki. W praktyce prosty wzorzec `^[^@\s]+@[^@\s]+\.[^@\s]+$` wystarcza do odsiania literówek, a właściwą walidację (czy adres istnieje) robi się przez wysyłkę potwierdzenia, nie przez regex.

□□ Ćwiczenia

> Wszystko poniżej wykonujesz na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — kilka wzorców na jednym pliku

Przygotuj plik testowy i dopasuj kilka wzorców.

```
cd /tmp
cat > dane.txt <<'EOF'
2026-09-17 app01 10.20.10.41 status: aktywny
# to jest komentarz
2026-13-45 bkp01.zielona.example 10.20.10.999 status: blad

prosimy o o pilny kontakt w tej sprawie
kontakt: jan.nowak@zielona.example
EOF
```

Zadania:

1. Wyciągnij same daty (format `RRRR-MM-DD`, bez sprawdzania, czy miesiąc/dzień są sensowne).
2. Znajdź linię komentarza.
3. Znajdź linię pustą.
4. Znajdź zdublowane słowo.
5. Napisz wzorzec ERE, który wyciągnie adresy IPv4 (bez walidacji zakresu 0–255) i sprawdź, że złapie też błędny `10.20.10.999`.

Odpowiedź

```
grep -oE '^[0-9]{4}-[0-9]{2}-[0-9]{2}' dane.txt
grep -E '^\s*#' dane.txt
grep -E '^\s*$' dane.txt
grep -E '\b(\w+)\s+\1\b' dane.txt
grep -oE '([0-9]{1,3}\.){3}[0-9]{1,3}' dane.txt
```

Ostatnia komenda złapie zarówno `10.20.10.41`, jak i `10.20.10.999` — wzorzec sprawdza tylko kształt, nie zakres liczbowy. Żeby odrzucić `999`, trzeba rozpisać każdy oktet jako `(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)` — działa, ale czytelność spada, i to jest moment, w którym warto rozważyć walidację poza regexem (np. w Pythonie przez `ipaddress`).

Pytania kontrolne

1. Masz wzorzec `sed 's/(kot)/PIES/' plik.txt` i nie działa — `sed` zgłasza błąd albo nie robi podstawienia. Dlaczego?

Odpowiedź

`sed` bez `-E` (albo `-r`) działa w trybie BRE, gdzie nawiasy grupujące muszą być escapowane: `\(kot\)`. Niesescapowane `(` i `)` w BRE są znakami literalnymi, więc wzorzec szuka dosłownego ciągu `(kot)` z nawiasami w tekście. Poprawka: `sed 's/\(kot\)/PIES/'` albo przejście na ERE: `sed -E 's/(kot)/PIES/'`.

2. Czym różni się `[a-z]` od `[[:lower:]]` i kiedy ta różnica ma znaczenie?

Odpowiedź

`[a-z]` to zakres oparty na kolejności znaków w tabeli kodowej bieżącego locale — w locale innym niż `C`/`POSIX` ta kolejność bywa inna niż w ASCII (np. wielkie i małe litery przeplecione w porządku sortowania), więc zakres może objąć więcej lub mniej znaków, niż zakładasz, albo w ogóle rzucić błędem przy niektórych narzędziach. `[[:lower:]]` to klasa POSIX zdefiniowana semantycznie jako „małe litery danego locale" i zachowuje się przewidywalnie niezależnie od ustawień. Różnica ujawnia się głównie na systemach z ustawionym locale innym niż domyślne `en_US.UTF-8`/`C`, np. `pl_PL.UTF-8`.

3. Dlaczego `grep -E '<.*>'` na linii z dwoma znacznikami HTML zwraca więcej, niż oczekujesz?

Odpowiedź

Kwantyfikator `*` jest domyślnie zachłanny — silnik regexu próbuje dopasować jak najdłuższy fragment spełniający wzorzec. Przy linii `<b>tekst</b> i <i>kursywa</i>` wzorzec `<.*>` dopasuje wszystko od pierwszego `<` do ostatniego `>` w linii, bo to wciąż spełnia warunek „zaczyna się od `<`, kończy na `>`, cokolwiek pomiędzy". Rozwiązanie: ograniczyć klasę znaków do „nie `>`": `<[^>]*>`.

4. Masz do wyboru `grep -P '\d+'` i `grep -E '[0-9]+'` do wyciągnięcia liczb z pliku konfiguracyjnego, który będzie uruchamiany na różnych serwerach, w tym części z systemami, gdzie `grep` nie ma wsparcia dla PCRE. Którego użyjesz?

Odpowiedź

`grep -E '[0-9]+'`. Flaga `-P` (PCRE) nie jest gwarantowana we wszystkich buildach `grep` — brakuje jej np. w części systemów BSD i w niektórych minimalnych obrazach kontenerowych. `[0-9]+` w ERE daje identyczny efekt praktyczny i działa wszędzie, gdzie jest `grep -E`, czyli praktycznie wszędzie. Zasada ogólna: PCRE dla wygody na własnej maszynie, POSIX ERE dla

przenośności w skryptach, które mają jeździć po cudzych serwerach.

Powiązane: grep: wyszukiwanie w treści plików · Cudzysłowy, escapowanie i ekspansja · Anatomia komendy: polecenie, opcje, argumenty

Źródła: `man 7 regex` · `man 1 grep` · `man 1 sed` · POSIX.1-2024, rozdział 9 (Regular Expressions) · GNU `sed`/`grep` manual (rozszerzenia GNU: `\w`, `\s`, `\<`, `\>`)

Revision #18
Created 2026-09-17 15:06:54 UTC by Tomasz Konieczny
Updated 2026-10-03 19:07:54 UTC by Tomasz Konieczny