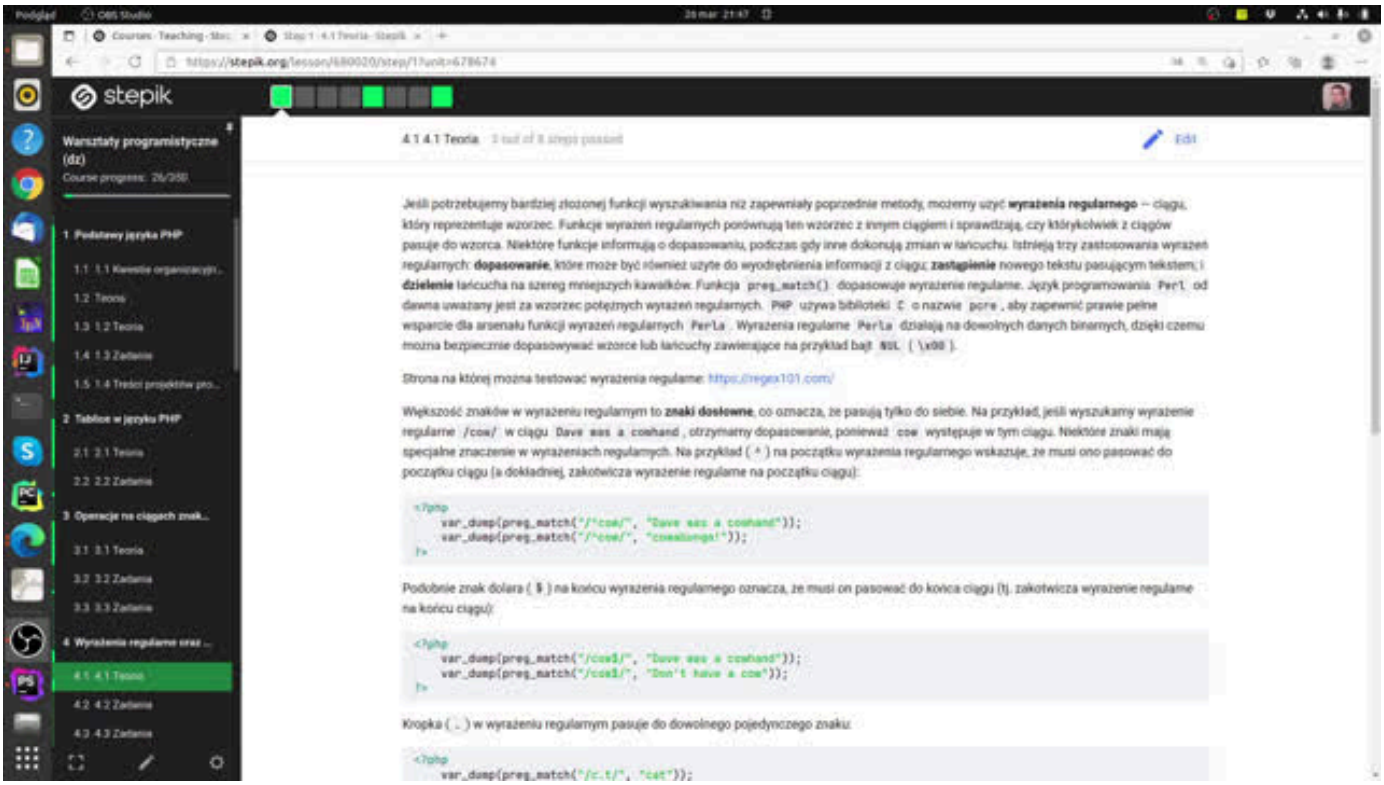
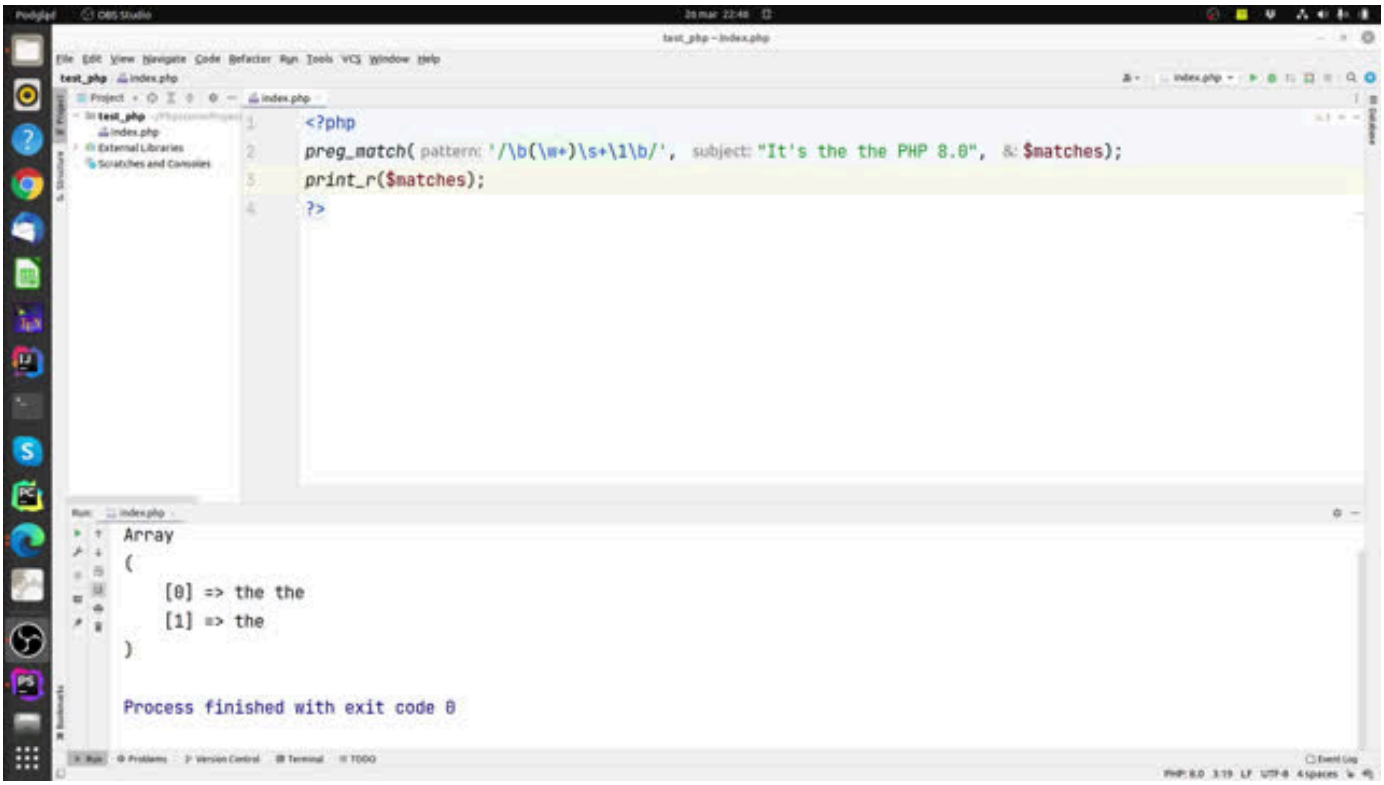
9.2 9.2 Teoria

Step 1



To watch this video please visit https://stepik.org/lesson/680020/step/1

Step 2



To watch this video please visit https://stepik.org/lesson/680020/step/2

Step 3

Jeśli potrzebujemy bardziej złożonej funkcji wyszukiwania niż zapewniały poprzednie metody, możemy użyć **wyrażenia regularnego** — ciągu, który reprezentuje wzorzec. Funkcje wyrażeń regularnych porównują ten wzorzec z innym ciągiem i sprawdzają, czy którykolwiek z ciągów pasuje do wzorca. Niektóre funkcje informują o dopasowaniu, podczas gdy inne dokonują zmian w łańcuchu. Istnieją trzy zastosowania wyrażeń regularnych: **dopasowanie**, które może być również użyte do wyodrębnienia informacji z ciągu; **zastąpienie** nowego tekstu pasującym tekstem; i **dzielenie** łańcucha na szereg mniejszych kawałków. Funkcja `preg_match()` dopasowuje wyrażenie regularne. Język programowania `Perl` od dawna uważany jest za wzorzec potężnych wyrażeń regularnych. `PHP` używa biblioteki `C` o nazwie `pcre`, aby zapewnić prawie pełne wsparcie dla arsenału funkcji wyrażeń regularnych `Perla`. Wyrażenia regularne `Perla` działają na dowolnych danych binarnych, dzięki czemu można bezpiecznie dopasowywać wzorce lub łańcuchy zawierające na przykład bajt `NUL` (`\x00`).

Strona na której można testować wyrażenia regularne: https://regex101.com/ (https://regex101.com/)

Większość znaków w wyrażeniu regularnym to **znaki dosłowne**, co oznacza, że pasują tylko do siebie. Na przykład, jeśli wyszukamy wyrażenie regularne `/cow/` w ciągu `Dave was a cowhand`, otrzymamy dopasowanie, ponieważ `cow` występuje w tym ciągu. Niektóre znaki mają specjalne znaczenie w wyrażeniach regularnych. Na przykład (`^`) na początku wyrażenia regularnego wskazuje, że musi ono pasować do początku ciągu (a dokładniej, zakotwicza wyrażenie regularne na początku ciągu):

```
<?php
    var_dump(preg_match("/^cow/", "Dave was a cowhand"));
    var_dump(preg_match("/^cow/", "cowabunga!"));
?>
```

Podobnie znak dolara (`$`) na końcu wyrażenia regularnego oznacza, że musi on pasować do końca ciągu (tj. zakotwicza wyrażenie regularne na końcu ciągu):

```
<?php
    var_dump(preg_match("/cow$/", "Dave was a cowhand"));
    var_dump(preg_match("/cow$/", "Don't have a cow"));
?>
```

Kropka (`.`) w wyrażeniu regularnym pasuje do dowolnego pojedynczego znaku:

```
<?php
    var_dump(preg_match("/c.t/", "cat"));
    var_dump(preg_match("/c.t/", "cut"));
    var_dump(preg_match("/c.t/", "c t"));
    var_dump(preg_match("/c.t/", "bat"));
    var_dump(preg_match("/c.t/", "ct"));
?>
```

Jeśli chcemy dopasować jeden ze znaków specjalnych (zwanych **metaznakiem**), to musimy odnieść się do niego za pomocą odwrotnego ukośnika `\` w znakach **pojedynczego cudzysłowia** :

```
<?php
    var_dump(preg_match('/\$5.00/', "Your bill is $5.00 exactly"));
    var_dump(preg_match('/$5.00/', "Your bill is $5.00 exactly"));
?>
```

W wyrażeniach regularnych domyślnie rozróżniana jest wielkość liter, więc wyrażenie regularne `/cow/`
nie pasuje do ciągu `COW`. Jeśli chcemy przeprowadzić dopasowanie bez rozróżniania wielkości liter,
należy określić odpowiednią flagę, co pojawi się w dalszej części teorii. Jak dotąd nie zrobiliśmy niczego, czego nie moglibyśmy zrobić ze znanymi funkcjami łańcuchowymi, takimi jak `strstr()`. Prawdziwa moc wyrażeń regularnych pochodzi z ich możliwość określenia abstrakcyjnych wzorców, które mogą pasować do wielu różnych sekwencji znaków.
W wyrażeniu regularnym można określić trzy podstawowe typy wzorców abstrakcyjnych:

- Zestaw dopuszczalnych znaków, które mogą pojawić się w ciągu (np. alfabetyczne znaki, cyfry, określone znaki interpunkcyjne)
- Zestaw alternatyw dla ciągu (np. „com”, „edu”, „net” lub „org”)
- Powtarzająca się sekwencja w ciągu (np. co najmniej jeden, ale nie więcej niż pięć liczb)

Te trzy rodzaje wzorów można łączyć na niezliczone sposoby, aby tworzyć wyrażenia regularne pasujące do takich rzeczy, jak prawidłowe numery telefonów i adresy URL.

Step 4

Aby określić zestaw dopuszczalnych znaków we wzorcu, możemy samodzielnie zbudować klasę znaków lub użyć predefiniowanej klasy znaków (np. alfanumeryczne, wielkie litery itd.). Możemy zbudować własną klasę postaci, umieszczając dopuszczalne znaki w nawiasach kwadratowych:

```
<?php
    var_dump(preg_match("/c[aeiou]t/", "I cut my hand"));
    var_dump(preg_match("/c[aeiou]t/", "This crusty cat"));
    var_dump(preg_match("/c[aeiou]t/", "What cart"));
    var_dump(preg_match("/c[aeiou]t/", "14ct gold"));
?>
```

Aparat wyrażeń regularnych znajduje `c`, a następnie sprawdza, czy następny znak to jeden z `a`, `e`, `i`, `o` lub `u`. Jeśli nie jest samogłoską, dopasowanie kończy się niepowodzeniem i silnik wraca do szukania kolejnego `c`. Jeśli samogłoska zostanie znaleziona, silnik sprawdza, czy następnym znakiem jest `t`. Jeśli tak, silnik jest na końcu dopasowania i zwraca prawdę. Jeśli następnym znakiem nie jest `t`, silnik wraca do szukania kolejnego `c`.

Możemy zanegować klasę znaków za pomocą karetki (`^`) na początku klasy:

```
<?php
    var_dump(preg_match("/c[^aeiou]t/", "I cut my hand"));
    var_dump(preg_match("/c[^aeiou]t/", "Reboot chthon"));
    var_dump(preg_match("/c[^aeiou]t/", "14ct gold"));
?>
```

W tym przypadku aparat wyrażeń regularnych szuka `c`, po którym następuje znak, który nie jest samogłoską, po którym następuje `t`.

Możemy zdefiniować zakres znaków za pomocą łącznika (`-`). Upraszcza to klasy znaków, takie jak „wszystkie litery” i „wszystkie cyfry”:

```
<?php
    var_dump(preg_match("/[0-9]%/", "we are 25% complete"));
    var_dump(preg_match("/[0123456789]%/", "we are 25% complete"));
    var_dump(preg_match("/[a-z]t/", "11th"));
    var_dump(preg_match("/[a-z]t/", "cat"));
    var_dump(preg_match("/[a-z]t/", "PIT"));
    var_dump(preg_match("/[a-zA-Z]!/", "11!"));
    var_dump(preg_match("/[a-zA-Z]!/", "stop!"));
?>
```

Kiedy określamy klasę znaków, niektóre znaki specjalne tracą swoje znaczenie, podczas gdy inne nabierają nowego znaczenia. W szczególności znak `$` i kropka tracą swoje znaczenie w klasie znaków, podczas gdy znak `^` neguje klasę znaków, jeśli jest pierwszym znakiem po otwartym nawiasie. Na przykład `[^\]]` dopasowuje dowolny znak nawiasu zamykającego, a `[$.^]` dopasowuje dowolny znak dolara, kropkę lub `^`. Różne biblioteki wyrażeń regularnych definiują skróty do klas znaków, w tym cyfry, znaki alfabetyczne i białe znaki.

Możemy również użyć pionowej kreski (`|`), aby określić alternatywy w wyrażeniu regularnym:

```
<?php
    var_dump(preg_match("/cat|dog/", "the cat rubbed my legs"));
    var_dump(preg_match("/cat|dog/", "the dog rubbed my legs"));
    var_dump(preg_match("/cat|dog/", "the rabbit rubbed my legs"));
?>
```

Pierwszeństwo alternatywy może być niespodzianką: w wyrażeniu `/^cat|dog$/` wybiera z `^cat` i `dog$`, co oznacza, że pasuje do linii zaczynającej się od `cat` lub kończącej się na `dog`. Jeśli chcemy, aby linia zawierała tylko `cat` lub `dog`, musimy użyć wyrażenia regularnego `/^(cat|dog)$/`.

Możemy łączyć klasy znaków i alternatywy, aby na przykład sprawdzić ciągi, które nie zaczynają się od wielkiej litery:

```
<?php
    var_dump(preg_match("/^([a-z]|[0-9])/", "The quick brown fox"));
    var_dump(preg_match("/^([a-z]|[0-9])/", "jumped over"));
    var_dump(preg_match("/^([a-z]|[0-9])/", "10 lazy dogs"));
?>
```

Aby określić powtarzający się wzorzec, należy użyć **kwantyfikatora**. Kwantyfikator podąża za powtarzającym się wzorcem i mówi, ile razy wzorzec ma zostać powtórzony. Poniższa tabela przedstawia kwantyfikatory obsługiwane przez wyrażenia regularne `PHP`.

Kwantyfikator	Znaczenie
`?`	0 lub 1
`*`	0 lub więcej
`+`	1 lub więcej
`{n}`	Dokładnie n razy
`{n,m}`	Co najmniej n i co najwyżej m razy
`{n, }`	Co najmniej n razy

Aby powtórzyć pojedynczy znak, po prostu należy umieścić kwantyfikator po znaku:

```
<?php
    var_dump(preg_match("/ca+t/", "caaaaaaat"));
    var_dump(preg_match("/ca+t/", "ct"));
    var_dump(preg_match("/ca?t/", "caaaaaaat"));
    var_dump(preg_match("/ca*t/", "ct"));
?>
```

Dzięki kwantyfikatorom i klasom znaków możemy zrobić coś pożytecznego, na przykład dopasowanie prawidłowych numerów telefonów:

```
<?php
    var_dump(preg_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/", "303-555-1212"));
    var_dump(preg_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/", "64-9-555-1234"));
?>
```

Możemy użyć nawiasów, aby pogrupować razem wystąpienia wyrażenia regularnego, aby były traktowane jako pojedyncza jednostka zwana **podwzorcem**:

```
<?php
    var_dump(preg_match("/a (very )+big dog/", "it was a very very big dog"));
    var_dump(preg_match("/^(cat|dog)$/", "cat"));
    var_dump(preg_match("/^(cat|dog)$/", "dog"));
?>
```

Nawiasy powodują również przechwycenie podciągu pasującego do podwzorca. Jeśli przekażemy tablicę jako trzeci argument do funkcji dopasowania, tablica zostanie wypełniona wszelkimi przechwyconymi podciągami:

```
<?php
    preg_match("/([0-9]+)/", "You have 42 magic beans", $captured);
    print_r($captured);
?>
```

Zerowy element tablicy jest ustawiany na cały dopasowywany ciąg. Pierwszy element to podciąg pasujący do pierwszego podwzorca (jeśli taki istnieje), drugi element to podłańcuch pasujący do drugiego podwzorca i tak dalej.

Wyrażenia regularne w stylu `Perla` emulują składnię `Perla` dla wzorców, co oznacza, że każdy wzorzec musi być ujęty w parę ograniczników. Tradycyjnie używany jest znak ukośnika (`/`); na przykład `/wzór/` . Jednak do rozgraniczenia wzorca w stylu `Perla` można użyć dowolnego znaku niealfanumerycznego innego niż znak ukośnika odwrotnego (`\`). Jest to przydatne do dopasowywania ciągów zawierających ukośniki, takich jak nazwy plików. Na przykład poniższe wyrażenia są równoważne:

```
<?php
    preg_match("/\/usr\/local\//", "/usr/local/bin/perl", $captured);
    preg_match("#/usr/local/#", "/usr/local/bin/perl", $captured1);
    print_r($captured);
    print_r($captured1);
?>
```

Nawiasy (`()`), nawiasy klamrowe (`{}`), nawiasy kwadratowe (`[]`) i nawiasy sześcienne (`<>`) mogą być używane jako ograniczniki wzorca:

```
<?php
    preg_match("{/usr/local/}", "/usr/local/bin/perl", $captured);
    print_r($captured);
?>
```

Step 5

Kropka (`.`) pasuje do dowolnego znaku z wyjątkiem znaku nowej linii (`\n`). Znak dolara (`$`) pasuje na końcu ciągu lub, jeśli ciąg kończy się znakiem nowej linii, tuż przed tym znakiem nowej linii:

```
<?php
    preg_match("/is (.*)$/", "the key is in my pants", $captured);
    print_r($captured);
?>
```

Poniższa tabela przedstawia wyrażenia regularne zgodne z `Perlem` , które definiują pewną liczbę nazwanych zestawów znaków, których można używać w klasach znaków. Każda klasa `[: coś :]` może być użyta zamiast znaku w klasie postaci. Na przykład, aby znaleźć dowolny znak będący cyfrą, wielką literą lub znakiem (`@`), można użyć następującego wyrażenia regularnego:

```
[@[:digit:][:upper:]]
```

Nie możemy jednak użyć klasy postaci jako punktu końcowego zakresu:

```
<?php
    preg_match("/[A-[:lower:]]/", "string", $captured);
    print_r($captured);
?>
```

Uwaga: Zakres znaków może zależeć od ustawień regionalnych komputera.

Niektóre ustawienia regionalne traktują pewne sekwencje znaków tak, jakby były pojedynczym znakiem — są to tak zwane **sekwencje zestawiania**. Aby dopasować jedną z tych wieloznakowych sekwencji w klasie znaków, należy uwzględnić ją w `[.` oraz `.]` . Na przykład, jeśli ustawienia regionalne mają sekwencję zestawiania `ch` , możemy dopasować `s` , `t` lub `ch` do następującej klasy znaków:

```
[st[.ch.]]
```

Ostatnim rozszerzeniem klas znaków jest klasa równoważności, którą określamy, umieszczając znak w obrębie `[=` i `=]` . Klasy równoważności dopasowują znaki, które mają taką samą kolejność zestawiania, jak zdefiniowano w bieżących ustawieniach regionalnych. Na przykład ustawienia regionalne mogą definiować `a` , `á` i `ä` jako mające ten sam priorytet sortowania. Aby dopasować dowolny z nich, klasa równoważności to `[=a=]` .

Klasa	Opis	Znaczenie
`[:alnum:]`	Znaki alfanumeryczne	`[0-9a-zA-Z]`
`[:alpha:]`	Znaki alfabetu	`[a-zA-Z]`
`[:ascii:]`	Znaki z tabeli ASCII	`[\x01-\x7F]`
`[:blank:]`	Białe znaki (spacja oraz tabulatory)	`[ \t]`

`[:cntrl:]`	Znaki kontrolne	`[\x01 - \x1F]`
`[:digits:]`	Liczby	`[0-9]`
`[:graph:]`	Znaki widoczne z wyjątkiem znaków ze `[:space:]`	`[^\x01-\x20]`
`[:lower:]`	Małe litery	`[a-z]`
`[:print:]`	Znaki widoczne z białymi znakami	`[\t\x20-\xFF]`
`[:punct:]`	Znaki interpunkcyjne	`[-!"#$%&'()*+,./:;<=>?@[\ \\]^_'{|}~]`
`[:space:]`	Znak nowej linii, powrotu karetki, tabulatory oraz spacja	`[\n\r\t  \x0B]`
`[:upper:]`	Duże litery	`[A-Z]`
`[:xdigit:]`	Znaki szesnastkowe	`[0-9a-fA-F]`
`\s`	Spacje	`[\r\n \t]`
`\S`	Negacja \s	`[^\r\n \t]`
`\w`	Słowa	`[0-9A-Za-z_]`
`\W`	Negacja słów	`[^0-9A-Za-z_]`
`\d`	Liczby	`[0-9]`
`\D`	Nie liczby	`[^0-9]`

Kolejna tabela przedstawia znaki, które służą do "zakotwiczania" wyrażeń obsługiwane przez wyrażenia regularne.

Znak	Znaczenie
`^`	Początek ciągu
`$`	Koniec ciągu
`[[:<:]]`	Początek słowa
`[[:>:]]`	Koniec słowa
`\b`	Granica słowa
`\B`	Negacja `\b`
`\A`	Początek ciągu
`\Z`	Koniec ciągu lub znak nowej linii
`^`	Początek linii
`$`	Koniec linii

Granica słowa jest zdefiniowana jako punkt między znakiem odstępu a znakiem identyfikacyjnym (alfanumerycznym lub podkreśleniem):

```
<?php
    preg_match("/[[:<:]]gun[[:>:]]/", "the Burgundy exploded", $captured);
    preg_match("/gun/", "the Burgundy exploded", $captured1);
    print_r($captured);
    print_r($captured1);
?>
```

Początek i koniec ciągu również kwalifikują się jako granice słowa.

Step 6

Kwantyfikatory wyrażeń regularnych są zazwyczaj **zachłanne**. Oznacza to, że w obliczu kwantyfikatora silnik dopasowuje tyle, ile może, jednocześnie spełniając resztę wzorca. Na przykład:

```
<?php
    preg_match("/(<.*>)/", "do <b>not</b> press the button", $captured);
    print_r($captured);
?>
```

Wyrażenie regularne pasuje od pierwszego znaku `<` do ostatniego znaku `>`. W efekcie `.*` dopasowuje wszystko po pierwszym znaku `<`, aż w końcu pojawia się znak `>`, który należy dopasować. Czasami jednak potrzebujemy minimalnego (nie zachłannego) dopasowania — to znaczy kwantyfikatorów, które pasują jak najmniej razy, aby spełnić resztę wzorca. `Perl` dostarcza równoległy zestaw kwantyfikatorów, które są minimalnie dopasowane. Są łatwe do zapamiętania, ponieważ są takie same jak kwantyfikatory zachłanne, ale z dołączonym znakiem zapytania (`?`). Tabela poniżej pokazuje odpowiednie kwantyfikatory zachłanne i nie zachłanne obsługiwane przez wyrażenia regularne w stylu `Perla`.

Zachłanny kwantyfikator	Nie zachłanny kwantyfikator
`?`	`??`
`*`	`*?`
`+`	`+?`
`{m}`	`{m}?`
`{m,}`	`{m,}?`
`{m,n}`	`{m,n}?`

Oto jak dopasować tag za pomocą kwantyfikatora, który nie jest zachłanny:

```
<?php
    preg_match("/(<.*?>)/", "do <b>not</b> press the button", $captured);
    print_r($captured);
?>
```

Innym, szybszym sposobem jest użycie klasy znaków, aby dopasować każdy znak `<` do następnego znaku `>` niż:

```
<?php
    preg_match("/(<[^>]*>)/", "do <b>not</b> press the button", $captured);
    print_r($captured);
?>
```

Jeśli umieścimy część wzorca w nawiasach, tekst pasujący do tego wzorca podrzędnego zostanie przechwycony i będzie można uzyskać do niego dostęp później. Czasami jednak chcemy utworzyć podwzorzec bez przechwytywania pasującego tekstu. W wyrażeniach regularnych zgodnych z `Perlem` można to zrobić za pomocą konstrukcji (`?: subpattern`):

```
<?php
    preg_match("/(?:ello)(.*)/", "jello biafra", $captured);
    print_r($captured);
?>
```

Możemy odwoływać się do tekstu przechwyconego wcześniej we wzorcu z odwołaniem wstecznym: `\1` odnosi się do zawartości pierwszego podwzorca, `\2` odnosi się do drugiego i tak dalej. Jeśli zagnieżdżamy podwzory, pierwszy zaczyna się od pierwszego otwierającego nawiasu, drugi zaczyna się od drugiego otwierającego nawiasu i tak dalej. Na przykład poniższy skrypt identyfikuje podwójne słowa:

```
<?php
    preg_match('/\b(\w+)\s+\1\b/', "It's the the PHP 8.0", $matches);
    print_r($matches);
?>
```

Funkcja `preg_match()` przechwytuje maksymalnie 99 podwzorców; reszta będzie ignorowana.

Wyrażenia regularne w stylu `Perla` umożliwiają umieszczanie opcji jednoliterowych (**flag**) po wzorcu wyrażenia regularnego w celu modyfikowania interpretacji lub zachowania dopasowania. Na przykład, aby dopasować bez rozróżniania wielkości liter, należy użyć flagi `i`:

```
<?php
    preg_match("/cat/i", "Stop, Catherine!",$matches);
    print_r($matches);
?>
```

Tabela poniżej przedstawia, które modyfikatory `Perla` są obsługiwane w zwykłym kompatybilnym z Perlem wyrażeniem regularnym.

Modyfikator	Znaczenie
`/regexp/i`	Ignorowanie wielkości liter
`/regexp/s`	Ustaw kropkę (.) na dowolny znak w tym znak nowej linii
`/regexp/x`	Usuń białe znaki oraz komentarze ze wzorca
`/regexp/m`	Ustaw znak ^ po, znak dolara $ przed i ustaw znaki nowej linii wewnętrznie
`/regexp/e`	Jeśli zastępczy ciąg jest kodem PHP, funkcja `eval()` uzyska rzeczywisty zastępujący ciąg

Wyrażenia regularne `PHP` obsługują również inne modyfikatory, które nie są obsługiwane przez `Perl`, które zostały wymienione w tabeli poniżej:

Modyfikator	Znaczenie
`/regexp/U`	Odwraca zachłanność podwzorca; `*` i `+` teraz pasują jak najmniej, zamiast jak najwięcej
`/regexp/u`	Powoduje, że łańcuchy wzorców są traktowane jako UTF-8
`/regexp/X`	Powoduje, że ukośnik odwrotny, po którym następuje znak bez specjalnego znaczenia, powoduje wyświetlenie błędu
`/regexp/A`	Powoduje, że początek ciągu jest zakotwiczony tak, jakby pierwszym znakiem wzorca był `^`
`/regexp/D`	Powoduje, że znak `$` pasuje tylko na końcu wiersza
`/regexp/S`	Powoduje, że parser wyrażeń dokładniej bada strukturę wzorca, więc następnym razem może działać nieco szybciej (na przykład w pętli)

Możliwe jest użycie więcej niż jednej opcji w jednym wzorcu, jak pokazano w poniższym przykładzie:

```
<?php
$message = <<< END
To: you@youcorp
From: me@mecorp
Subject: pay up
Pay me or else!
END;
    preg_match("/^subject: (.*)/im", $message, $match);
    print_r($match);
?>
```

Step 7

Oprócz określania opcji dla całego wzorca po ograniczniku, który zamyka wzorzec, można określić opcje we wzorcu, aby zastosować je tylko do części wzorca. Składnia tego jest następująca:

```
(?flags:subpattern)
```

Na przykład tylko słowo „PHP” nie uwzględnia wielkości liter:

```
<?php
    preg_match('/I like (?i:PHP)/', 'I like pHp', $match);
    print_r($match) ;
?>
```

W ten sposób można wewnętrznie zastosować opcje `i`, `m`, `s`, `U`, `x` i `X`. Możemy użyć wielu opcji jednocześnie:

```
<?php
    preg_match('/eat (?ix:foo d)/', 'eat FoOD', $match);
    print_r($match) ;
?>
```

Poprzedzenie opcji łącznikiem (`-`), powoduje wyłączenie tej opcji:

```
<?php
    preg_match('/I like (?-i:PHP)/', 'I like pHp', $match);
    print_r($match);
?>
```

Włączenie lub wyłączenie flagi można też uzyskać poprzez dodanie nawiasów okrągłych przed końcem otaczającego podwzorca lub wzoru:

```
<?php
    preg_match('/I like (?i)PHP/', 'I like pHp', $match);
    preg_match('/I (like (?i)PHP) a lot/', 'I like pHp a lot', $match1);
    print_r($match);
    print_r($match1);
?>
```

Wbudowane flagi nie umożliwiają przechwytywania. Aby to zrobić, będziemy potrzebować dodatkowego zestawu nawiasów przechwytujących.

We wzorach czasami przydaje się możliwość powiedzenia „dopasuj tutaj, jeśli to jest następne”. Jest to szczególnie powszechne, gdy dzielimy ciąg znaków. Wyrażenie regularne opisywany jest przez separator, który nie jest zwracany. Możemy użyć **lookahead**, aby upewnić się (bez dopasowywania go, zapobiegając w ten sposób jego zwróceniu), że po separatorze jest więcej danych. Podobnie **lookbehind** sprawdza poprzedzający tekst. **Lookahead** i **lookbehind** występują w dwóch formach: pozytywnej i negatywnej. **Pozytywny lookahead** lub **lookbehind** mówi „następny/poprzedni tekst musi wyglądać tak”. **Negatywny lookahead** lub **lookbehind** oznacza, że „następny/poprzedni tekst nie może wyglądać tak”. Tabela poniżej przedstawia cztery konstrukcje, których można użyć we wzorcach zgodnych z `Perlem`. Żadna z tych konstrukcji nie przechwytuje tekstu.

Konstrukcja	Znaczenie
`(?=podwzorzec)`	Pozytywny lookhead

`(?!podwzorzec)`	Negatywny lookhead
`(?<=podwzorzec)`	Pozytywny lookbehind
`(?<!podwzorzec)`	Negatywny lokkbehind

Prostym zastosowaniem **pozytywnego lookahead** jest podzielenie pliku pocztowego Unix mbox na pojedyncze wiadomości. Słowo `From` rozpoczynające linię samo w sobie wskazuje początek nowej wiadomości, więc możemy podzielić skrzynkę pocztową na wiadomości, określając separator jako punkt, w którym następny tekst to `From` na początku linii:

```
<?php
$mailbox = <<< END
From mmiotk@pjwstk.edu.pl
Subject: Test
Body: This is a test

From mmiotk@pjwstk.edu.pl
Subject: Test2
Body: This is a test2
END;
    $messages = preg_split('/(?=^From )/m', $mailbox);
    print_r($messages);
?>
```

Prostym zastosowaniem **negatywnego lookbehind** jest wyodrębnienie ciągów w cudzysłowie, które zawierają ograniczniki w cudzysłowie. Na przykład, oto jak wyodrębnić łańcuch w pojedynczym cudzysłowie (zauważ, że wyrażenie regularne ma użyty modyfikator `x`):

```
<?php
$input = <<< END
name = 'Matthew O\'Reilly';
END;
$pattern = <<< END
' # opening quote
( # begin capturing
.*? # the string
(?<! \\\\ ) # skip escaped quotes
) # end capturing
' # closing quote
END;
preg_match( "($pattern)x", $input, $match);
print_r($match);
?>
```

Jedyną trudną częścią jest to, że aby uzyskać wzorzec, za pomocą **lookbehind**, jest sprawdzenie, czy ostatni znak był ukośnikiem odwrotnym. Wówczas musimy uciec od tego ukośnika, aby silnik wyrażeń regularnych nie widział `\`), co oznaczałoby dosłowny nawias zamykający. Innymi słowy, musimy wykonać odwrotny ukośnik: `\\`). Ale reguły `PHP` dotyczące cudzysłowów mówią, że `\\` tworzy dosłowny pojedynczy ukośnik odwrotny, więc w końcu potrzebujemy czterech ukośników odwrotnych, aby przejść przez wyrażenie regularne! Dlatego wyrażenia regularne mają opinię trudnych do odczytania. `Perl` ogranicza **lookbehind** do wyrażeń o stałej szerokości. Oznacza to, że wyrażenia nie mogą zawierać kwantyfikatorów, a jeśli używamy alternatywy, wszystkie wybory muszą mieć tę samą długość. Aparat wyrażeń regularnych zgodny z Perlem również zabrania kwantyfikatorów w **lookbehind**, ale dopuszcza alternatywy o różnych długościach.

Step 8

Istnieje pięć klas funkcji, które współpracują z wyrażeniami regularnymi zgodnymi z `Perlem`: dopasowywanie, zastępowanie, dzielenie, filtrowanie i funkcja narzędziowa do cytowania tekstu.

Funkcja `preg_match()` wykonuje dopasowanie wzorca w stylu `Perla` na łańcuchu. Jest to odpowiednik operatora `m//` w `Perlu`. Funkcja `preg_match_all()` przyjmuje te same argumenty i zwraca taką samą wartość zwrotną jak funkcja `preg_match()`, z wyjątkiem tego, że przyjmuje wzorzec w stylu `Perla` zamiast wzorca standardowego. Funkcja `preg_match_all()` wielokrotnie dopasowuje miejsce, w którym zakończyło się ostatnie dopasowanie, aż nie będzie można wykonać więcej dopasowań:

```
$found = preg_match_all(pattern, string, matches [, order ]);
```

Wartość `order` może być ustawiona na `PREG_PATTERN_ORDER` lub `PREG_SET_ORDER`, która określa układ dopasowań.

```
<?php
$string = <<< END
13 dogs
12 rabbits
8 cows
1 goat
END;
    preg_match_all('/(\d+) (\S+)/', $string, $m1, PREG_PATTERN_ORDER);
    preg_match_all('/(\d+) (\S+)/', $string, $m2, PREG_SET_ORDER);
    print_r($m1);
    print_r($m2);
?>
```

Z `PREG_PATTERN_ORDER` (domyślnie), każdy element tablicy odpowiada określonemu podwzorcowi przechwytywania. Zatem `$m1[0]` jest tablicą wszystkich podciągów pasujących do wzorca, `$m1[1]` jest tablicą wszystkich podciągów pasujących do pierwszego podwzorca (liczb), a `$m1[2]` jest tablicą wszystkich podciągów pasujących do drugiego podwzorca (słowa). Tablica `$m1` ma o jeden element więcej niż wszystkie podwzore. Dzięki `PREG_SET_ORDER` każdy element tablicy odpowiada następnej próbie dopasowania całego wzorca. Zatem `$m2[0]` to tablica pierwszego zestawu dopasowań („13 dogs", „13", „dog"), `$m2[1]` to tablica drugiego zestawu dopasowań („12 rabbits", „12", „rabbits") i tak dalej. Tablica `$m2` ma tyle elementów, ile udało się dopasować całego wzorca.

Funkcja `preg_replace()` zachowuje się jak operacja wyszukiwania i zastępowania w edytorze tekstu. Znajduje wszystkie wystąpienia wzorca w ciągu i zmienia je na coś innego:

```
$new = preg_replace(pattern, replacement, subject [, limit ]);
```

Najczęstszym zastosowaniem są wszystkie ciągi argumentów z wyjątkiem wartości zmiennej `limit`. `Limit` to maksymalna liczba wystąpień wzorca do zastąpienia (wartość domyślna i zachowanie po przekroczeniu limitu `-1` to wszystkie wystąpienia):

```
<?php
    $better = preg_replace('/<.*?>/', '!', 'do <b>not</b> press the button');
    echo $better;
?>
```

Przekazanie tablicy ciągów znaków dokona podstawienia na wszystkich zgadzających się elementach. Nowe ciągi są zwracane przez `preg_replace()`:

```
<?php
    $names = array('Fred Flintstone','Barney Rubble','Wilma Flintstone', 'Betty Rubble');
    $tidy = preg_replace('/(\w)\w* (\w+)/', '\1 \2', $names);
    print_r($tidy);
?>
```

Aby wykonać wielokrotne podstawienia na tym samym ciągu lub tablicy ciągów za pomocą jednego wywołania `preg_replace()`, należy przekazać tablice wzorców i ich zamienników:

```
<?php
    $contractions = array("/don't/i", "/won't/i", "/can't/i");
    $expansions = array('do not', 'will not', 'can not');
    $string = "Please don't yell - I can't jump while you won't speak";
    $longer = preg_replace($contractions, $expansions, $string);
    echo $longer;
?>
```

Jeśli podamy mniej zamienników niż wzorców, tekst pasujący do dodatkowych wzorców zostanie usunięty. To wygodny sposób na usunięcie wielu rzeczy naraz:

```
<?php
    $htmlGunk = array('/<.*?>/', '/&.*?;/');
    $html = '&eacute; : <b>very</b> cute';
    $stripped = preg_replace($htmlGunk, array(), $html);
    print_r($stripped);
?>
```

Podstawienie może korzystać z odwołań wstecznych. Jednak w przeciwieństwie do odwołań wstecznych we wzorcach, preferowana składnia odwołań wstecznych w podstawieniach to `$1`, `$2`, `$3` i tak dalej.

```
<?php
    echo preg_replace('/(\w)\w+\s+(\w+)/', '$2, $1.', 'Fred Flintstone');
?>
```

Odmianą `preg_replace()` jest `preg_replace_callback()`. To wywołuje funkcję (podaną jako drugi argument), aby uzyskać podstawienie. Do funkcji jest przekazywana tablica dopasowań (elementem zerowym jest cały tekst pasujący do wzorca, pierwszy to zawartość pierwszego przechwyconego podwzorca i tak dalej).

```
<?php
    function titlecase($s)
    {
        return ucfirst(strtolower($s[0]));
    }
    $string = 'goodbye cruel world';
    $new = preg_replace_callback('/\w+/', 'titlecase', $string);
    echo $new;
?>
```

Step 9

Podczas gdy używasz `preg_match_all()` służy do wyodrębniania fragmentów łańcucha, użycie `preg_split()`, wyodrębnia te fragmenty, gdy wiemy, co oddziela te fragmenty od siebie:

```
$chunks = preg_split(pattern, string [, limit [, flags ]])
```

Wzorzec dopasowuje separator między dwoma kawałkami. Domyślnie separatory nie są zwracane. Opcjonalny `limit` określa maksymalną liczbę porcji do zwrócenia (`-1` to wartość domyślna, co oznacza wszystkie porcje). Argument `flags` jest bitową kombinacją operatora lub, na którą mamy flagi `PREG_SPLIT_NO_EMPTY` (puste fragmenty nie są zwracane) i `PREG_SPLIT_DELIM_CAPTURE` (zwracane są części łańcucha przechwyconego we wzorcu). Na przykład, aby wyodrębnić tylko operandy z prostego wyrażenia liczbowego, należy użyć:

```
<?php
    $ops = preg_split('{[+*/-]}', '3+5*9/2');
    print_r($ops);
?>
```

Aby wyodrębnić operandy i operatory, można użyć następującego kodu:

```
<?php
    $ops = preg_split('{([+*/-])}', '3+5*9/2', -1, PREG_SPLIT_DELIM_CAPTURE);
    print_r($ops);
?>
```

Pusty wzorzec pasuje do każdej granicy między znakami w ciągu oraz na początku i na końcu ciągu. Pozwala to podzielić ciąg na tablicę znaków:

```
<?php
    $string = "ala ma kota";
    $array = preg_split('//', $string);
    print_r($array);
?>
```

Funkcja `preg_grep()` zwraca te elementy tablicy, które pasują do podanego wzorca:

```
$matching = preg_grep(pattern, array);
```

Na przykład, aby uzyskać tylko nazwy plików, które kończą się na `.txt`, należy użyć następującego kodu:

```
<?php
    $filenames = array("documents.txt", "1.pdf", "picture.jpg", "2.txt");
    $textfiles = preg_grep('/\.txt$/', $filenames);
    print_r($textfiles);
?>
```

Funkcja `preg_quote()` tworzy wyrażenie regularne pasujące tylko do podanego ciągu:

```
$re = preg_quote(string [, delimiter ]);
```

Każdy znak w łańcuchu, który ma specjalne znaczenie w wyrażeniu regularnym (np. `*` lub `$`) jest poprzedzony odwrotnym ukośnikiem:

```
<?php
    echo preg_quote('$5.00 (five bucks)');
?>
```

Opcjonalny drugi argument to dodatkowy znak do cytowania. Zwykle podajemy tutaj ogranicznik wyrażenia regularnego:

```
<?php
    $toFind = '/usr/local/etc/rsync.conf';
    $re = preg_quote($toFind, '/');
    if (preg_match("/{$re}/", $toFind,$filename)) {
      print_r($filename);
    }
?>
```

Step 10

Chociaż bardzo podobne, implementacja wyrażeń regularnych w stylu `Perla` w `PHP` ma kilka drobnych różnic w stosunku do rzeczywistych wyrażeń regularnych `Perla` :

- Znak `NULL` (ASCII 0) nie jest dozwolony jako znak dosłowny w ramach jako wzorzec. Możemy jednak odwoływać się do niego w inny sposób (`\000` , `\x00` , itp.).
- Opcje `\E` , `\G` , `\L` , `\l` , `\Q` , `\u` i `\U` nie są obsługiwane.
- Konstrukcja (? `{pewny kod perla}`) nie jest obsługiwana.
- Obsługiwane są modyfikatory `/D` , `/G` , `/U` , `/u` , `/A` i `/X` .
- Pionowy tabulator `\v` liczy się jako znak odstępu.