WYRAŻENIA REGULARNE

Wyrażenia Regularne

Zrozumienie Potrzeb

Wyrażenia rgularne (lub w skrócie regex), są narzędziami, i jak wszystkie narzędzia, wyrażenia regularne są stworzone po to aby rozwiązywać określone potrzeby. Najlepszym sposobem na zrozumienie wyrażeń regularnych i to czemu służą, jest zrozumienie jakie problemy rozwiązują.

Rozważmy następujący scenariusze:

- Szukasz pliku zawierającego tekst sad (bez względu na wielkość liter), ale nie chcesz znajdować sad umieszczonego w jakimś słowie (np. osad, sadownik itp)
- Dynamicznie generujemy stronę WWW (korzystając z apliakcji serwera) i potrzebujemy wyświetlić tekst przywrócony z bazy danych. Tekst może zawierać URL'e, a ty chcesz aby te URL'e można było klikać w wygenerowanej stronie (tak aby zamiast generować tekst, wygenerować poprawny tekst HTML <A HREF><A>)
- Tworzysz stronę internetową z formularzem. Formularz prosi użytkownika o wprowadzenie danych , włącznie z adresem e-mail. Musisz zweryfikować czy ten określony adres jest poprawnie sformatowany (to znaczy , czy jest poprawny składniowo)
- Edytujesz kod źródłowy i musisz zastąpić wszystkie wystąpienia size na iSize, ale tylko size a nie size jako częśćinnego słowa.
- Wyświetlasz listę wszystkich plików z komputerowego systemu plików i chcesz wykonać filtrowanie aby znaleźć tylko pliki zawierające tekst Aplikacja.
- Importujesz dane do aplikacji. Dana jest ograniczona tabulatorem i apliakcja obsługuje format plików CSV (jeden wiersz na linię, wartości oddzielone przecinkami, każda otoczona cudzysłowami)
- Musisz wyszukać plik z określonym tekstem, ale tylko w określonym położeniu (być może na początku linii lub końcu zdania)

Każdy z tych scenariuszy przedstawia unikalne wyzwanie programistyczne. Wszystkie z nic mogą rozwiązane w dowolnym języku, który obsługuje przetwarzanie warunkowe i maniulację łańcuchami. Ale jak złożone zadania posłużą do ich rozwiązania? Będziesz musiał przejść pętlą przez słowa lub znaki raz w danym czasie, wykonywać sortowania, testowanie instrukcji, śledzić flagi, sprawdzać białe znaki i znaki specjalne i wiele więcej. I musisz to wszystko zobić ręcznie.

Albo możesz użyć wyrażeń regularnych. Każde z powyższych wyzwań można rozwiązać przy użyciu dobrze przyygotowanych instrukcji - bardzo zwięzłych ciągów zawierających tekst i specjalne instrukcje – które mogą wyglądać np tak:

\b[Cc] [Aa] [Rr] \b.

Jak są używane wyrażenia regularne

Spójrzmy ponownie na problem scenariuszy i zauważmy ,że wszystkie sprowadzają się do dwóch typów: Albo informacja jest lokalizowana (szukanie) albo informacja jest lokalizowana i edytowana (zastępowanie) .Faktycznie, wyrażenia regularne są używane zawsze dla : wyszukiwania i zastępowania. Kaze wyrażenie regularne albo dopasowuje tekst (wykonywanie wyszukiwania) albo dopasowuje i zastępuje tekst (wykonywanie zastępowania)

Wyszukiwanie RegEx
Wyrażenia regularne używane są podczas wyszukiwania, gdy tekst do wyszukania jest bardzo dynamiczny, jak wyszukiwanie sad w powyższym wcześniejszym scenariuszu. Na początku należy

zlokalizować sad, SAD lub Sad a nawet SaD;to najłatwiejsza część (wiele narzędzi wyszukiwania jest zdolnych do wykonywania zapytań, które nie uwzględniają wielkości liter). Trudniejsze jest zapewnienie ,że osad, podsadzenie czy wsad nie zostaną dopasowane. Niektóre bardziej wyrafinowane edytory mają opcję Dopasuj Tylko Całe Słowo, ale wiele nie i nie możemy wprowadzić tej zmiany w dokumencie jaki edytujemy. Użycie wyrażeń regularnych dla wyszukiwania, nie tylko tekstu sad, rozwiązuje ten problem. Warto zauważyć ,że testowanie równości (na przykład, czy ten adres e-mail określony przez użytkownika pasuje do tego wyrażenia reuglarnego?) jest działaniem wyszukiwania. Przeszukiwany jest cały ciąg dostarczony przez użytkownika dla dopasowania (w przeciwieństwie do wyszukiwania podciągów, które jest zwykle wykonywane)

Zastępowanie RegEx
Wyszukiwania przez wyrażenia regularne są niezwykle mocne, bardzo przydatne i nie trudno się ich nauczyć. Jednak prawdziwą siłę wyrażeń regularnych poznajemy podczas operacji zastępowania, tak jak we wcześniejszym scenariuszu wymiany URL'i na klikalne URL'e. Na początku wyagane jest znalezienie URL'i wewnątrz tekstu (być może wyszukiwanie ciągu znaków zaczynających się na http:// lub https://, a kończących się kropką, przecinkiem lub białym znakiem). Wymaga to wtedy aby zastąpić znaleziony URL dwoma wystąpieniami dopasowanego ciągu znaków z osadzonym HTML tak aby

http://www.e-wiki.pl/

zpostało zastąpione przez

<A HREF="http://www.e-wiki.pl">http://www.e-wiki.pl</A>

Co to jest wyrażenie regularne?

Teraz kiedy wiemu jak używane są wyrażenia regularnem, kolej na definicję. Krótko mówiąc *wyrażenia regularne* są łańcuchami, które są używane w celu dopasowania i przetwrzania tekstu. Wyrażenia regularne są tworzone przy wykorzystaniu języka wyrażeń regularnych, wyspecjalizowanego języka zaprojektowanego do robienia tego co omawialiśmy i wiele więcej. Podobnie jak każdy język, wyrażenia regularne mają specjalną skłądnię i instrukcje, których trzeba się nauczyć. Język wyrażeń regularnych nie jest pełnym językiem programowania. Zwykle nie ma nawet rzeczywistego programu lub narzędzia, które można zainstalować i używać. Najczęściej wyrażenia regularne są mini językiem wbudowanym w inne języki lub produkty. Dobra wiadomość jest taka ,że prawie każdy przyzwoity język lub narzędzie obecnie obsługuje wyrażenia regularne.Zła wiadomość jest taka ,że język wyrażeń regularnych sam nie będzie wyglądał jak język lub narzędzie którym się można posłużyć
UWAGA: Wyrażenia regularne pochodzą z badań z lat 50'tych w dziedzinie matematyki. Lata później, zasady i idee pochodzące z tych wczesnych prac znalazły się w świecie Unix w języku Perl i narzędzi, takich jak grep. Przez wiele lat, wyrażenia regularne były wyłączną domeną społeczności Unix, ale to się zmieniło, a teraz wyrażenia regularne są obsługiwane w różnych formach niemal na wszystkich platformach komputerowych. Poniżej umieszczone są poprawne wyrażenia regularne:

- Ben
- .
- www\.e-wiki\.pl
- [a-zA-Z0-9_.]*
- <[Hh]1.*</[Hh]1>

- \r\n\r\n
- \d{3,3} - \d{3,3} - \d{4,4}

Ważne jest ,że ta składnia jest najprostszą częścią dla opanowania wyrażeń regularnych. Prawdziwym wyzwaniem jest jednak, nauczyć się jak zastosować tą składnię, jak wnikliwie rozwiązywać problemy w regex. Tego nie można się nauczyć z książek, ale jak wszędzie mistrzostwo przychodzi z ptraktyką

Używanie wyrażeń regularnych

Jak wspominaliśmy wcześniej, nie ma żadnego programu dla wyrażeń regularnych;nie ma aplikacji jaką można uruchomić ani oprogramowania które można kupić. Zamiast tego, język wyrażeń regularnych jest implementowany w wielu produktach softwareowych, językach ,narzędziach i środowiskach projektowych. Jak są stosowane wyrażenia regularne i jak funkcjonują wyrażenia regularne , różni się między aplikacjami. Niektóre aplikacje używają opcji menu i okien dialogowych dla dostępu dla wyrażeń regularnych, inne języki programowania dostarczają funkcji i klas obiektów, które wykorzystują funkcjonalność regex. Co więcej, nie wszystkie wyrażenia regularne implementowane są jednakowo. Występują czasami subtelne (czasami mniej subtelene) różnice między składnią a funkcją.
Zanim przejdziemy dalej, odnotujmy parę ważnych spraw:

- Kiedy użyjemy wyrażeń regularnych, odkryjemy ,że jest zawsze wiele rozwiązań dowolnego problemu Niektóre są prostsze, niektóre szybsze, niektóre bardziej przenośne. Rzadko mamy dobre lub złe rozwiązanie kiedy piszemy wyrażenia regularne (tak długo ,rzecz jasna ,jak rozwiązanie działa)
- Istnieją różnice między implementacjami regex.
- Jak przy każdym języku, kluczem do nauczenia się wyrażeń regularnych jest praktyka, praktyka , praktyka

Podsumowanie

Wyrażenia regularne są jednym z najpotężniejszych narzędzi do przetwarzania tekstu. Język wyrażeń regularnych służy do budowania wyrażeń regularnych, a wyrażenia regularne są używane do wykonywania zarówno operacji wyszukiwania jak i zastępowania.

Dopasowanie tekstu dosłownego

Ben jest wyrażeniem regularnym.Ponieważ jest to zwykły tekst, może nie wyglądać jak wyrażenie regularne, ale jest. Wyrażenia regularne mogą zawierać zwykły tekst (i mogą zawierać tylko zwykły tekst). Oczywiście jest to całkowite marnotrawstwo przetwarzania wyrażeń regularnych ,ale jest to dobre miejsce na start

TEXT

Cześć, mam na imię Ben. Proszę odwiedzić mnie
na mojej stronie http://e-wiki.pl

RegEx

Ben

WYNIK

Cześć, mam na imię Ben. Proszę odwiedzić mnie
na mojej stronie http://e-wiki.p

ANALIZA

Wyrażenie regularne tu użyte jest tekstem dosłownym i pasuje do Ben w tekście oryginalnym.
Spójrzmy na inny przykład używający tego samego tekstu ale innego wyrażenia regularnego

TEKST

Cześć, mam na imię Ben. Proszę odwiedzić mnie
na mojej stronie http://e-wiki.pl

RegEx

na

WYNIK

Cześć, mam na imię Ben. Proszę odwiedzić mnie
na mojej stronie http://e-wiki.pl

ANALIZA

na jest również tekstem statycznym, ale zauważmy dwa wystąpienia dopasowania na.

Ile dopasowań?
Domyślnym zachowaniem wiekszości silników wyrażenia regularnego jest zwracanie dopiero pierwszego wzorca. W poprzednim przykładzie, zwykle pierwsze na byłoby dopasowane, ale nie drugie. Dlaczego więc były dwa dopasowanie. Większość implementacji regex tworzy mechanizm, dla uzyskania listy wszystkich dopasowań (zwykle zwracanych w tablicy lub innej specjalnej postaci). W JavaScript, na przykład, używając opcjonanej flagi g (global) ,zwaracana jest tablica zawierająca wszystkie dopasowania

Obsługa czułości wielkości liter
Wyrażenia regularne są czułe na wielkość liter, więc Ben nie pasuje z ben. Jednak, większość implementacji regex również obsługuje dopasowania które nie są czułe na wielkość liter. Uzytkownicy JavaScript, na przykład,mogą określać opcjonalną flagę i dla wymuszenia wyszukiwania, które nie jest czułe na wielkość liter

Dopasowanie dowolnych znaków

Wyrażenia regularne do tej pory dopasowywały tylko tekst statyczny .Terazz spojrzymy na dopasowywanie nieznanych znaków. W wyrażeniach regularnych, znaki specjalne (lub zbiór znaków) sa wykorzystywane do określenia co jest do wyszukania. Znak . (kropka) oznacza jeden dowolny znak. Dlatego wyszukanie k.t dopasuje kat i kot.
Poniżej mamy przykłady:

TEKST

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

RegEx

sala.

WYNIK

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

ANALIZA

Tu mamy regex sala., którego użyjemy do znalezienia wszystkich nazw plików zaczynających się na sala i po których występuje jakiś znak. Trzy z dziewięciu plików pasują do wzorca.

Często używamy terminu *wzorzec* dla opisania rzeczywistego wyrażenia regularnego
Zwróć uwagę ,że wyrażenia regularne dopasowują wzorce do zawartości łańcucha. Dopasowania nie zawsze będa całymi łańcuchami, ale znakami które pasują do wzorca – nawet jeśli są tylko częścią łańcucha. W tu użytym przykładzie, wyrażenie regularne nie pasuje do nazwy pliku; raczej pasuje częśćnazwy pliku. To rozróżnienie jest ważne do zapamiętania kiedy przekazujemy wyniki wyrażenia regularnego do jakiegoś innego kodu lub aplikacji do przetwarzania

. dopasowuje dowolny znak, znak alfabetu, cyfrę a nawet samą **. :**

TEKST

sala.xls
sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls

sa1.xls

RegEx

sala.

WYNIK

sala.xls
sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

ANALIZA

Ten przykład zawiera jeden doadtkowy plik , sala.xls. Ten plik został dopasowany przez wzorzec sala. ponieważ **.** dopasowuje dowolny znak. Wiele **.** może być użytych albo razem (jedna po drugiej – użycie . . dopasuje dowolne dwa znaki obok siebie) albo w różnych miejscsach wzorca. Spójrzmy na inny przykład z tym samym tekstem. Tym razem musimy znaleźć wszystkie plik dla na lub sa :

TEKST

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

RegEx

. a .

Wynik

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls

sa1.xls

ANALIZA

regex . a . Znajduje na1, na2 i sa1 ale również znajduje czery inne dopasowania. Dlaczego? Ponieważ wzorzec dopasowania dwowolnych trzech znaków tak długo jak w środku jest a.

Oto kolejna próba

TEKST

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

RegEx

.a..

WYNIK

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

ANALIZA

.a.. nie działa lepiej niż .a.; dopasowuje tylko dodatkowy znak (bez względu na to jaki jest) .Więc jak można wyszukać . kiedy . Jest specjalnym znakiem, który oznacza dowolny znak?

Dopasowanie znaków specjalnych

. ma specjalne znaczenie w regex. Jeśli potrzebujesz . W swoim wzorcu, musisz znaleźć sposób aby powiedzieć regex, że w rzeczywistości chcesz kropkę a nie znak specjalny . Aby to zrobić, ujmij kropkę, w poprzedzający ją \ (backslash). \ jest metaznakiem (sposób na określenie ,ze jest to znak specjalnym w przeciwieństwie do samego znaku). Dlatego też . Oznacza dopasowanie dla dowolnego znaku, a \ oznacza dopasowanie dla samej kropki. Wypróbujemy poprzedni przykład, tym razem ujmując kropkę w backslash

TEKST

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

RegEx

. a . \.xls

WYNIK

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls

ANALIZA

. a .\ .xls załatwiło sprawę. Pierwsza kropka dopasowała n (w pierwszych dwóch dopasowaniach) lub s w trzecim Druga kropka dopasowała 1 (w pierwszym i trzecim dopasowaniu) lub 2 (w drugim). \. dopasował wtedy kropkę odzielającą nazwę pliku od rozszerzenia, i samo xls.
W wyrażeniach regularnych, \ zawsze używa się do ozanczania poczatku bloku jednego lub więcej znaków, które mają specjalne zanczenie

Podsumowanie

Wyrażenia regularne ,zawne również wzorcami, są łańcuchami zbudowanymi ze znaków .Znaki te mogą być dosłowne (sam teskt) lub metaznakami (znaki specjalne o specjalnym znaczeniu). Nauczyliśmy się jak dopasować pojedynczy znak używając zarówno tekstu dosłownego jak i metaznaków. Kropka dopasowuj dowolny znak. \ jest używane do ustawiania przed znakami dla rozpoczęcia sekwencji znaków specjalnych

Dopasowanie jednego z kilku znaków

Jak dowiedzieliśmy się z poprzedniej części, **.** Dopasowuje dowolny jeden znak.W ostatnim przykładzie, **.** była używana dla dopasowanie na i sa , **.** dopasowała zrówno n i s . Ale co jeśli był plik nazwany ca1.xls a nadal chcesz dopasować tylko na i sa? **.** Dopasuje również c a więc nazwa pliku również będzie dopasowana. Aby znaleźć n i s nie chcesz dopasowywać każego znaku,

chcesz dopasowywać tylko te dwa znaki W wyrażeniach regularnych, zbiór znaków jest definiowany przy użyciu metaznaków [i] . [i] definiują zbiór znaków,wszystko między nimi jest częścią zbioru i jeden z elementów tego zbioru musi być dopasowanu (ale nie wszystkie).
Tu mamy przykład z poprzedniej części :

TEKST

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls
ca1.xls

RegEx

[ns]a. \ .xls

WYNIK

sala1.xls
order3.xls
sala2.xls
sala3.xls
apacz1.xls
europa2.xls
na1.xls
na2.xls
sa1.xls
ca1.xls

ANALIZA

Wyrażenie regularne używane tutaj zaczyna się od [ns]; to dopasowuje albo n albo s (ale nie c lub inny znak). [i] nie dopasowują żadnego znaku - definiują zbiór .Dosłowne a pasuje do a, **.** dopasowuje dowolny znak, \. dopasowuje **.** , a dosłowne xls dopasowuje xls. Kiedy używasz tego wzoru, tylko trzy różne żądane nazwy plików zostaną dopasowane.
Faktycznie, [ns]a. \.xls nie jest całkiem tak dokładne. Jeśli istaniłby plik o nazwie usa1.xls, byłby również dopasowany. Jak widać, testowanie wyrażeń regularnych może być trudne. Sprawdzanie czy pasuje do wzorca jakiego chcesz, jest łatwe. Zestawy znaków są często wykorzystywane do wyszukiwania (lub określania jej części) bez względu na wielkość liter. Na przykład:

TEKST

Fraza "regular expression" jest często
skracane do RegEx lub regex

RegEx

[Rr]eg[Ee]x

WYNIK

Fraza "regular expression" jest często
skracane do RegEx lub regex

ANALIZA

Wzorzec tu zastosowany zawiera dwa zbiory znaków : [Rr] pasuje do R i r, a [Ee] pasuje do E i e. W ten sposób, RegEx i regex , oba są dopasowane. REGEX jednak już nie.

Użycie zakresu zbioru znaków

Spójrz na listę pliku z przykładu. Ostatni używany wzorzec [ns]a.\.xls, ma inny problem Co jeśli nazwa pliku to sam.xls?Również będzie dopasowany ponieważ **.** dopasowuje wszystkei znaki , nie tylko cyfry. Zbiór znaków może rozwiązać ten problem

TEKST

```
sales1.xls

orders3.xls

sales2.xls

sales3.xls

apac1.xls

europe2.xls

sam.xls

na1.xls

na2.xls

sa1.xls

ca1.xls
```

RegEx

[ns]a[0123456789]\.xls

WYNIK

```
sales1.xls

orders3.xls

sales2.xls

sales3.xls

apac1.xls

europe2.xls

sam.xls

na1.xls

na2.xls

sa1.xls

ca1.xls
```

ANALIZA

W tym przykładzie, wzorzec został zmodyfikowany tak aby pierwszy znak miałbyć albo n albo s, drugi znak musi być a , a trzeci może być dowolną cyfrą (określoną jako [0123456789]) Zwróć uwagę,że plik sam.xls nie pasuje, ponieważ m nie pasuje do listy dozwolonych znaków (10 cyfr). Kiedy pracujemy z wyrażeniami regularnymi, odkrywamy najczęściej określany zakres znaków (od 0 do 9, od A do Z itd). Dla uproszczenia pracy z zakresem znaków, regex dostarcza specjhalny metazbaj : - (myślnik) używany do określania zakresu:

TEKST

```
sales1.xls

orders3.xls

sales2.xls

sales3.xls

apac1.xls

europe2.xls

sam.xls

na1.xls

na2.xls

sa1.xls

ca1.xls
```

RegEx

[ns]a[0-9]\.xls

WYNIK

```
sales1.xls

orders3.xls

sales2.xls

sales3.xls

apac1.xls

europe2.xls

sam.xls

na1.xls

na2.xls

sa1.xls

ca1.xls
```

ANALIZA

Wzorzec [0-9] jest funkcjonalnym odpowiednikiem [0123456789], a wyniki są identyczne z tymi w poprzednim przykładzie. Zakres nie jst ograniczony do cyfr. Poniżej mamy poprawne zakresy:

- A-Z dopasowuje wszystkie duże znaki od A do Z
- a-z dopasowuje wszystkie małe litery od a do z
- A-F dopasowuje tylko duże znaki od A do F
- A-z dopasowuje wszystkie znaki między ASCII A a ASCII z (prawdopodobnie nigdy nie użyejsz tego wzorca, ponieważ zawiera również znaki takie jak [i ^, które leżą między Z a a w tabeli ASCII)

Dowolne dwa znaki ASCII mogą być określone przez początek i koniec zakresu. W praktyce, jednak, zakresy są zazwyczaj tworzeone przez pewne lub wszystkie cyfry i pewane lub wszystkie znaki alfabetu.
Pamiętaj, że kiedy używasz zakresów, nie stosuj zakresu końcowego, który jest mniejszy niż zakres początkowy (np [3-1]). To nie zadziała.

Wiele zakresów może być połączonych w jeden zbiór.Na przykład, poniższe wzorce doapsowania , dopasowują znaki alfanumeryczne duże i małe, ale nie wszystko co nie jest cyfrą alni znakiem alfabetu:

```
[A-Za-z0-9]
```

Ten wzorzec jest skrótem dla

```
[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01234567890]
```

Jak widzisz, zakresy czynią składnię regex dużo wyraźniejszą

Poniżej mamy przykład znajdowania wartości RGB (kolory przedstawiane w notacji szesnastkowej, czlyli ilość czerwonego, zielonego i niebieskiego do tworzenia kolorów). Na stronach WWW, wartości RGB są określane jako #000000 (czarny), #FFFFFF (biały), #FF0000 (czerwony) itd. Wartości RGB mogą być określane jako duże lub małe litery a #FF00ff jest również poprawne. Oto przykład

TEKST

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

RegEx

```
#[0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]
```

WYNIK

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

ANALIZA

Wzorzec użyty tu zawiera # jako tekst literalny a potem zbiór znaków [0-9A-Fa-f] powtórzony sześć razy. Dopasowujemy # po któym występuje sześć znaków, każdy musi być cyfra lub A do F (duże lub małe litery)

Dopasowanie "Anything But"

Zbiory znaków są zazwyczaj używane do określania listy znaków ,które muszą być dopasowane. Ale okazjonalnie, chcesz to odwrócić – lista znaków , której nie chcesz doapsować. Innymi słowy *niczego co jest określone na liście.* Zamiast wymieniać każdy znak jaki chcesz, zbiór znaków może być zanegowany metaznakiem ^. Oto przykład:

TEKST

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

RegEx

```
[ns]a[^0-9]\.xls
```

WYNIK

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

ANALIZA

Wzorzec użyty w tym przykładzie jest dokładnym przeciwieństwem użytego poprzednio. [0-9] dopasowuje wszystkie cyfry (i tylko cyfry). [^0-9] nie dopasowuej niczego z zakresu liczb. Tak

więc, [ns]a[^0-9]\.xls dopasowuj sam.xls, ale nie na1.xls,na2.xls lub sa1.xls.

^ neguje wszystkie znaki lub zakres w zbiorze, a nie tylko znak lub zakres jako poprzedza

Podsumowanie

Metaznaki [i] są używane do definiowania zbiorów znaków,z kórych każy musi się zgadzać (OR w przeciwieństwie do AND) .Zbiór znaków może być określony wyraźnie lub określony jako zakres przy użyciu metaznaku - . Zbiór znaków może być negowany znakiem ^; to wymusza niedopasowanie określonej części znaków.

Escaping

Zanim zagłębimy się w świat meta znaków, ważne jest zrozumienie escapingu. Metazanki są znakami, kóre mają specjalne znaczenie z wyrażeniami regularnymi. Kropka (.) jest metaznakiem; jest używana do dopaspowania pojedynczego znaku. Podobnie lewy nawias kwadratowy ([ ) jest metaznakiem; jest używany do oznaczania początku zbioru. Ponieważ metaznaki przybierają specjalnego znaczenia kiedy sa używane w wyrażeniach regularnych, znaki te nie mogą być wużywane w odniesieniu do samych siebie Na przykład, nie można użyć [do dopasowania [lub . do dopasowania .. Spójrzmy na poniższy przykład. Wyrażenie regularne użyte do próby dopasowania tablicy JavaScript zawierającej [i] :

TEKST

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

RegEx

```
myArray[0]
```

WYNIK

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

ANALIZA

W tym przykładzie, blok tekstu jest kodem JavaScript (lub częścią) Wyrażenie regularne jest typem , prawdopodbnie używanym wewnątrz edytora tekstu. Można przypuszczać ,że dopasowało

tekst literalny myArray[0],ale nie. Dlaczego nie? [i] są metaznakami wyrażenia regularnego, które są używane do definiowania zbioru (ale nie znaków [i]) .Jako taka, myArray[0] dopasuje myArray po której wystąpi jedna ze składowych zbioru, a 0 jest jedyną składową. Dlatego, myArray[0] dopasuje tylko myArray0. Jak wcześniej wspomniano, metaznaki mogą być zmieniane przez poprzedzenie ich backslashem. Dlatego \. dopasuje . , a \[dopasuje [. Każdy metaznak może być zmieniony przez poprzedzenie go backslahem; kiedy tak uczynimy, sam znak jest dopasowywany, zamiast specjalnego znaczenia jaki przedstawia. Dla dopasowania [i] , znaki te muszą być pominięte. Oto ten sam przykład, tym razem ze zmianą metaznaków :

TEKST

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

RegEx

```
myArray\[0\]
```

WYNIK

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

ANALIZA

Tym razem wyszukano poprawnie. \[dopasowało [a \] dopasowało]; więc myArray\[0\] dopasowało myArray[0].
Użycie wyrażenia regularnego w tym przykładzie jest niekonieczne – prosty tekst wystarczyłby i byłoby to łatwiejsze. Ale wyobraź sobie ,że chciałbyś dopasować nie tylko myArray[0] ale również myArray[1], myArray[2] itd. Wtedy użycie wyrażenia regualrnego będzie miało sens. Możesz zamineić [i] i określić znaki do dopasowania między nimi. Jeśli chcesz dopasować elementy tablicy od 0 do 9, możesz użyć wyrażenia takiego jak to:

```
myArray\[[0-9]\]
```

Metaznaki , które są częścią pary (takie jak [lub]) muszą być zmienione jest będą używane jako metaznaki, lub parser wyrażenia regularnego wykaże błąd.

\ jest używane do zmiany metaznaków. Oznacza to ,że sam \ jest metaznakiem; jest używany do zmiany innych znaków.. Spójrzmy na poniższy prosty przykład. Tekst jest ścieżką pliku używającą backslashy (używane w DOS i Windows).Teraz wyobraź sobie ,że musisz użyć tej ścieżki w

systemie Linux lub Unix, i tak ,że musisz zmienić wszystkie backslashe na slashe:

TEKST

```
\home\ben\sales\
```

RegEx

```
\\
```

WYNIK

```
\home\ben\sales\
```

ANALIZA

\\ dopasowuje \ , i znaleziono czteryu dopasowania. Gdyby po prostu określono \ jako wyrażenie rgularne prawdopodobnie wystąpiłby błąd (ponieważ parser wyrażeń regualrnych słusznie by założył ,że wyrażenie jest niekompletne; po wszystkim, po \ zawsze występuje inny znak w wyrażeniu regularnym)

Dopasowanie białych znaków

Metaznaki generalnie dzielą się na dwie kategorie : te używane dla dopasowania tesktu (takie jak **.**) i te używane jako część składni wyrażenia regularnego (takie jak [i]). Odkryjemy wiele różnych metaznaków obu typów, poczynając od metznaków białych znaków. Kiedy przygotowujemy wyszukianie wyrażenia regularnego, często musimy dopasować niedrukowalne białe znaki osadzone w tekście. Na przykład, możesz chcieć znaleź wszystkie znaki tabulacji, lub przełamania linii. Ponieważ wpisanie tych znaków bezpośrednio do wyrażenia regularnego bardzo trudne, można użyć specjalnych metaznaków z poniższej tabelki:

Metaznaki	**Opis**
[\b]	backspace
\f	nowa strony
\n	nowy wiersz
\r	powrót karetki
\t	tabulator
\v	tabulator pionowy

Spójrzmy na przykład. Poniższy blok tekstu zawiera szereg rekodów w formacie ograniczenia przecinkami (często nazywan CSV). Przed przetworzeniem rekordów, musimy usunąć puste linie w danych.

TEKST

```
"101","Ben","Forta"

"102","Jim","James"


"103","Roberta","Robertson"

"104","Bob","Bobson"
```

RegEx

```
\r\n\r\n
```

WYNIK

```
"101","Ben","Forta"

"102","Jim","James"


"103","Roberta","Robertson"

"104","Bob","Bobson"
```

ANALIZA

\r\n dopasowują połączenie powrotu karetki i wysuwu linii używane (przez Windows) jako znacznik końca linii. Wyszukiwanie \r\n\r\n dopasowuje dwa znaczniki końca linii a zatempuste linie są między dwoma rekordami.

W systemie Linux (Unix) używany jest znak wysuwu linii. W tych systemach prawdopodobnie chcielibyśmy użyć \n (a nie \r)

Dopasowanie określonych typów znaków

Do tej pory widzieliśmy jak dopasować określone znaki, dowolny znak (korzystając z **.**), jedne zbió znaków (używając [i]) i jak zanegować dopasowanie (używając ^) .Zestaw znaków (dopasowany jeden zbió) jest najczęstszą formą dopasowania, a zpecjalne metaznaki mogą być używane w miejsce powszechnie używanych zbiorów. Te metaznaki dopasowują klasy znaków. Klasa metaznaków nigdy nie jest potrzebna w rzeczywistości (zawsze można wymienić znaki do dopasowania lub użyć zakresów), ale bez wątpienia jest niezwykle przydatna.

Dopasowanie cyfr (i nie cyfr)
Jak już wiesz , [0-9] jest skrótem dla [0123456789] i jest używane dla dopasowania cyfr. Aby dopasować coś innego niż cyfra, zbiór może być zanegowany jako [^0-9]. Poniżej mamy pokazane klasy skrótów dla cyfr i nie cyfr.

Metaznak	**Opis**

\d	Dowolna cyfra (to samo co [0-9])
\D	Dowolna nie cyfra (to samo co [^0-9])

TEKST

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

RegEx

```
myArray\[\d\]
```

WYNIK

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

ANALIZA

\[dopasowuje [, \d dopasowuje dowolną pojedynczą cyfrę, a \] dopasowuj] , więc myArray\[\d\] dopasowuje myArray[0].myArray\[\d\] jest skrótem dla myArray\[0-9\], które jest skrótem dla myArray\[0123456789\]. To wyrażenie regularne również dopasuej myArray[1], myArray[2] itd, (ale nie myArray[10].
Wyrażenia regularne są czułe na wielkość liter. \d dopasowuje cyfry. \D jest dokładnym przeciwieństwem \d.

Dopasowywanie znaków alfanumerycznych (i znaków nie alfanumerycznych)
Innym częstym użyciem zbioru są wszystkie znaki alfanumeryczne, od A do Z (małymi i dużymi literami), cyfry i znak podkreślenia (często używany w nazwach plików i katalogów, nazwach zmiennych apliakcji, nazwach obiektów bazy danych itd). Poniżej mamy skrót klas dla znaków alfanumerycznych i nie alfanumerycznych

Metaznak	**Opis**
\w	Dowolny znak alfanumeryczny z małej lub dużej litery i znak podkreślenia (to samo co [a-zA-Z0-9_])
\W	Dowolny znak nie alfanumerycznylub znak podkreślenia (to samo co [^a-zA-Z0-9_])

TEKST

```
11213

A1C2E3

48075

48237

M1B4F2

90046

H1H2H2
```

RegEx

```
\w\d\w\d\w\d
```

WYNIK

```
11213

A1C2E3

48075

48237

M1B4F2

90046

H1H2H2
```

ANALIZA

Wzorzec tu użyty łączy meatzanki \w i \d dla uzyskania kanadysjkich kodów pocztopwych

Dopasowanie białych znaków (i nie białych znaków)
Końcową klasą na jak spojrzymy to klasa białych znaków. Wcześniej uczyliśmy się o metaznakach dla określania białych znaków. Poniżej mamy skórty klasy dla wszystkich białych znaków

Metaznaki	**Opis**
\s	Dowolny biały znak (to samo co [\f\n\r\t\v])
\S	Dowolny nie biały znak (to samo co [^\f\n\r\t\v])

Określanie wartości szesanstkowych lub ósemkowych
Chociaż nie będziemy się odności od określonych znaków przez ich wartości ósemkowe lub szesnastkowe, warto zobaczyć jak to jest robione.

Wartości szesnastkowe
Wartości szesnastkowe (podstawa 16) mogą być określone przez poprzedzenie ich \x. Dlatego , \x0A (znak ASCII 10, znak wysuwu linii) jest funkcjonalnym ekwiwalentem dla \n.

Wartości ósemkowe
Wartości ósemkowe (podstawa 8) mogą być określone jako dwu lub trzy cyfrowan liczba poprzedzona przez \0 .Dlatego , \011 (znak ASCII 9, znak tabulacji) jest funkcjonalnym ekwiwalentem dla \t.

Użycie klas znaków POSIX

Nasza nauka o metaznakach i skrótach dla różnych zbiorów znaków nie byłaby kompletna bez wiedzy o klasie znaków POSIX. Jest to jeszcze inna forma skrótu która jest wspeirana przez wiele (ale nie wszystkie) implementacje wyrażeń regularnych
JavaScript nie obsługuje klas znaków POSIX w wyrażeniach regularnych

Klasa	Opis
[:alnum:]	Dowolna litera lub cyfra (to samo co [a-zA-Z0-9])
[:alpha:]	Dowolna litersa (to samo co [a-zA-Z])
[:blank:]	Spacja lub tabulator (to samo co [\t])
[:cntrl:]	Znaki kontrolne ASCII (ASCII 0 do 31 i 127)
[:digit:]	Dowolna cyfra (to samo co [0-9])
[:graph:]	To samo co [:print:] ale bez spacji
[:lower:]	Dowolna mała litera (to samo co [a-z])
[:print:]	Dowlony znak drukowalny
[:punct:]	Dowolny znak, który nie jest ani [:alnum:] ani [:cntrl:].
[:space:]	Dowolny biały znak w tym spacja (to samo co [\f\n\r\t\v])
[:upper:]	Dowona duża litera (to samo co [A-Z])
[:xdigit:]	Dowlona liczba szesnastkowa (to samo co [a-fA-F0-9])

Składnia POSIX jest całkiem inna niż metaznaki jakie poznaliśmy dootąd.Aby zademonstrować użycie POSIX, wykorzystamy przykład z lokalizowaniem wartości RGB w bloku kodu HTML

TEKST

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

RegEx

```
#[[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]]
```

WYNIK

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

ANALIZA

Wzorzec używany ostatnio do tego przykładu powtarzał sześć rzy zbiór znaków [0-9A-Fa-f]. Tu każde [0-9A-Fa-f] zastapiliśmy [[:xdigit:]]. Wynik jest taki sam

Zauważ ,że wyrażenie regularne tu użyte zaczyna się od [[i kończy]] (dwa zbiory nawiasów). To jest ważne i wymagane kiedy używamy klas POSIX. Klasy POSIX są otoczone wewnątrz [: i :]; POSIX używa [:xdigit:] a nie :xdigit:. Zewnętrzne [i] definiują zbiór; wewnętrzne [i] są częścią samej klasy POSIX

PODSUMOWANIE

Wprowadziliśmy metaznaki, które dopasowują określone znaki (takie jak tabulatory i wysuwanie wiersza) lub całe zestawy klas znaków (takich jak cyfry lub znaki alfanumeryczne). Te skróty metaznaków i klas POSIX mogą być użyte do upraszczania wzorców wyrażeń regularnych.

Ile dopasowań?

Uczyliśmy się o wszystkich podstawowych wzorców dopasowania wyrażeń regularnych, ale wszystkie przykłady mają jedno poważne ograniczenie. Rozważmy co się dzieje, kiedy piszemy wyrażenie regularne dla dopasowania adresu email. Podstawowy format adresu email wygląda jak coś takiego:

text@text.text

Używając metaznaków omówionych w poprzednich częściach, możemy stworzyć wyrażenie regularne jak poniżej:

\w@\w.\w

\w będzie dopasowywać wszystkie znaki alfanumeryczne (plus znak podkreślenia, który jest poprawny w adresie email); @ nie musi być podmieniane, ale . tak.
To jest dokładnie poprawne wyrażenie regularne, aczkolwiek raczej bezużyteczne. Byłby zgodny z adresem email, który wygląda jak a@b.c (które choć synktatycznie poprawne, oczywiście nie jest poprawnym adresem email) Problem z tym jest taki ,że \w dopasowuje pojedynczy znak i nie może wiedzieć ile ma znaków do przetestowania. Po wszystkim następujące adresy email są poprawen, ale wszystkie mają różną liczbę znaków przez @:

```
b@forta.com

ben@forta.com

bforta@forta.com
```

To co konieczne to sposó na dopasowanie wielu znaków, i jest to wykonalne przy użyciu jednego z kilku specjalnych metaznaków.

Dopasowanie jednego lub więcej znaków
Aby dopasować jeden lub więcej instancji znaku (lub zbioru) po prostu dołączamy znak + . + dopasowuje jeden lub więcej znaków (przynajmniej jeden; zero nie będzie dopasowywane). Podczas gdy a dopasowuje a, a+ dopasowuje jedno lub więcej a. Podobnie, podczas gdy [0-9] dopasowuje dowolne cyfry, [0-9]+ dopasowuje jedną lub więcej cyfr.
Kiedy używamy + ze zbiorami, + powinno być umieszczone za zbiorem. Dlatego [0-9]+ jest poprawne, a [0-9+] już nie. [0-9+] w rzeczywistości jest poprawnym wyrażeniem regualrnym, ale nie będzie dopasowywał jednej lub więcej cyfr. Przeciwnie, określa zbiór od 0 do 9 lub znak +, i będzie pasował do dowolnej pojedynczej cyfry lub znaku plus. Mimo ,że jest to poprawne, raczej nie o to nam chodziło.

TEKST

```
Send personal email to ben@forta.com. For questions

about a book use support@forta.com. Feel free to send

unsolicited email to spam@forta.com (wouldn't it be

nice if it were that simple, huh?).
```

RegEx

```
\w+@\w+\.\w+
```

WYNIK

```
Send personal email to ben@forta.com. For questions

about a book use support@forta.com. Feel free to send

unsolicited email to spam@forta.com (wouldn't it be

nice if it were that simple, huh?).
```

ANALIZA

Wzorzec dopasował wszystkie trzy adresy poprawnie. Wyrażenie regularne najpierw dopasowuj jeden lub więcej znaków alfanumerycznych używając \w+. Następnie dopasowuje @ po któym występuje jeden lub więcej znaków, ponownie używając \w+ .Potem dopasowuje **.** (używając zamiany \ .) i innego \w+ dla dopasowanie końca adresu.
+ jest metaznakiem. Dla dopasowania + musimy użyć escapingu \+. + może być również użyty dla dopasowania jednego lub więcej zbiorów znaku. Aby to zademonstrować, poniższy przykład pokazuje to samo wyrażenie regularne ale na innym tekście

TEKST

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

RegEx

```
\w+@\w+\.\w+
```

WYNIK

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

ANALIZA

Wyrażenie regularne dopasowało pięć adresów, ale dwa z nich są niekompletne.Dlaczego tak jest? \w+@\w+.\w+ sprawia ,że nie ma miejsca na znak **.** Przed @, i pozwala tylko na pojedynczą **.** Oddzielającą dwa łańcuchy po @. Chociaż ben.forta@forta.com jest poprawnym adresem email, wyrażenie regualrne dopasuje tylko forta (zamiast ben.forta) ponieważ \w dopasowuje znaki alfanumeryuczne ale nie **.** w środku łańcucha tekstu.

TEKST

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

RegEx

```
[\w.]+@[\w.]+\.\w+
```

WYNIK

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

ANALIZA

To zdawałoby się załatwić sprawę. [\w.] + dopasowuje jedną lub więcej instancji znaku alfanumerycznego, znaku podkreślenia i **.** , a więc ben.forta jest dopasowane. [\w.]+ jest również używane do łańcucha po @ więc ta głębsza nazwa domeny (lub hosta) będzie dopasowana.
Zauważ ,ze dla końcowego dopasowania, użyliśmy \w+ a nie [\w.]+. Chcesz wiedzieć dlaczego? Spróbuj użyć [\w.] dla końcowego wzorca a zobaczysz co złego będzie z drugim , trzecim i czwartym dopasowaniem.

Dopasowanie zera lub więcej znaków
+ dopasowuje jedno lub więcej znaków. Zero znaków nie będzie dopasowane - musi być przynajmniej jeden. Ale co jeśli chcesz dopasować całkowicie opcjonalne znaki, tak aby było dozwolone zero znaków?
Aby to zrobić, użyjemy metaznaku * . * jest używany dokładnie jak + ;jest umieszczoen po znaku lub zbiorze i dopasowuje zero lub więcej instancji znaku lub zbioru. Dlatego ,wzór B.*Forta, dopasowuje B Forta, B. Forta, Ben Forta i inne kombinacje
Spójrzmy na zmodyfikowany przykład z adresem email

TEKST

```
Hello .ben@forta.com is my email address.
```

RegEx

```
[\w.]+@[\w.]+\.\w+
```

WYNIK

```
Hello .ben@forta.com is my email address.
```

ANALIZA

Przypomnij sobie ,że [\w.]+ dopasowuje jedne lub więcej instancji znaków alfanumerycznych i **.** , i tak .ben został dopasowany. Istnieje oczywiście literówka w poprzednim tekście (dodatkowa kropka w środku tekstu), ale to nie ma znaczenia. Większy problemem jest to ,ze chociaż **.** jest prawidłowym znakiem w adresie email, nie jest poprawnym znakiem jakim zaczyna się adres email. Innymi słowy, to co rzeczywiście dopasowujemy jest alfanumeryczny tekst w opcjonalnymi dodatkowymi znakami:

TEKST

```
Hello .ben@forta.com is my email address.
```

RegEx

```
\w+[\w.]*@[\w.]+\.\w+
```

WYNIK

```
Hello .ben@forta.com is my email address.
```

ANALIZA

Ten wzór wygląda na bardziej złożony (ale naprawdę nie jest). \w+ dopasowuje znak alfanumeryczny ale nie **.** (poprawny znak od którego zaczyna się adres email) Po początkowym poprawnym znaku, możliwe ,że mamy **.** i dodatkowe znaki, chociaż to może faktycznie nie być obecne. [\w.]* dopasowuje zero lub więcej instancji **.** lub znaków alfanumerycznych, co jest czym czego oczekujemy.

Dopasowanie zera lub jednego znaku

Innym bardzo użytecznym metaznakiem jest ?. Podobnie jak + , ? dopasoweuj tekst opcjonalny .Ale w przeciwieństwie do + , ? dopasowuje tylko zeor lub jedną instancję znaku (lub zbioru), ale nie więcej niż jedną. ? Jest bardzo przydatny dla określonego wzorca, pojedynczy opcjonalny znak w bloku tekstu. Rozważmy poniższy przykład:

TEKST

```
The URL is http://www.forta.com/, to connect

securely use https://www.forta.com/ instead.
```

RegEx

```
http://[\w./]+
```

WYNIK

```
The URL is http://www.forta.com/, to connect

securely use https://www.forta.com/ instead.
```

ANALIZA

Wzorzec używa dopasowania URL jako http:// (który jest literalnym tekstem i dlatego dopasowuje tylko siebie) po którym [\w./]+, które dopasowuje jedną lub więcej instancji zbioru, któryu pozwala na znaki alfanumeryczne, **.** , i slash. Ten wzorzec może dopasować tylko pierwszy URL (ten który zaczyna się od http://) ale nie drugi (zaczynający się od https://). A s* (zero lub więcej instancji s) nie bedzie poprawne ponieważ pozwalałoby to też na httpsssss:// (co jest zdecydowanie niepoprawne). Rozwiązanie? Użycie s? Jak widać w przykładzie:

TEKST

```
The URL is http://www.forta.com/, to connect

securely use https://www.forta.com/ instead.
```

RegEx

```
https?://[\w./]+
```

WYNIK

```
The URL is http://www.forta.com/, to connect

securely use https://www.forta.com/ instead.
```

ANALIZA

Wzorzec zaczyna się od https?:// oznacza ,że poprzedni znak (s) powinen być dopasowany jeśli nie jest obecny lub jeśli pojedyncza instancja kego jest obecna. Innymi słowy, https?:// dopasuje zarówno http:// i https:// (ale nic więcej). Nawiasem mówiąc, ? Jest rozwiązaniem problemu z poprzedniej części. Popatrzmy na przykład gdzie użyto \r\n do dopasowania końca linii, i że wspomniałem ,że w Linux i Unix, używamy \n (bez \r) i ,że idealnym rozwiązaniem byłoby dopasowanie opcjonale \r po którym następuje \n.

TEKST

```
"101","Ben","Forta"

"102","Jim","James"


"103","Roberta","Robertson"

"104","Bob","Bobson"
```

RegEx

```
[\r]?\n[\r]?\n
```

WYNIK

```
"101","Ben","Forta"

"102","Jim","James"


"103","Roberta","Robertson"

"104","Bob","Bobson"
```

ANALIZA

[\r]?\n dopasowuje opcjonalnie pojedynczą instancję \r po której występuje \n.
Zwróć uwagę ,że wyrażenie regularne tu używa [\r]? zamiast \r?. [\r] definiuje zbiór zawierający pojedynczy metaznak, zbiór metaznaków, więc [\r]? jest funkcjonalnie identyczny z \r?. [ ] jest zwykle używane do zdefiniowania zbioru znaków, ale niektórzy projektanci lubią używać ich nawet do pojedynczych znaków dla zapobieżeniu dwuznaczności.

Użycie interwałów

+, * i ? są używane do rozwiązania wielu problemów z wyrażeniami regularnymi, ale czasami to nie wystarcza. Rozważmy co następuje:

- + i * dopasowuje nieograniczaną liczbę znaków. Nie określają one maksymalnej liczby znaków dopasowania
- Jedyne minima obslugiwane przez +, * i ? to zero i jeden. Nie dostarczają sposóbu na ustawienie wyraźnej minimalnej liczby dopasowań
- Nie ma również sposobu na określenie dokładnej liczby żądanych dopasowań

Dla rozwiązania tych problemów i zapewnieniw większego stopnia kontroli nad powtarzalnymi dopasowaniami, wyrażenia regualrne pozwalają na korzystanie z interwałów. Interwały są określone między znakami { i } .
{ i } są metaznakami, i powinny być poprzedzane znakiem \ kiedy mają być tekstem literalnym. Warto zauważyć ,że wiele wyrażeń regularnych wydaje się być poprawnie przetwarzać { i } nawet

jeśli nie są poprzedzone tym znakiem. Jednak , lepiej jest nie polegać na tym zachowaniu i poprzedzać je tym znakiem kiedy potrzebne są w sensie dosłownym

Dokładne dopasowanie interwałów

Aby określić dokładną liczbę dopasowa, umieść tą liczbę między { i } .Dlatego {3} oznacza dopasowanie trzech instancji poprzedniego znaku lub zbioru. Jeśli są tylko dwie instancje, wzorzec nic nie dopasuje. Aby to zademonstrować, powróćmy do przykładu RGB. Pamiętaj ,że wartości RGB są określane jak trzy zbiory liczb szesnastkowych (każdy z 2 znakami) Pierwszy wzorzec używa dopasowania wartości RGB jak następuje:

```
#[0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]
```

Używaliśmy też klasy POSIX i zmieniliśmy wzorzec na :

```
#[[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]]
```

Problem z oboma wzorcami polega na tym,że musimy powtarzać każdy zbiór znaków (lub klasy) sześć razy. Oto ten sam przykład, tym razem z wykorzystaniem dopasowania interwału:

TEKST

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

RegEx

```
#[[:xdigit:]]{6}
```

WYNIK

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

ANALIZA

[:xdigit:] dopasowuje znak szesnastkowy , a {6} powtarza klasę POSIX 6 razy.

Zakres dopasowania interwałów

Interwały mogą być również używane do określania zakresu wartości – minimalna i maksymalna liczba instancji, które mają być dopasowane. Zakresy są określane tak {2 , 4} (co oznacza minimum 2, maksimum 4) Przykładem tego jest wyrażenie regularne używanedo walidacji formatu dat:

TEKST

```
4/8/03

10-6-2004

2/2/2

01-01-01
```

RegEx

```
\d{1,2}[-\/]\d{1,2}[-\/]\d{2,4}
```

WYNIK

```
4/8/03

10-6-2004

2/2/2

01-01-01
```

ANALIZA

Daty tu wyświetlone są wartościami jakie użytkownicy wpisują w polach formularza – wartości muszą być walidowane jako poprawne formaty dat. \d{1,2} dopasowuje jedną lub dwie cyfry (to testuje zarówno dzień i miesiąc); \d{2,4} dopasowuje rok; a [-\ /]dopasowuje albo – albo / jako separator daty. Jako takie, daty te zostały dopasowane ale nie 2/2/2 (która jest błędna, ponieważ rok jest zbyt krótki)
Ważne jest ,że poprzedni wzorzec nie sprawdza daty; niepoprawna data taka jak 54/67/9999 przejdzie ten test
Interwały mogą zaczynać się od 0. Interwał {0,3} oznacza dopasowanie ,zero, jedną ,dwie lub trzy instancje.

Dopasowanie interwału "Co najmniej"

Ostatnim użyciem interwałów jest określenie minimalnej liczby instancji do dopasowanie (bez maksimum). Składnia dla tego typu interwału jest podobna do zakresu, ale z pominięciem maksimum. Na przykład {3,} oznacza dopasowanie co najmniej 3 instancji, lub inaczej, dopasowuje 3 lub więcej instancji.
W tym przykładzie, wyrażenie regularne jest użyte do zlokalizowania wszystkich zamówień o wartości 100$ lub więcej:

TEKST

```
1001: $496.80

1002: $1290.69

1003: $26.43

1004: $613.42

1005: $7.61

1006: $414.90

1007: $25.00
```

RegEx

```
\d+: \$\d{3,}\.\d{2}
```

WYNIKI

```
1001: $496.80

1002: $1290.69

1003: $26.43

1004: $613.42

1005: $7.61

1006: $414.90

1007: $25.00
```

ANALIZA

Poprzedni tekst jest raporte zawierający numery porządkowe a następnie wartości zamówienia. Wyrażenie regularne najpierw używa \d+: dla dopasowania liczby porządkowej. Wzorzec \ $\d{3,}\.\d{2} jest używane do dopasowania samej ceny. \$ dopasowuje $, \d{3,} dopasowuje liczbę przynajmnej 3 cyfry (a zatem przynajmniej $100), \. dopasowuje, i w końcu \d{2} dopasowuje 2 cyfry po miejscu dziesiętnym. Wzorzec poprawnie dopasowuje cztery z siedmiu zamówień.

Zapobieganie przed dopasowniem

Dopasowania ? są ograniczone w zakresie (sero lub jeden), a więc są dopasowania interwału kiedy używamy dokładnej ilości lub zakresów. Ale inne formy powtarzania opisane w tej sekcji mogą dopasować niograniczoną liczbę dopasoań – czasami zbyt dużo. Wszystkie przykłady narazie zostały dobrane tak ,aby nie popaść w ciąg dopasowywania,ale rozważmy kolejny przykład. Poniższy tekst jest częścią strony WWW i zawiera tekst z osadzonymi znacznikami <B> HTML'a.

Wyrażenie regularne musi dopasować dowolny tekst wewnątrz znaczników <B>

TEKST

```
This offer is not available to customers

living in <B>AK</B> and <B>HI</B>.
```

RegEx

```
<[Bb]>.*</[Bb]>
```

WYNIK

```
This offer is not available to customers

living in <B>AK</B> and <B>HI</B>.
```

ANALIZA

<[Bb]>dopasowuje znacznik otwierający <B> (z małej lub dużej litery), a </[Bb]> dopasowuje znacznik zamykający (również z małej lub dużej litery). Ale zamiast dwóch dopasowań, znleziono tylko jedno; .* dopasowało wszystko po pierwszym <B> aż do ostatniego </B> tak więc tekst AK</B> and <B>HI został dopasowany. Obejmuje to tekst jaki chcieliśmy dopasować, ale również inne przypadki znaczników. Powód jest taki ,że metaznaki takie jak * i + są zachłanne; to znaczy, wyszukują wszystkie najmożliwsze dopasowania jako przeciwieństwo najmniejszych. To prawie tak ,jeśli rozpocznie się od konca tekstu, pracując wstecz do kolejnego znalezionego dopasowania, w przeciwieństwie do pracy od początku. Jest to celowe i zgodne z projektem, kwantyfikatory są zachłanne. Ale co jeśli nie chcesz zachłannego dopasowania? Rozwiązaniem są leniwe wersje tych kwantyfikatorów (odnosimy się do nich jako leniwych ponieważ dopasowują najmniej znaków zamiast najwięcej) Leniwe kwantyfikatory są definiowane przez dodanie ? do kwantyfikatora jakiego używamy, a każdy zachłanny kwantyfikator ma swój leniwy odpowiednik:

Zachłanny	**Leniwy**
*	***?**
+	**+?**
{n,}	{n,}?

*? jest leniwą wersją *, więc wróćmy do naszego przykładu , tym razem używając *? :

TEKST

```
This offer is not available to customers

living in <B>AK</B> and <B>HI</B>.
```

RegExx

```
<[Bb]>.*?</[Bb]>
```

WYNIK

```
This offer is not available to customers

living in <B>AK</B> and <B>HI</B>.
```

ANALIZA

Przez użycie leniwego *? tylko AK zostało dopasowane w pierwszym dopasowaniu, pozwalając <B>HI</B> na niezależne dopasowanie

PODSUMOWANIE

Prawdziwa siła wzorców wyrażeń regularnych staje się oczywista podczas pracy z powtarzającymi się dopasowaniami. Tu wprowadziliśmy + (dopasowujące jeden lub więcej), * (dopasowuje zero lub więcej), ? (dopasowuje zero lub jeden) jako sposoby wykonywania powtarzających się dopasowań. Dla większej kontroli mogą być używane interwały dla określenia dokładnej liczby powtórzeń jak również minimów i maksimów. Kwantyfikatory są zachłanne i mogą za dużo dopasować; dla zabezpieczenia się przed tym, używamy leniwych kwantyfikatorów.

Używanie granic

Dopasowanie pozycji jest używane do określania, gdzie w ciagu tekstowym powinno wystąpić dopasowanie. Aby zrozumieć dopasowanie pozycji, rozważmy następujący przykład:

TEKST

```
The cat scattered his food all over the room.
```

RegEx

```
cat
```

WYNIK

```
The cat scattered his food all over the room.
```

ANALIZA

Wzorzec cat dopasowuje wszystkie wystąpienia cat, nawet cat wewnątrz słowa scattered. Może to być faktycznie pożądany effekt, ale więcej niż prawdopodobne że nie jest. Jeśli wykonaliśmy wyszukiwanie dla zastąpienia wszystkich wystąpień cat na dog, by skończyć z następującym nonsensem:

```
The dog sdogtered his food all over the room.
```

To prowadzi nas do użycia granic, lub specjalnych metaznaków używanych do określania pozycji (granic) przed lub po wzorcu

Używanie granic wyrazów

Pierwszą granicą (i jedną z najczęściej używanych) jest granica wyrazów określona jkako \b. Jak

sugeruje nazwa, \b jest używane do dopasowania początku i końca słowa. Aby zademonstrować użycie \b. Wykorzystamy poprzedni przykład, tym razem z określoną granicą:

TEKST

```
The cat scattered his food all over the room.
```

RegEx

```
\bcat\b
```

WYNIK

```
The cat scattered his food all over the room.
```

ANALIZA

Słowo cat ma spację przed i po nim, więc dopasowuje \bcat\b (spacja jest jednym ze znaków używanym do oddzielania słów) Słowo cat w scattered, nie jest jednak dopasowane, ponieważ znak przed nim to s a znak po nim to t (żadne nie pasuje do \b).
Więc co dokładnie oznacza \b? Silnik wyrażenia regularnego nie rozumie angielskiego , lub dowolnego języka więc nie wie co to są granice wyrazów. \b po prostu dopasowuje położenie między znakami, które jest zazwyczaj częścią słowa (znaki alfanumeryczne i podkreślenia, tekst , który byłby dopasowany przez \w) .Ważne jest aby uświadomić sobie, że aby dopasować cały wyraz, \b musi być zastosowane zarówno przed jak i po tekście, który będzie dopasowywany. Rozważmy przykład:

TEKST

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

RegEx

```
\bcap
```

WYNIK

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

ANALIZA

Wzorzec \bcap dopasowuje dowolne słowo, które zaczyna się od cap, więc dopasowało cztery

słowa, w tym trzy, które nie są samym słowem cap. Poniżej ten sam przykład ale z końcowym \b :

TEKST

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

RegEx

```
cat\b
```

WYNIK

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

ANALIZA

cap\b dopasowuje słowo, które kończy się cap, więc dopasowało dwa słowo, w tym jedno , które nie jest słowem cap. Jeśli chcemy aby było dopasowane samo słowo cap, poprawny wzorzec to \bcap\b.
\b w rzeczywistości nie dopasowujke znaku, raczej, dopasowuje pozycję. Więc łańcuch dopasowania używający \bcat\b będzie długości trzech znaków (c,a i t), a nie dlugości pięciu znaków.
W poniższym przykładzie metaznaki \B pomagajką zlokalizować myślniki w zewnętrznej przestrzeni wokół nich.

TEKST

```
Please enter the nine-digit id as it
appears on your color - coded pass-key.
```

RegEx

```
\B-\B
```

WYNIK

```
Please enter the nine-digit id as it
appears on your color - coded pass-key.
```

ANALIZA

\B-\B dopasowuje myślnik otoczony znakami przerwy.Myślnik w nine-digit i pass-key nie są dopasowywane, ale color – coded tak.

alizowanie dopasoowań w oparciu o pozycję słowa (początek słowa, konic słowa, całe słowo itd). Granice łańcucha wykonują podobną funkcje ale są używane do dopasowania wzorców na początku i końcu całego łańcucha. Metaznakiem granicy łańcucha jes ^ dla początku lańcucha i $ dla końca łańcucha. ^ jest jednym z metaznaków, które ma wiele zastosowań. Neguje zbiór tylko jeśli jest to zbiór (otoczony wewnątrz [i]) i jest pierwszym znakiem po otwierajacym [. Dla zademosntrowania użycia granicy łańcuchów, spójrzmy na przykład. Poprawny dokument XML zaczyna się od <? xml> i ma dodatkowe atrybuty (możliwy numer wersji, jak w <xml version="1.0" ?>.

TEKST

```
<?xml version="1.0" encoding="UTF-8" ?>

<wsdl:definitions targetNamespace="http://tips.cf"

xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"

xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

RegEx

```
<\?xml.*\?>
```

WYNIK

```
<?xml version="1.0" encoding="UTF-8" ?>

<wsdl:definitions targetNamespace="http://tips.cf"

xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"

xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

ANALIZA

Wzorzec wziął się do pracy. <\?xml dopasopwuje <?xml, . * dopasowuje dowolny inny tekst (zero lub więcej instancji .), a \?> dopasowuje kńcowe ?>. Ale jest to bardzo niedokładne badania. Spójrzmy na przykład, ten sam schemat jest używany dla dopasowania tekstu z zewnętrznego tekst przed otwarciem XML:

TEKST

```
This is bad, real bad!
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

RegEx

```
<\?xml.*\?>
```

WYNIKI

```
This is bad, real bad!
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

ANALIZA

Wzorzec <\?xml.*\?> dopasowuje drugą linię tekstu. I chociaż zancznik otwierający XML, faktycznie będzie drugą linią tekstu, ten przykład jest definitywnie niepoprawny (a przetwarzanie tekstu jako XML może powodować wiele problemów)
Potrzebny jest test, któy zapewni ,że znacznik otwierający XML jest pierwszym rzeczywistym tekstem w łańcuchu, a to doskonała praca dla naszego metaznaku ^ :

TEKST

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

RegEx

```
^\s*<\?xml.*\?>
```

WYNIKI

```
<?xml version="1.0" encoding="UTF-8" ?>

<wsdl:definitions targetNamespace="http://tips.cf"

xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"

xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

ANALIZA

Otwierający metaznak ^ zaczyna łańcuch; ^\s* dlatego dopasowuje początek łańcucha po którym występuje zero lub więcej białych znaków. Całe ^\s*<\xml.*\?> zatem dopasowuje znacznik otwierający XML z atrybutami i poprawnie obsługuje białe znaki.
Wzorzec ^\s*<\xml.*\?> działa ale tylko dlatego ,że pokazany tu przykład XML jest niekompletny $ jest używane w podobny sposób. Wzorzec ten może być użyty do sprawdzenia ,czy coś jest po znaczniku zamykającym <\html> na stronie WWW:

RegEx

```
</[Hh][Tt][Mm][Ll]>\s*$
```

ANALIZA

Zbiory są używane dla każdego ze znaków H,T,M i L a \s*$ dopasowuje białe znaki występujace na końcu łańcucha.
Wzorzec ^.*$ jest synktatycznie poprawnym wyrażeniem regularnym; prawie zawsze znajduje dopasowanie, i jest zupełnie bezużyteczny

Używanie trybu wieloliniowego

^ dopasowuje początek łańcucha a $ dopasowuje koniec łańcucha – zwykle. Jest wyjątek, lub raczej ,sposób zmiany tego zachowania. Wiele implementacji wyrażenia regualrnego wspiera użycie specjalnych metaznaków, które modyfikują to zachowanie innych metaznaków, a jednym z nich jest (?m) , który włącza tryb wieloliniowy. Tryb wieloliniowy wymusza na silniku wyrażenia regularnego traktowanie przełamania linii jako separatora łańcucha, więc ^ dopasowuje początek łańcucha lub początek początek po przełamaniu linii (nowa linia), a $ dopasowuje koniec łańcucha lub koniec po przełamaniu linii. Jeśli użyjemy (?m) musimy umieścić go na samym przodzie wzorca, jak pokazano w poniższym przykładzie, który używa wyrażenia regularnego do zlokalizowania komentarzy JavaScript wewnątrz bloku kodu:

TEKST

```
<SCRIPT>
function doSpellCheck(form, field) {
   // Make sure not empty
   if (field.value == '') {
      return false;
   }
   // Init
   var windowName='spellWindow';
   var spellCheckURL='spell.cfm?formname=comment&fieldname='+field.name;
...
   // Done
   return false;
}
</SCRIPT>
```

RegEx

```
(?m)^\s*//.*$
```

WYNIK

```
<SCRIPT>
function doSpellCheck(form, field) {
   // Make sure not empty
   if (field.value == '') {
      return false;
   }
   // Init
   var windowName='spellWindow';
   var spellCheckURL='spell.cfm?formname=comment&fieldname='+field. name;

   // Done
   return false;
}
</SCRIPT>
```

ANALIZA

^\s*//.*$ dopasowuje początek łańcucha, po którym są białe znaki, po nich // (używane do definiowania komentarzy JavaScript), po nich tekst a potem koniec łańcucha. Ale ten wzorzec będzie dopasowywał tylko pierwszy komentarz (i tylko jeśli będzie jedynym tekstem na stronie). Modyfikator (?m) w (?m)^\s*//.*$ wymusza wzorzec ,który traktuje przełamanie linii jako separator łańcucha więc wszystkie komentarze były dopasowane.

UWAGA: (?m) nie jest wspierane przez wiele implementacji wyrażeń regularnych

PODSUMOWANIE

Wyrażenia regularne moga dopasowywać bloki tekstu pod określonym położeniem wewnątrz łańcucha. \b jest używany do określania granicy słowa (a \B dokładnie odwrotnie) ^ i $ oznaczają granice łańcuchów (początek ciagu i koniec sciągu, odpowiednio), chociaż kiedy używane z modyfikatorem (?m) , ^ i $ również dopasowują łańcuchy ,które zaczynają i kończą przełamaniem linii.

Zrozumienie podwyrażeń

Dopasowanie wielu wystąpień znaku omawiane było przy powtarzalnym dopasowaniu Mówiliśmy tam ,ze \d+ dopasowuje jedną lub więcej cyfr a https?:// dopasowuje http:// lub https://. W obu tych przypadkach powtarzalne metaznaki (? lub * lub {2}, na przykład) stosujemy do poprzednich znaków lub metaznaków. Na przykład, projektanci HTML często umieszczają spacje nierozdzielające między słowami aby tekst nie zwijał się między tymi słowami. Załóżmy ,że musimy zlokalizować wszystkie powtarzalne nierozdzielające spacje HTML (any zastąpić je czymś

innym). Oto przykład:

TEKST

```
Hello, my name is Ben Forta, and I am

the author of books on SQL, ColdFusion, WAP,

Windows  2000, and other subjects.
```

RegEx

```
 {2,}
```

WYNIK

```
Hello, my name is Ben Forta, and I am

the author of books on SQL, ColdFusion, WAP,

Windows  2000, and other subjects.
```

ANALIZA

  odnosi sięd nierozdzielającej spacji HTML. Wzór {2,} powinien dopasować 2 lub więcej instancji . Ale nie robi tego. Dlaczego? Ponieważ {2 , } jest określoną liczbą powtórzeń bezpośrednio poprzedzających, w tym przypadku średnik. ;;;; zostałoby dopasowane ale   już nie

Grupowanie z podwyrażeniami

To przenosi nas do tematu podwyrażeń. Podwyrażenia są częścią większego wyrażenia; te części są pogrupowane razem, tak aby były traktowane jak jedna jednostka. Podwyrażenia są otoczone znakami (i).
(i) są metaznakami. Aby dopasować aktualny znak (i), musisz poprzedzić jes \ (i \ ) ,odpowiednio. Dla zademonstrowania podwyrażeń wrócimy do poprzedniego przykładu:

TEKST

```
Hello, my name is Ben Forta, and I am

the author of books on SQL, ColdFusion, WAP,

Windows  2000, and other subjects.
```

RegEx

```
( ){2,}
```

WYNIK

```
Hello, my name is Ben Forta, and I am

the author of books on SQL, ColdFusion, WAP,

Windows  2000, and other subjects.
```

ANALIZA

( ) jest podwyrażeniem i jest traktowane jako pojedyncza jednostka. Jako takie, {2, }po nim występujące odnosi się do całego podwyrażenia, a ie tylko do średnika. A oto inny przykład – tym razem wyrażenie regularne jest używane do zlokalizowania adresów IP. Adresy IP są formatowane jako cztery zbiory liczb oddzielonych kropkami, takie jak 12.159.46.200 . Ponieważ każda z tych liczb może mieć jedną, dwie lub trzy cyfry, wzorzec dopasowania każdej liczby może być wyrażone jako \d{1,3}.

TEKST

```
Pinging hog.forta.com [12.159.46.200]

with 32 bytes of data:
```

RegEx

```
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
```

WYNIK

```
Pinging hog.forta.com [12.159.46.200]

with 32 bytes of data:
```

ANALIZA

Każda instancja \d{1,3} dopasowuje jedna z liczb adresu IP. Cztery liczby są oddzielone przez ., która jest poprzedzona \. Wzorzez \d{1,3}\. . jest powtarzany trzy razy może być zatem wyrażony przez powtórzenie. Poniżej mamy alternatywną wersję tego samego przykładu:

TEKST

```
Pinging hog.forta.com [12.159.46.200]

with 32 bytes of data:
```

RegEx

```
(\d{1,3}\.){3}\d{1,3}
```

WYNIK

```
Pinging hog.forta.com [12.159.46.200]

with 32 bytes of data:
```

ANALIZA

Ten wzorzec działa podobnie jak poprzedni, ale inna jest składnia. Wyrażenie \d{1,3}\. Otoczone jest przez (i) tworząc podwyrażenie. (\d{1,3}\.){3} powtarza podwyrażenie 3 razy (dla pierwszych trzech liczb w adresie IP), a potem \d{1,3} dopasowuje liczbę końcową.
(\d{1,3}\.){4} nie jest poprawną alternatywą wzorca.
Niektórzy użytkownicy lubią zamykać część wyrażeń jako podwyrażenia do poprawy czytelności; poprzedni wzorzec był wyrażony jako `(\d{1,3}\.){3}(\d{1,3})` .Ta praktyka jest poprawna, a użycie jej nie wpływa na aktualne zachowanie wyrażenia
Użycie podwyrażeń dl grupowania jest tak ważne, że warto spojrzeć na więcej przykładów

TEKST

```
ID: 042

SEX: M

DOB: 1967-08-17

Status: Active
```

RegEx

```
19|20\d{2}
```

WYNIKI

```
ID: 042

SEX: M

DOB: 1967-08-17

Status: Active
```

ANALIZA

W tym przykładzie wzorzec miał cztery cyfry dla roku,ale dla większej dokładności, pierwsze dwie cyfry są wyraźnie określone na 19 i 20. | jest opertorem OR, więc 19|20 dopasowuje albo 19 albo 20, a wzorzec 19|20\d2 powinien dopasować cztery cyfry zaczynające się od 19 lub 20 (po nich występują dwie cyfry). Oczywiście to nie działa. Dlaczego? Operator | patrzy na to co jest po lewej stronie i po prawej i odczytuje wzór 19|20\d{2} jako 19 lub 20\d{2} (myśli ,że \d{2} jest częścią wyrażenia,że zaczyna się od 20) .Innymi słowy, dopasowuje liczbę 19 lub cztery cyfry roku zaczynające się od 20. Rozwiązaniem jest zgupowanie 19|20 jako podwyrażenia:

TEKST

```
ID: 042

SEX: M

DOB: 1967-08-17

Status: Active
```

RegEx

```
(19|20)\d{2}
```

WYNIK

```
ID: 042

SEX: M

DOB: 1967-08-17

Status: Active
```

ANALIZA

Z opcją wszystkiego wewnątrz podwyrażenia, | wie ,że jest to jedna opcja wewnątrz grupy. (19|20)\d{2} zatem poprawnie dopasowuje 1967 i również dopasowuje cztery cyfry zaczynające się od 19 lub 20.

Zagnieżdżanie podwyrażeń

Podwyrażenia mogą być zagnieżdżane. Faktycznie, podwyrażenia mogą być zagnieżdżane wewnątrz podwyrażeń zagnieżdżonych wewnątrz podwyrażeń. Zdolność zagnieżdżania podywrażeń powala na uczynienie wyrażeń niezwykle potężnymi, ale można również stworzyć wzorce wyglądające na skomplikowane, trudne do odczytania i dekodowania i nieco onieśmielające. Prawda jest jednak taka,że zagnieżdżone podwyrażenia są rzadko tak skomplikowane. Dla zademnostrowania użycia zagnieżdzonych podwyrażeń, spójrzmy na nasz przykład z adresem IP. To jest wzorzc użyty poprzednio .

RegEx

```
(\d{1,3}\.){3}\d{1,3}
```

Więc co złego jest w tym wzorcu? Synktatycznie nic. Adres IP w rzeczywistości składa sie z czterech liczb; każda składa się z jednej do trzech cyfr i oddzielnych kropkami. Wzorzec jest poprawny, i będzie dopasowywał każdy porpawny adres IP. Ale to nie wszystko co może dopasować;nieporawny adres IP również będzie dopasowany. Adres IP jest tworzony z 4 bajtów, i adres IP przedstawiony jako 12.159.46.200 jest reprezentowany tymi 4 bajtami. Cztery liczby w

adresie IP mają zakres wartości pojedynczego bajta od 0 do 255. Oznacza to ,że żadna z liczb w adresie IP nie może być większy niż 255. Jednak używany wzór będzie również dopasowywał 345 i 700 i 999, wszystkie niepoprawne liczby w adresie IP.
Jest to ważne o czym teraz mówimy. Łatwo jest pisać wyrażenia regularne dopasowujące to co chsesz i czego oczekujesz. Trudniej jest pisać wyrażenia regularne, które antycypują wszystkie możliwe scenariusze tak aby nie dopasowywały tego czego nie chcesz.

Byłoby miło móc określić zakres poprawnych wartości, ale wyrażenia regularne dopasowują znaki i nie mają relanej wiedzy jakie to są znaki, Matematyczne wyliczenia nie są rozwiązaniem. Czy jest jakaś opcja? Być może. Budując wyrażenie regularne musisz wyraźnie zdefiniować co chcesz zdefiniować a czego nie. Poniżej mamy zasady definiowania poprawnych kombinacji w każdej liczbie adresu IP:

- Jedna lub dwie cyfry liczby
- Trzycyfrowa liczba zaczynająca się od 1
- Trzycyfrowa liczba zaczynająca się od 2 jeśli drugą cyfrą jest od 0 do 4
- Trzycyfrowa liczba zaczynająca się od 25 jeśli trzecia cyfra jest od 0 do 5

Kiedy określimy to sekwencyjnie, staje się jasne ,że rzeczywiście jest wzór, który może działać. Oto przykład:

TEKST

```
Pinging hog.forta.com [12.159.46.200]

with 32 bytes of data:
```

RegEx

```
(((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.){3}((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))
```

WYNIK

```
Pinging hog.forta.com [12.159.46.200]

with 32 bytes of data:
```

ANALIZA

Wzorzec oczywiście działa. To co robi ten wzorzec robi to szereg zagnieżdżeń podwyrażeń. Zaczynamy `(((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.)` ,zbioru czterech

zagnieżdżonych podwyrażeń. (\d{1,2}) dopasowuje jedną lub dwie cyfry liczby lub liczby od 0 do 99. (1\d{2}) dopasowuje dowolną trzycyfrową liczbę zaczynając od 1 (po 1 występują dowlne dwie cyfry), lub liczby od 100 do 199. (2[0-4]\d) dopasowuje liczby 200 do 249. (25[0-5] \) dopasowuje liczby 250 do 255. Każde z tych podwyrażeń jest otoczone innym podwyrażeniem z | między nimi (tak więc jedno z tych czterech podwyrażeń musi być dopasowane , a nie wszystkie). Po zakresie l-iczb przychodzi \. dopasowujące **.** , a potem cała seria (wszystkie opcje liczby plus \.) jest otoczona przez jeszcze jedno podwyrażenie i powtarzane trzy razy przy użyciu {3}. W końcu, zakres liczb jest powtarzany (tym razem bez kończącego \.) dla dopasowania końcowego numeru adresu IP. Przez ograniczenie każdej z czterech liczb do wartości między 0 a 255, wzorzec ten może

dopsowywać poprawne adresy IP i odrzucać niepoprawne.

PODSUMOWANIE

Podwyrażenia są używane do grupowania części wyrażenia razem, i są definiowane przy użyciu (i). Powszechnym użyciem dla podwyrażenia jest kontrola dokładnie tego co musi być powtarzane przez powtórzenie metaznaków i właściwie zdefiniowane warunku OR. Jeśli jest potrzeba , podwyrażenia mogą być zagnieżdżane.

Zrozumienie referencji wstecznych

Najlepszym sposobem zrozumenia potrzeb referencji wstecznej jest spojrzenie na przykład. Projektanci HTML używają znaczników nagłówkowych (od <H1> do <H6>, z odpowiednimi znacnzikami zamykającymi) dla zdefiniowana formatu tekstu nagłówkowego wewnątrz strony iternetowej. Załóży ,że musimy zlokalizować wszystkie teksty nagłówkowe, bez względu na poziom nagłówka. Oto przykład:

TEKST

```
<BODY>

<H1>Welcome to my Homepage</H1>

Content is divided into two sections:<BR>

<H2>ColdFusion</H2>

Information about Macromedia ColdFusion.

<H2>Wireless</H2>

Information about Bluetooth, 802.11, and more.

</BODY>
```

RegEx

```
<[hH]1>.*</[hH]1>
```

WYNIK

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
</BODY>
```

ANALIZA

Wzorzec <[hH]1>.*</[hH]1> dopasowuje pierwsz nagłówek (od <H1> do </H1>) i również <h1> (ponieważ HTML nie jest czuły na wielkość liter). Ale jaki wzorzec może być użyty do dopasowanie dowolnego nagłówka (który mógłby być użyty do sześciu poprawnych poziomów nagłówków). Jedną z pocji może być użycie prostego zakresu zamiast 1, jak tu:

TEKST

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
</BODY>
```

RegEx

```
<[hH][1-6]>.*?</[hH][1-6]>
```

WYNIK

```
<BODY>

<H1>Welcome to my Homepage</H1>

Content is divided into two sections:<BR>

<H2>ColdFusion</H2>

Information about Macromedia ColdFusion.

<H2>Wireless</H2>

Information about Bluetooth, 802.11, and more.

</BODY>
```

ANALIZA

Wydaje się ,że to działa; <[hH][1-6]> dopasowuje dowolny znacznik nagłówka startowego (<H1> i <H2> w tym przykładzie) i </[hH[1-6]> dopasowuje odpowiedni znacznik zamykający (</H1> i </H2>)

Zwróć uwagę ,że użyliśmy .*? (leniwego) a nie .* (chciwego). Jak wyjaśnialiśmy wcześniej, kwantyfikatory takie jak * są chciwe, a więc wzorzec <[hH][1-6]>.*,/[hH][1-6]> może dopasować od znacznika otwierającеh <H1> dw drugiej linii aż do znacznika zamykającego </H2> w szóstej linii. Użycie leniwego kwantyfikatora rozwiązuje ten problem. Powiedziałem może a nie dopasuje, ponieważ ten określony przykład będzie prawdopodobnie działała nawet z chciwym kwatyfikatorem. Metaznak . Zazwyczaj nie dopasowuje lini przełamania,a w tym przykładzie ,każdy nagłówek jest w swojej własnej linii. Nie ma jednak przeszkód dla stosowania leniwych kwantyfikatorów tu, więc lepiej dmuchać na zimne. Sukces? Niedokładnie. Spójrzmy na poniższy przykład (użyjemy tego samego wzorca):

TEKST

```
<BODY>

<H1>Welcome to my Homepage</H1>

Content is divided into two sections:<BR>

<H2>ColdFusion</H2>

Information about Macromedia ColdFusion.

<H2>Wireless</H2>

Information about Bluetooth, 802.11, and more.

<H2>This is not valid HTML</H3>

</BODY>
```

RegEx

```
<[hH][1-6]>.*?</[hH][1-6]>
```

WYNIK

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```

ANALIZA

Startowy znacznik nagłówka <H2> i kończący </H3> jest niepoprawny, a jeszcze wzorzec użyty tu je dopasowuje. Problem jest taki ,że druga część dopasowania (część dopasowująca znacznik końcowy) nie ma wiedzy o części pierwszej dopasowania (część dopasowująca znacznik statrtowy). I jest to miejsce gdzie referencje wsteczne stają się bardzo przydatne.

Dopasowanie z referencją wsteczną

Wrócimy do problemu nagłówka. Teraza spojrzymy na prostszy przykład, który nie może być rozwiązany bez użycia referencji wstecznej. Załóżmy, że miałeś blok tekstu i chciałeś zlokalizować wszystkie powtórzane słowa. Oczywiście, kiedy wyszukujemy drugie wystąpienie słowa, słowo musi być znane. Referencja wsteczna pozwalają wzorcom wyrażenia regularnego odnosić się do poprzedniego wzorca (w tym przypadku, poprzednio dopasowane słowo). Najlepszym sposobem zrozumienia tego jest zobaczyć zastosowanie. Tu jest tekst zawierają trzy zbiory powtarzalnych słów

TEKST

```
This is a block of of text,
several words here are are
repeated, and and they
should not be.
```

RegEx

```
[ ]+(\w+)[ ]+\1
```

WYNIK

```
This is a block of of text,

several words here are are

repeated, and and they

should not be.
```

ANALIZA

Wzór najwyraźniej działa, ale jak działa.[ ]+ dopasowuje jedną lub więcej spacji,\w+ dopasowuje jeden lub więcej znaków alfanumerycznych a [ ]+ wtedy dopasowuje spacje końcowe. Ale zauważ ,że \w+ jest otoczone nawiasami, tworząc z niego podwyrażenie. To wyrażenie nie jest używane dla powtarzalnych dopasowań ,nie ma tu powtarzalnych dopasowań. Raczej to podwyrażenie jest użyte poprostu do grupowania wyrażenia,oznaczenie go i identyfikacji do późniejszego użycia. Końcowa część tego wzorca to \1; jest to odniesienie wsteczne do podwyrażenia, i tak kiedy (\w+) dopasowuje słowo of , więc nie \1, a kiedy (\w+) dopasowuje słowo and, więc nie \1.
termin odniesienie wsteczne odnosi się do faktu ,że te jednostki odnoszą się wstecz do porzedniego wyrażenia.
Co dokładnie oznacza \1? Dopasowuje pierwsze podwyrażenia użyte wewzorcu. \2 będzie dopasowywało drugie pod wyrażenie, \3 trzecie itd. [ ]+(\w+)[ ]+ \1 zatem dopasowuje dowolne słowo a potem to samo słowo ponownie jak widać w porzednim przykładzie.
Można myśleć o odniesieniach wstecznych podobnie jak o zmiennych
Teraz widziałeś jak są używane odniesienia wsteczne, wróćmy więc do przykładu nagłówka. Używając odniesienia wstecznego możliwe jest stworzenie wzorca ,który dopasuje dowolny znacznik pocztkowy nagłówka i dopasuje znacznik zamykający (ignorując pary mieszane)>Oto przykład:

TEKST

```
<BODY>

<H1>Welcome to my Homepage</H1>

Content is divided into two sections:<BR>

<H2>ColdFusion</H2>

Information about Macromedia ColdFusion.

<H2>Wireless</H2>

Information about Bluetooth, 802.11, and more.

<H2>This is not valid HTML</H3>

</BODY>
```

Niestety składnia odniesienia wstecznego różni się znacznie od jednej implementacji regex do drugiej. JavaScript używa \ dla oznaczenia odniesienia wstecznego (z wyjątkeim operacji zastąpienia gdzie jest używane $), podobnie jak Macromedia ColdFusion i vi. Perl używa $ (więc jest $1 zamiast \1) Wyrażenia regularne .NET wspierają powrót obiektu zawierającego właściwą nazwę Groups, która zawiera dopasowania, więc match.Groups[1] odnosi się do pierwszego dopasowania w C# a match.Groups(1) odnosi się do tego samego dopasowania w VisualBasic .NET. PHP zwraca tą informację w tablicy nazwanej $matches, więc $matches[1] odnosi się do pierwszego dopasowania (chociaż to zachowanie oże być zmienione w oparciu o użycie flag). Java i Python zwracają dopasowany obiekt zawierający nazwę tablicy group.

RegEx

```
<[hH]([1-6])>.*?</[hH]\1>
```

WYNIK

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```

ANALIZA

Ponownie znaleźliśmy trzy dopasowania: jedną parę <H1> i dwie pary <H2>. Podobnie jak przedtem, <[hH]([1-6])> dopasowuje dowolny nagłówkowy znacznik początkowy. Ale w odróżnieniu od poprzedniego, [1-6] jest otoczone nawiasami (i) więc tworzy z niego podwyrażenie. W ten sposób, wzorzec końcowego znacznika nagłówka może odnosić się do tego podwyrażenia jak \1 w </[hH]\1>. ([1-6]) jest podwyrażeniem , które dopasowuje cyfry od 1 do 6, a \1 dlatego dopasowuje tylko te same cyfry. W ten sposób ,<H2> This is not valid HTML</H3> nie będzie dopasowane.
Odniesienie wsteczne będzie działało tylko jeśli będzie odnosić się do podwyrażenia.
Dopasowania są zwykle określane zaczynając od 1. W wielu implementacjach, dopasowanie 0 może być użyte do odniesienia do całego wyrażenia . Jak widać, podwyrażenia są określone przez ich względne pozycje : \1 dla , \5 dla piątej itd. Chociaż powszechnie wspierana, składnia ta ma jedno poważne ograniczenie: przeniesieni lub edytowanie podwyrażenia (zatem zmiana porzędk upodwyrażenia) może złamać wzorzec, a dodanie lub usunięcie podwyrażenia może być nawet problematyczne. Aby wypełnić tę lukę ,niektóe nowsze implementacje wyrażeń regularnych wspierają przechwytywanie nazw, funkcję, gdzie każdemu podwyrażeniu może być nadana unikalna nazwa które może być następnie używana w odniesieniu do tego podwyrażenia (zamiast

pozycji względnej).

Wykonywanie operacji zastąpienia

Każde wyrażnie w tym tekście było używane do wyszukiwania – lokalizowania tekstu wewnątrz dużego bloku tekstu. Rzeczywiście, jest możliwe ,że większość wzorców regex, które będziesz pisał będzie używanych do wyszukiwania tekstu. Ale to nie wszystko co regularne wyrażenia mogą robić; wyrażenia regularne mogą być również używane do wykonywania operacji zastępowania. Zastąpienie prostego tekstu nie wymaga wyrażenia regularnego. Na przykład, zastąpienie wszystkich instancji CA na California i MI na Michigan zdecydowanie nie jest zadaniem dla wyrażenia regularnego. Operacja zastąpienia regex staje się istotna , kiedy używamy odniesienia wstecznego.

TEKST

```
Hello, ben@forta.com is my email address.
```

RegEx

```
\w+[\w\.]*@[\w\.]+\.\w+
```

WYNIKI

```
Hello, ben@forta.com is my email address.
```

ANALIZA

Ten wzorzec identyfikuje adresy email wewnątrz bloku tekstu. Ale co jeśli chcesz uczynić adres email linkowanym? W HTML używasz <A HREF="mailto:user@adres.com">user@adres.com</A> dla stworzenia klikalnego adresu email. Czy wyrażenie regularne może skonwertować adres do formatu takiego klikalnego adres? W rzeczywistości, tak ,i bardzo łatwo (tak długo jak używamy odniesienia wstecznego):

TEKST

```
Hello, ben@forta.com is my email address.
```

RegEx

```
(\w+[\w\.]*@[\w\.]+\.\w+)
```

ZASTĄPIENIE

```
<A HREF="mailto:$1">$1</A>
```

WYNIK

```
Hello, <A HREF="mailto:ben@forta.com">ben@forta.com</A>

is my email address.
```

ANALIZA

W operacji zastąpienia, są używane dwa wyrażenia regularne: jedno dla określenis wzorca wyszukiwania i drugie dla określenia tekstu jaki będzie zastępował dopasowany tekst. Odniesienia wsteczne mogą obejmować wzorce, więc podwyrażenia dopasowane w piwerszym wzorcu mogą być użyte w drugim wzorcu. `(\w+[\w\.]*@[\w\.]+\.\w+)` Jest tym, samym wzorcem użytym poprzednio (dla lokalizacji adresu email), ale tym razem jest określone jako podwyrażenie. W ten sposób dopasowany tekst może być użyty we wzorcu zastąpienia. <A HREF="mailto:$1>$1</A> używa twego dopasowanego podwyrażenia dwukrotnie – raz w atrybucie HREF (dla zdefiniowania mailto:) a drugi raz dla tekstu klikalnego. Tak więc ben@forta.com staje się < A HREF="mailto:ben@forta.com">ben@forta.com</A>, co jest dokładnie tym co chcieliśmy.
Jak mówiliśmy poprzednio, musimy zmodyfikować desygnator odniesienia wstecznego w zależności od użytej implementacji. Użytkownicy JavaScript muszą użyć $ zmiast \. Użytkownicy ColdFusion powinni używać \ dla operacji znajdowania i zastępowania. Jak widać w tym przykładzie, podwyrażenia mogą być wykorzystywane wielokrotnie po prostu przez odniesienie do odniesienia wstecznego. Spójrzmy na inny przykład. Informacja o użytkowniku jest przechowywana w bazie danych, a numer telefonu jest przechowywany w postaci 313-555-1234. Jednak, musimy przekształcić numer telefonu tak (313) 555-1234. Oto przykład:

TEKST

```
313-555-1234

248-555-9999

810-555-9000
```

RegEx

```
(\d{3})(-)(\d{3})(-)(\d{4})
```

ZASTĄPIENIE

```
($1) $3-$5
```

WYNIK

```
(313) 555-1234

(248) 555-9999

(810) 555-9000
```

ANALIZA

Ponownie, dwa wzorce wyrażenia regularnego są tu używane. Pierwszy wygląda na bardziej skomplikowany niż jest. `(\d{3})(-)(\d{3})(-)(\d{4})` Dopasowuje numer telefonu, ale jest podzielony na pięć podwyrażeń .(\d{3}) dopasowuje pierwsze trzy cyfry podwyrażenia, (-) dopasowuje – jako drugie podwyrażenie itd. Wynik końcowy jest taki ,że numer telefonu jest podzielony na pięć części (każda część jest własnym podwyrażeniem): obszar kodu, myślinik, pierwsze trzy cyfry numru, inny myślinik a potem końcowe cztery cyfr. Te pięć części może być używanych indywidulanie, a ($1) $3-$5 po prostu reformuje numer używając tylko trzech z podwyrażeń i ignoruje pozostałe dwa, zamieniając 313-555-1234 na (313) 555-1234.

Przyapdki konwersji

Niektóre wyrażenia regularne wspierają użycioe operacji konwersji przez metaznaki pokaznae poniżej

Metaznak	Opis
\E	Zakończenie konwersji \L lub \U.
\l	Konwersja kolejnego znaku na małą literę
\L	Konwersja wszystkich znaków do \E na małę litery
\u	Konwersja kolejnego znaku na dużą literę
\U	Konwersja wszystkich znaków do \E na duże litery

\l i \u są umieszczane przed znakiem (lub wyrażeniem) aby konwertować kolejny znak. \L i \U konwertują wszystkie znaki do osiągnięcia \E. Poniższy prosty przykład konwertuje tekst wewnątrz pary znaczników <H1> do dużych liter:

TEKST

```
<BODY>

<H1>Welcome to my Homepage</H1>

Content is divided into two sections:<BR>

<H2>ColdFusion</H2>

Information about Macromedia ColdFusion.

<H2>Wireless</H2>

Information about Bluetooth, 802.11, and more.

<H2>This is not valid HTML</H3>

</BODY>
```

RegEx

```
(<[Hh]1>)(.*?)(</[Hh]1>)
```

ZASTĄPIENIE

```
$1\U$2\E$3
```

WYNIK

```
<BODY>

<H1>WELCOME TO MY HOMEPAGE</H1>

Content is divided into two sections:<BR>

<H2>ColdFusion</H2>

Information about Macromedia ColdFusion.

<H2>Wireless</H2>

Information about Bluetooth, 802.11, and more.

<H2>This is not valid HTML</H3>

</BODY>
```

ANALIZA

Wzorzec (<[Hh]1>)(.*?)(</[Hh]1>) dzieli nagłówek na trzy podwyrażenia: znacznik otwierający, tekst, i znacznik zamykający. Drugi wzorzec wstawia potem teskt razem: $1 zawiera znacznik startowy, \U$2\E konwertuje drugie podwyrażenie (tekst nagłówka) na duże litery a $3 zawiera znaczniki zamykający

Wprowadzenie do lookaround

Poniownie zaczniemy od przykładu. Musimy wyodrębnić tytuł strony WWW; tytuł strony HTML jest umieszczony między znacznikami <TITLE> a </TITLE> w sekcji <HEAD> kodu HTML.Oto przykład

TEKST

```
<HEAD>

<TITLE>Ben Forta's Homepage</TITLE>

</HEAD>
```

RegEx

```
<[tT][iI][tT][lL][eE]>.*</[tT][iI][tT][lL][eE]>
```

WYNIK

```
<HEAD>

<TITLE>Ben Forta's Homepage</TITLE>

</HEAD>
```

ANALIZA

```
<[tT][iI][tT][lL][eE]>.*</[tT][iI][tT][lL][eE]>
```

dopasowuje znacznik otwierający <TITLE> (dużymi, małymi lub mieszanymi literami), znacznik zamykający </TITLE> i to co jest między nimi. To działa.
Albo nie? To co było potrzebne to był tekst tytułu, ale uzyskaliśmy również znaczniki otwierający i zamykający <TITLE>. Czy jest możliwe aby powrócić tylk odo tytułu ? Jednym, z rozwiązań może być użycie podwyrażeń. Pozwoliłoby to na pobieranie dopasowanego tekstu w trzech częściach; znacznika otwierającego, tekstu i znacznika zamykającego. Z dopasowanym tekstem, podzielonym an części, nie byłoby trudno wyodrębnić już tylko część jaką chcemy. Ale nie ma sensu wkładać wysiłek dla uzyskania czegoś czego w rzeczywistości nie chcemy, tylko musimy to usunąć ręcznie. To co naprawdę musimy zrobić to skonstruować wzorzec tak aby zawierał dopasowania, które nie są zwracane – dopasowania, które są używane aby znajdować odpowiednie położenia dopasowania,ale nie są używane jako część podstawowego dopasowania. Innymi słowy, potrzeba się rozejrzeć.

Looking Ahead

Lookahead określa wzorzec jaki ma być dopasowany ale nie zwracany. Lookahead jest w rzeczywistości podwyrażeniem i formatowany jest tako taki. Składnia dla wzorca lookahead jest podwyrażeniem poprzedzonym przez ?=, a tekst do dopasowania znakiem =.
Oto przykład. Poniższy tekst zawiera listę URL'i i musimy wyodrębnić z każdego część protokołu.

TEKST

```
http://www.forta.com/

https://mail.forta.com/

ftp://ftp.forta.com/
```

RegEx

```
.+(?=:)
```

WYNIK

```
http://www.forta.com/

https://mail.forta.com/

ftp://ftp.forta.com/
```

ANALIZA

W wylistownych URL'ach , protokoły są oddzielone od nazwy hostów przez **:**. Wzorzec .+ dopasowuej dowolny tekst (http w pierwszym dopasowniu) , a podwyrażenie (?+:) dopasowuje **:** .Ale zauważ ,że : nie zostało doapsowane; ?= mówi silnikowi wyrażenia regularnego aby dopasował **:** ale do lookahead. Aby lepiej zrozumieć co robi ?= , tu mamy ten sam przykład, tym razem bez metaznaków lookahead:

TEKST

```
http://www.forta.com/

https://mail.forta.com/

ftp://ftp.forta.com/
```

RegEx

```
.+(:)
```

WYNIK

```
http://www.forta.com/

https://mail.forta.com/

ftp://ftp.forta.com/
```

ANALIZA

Podwyrażenie (:) poprawnie dopasowuje **:** , ale dopasowany tekst jest zwracany jako część dopasowania .Różnica między tymi dwoma przykładami jest taja ,że wzorzec używający (?=:) dopasowuje : a ostatni używa (:). Oba te wzorce dopasowują tą samą rzecz; oba dopasowuję : po protokole. Róznica jest w tym , czy dopasowane : było zawarte w dopasowanym tekście. Kiedy używamy lookahead, wyrażenie regularne parsuje dla przetworzenia dopasowania :, ale nie przetwarza go jako części podstawowego wyszukiwania .+(:) znajduje tekst włącznie z :. .+(?=:) znajduje teskt ale juz bez :.

Dopasowania lookahead (i lookbehind) w rzeczywistości zwraca wynik, ale wynikiem jest zawsze o znaków długości. Jako takie, odnosimy się czasami do operacji znajdowania lookaround jako zerowej szerokości. Dowolne podwyrażenie może być dołączone do wyrażenia lookahead przez proste poprzedzenie tekstu ?=. Można użyć wielu wyrażeń lookahead we wzorcu wyszukiwania,a one mogą pojawiać się gdziekolwiek we wzorcu (nie tylko na początku)

Looking Behind

Jak widziałeś, ?= patrzy na to co przychodzi po dopasowanym tekście, ale nie konsumuje tego co znajdzie. Zatem do ?= odnosimy się jako do operatora lookahea. Oprócz looking ahead, wiele implementacji wyrażeń regularnych obsługuje looking behind. Patrzy na to co jest przed tekstem będącym zwracanym obejmując looking behind, i operator looking behind ?<=.

?<= jest używane w ten sam sposób co ?=; jest używane wewnątrz podwyrażenia i występuje po tekście do dopasowania. Oto przykład. Baza danych zawiera listę produktów, a my potrzebujemy tylko cen

TEKST

```
ABC01: $23.45

HGG42: $5.31

CFMX1: $899.00

XTC99: $69.96

Total items found: 4
```

RegEx

```
\$[0-9.]+
```

WYNIK

```
ABC01: $23.45

HGG42: $5.31

CFMX1: $899.00

XTC99: $69.96

Total items found: 4
```

ANALIZA

\$ dopasowuje $, a [0-9.]+ dopasowuje cenę. To działa. Ale co jeśli nie chcemy znaku $ w dopasowanym tekście? Może po prostu usuniemy \$ ze wzorca?

TEKST

```
ABC01: $23.45

HGG42: $5.31

CFMX1: $899.00

XTC99: $69.96

Total items found: 4
```

RegEx

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

WYNIK

```
ABC01: $23.45

HGG42: $5.31

CFMX1: $899.00

XTC99: $69.96

Total items found: 4
```

ANALIZA

To oczywiście nie zadziała. Musi być \$ dla określenia jaki tekst dopasować, ale nie chcemy zwracanego $. Rozwiązanie? Dopasowanie lookbehind

TEKST

```
ABC01: $23.45

HGG42: $5.31

CFMX1: $899.00

XTC99: $69.96

Total items found: 4
```

RegEx

```
(?<=\$)[0-9.]+
```

WYNIKI

```
ABC01: $23.45

HGG42: $5.31

CFMX1: $899.00

XTC99: $69.96

Total items found: 4
```

ANALIZA

To jest sztuczka. (?<=\$) dopasowuje $ ale nie pobiera go, a tylko cenę (bez poprzedzającego znaku $) Porównajmy pierwsze i ostatnie wyrażenia użyte w tym przykładzie. \$[0-9.]+ dopasowuje $ następnie ilość dolarów.(?<=\$)[0-9.]+ również dopasowuje $ a następnie ilość dolarów ..Różnica między nimi nie jest w tym co znajduje się w trakcie wykonywania wyszukiwania, ale w tym co jest uzwględnione w wynikach. Pierwszy lokalizuje i dołącza $. Drugi lokalizuje $ aby można znaleźć cenę, ale nie zawiera $ w dopasowanych wynikach.
Wzorce lookahead mogą być zmiennej długości. Mogą zawierać . i + , an przykład, więc są jaby

dynamiczne. Wzorce lookbehind , z drugiej strony , muszą generalnie być stałej długości. Jest to ograniczenie wymagane przez prawie wszystkie implementacje wyrażeń regularnych.

Połączenie Lookahead i Lookbehind

Operacje lookahead i lookbehind mogą być połączone, jak w poniższym przykładzie (wyszukiwanie tytułu strony WWW).

TEKST

```
<HEAD>
<TITLE>Ben Forta's Homepage</TITLE>
</HEAD>
```

RegEx

```
(?<=<[tT][iI][tT][lL][eE]>).*(?=</[tT][iI][tT][lL][eE]>)
```

WYNIK

```
<HEAD>
<TITLE>Ben Forta's Homepage</TITLE>
</HEAD>
```

ANALIZA

To działa. `(?<=<[tT][iI][tT][lL][eE]>)` jest operacją lookbehind, która dopasowuje (ale nie pobiera) <TITLE>;

`(?=</[tT][iI][tT][lL][eE]>)` podobnie dopasowuje (ale nie pobiera) </TITLE>.

Wszystko co jest zwracane to tytuł strony.

Negowanie Lookaround

Jak widać, lookahead i lookbehind są zwykle używane do dopasowania tekstu, szczególnie dla określenia położenia tekstu będącego zwracanym (przez określenie tekstu przed i po żądanym dopasowaniem). Jest to znane jako pozytywne lookahead i pozytywne lookbehind. Termi pozytywny odnosi się do faktu ,ich spojrzenia na dopasowanie. Rzadziej używaną formą lookaround jest negatywny lookaround. Negatywny lookahead wykonuje look ahead dla tekstu ,który nie pasuje do określonego wzorca, a negatywny lookbehind podobnie wykonuje look behind dla tekstu ,który nie pasuje do określonego wzorca. Możesz oczekiwac zastosowania ^ dla zanegowania lokkaround, ale nie , składnia jest trochę inna. Operacje lookaround są negowane przy użyciu ! (które zastępuje =) . Poniższa lista pokazuje wszystkie operacje lookaround

Klasa	Opis
(?=)	Pozytywny lookahead
(?!)	Negatywny lookahead

(?<=) Pozytywny lookbehind
(?<!) Negatywny lookbehind

Aby zademonstrować różnicę miezy pozytywnym a negatywnym lookbehind, mamy przykład. Poniższy blok tekstu zawiera liczby, zarówno ceny jak i ilości. Najpierw uzyskajmy cenę

TEKST

```
I paid $30 for 100 apples,

50 oranges, and 60 pears.

I saved $5 on this order.
```

RegEx

```
(?<=\$)\d+
```

WYNIK

```
I paid $30 for 100 apples,

50 oranges, and 60 pears.

I saved $5 on this order.
```

ANALIZA

To jest nieco podobne do przykładu widzianego poprzednio. \d+ dopasowuje lcizby (jedną lub więcej cyfr), a (?<=\$) look behind dopasowanie (ale nie pobranie) $ (poprzedzonego \$) Dlatego , liczby w dwóch cenach zostały dopasowane, ale nie ilości. Teraz dla przeciwieństwa, lokalizujemy jednostki a nie ceny:

TEKST

```
I paid $30 for 100 apples,

50 oranges, and 60 pears.

I saved $5 on this order.
```

RegEx

```
\b(?<!\$)\d+\b
```

WYNIK

```
I paid $30 for 100 apples,

50 oranges, and 60 pears.

I saved $5 on this order.
```

ANALIZA

Ponownie, \d+ dopasowuje liczby, aletym razem tylko dopasowano ilości a nie ceny. Wyrażenie (?<!\$) jest negatywnym lookbehind,które dopasowuje tylko kiedy poprzednia liczba nie jest \$. Zmieniamy = w lookbehind zmieniając wzorzec z pozytywnego na negatywnego. Być może zastanawiasz się, dlaczego wzorzec w przykładzie negatywnym lookbehind definiuje granice słowa (przy użyciu \b). Aby zrozumieć ,dlaczego jest to konieczne tu mamy przykład bez tych granic:

TEKST

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

RegEx

```
(?<!\$)\d+
```

WYNIKI

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

ANALIZA

Bez granicy słó, 0 w $30 również dopasowane.Dlaczego? Ponieważ jest $ przed nim. Otoczenie całego wzorca wewnątrz granic słowa rozwiąże ten problem.

PODSUMOWANIE

Looking ahead i begind, dostarczają wspaniałej kontroli nad tyum co jst zwracane kiedy wykonywane jest dopasowanie. Operacje lookaround pozwalają podwyrażeniom być używanymi do określenia położenia tekstu jaki ma być dopasowany, ale nie pobierane (dopasowuje, anie obejmuje w samym tekście dopasowanym) Pozytywny lookahead jest definiowany przez użycie (?=), a negatywne lookahead jest definiowany przy użyciu (?!). Niektóre implementacje wyrażeń regularnych również obsługują lookbehind przy użyciu (?<=) a negatywny lookahead przy użyciu (?<!).

Dlaczego warunki osadzone?

(123)456-7890 i 123-456-7890 są akceptowanym formatami dla numerów telefonów w USA. 1234567890, (123)-456-7890 i (123-456-7890) wszystkie zawierają poprawne liczby cyfr, ale źle sformatowane. Jak można napisać wyrażenie regularne dla dopasowania tylko akceptowalnych postaci a nie wszystkich. To nei jest trywialny problem; rozważmy to oczywiste rozwiązanie:

TEKST

```
123-456-7890

(123)456-7890

(123)-456-7890

(123-456-7890

1234567890

123 456 7890
```

RegEx

```
\(?\d{3}\)?-?\d{3}-\d{4}
```

WYNIK

```
123-456-7890

(123)456-7890

(123)-456-7890

(123-456-7890

1234567890

123 456 7890
```

ANALIZA

\(? dopasowuje opcjonalnie otwierający nawias (zauważ ,że (musi być poprzedzony \), \d{3}dopasowuje pierwsze trzy cyfry, \)? dopasowuje opcjonalnie nawias zamykający -? dopasowuje opcjonalnie myślnik, a \d{3} - \d{4} dopasowuje pozostałe siedem cyfr (oddzielonych myślnikiem. Wzorzec nie dopasowuj poprawnie ostatnich dwóch linii, ale dopasowuje trzecią i czwartą – obie które są niepoprawne (trzecia zawiera zarówno) i - , a czwarta ma niedopasowany nawias). Zastąpeienie \)?-? z [\)=]? pomoże wyeliminować trzecią linię (tylko z) lub - ,ale nie oba) ale czwarta linia ma problem. Wzorzec musi dopasować) tylko jeśli jest otwierający (.

Używanie warunków

Warunki wyrażenia regularnego są definiowane przez ?. Faktycznie, widziałeś parę określonych warunków:

- ? dopaspowuje porzedni znak lub wyrażenie jeśli istnieje
- ?= i ?<= dopasowują tekst przed lub po , jeśli istnieje

Składnia warunków osadzonych również używa ? , co nie jest niespodzianką że warunki które są osadzone są jako:

- Przetwarzanie warunkowe oparte o referencję wsteczną
- Przetwarzanie warunkowe oparte o lookaround

Warunki referencji wstecznej
Warunek referncji wstecznej pozwala na używanie wyrażenia tylko jeśli poprzednie podwyrażenie wyszukało z powodzeniem. Rozważmy przykład: Musisz zlokalizować wszystkie znaczniki <IMG> w tekście; dodatkowo, jeśli dowolny znaczniki <IMG> są łączami (otoczone przez znaczniki <A> i </A>), musisz również dopasować cały link. Składnia dla tego typu warunku to (?(bacreference) true). ? zaczyna warunek, referencja wsteczna jest określona wewnątrz nawiasów, a wyrażenie będzie wyliczone tylko jeśli referencja wsteczna jest obcena zaraz po.

TEKST

```
<!-- Nav bar -->
<TD>
<A HREF="/home"><IMG SRC="/images/home.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/search"><IMG SRC="/images/search.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/help"><IMG SRC="/images/help.gif"></A>
</TD>
```

RegEx

```
(<[Aa]\s+[^>]+>\s*)?<[Ii][Mm][Gg]\s+[^>]+>(?(1)\s*</[Aa]>)
```

WYNIK

```
<!-- Nav bar -->
<TD>
<A HREF="/home"><IMG SRC="/images/home.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/search"><IMG SRC="/images/search.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/help"><IMG SRC="/images/help.gif"></A>
</TD>
```

ANALIZA

Ten wzorzec wymaga wyjaśnienia. `(<[Aa]\s+[^>]+>\s*)?` Dopasowuje otwierający znacznik <A> lub <a> (z dowolnym atrybutem jaki może być obecny), jeśli jest obecny (zmaykający ? Tworzy wyrażenie opcjonalne). < [Ii] [Mm] [Gg]\s+[^>]+> wtedy dopasowuje znacznik <IMG> (bez wzgledu na wielkość liter) z dowolnymi atrybutami.

`(?(1)\s*</[Aa]>)` zaczyna od warunku: ?(1) oznacza wykonanie tylko tego co przychodzi jeśli istnieje referencja wsteczna 1 (znacznik otwierający <A>)(lub innymi słwoy, wykonuje się tylko jeśli pierwsza <A> zostało dopasowane). Jeśli (1) istnieje, wtedy \s*</[Aa] dopasowuje końcowe białe znaki po których jest znacznik zmaykający </A>.

?(1) sprawdza czy istnieje referencja wsteczna 1. Liczba referencji wstecznej (1 w tym przykładzie) nie musi być dodana w warunkach. Więc ?(1) jest poprawne, a ?(\1) nie.

Wzorzec używa wykonania wyrażenia jeśli napotka warunek. Warunki mogą mieć również wyrażenie else, wyrażenie, które jest wykonywane tylko jeśli referencja wsteczna nie istnieje (nie napotkano warunku) Składnia dla tego typu warunku to (?(backreference) true|false). Składnia akceptuje warunek, jak również wyrażenie będzie wykonane jeśli warunek zostanie napotkany lub nie. Składnia ta dostarcza rozwiązania z problemem numerów telefonicznych

TEKST

```
123-456-7890

(123)456-7890

(123)-456-7890

(123-456-7890

1234567890

123 456 7890
```

RegEx

```
(\()?\d{3}(?(1)\)|-)\d{3}-\d{4}
```

WYNIK

```
123-456-7890

(123)456-7890

(123)-456-7890

(123-456-7890

1234567890

123 456 7890
```

ANALIZA

Wzorzec wydaje się działać, ale dlaczego? Jak poprzednio, (\( ) ? sprawdza początek pary nawiasów, ale tym razem wyniki są otoczone wewnątrz nawiasów tak aby można było stworzyć podwyrażenie. \d{3} dopasowuje trzy cyfry. (?(1)\) | -) dopasowuje albo) albo – w zależności od tego czy warunek jest spełniony. Jeśli istnieje (1) (oznaczające ,że został znaleziony otwierający nawias), wtedy musi być dopasowane \); w przwciwnym razie, - musi być dopasowane. W ten sposób, nawiasy muszą zawsze być w parze, a myślnik oddzielający kod obszaru od numeru jest dopasowany tylkojęś nei są używane nawiasy.

Warunki lookaround

Warunek lookaround pozwala wyrażeniom na bycie wykonywanymi w oparciu o to czy operacja lookahead czy lookbehind zakończyła się powodzeniem. Składnia dla warunków lookaround jest taka sama jak dla warunków referencji wstecznej, z wyjątkiem tego ,że referencja wsteczna (liczba wewnątrz nawiasów) jest zastępowana przez całkowite wyrażenie lookaround.
Jako przykład rozważmy kody pocztowe z USA. Może to być pięciocyfrowy kod ZIP w foramcie 12345 lub kod ZIP+4 w postaci 12345-6789. Myślnik jest używany tylko jeśli są obecne dodatkowe cztery cyfry. Oto rozwiązanie:

TEKST

```
11111
22222
33333-
44444-4444
```

RegEx

```
\d{5}(-\d{4})?
```

WYNIK

```
11111
22222
33333-
44444-4444
```

ANALIZA

\d{5} dopasowuje pierwsze pięć cyfr a (-\d{4})? Dopasowuje myślnik a po nim cztery cyfry jeśli istnieją. Ale co jeśli nie chcesz dopasować źle sformatowanych kodów? Trzecia linia w tym przykładze ma myślnik końcowy, którego tam nie powinno prawdopodobnie być. Poprzedni wzorzec dopasował cyfry bez myślnika, ale jak można nie dopasować tego całego kodu ZIP

ponieważ jest źle sformatowany? Ten przykład może wydawać się trochę zmyślony ,ale po prostu demonstruje użycie warunku lookaround

TEKST

```
11111

22222

33333-

44444-4444
```

RegEx

```
\d{5}(?(?=-)-\d{4})
```

WYNIKI

```
11111

22222

33333-

44444-4444
```

ANALIZA

Ponownie, \d{5} dopasowuje otwierające pięć cyfr. Potem przychodzi warunek w postaci (?(=-)-\d{4}). Warunek używa ?=- dla dopasowania (ale nie pobrania) myślnika ,a jeśli napotkano warunek (istnieje myślnik), wtedy -\d{4} dopasowuje ten myślnik po którym następują cztery czyfr. W ten sposób 33333- nie zostanie dopasowane(mam myślnik i napotkano warunek ,ale nie ma dodatkowych czterech cyfr). Lookahead u lookbehind (pozytywne lub negatywne) moga być używane jako warunek ,i opcjonalne wyrażenie else, może zostać użyte również (używamy tej samej składni co poprzednio , | wyrażenie)

PODSUMOWANIE

Warunki mogą być osadzane we wzorcach wyrażeń regularnych aby można było wykonywać wyrażenia tylko jeśli napotkano (lub nie) warunek. Warunek może być referencją wsteczną (warunek jest wtedy sprawdzany na istnienie) lub operacją lookaround.

WYRAŻENIA REGULARNE W POPULARNYCH APLIKACJACH I JĘZYKACH

grep

grep jest narzędziem Unixa dla wykonywania wyszukiwań tekstu w plikach i standardowym tekście wejściowym. grep obsługuje podstawowe, rozszerzone i wyrażenia regularne Perla, w zleżności od określonej opcji:

- -E używa rozszerzonych wyrażeń regularnych
- -G używa podstawowych wyrażeń regularnych
- -P używa wyrażeń regularnych Perla

Uwagi:

- Domyślnie , grep wyświetla całą linię dla linii zawierającej dopasowanie; dla pokazania tylko dopasowania, użyj opcji -o
- Użyj opcji -v dla zanegowania dopasowania i wyświetlenia tylko niedoapsowanej linii
- Użyj opcji -c dla wyświetlenia licznika (liczby dopasowań0 zamiast wszystkich szczegółów dopasowania
- Użyj opcji -i dla dopasowania tego co nie jest czułe na wielkość liter
- grep jest używane do operacji wyszukiwania , a nie dla operacji zastępowania. Jako taka , funkcjonalność zastępowania nie jest wspierana

JavaScript
JavaScript implementuje wyrażenia regularne przetwarzane w obiektach String i RegEx przez następujące metody:

- exec jest metodą RegEx używaną dla wyszukiwania dopasowania
- match jest metodą String używaną do dopasowania łańcucha
- replace jest metodą String używaną do wykonania operacji zastąpienia
- search jest metodą String używaną do testowanai dla dopasowania w łańcuchu
- split jest metodą String używaną do podzielenia łańcucha na wiele łańcuchów
- test jest metodą RegEx używaną dla testowania dopasowania w łańcuchu

Wyrażenia regularne JavaScript są modelowane na te w Perla, ale uważaj na następujace rzeczy:

- JavaScript używa flag do zarządzania globalnego wyszukiwania ze względu na wielkość liter: g włącza globalne, i tworzy dopasowania nie czułe na wielkość liter, a te dwie flagi mogą być połączone jak gi.
- Dodatkowe modyfikatory to m dla obsługi wieloliniowych łańcuchów, s dla jednoliniowego łańcucha, a x dla ignorowania białych znków wewnątrz wzorca regex
- Kiedy używamy referencji wsteczne, $' zwraca wszystko przed dopasowanym łańcuchem, $' zwraca wszystko po dopasowanym łańcuchu, $+ zwraca ostatnie dopasowane podwyrażenie, a $& zwaraca wszystko dopasowane.
- Funkcja JavaScript o globalnej nazwie RegExp, która może być dostępna dla uzyskania informacji o wykonaniu po wyrażeniu regularnym zostaje wykonana
- Klasy znaków POSIX nie są obsługiwane
- \A i \Z nie są wspierane

Macromedia ColdFusion
ColdFusion dostarcza obsługi wyrażeń regularnych przez cztery funkcje:
- REFind () dla wykonania wyszukiwania
- REFindNoCase () dla wykonania wyszukiwania ,które nie są czułe na wielkość liter
- RERplace () dla wykonania zastępowania
- REPlaceNoCase () dla wykonania zastąpienia, które nie jest czułe na wielkość liter

ColdFusion również obsługuje wyrażenia regularne dla walidacji danych wejściowych w znaczniku <CFINPUT>. Jednak , znacznik ten w rzeczywistości nie przetwarza wyrażeń regularnych; raczej

po prostu przekazuje je do generowane JavaScript po stronie klienta do przetworzenia. Jako takie, wyrażenie regularne używane z <CFINPUT> są regulowane przez zasady i informacje, która mają zastosowanie do wyrażeń regularnych w JavaScript.

ColdFusion obsługuje wyrażenia regularne, które są zgodne z Perl, z kilkoma wyjątkami:

- **.** zawsze dopasowuje nową linię
- Kiedy używasz referencji wstecznych, używaj \n zamiast $n dla zmiennej referencji wstecznej. ColdFusion poprzedza wszystkie $ w zastępowanym łańcuchu
- Nie musisz poprzedzać backslahe w zastępowanych łańcuchach. ColdFusion poprzedza je, z wyjątkiem sekwencji konwersji wielkości liter lub wersji poprzedzonej (na przykład, \u lub \\u)
- Osadzone modyfikatory ((?i) itd) zawsze wpływają na całe wyrażenie, nawet jeśli są wewnątrz grupy
- \Q i połączenia \u\L i \l\U nie są obsługiwane w zastępowanych łańcuchach
- Lookbehind (?<= i ?<!) nie jest obsługiwane
- Przetwarzanie warunkowe nie jest obsługiwane
- \x,\N,\p i \C nie są obsługiwane

Macromedia Dreamweaver
Macromedia Dreamweaver dostarcza obsługi wyrażeń regularnych dla operacji wyszukiwania i zastępowania. Aby skorzystać z wyrażeń regularnych zrób co następuje:

- Wybierz Find and Replace z menu Edit i zaznacza pole Use Regular Expression

Zauważ co następuje:

- Składnia $ (jak $1) jest używana do odniesienia się do referencji wstecznych w zastępowanym wzorcu , ale składnia \ (jak w \1) jest używana do odniesienia do referencji wstecznej w tym samym wzorcu
- Wzorce wyrażeń regularnych mogą być zapisane i użyte ponownie jeśli jest taka potrzeba

Macromedia HomeSite (i ColdFusion Studio)
Macromedia HomeSite(w tym ColdFusion Studio) dostarcza wyrażenia regularnego dla operacji wyszukiwania i zastępowania. Aby skorzystać z wyrażeń regularnych rób co następuje

- Wybioerz Extended Fidn lub Extended Replace z menu Search
- Zaznacz pole wyboru Regular Expressions

Zauważ ,że:

- Obsługiwane wyrażenia regularne HomeSite są modelowane na wyrażeniach regularnych ColdFusion
- Obsługiwane są klasy POSIX
- Obsługiwane sa referencje wstweczne i używają składni \1.
- **.** zawsze dopasowuje nową linię
- Wzorce wyrażeń regularnych mogą być zapisane i użyte ponownie jeśli jest taka potrzeba

Microsoft ASP
Wszystkie języki skryptowe ASP obsługują wyrażenia regularne. Wyrażenie regularne jest dostarczane poprzez obiekt nazwany RegExp, kóry zawiera następujące metody;

- Execute () wykonujące wyszukiwanie wyrażenia regularnego
- Replace () wykonuje operacje wyszukiwania i zastępowania
- Test () sprawdza czy dopasowano łańcuch określonym wyrażeniem regularnym

Obsługa wyrażeń regularnych ASP jest nieco ograniczona:

- Instancja obiektu RegExp musi być stworzona i wypełniona zanim poprzednia metoda zostanie wykonana
- Wyrażenie regularne jest przechowywane w RegEp.Pattern
- Globalne i czułe na wielkość modyfikatory są obsługiwane. Wartości boolowskie przechowywane w RegExp.Global i są one wielkością boolowską przechowywaną w RegExp.IgnoreCase
- Execute () zwraca obiekt Match, który dostarcza dostępu do wszystkich dopasowań
- Lookahead (?= i ?!) i lookbehind (?<= i ?<!) nie są obsługiwane

Microsoft .NET
.NET Fram,ewaork dostarcza mocnego i elastycznego przetwarzania wyrażeń regularnych jako części biblioteki klasy bazowej. Jako takie, są dostępne dla użycia przez języka .NET i narzędzia (w tym ASP.NET, C# i Visual Studio .NET). Wyrażenia regularne wspierane w .NET są dostarczane w klasie Regex (jak również dodatkowo obsługiwane klasy). Regex obsługuje poniższe metody;

- IsMatch () sprawdza czy znaleziono dopasowanie w określonym łańcuchu
- Match () wyszukuje pojedyncze dopasowanie, które jest zwracane jako obiekt Match
- Matches () wyszukuje wyszukuje wszystkie dopasowania, które są zwracane jako obiekt MatchCollection
- Replace () wykonuje operację zastępowania na określonym łańcuchu
- Split () dzieli łańcuch na tablicę łańcuchów

Jest również możliwe, przez funkcje wrapper, wykonanie wyrażenia regularnego bez konieczności instancji i pracy z klasą Regex:

- Regex.IsMatch () jest funkcjonalnym odpowiednikiem metody IsMatch () opisanej powyżej
- Regex.Match () jest funkcjonalnym odpowiednikiem metody Match ()
- Regex.Matches () jest funkcjonalnym odpowiednikiem metody IsMatch ().
- Regex.Replace jest funkcjonalnym odpowiednikiem metody Replace ().
- Regex.Split () jest funkcjonalnym odpowiednikiem metody Split ().

Oto kilka ważnych punktów dotyczących obssługi wyrażeń regularnych w .NET:

- Aby użyć wyrażeń regularnych, obiekty wyrażenia regularnego muszą być zaimportowane przy użyciu Imports System.Text.RegularExpression.
- Dla szybiego przetwarzania inline wyrażeń regularnych, funkcje wrapper są idealne
- Opcje wyrażenia regularnego są określone za pomocą właściwości Regex.Options – enumeracja Regex.Option, która może być używana do ustawienia składowych takich ja IgnoreCase, Multiline, Singleline i wiecej
- .NET wspeira przechwytywanie nazw, możliwość nazwania podwyrażenia (ta aby można było się odnosić do niego przez nazwę a nie liczbę). Składnia dla tego to ?<nazwa> , nazwa podwyrażenia, \k<nazwa> do odniesienia wstecznego, a $(nazwa) do odniesienia sięgo wzorca zastępowania.
- Iedy używamy odniesienia wsteczne, $` zwraca wszystko przed dopasowanym łańcuchem, $' zwraca wszystko po dopasowanym łańcuchu, $+ zwraca ostatnie dopasowane

podwyrażenie, $_ zwraca cały oryginalny łańcuch, a $& zwraca cały dopasowany łańcuch
- Konwersja wielkości liter \E,\l,\L,\u i \Unie jest obsługiwana
- Klasy znaków POSIX nie są obsługiwane

Microsoft Visual Studio .NET
Wyrażenia regularne obsługiwane w Visual Studio .NET są dostarczane przez .NET Framework. Aby użyć wyrażenia rgularnego zrób co następuje:

- Wybierz Find and Replace z menu Edit
- Wybierz Find, Replace, Find in Files lub Replace in Files
- Zaznacz Use check box, i wybierz Regular expressions z listy rozwijanej

Zauważ co następuje:

- Używamy @ zamiast *? .
- Używamy # zamiast +? .
- Używamy ^n zamiast {n}.
- W operacji zastępowania, odniesienia wsteczne mogą być wzmocnione tak ,aby były wyrównane do lewej strony przez użycie \(w,n) (gdzie w jest szerokością a n jest odniesieniem wstecznym). Aby wyrównać do prawa , używamy \ (-w , n).
- Visual Studio .NET używa specjalnych metaznaków i symboli : :a dla [a-zA-Z0-9], :c dla [a-zA-Z], :d dla \d, :h dla [a-fA-F] (znaki szesnastkowe), :i dla poprawnych identyfikatorów [a-zA-Z_$][a-zA-Z_0-9$]*, :q dla cudzysłowu, :w dla [a-zA-Z]+, :z dla \d+.
- \n jest znakiem przełamania linii niezależnym od platformy i wsatwia nową linię kiedy jest stosowana operacja zastąpienia
- Poniższe specjalne litery dopasowują znaki jakie są wspierane :Lu dopasowuje duże znaki, :Ll dopasowuje małe znaki, :Lt dopasowuje tytuł (pierwsza litera duża), :Lm dla znaków interpunkcyjnych
- :Nd dla [0-9]+, :Nl dla cyfr rzymskich
- Następujące specjalne znaki interpunkcyjne dopasowują obsługiwane znaki: :Ps dla otwierających znaków interpunkcyjnych, :Pe dla zamykających znaków interpunkcyjnych, :Pi dla podwójnych znaków cudzysłowów, :Pf dla pojedynczego cudzysłowu, :Pd dla kreski (myślnika), :Pc dla znaku podkreślenie, :Po dla innych znaków interpunkcyjnych
- Poniższe specjalne symbole dopasowują obsługiwane znaki : :Sm dla matematycznych symboli, :Sc dla symboli walut, :Sk dla modyfikatorów akcentu, :So dla innych specjalnych symboli
- Inne specjalne znaki są również obsługiwane

MySQL
MySQL jest popularną bazą danych open source. Wyrażenia regularne obsługiwane w MySQL są dostępne w klauzuli WHERE w poniższym formacie:

REGEXP "wyrażenie"

Pełna instrukcja MySQL używająca wyrażenia regularnego będzie używać składni takiej jak ta: SELECT * FROM table WHERE REGEXP "wzorzec" .

Obsługiwane wyrażenia regularne MySQL są przydatne i mocne, ale nie jest to pełna imlementacja wyrażenia regularnego:

- Dostarczona jest tylko obsługa wyszukiwanial nie ma obsługi zastępowania

- Wyszukiwanie nie jest czułe na wielkość liter. Aby wykonać wyszukiwanie z uwzględnieniem wielkości liter , użyj słowa kluczowego BINARY (między REGEXP a samym wyrażeniem)
- Używamy [[:<:]] dla dopasowania początkowegosłowa a [[:>:]] dla dopasowania słowa końcowego
- Lookaround nie jest obsługiwane
- Warunki osadzone nie są obsługiwane
- Wyszukiwanie znaków ósemkowych nie jest obsługiwane
- \a,\b\e,\f i \v nie jest obsługiwane
- Nie są obsługiwane odniesienia wsteczne

Perl
Perl jest najstarszą implementacją wyrażeń regularnych, a większość implementacji próbuje być kompatybilnych z Perl. Wyrażenia regularne sa jądrem części Perla i jest używane po prostu przez określenie operacji i wzorca:

- m/wzorzec/ dopasowuej określony wzorzec
- s/wzorzec/wzorzec wykonuje operację zastąpienia
- qr/wzorzec zwraca obiekt Regex , który może być potem użyty później
- split() dzieli łańcuch nad podłańcuchy
- Modyfikatory mogą być przekazywane po wzorcu. Używamy /i dla wyszukiwania które nie zależy od wielkości liter i /g dla globalnego (dopasowanie wszystkich dopasowań)

PHP
PHP dostarcza obsługi wyrażeń regularnych kompatybilnych z Perl przez pakiet PCRE. Poniższe funkcje wyrażenia regularnego są obsługiwane prze PCRE;

- preg_grep () wykonuej wyszukiwanie i zwracanie tablicy dopasowań
- preg_match () wykonuje wyszukiwanie wyrażenia regularnego dla pierwszego dopasowania
- preg_match_all () wykonuje wyszukiwanie globalen wyrażenia regularnego
- preg_quote () pobiera wzorzec i zwraca wersję z backslahem przed nim
- preg_replace () wykonuje operację wyszukiwania i zastępowania
- preg_replace_callback () wykonuje operację wyszukiwania i zastępowania, ale używa funkcji wywołania zwrotnego dla wykonania rzeczywistego zastąpienia
- preg_spit () dzieli łańcuch na podłańcuchy

Zuważ co następuje:

- Dla dopasowania, które nie jest czułe na wielkość liter , używamy modyfikatora i.
- Łańcuchy wieloliniowe mogą być włączane modyfikatorem m.
- PHP może wyliczać łańcuchy zastąpienia jako kod PHP. Aby włączyć tą funkcjonalność, użyj modyfikatora e.
- preg_replace () ,preg_replace_callback () i preg_spit () wszystkie wspierają parametr opcjonalny, który określa limit – maksymalną liczbę zastąpień lub dzieleń wykonywanych
- Odniesienia wsteczne moga być stsosowane przy użyciu składni Perla $ ($1 ,na przykład)
- Nie są obsługiwane \l,\u,\L,\U,\E,\Q i \v

Sun Java
Dopasowania wyrażeń regularnych jest wykonywane przez klasę java.util.rgex.matcher i te metody:

- find() znajduje wystąpienie wzorca wewnątrz łańcucha
- lookingAt () próbuje dopasować początek łańcucha sepcjalnym wzorcem
- matches () próbuje dopasować cały łańcuch specjalnym wzorcem
- replaceAll () wykonuje operację zastąpienia i zastąpienie wszystkich dopasowań
- replaceFirst () wykonuej operację zastąpienia i zastęouje pierwsze dopasowania

Dodatkowe metody klasy dostarczają większej kontroli nad określonymi operacjami.

- compile () kompiluje wyrażenie regularne we wzocu
- flags () zwraca flagi dopasowania wzorca
- matches () jest funkcjonalnym odpowiednikiem metody matches () opisanej wcześniej
- pattern () zwraca wyrażenie regularne z którego wzorzec został stworzony
- split () dzieli łańcuch na podłańcuchy

Obsługa wyrażeń regularnych Sun jest oparta na implementacji Perl , ale należy pamiętać o:

- Aby użyć wyrażenia regularnego, pakiet wyrażenia regulrnego musi byż zaimportowany przy użyciu import java.util.regex.*
- Warunki osadzone nie są obsługiwane
- Nie są obsługiwane konwersje wielkości liter \E ,\l, \L , \u i \U.
- Nie jest obsługiwane dopasowanie z \b.
- Nie jest obsługiwane \z.